Algorithmen zur Division

- Umkehrung der Multiplikation: Berechnung von $q = a / b$ durch wiederholte **bedingte Subtraktionen** und **Schiebeoperationen**
- in jedem Schritt wird Divisor b **testweise** vom aktuellen Rest r subtrahiert: $q_i = 1$, falls $r = r - b > 0$
  $q_i = 0$ und **Korrektur** durch $r = r + b$, falls $r < 0$
- Beispiel: $103_{10} / 9_{10} = 11_{10}$ mit Rest 4_{10}

```
0001100111   / 01001   = 01011
- 01001                    ↑
    00111               Quotient
  - 01001
    11110
  + 01001   <----  Korrektur
    001111
   - 01001
     001101
    - 01001
      00100  <--   Rest
```


Algorithmen zur Division (Forts.)

- allgemein gilt: $a = q \cdot b + r$ mit Rest $r < b$
- im folgenden sei angenommen, daß b eine positive n-Bit Zahl und a eine positive $2n$-Bit Zahl darstellen
  $\Rightarrow$ es muß gelten: 1) $a < 2^{n-1} \cdot b$ bzw. $q < 2^{n-1}$
  2) $b \neq 0$ ($\Rightarrow$ Ausnahmebehandlung!)
- alle Divisionsalgorithmen führen in Schritt i folgende Operation aus: $r(i) = 2 \cdot r(i-1) - q_{n-i} \cdot 2^n \cdot b$ mit $i = 1,\dots,n$ und $r(0) = a$
- Es wird korrektes Ergebnis berechnet, da für Rest $r = r(n)$ gilt:

$$\begin{aligned} r(n) &= 2 \cdot r(n-1) - q_0 \cdot 2^n \cdot b \\ &= 2 \cdot (2 \cdot r(n-2) - q_1 \cdot 2^n \cdot b) - q_0 \cdot 2^n \cdot b = \dots \\ &= 2^n\, r(0) - (2^{n-1} q_{n-1} + \dots + 2q_1 + q_0) \cdot 2^n \cdot b \end{aligned}$$

Somit folgt: $a = r(0) = q \cdot b - r(n)/2^n$

Algorithmen zur Division (Forts.)

- Algorithmus **mit Wiederherstellung des Rests** durch **Addition** („*Restoring Division*")

- statt einer $2n$-Bit Addition $r = r + 2^n \cdot b$ genügt hier auch eine n-Bit Addition $r' = r' + b$, wenn r' die aktuellen höherwertigen n Bit von r darstellt

```
r = a
q = 0
for i = 0 to n-1 {
   shift left r by 1
   r = r - 2^n b
   if (r >= 0)
      qbit = 1
   else
      qbit = 0
      r = r + 2^n b
   q = 2q + qbit
}
```


Algorithmen zur Division (Forts.)

- Algorithmus mit **Bildung eines neuen Rests** nur in dem Fall, daß die Differenz nicht negativ ist („*Non-Performing Division*")

```
r = a
q = 0
for i = 0 to n-1 {
   shift left r by 1
   tmp = r - 2^n b
   if (tmp >= 0)
      qbit = 1
      r = tmp
   else
      qbit = 0
   q = 2q + qbit
}
```

Algorithmen zur Division (Forts.)

- Algorithmus **mit Beibehaltung eines negativen Restes** („*Non-Restoring Division*")
- korrigierende **Addition** anstatt Subtraktion in den Folgeschritten, bis Partialrest *r* wieder positiv ist
- ggf. Korrektur bei negativem Rest erforderlich

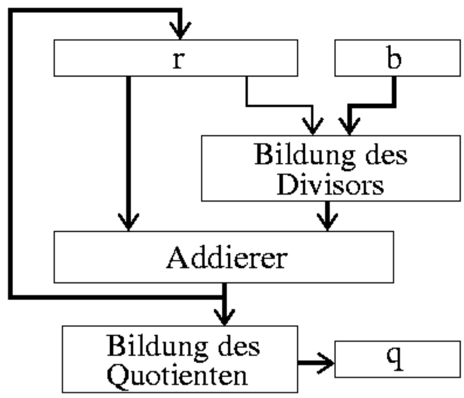
```
r = a
q = 0
for i = 0 to n-1 {
   shift left r by 1
   if (r >= 0)
      r = r - 2^n b
   else
      r = r + 2^n b
   if (r >= 0)
      qbit = 1
   else
      qbit = 0
   q = 2q + qbit
}
if (r < 0)
   r = r + 2^n b
   q = q - 1
```


Algorithmen zur Division (Forts.)

- zur **Äquivalenz** von „*Restoring*" und „*Non-Restoring*" Division:
  - „*Restoring*":
    $r(i) = 2 \cdot r(i–1) – 2^n \cdot b < 0 \Rightarrow r(i+1) = 2 \cdot r(i) – 2^n \cdot b = 4 \cdot r(i–1) – 2^n \cdot b$
  - „*Non-Restoring*":
    $r(i) = 2 \cdot r(i–1) – 2^n \cdot b < 0 \Rightarrow r(i+1) = 2 \cdot r(i) + 2^n \cdot b = 4 \cdot r(i–1) – 2^n \cdot b$
- **Aufwand**: im Worst-Case Fall sind genau $n – 1$ Nullen im Quotient q enthalten
  - „*Restoring*" Division:
    $n + n – 1$ Additionen/Subtraktionen
  - „*Non-Performing*" Division:
    n Subtraktionen und $n – 1$ Kopieroperationen
  - „*Non-Restoring*" Division:
    n Additionen/Subtraktionen (ggf. +1 Korrekturaddition)

  $\Rightarrow$ „*Non-Restoring*" Division ist das schnellste Verfahren!

Implementierung:



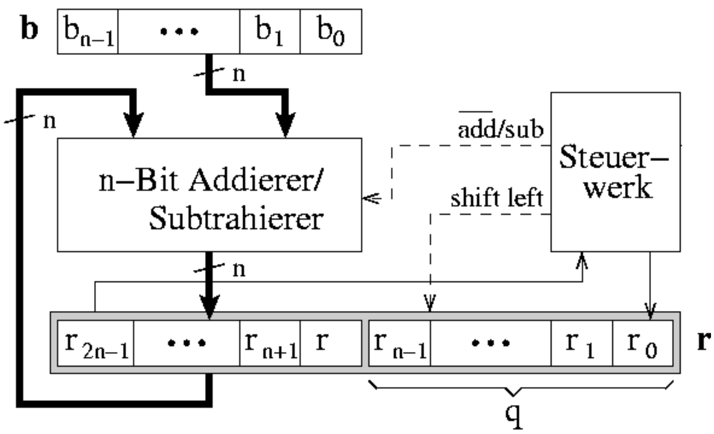
- allgemeiner Aufbau eines Dividierers:
- Behandlung **negativer Dividenden** und **Divisoren** sehr umständlich:
  – es gibt kein Äquivalent zum Booth-Algorithmus !
  – i.a. Umwandlung in Vorzeichen und Betrag
- es gibt verschiedene Möglichkeiten, zur **Beschleunigung** der Division:
  – Verwendung von schnellen Addierern bzw. Carry-Save Addierern
  – Überspringen von k führenden Nullen im Rest r: schiebe Rest r um k Positionen nach links und setze die ersten k Bits von q auf 0
  – simultane Generierung mehrerer Quotientenbits durch Subtraktion des Vielfachen von b
  – parallele Subtraktionen sind jedoch nicht möglich !

sequentieller Dividierer

- sequentielle Division ist **direkt in Hardware** implementierbar

  mit

  n-Bit Register b,
  $2n$-Bit Register r,
  n-Bit Addierer/Subtrahierer



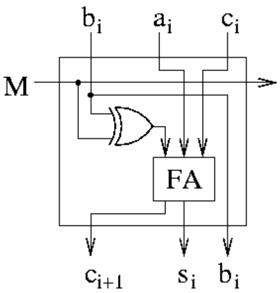
- nach n Schritten steht Quotient in $r_{n-1} \ldots r_0$, Rest in $r_{2n-1} \ldots r_n$
- das Steuerwerk implementiert Algorithmus, z.B. gilt für „*Non-Restoring*“ Division: $\overline{\text{add}}/\text{sub} := r_{2n-1}$ und $r_0 := \overline{r_{2n-1}}$
- Zeit: $n \cdot (\Delta_{\text{Add}} + 3\tau)$

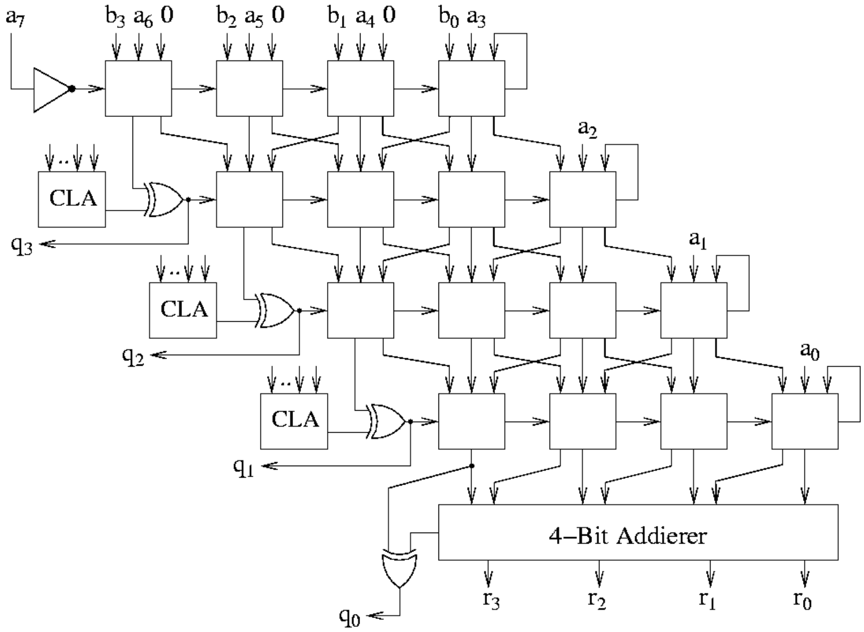
paralleler Dividierer

- wiederholte Subtraktion auch durch **Felddividierer** („*array divider*“) mit CSA-Addierern implementierbar:

- Zeit:
  $\Delta_{\text{Add}} + (n-1)8\tau + 5\tau$

- Aufbau einer Zelle:





SRT-Dividierer

- benannt nach den Entwicklern Sweeny, Robertson und Tocher, die diesen Algorithmus fast gleichzeitig vorstellten (1958)

- Algorithmus mit **dreiwertiger Kodierung der Quotientenbits**
  $q'_i \in \{\overline{1}, 0, 1\}$ bzw.
  $q'_i \in \{-1, 0, 1\}$:

- Quotient *q* wird somit **redundant** kodiert

- weniger Additionen und Subtraktionen als bei der „*Non-Restoring*“ Division

```
r = a
for i = 1 to n {
   shift left r by 1
   if (r >= 2^n b)
      qbit = 1
      r = r - 2^n b
   else if (r < -2^n b)
      qbit = -1
      r = r + 2^n b
   else
      qbit = 0
   q'= 2q' + qbit
}
if (r < 0)
   r = r + 2^n b
   q'= q'- 1
```

SRT-Dividierer (Forts.)

- *Problem*: SRT-Dividierer benötigt je Schritt Vergleich von r sowohl mit $2^n\, b$ als auch mit $-2^n\, b$
- *Lösung*: der Divisor b wird zuvor durch Schiebeoperationen derart **normalisiert**, daß hinter dem Vorzeichenbit das erste Nachkommabit $\neq 0$ ist, d.h. es gilt: $½ \leq b < 1$
- der Vergleich wird dann wie folgt angenähert:

  Dividend a und Rest r sind hierbei auch Zahlen aus [0,1)

```
if (r >= 1/2)
   qbit = 1
   r = r - b
else if (r < -1/2)
   qbit = -1
   r = r + b
else
   qbit = 0
```

- aufgrund der redundanten Darstellung von q ist Ergebnis korrekt!
- Rückwandlung von q in binär kodierte Zahl erforderlich!

SRT-Dividierer (Forts.)

- Zahl der erforderlichen Operationen ist **datenabhängig**!
- für einen n-Bit Dividenden benötigt SRT Algorithmus im Mittel nur $n/2.67$ Additionen
- weitere Beschleunigung durch simultane Generierung **mehrerer** Quotientenbits, d.h. je Schritt Bestimmung einer Quotientenziffer $q_i \in \{ -\alpha, -\alpha+1, ..., -1, 0, 1, ..., \alpha-1, \alpha\}$
  (dies wird auch als Radix-2^α SRT Verfahren bezeichnet)
- Abschätzung von q_i erfolgt i.a. über in PLAs gespeicherten **Tabellen** in Abhängigkeit von den höherwertigen Bits des aktuellen Rests r und den höherwertigen Bits des Divisors b
- bei dem 1994 im Intel Pentium Prozessor entdeckten Bug waren 5 Einträge in einer solchen Tabelle falsch!

Rechnen bei eingeschränkter Präzision

- Integer-Rechenwerke sind optimiert für die Verwendung ganzer Zahlen, nicht für das Rechnen mit Festkommazahlen!
- prinzipiell werden reelle Zahlen x aus einer Anwendung mittels **Skalierung** auf ganze n-Bit Zahlen x' abgebildet, die als Festkommazahlen interpretiert werden: $x' = \lfloor 2^q \cdot x \rfloor$
- drei Fälle*:*
  1) fester beschränkter Wertebereich von x : $x \in [a, b]$ mit $a < 0$
     $\Rightarrow$ Zahl der Vorkommabitstellen: $s = \lceil \log_2(\max\{|a|, |b|\}) \rceil + 1$
  2) fester beschränkter Wertebereich von x : $x \in [a, b]$ mit $a \geq 0$
     $\Rightarrow$ Zahl der Vorkommabitstellen: $s = \lceil \log_2 |b| \rceil$
  3) Wertebereich von x unbeschränkt, lediglich ein typischer Wert (z.B. Startwert) $x_0 \neq 0$ ist bekannt
     $\Rightarrow$ Zahl der Vorkommabitstellen: $s = \lceil \log_2 |x_0| \rceil + \alpha$ mit α abgeschätzt

  Zahl der Nachkommabitstellen in allen drei Fällen: $q = n - s$

Rechnen bei eingeschränkter Präzision (Forts.)

- **Probleme** beim Rechnen mit Festkommazahlen:
  – Wahl eines guten Skalierungsfaktors 2^q, mit dem eine reelle Zahl x in eine Festkommazahl $x' = \lfloor x \cdot 2^q \rfloor$ umgerechnet werden kann
    $\Rightarrow$ Festkommazahl x' ist mit **Quantisierungsfehler** ε_x behaftet: $x' = x + \varepsilon_x$
  – ist Zahl der Vorkommastellen s zu klein, so ist die **Dynamik** zu niedrig: Wahrscheinlichkeit für Überlauf ist groß!
  – ist Zahl der Nachkommastellen q zu klein, so ist die **Präzision** zu gering: Genauigkeit kann insbesondere für iterative Verfahren unzureichend sein!
  – der aus dem Zweierkomplement resultierende **asymmetrische Zahlenbereich** ist insbesondere bei kleinen Wortbreiten n oft ungünstig
- betragsmäßig sehr kleine Festkommazahlen können mit einer **negativen Vorkommastellenzahl** s kodiert werden:

  Beispiel: $n = 8, s = -4 \;\Rightarrow$ Kodierung von $x \in [2^{-12}, 2^{-4} - 2^{-12}]$ möglich !

Fehlerfortpflanzung

- Seien ε_a und ε_b die Fehler, mit denen zwei Festkommavariablen a' und b' behaftet sind: $a' = a + \varepsilon_a$, $b' = b + \varepsilon_b$
- für die Addition $a + b$ folgt:
  $a' + b' = a + \varepsilon_a + b + \varepsilon_b \;\Rightarrow\; \varepsilon_{a+b} = \varepsilon_a + \varepsilon_b$
- für die Multiplikation $a \cdot b$ folgt:
  $a' \cdot b' = a \cdot b + a \cdot \varepsilon_b + b \cdot \varepsilon_a + \varepsilon_a \cdot \varepsilon_b + \varepsilon_{\text{mult}}$
  $\Rightarrow\; \varepsilon_{a \cdot b} \approx a \cdot \varepsilon_b + b \cdot \varepsilon_a + \varepsilon_{\text{mult}}$
  (wobei $\varepsilon_{\text{mult}}$ ein bei Multiplikation entstehender Fehler ist, z.B. durch Bildung eines n-Bit Wertes aus einem $2n$-Bit Produkt)
- bei Anwendung einer Funktion $y = \phi(x)$ gilt:
  $y' = \phi(x + \varepsilon_x) + \varepsilon_\phi \approx \phi(x) + \varepsilon_x \cdot \phi'(x) + \varepsilon_\phi \;\Rightarrow\; \varepsilon_{\phi(x)} \approx \varepsilon_x \cdot \phi'(x) + \varepsilon_\phi$
- bei einer Operation $y = \varphi(x_1,\ldots, x_k)$ gilt: $\varepsilon_{\varphi(x_1,\ldots,x_k)} \approx \sum_{j=1}^{k} \varepsilon_{x_j} \cdot \frac{\partial y}{\partial x_j} + \varepsilon_\varphi$

Truncating vs. Rounding

- beim Rechnen mit Festkommazahlen müssen oft von einer m-Bit Zahl x mit q *Nachkommabitstellen* die niedrigstwertigen r Bits abgeschnitten werden, um eine n-Bit Zahl mit $n < m$ zu erhalten
- im folgenden wird Gleichverteilung für x angenommen
- zwei Techniken:
  1) **Abschneiden** („***Truncating***“):
     Abschneiden der Bitpositionen $x_{-q+r-1}, \ldots, x_{-q+1}, x_{-q}$
     $\Rightarrow\; \varepsilon_x \in [-2^{-q+r} + 2^{-q}, 0]$, mittlerer Fehler: $E[\varepsilon_x] = -½ \cdot (2^{-q+r} - 2^{-q})$
  2) **Runden** („***Rounding***“):
     Abschneiden der Bitpositionen $x_{-q+r-1}, \ldots, x_{-q+1}, x_{-q}$ und Addition von 2^{-q+r}, falls $(x_{-q+r-1} \ldots x_{-q+1}\, x_{-q})_2 \cdot 2^{-q} \geq 2^{-q+r-1}$, d.h. falls gilt: $x_{-q+r-1} = 1$
     $\Rightarrow\; \varepsilon_x \in [-2^{-q+r-1} + 2^{-q}, 2^{-q+r-1}]$, mittlerer Fehler: $E[\varepsilon_x] = -½ \cdot 2^{-q}$
- Runden ist stets vorzuziehen, wird bei Festkomma-Arithmetik i.a. aber nicht durch Hardware unterstützt !

Sättigung

- Eine andere Möglichkeit, aus einer m-Bit Festkommazahl eine n-Bit Zahl (mit $n < m$) zu erhalten, besteht im Abschneiden von r führenden *Vorkommabitstellen*
- Abschneiden der Bitpositionen x_{s-1}, x_{s-2}, ... , x_{s-r} ist fehlerfrei möglich, wenn gilt: $x_{s-1} = x_{s-2} = \ldots = x_{s-r} = x_{s-r-1}$

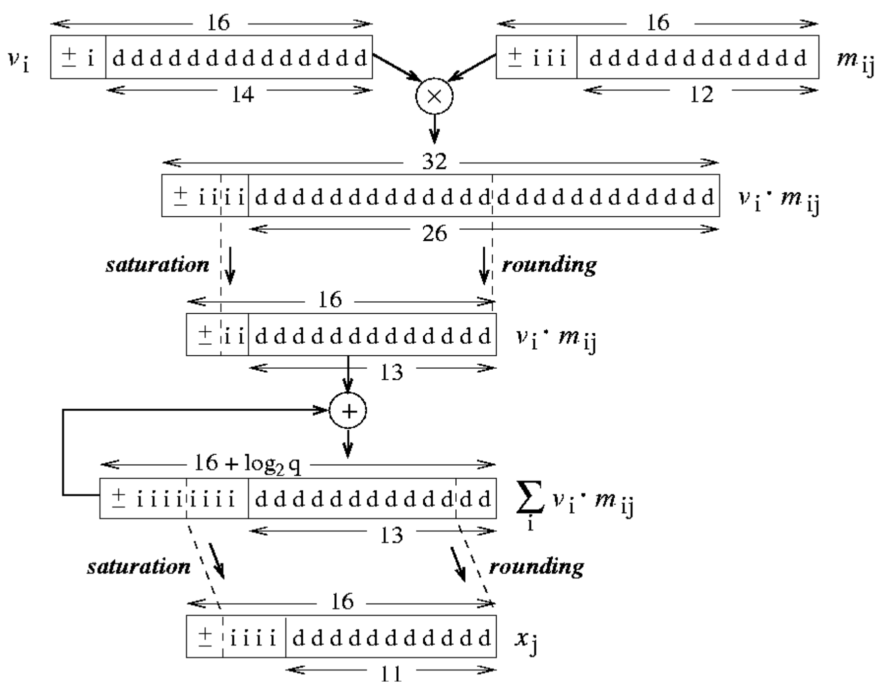
  $\Rightarrow$ $\varepsilon_x \in [-2^{s-1}, 2^{s-1}]$, d.h. der resultierende Fehler liegt in der gleichen Größenordnung wie die Zahl x !
- alternativ ist eine **Sättigung** („***Saturation***“) denkbar, wird von heutiger Integer Arithmetik-Hardware i.a. aber nicht unterstützt:

  Wenn eine der abgeschnittenen Bitpositionen x_{s-2}, ... , x_{s-r} ungleich dem Vorzeichen x_{s-1} ist, so wird in $x_{s-r-1} \ldots x_0$ der größtmögliche darstellbare positive oder negative Wert generiert

  $\Rightarrow$ $\varepsilon_x \in [-2^{s-1} + 2^{s-r-1}, 2^{s-1} - 2^{s-r-1}]$
- typische Anwendung: Addition zweier n-Bit Zahlen

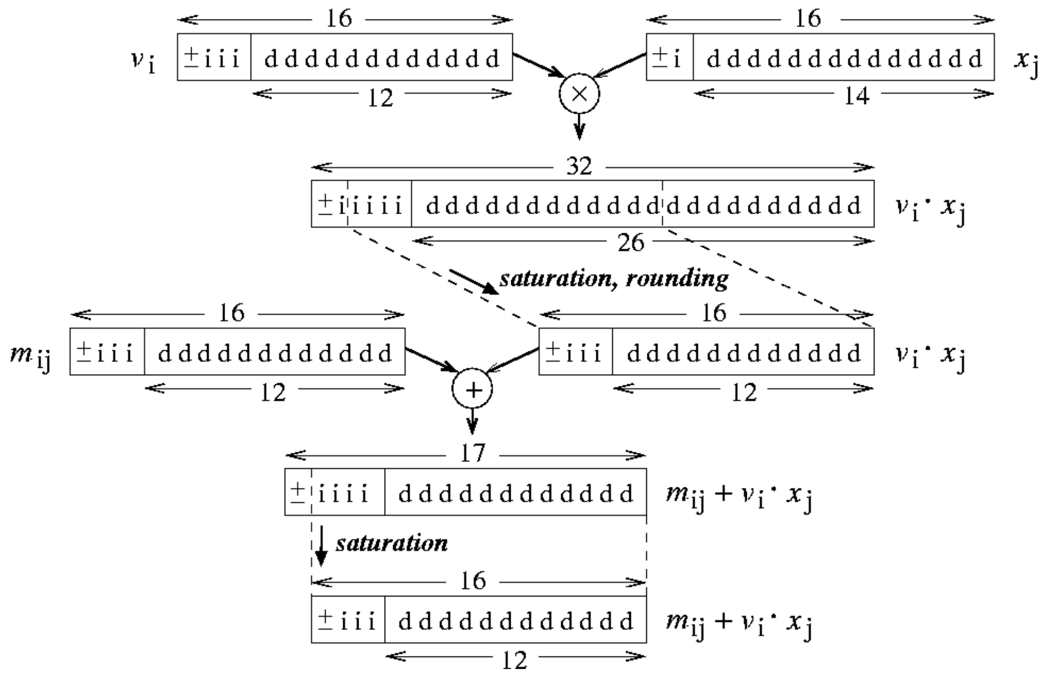
Beispiel 1:

Berechnung von $x_j = \sum_{i=1}^{q} v_i \cdot m_{ij}$ mit 16-Bit Festkommazahlen:



Beispiel 2:

Berechnung von $m_{ij} = m_{ij} + v_i \cdot x_j$ mit 16-Bit Festkommazahlen :



Beispiel 3:

Berechnung von $m_{ij} = m_{ij} + \eta \cdot (v_i - m_{ij}) \cdot x_j$: