Inhalt

1. Einleitung
2. Vorzeichenbehaftete und vorzeichenlose Zahlen
3. Grundlagen der ganzzahligen Arithmetik –
   3.1 Addition und Subtraktion
   3.2 Multiplikation und Division
   3.3 Beschleunigen der ganzzahligen Multiplikation - Booth’s Algorithmus
4. Entwurf einer ALU
   4.1 Beschleunigen der ganzzahligen Addition - Carry-Lookahead Addierer
   4.2 Beschleunigen der ganzzahligen Addition - Carry-Select, Carry-Skip, Carry-Save Addierer
5. Integer Multiplizierer
6. Integer Dividierer
7. Fließkomma Arithmetik
8. Fließkomma Addition
9. Fließkomma Multiplikation
10. Fließkomma Arithmetik – Übungsaufgaben
11. Beschleunigung der ganzzahligen Division und Multiplikation


Literatur


machines, and parallel processing.

suitable for graduate level courses in computer science (there are chapters on scientific applications, vector
book for computational scientists; in addition to detailed explanations of pipelining and memory organization
collection of original chapters and primary source material on historic architectures and networks, including IBM
A
[6] Patterson, D.A., Reduced Instruction Set Computers, Comm. ACM 28(1), January 1985, pp. 8-20. [Discussion of

Literatur
Hardware Algorithmen

Einführung
Computer Arithmetik

Hardware Algorithmen:
- Die Implementing einer ALU Architektur

Quellen:
Malek, M.: Vorlesung "Rechnerarchitektur", Sommersemester 2000, HU Berlin,  
[www.informatik.hu-berlin.de/~rok/ca](http://www.informatik.hu-berlin.de/~rok/ca)
Dave Patterson Computer Architecture and Engineering, September 12, 1997, © U.C.B.  
[www.cs.berkeley.edu/~patterson](http://www.cs.berkeley.edu/~patterson)
Zahlen

- Bits sind einfach Bits (keine inherente Deutung)
  — Konventionen definieren die Beziehung zwischen Bits und Zahlen

- Binäre Zahlen (Basis 2)
  0000 0001 0010 0011 0100 0101 0110 0111 1000 1001...
  dezimal: 0...2^n-1

- Einige Eigenschaften von Zahlen:
  Zahlen haben einen endlichen Definitionsbereich (Overflow)
  Brüche und reelle Zahlen
  Negative Zahlen

- Wie können negative Zahlen dargestellt werden?
  u.a., welches Bit will welche Zahlen darstellen?
Mögliche Darstellungen

<table>
<thead>
<tr>
<th></th>
<th>1-er Komplement</th>
<th>2-er Komplement</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 = +0</td>
<td>000 = +0</td>
<td>000 = +0</td>
</tr>
<tr>
<td>001 = +1</td>
<td>001 = +1</td>
<td>001 = +1</td>
</tr>
<tr>
<td>010 = +2</td>
<td>010 = +2</td>
<td>010 = +2</td>
</tr>
<tr>
<td>011 = +3</td>
<td>011 = +3</td>
<td>011 = +3</td>
</tr>
<tr>
<td>100 = -0</td>
<td>100 = -3</td>
<td>100 = -4</td>
</tr>
<tr>
<td>101 = -1</td>
<td>101 = -2</td>
<td>101 = -3</td>
</tr>
<tr>
<td>110 = -2</td>
<td>110 = -1</td>
<td>110 = -2</td>
</tr>
<tr>
<td>111 = -3</td>
<td>111 = -0</td>
<td>111 = -1</td>
</tr>
</tbody>
</table>
MIPS

- 32 Bit vorzeichenbehaltete Zahlen:

\[
\begin{align*}
0000 & 0000 0000 0000 0000 0000 0000 0000_{tm} = 0_{ten} \\
0000 & 0000 0000 0000 0000 0000 0000 0001_{tm} = + 1_{ten} \\
0000 & 0000 0000 0000 0000 0000 0000 0010_{tm} = + 2_{ten} \\
\ldots
0111 & 1111 1111 1111 1111 1111 1110_{tm} = + 2,147,483,646_{ten} \\
0111 & 1111 1111 1111 1111 1111 1111_{tm} = + 2,147,483,647_{ten} \\
1000 & 0000 0000 0000 0000 0000 0000 0000_{tm} = - 2,147,483,648_{ten} \\
1000 & 0000 0000 0000 0000 0000 0000 0001_{tm} = - 2,147,483,647_{ten} \\
1000 & 0000 0000 0000 0000 0000 0000 0010_{tm} = - 2,147,483,646_{ten} \\
\ldots
1111 & 1111 1111 1111 1111 1111 1110_{tm} = - 3_{ten} \\
1111 & 1111 1111 1111 1111 1111 1111_{tm} = - 2_{ten} \\
1111 & 1111 1111 1111 1111 1111 1111_{tm} = - 1_{ten}
\end{align*}
\]

Quellen:
2-er Komplement Operationen

- Negieren einer 2-er Komplement Zahl: invertiere alle Bits und addiere 1
  - merke: "negate" und "invert" unterscheiden sich sehr!
- Konvertieren einer n Bit Zahl in Zahlen mit mehr als n Bits:
  - kopieren des "Most Significant Bit" (Vorzeichenbit) in den andern Bits
    
    \[
    \begin{align*}
    0010 & \rightarrow 0000\ 0010 \\
    1010 & \rightarrow 1111\ 1010 \\
    \end{align*}
    \]
  - "Vorzeichen Erweiterung"
Addition & Subtraktion

- **Carry und Borrow 1s**
  \[
  \begin{array}{ccc}
  0111 & + & 0110 \\
  0111 & - & 0100 \\
  0110 & - & 0101 \\
  \end{array}
  \]

- **2-er Komplement Operationen**
  - **Subtraktion durch Addition von negativen Zahlen**
    \[
    \begin{array}{c}
    0111 \\
    + 1010 \\
    \end{array}
    \]

- **Overflow (Ergebnis überschreitet die Wortgrenze):**
  - das Addieren von zwei n-bit Zahlen muß nicht eine n-bit Zahl ergeben
    \[
    \begin{array}{c}
    0111 \\
    + 0001 \\
    \quad 1000 \\
    \end{array}
    \]

**Quellen:**
Overflow erkennen

• Kein Overflow, wenn positive und negative Zahlen addiert werden
• Kein Overflow, wenn in einer Subtraktion die Vorzeichen gleich sind.
• Overflow entsteht, wenn das Ergebnis das Vorzeichenbit ändert:
  – wenn die Addition von zwei positiven Zahlen eine negative Zahl ergibt
  – oder wenn die Addition von zwei negativen Zahlen eine positive Zahl ergibt
  – oder wenn die Subtraktion einer negativen Zahl von einer positiven Zahl eine negative Zahl ergibt
  – oder wenn die Subtraktion einer positiven Zahl von einer negativen Zahl eine positive Zahl ergibt

Betrachten Sie die Operationen A + B, und A – B
Kann Overflow entstehen wenn B = 0 ?
Kann Overflow entstehen wenn A = 0 ?

Quellen:
Malek, M.: Vorlesung "Rechnerarchitektur", Sommersemester 2000, HU Berlin,
www.informatik.hu-berlin.de/~rok/ca
Dave Patterson Computer Architecture and Engineering, September 12, 1997, © U.C.B.
www.cs.berkeley.edu/~patterson
Effects of Overflow

- An exception (interrupt) occurs
  - Control jumps to predefined address for exception
  - Interrupted address is saved for possible resumption
ALGORITHMEN II.

2. Vorzeichenbehaftete und vorzeichenlose Zahlen

### ZAHLENDARSTELLUNG (1)

<table>
<thead>
<tr>
<th>Bit-Muster</th>
<th>Zeichen und Größe</th>
<th>Einer-Komplement</th>
<th>Zweier-Komplement</th>
</tr>
</thead>
<tbody>
<tr>
<td>0111</td>
<td>+7</td>
<td>+7</td>
<td>+7</td>
</tr>
<tr>
<td>0110</td>
<td>+6</td>
<td>+6</td>
<td>+6</td>
</tr>
<tr>
<td>0101</td>
<td>+5</td>
<td>+5</td>
<td>+5</td>
</tr>
<tr>
<td>0100</td>
<td>+4</td>
<td>+4</td>
<td>+4</td>
</tr>
<tr>
<td>0011</td>
<td>+3</td>
<td>+3</td>
<td>+3</td>
</tr>
<tr>
<td>0010</td>
<td>+2</td>
<td>+2</td>
<td>+2</td>
</tr>
<tr>
<td>0001</td>
<td>+1</td>
<td>+1</td>
<td>+1</td>
</tr>
<tr>
<td>0000</td>
<td>+0</td>
<td>+0</td>
<td>+0</td>
</tr>
<tr>
<td>1000</td>
<td>-7</td>
<td>-7</td>
<td>-7</td>
</tr>
<tr>
<td>1001</td>
<td>-6</td>
<td>-6</td>
<td>-6</td>
</tr>
<tr>
<td>1010</td>
<td>-5</td>
<td>-5</td>
<td>-5</td>
</tr>
<tr>
<td>1011</td>
<td>-4</td>
<td>-4</td>
<td>-4</td>
</tr>
<tr>
<td>1100</td>
<td>-3</td>
<td>-3</td>
<td>-3</td>
</tr>
<tr>
<td>1101</td>
<td>-2</td>
<td>-2</td>
<td>-2</td>
</tr>
<tr>
<td>1110</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>1111</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

(a) Kreisdarstellung der ganzen Zahlen Modulo N

(b) Modulo 16 System für 2'er-Komplement Zahlen

ZAHLENDARSTELLUNG (2) BCD

   - 1010 steht für "+" und 1011 für "-"

   Beispiel:

   4739+1281=6020

   in BCD-Code:
   
   \[
   \begin{array}{ccccccc}
   0100 & 0111 & 0011 & 1001 \\
   + & 0001 & 0010 & 1000 & 0001 \\
   \hline
   \end{array}
   \]
   
   *(0101 1001 1011 1010)

   Die Werte müssen angepasst werden indem man durchlaufend 0110 addiert:

   \[
   \begin{array}{ccccccc}
   0101 & 1001 & 1011 & 1010 \\
   1 & 0110 \\
   \hline
   0101 & 1001 & 1100 & 0000 \\
   1 & 0110 \\
   \hline
   0101 & 1010 & 0010 & 0000 \\
   1 & 0110 \\
   \hline
   0110 & 0000 & 0010 & 0000 \\
   \hline
   6 & 0 & 2 & 0
   \end{array}
   \]

   \[D\rightarrow A\rightarrow X\rightarrow ALU\rightarrow T\]

ZAHLENDARSTELLUNG (3)
BINÄRE REPRÄSENTATION

1. Position und Wert
   - \( B=b_{n-1} \ldots b_1 b_0 \)
   - \( V(B)= b_{n-1} 2^{n-1} + \ldots + b_1 2^1 + b_0 2^0 \)

2. Vorzeichenbehaftete Zahlen
   \( b_{n-1}=0 \) positive \quad \( b_{n-1}=1 \) negative
   \( n \) - Anzahl von Bit \quad \( N \) - der effektive Wert
   - Vorzeichen und Größe
     \[
     \begin{array}{c|c}
     S & \text{Größe} \\
     \hline
     b_{n-1} & b_0 \\
     \end{array}
     \]
   - 1'er-Komplement \( \overline{N} \)
     \[ \overline{N}=(2^n-1)-N \] Negation
   - 2'er-Komplement \( N^* \)
     \[
     \begin{align*}
     N^* &= 2^n - N \\
     &= (2^n - 1) - N + 1 \\
     &= \overline{N} + 1
     \end{align*}
     \] Negation plus Eins

GEBROCHENE ZWEIER-KOMPLEMENT-DARSTELLUNG

\[ N^* = 2^{n+1} - N \]

- BRUCH-FORM
  \[ X.XXX \ldots X \]
  \[ 0.-1.-2 \ldots .-m \]

- FÜR BRÜCHE
  \[ n=0 \]
  \[ N^* = 2^1 - N \]

- BEISPIEL

Sei \[ N = 0.0100101 \]

\[
\begin{align*}
2 & = 10.0000000 \\
- (N) & = 0.0100101 \\
N^* & = 1.1011011
\end{align*}
\]
GE BROCHENE ZWEIER-KOMPLEMENT-DARSTELLUNG (II)

<table>
<thead>
<tr>
<th>dezim.</th>
<th>binär positive</th>
<th>dezim.</th>
<th>binär negative</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.000</td>
<td>-.875</td>
<td>1.001</td>
</tr>
<tr>
<td>.125</td>
<td>0.001</td>
<td>-.750</td>
<td>1.010</td>
</tr>
<tr>
<td>.250</td>
<td>0.010</td>
<td>-.625</td>
<td>1.011</td>
</tr>
<tr>
<td>.375</td>
<td>0.011</td>
<td>-.500</td>
<td>1.100</td>
</tr>
<tr>
<td>.500</td>
<td>0.100</td>
<td>-.375</td>
<td>1.101</td>
</tr>
<tr>
<td>.625</td>
<td>0.101</td>
<td>-.250</td>
<td>1.110</td>
</tr>
<tr>
<td>.750</td>
<td>0.110</td>
<td>-.125</td>
<td>1.111</td>
</tr>
<tr>
<td>.875</td>
<td>0.111</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ N^* = 2 - N \]

\[
\begin{align*}
2 & = 10.000 \\
- (0.375) & = -.011 \\
-0.375 & = 1.101 \\
.375 & = 0.011 \\
+ (0.250) & = 0.010 \\
+ (-0.250) & = -0.010 = 1.110 \\
.625 & = 0.101 \\
.125 & = 0.001 = 0.001
\end{align*}
\]

* Carry ignoriert

ALGORITHMEN II.

3. Grundlagen der ganzzahligen Arithmetik


**FIGURE A.1 Ripple-carry adder, consisting of \( n \) full adders.**

Die Werte des Carry-Bits wandern vom LSB rechts zum MSB links.

**Quelle:** [01]
Jede Multiplikation besteht aus Addition des Inhaltes von \( P \) zu \( B \) oder 0 in Abhängigkeit vom LSB von \( A \), ersetzen \( P \) mit der Summe und verschieben beide Operanden \( P \) und \( B \) um einen Bit nach rechts.

Jeder Divisionsschritt beinhaltet die Verschiebung der beiden Operanden \( P \) und \( B \) um eine Bit nach links, subtrahieren \( B \) aus \( P \) und wenn die Differenz nicht negativ ist, so wird \( P \) mit dem Ergebnis der Subtraktion ersetzt. Sollte die Differenz nicht negativ sein, wird der LSB von \( A \) mit dem Wert "1" gesetzt.

Quelle: [01]
### FIGURE A.3 Numerical example of (a) restoring division and (b) nonrestoring division.

Quelle: [01]
FIGURE A.3 Numerical example of (a) restoring division and (b) nonrestoring division.

Quelle: [01]
FIGURE A.4 Numerical example of Booth recoding.

Multiplikation von \(a = -6\) mit \(b = -5\) mit dem Ergebnis +30

Quelle: [01]
Literatur


ALGORITHMEN II.

3.1. Addition und Subtraktion

Figure 4.2: Prinzip der binären n-Bit parallelen Addition

Quelle [02]

Quelle: [01]
Literatur


ALGORITHMEN II.

3.2 Multiplikation und Division

Jede Multiplikation besteht aus Addition des Inhaltes von P zu B oder 0 in Abhängigkeit vom LSB von A, ersetzen P mit der Summe und verschieben beide Operanden P und B um einen Bit nach rechts.

Jeder Divisionsschritt beinhaltet die Verschiebung der beiden Operanden P und B um eine Bit nach links, subtrahieren B aus P und wenn die Differenz nicht negativ ist, so wird P mit dem Ergebnis der Subtraktion ersetzt. Sollte die Differenz nicht negativ sein, wird der LSB von A mit dem Wert "1" gesetzt.

Quelle: [01]
**FIGURE A.3** Numerical example of (a) restoring division and (b) nonrestoring division.

Quelle: [01]

---

*Prof. Dr. V. Iossifov, TI, FHTW Berlin*
FIGURE A.3 Numerical example of (a) restoring division and (b) nonrestoring division.
Algorithmen II
Multipikation und Division mit ganzzahliger Arithmetik

5 von 6

Multiplikation von \( a = -6 \) mit \( b = -5 \) mit dem Ergebnis +30

Quelle: [01]

Prof. Dr. V. Iossifov, TI, FHTW Berlin
Literatur


BOOTH ALGORITHMUS


2. a. Es werden Beispiele dieses Schemas gezeigt (ein positiver Multiplizierer und ein negativer Multiplizierer). Das erste Beispiel zeigt, wie Nullen für eine schnellere Implementierung übersprungen werden können.

b. Die Zeichnung veranschaulicht, wie die Booth-Listen erstellt werden, und die nächste Zeichnung zeigt die Übergangstabelike.

c. So zeigt ein Flußdiagramm bzw. eine Zeichnung die Benutzen des Algorithmus für die Multiplikation von 2-er Komplementzahlen.


e. Der Algorithmus erreicht das Folgende:

3. Es wäre wünschenswert die Booth-Technik so anzuwenden, daß einige der Daten-Abhängigkeit entfallen

Quellen:
Malek, M.: Vorlesung "Rechnerarchitektur", Sommersemester 2000, HU Berlin,
www.informatik.hu-berlin.de/~rok/ca
Dave Patterson Computer Architecture and Engineering, September 12, 1997, © U.C.B.
www.cs.berkeley.edu/~patterson
REPRÄSENTATION VON BOOTH

- Jede binäre Zahl kann durch die Summe oder Differenz anderer binärer Zahlen repräsentiert werden.
- Z. B. kann \(30_{10}(0011110_2)\) durch \(32_{10}(0100000_2)\) minus \(2_{10}(000010_2)\) repräsentiert werden. Siehe folgendes Beispiel:

\[
\begin{array}{cccccccc}
0 & 1 & 0 & 0 & 0 & 0 & 0 & (32) \\
- & & & & & 0 & 0 & 0 & 0 & 1 & 0 & (2) \\
0 & 0 & 1 & 1 & 1 & 1 & 0 & (30) \\
\end{array}
\]

\[0 \times 1 \rightarrow 0 \times 0 \rightarrow -1 \times 0 \text{ Booth}\]

- Benutzt man eine vorzeichenbehafte binäre Notation kann \(30_{10}\) als \((0 \times 1 \times 0 \times 0 \times -1 \times 0)\) repräsentiert werden.
- Beachten Sie, daß Übergänge im Schema Multiplikatoren festlegen:
  - \(0 \rightarrow \rightarrow 1 \quad \#(1) \text{ Multiplikator}\)
  - \(1 \rightarrow \rightarrow 0 \quad \#(-1) \text{ Multiplikator}\)
  - Wiederholungen \#(0) \text{ Multiplikator}
- Von rechts nach links scannen.

Quellen:
Malek, M.: Vorlesung "Rechnerarchitektur", Sommersemester 2000, HU Berlin,
www.informatik.hu-berlin.de/~rok/ca
Dave Patterson Computer Architecture and Engineering, September 12, 1997, © U.C.B.
www.cs.berkeley.edu/~patterson
Integer Multiplikation

Booth's Algorithmus

\[
\begin{array}{cccccc}
1 & 0 & 0 & 1 & 1 & (-13) \\
\times & 0 & 1 & 0 & 1 & 1 \\
\hline
1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & (-143)
\end{array}
\]

Vorzeichenerweiterung für einen negativen Multiplikand

\[
\begin{array}{cccccccc}
0 & 1 & 0 & 1 & 1 & 0 & 1 & (-45) \\
0 & 0 & +1 & +1 & +1 & 0 & (-30) \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 1 & 0 & 1 \\
0 & 1 & 0 & 1 & 1 & 0 & 1 \\
0 & 1 & 0 & 1 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & (+1350) \\
0 & 1 & 0 & 1 & 1 & 0 & 1 \\
0 & +1 & 0 & 0 & 0 & 0 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \text{ (2-er Kompl. des Multiplikanden)} \\
0 & \cdots & \cdots & \cdots & 0 \\
0 & \cdots & \cdots & \cdots & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0
\end{array}
\]

Normales und Booths Multiplikationsschema.

**Quellen:**
Booth-Multiplikation mit einem negativen Multiplikator

<table>
<thead>
<tr>
<th>Multiplikator</th>
<th>Version des Multiplikators, der von Bit i ausgewählt wird</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit i</td>
<td>Bit i-1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Booth Multiplikator-Umkodierungstabelle.

Quellen:
**BIT-PAAR**

\[
\begin{array}{cccccccccccc}
0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\
\end{array}
\Rightarrow
\begin{array}{cccccccccccc}
+1 & -1 & +1 & -1 & +1 & -1 & +1 & -1 & +1 & -1 & +1 & -1 \\
0 & -1 & 0 & 0 & +1 & -1 & +1 & 0 & -1 & +1 & 0 & 0 \\
0 & 0 & 0 & +1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & +1 \\
\end{array}
\Rightarrow
\]

-1 schlecht

0 -1 0 0 +1 -1 +1 0 -1 +1 0 0 -1 normal

0 0 0 +1 0 0 0 0 -1 0 0 0 +1 0 0 -1 gut

- Booth-umkodierte Multiplikatoren

\[\begin{array}{lllll}
\text{Vorzeichen} & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\
\text{erweiterung} & 0 & 0 & -1 & +1 & -1 & 0 \\
\text{impliziert} 0 \text{ rechts am LSB} & -1 & -2
\end{array}\]

- Beispiel einer Bit-Paar Umkodierung durch Booth

<table>
<thead>
<tr>
<th>Multiplikat. Bit-Paar</th>
<th>Multiplikat. Bit rechts</th>
<th>Multiplicand ausgewählt bei Spiegelsänderung</th>
</tr>
</thead>
<tbody>
<tr>
<td>i+1 i</td>
<td>i-1</td>
<td>bei Spiegelsänderung</td>
</tr>
<tr>
<td>0 0</td>
<td>0</td>
<td>0 x M</td>
</tr>
<tr>
<td>0 0</td>
<td>1</td>
<td>+1 x M</td>
</tr>
<tr>
<td>0 1</td>
<td>0</td>
<td>+1 x M</td>
</tr>
<tr>
<td>0 1</td>
<td>1</td>
<td>+2 x M</td>
</tr>
<tr>
<td>1 0</td>
<td>0</td>
<td>-2 x M</td>
</tr>
<tr>
<td>1 1</td>
<td>0</td>
<td>-1 x M</td>
</tr>
<tr>
<td>1 1</td>
<td>1</td>
<td>0 x M</td>
</tr>
</tbody>
</table>

- Tabelle der Multiplikandenauswahl

---

**Quellen:**


Booth-Algorithmus - ein Flussdiagramm

Quellen:
<table>
<thead>
<tr>
<th>M</th>
<th>A</th>
<th>Q</th>
<th>Kommentar</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.010</td>
<td>0.000</td>
<td>10110</td>
<td>Subtrahiere M</td>
</tr>
<tr>
<td>0.010</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.110</td>
<td>1.110</td>
<td></td>
<td>Verschiebe A,Q</td>
</tr>
<tr>
<td>1.111</td>
<td>01011</td>
<td></td>
<td>Keine Add. oder Subtrakt.</td>
</tr>
<tr>
<td>1.111</td>
<td>10101</td>
<td></td>
<td>Verschiebe A,Q</td>
</tr>
<tr>
<td>0.010</td>
<td></td>
<td></td>
<td>Addiere M</td>
</tr>
<tr>
<td>0.001</td>
<td>10101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.000</td>
<td>11010</td>
<td></td>
<td>Verschiebe A,Q</td>
</tr>
<tr>
<td>0.010</td>
<td></td>
<td></td>
<td>Subtrahiere M</td>
</tr>
<tr>
<td>1.110</td>
<td>11010</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

--- Produkt P --------

Beispiel einer 2er-Komplement-Multiplikation, die den Booths Algorithmus benutzt.

Quellen:
Malek, M.: Vorlesung “Rechnerarchitektur”, Sommersemester 2000, HU Berlin,
[www.informatik.hu-berlin.de/~rok/ca](http://www.informatik.hu-berlin.de/~rok/ca)
Dave Patterson Computer Architecture and Engineering, September 12, 1997, © U.C.B.
[www.cs.berkeley.edu/~patterson](http://www.cs.berkeley.edu/~patterson)
BOOTH'S Algorithmus - Beispiele

Motivation for Booth’s Algorithm

◦ Example 2 x 6 = 0010 x 0110:

\[
\begin{array}{c}
0010 \\
\times 
\end{array} \begin{array}{c}
0110 \\
+ 
\end{array} \begin{array}{c}
0000 \\
+ 
\end{array} \begin{array}{c}
0010 \\
+ 
\end{array} \begin{array}{c}
0100 \\
+ 
\end{array} \begin{array}{c}
0000 \\
\end{array} \begin{array}{c}
\text{shift (0 in multiplier)} \\
\text{add (1 in multiplier)} \\
\text{add (1 in multiplier)} \\
\text{shift (0 in multiplier)} \\
\end{array}
\]

00001100

◦ ALU with add or subtract gets same result in more than one way:

\[
\begin{array}{c}
6 \\
0110 \\
\end{array} \begin{array}{c}
= - 2 + 8 \\
= - 00010 + 01000 = 11110 + 01000 \\
\end{array}
\]

◦ For example

\[
\begin{array}{c}
0010 \\
\times 
\end{array} \begin{array}{c}
0110 \\
\end{array} \begin{array}{c}
0000 \\
\text{shift (0 in multiplier)} \\
\end{array} \begin{array}{c}
\text{sub (first 1 in multiplier)} \\
0000 \\
\text{shift (mid string of 1s)} \\
+ 
\end{array} \begin{array}{c}
0010 \\
\text{add (prior step had last 1)} \\
\end{array}
\]

00001100

Source: Dave Patterson, Computer Architecture and Engineering © U.C.B.

Quellen:
Malek, M.: Vorlesung "Rechnerarchitektur", Sommersemester 2000, HU Berlin,
[www.informatik.hu-berlin.de/~rok/ca](http://www.informatik.hu-berlin.de/~rok/ca)
Dave Patterson Computer Architecture and Engineering, September 12, 1997, © U.C.B.
[www.cs.berkeley.edu/~patterson](http://www.cs.berkeley.edu/~patterson)
Booth’s Algorithm

<table>
<thead>
<tr>
<th>Current Bit</th>
<th>Bit to the Right</th>
<th>Explanation</th>
<th>Example</th>
<th>Op</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>Begins run of 1s</td>
<td>0001111000</td>
<td>sub</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Middle of run of 1s</td>
<td>0001111000</td>
<td>none</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>End of run of 1s</td>
<td>0001111000</td>
<td>add</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Middle of run of 0s</td>
<td>0001111000</td>
<td>none</td>
</tr>
</tbody>
</table>

Originally for Speed (when shift was faster than add)

- Replace a string of 1s in multiplier with an initial subtract when we first see a one and then later add for the bit after the last one

\[
\begin{align*}
-1 & \quad + 10000 \\
& \quad = 01111
\end{align*}
\]

Source: Dave Patterson, Computer Architecture and Engineering © U.C.B.

Quellen:
### Booth's Example (2 x 7)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Multiplicand</th>
<th>Product</th>
<th>next?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0. initial value</td>
<td>0010</td>
<td>0000 0111 0</td>
<td>10 -&gt; sub</td>
</tr>
<tr>
<td>1a. P = P - m</td>
<td>1110</td>
<td>+ 1110 0111 0</td>
<td>shift P (sign ext)</td>
</tr>
<tr>
<td>1b.</td>
<td>0010</td>
<td>1111 0011 1</td>
<td>11 -&gt; nop, shift</td>
</tr>
<tr>
<td>2.</td>
<td>0010</td>
<td>1111 1001 1</td>
<td>11 -&gt; nop, shift</td>
</tr>
<tr>
<td>3.</td>
<td>0010</td>
<td>1111 1100 1</td>
<td>01 -&gt; add</td>
</tr>
<tr>
<td>4a.</td>
<td>0010</td>
<td>+ 0010 0001 1100 1</td>
<td>shift</td>
</tr>
<tr>
<td>4b.</td>
<td>0010</td>
<td>0000 1110 0</td>
<td>done</td>
</tr>
</tbody>
</table>
### Booths Example (2 x -3)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Multiplicand</th>
<th>Product</th>
<th>next?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0. initial value</td>
<td>0010</td>
<td>0000 1101 0</td>
<td>10 -&gt; sub</td>
</tr>
<tr>
<td>1a. P = P - m</td>
<td>1110</td>
<td>+ 1110</td>
<td>shift P (sign ext)</td>
</tr>
<tr>
<td>1b.</td>
<td>0010</td>
<td>1111 0110 1</td>
<td>01 -&gt; add</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+ 0010</td>
<td></td>
</tr>
<tr>
<td>2a.</td>
<td></td>
<td>0001 0110 1</td>
<td>shift P</td>
</tr>
<tr>
<td>2b.</td>
<td>0010</td>
<td>0000 1011 0</td>
<td>10 -&gt; sub</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+ 1110</td>
<td></td>
</tr>
<tr>
<td>3a.</td>
<td>0010</td>
<td>1110 1011 0</td>
<td>shift</td>
</tr>
<tr>
<td>3b.</td>
<td>0010</td>
<td>1111 0101 1</td>
<td>11 -&gt; nop</td>
</tr>
<tr>
<td>4a</td>
<td>1111 0101 1</td>
<td>shift</td>
<td></td>
</tr>
<tr>
<td>4b.</td>
<td>0010</td>
<td>1111 1010 1</td>
<td>done</td>
</tr>
</tbody>
</table>

Quellen:
Dave Patterson Computer Architecture and Engineering, September 12, 1997, © U.C.B.
www.cs.berkeley.edu/~patterson
ALGORITHMMEN II.

4.0 Integer Additierer

FIGURE A.14: Pure carry-lookahead circuit for computing the carry-out $c_n$ of an $n$-bit adder

Quelle: [01]
Die Werte für $a_i$ und $b_i$ werden im binären Baum der Voll-Adder berechnet.

Quelle: [01]
Signalfluß vom Wurzel zu den Zweigen des binären Voll-Adder Baumes verbindet die Werte von G, P und bestimmt die Werte der Carry-Bits $c_i$.

Quelle: [01]
Das ist die Kombination Bilder 15. und 16. Die Zahlen, die addiert werden sollen, werden in den Blättern des binären Voll-Adder Baumes eingegeben, werden zum Wurzel durchgeschaltet um mit den Carry-Werten $c_0$ verknüpft zu werden und werden zurück zu den Blättern geführt um die Bits der Summe zu bilden.

Quelle: [01]
FIGURE A.18 Combination of CLA and ripple-carry adder.

Quelle: [01]
Beschleunigen der gazzahligen Addition

FIGURE A.19 Carry-ripple adder.

Quelle: [01]
FIGURE A.20 Simple carry-select adder.

Die Summe der LS vier Bits und die Summe der MS vier Bits werden gleichzeitig gebildet, wobei die Summe der MS vier Bits wird zwei mal in parallel gebildet - für $c_4 = 1$ und $c_4 = 0$.

Quelle: [01]
Sobald der Carry-Ausgang des rechtesten Blocks bestimmt wird, wird er eingesetzt um die anderen Bit der Summe zu bestimmen.

Quelle: [01]
Literatur


ALGORITHMEN II.

4. Entwurf einer ALU

Entwurf einer ALU

1. Entwurf eines 1-Bit ALU
   1.1 Entwurf der Logikeinheit
   1.2 Entwurf des VA aus der Wertetabelle des VA

2. Entwurf eines 32-Bit ALU
   2.1 Schaltung des 32-Bit ALU
   2.2 Schaltung und Steuerung des 2-er Komplement berechnung

3. Weiterentwicklung des ALU
   3.1 Entwurf des "kleiner als"-Befehl
   3.2 Carry Lookahead Schaltung
      3.2.1 Fast Carry Schaltung mit Carry Propagate (pi) und Carry Generate (qi).
      3.2.2 16-bit Adder mit vier 4-bit ALU und "Carry-Lookahead" Logik.
      3.2.3 Relative Leistung Ripple Carry/Lookahead Carry

Prof. Dr. V. Iossifov, TI, FHTW Berlin
1. **AND gate** \((c = a \cdot b)\)

\[
\begin{array}{c|c|c|c}
 a & b & c = a \cdot b \\
\hline
 0 & 0 & 0 \\
 0 & 1 & 0 \\
 1 & 0 & 0 \\
 1 & 1 & 1 \\
\end{array}
\]

2. **OR gate** \((c = a + b)\)

\[
\begin{array}{c|c|c|c}
 a & b & c = a + b \\
\hline
 0 & 0 & 0 \\
 0 & 1 & 1 \\
 1 & 0 & 1 \\
 1 & 1 & 1 \\
\end{array}
\]

3. **Inverter** \((c = \bar{a})\)

\[
\begin{array}{c|c|c}
 a & c = \bar{a} \\
\hline
 0 & 1 \\
 1 & 0 \\
\end{array}
\]

4. **Multiplexer**
   (if \(d = 0\), \(c = a\);
   else \(c = b\))

\[
\begin{array}{c|c|c}
 d & c \\
\hline
 0 & a \\
 1 & b \\
\end{array}
\]

**Figure 4.7:** Vier logische Komponenten, die bei der Konstruktion einer ALU eingesetzt werden.

Quelle [02]
1-Bit LU für AND und OR

Quelle: [02]
Figure 4.8 1-Bit Addierer. Das ist ein Voll-Addierer, also ein (3,2) Addierer mit 3 Eingängen und 2 Ausgängen. Ein (2,2) Addierer mit 2 Eingängen und 2 Ausgängen ist ein Halb-Addierer.

Quelle: [02]
Figure 4.10 Die Schaltung des Voll-Addierers

Quelle: [02]
Figure 4.11 1-Bit ALU für die Operationen AND, OR und Addition
Quelle: [02]
Figure 4.12 32-Bit ALU entworfen mit 32 1-Bit ALUs. CarryOut des Less Significant Bit wird mit dem CarryIn des More Significant Bit verbunden und so die Fortpflanzung des Carry-Wertes durchgeführt. Diese ALU Organization wird ripple carry genannt.

Quelle: [02]
Figure 4.13 1-Bit ALU für die Operationen AND, OR und Addition für die Werte a und b oder a und ~b.

Quelle: [02]
Figure 4.14 1-Bit ALU für die Operationen AND, OR und Addition für die Werte a und b oder ¬b.

Figure 4.15 1-Bit ALU für den *Most Significant Bit*.

Quelle: [02]
Quelle: [02]  Figure 4.16: Ein 32-Bit ALU konstruiert aus 31 1-bit ALUs aus Fig. 4.14 und ein 1-bit ALU aus Fig. 4.15. Die Less-Eingänge werden mit NULL belegt mit Ausnahme des Least Significant Bit und dieser Eingang ist mit dem Set-Ausgang des Most Significant Bit.

Quelle: [02]
Figure 4.17: Die endgültige Schaltung des 32-Bit ALU. Diese Schaltung ergänzt die Schaltung aus Fig. 4.16 mit einem NULL-Detektor als eine 1-Bit ALU.

Quelle: [02]
Figure 4.18: Das Symbol, welches für die Darstellung eines ALU eingesetzt wird.

Quelle: [02]
Figure 4.22 Ein hydraulischer Vergleich für "carry lookahead" für 1 Bit, 2 Bits und 4 Bits mit Einsatz von Wasser, Röhren und Ventilen.

Quelle: [02]
Figure 4.23 Ein hydraulischer Vergleich für den "next-level carry-lookahead" Signale P0 und G0.

Quelle: [02]
Figure 4.24 Vier 4-Bit "carry-lookahead" ALUs bilden einen 16-bit Addierer.

Quelle: [02]
Literatur


Arithmetisch-Logische Einheiten
(arithmetic logic units, ALU's)

- ALU's stellen eine Reihe von Operationen zur arithmetischen und logischen Verknüpfung in Abhängigkeit von Steuersignalen zur Verfügung
- ALU's sind zentraler Bestandteil von Prozessoren
- Beispiel: "klassische" SN74181 ALU
Schema:

| S3 | S2 | S1 | S0 | M = H | M = L
|----|----|----|----|------|------|
| L  | L  | L  | L  | F = \text{not}(A) | F = A
| L  | L  | L  | H  | F = \text{not}(A+B) | F = A + B
| L  | L  | H  | L  | F = \text{not}(A)B | F = A + \text{not}(B)
| L  | L  | H  | H  | F = 0 | F = \text{MINUS} 1
| L  | H  | L  | L  | F = \text{not}(A \land B) | F = A \text{ PLUS} A \text{ not}(B)
| L  | H  | L  | H  | F = \text{not}(B) | F = (A + B) \text{ PLUS} A \text{ not}(B)
| L  | H  | H  | L  | F = A \text{ \& } B | F = A \text{ MINUS} B \text{ MINUS} 1
| L  | H  | H  | H  | F = A \text{ not}(B) | F = A \text{ not}(B) \text{ MINUS} 1
| H  | L  | L  | L  | F = \text{not}(A+B) | F = A \text{ PLUS} A \text{ not}(B)
| H  | L  | L  | H  | F = B | F = (A + \text{not}(B)) \text{ PLUS} AB
| H  | L  | H  | H  | F = A \text{ \& } B | F = A \text{ \& } B \text{ MINUS} 1
| H  | H  | L  | L  | F = 1 | F = A \text{ PLUS} A
| H  | H  | L  | H  | F = A + \text{not}(B) | F = (A + B) \text{ PLUS} A \text{ PLUS} 1
| H  | H  | H  | L  | F = A + B | F = (A + \text{not}(B)) \text{ PLUS} A
| H  | H  | H  | H  | F = A | F = A

Schaltung:

Multiplizierer

Multiplikationsbeispiel:

\[ A_3 A_2 A_1 A_0 = 1011 \quad (~ 11) \]

\[ \begin{array}{ccc}
1 & 0 & 1 \ 1 \\
0 & 0 & 0 \ 0 \\
0 & 0 & 0 \ 0 \\
1 & 0 & 1 \ 1 \\
\end{array} \quad \begin{array}{c}
B_0 \\
B_1 \\
B_2 \\
B_3 \\
\end{array} \quad (~ 9) \]

\[ \begin{array}{c}
0 \ 1 \ 1 \ 0 \ 0 \ 0 \ 1 \ 1 \\
\end{array} \quad (~ 99) \]
4-bit Multiplizierer:

\[ \begin{array}{cccccc}
A_3 & A_2 & A_1 & A_0 \\
0 & 0 & 0 & 0 \\
\end{array} \]

\[ \begin{array}{ccccccc}
B_0 & B_1 & B_2 & B_3 \\
0 & 0 & 0 & 0 \\
\end{array} \]

\[ \begin{array}{cccccc}
P_7 & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0 \\
\end{array} \]

Aufbau jeder Zelle:
Durchlaufzeit: \( \sim 3n-2 \)
Durchlaufzeit: \( \sim 3n-2 \)
Multiplizierer mit Carry-Save Addierern

- Prinzip: die in einem Partialprodukt entstehenden Überträge werden an die Berechnung des nächsten Partialprodukts weitergereicht.

- Beispiel für eine Addition mit Carry-Save Addierer:

```
  1 0 1 1  ~ 11
+ 0 1 1 1  ~  7
  
  Summe:  1 1 0 0  ~ 12
  Übertrag: 0 0 1 1  ~  6
            1 0 0 1 0
```

die Überträge werden nicht an die folgende Stelle weitergereicht. Nochmalige Addition mit Übertrag ist zur Ermittlung des richtigen Resultats notwendig.
> multiplizieren nach dem Carry-Save Prinzip:

Multiplikand

\[ A_3 A_2 A_1 A_0 = 1 0 1 1 \sim 11 \]

1. Summe: \[ 1 0 1 1 \quad 1 \quad B_0 \]
1. Übertrag: \[ 0 \quad 0 \quad 0 \quad 0 \]

\[ + \quad 1 \quad 0 \quad 1 \quad 1 \quad 1 \quad B_1 \]

2. Summe: \[ 1 \quad 1 \quad 1 \quad 0 \]
2. Übertrag: \[ 0 \quad 0 \quad 0 \quad 1 \]

\[ + \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1 \quad B_2 \]

3. Summe: \[ 0 \quad 1 \quad 1 \quad 0 \]
3. Übertrag: \[ 0 \quad 0 \quad 0 \quad 1 \]

\[ + \quad 1 \quad 0 \quad 1 \quad 1 \quad 1 \quad B_3 \]

4. Summe: \[ 1 \quad 0 \quad 0 \quad 1 \]
4. Übertrag: \[ 0 \quad 0 \quad 1 \quad 1 \]

Endaddition

Ripple-Carry: \[ 0 \quad 0 \quad 1 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \sim 121 \]

Schaltung:

> Durchlaufzeit
bei Ripple-Carry
Addierer in der
untersten Ebene: \( \sim 2n \)

Quellen: Rechnerarchitektur I, II, III, Fachgebiet Rechnersysteme, TU Darmstadt,
http://www.rs.e-technik.tu-darmstadt.de/TUD/lectures/lectures.html
Durchlaufzeit bei Ripple-Carry Addierer in der untersten Ebene: \( \sim 2n \)
Multiplier mit einer Konstanten

> "#" bedeutet Vektor-Konkatenation, z.B. 0 # 1 = 01

\[
\begin{align*}
a & \quad 0100010 \\
8 & \quad 8 \\
16 & \quad r \\
a \times 64 + a \times 2
\end{align*}
\]

\[
\begin{align*}
a\#000000 & \quad 00000\#a\#0 \\
14 & \quad 14 \\
15 & \quad r \\
0
\end{align*}
\]
\[ a \cdot 01111110 = a \cdot (10000000 - 00000010) \]
N-BIT ADDIERER/SUBTRAHIERER MIT 2'ER-KOMPLEMENT ADD/SUBTRAKT KONTROLLE
2'ER-KOMPLEMENT ADD/SUBTRAKT KONTROLLE

2'er-Komplement's großer Vorteil:
der gleiche Schaltkreis für ADD/SUB

\[ ADD \quad \rightarrow \quad S = X + Y \]
\[ SUB \quad \rightarrow \quad S = X + \overline{Y} + 1 \]

---

Malek, M.: Vorlesung "Rechnerarchitektur", Sommersemester 2000, HU Berlin,
www.informatik.hu-berlin.de/~rok/ca
LOGISCHE SCHALTUNG FÜR ADDITION ZWEIER BITS

\[ s_i = \bar{x}_i \bar{y}_i c_i + \bar{x}_i y_i c_i + x_i \bar{y}_i c_i + x_i y_i c_i \]
\[ c_{i+1} = x_i c_i + y_i c_i + x_i y_i \]
\[ s_i = (a_i + b_i + c_i) \mod 2 \]
\[ c_{i+1} = \left\lfloor (a_i + b_i + c_i) / 2 \right\rfloor \]
ENTWURF EINES SCHNELLADDIERERS

Die Gleichungen für zweistufige logische Ausdrücke sind:

(1) \[ s_i = x_i y_i c_i + x_i y_i c_i + x_i y_i c_i + x_i y_i c_i \]

\[ c_{i+1} = x_i c_i + y_i c_i + x_i y_i \]

Umformung des zweiten Ausdrucks (Carry-Out Gleichung)

(2) \[ c_{i+1} = x_i y_i + (x_i + y_i) c_i \]

und definieren eine Generierungsfunktion

(3) \[ G_i = x_i y_i \]

und eine Propagierungsfunktion

(4) \[ P_i = x_i + y_i \]

können wir schreiben

(5) \[ c_{i+1} = G_i + P_i c_i \]

\[ c_{i-1+1} = G_{i-1} + P_{i-1} c_{i-1} \]

(6) \[ c_i = G_{i-1} + P_{i-1} c_{i-1} \]

\[ c_{i+1} = G_i + P_i (G_{i-1} + P_{i-1} c_{i-1}) \]
(7) \[ c_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} c_{i-1} \]

(8) \[ c_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + \ldots + P_{i-1} G_0 + P_{i-1} P_0 c_0 \]

Reine Carry-Lookahead Schaltung für Berechnung des Carry out \( c_n \) eines \( n \)-bit Addierer

\[ c_i = G_{i-1} + P_{i-1} G_{i-2} + \ldots + P_{i-1} \ldots P_1 G_0 + P_{i-1} \ldots P_0 c_0 \]
BLOCK LOOKAHEAD

\[ \tilde{G}_K, \tilde{F}_K \]
\[ \tilde{R}_0 = P_3 P_2 P_1 P_0 \]
\[ \tilde{G}_0 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 G_0 \]

Carry für einen 16-Bit addierer:
\[ c_{16} = \tilde{G}_3 + \bar{P}_3 \tilde{G}_2 + \bar{P}_3 \bar{P}_2 \tilde{G}_1 + \bar{P}_3 \bar{P}_2 \bar{P}_1 \tilde{G}_0 + \bar{P}_3 \bar{P}_2 \bar{P}_1 \bar{P}_0 c_0 \]

Beispiel: ein 4-Bit addierer
\[ c_4 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 G_0 + P_3 P_2 P_1 P_0 c_0 \]

4-Bit integrierter ALU block
TIMING FÜR CARRY LOOKAHEAD

Zwei Ausdrücke müssen ausgewertet werden

a) Carry Lookahead
   \[ c_j = G_{j-1} + P_{j-1}G_{j-2} + \ldots + P_{j-1}G_0 + P_{j-1} \ldots P_0c_0 \]

b) Die Summe
   \[ s_j = \overline{x_jy_j}c_i + \overline{x_jy_j}G_i + \overline{x_j}y_jG_i + x_jy_jc_i \]

Ad a) Der Carry kann innerhalb der Verzögerungszeit dreier logischer Gatter generiert werden.

1 Erzeuge \( P_i, G_i \)
1 AND \( P \)'s, \( G \)'s
1 OR resultierendes AND \( P \)'s, \( G \)'s
3

Ad b) Die vollständige Summe kann innerhalb von 3 weiteren Gatterlaufzeiten komplettiert werden.

1 Form \( c_j \)
1 AND \( x_j, y_j, c_i, x_j, y_j, c_i \)
1 OR AND Produkte
3

D = CA + X + A U + B
**BESCHRÄNKUNG**

a) **Carry Lookahead mit 4 Blöcken (32 Bit, k=8)**

Gatter fan-in ist auf 8 beschränkt (übliche Schaltungsbeschränkung)

<table>
<thead>
<tr>
<th></th>
<th>Generiere $G_i$</th>
<th>Generiere $P_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Form $C_8$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Form $C_{16}$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Form $C_{24}$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Form $C_{31}$</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Form $S_{31}$</td>
<td></td>
</tr>
</tbody>
</table>

12 Delays bei 5 ns pro 1 Gatter → 60 ns für ein Addierer.

b) **Carry Lookahead vollständig integriert (32 bits, k=32)**

ohne Beschränkungen der logischen Gatter

<table>
<thead>
<tr>
<th></th>
<th>Generiere $G_k$</th>
<th>Generiere $P_k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Form $C_{31}$</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Form $S_{31}$</td>
<td></td>
</tr>
</tbody>
</table>

7 Delays bei 5 ns pro 1 Gatter → 35 ns für ein Addierer

$D = C A \cdot X \cdot A U + 19$
CARRY SKIP ADDIERER

Notation: $P_{i,j} = P_i \cdot P_{i+1} \cdots P_j$

CARRY SELEKT ADDIERER

- Parallel werden zwei Additionen ausgeführt: In einem Fall wird Carry = 0 vorausgesetzt; im anderen Fall Carry = 1
- Wenn der tatsächliche Carry bekannt ist, wird die richtige Summe gewählt

$D = CA \cdot X \cdot A \cdot U \cdot ZD$
ZUSAMMENFASSUNG

ADDITIONSTECHNIKEN

Seriente Addition
einfache Logik ↔ langsamer Ausführung

Parallel addition
komplexe Logik ↔ schnellere Ausführung

- Ripple carry
- Carry lookahead
- Carry skip
- Carry select

<table>
<thead>
<tr>
<th>Techniken</th>
<th>Zeit</th>
<th>Raum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ripple</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>CLA (Carry Lookahead)</td>
<td>O(log n)</td>
<td>O(n log n)</td>
</tr>
<tr>
<td>Carry skip</td>
<td>O(√n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Carry select</td>
<td>O(√n)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>
2. Organisationsprinzip: n-Bit Carry-Lookahead Addierer

- parallele Berechnung des Übertrags

\[ s_i = a_i \oplus b_i \oplus c_i \]
\[ c_{i+1} = a_i \cdot b_i + a_i \cdot c_i + b_i \cdot c_i \]
\[ = a_i \cdot b_i + (a_i \oplus b_i) \cdot c_i \]
\[ = g_i + p_i \cdot c_i \]

\[ a_i \quad 0 \quad 1 \quad 0 \quad 1 \quad 0 \quad 1 \quad 0 \quad 1 \]
\[ b_i \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \quad 1 \]
\[ c_i \quad 0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 1 \quad 1 \quad 1 \]
\[ s_i \quad 0 \quad 1 \quad 1 \quad 0 \quad 1 \quad 0 \quad 0 \quad 1 \]
\[ c_{i+1} \quad 0 \quad 0 \quad 0 \quad 1 \quad 0 \quad 1 \quad 1 \quad 1 \]

- ein Übertrag entsteht dadurch, daß er entweder in Stufe i generiert (generate) oder weitergereicht (propagate) wird

- falls \( g_i = 1 \), dann ist unabhängig von \( c_i \) der Ausgangsübertrag \( c_{i+1} = 1 \)

- falls \( p_i = 1 \), dann ist der Ausgangsübertrag \( c_{i+1} = c_i \), \( c_i \) kann also direkt weitergereicht werden
allgemeines Schema:

\[ s_i = a_i \oplus b_i \oplus c_i \]
\[ c_{i+1} = a_i \cdot b_i + a_i \cdot c_i + b_i \cdot c_i \]
\[ = a_i \cdot b_i + (a_i \oplus b_i) \cdot c_i \]
\[ = g_i + p_i \cdot c_i \]

\[ c_1 = g_0 + p_0 \cdot c_0 \]
\[ c_2 = g_1 + p_1 \cdot c_1 = g_1 + p_1 \cdot (g_0 + p_0 \cdot c_0) \]
\[ = g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot c_0 \]

\[ \vdots \]
\[ c_n = g_{n-1} + p_{n-1} \cdot c_{n-1} \]
\[ = g_{n-1} + p_{n-1} \cdot g_{n-2} + p_{n-1} \cdot p_{n-2} \cdot g_{n-3} + \cdots + p_{n-1} \cdots p_0 \cdot c_0 \]
Realisierung mit Gattern:

\[ s_i = a_i \oplus b_i \oplus c_i \]

\[ c_{i+1} = a_i \cdot b_i + a_i \cdot c_i + b_i \cdot c_i \]
\[ = a_i \cdot b_i + (a_i + b_i) \cdot c_i \]

Beispiel n=3:

\[ c_1 = g_0 + p_0 \cdot c_0 \]
\[ c_2 = g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot c_0 \]
\[ c_3 = g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot c_0 \]
\[ c_4 = g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0 + p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot c_0 \]

Eine zweistufige Realisierung zusammen mit der einstufigen Realisierung der \( g \)- und \( p \)-Signale bedeutet eine Durchlaufzeit von drei Gatterstufen.

Problem: beim n-Bit Addierer sind Gatter mit \( n+1 \) Eingängen erforderlich und Signale müssen vielfach verzweigt werden.

Daher: Beschränkung dieses Schemas auf z.B. 4 Bit Addierer.
Int Add Sub

Carry-Lookahead Addierer

Quellen: Rechnerarchitektur I, II, III, Fachgebiet Rechnersysteme, TU Darmstadt,
http://www.rs.e-technik.tu-darmstadt.de/TUD/lectures/lectures.html
Int Add Sub

Carry-Lookahead Addierer

Quellen: Rechnerarchitektur I, II, III, Fachgebiet Rechnersysteme, TU Darmstadt,
http://www.rs.e-technik.tu-darmstadt.de/TUD/lectures/lectures.html
Int Add Sub

Carry-Lookahead Addierer

Quellen: Rechnerarchitektur I, II, III, Fachgebiet Rechnersysteme, TU Darmstadt,
http://www.rs.e-technik.tu-darmstadt.de/TUD/lectures/lectures.html
Int Add Sub

Carry-Lookahead Addierer

- Erweiterung des Carry-generate und -propagate Schemas auf Bit-Gruppen

- der Ausgangsübertrag $c_4$ eines 4-Bit Addierers entsteht dadurch, daß er unabhängig vom Eingangsübertrag $c_0$ generiert wird oder daß er gleich dem Eingangsübertrag $c_0$ ist:

$$c_4 = g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0 + p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot c_0$$
Erweiterung des Carry-generate und -propagate Schemas auf Bit-Gruppen

der Ausgangsübertrag $c_4$ eines 4-Bit Addierers entsteht dadurch, daß er unabhängig vom Eingangsübertrag $c_0$ generiert wird oder daß er gleich dem Eingangsübertrag $c_0$ ist:

$$c_4 = g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0 + p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot c_0$$

$$= G_{3:0} + P_{3:0} \cdot c_0$$

$G_{3:0} = g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0$

$P_{3:0} = p_3 \cdot p_2 \cdot p_1 \cdot p_0$

Durchlaufzeit bis $c_4$ : 3 Gatterstufen (nach obiger Schaltung)
4-Bit Addierer mit Erzeugung der Generate/Propagate-Signale:
Erweiterung auf Gruppen von 4-Bit Addierern, z.B. 16-Bit Addierer:

\[ c_4 = g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0 + p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot c_0 \]
\[ = G_{3:0} + P_{3:0} \cdot c_0 \]
\[ G_{3:0} = g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0 \]
\[ P_{3:0} = p_3 \cdot p_2 \cdot p_1 \cdot p_0 \]

\[ c_8 = G_{7:4} + P_{7:4} \cdot c_4 \]
\[ c_{12} = G_{11:8} + P_{11:8} \cdot c_8 \]
\[ c_{16} = G_{15:12} + P_{15:12} \cdot c_{12} \]

Durchlaufzeit bis \( c_{16} : 3+2\cdot3 = 9 \) Gatterstufen

daher: nochmals Anwendung des Look-ahead-Prinzips
statt:

\[ c_4 = G_{3:0} + P_{3:0} \cdot c_0 \]
\[ c_8 = G_{7:4} + P_{7:4} \cdot c_4 \]
\[ c_{12} = G_{11:8} + P_{11:8} \cdot c_8 \]
\[ c_{16} = G_{15:12} + P_{15:12} \cdot c_{12} \]

durch Substitution:

\[ c_4 = G_{3:0} + P_{3:0} \cdot c_0 \]
\[ c_8 = G_{7:4} + P_{7:4} \cdot G_{3:0} + P_{7:4} \cdot P_{3:0} \cdot c_0 \]
\[ c_{12} = G_{11:8} + P_{11:8} \cdot G_{7:4} + P_{11:8} \cdot P_{7:4} \cdot G_{3:0} + P_{11:8} \cdot P_{7:4} \cdot P_{3:0} \cdot c_0 \]
\[ c_{16} = G_{15:12} + P_{15:12} \cdot G_{11:8} + P_{15:12} \cdot P_{11:8} \cdot G_{7:4} + P_{15:12} \cdot P_{11:8} \cdot P_{7:4} \cdot G_{3:0} + P_{15:12} \cdot P_{11:8} \cdot P_{7:4} \cdot P_{3:0} \cdot c_0 \]

> Durchlaufzeit bis \( c_{16} : 3+2 = 5 \) Gatterstufen
**Carry-Lookahead 16-Bit Addierer**

[Diagram of Carry-Lookahead Adder with labeled inputs and outputs]

*Quellen:*
Rechnerarchitektur I, II, III, Fachgebiet Rechnersysteme, TU Darmstadt, [http://www.rs.e-technik.tu-darmstadt.de/TUD/lectures/lectures.html](http://www.rs.e-technik.tu-darmstadt.de/TUD/lectures/lectures.html)
Int Add Sub  
Carry-Lookahead Addierer

Quellen: Rechnerarchitektur I, II, III, Fachgebiet Rechnersysteme, TU Darmstadt, 
http://www.rs.e-technik.tu-darmstadt.de/TUD/lectures/lectures.html
Quellen: 
Rechnerarchitektur I, II, III, Fachgebiet Rechnersysteme, TU Darmstadt,  
http://www.rs.e-technik.tu-darmstadt.de/TUD/lectures/lectures.html 
Int. Add Sub

Carry-Lookahead Addierer

Quellen: Rechnerarchitektur I, II, III, Fachgebiet Rechnersysteme, TU Darmstadt,
http://www.rs.e-technik.tu-darmstadt.de/TUD/lectures/lectures.html
entsprechend werden beim 64-Bit Addierer Generate- und Propagate-Signale von vier 16-Bit Addierern erzeugt: \( G_{63:48}, G_{47:32}, G_{31:16}, G_{15:0} \) bzw. \( P_{63:48}, P_{47:32}, P_{31:16}, P_{15:0} \).

- Durchlaufzeit: 3+2+2+3 Gatterstufen
- Resultat:

<table>
<thead>
<tr>
<th>Addierer</th>
<th>Laufzeit</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-Bit</td>
<td>4</td>
</tr>
<tr>
<td>16-Bit</td>
<td>8</td>
</tr>
<tr>
<td>64-Bit</td>
<td>12</td>
</tr>
</tbody>
</table>

Die Laufzeit des Carry-Lookahead Addierers wächst logarithmisch in \( n \), d.h. \(~ \log(n)\).
ALGORITHMEN II.

4.2 Carry-Select, Carry-Safe Addierer
3. Organisationsprinzip: n-Bit Carry-Select Addierer

- parallele Berechnung der höherwertigen Stellen für **beide** möglichen Werte des Übertrags
- Beispiel:

![Diagram of Carry-Select Adder](image)

Quellen: Rechnerarchitektur I, II, III, Fachgebiet Rechnersysteme, TU Darmstadt, [http://www.rs.e-technik.tu-darmstadt.de/TUD/lectures/lectures.html](http://www.rs.e-technik.tu-darmstadt.de/TUD/lectures/lectures.html)
3. Organisationsprinzip: n-Bit Carry-Select Addierer

- parallele Berechnung der höherwertigen Stellen für beide möglichen Werte des Übertrags
- Beispiel:

![Diagram of Carry-Select Adder]

vier 2:1 Multiplexoren
**Carry-Save Addierer (CSA)**

- Kette von Volladdierern, die jeweils Summe und Übertrag produzieren
- Die Verzögerungszeit ist die eines einzelnen Volladdierers

\[
\begin{align*}
\sum & \quad \sum & \quad \sum & \quad \sum \\
\downarrow & \quad \downarrow & \quad \downarrow & \quad \downarrow \\
c_{o4} & s_3 & c_{o3} & s_2 \\
\downarrow & \quad \downarrow & \quad \downarrow & \quad \downarrow \\
c_{o1} & s_0
\end{align*}
\]
Die Abbildung zeigt den Carry-Save Addierer und den Ripple-Carry Addierer. Der Carry-Save Addierer (CSA) verwendet ein Carry Select-Mechanismus, während der Ripple-Carry Addierer eine Ripple-Carry-Funktion zeigt. Die Eingänge sind mit $a_{3:0}$, $b_{3:0}$ und $c_{i:0}$ bezeichnet, wobei $c_{i:0}$ der Carry-In-Input und die Summe $s_{3:0}$ resultiert.
Anwendungsbeispiel: Addition dreier Zahlen

\[
\begin{align*}
1 & 0 & 1 & 1 & \sim & 11 \\
+ & 0 & 1 & 1 & 1 & \sim & 7 \\
+ & 1 & 0 & 0 & 1 & \sim & 9 \\
\text{Summe:} & & 0 & 0 & 1 & 0 & 1 & \sim & 5 \\
\text{Übertrag:} & & 1 & 0 & 1 & 1 & 0 & \sim & 22 \\
\hline
1 & 1 & 0 & 1 & 1 & \sim & 27
\end{align*}
\]

die Überträge werden nicht an die folgende Stelle weitergereicht. Nochmalige Addition mit Übertrag ist zur Ermittlung des richtigen Resultats notwendig.

Quellen: Rechnerarchitektur I, II, III, Fachgebiet Rechnersysteme, TU Darmstadt, [http://www.rs.e-technik.tu-darmstadt.de/TUD/lectures/lectures.html](http://www.rs.e-technik.tu-darmstadt.de/TUD/lectures/lectures.html)
Subtrahierer:

- die Subtraktion in K2-Darstellung läßt sich auf die Addition des invertierten Vektors plus 1 zurückführen

- gesteuerter Addierer (k=0) / Subtrahierer (k=1):

\[ \begin{align*}
& \text{Int Add Sub} \\
& \text{Carry Select / Carry Safe Addierer} \\
& \text{8 aus 8} \\
& \text{Quellen: Rechnerarchitektur I, II, III, Fachgebiet Rechnersysteme, TU Darmstadt,} \\
& \text{http://www.rs.e-technik.tu-darmstadt.de/TUD/lectures/lectures.html} \\
\]
ALGORITHMEN II.

5. Integer Multiplikation

Figure 4.25 Version 1. der Multiplikationsschaltung. Multiplikand-Register, ALU und Produktregister sind 64 Bit breit. Multiplikator-Register ist 32 Bit breit. Der 32 Bit Multiplikand liegt am Anfang der Multiplikation in der rechten Hälfte des Multiplikand-Registers und wird in jedem Schritt um 1 Bit nach links verschoben. Der Multiplikator wird in jedem Schritt um 1 Bit nach rechts verschoben. Der Algorithmus beginnt mit einem mit NULL initialisierten Produkt. Der Control-Block entscheidet wann die Inhalte der Multiplikand- und Multiplikator-Register verschoben werden sollen und wann neue Werte im Produkt-Register geschrieben werden sollen.

Quelle: [02]
Figure 4.26 Version 1. des Multiplikationsalgorithmus nach Figure 4.25. Sollte der least significant bit des Multiplikators den Wert 1 haben, so wird der Multiplikand zum Produkt addiert. Wenn LSB des Multiplikators den Wert 0 haben sollte, so gehe zum nächsten Schritt über. Verschiebe den Multiplikand nach links und den Multiplikator nach rechts um 1 Bit in den nächsten zwei Schritten. Diese drei Schritte werden 32 mal für 32-Bit Operanden wiederholt.

Quelle: [02]
Figure 4.28 Version 2. der Multiplikationsschaltung. Multiplikand-Register, ALU und Multiplikator-Register sind 32 Bit breit. Nur der Produktregister ist 64 Bit breit. Jetzt wird der Produkt wird in jedem Schritt um 1 Bit nach rechts verschoben.

Quelle: [02]
Figure 4.29 Version 2. des Multiplikationsalgorithmus nach Figure 4.23. In dieser Version wird das Produkt an Stelle des Multiplikanden nach rechts verschoben.

Quelle: [02]

Prof. Dr. V. Iossifov, TI, FHTW Berlin
Figure 4.31 Version 3. der Multiplikationsschaltung. Im Vergleich zur versuin 2. Wurde hier auf einen Multiplikator-Register verzichtet. Der Multiplikator wird in der rechten Hälfte des Produkt-Registers plaziert.

Quelle: [02]
Figure 4.32 Version 3. Des Multiplikationsalgorithmus. Dieser Algorithmus braucht nur zwei Schritte, weil die Produkt- und Multiplikator-Register zusammengelegt wurden.

Quelle: [02]
Literatur


Hardware Algorithmen

Integer

Multiplikation, Shift und Division

Mai, 2001
MULTIPLIKATION
"PAPIER UND BLEISTIFT METHODE"

\[
\begin{array}{c}
1 & 1 & 0 & 1 \\
\times & 1 & 0 & 1 & 1 \\
\hline
1 & 1 & 0 & 1 \\
1 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 \\
\hline
1 & 0 & 0 & 0 & 1 & 1 & 1 & 1
\end{array}
\]

(13) Multiplikand M
(11) Multiplizierer Q
(143) Produkt P

Quellen:
TYPISCHE ZELLE DER MULTIPLIKATION
FELDIMPLEMENTIERUNG

Bit eines eingehenden partiellen Produktes (PP\(i\))

\[ m_j \]

\[ q_i \]

Carry-out

\[ A \]

Carry-in

Summe

Bit des ausgehenden partiellen Produktes

\[ PP(i+1) \]

\[ 0 \cdot CA \cdot Xi \cdot A \cdot U \cdot T \]

Quellen:
Malek, M.: Vorlesung "Rechnerarchitektur", Sommersemester 2000, HU Berlin,
www.informatik.hu-berlin.de/~rok/ca
Dave Patterson Computer Architecture and Engineering, September 12, 1997, © U.C.B.
www.cs.berkeley.edu/~patterson
REGISTER KONFIGURATION FÜR
SEQUENTIELLE SCHALTUNGEN BINÄRER
MULTIPLIZIERER

Ein Register (initial 0)

C

\( a_{n-1} \ldots a_0 \)

Multiplizierer Q

\( q_{n-1} \ldots q_0 \)

1 \( \rightarrow \) ADD

0 \( \rightarrow \) NO ADD

n-bit
Addie-
rer

 Kontroll-
sequenzer

ADD/NO ADD

0

\( m_{n-1} \ldots m_0 \)

Multiplikand M

Quellen:
Malek, M.: Vorlesung "Rechnerarchitektur", Sommersemester 2000, HU Berlin,
www.informatik.hu-berlin.de/~rok/ca
Dave Patterson Computer Architecture and Engineering, September 12, 1997, © U.C.B.
www.cs.berkeley.edu/~patterson
MULTIPLIKATIONSBEISPIEL

\[
\begin{array}{c}
M \\
1101
\end{array}
\begin{array}{c}
0 \\
0000
\end{array}
\begin{array}{c}
C \\
A
\end{array}
\begin{array}{c}
1011 \\
Q
\end{array}
\]

Ausgangskonfiguration

Erster Durchlauf

Zweiter Durchlauf

Dritter Durchlauf

Vierter Durchlauf

Produkt

PP_0 ist in A nach dem ersten ADD

Quellen:
Malek, M.: Vorlesung "Rechnerarchitektur", Sommersemester 2000, HU Berlin,
www.informatik.hu-berlin.de/~rok/ca
Dave Patterson Computer Architecture and Engineering, September 12, 1997, © U.C.B.
www.cs.berkeley.edu/~patterson
Ununsigned Kombinatorischer Multiplizierer

- Stufe i akkumuliert $A \times 2^i$ wenn $B_i = 1$

- Frage: Wieviel Hardware ist für ein 32 Bit Multiplizierer notwendig? Kritischer Pfad?

Quellen:
Wie funktioniert das Multiplizieren?

◦ In jeder Stufe shift A nach links (x 2)
◦ nutze den nächsten Bit von B zu bestimmen, was dem Multiplikand hinzuaddiert werden soll.
◦ akkumuliere 2n Bit Teilprodukt in jedem Schritt

Quellen:
FLOWCHART FOR 2'S-COMPLEMENT MULTIPLICATION

Start

A ← 0
COUNT ← 0
V ← 0
M ← multiplicand

Q ← multiplier

Yes

Q(7) = 0?

Yes

COUNT ← COUNT + 1

No

A ← A + M

COUNT = 7?

Yes

A(0) ← A(0)
A(1:n-1), Q ← A,Q(0:n-2)

No

A(0) ← M(0) • Q(7)
Q(7) ← 0

A(0) ← M(0) + Q(7)

OUTBUS ← A

OUTBUS ← Q(0:n-1)

Stop

Quellen:
Malek, M.: Vorlesung "Rechnerarchitektur", Sommersemester 2000, HU Berlin,
www.informatik.hu-berlin.de/~rok/ca
Dave Patterson Computer Architecture and Engineering, September 12, 1997, © U.C.B.
www.cs.berkeley.edu/~patterson
BLOCKDIAGRAM OF 2'S-COMPLEMENT FIXED-POINT MULTIPLIER

Quellen:
Malek, M.: Vorlesung "Rechnerarchitektur", Sommersemester 2000, HU Berlin,
www.informatik.hu-berlin.de/~rok/ca
Dave Patterson Computer Architecture and Engineering, September 12, 1997, © U.C.B.
www.cs.berkeley.edu/~patterson
PDP-11 REGISTER ARRANGEMENT
FOR SOFTWARE MULTIPLY

MULTIPLIER Q
R1

C

R0

SHIFT
LEFT

A register (initially 0)

C

R1

MSB

R0

LSB

ASL R0
ROL RI

Long
Shift
Left

R1;R0

R2

MULTIPlicAND M

Quellen:
Malek, M.: Vorlesung "Rechnerarchitektur", Sommersemester 2000, HU Berlin,
www.informatik.hu-berlin.de/~rok/ca
Dave Patterson Computer Architecture and Engineering, September 12, 1997, © U.C.B.
www.cs.berkeley.edu/~patterson
SOFTWARE MULTIPLICATION

CLR R0
Clear register R0

MOV #16,R3
Set register R3 to -16 → use as cycle counter

mloop:

ASL R0
ARithmetic Shift Left instruction shifts contents of R0 (low-order half of partial product) left one bit. MSB of R0 is shifted into C bit. LSB of R0 is set to 0

ROL R1
Rotate Left instruction rotates contents of R1 and the C bit by one bit position. Old contents of C is moved into LSB of R1, old contents of MSB of R1 is moved into C.

The combination of ASL and ROL have thus shifted the (R1, R0) pair left one bit position, placing the old MSB of R1 into C.

BCC noadd
The multiplier bit to be checked (first q_{n-1}, then q_{n-2}, etc.) is now in C, and the instruction BCC causes a branch around the exit operation if C is 0.

ADD R2,R0
If C=1, multiplicand in R2 is added into the low-order end of the partial product in R0. Any carry-out that goes into C must be added to the high-order half of the partial product. This is done by the ADC which adds contents of C to the LSB position of R1. Note that this process will never damage the unused portion of the multiplier in R1 because the very first time that (R1, R0) is shifted left, a = is placed in the LSB position of R1, creating enough space for any later encroachment by the partial product.

noadd:

INC R3
Counter is incremented and BNE (Branch if Not Equal to 0) instruction does branch back to mloop as long 16 cycles are not completed.

Quellen:
Unisigned shift-add Multiplizierer (Version 1)

- 64-bit Multiplikand in reg, 64-Bit ALU, 64-Bit Produkt reg, 32-Bit Multiplikator reg

\[ \text{Multiplikand} \quad \text{64 bits} \]
\[ \text{64-bit ALU} \]
\[ \text{Produkt} \quad \text{64 bits} \]

\[ \text{Shift Left} \]
\[ \text{Shift Right} \]
\[ \text{Write} \]

\[ \text{Control} \]

\[ \text{Multiplikator} \quad \text{32 bits} \]

Quellen:

MUL Algorithmus Version 1

Start

1. Test Multiplikator
   Multiplikator0 = 1
   Multiplikator0 = 0

1a. Add Multiplikand zum Produkt & plaziere Ergebniss in Produkt Register

2. Shift Multiplikand Register left 1 Bit.

3. Shift Multiplizierer Register right 1 Bit.

No: < 32 Wiederh.

32 Wiederh?

Yes: 32 Wiederh.

Stop

Quellen:
Disskussion der MUL Version 1

- 1 Clock per Cycle => - 100 clocks per MUL
  - Ratio of multiply to add 5:1 to 100:1

- 1/2 Bits in Multiplikand sind immer “0”
  => 64-Bit Adder wird nicht voll genutzt

- Nullen werden von Links beim Shiften dem Multiplikand zugefügt.
  => Least Significant Bits des Produktes ändert sich nicht.

- Statt den Multiplikand nach links zu shiften, shift Produkt nach rechts?
MUL HARDWARE Version 2

° 32-Bit Multiplikand reg, 32-Bit ALU, 64-Bit Produkt reg, 32-Bit Multiplikator reg
MUL Algorithmus Version 2

1. Test Multiplier0
   - Multiplier0 = 1
   - Multiplier0 = 0

2. Shift the Product register right 1 bit.

3. Shift the Multiplier register right 1 bit.

32nd repetition?
   - No: < 32 repetitions
   - Yes: 32 repetitions

Start

Product Multiplier Multiplikand
0000 0000 0011 0010

Source: Dave Patterson, Computer Architecture and Engineering © U.C.B.

Quellen:
Malek, M.: Vorlesung "Rechnerarchitektur", Sommersemester 2000, HU Berlin,
www.informatik.hu-berlin.de/~rok/ca
Dave Patterson Computer Architecture and Engineering, September 12, 1997, © U.C.B.
www.cs.berkeley.edu/~patterson
Wie wird multipliziert?

Der Multiplikand bleibt am Platz, das Produkt bewegt sich nach rechts.

Quellen:
Malek, M.: Vorlesung "Rechnerarchitektur", Sommersemester 2000, HU Berlin,
www.informatik.hu-berlin.de/~rok/ca
Dave Patterson Computer Architecture and Engineering, September 12, 1997, © U.C.B.
www.cs.berkeley.edu/~patterson
MUL Algorithmus Version 2

1. Test Multiplier0
   - Multiplier0 = 1
   - Multiplier0 = 0

1a. Add multiplicand to the left half of product & place the result in the left half of Product register

- Produkt Multiplikator Multiplikand
  - 0000 0011 0010
  - 0010 0000
  - 0001 0000 0010
  - 0011 00 0010
  - 0001 1000 0010
  - 0000 1100 0010
  - 0000 0110 0010

2. Shift the Product register right 1 bit.
3. Shift the Multiplier register right 1 bit.

32nd repetition?
- No: < 32 repetitions
- Yes: 32 repetitions

Quellen:
MUL HARDWARE Version 3

- 32-Bit Multiplikand reg, 32-Bit ALU, 64-Bit Produkt reg, (64-Bit Multiplikator reg)

Quellen:
**MUL Algorithmus Version 3**

**Multiplikand**  
0010

**Produkt**  
0000 0011

1. Test **Product0**

1a. Add multiplicand to the left half of product & place the result in the left half of Product register

2. Shift the Product register right 1 bit.

32nd repetition?

No: < 32 repetitions

Yes: 32 repetitions

Done

---

**Quellen:**
Malek, M.: Vorlesung "Rechnerarchitektur", Sommersemester 2000, HU Berlin,  
[www.informatik.hu-berlin.de/~rok/ca](http://www.informatik.hu-berlin.de/~rok/ca)  
Dave Patterson Computer Architecture and Engineering, September 12, 1997, © U.C.B.  
[www.cs.berkeley.edu/~patterson](http://www.cs.berkeley.edu/~patterson)
Multiplizierer

Multiplikationsbeispiel:

Multiplikand: $A_3A_2A_1A_0 = 1011$ (~ 11)

1011 1 $B_0$
0000 0 $B_1$
0000 0 $B_2$
1011 1 $B_3$

01100011 (~ 99)

Quellen: Rechnerarchitektur I, II, III, Fachgebiet Rechnersysteme, TU Darmstadt,
http://www.rs.e-technik.tu-darmstadt.de/TUD/lectures/lectures.html
4-bit Multiplizierer:
Aufbau jeder Zelle:
Durchlaufzeit: \ (~3n-2)}
Durchlaufzeit: \( \sim 3n-2 \)
Multiplizierer mit Carry-Save Addieren

- Prinzip: die in einem Partialprodukt entstehenden Überträge werden an die Berechnung des nächsten Partialprodukts weitergereicht.

- Beispiel für eine Addition mit Carry-Save Addieren:

\[
\begin{array}{cccc}
1 & 0 & 1 & 1 \\
+ & 0 & 1 & 1 \\
\hline
1 & 1 & 0 & 0 \\
\end{array}
\]

Summe: 1 1 0 0 \(\sim 12\)

Übertrag: 0 0 1 1 \(\sim 6\)

1 0 0 1 0

die Überträge werden nicht an die folgende Stelle weitergereicht. Nochmalige Addition mit Übertrag ist zur Ermittlung des richtigen Resultats notwendig.
- multiplizieren nach dem Carry-Save Prinzip:

Multiplikand

\[ A_3 A_2 A_1 A_0 = 1 \ 0 \ 1 \ 1 \ \sim \ 1 1 \]

Multiplikator

\[ \sim 1 1 \]

1. Summe:
   \[ \begin{array}{c}
   1 0 1 1 \\
   0 0 0 0 \\
   + 1 0 1 1 \\
   \end{array} \]

   \[ 1 \ B_0 \]

1. Übertrag:

2. Summe:
   \[ \begin{array}{c}
   1 0 1 1 \\
   0 0 0 1 \\
   + 0 0 0 0 \\
   \end{array} \]

   \[ 1 \ 1 \ 1 0 \]

2. Übertrag:

3. Summe:
   \[ \begin{array}{c}
   1 1 \ 1 0 \\
   0 0 0 1 \\
   + 1 0 1 1 \\
   \end{array} \]

   \[ 0 \ B_2 \]

3. Übertrag:

4. Summe:
   \[ \begin{array}{c}
   1 0 0 1 \\
   0 0 1 1 \\
   \end{array} \]

Endaddition

Ripple-Carry:
\[ 0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 0 \ 0 \ 1 \ \sim \ 1 2 1 \]
Schaltung:

Durchlaufzeit bei Ripple-Carry Addierer in der untersten Ebene: \( \sim 2n \)
Durchlaufzeit bei Ripple-Carry Addierer in der untersten Ebene: \( \sim 2n \)
Multiplikation mit einer Konstanten

> "#" bedeutet Vektor-Konkatenation, z.B. 0 # 1 = 01

\[ a \times 64 + a \times 2 \]

\[ a \cdot 01111110 = a \cdot (10000000 \cdot 00000010) \]

ALGORITHMEN II.

6. Integer Division

Figure 4.36 Version 1. der Divisionsschaltung. Divisor-Register, ALU und Rest-Register sind 64 Bit breit, nur der Quotient-Register ist 32 Bit breit. Der 32-Bit Divisor liegt am Anfang der Division in der linken Hälfte des Divisor-Registers und wird in jedem Schritt um 1 Bit nach rechts verschoben. Der Rest wird mit dem Dividend initialisiert. Der Control-Block entscheidet, wann die Inhalte der Divisor- und Quotient-Register verschoben werden sollen und wann neue Werte im Rest-Register geschrieben werden sollen.

Quelle: [02]
Figure 4.37 Version 1. des Divisionsalgorithmus nach Figure 4.36. Wenn der Rest positiv ist, so läßt sich der Dividend durch den Divisor teilen (Quotient > 1), so ergibt Schritt 2a eine 1 im Quotient. Ein negativer Rest nach diesem Schritt bedeutet, daß sich der Dividend durch den Divisor nicht teilen läßt (Quotient > 0), so ergibt Schritt 2b eine 0 im Quotient und addiert den Quotient zum Rest, in dem die Subtraktion im Schritt 1 umgekehrt wird. Die Verschiebung um 1 Bit im Schritt 3. Bereitet den Divisor für den nächsten Schritt. Diese Schritte werden 33 mal für 32-Bit Operanden wiederholt.

Quelle: [02]

Prof. Dr. V. Iossifov, TI, FHTW Berlin
Figure 4.39 Version 2. der Divisionsschaltung. Divisor-Register, ALU und Quotient-Register sind 32 Bit breit, nur der Rest-Register ist 64 Bit breit.

Quelle: [02]
Figure 4.39a Version 2. des Divisionsalgorithmus nach Figure 4.39. Im Unterschied zum Algorithmus vom Fig. 4.34 ändert sich nur die linke Hälfte des Restes und der Rest wird an Stelle des Divisors nach rechts verschoben.

Quelle: [02]
Figure 4.40 Version 3. des Divisionsalgorithmus nach Figure 4.38. Im Unterschied zum Algorithmus vom Fig. 4.35 wird der Rest-Register nach links verschoben, so werden Schritte 1. Und 3. Aus Fig. 4.35 kombiniert.

Quelle: [02]
Figure 4.41 Version 3. der Divisionsschaltung. Die Version kombiniert den Quotient-Register mit der rechten Hälfte des Resr-Registers.

Quelle: [02]
Literatur


Hardware Algorithmen

Integer
Multiplikation, Shift und Division

Mai, 2001
Dividieren: Papier und Bleistift

\[
\begin{array}{c|c|c}
\text{Divisor} & 1001010 & \text{Dividend} \\
\hline
1000 & 10 & 101 \\
-1000 & 1010 & -1000 \\
\hline
& 10 & 10 \\
\end{array}
\]

Quotient

Rest (oder Modulo Ergebnis)

In Abhängigkeit davon ob die SUB einen >0 oder <0 Ergebnis liefert, wird ein Quotient Bit in jedem Schritt erzeugt:

Binary => 1 * Divisor oder 0 * Divisor

Dividend = Quotient x Divisor + Remainder

=> | Dividend | = | Quotient | + | Divisor |

=> 3 Versionen der Division.

Quellen:
Malek, M.: Vorlesung "Rechnerarchitektur", Sommersemester 2000, HU Berlin,
www.informatik.hu-berlin.de/~rok/ca
Dave Patterson Computer Architecture and Engineering, September 12, 1997, © U.C.B.
DIV HARDWARE Version 1.
° 64-bit Divisor reg, 64-bit ALU, 64-bit Rest reg, 32-bit Quotient reg

Quellen:
Diskussion der integer Division Version 1.

° 1/2 Bits im Divisor immer 0  
   => 1/2 der 64-Bit Adder ist umsonst  
   => 1/2 des Divisor ist umsonst

° Statt den Divisor nach rechts verschieben,  
   sollte der Rest nach links verschoben werden?

° 1er Schritt kann keine "1" im Quotient Bit erzeugen  
   (otherwise too big)  
   => zuerst sollte shift ausgeführt werden und dann  
   subtract,  
   so kann eine Iteration gespart werden.
DIV HARDWARE Version 2:

° 32-Bit Divisor reg, 32-Bit ALU, 64-bit Rest reg, 32-Bit Quotient reg

Quellen:
Dave Patterson Computer Architecture and Engineering, September 12, 1997, © U.C.B.
DIV Algorithm Version 2

1. Shift the Remainder register left 1 bit.

2. Subtract the Divisor register from the left half of the Remainder register, & place the result in the left half of the Remainder register.

3a. Shift the Quotient register to the left setting the new rightmost bit to 1.

3b. Restore the original value by adding the Divisor register to the left half of the Remainder register, & place the sum in the left half of the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0.

 cheating

remainder >= 0

Test Remainder

remainder < 0

8 repetition?

no: < n repetitions

yes: n repetitions (n = 4 here)

Done

Source: Dave Patterson, Computer Architecture and Engineering © U.C.B.
Observations on Divide Version 2

° Eliminieren des Quotientregisters durch kombinieren mit Rest as shifted left
  • Start by shifting the Remainder left as before.
  • Thereafter loop contains only two steps because the shifting of the Remainder register shifts both the remainder in the left half and the quotient in the right half.
  • The consequence of combining the two registers together and the new order of the operations in the loop is that the remainder will shifted left one time too many.
  • Thus the final correction step must shift back only the remainder in the left half of the register.
DIV HARDWARE Version 3

° 32-Bit Divisor reg, 32-Bit ALU, 64-Bit Rest reg, (0-Bit Quotient reg)
DIV Algorithm Version 3

Start: Place Dividend in Remainder

1. Shift the Remainder register left 1 bit

2. Subtract the Divisor register from the left half of the Remainder register, & place the result in the left half of the Remainder register.

3a. Shift the Remainder register to the left setting the new rightmost bit to 1.

3b. Restore the original value by adding the Divisor register to the left half of the Remainder register, & place the sum in the left half of the Remainder register. Also shift the Remainder register to the left, setting the new least significant bit to 0.

Remainder \( \geq 0 \)

Test

Remainder \( < 0 \)

repetition?

No: \( \leq n \) repetitions

Yes: \( n \) repetitions (\( n = 4 \) here)

Done

Source: Dave Patterson, Computer Architecture and Engineering © U.C.B.

Quellen:
Malek, M.: Vorlesung "Rechnerarchitektur", Sommersemester 2000, HU Berlin,
www.informatik.hu-berlin.de/~rok/ca
Dave Patterson Computer Architecture and Engineering, September 12, 1997, © U.C.B.
www.cs.berkeley.edu/~patterson
Observations on Divide Version 3

- Same Hardware as Multiply: just need ALU to add or subtract, and 63-bit register to shift left or shift right

- Hi and Lo registers in MIPS combine to act as 64-bit register for multiply and divide

- Signed Divides: Simplest is to remember signs, make positive, and complement quotient and remainder if necessary
  - Note: Dividend and Remainder must have same sign
  - Note: Quotient negated if Divisor sign & Dividend sign disagree e.g., \(-7 + 2 = -3\), remainder = \(-1\)

- Possible for quotient to be too large: if divide 64-bit integer by 1, quotient is 64 bits (“called saturation”)

Quellen:
Dave Patterson Computer Architecture and Engineering, September 12, 1997, © U.C.B.
www.cs.berkeley.edu/~patterson
ALGORITHMEN II.

7. Fließkomma Arithmetik

Quellen:
Malek, M.: Vorlesung "Rechnerarchitektur", Sommersemester 2000, HU Berlin,
www.informatik.hu-berlin.de/~rok/ca
Dave Patterson Computer Architecture and Engineering, September 12, 1997, © U.C.B.
www.cs.berkeley.edu/~patterson
FLIEßKOMMA - ARITHMETIK
wissenschaftliche Notation (1)

KOMPONENTE DER DARSTELLUNG VON FLIEßKOMMAZahlen

- Vorzeichen
- Exponent \((X_e)\) \((Y_e)\)
- Mantisse \((X_m)\) \((Y_m)\)
\[ X = X_m \times B^{X_e} \]

Beispiele:
\[ +1.23 \times 10^2 \]
\[ 9.999\,999 \times 10^9 \]

Quellen:
Malek, M.: Vorlesung "Rechnerarchitektur", Sommersemester 2000, HU Berlin,
www.informatik.hu-berlin.de/~rok/ca
Dave Patterson Computer Architecture and Engineering, September 12, 1997, © U.C.B.
www.cs.berkeley.edu/~patterson
FLIEßKOMMA - ARITHMETIK
wissenschaftliche Notation (2)

a. **Vorzeichen** wird als eine Erweiterung der Mantisse behandelt.

b. **Exponent**
   1. **Scalierungsfactor (B) Basis oder Radix**
   2. **Verschiebung**
      Die Exponentwerte sind normalerweise um einen bestimmten Wert verschoben. Wenn wir z. B. ein Exponent-Feld haben, das Werte zwischen 00 und 99 annehmen kann, würden wir, um positive und negative Werte darzustellen um den Wert 50 verschieben. Der Exponent -50 würde dann durch den Wert 0 repräsentiert, Exponent 0 durch 50, und Exponent 49 durch 99.

      \[
      0 \leq E_B \leq 99 \text{ Exponent-Feld} \\
      -50 \leq E \leq 49 \text{ dargestellter Exponent}
      \]

Der Grund, diese Technik zu nutzen liegt darin, daß die Größenordnung der Zahlen verglichen werden kann, ohne darauf achten zu müssen, ob die Zahl im Fließkommaformat dargestellt ist oder nicht, z.B. können die Größenordnungen in der gleichen Weise wie in der Ganzzahlanrithmetik verglichen werden.

**FP 2**

**Quellen:**
FORMATE DER FLIEßKOMMAZAHLEN

a) Das 32-Bit binäre Format

<table>
<thead>
<tr>
<th>E</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vorzeichen</td>
<td>7-Bit vorzeichen-vehafelter Exponent</td>
</tr>
</tbody>
</table>

32 Bit

b) Binäre Normalisation - ein Beispiel

<table>
<thead>
<tr>
<th>0</th>
<th>0001001</th>
<th>001.......</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0.001...x2^3</td>
<td>Nichtnormalisierter Wert</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>0000111</th>
<th>1.......</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0.1...x2^7</td>
<td>Normalisierter Wert</td>
<td></td>
</tr>
</tbody>
</table>

Normalisation hält die Mantisse beim Wert von

\[ \frac{1}{2} < M < 1 \quad -64 \leq E \leq 63 \]

Excess - 64 Format \( E' = E + 64 \quad 0 \leq E' \leq 127 \)

c) Hexadezimale (Basis 16) Normalisation

Beispiel mit Excess 64-Exponenten (Bias 64)

<table>
<thead>
<tr>
<th>1</th>
<th>0000011</th>
<th>0000101.......</th>
</tr>
</thead>
<tbody>
<tr>
<td>-0.00000101...x16^3</td>
<td>Nichtnormalisierter Wert</td>
<td></td>
</tr>
</tbody>
</table>

Ausnahme: Unterlauf \( E \leq 64 \) Überlauf \( E > 63 \) - setze auf 0

<table>
<thead>
<tr>
<th>1</th>
<th>0000010</th>
<th>0101.......</th>
</tr>
</thead>
<tbody>
<tr>
<td>-0.0101...x16^2</td>
<td>Normalisierter Wert</td>
<td></td>
</tr>
</tbody>
</table>

Quellen:
DAS IEEE STANDARTFORMAT

Wert = (-1)S x (1 + M1 x 2⁻¹ + M2 x 2⁻² + ... + M23 x 2⁻²²) x 2E7..E10-127
(kurze reelle Zahl)
Wert = (-1)S x (1 + M1 x 2⁻¹ + M2 x 2⁻² + ... + M51 x 2⁻52) x 2E10..E14-1023
(lange reelle Zahl)
Wert = (-1)S x (M3 + M1 x 2⁻¹ + M2 x 2⁻² + ... + M63 x 2⁻63) x 2E14..E16-16383
(temporäre reelle Zahl)

• Kurze reelle Zahl (32 bits)
  - Rang vom Wert 1.18 x 10⁻³⁸ < |x| < 3.40 x 10⁻³⁸, Präzision 24 Bit
  31 30 29 28 27 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2
  | S | E₁₀...E₀ | M₁ | .......... | M₂³ |
  - S: sign bit (1= negative Mantisse, 0= positive Mantisse)
  - E7..E0: exponent (8 bits, bias 127)
  - M₁..M₂³: Mantisse (23 bits plus implizit M₀=1)

• Lange reelle Zahl (64 Bit)
  - Rang vom Wert 2.23 x 10⁻³⁰⁸ < |x| < 1.79 x 10⁺³⁰⁸, Präzision 53 Bit
  63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40
  | S | E₁⁰...E₀ | M₁ | .......... | M₅₂ |
  - Verschiebung 1023

• Temporäre reelle Zahl (80 Bit)
  - Rang vom Wert 3.37 x 10⁻⁴⁹³² < |x| < 1.79 x 10⁺⁴⁹³², Präzision 64 Bit
  79 78 77 76 75 74 73 72 71 70 69 68 67 66 65 64 63 62 61 60 59 58 57 56
  | S | E₁⁴...E₀ | M₁ | .......... | M₆₁ |
  - Verschiebung 16.38: FP 4

Quellen:
Malek, M.: Vorlesung "Rechnerarchitektur", Sommersemester 2000, HU Berlin,
www.informatik.hu-berlin.de/~rok/ca
Dave Patterson Computer Architecture and Engineering, September 12, 1997, © U.C.B.
www.cs.berkeley.edu/~patterson
**FLIEßKOMMA DARSTELLUNG**

**IEEE STANDART**

IEEE Standard (einstufige Präzision)

\[ (-1)^S \cdot (1.M) \cdot 2^E \text{ Verschiebung} \quad (\text{z.B., Verschiebung} = 127) \]

<table>
<thead>
<tr>
<th>S</th>
<th>8-Bit Exponent</th>
<th>23-Bit Mantisse</th>
</tr>
</thead>
</table>

Mantisse mit eingesetzter 1 24 Bits \( 1 \leq M' \leq 2 \)
Größtes Fehler \( 2^{-24} \)
Präzision \( \sim 7 \text{ dezimale Einheiten} \)
Verschiebung 127
Exponentenrang \(-126 \leq E' \leq 127 \)
Kleinste Zahl \( 2^{-128} = 1.2 \times 10^{-38} \)
Größte Zahl \((2 - 2^{-23}) \cdot 2^{127} = 3.4 \times 10^{38} \)

**Sonderfälle**

a) Null \( (-1)^S \cdot 0 \) \( E=0, M=0 \)
b) Anfangs \( (-1)^S \infty \) \( E=255, M=0 \)
c) Nicht Zahl \( E=255, M=0 \)
d) Normalisiert \( (-1)^S \cdot (1.M) \cdot 2^{E-127} \) \( 0 < E < 255 \)
e) Nichtnormalisiert \( (-1)^S \cdot (0.M) \cdot 2^{-128} \) \( E=1, M=0 \)

*(Diese Zahl kann nicht normalisiert werden, weil die Linksverschiebung einen Unterlauf verursachen würde)*

**Beispiele**

<table>
<thead>
<tr>
<th>S</th>
<th>Exponent</th>
<th>Mantisse mit 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10000001</td>
<td>1.00...0 ( = 1.0 \times 2^{129-127} = 4.0 )</td>
</tr>
<tr>
<td>0</td>
<td>01111111</td>
<td>1.10...0 ( = 1.5 \times 2^{127-127} = 1.5 )</td>
</tr>
<tr>
<td>0</td>
<td>10000000</td>
<td>1.010...0 ( = 1.25 \times 2^{128-127} = 2.5 )</td>
</tr>
</tbody>
</table>

**Quellen:**
UNTERLAUF UND ÜBERLAUF

- Ein Unterlauf tritt auf, wenn der resultierende verschobene Exponent kleiner als Null (0) ist. In diesem Fall wird das gesamte Fließkommawort auf Null gesetzt.
- Ein Unterlauf tritt auf, wenn der resultierende verschobene Exponent größer ist, als der für das Exponent-Feld erlaubte Maximalwert.
- In beiden Fällen wird normalerweise ein error-Bit im Status-Word gesetzt oder ein Interrupt ausgelöst, so daß ein Programm entscheiden kann, welche Aktion auf einen Unter- oder Überlauf folgen soll.

Quellen:
NORMALISIERUNG UND SCALIERUNG

- Normalisierung ist ein Prozess von Sicherstellung der maximaler Genauigkeit einer Fließkommazahl
- Die Normalisierung und das Ändern des Skalierungsfaktors funktioniert wie die "pen and pencil"-Methode in der Wissenschaft.

Beispiele:
\[
\begin{align*}
1.27 \times 10^5 &= .127 \times 10^6 \\
0.03 \times 10^2 &= .30 \times 10^1 \\
42.1 \times 10^{-6} &= .421 \times 10^4 \\
0.022 \times 10^{-4} &= .220 \times 10^{-5}
\end{align*}
\]

- Für eine binäre Verschiebung braucht man eine unäre (schrittweise), während für eine hexadezimale Verschiebung um 4 Positionen notwendig ist (Wurzel 16)

Quellen:
**SCHWIERIGKEITEN UND RUNDUNGEN**

Arithmetik

a. Multiplikation und Division sind nicht so kompliziert wie Addition oder Subtraktion.

b. Die Addition oder Subtraktion von Zahl ist schwierig, weil kleine Zahlen müssen verschoben werden, um an die größeren Zahlen angepasst zu werden, bevor die arithmetische Operation ausgeführt ist.

Runden

Gebrochene binäre Arithmetik kann aufgrund der begrenzten Bit-Präsentation zu fehlerhaften Ergebnissen führen. **Guard bits** oder erweiterte Präzision (größere Register) Bit werden oft bei der Präzisierungshöhung gebraucht. Das Ergebnis wird nach der Operation gerundet.

<table>
<thead>
<tr>
<th>Rundungsmethoden</th>
<th>Ergebnisse</th>
<th>Guard Bits</th>
<th>Kommentare</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chopping (Abschneiden)</td>
<td>.0010</td>
<td>110</td>
<td>Verschiebungsfehler</td>
</tr>
<tr>
<td>Das Ergebnis wird nicht gerundet.</td>
<td>.0010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Normale Rundung</td>
<td>.0010</td>
<td>110</td>
<td>Verschiebungsfehler</td>
</tr>
<tr>
<td>1wird zum Ergebniss addiert</td>
<td>.0011]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSB wenn das Ergebnis von</td>
<td>.0011</td>
<td>110</td>
<td>-½ to +½ (LSB)</td>
</tr>
<tr>
<td>einem Guard Bit gefolgt wird.</td>
<td>.0100]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| Von Neumann                            | .0010      | 110        | Komplexere Umsetzung, Verschie-
| 1Bit wird dem Ergebnis hinzufügt,     | .0011]     |            | bungsfehler:                |
| LSB wenn dem Ergebnis 1 Guard Bit folgt.| .0011]     |            | -1 to +1 (LSB)              |

**Quellen:**
Dave Patterson Computer Architecture and Engineering, September 12, 1997, © U.C.B.
www.cs.berkeley.edu/~patterson
GLEICHUNGEN FÜR FLIEßKOMMA OPERATIONEN

\( X_E, Y_E \) - Exponenten von \( X \) und \( Y \)
\( X_M, Y_M \) - Mantisse von \( X \) und \( Y \)

\[
\text{ADD} \quad X + Y = (X_M \times 2^{X_E-Y_E} + Y_M) \times 2
\]

\[
\text{SUB} \quad X - Y = (X_M \times 2^{X_E-Y_E} - Y_M) \times 2
\]

\[
\text{MULT} \quad X \times Y = (X_M \times Y_M) \times 2
\]

\[
\text{DIV} \quad X \div Y = (X_M \times Y_M) \times 2
\]

Quellen:
Malek, M.: Vorlesung "Rechnerarchitektur", Sommersemester 2000, HU Berlin,
www.informatik.hu-berlin.de/~rok/ca
Dave Patterson Computer Architecture and Engineering, September 12, 1997, © U.C.B.
www.cs.berkeley.edu/~patterson
FLIEßKOMMAARITHMETIK
ALGORITHMUS

ADD/SUB  1) AUSWAHL DER ZAHL MIT DEM
KLEINEREN EXPONENTEN UND
VERSCHIEBUNG SEINER MANTISSE
SCHRITTWEISE NACH RECHTS
ENTSPRECHEND DER DIFFERENZ DER
EXPONENTEN.

2) SETZE DEN EXPONENTEN DES
ERGEBNISSES GLEICH DEM
GRÖßEREN EXPONENTEN.

3) FÜHRE DIE ADDITION-SUBTRAKTION
DER MANTISSEN AUS UND
ENTSCHEIDE DAS VORZEICHEN DES
ERGEBNISSES.

4) NORMALISIERE DAS ERGEBNIS
(FALLS NÖTIG).

5) PRÜFE ÜBERLAUF/UNTERLAUF

Quellen:
Malek, M.: Vorlesung "Rechnerarchitektur", Sommersemester 2000, HU Berlin,
www.informatik.hu-berlin.de/~rok/ca
Dave Patterson Computer Architecture and Engineering, September 12, 1997, © U.C.B.
www.cs.berkeley.edu/~patterson
FLIEßKOMMA-MULTIPLIKATION UND DIVISION

MULT
1) ADDIERE EXONENTEN

2) MULTIPLIZIERE MANTISSEN UND ENTSCHEIDE VORZEICHEN DES ERGEBNISSES

3) NORMALISIERE DEN RESULTIERENDEN WERT (FALLS NÖTIG)

4) PRÜFE ÜBERLAUF/UNTERLAUF

DIV
1) SUBTRAHIERE EXONENTEN

2) DIVIDIERE MANTISSEN & ENTSCHEIDE VORZEICHEN DES ERGEBNISSES

3) NORMALISIERE DEN RESULTIERENDEN WERT (FALLS NÖTIG)

4) PRÜFE ÜBERLAUF/UNTERLAUF

ÜBERLAUF \( \text{EXP} > + \text{BEREICH} \)
UNTTERLAUF \( \text{EXP} < - \text{BEREICH} \)

Quellen:
Malek, M.: Vorlesung "Rechnerarchitektur", Sommersemester 2000, HU Berlin,
www.informatik.hu-berlin.de/~rok/ca
Dave Patterson Computer Architecture and Engineering, September 12, 1997, © U.C.B.
www.cs.berkeley.edu/~patterson
**FLIEßKOMMA-ARITHMETIK-SCHEMA AUS ZWEI ELEMENTEN DER FESTKOMMA-ARITHMETIK**

Datenbus

Exponenteneinheit → Mantisseneinheit

Steuerbus

Steuerungseinheit

**Datenverarbeitender Teil einer einfachen Einheit zur Fließkomma-Arithmetik**

Exponenteneinheit:

- E
- E1
- E2

Mantisseneinheit:

- AC
- MQ
- DR

Addierer

Quellen:
Malek, M.: Vorlesung "Rechnerarchitektur", Sommersemester 2000, HU Berlin,
[www.informatik.hu-berlin.de/~rok/ca](http://www.informatik.hu-berlin.de/~rok/ca)
Dave Patterson Computer Architecture and Engineering, September 12, 1997, © U.C.B.
[www.cs.berkeley.edu/~patterson](http://www.cs.berkeley.edu/~patterson)
EIN KOMBINIERTES SCHEMA ZUR FLIEßKOMMA-ARITHMETIK

Quellen:
Malek, M.: Vorlesung "Rechnerarchitektur", Sommersemester 2000, HU Berlin,
www.informatik.hu-berlin.de/~rok/ca
Dave Patterson Computer Architecture and Engineering, September 12, 1997, © U.C.B.
www.cs.berkeley.edu/~patterson
Fließkommaaddierer des IBM mainframe

Quellen:
Fließkomma Einheit Für Addition/Subtraktion

32-Bit Operanden \( \{ A: S_A, E_A, M_A \} \)

\( B: S_B, E_B, M_B \)

- \( E_A \)
- \( E_B \)

Vorzeichen

7-Bit Subtrahierer

\( S_A, S_B \)

Steuerwerk

7-Bit Addierer/Subtrahierer

SWAP

\( M_A \)

\( M_B \)

M der Zahl mit kleinerem E

M der Zahl mit größerem E

\( n = |E_A - E_B| \)

Shifter

n hex Zeichen nach rechts

7-Bit Addierer/Subtrahierer

Normalisierung und Runden

\( E \)

Test auf führende hex-Nullen

7-Bit Subtrahierer

\( E - X \)

\( S_R \)

\( E_R \)

\( M_R \)

32-Bit Ergebnis

\( R = A \pm B \)

Quellen:
FLIEßKOMMA-ADDITION-SUBTRAKTION
ALGORITHMUS

Lade input Operanden

Vergleiche Exponenten

Angleichen der Exponenten

Addiere-Subtrahiere Mantissen

Start

E1 ← XE
AC ← XM

E2 ← YE
DR ← YM

E ← E1 - E2

E < 0 ?
Ja
Rechts-shift AC
E ← E + 1

Nein

E > 0 ?
Ja
Rechts-shift DR
E ← E + 1

Nein

Addiere Befehl?
Ja

AC ← AC + DR
E ← max (E1, E2)

Nein

AC ← AC - DR
E ← max (E1, E2)

1. Vergleich der Exponenten durch Subtraktion
2. Angleichung der Mantissen durch Verschieben
3. Addition oder Subtraktion der Mantissen
4. Normalisieren der Ergebnisse
Teste auf den Überlauf und auf die Null-Mantisse

FP 16

Quellen:
Malek, M.: Vorlesung "Rechnerarchitektur", Sommersemester 2000, HU Berlin,
www.informatik.hu-berlin.de/~rok/ca
Dave Patterson Computer Architecture and Engineering, September 12, 1997, © U.C.B.
www.cs.berkeley.edu/~patterson
PIPELINED-VERSION DES FLIEßKOMMA ADDIERERS

Input bus

Exponentenvergleich

Normalisierung

Mantissensaddition

Normalisierung

Output bus

Quellen:
Malek, M.: Vorlesung "Rechnerarchitektur", Sommersemester 2000, HU Berlin,
www.informatik.hu-berlin.de/~rok/ca
Dave Patterson Computer Architecture and Engineering, September 12, 1997, © U.C.B.
www.cs.berkeley.edu/~patterson
ALGORITHMen II.

8. Fließkomma Addition

Fließkomma Addition

1. Compare the exponents of the two numbers. Shift the smaller number to the right until its exponent would match the larger exponent.

2. Add the significands.

3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent.

4. Round the significand to the appropriate number of bits.

Overflow or underflow? Yes

No

Still normalized? Yes

No

Exception

Done

Figure 4.44 Fließkomma Addition.

Quelle: [02]
Figure 4.45 Blockdiagramm einer ALU für Fließkomma Addition. Die Schritte aus fig. 4.44 korrespondieren zu jedem Block von oben nach unten gesehen. Zuerst wird der Exponent des einen Operanden vom Exponenten des anderen Operanden im "Small ALU" subtrahiert um herauszufinden welcher Exponent größer ist und um wieviel. Die Differenz der Exponenten steuert die drei Multiplexer an. Diese selektieren den größeren Exponenten, die Mantisse der kleineren und die Mantisse der größeren Zahl. Die kleinere Mantisse wird nach rechts verschoben und die beiden Mantissen werden im "BIG ALU" addiert. Der Normalisierungsschritt verschiebt die Summe nach links oder nach rechts und inkrementiert oder dekrementiert den Exponenten. Die Rundung bringt das Ergebnis im gewünschten Datenformat (float, double float).

Quelle: [02]
Literatur


ALGORITHMEN II.

9. Fließkomma Multiplikation

Algorithmus

1. Add the biased exponents of the two numbers, subtracting the bias from the sum to get the new biased exponent

2. Multiply the significands

3. Normalize the product if necessary, shifting it right and incrementing the exponent

4. Round the significand to the appropriate number of bits

5. Set the sign of the product to positive if the signs of the original operands are the same; if they differ make the sign negative

Done

Figure 4.46 Fließkomma Multiplikation.

Quelle: [01]

Prof. Dr. V. Iossifov, TI, FHTW Berlin
Literatur

ALGORITHMEN II.

9. Fließkomma Multiplikation

Runden

a) $1.23 \times 0.76 = 0.93394$  
   $r = 3 < 5$ so round down  
   rounds to 0.93

b) $2.83 \times 4.47 = 12.6501$  
   $r = 5$ and a following digit + 0 so round up  
   rounds to $1.27 \times 10^1$

c) $1.28 \times 6.81 = 0.99968$  
   $r = 9 > 5$ so round up  
   rounds to $1.00 \times 10^1$

**FIGURE A.9** Examples of rounding a multiplication.

*Quelle: [01]*
FIGURE A.10 The two cases of the floating-point multiply algorithm.

Quelle: [01]
Literatur

ALGORITHMEN II.

11. Fließkomma Addition und Multiplikation

Übungsaufgaben
Gleitkommazahlen

\[ \pm 0,111101 \times 2^{010011} \]

\[ \text{Vorzeichen} \quad \text{Exponent} \quad \text{Mantisse} \quad \text{Basis} \]

- Eindeutigkeit der Darstellung durch normalisierte Mantisse \( \Rightarrow \) erstes Bit muß nicht repräsentiert werden ("hidden bit")

Quelle: [01]
Gleitkommazahl nach IEEE 754

- hier: einfache Genauigkeit
  - 32 Bit
  - 1 Vorzeichenstelle: $s = 0$ positiv, $s = 1$ negativ
  - 8 Stellen für den Exponenten
  - 23 Stellen für die Mantisse $1 \leq m < 2$, d.h. $1,..., \text{plus eine versteckte Stelle (hidden bit)}$

\[
\begin{array}{cccccc}
31 & 30 & 23 & 22 & 0 \\
| s & e_7 & e_0 & f_{22} & f_0 |
\end{array}
\]

Quelle: [01]
Gleitkommazahl nach IEEE 754

- Exponent in versetzter Darstellung (biased, exponent, Charakteristik)

- der tatsächliche Exponent entsteht aus dem dargestellten durch die Subtraktion von $127_{10} = (2^7 - 1)_{10} = 01111111_2$

- "regulärer" Exponentenbereich
  - $-126 \leq E \leq 127$
  - und $-127$ wenn Mantisse null ist (Darstellung ± 0)

Quelle: [01]
Übungsaufgaben

1. Aufgabe
Welche Dezimalzahl wird durch das folgende 32-Bit Wort, interpretiert als IEEE 754 Gleitkommazahl, dargestellt?

1 1 0 0 0 0 0 0 1 0 1 0 0 0 ... 0 0

2. Aufgabe
Gegeben seien die Gleitkommazahlen
A = 1,000 · 2⁻¹
und
B = 1,110 · 2⁻².
Führen Sie die Operationen
A+B,
-A-B,
A · B
aus. Das Resultat soll jeweils wieder auf vier Stellen gerundet werden.

3. Aufgabe
Das Assoziativgesetz
\[ x + (y + z) = (x + y) + z \]
gilt nicht generell innerhalb der Bereichsgrenzen für Gleitkommazahlen. Geben Sie ein einfaches Beispiel an, das diese Aussage belegt.

Literatur

www.tu-darmstadt.de/vwvs00-01/ws_00-01_FB20.tud

Vorgetragen von Prof. Dr. V. Iossifov, TI, FHTW Berlin
1. Aufgabe

\[
\begin{array}{c|c|c|c|c}
31 & 30 & 23 & 22 & 0 \\
\hline
1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & \ldots & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

\( s \quad e_7 \quad e_0 \quad f_{22} \quad f_0 \)

- \( s = 1 \Rightarrow \) negatives Vorzeichen

- \( \text{Biased exponent} \quad 1 0 0 0 0 0 0 1_2 \quad +129_{10} \)
  
  \[
  - 0 1 1 1 1 1 1 1_2 \quad -127_{10}
  \]
  
  \[
  0 0 0 0 0 0 1 0_2 \quad +2_{10}
  \]

- \( e = 2 \Rightarrow \) "regulärer" Bereich

- \( \text{verstecktes Bit der Mantisse} \Rightarrow m = 1,01000...000 = 1,01_2 = 1,25_{10} \)

Ergebnis: \(-1,25 \cdot 2^2 = -5,0\)

Quelle: [01]
1. Aufgabe

*Kurzfassung:*

\[ (-1)^s \cdot (1, f_{22} \ldots f_0) \cdot 2^{(e_7 \ldots e_0 - 127)} \]

\[ (-1)^1 \cdot (1,0\,1\,0\,0 \ldots 0\,0) \cdot 2^{(129 - 127)} \]

\[ = -1,25 \cdot 2^2 \]

\[ = -5,0 \]

Quelle: [01]
1. Aufgabe

Kurzfassung:

\((-1)^s \cdot (1, f_{22} \ldots f_0) \cdot 2^{e_7 \ldots e_0 \cdot 127}\)

\((-1)^1 \cdot (1,0,1,0,0 \ldots 0,0) \cdot 2^{129 \cdot 127}\)

\[= -1,25 \cdot 2^2\]

\[= -5,0\]

Quelle: [01]
MULTIPLIKATION

2) Mantissen multiplizieren

\[
\begin{array}{c}
1,000_2 \\
1,110_2 \\
\hline
0000 \\
1000 \\
1000 \\
1000 \\
\hline
01,110000_2
\end{array}
\]

Auch durch Überlegung möglich, besser aber ausführlich

3) Normalisieren

⇒ Produkt ist bereits normalisiert: 1,110000_2 \cdot 2^{-3}

Quelle: [01]
MULTIPLIKATION

5) Check auf under-/overflow: \(-126 \leq -3 \leq 127\)
   \[ \Rightarrow \text{kein under-/overflow} \]

4) Runden
   \[ 1,11000_2 \cdot 2^{-3} \Rightarrow 1,110_2 \cdot 2^{-3} \]
   (ggf. neu normalisieren+check, hier nicht nötig!)

6) Vorzeichenbestimmung: Vorzeichen sind verschieden \[ \Rightarrow \text{Produkt negativ} \]

Ergebnis: \[ A \cdot B = -1,110_2 \cdot 2^{-3} \]
\[ = -0,21875_{10} \]

Check: \[ 0,5_{10} \cdot -0,4375_{10} = -0,21875_{10} \]

Quelle: [01]
**ADDITION**

\[ A = 1,000_2 \cdot 2^{-1} \quad \text{bzw.} \quad B = -1,110_2 \cdot 2^{-2} \]

**in diesem Fall: unterschiedliche Vorzeichen**

1) **Check:** \( e_1 > e_2 \) ? \( \text{ja} \quad \Rightarrow \text{nicht vertauschen} \)

2) **Bilden Zweierkomplement der Mantisse** \( m_2 \)

\[
\begin{array}{c}
01,110 \\
10,001 \\
+ 00,001 \\
10,010 \\
\end{array}
\]

**immer m_2!**

3) **Denormalisieren** (rechtsschieben) von \( m_2 \) um \( e_1 - e_2 = 1 \)

\[ 10,010 \Rightarrow 11,0010 \]

**Beibehaltung Vorzeichen!**

Quelle: [01]
**ADDITION**

4) **Addition der Mantissen**
   
   $01,0000 \cdot 2^{-1}$  
   $+11,0010 \cdot 2^{-1}$  
   $100,0010 \cdot 2^{-1}$  

5) **normalisieren um 3 Stellen**
   
   $00,0010 \cdot 2^{-1} \Rightarrow 1,0000_2 \cdot 2^{-4}$  
   (bei gleichen Vorzeichen check auf over-/underflow)

6) **Runden**
   
   $1,0000_2 \cdot 2^{-4} \Rightarrow 1,000_2 \cdot 2^{-4}$  
   (ggf. neu normalisieren+check, hier nicht nötig)

7) **Ggf. Berücksichtigung Vorzeichenwechsel**
   
   Für $A + B$: Fall $n_1 - n_2 \Rightarrow 1,000_2 \cdot 2^{-4} = 0,0625_{10}$
   
   Für $-A - B$: Fall $n_2 - n_1 \Rightarrow -1,000_2 \cdot 2^{-4} = -0,0625_{10}$

Quelle: [01]
3. Aufgabe

Fehler bei Gleitkommaoperationen

- Das Assoziativgesetz

\[ x + (y + z) = (x + y) + z \]

gilt nicht generell innerhalb der Bereichsgrenzen für Gleitkommazahlen.

- Nicht gültige Beispiele:
  - außerhalb der Bereichsgrenzen, d.h. over- oder underflow
  - denormale Zahl

Quelle: [01]
Gültiges Beispiel:

\[ x = 2^{-100} \quad y = 2^0 \quad z = -2^0 \]

\[ x + (y + z) = 2^{-100} + (2^0 - 2^0) = 2^{-100} + 0 = 2^{-100} \]

\[ (x + y) + z = (2^{-100} + 2^0) - 2^0 = 2^0 + 2^0 = 0 \]

Wegen Rundungsfehler!
Literatur

www.tu-darmstadt.de/vvws00-01/ws_00-01_FB20.tud
ALGORITHMEN II.

Fließkomma Addition und Multiplikation
Übungsaufgaben

1. Aufgabe
Welche Dezimalzahl wird durch das folgende 32-Bit Wort, interpretiert als IEEE 754 Gleitkommazahl, dargestellt?

\[ 11000000101000 \ldots 00 \]

2. Aufgabe
Gegeben seien die Gleitkommazahlen
\[ A = 1,000 \cdot 2^{-1} \]
und
\[ B = 1,110 \cdot 2^{-2}. \]
Führen Sie die Operationen
\[ A+B, \]
\[ -A-B, \]
\[ A \cdot B \]
aus. Das Resultat soll jeweils wieder auf vier Stellen gerundet werden.

3. Aufgabe
Das Assoziativgesetz
\[ x + (y +z) = (x + y) +z \]
gilt nicht generell innerhalb der Bereichsgrenzen für Gleitkommazahlen. Geben Sie ein einfaches Beispiel an, das diese Aussage belegt.

Literatur


www.tu-darmstadt.de/vvws00-01/ws_00-01_FB20.tud

Vorgetragen von Prof. Dr. V. Iossifov, TI, FHTW Berlin
ALGORITHMEN II.

11. Beschleunigung der ganzzahligen Division und Multiplikation

### Algorithmen II

**Beschleunigung der ganzzahligen Division und Multiplikation**

*Quelle: [01]*

Prof. Dr. V. Iossifov, TI, FHTW Berlin

<table>
<thead>
<tr>
<th>P</th>
<th>A</th>
<th>Beschreibung</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>1000</td>
<td>( \text{Divide } 8 = 1000 \text{ by } 3 = 0011. \text{ B contains } 0011. )</td>
</tr>
<tr>
<td>00100</td>
<td>0000</td>
<td>( \text{step 1: } B \text{ had two leading } 0s, \text{ so shift left by } 2. \text{ B now contains } 1100. )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{step 2.1: Top three bits are equal. This is case (a), so} )</td>
</tr>
<tr>
<td>01000</td>
<td>0001</td>
<td>( \text{set } q_4 = 0 \text{ and shift.} )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{step 2.2: Top three bits not equal and } P \geq 0 \text{ is case (c), so} )</td>
</tr>
<tr>
<td>01100</td>
<td>0001</td>
<td>( \text{set } q_5 = 1 \text{ and shift.} )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{Subtract } B. )</td>
</tr>
<tr>
<td>11000</td>
<td>0001</td>
<td>( \text{step 2.3: Top bits equal is case (a), so} )</td>
</tr>
<tr>
<td>11000</td>
<td>0010</td>
<td>( \text{set } q_6 = 0 \text{ and shift.} )</td>
</tr>
<tr>
<td>10000</td>
<td>0111</td>
<td>( \text{step 2.4: Top three bits unequal is case (b), so} )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{set } q_7 = -1 \text{ and shift.} )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{Add } B. )</td>
</tr>
<tr>
<td>11100</td>
<td>0111</td>
<td>( \text{step 3: Remainder is negative so restore it and subtract } 1 \text{ from } q. )</td>
</tr>
<tr>
<td></td>
<td>01100</td>
<td>( \text{Must undo the shift in step 1, so right shift by } 2 \text{ to get true remainder.} )</td>
</tr>
<tr>
<td>01000</td>
<td></td>
<td>( \text{Remainder} = 10, \text{ quotient} = 0111 \text{ } - 1 = 0010. )</td>
</tr>
</tbody>
</table>

**FIGURE A.23** SRT division of \( 1000 \div 0011 \).
Jede Schaltung repräsentiert einen unabhängig arbeitenden (3,2) Adder. In jedem Schritt das Bit aus \( P \) das geshiftet werden soll ist das Ls Bit der Summe.

Quelle: [01]
### FIGURE A.26 Multiplication of \(-7 \times -5\) using radix-4 Booth recoding.

Professor Dr. V. Iossifov, TI, FHTW Berlin
(a) Blockdiagramm eines Feldmultiplizierers; (b) die Eingänge des Feldes; (c) das Feld in expandierter Form um alle Addierer darzustellen.

Quelle: [01]
FIGURE A.28 Multipass array multiplier.

Multiplizierer von 8 Bit Zahlen. Der Multiplizierer ist in der Lage 8 Bit Zahlen mit etwa der Hälfte des Aufwandes aus Fig. A 27 zu multiplizieren. Nach dem Ende des 2. Passes die Bits stömen in den "Carry-Propagate Adder (CPA)",

Quelle: [01]

Quelle: [01]
FIGURE A.30 Wallace tree multiplier.

Quelle: [01]
Die linke Summe zeigt, daß die Berechnung von \(1 + 1\) ein Ergebnis von "0" liefert und der Carry Bit bekommt den Wert von "1".

Quelle: [01]

Quelle: [01]
FIGURE A.33 Quotient selection for radix-4 division with quotient digits \(-2, -1, 0, 1, 2\).
**FIGURE A.35** Example of radix-4 SRT division.

Quelle: [01]
Literatur


Data-range Limit

The data-range limit differs, depending on the data type.

### Signed Numbers

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Lower Limit</th>
<th>Upper Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Hex</td>
<td>Decimal</td>
</tr>
<tr>
<td><strong>Byte</strong></td>
<td>80</td>
<td>-128</td>
</tr>
<tr>
<td><strong>Word</strong></td>
<td>8000</td>
<td>-32,768</td>
</tr>
<tr>
<td><strong>Doubleword</strong></td>
<td>80000000</td>
<td>-2,147,483,648</td>
</tr>
</tbody>
</table>

### Unsigned Numbers

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Lower Limit</th>
<th>Upper Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Hex</td>
<td>Decimal</td>
</tr>
<tr>
<td><strong>Byte</strong></td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td><strong>Word</strong></td>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td><strong>Doubleword</strong></td>
<td>00000000</td>
<td>0</td>
</tr>
<tr>
<td>Nr.</td>
<td>Datum</td>
<td>Thema</td>
</tr>
<tr>
<td>-----</td>
<td>---------</td>
<td>---------------</td>
</tr>
<tr>
<td>1</td>
<td>28.03</td>
<td>V. 1, V. 2</td>
</tr>
<tr>
<td>2</td>
<td>04.04</td>
<td>V. 3</td>
</tr>
<tr>
<td>3</td>
<td>11.04</td>
<td>V. 3.1</td>
</tr>
<tr>
<td>4</td>
<td>25.04</td>
<td>V. 3.2</td>
</tr>
<tr>
<td>5</td>
<td>02.05</td>
<td>V. 4</td>
</tr>
<tr>
<td>6</td>
<td>09.05</td>
<td>V. 4.1</td>
</tr>
<tr>
<td>7</td>
<td>16.05</td>
<td>V. 4.2</td>
</tr>
<tr>
<td>8</td>
<td>23.05</td>
<td>V. 5</td>
</tr>
<tr>
<td>9</td>
<td>30.05</td>
<td>V. 6</td>
</tr>
<tr>
<td>10</td>
<td>06.06</td>
<td>V. 7</td>
</tr>
<tr>
<td>11</td>
<td>13.06</td>
<td>V. 8.1</td>
</tr>
<tr>
<td>12</td>
<td>20.06</td>
<td>V. 8.2</td>
</tr>
<tr>
<td>13</td>
<td>27.06</td>
<td>V. 9</td>
</tr>
<tr>
<td>14</td>
<td>04.07</td>
<td>V. 10</td>
</tr>
<tr>
<td>15</td>
<td>11.07</td>
<td>Klausur</td>
</tr>
<tr>
<td>16</td>
<td>18.07</td>
<td>Auswertung der Klausur V</td>
</tr>
</tbody>
</table>
Klausurankündigung

Lehrveranstaltung  
Algorithmen/Anwendungen II. Vorlesung

Umfang und Bewertung  
Der Leistungsnachweis für die Bildung der Semesternote wird durch eine **schriftliche Leistungskontrolle** erbracht.

Themen schwerpunkte  
ALU Algorithmen der Addition, Subtraktion, Multiplikation und Division (integer und FP).

Insgesamt 50 erreichbare Punkte. Zum Bestehen der LK sind mindestens 20.5 Punkte erforderlich.

Hilfsmittel  
*Arbeitsblätter und Mitschriften aus der Vorlesung*

Termin /Dauer  
max. 90 min.

Prof. Dr. V. Iossifov
Kolloquium

Lehrveranstaltung

Algorithmen/Anwendungen II. - Labor

Umfang und Bewertung

Der Leistungsnachweis für die Bildung der Semesternote wird durch einen schriftlichen Beleg in Gruppen von DREI Studenten erbracht.


Der Beleg wird mit max. 70 Punkten und das Kolloquium/schriftlichen Nachweis mit max. 30 Punkten bewertet.

Es gilt:

Beleg * (≤70 Pkt.) • Kolloquium * (≤30 Pkt.) = SEN (≥4.0)

Thema

Entwurf eines ALU Modells in VHDL. Das Verhalten der VHDL Modelle soll im MULTISIM/XILINX CAD System compiliert und simuliert werden.

Insgesamt 100 erreichbare Punkte. Zum Bestehen „mit Erfolg“ sind mindestens 41 Punkte erforderlich.

Belegabgabe und -verteidigung

18.07.2003, AdK, I25
Belegthemen:

Thema 1/Gruppe1: Entwurf eines Ripple-Carry Addieres in VHDL
Thema 2/Gruppe2: Entwurf eines Carry-Lookahead Addieres in VHDL
Thema 3/Gruppe3: Entwurf eines unsigned int Restoring Division Dividers in VHDL
Thema 4/Gruppe4: Entwurf eines unsigned int Non-restoring Division Dividers in VHDL
Thema 5/Gruppe5: Entwurf eines sign int Multiplizierers nach Booth’s Algorithm in VHDL
Thema 6/Gruppe6: Entwurf eines integer unsigned int shift-add Multiplizierers in VHDL
Thema 7/Gruppe7: Entwurf eines integer Dividierers (Version1) in VHDL
Thema 8/Gruppe8: Entwurf eines integer Dividierers (Version2) in VHDL
Thema 9/Gruppe9: Entwurf eines integer Dividierers (Version3) in VHDL
Thema 10/Gruppe10: Entwurf eines FP Addierers in VHDL
Thema 11/Gruppe11: Entwurf eines FP Multiplizierers in VHDL
Thema 12/Gruppe12: Entwurf eines Wallas-Tree Multiplizierers in VHDL
Thema 13/Gruppe13: Entwurf eines integer Multiplizierers (Version1) in VHDL
Thema 14/Gruppe14: Entwurf eines integer Multiplizierers (Version2) in VHDL
Thema 15/Gruppe15: Entwurf eines integer Multiplizierers (Version3) in VHDL
Thema 16/Gruppe16: Entwurf eines integer Carry Save Addierers in VHDL
Thema 17/Gruppe17: Entwurf eines integer Carry Select Addierers in VHDL
Thema 18/Gruppe18: Entwurf eines integer Multiplizierers mit Carry Save Addierern in VHDL

Prof. Dr. V. Iossifov