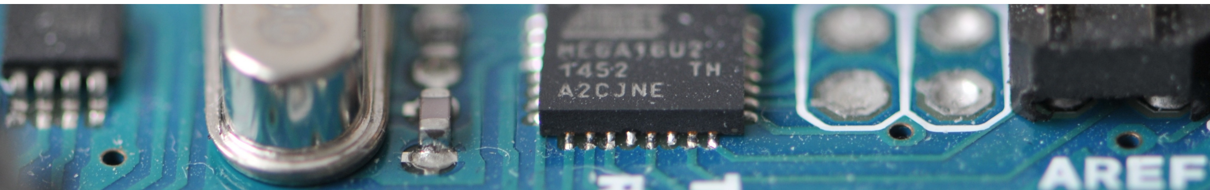
universität
innsbruck



Rechnerarchitektur

Arithmetik II

Univ.-Prof. Dr.-Ing. Rainer Böhme

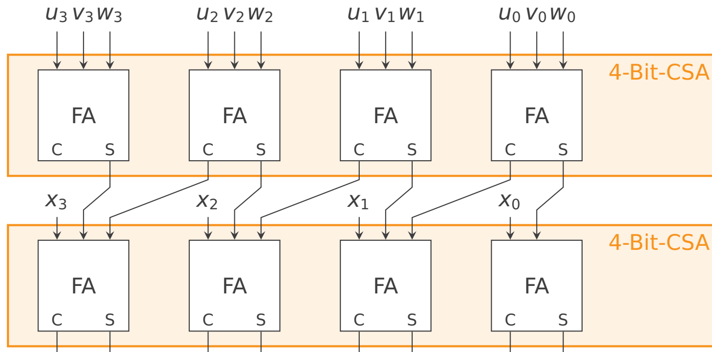
Wintersemester 2020/21 · 18. November 2020

Gliederung heute

1. **Multiplikation**
2. Division
3. Rechnen mit Nachkommastellen

Carry-Save-Addierer (CSA)

Anordnung zur **partiellen Addition**: Ein CSA-Baustein integriert *n* Volladdierer mit **separaten** Carry-Ausgängen



→ Kaskadierung zur schnellen Addition mehrerer Summanden

Konstruktion von Multiplizierern

Ansätze (Auswahl)

a. Zweistufiges Schaltnetz (vgl. VL Kombinatorische Logik II)
b. Serielles Schaltwerk (vgl. serielles Addierwerk)
c. Feldmultiplizierer (vgl. paralleles Addierwerk)
d. Optimierungsmöglichkeiten

Multiplikation als Schaltnetz

$n \times n$-Bit-Multiplizierer als Schaltnetz mit je $2n$ Ein- und Ausgängen

- Implementierung z. B. in PROM mit 2^{2n} Zeilen aus $2n$-Bit-Worten
- **Sehr geringe Zeitverzögerung:** zwei Stufen $\Rightarrow$ ca. 2τ
- **Sehr hoher Schaltungs-/Speicheraufwand**

n	Produkt: $2n$	Zeilen: 2^{2n}	(P)ROM Größe
2	4	16	64 Bit
4	8	256	256 Byte
8	16	65 536	128 Kilobyte
16	32	$4.3 \cdot 10^9$	16 Gigabyte
32	64	$1.8 \cdot 10^{19}$	148 Exabyte

$\rightarrow$ „Skaliert nicht.“ Präziser: Speicheraufwand exponentiell in n

Multiplikation positiver Binärzahlen

Das Produkt $y = a \times b$ aus zwei n-Bit-Faktoren hat $2n$ Stellen.

Algorithmus wie bei schriftlicher Dezimal-Multiplikation:

Pseudocode

$y \leftarrow 0$
for $i = 0$ to $n - 1$ **do**
  if $b_i = 1$ **then**
    $x \leftarrow a$ um i Bit nach links verschoben
    $y \leftarrow y + x$
  end if
end for

Beispiel für $n = 5$

$a \times b$:	01010 $\times$ 01101		
	01010	$\times 1$	b_0
	00000	$\times 0$	b_1
	01010	$\times 1$	b_2
	01010	$\times 1$	b_3
	00000	$\times 0$	b_4
$y =$	0010000010		

$\rightarrow$ Zurückführung auf bedingte Addition und Schiebeoperationen

Modifizierter Algorithmus

Ersetzen der Linksverschiebung von a durch Rechtsverschiebung von y.

Pseudocode

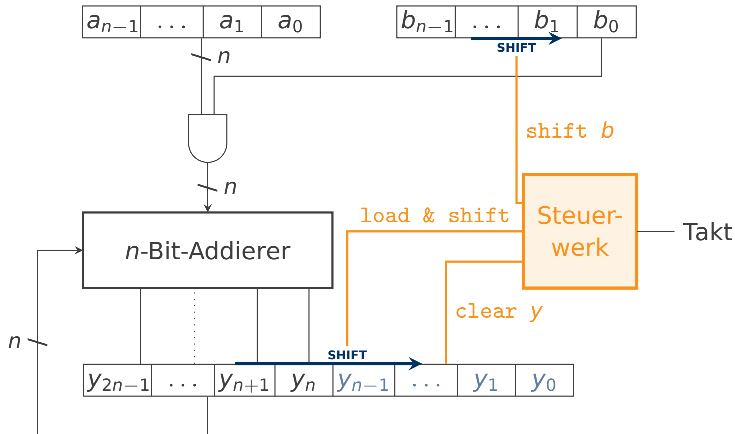
$y \leftarrow 0$
for $i = 0$ to $n-1$ **do**
  if $b_i = 1$ **then**
    $(y_{2n-1}, \ldots, y_n) \leftarrow (y_{2n-1}, \ldots, y_n) + a$
  end if
  {Verschiebe y um ein Bit nach rechts.}
  $(y_{2n-1}, \ldots, y_0) \leftarrow (0, y_{2n-1}, \ldots, y_1)$
end for

Vorteil: Nur ein Schieberegister mit $2n$ Bit

Beispiel

```
      01010    ×  01101
      ----------------
      0000000000
    + 01010       add a
      ----------
      0101000000
      0010100000  shift
      0001010000  shift
    + 01010       add a
      ----------
      0110010000
      0011001000  shift
    + 01010       add a
      ----------
      1000001000
      0100000100  shift
  y = 0010000010  shift
```

Realisierung als serielles Multiplizierwerk



Die Berechnung dauert n Taktzyklen.

Hörsaalfrage

Sei $n = 3$, $a = (101)_2$ und $b = (010)_2$.

Welche Folge aus Steuersignalen führt eine erfolgreiche Multiplikation $y = a \times b$ aus?

Antwort A	Antwort B	Antwort C
1. clear y	1. clear y	1. clear y
2. shift b	2. load & shift	2. load & shift
3. load & shift	3. shift b	3. shift b
4. shift b	4. load & shift	4. load & shift
5. load & shift	5. shift b	5. shift b
6. shift b	6. load & shift	6. load & shift
7. load & shift	7. shift b	7. clear y

Hinweis: Nehmen Sie an, dass pro Takt nur ein Steuersignal erzeugt wird. (In der Praxis müssen mehrere Signale kombiniert werden, um die genannte Laufzeit von n Takten zu erreichen. Überlegen Sie als **Hausaufgabe**, welche.)

Zugang: `https://arsnova.uibk.ac.at` mit Zugangsschlüssel **24 82 94 16**. Oder scannen Sie den QR-Kode.

Direkte Schaltung einer schriftlichen Multiplikation

$$
\begin{array}{cccccccc}
(a_3, a_2, a_1, a_0) & & & \times & & & (b_3, b_2, b_1, b_0) & \\
\hline
 & & & & a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 \\
 & & & a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 & 0 \\
 & & a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 & 0 & 0 \\
 & a_3b_3 & a_2b_3 & a_1b_3 & a_0b_3 & 0 & 0 & 0 \\
\hline
y_7 & y_6 & y_5 & y_4 & y_3 & y_2 & y_1 & y_0
\end{array}
$$

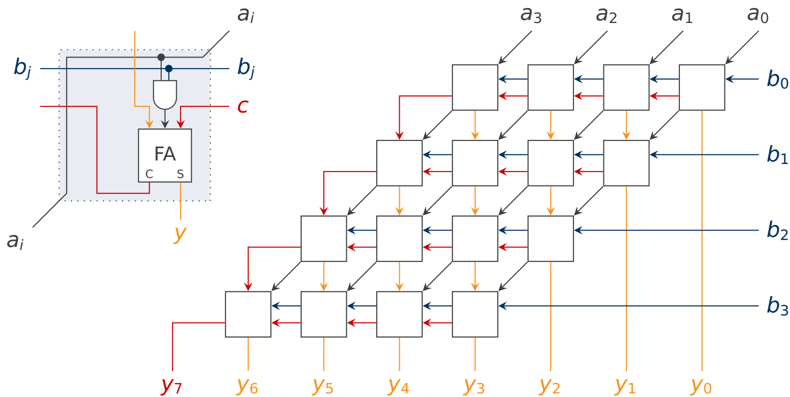
n^2 **Bitprodukte**

n **partielle** Produkte

Feldmultiplizierer

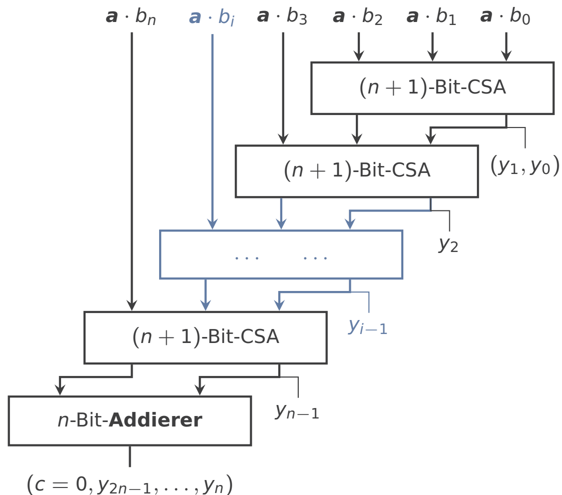
(engl. *array multiplier*)

Strukturierter Aufbau aus n^2 **Multiplizierzellen**



Variante mit CSA-Kaskade

(Skizze)



Multiplikation negativer Zahlen

Bislang Beschränkung auf **positive** Faktoren. Was passiert bei der Multiplikation von **negativen** Zahlen in **Zweierkomplement**-Darstellung ?

$$a \cdot (-b) \equiv a \cdot (2^n - b) = a \cdot 2^n - a \cdot b \qquad [\text{statt } 2^{2n} - a \cdot b]$$

$$(-a) \cdot b \equiv (2^n - a) \cdot b = b \cdot 2^n - a \cdot b \qquad [\text{statt } 2^{2n} - a \cdot b]$$

$$(-a) \cdot (-b) \equiv (2^n - a) \cdot (2^n - b) = 2^{2n} - a \cdot 2^n - b \cdot 2^n + a \cdot b \qquad [a \cdot b]$$

$\rightarrow$ **Naives Multiplizieren liefert falsche Ergebnisse.**

Lösungsansätze

1. Trennung von Vorzeichen und Betrag
2. Addition von Korrekturtermen
3. Algorithmus von Booth (mit vorzeichenrichtiger Ergänzung)

Optimierungsmöglichkeiten

Beobachtung: Jede Eins in b kostet eine Addition.

$$a \times 111111 \quad \text{vs.} \quad a \times 100001$$

$$a \times 111111 = a \times 1000000 - a \times 0000001$$

Die Multiplikation mit einer 1-Folge kann immer durch **eine** Addition und **eine** Subtraktion ersetzt werden.

Anwendung

Der Algorithmus von **Booth** analysiert zwei benachbarte Bits b_i und b_{i-1}:

- 11 nichts tun (wie nach wie vor bei 00)
- 10 Subtraktion von $a \times 2^i$
- 01 Addition von $a \times 2^i$

Wichtig um den Anfang (von rechts) einer Folge nicht zu verpassen:
Initiale Ergänzung $b_{-1} = 0$, falls $b_0 = 1$.

Booth 1951

Algorithmus von Booth

Beispiel für $n = 8$:

$$42 \times 92 = (0010\,1010)_2 \times (0101\,1100)_2$$

$$-42 = (1101\,0110)_2$$

$$\begin{array}{r c l}
0010\,1010 \times 0101\,1100 & & \\
\hline
 & & 0101\,1100 \\
1111\,1111\,0101\,10 & \leftarrow & 0101\,1100 \\
 & & 0101\,1100 \\
 & & 0101\,1100 \\
0000\,0101\,010 & \leftarrow & 0101\,1100 \\
1111\,0101\,10 & \leftarrow & 0101\,1100 \\
0001\,0101\,0 & \leftarrow & 0101\,1100 \\
\hline
1\,0000\,1111\,0001\,1000 & = & (3864)_{10}
\end{array}$$

Weitere Beispiele

mit negativen Faktoren und initialer Ergänzung ($n = 5$):

$(10)_{10} = (01010)_2$
$(-10)_{10} = (10110)_2$
$(-13)_{10} = (10011)_2$

$(10)_{10} \times (-13)_{10}$

$$
\begin{array}{rcl}
01010 \times 100110 & = b_{-1} & \\
\hline
11\,1111\,0110 & \leftarrow & 100110 \\
00\,0010\,10 & \leftarrow & 100110 \\
11\,0110 & \leftarrow & 100110 \\
\hline
111\,0111\,1110 & = (-130)_{10} &
\end{array}
$$

$(-10)_{10} \times (-13)_{10}$

$$
\begin{array}{rcl}
10110 \times 100110 & = b_{-1} & \\
\hline
00\,0000\,1010 & \leftarrow & 100110 \\
11\,1101\,10 & \leftarrow & 100110 \\
00\,1010 & \leftarrow & 100110 \\
\hline
100\,1000\,0010 & = (130)_{10} &
\end{array}
$$

Gliederung heute

1. Multiplikation
2. **Division**
3. Rechnen mit Nachkommastellen

Schriftliche Division von Binärzahlen mit Rest

Beispiel 29 : 6 = 4 **Rest** 5

$n = 5$ Bits

	000011101 : 00110 = 00100	
	− 00110	**Quotient**
	11011	
Korrektur	+ 00110	
	00011	
	− 00110	
	11101	
Korrektur	+ 00110	
	000111	
	− 00110	
	000010	
	− 00110	
	11100	
Korrektur	+ 00110	
	000101	
	− 00110	
	11111	
Korrektur	+ 00110	
Rest	00101	

Divisionsalgorithmus mit „Restoring“

Verwendung von bedingter Addition, **Subtraktion** und Schiebeoperationen

Pseudocode

Require: Dividend a, Divisor b (jeweils n Bit)

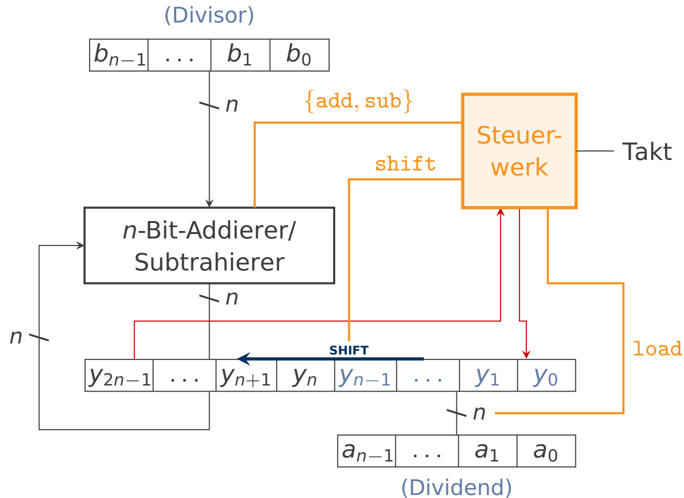
$(y_{n-1}, \ldots, y_0) \leftarrow a$ {Initialisiere (“`load`”) $2n$-Bit-Register y.}
$(y_{2n-1}, \ldots, y_n) \leftarrow 0$

for $i = 0$ to $n-1$ **do**
  $(y_{2n-1}, \ldots, y_0) \leftarrow (y_{2n-2}, \ldots, y_0, 0)$ {Schiebe y um ein Bit nach links.}
  $(y_{2n-1}, \ldots, y_n) \leftarrow (y_{2n-1}, \ldots, y_n) - b$

  if $y_{2n-1} = 0$ **then**
    $y_0 \leftarrow 1$
  else
    $(y_{2n-1}, \ldots, y_n) \leftarrow (y_{2n-1}, \ldots, y_n) + b$ {Wiederherstellung des Rests}
  end if
end for

$r \leftarrow (y_{2n-1}, \ldots, y_n)$
$q \leftarrow (y_{n-1}, \ldots, y_0)$ {Ergebnis. Es gilt: $a = b \times q + r$ }

Realisierung als serielles Dividierwerk



Steuersignale für Division mit „Restoring“

Beispiel 29 : 6 = 4 **Rest** 5

$n = 5$ Bits

	Rechnung			Register	Signal
	000011101 : 00110 =	00100		00000 11101	load
	− 00110	**Quotient**		00001 11010	shift
	11011			11011 11010	sub
Korrektur	+ 00110				
	00011			00001 11010	add
	− 00110			00011 10100	shift
	11101			11101 10100	sub
Korrektur	+ 00110				
	000111			00011 10100	add
	− 00110			00111 01000	shift
	000010			00001 01001	sub
	− 00110			00010 10010	shift
	11100			11100 10010	sub
Korrektur	+ 00110				
	000101			00010 10010	add
	− 00110			00101 00100	shift
	11111			11111 00100	sub
Korrektur	+ 00110				
Rest	00101			00101 00100	add

Optimierungsmöglichkeiten bei der Division

Verbesserungen der „langsamen Verfahren“
(Berechnung von 1–2 Ergebnisstellen pro Iteration)

- CSA statt Addierer/Subtrahierer und Verrechnung der Überträge bei Korrektur bzw. im Folgeschritt ($\rightarrow$ SRT-Division)
- Radix-4-Zahlendarstellung $\{-2, -1, 0, +1, +2\}$ oft kombiniert mit Quotienten-Tabelle in ROM ($\rightarrow$ Intel Pentium-FDIV-Bug 1994)

  “Is there a list of Pentium jokes? I need one! :-)”
  “No, no. You meant to say you need .999856738903.”

„Schnelle Verfahren“ beginnen mit einer Schätzung und verdoppeln die Genauigkeit in jedem Schritt

- Das **Goldschmidt**-Verfahren multipliziert Dividend und Divisior mit Faktoren bis Divisor zum Wert 1 konvergiert.
- Die **Newton-Raphson**-Division sucht Kehrwert und multipliziert.

SRT: Sweeney, Robertson, Tochter (1958)

Gliederung heute

1. Multiplikation
2. Division
3. **Rechnen mit Nachkommastellen**

Darstellung rationaler und reeller Zahlen

1. **Festkomma:** Jede (darstellbare) Kommazahlen z wird durch lineare Skalierung auf eine ganze Zahl z' abgebildet.

   Rechner arbeitet unverändert auf ganzzahliger Abbildung.

2. **Gleitkomma:** Darstellung von Kommazahlen durch Argument (Mantisse) a und Charakteristik (Exponent) c zur Basis r:

   $z = a \times r^c$ Bsp. für $r = 10$: $0.000035 \Rightarrow 3.5 \times 10^{-5}$

   Spezielle Rechenwerke erforderlich (oder Realisierung in Software)

3. **Exakte Darstellung rationaler Zahlen** durch separate Speicherung und Verarbeitung von Zähler und Nenner als Ganzzahlen

   Realisiert in Software für exaktes wissenschaftliches Rechnen

   (hier nicht weiter behandelt)

Festkommazahlen

Zahl zur Basis b mit **fester Anzahl** $k < n$ Stellen **nach** dem Komma:

$$z = (\underbrace{z_{n-1}, z_{n-2}, \ldots, z_{k+1}, z_k}_{\text{ganzzahliger Teil}} \cdot \underbrace{z_{k-1}, z_{k-2}, \ldots, z_1, z_0}_{\text{gebrochener Teil}})_b$$

$$= z_{n-1} \cdot b^{n-k} + z_{n-2} \cdot b^{n-k-1} + \cdots + z_{k+1} \cdot b^1 + z_k \cdot b^0 + $$
$$z_{k-1} \cdot b^{-1} + z_{k-2} \cdot b^{-2} + \cdots + z_1 \cdot b^{-k+1} + z_0 \cdot b^{-k}$$

- Die Konstante k kennt nur die Anwendung. Sie dient der **Interpretation** der Zahlen.
- Die Rechenwerke arbeiten **transparent** mit skalierten ganzen Binärzahlen $z' = z \cdot 2^k$.

Beispiel und Hörsaalfrage

24 82 94 16

Ein 8-Bit-Register enthält die Binärzahl $z' = (01101110)_2$.

Für $k = 3$ gilt:

$$z = (01101.110)_2 = 2^3 + 2^2 + 2^0 + 2^{-1} + 2^{-2} = 13.75$$

Hörsaalfrage

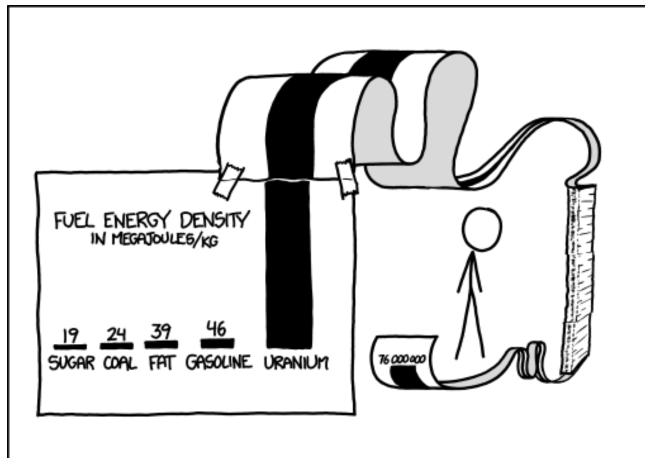
Welche Bitfolge ist die Festkommadarstellung von $z = 13.1875$ mit $k = 4$?

1. 00111101
2. 11010110
3. 011010010
4. 011010011
5. 110100011

Zugang: `https://arsnova.uibk.ac.at` mit Zugangsschlüssel **24 82 94 16**. Oder scannen Sie den QR-Kode.

Gleitkommazahlen

(auch Fließkomma, engl. *floating point*)



SCIENCE TIP: LOG SCALES ARE FOR QUITTERS WHO CAN'T FIND ENOUGH ~~PAPER~~ TO MAKE THEIR POINT *PROPERLY*.

FLIPFLOPS

Quelle: xkcd.com

Gleitkommazahlen zur Basis $r = 2$

Allgemeine Darstellung nach IEEE 754

$$z = (-1)^s \times 1.f \times 2^{e-b}$$

Binärformat

$$\overbrace{(s, e_{p-1}, \ldots, e_1, e_0, f_{m-1}, \ldots, f_1, f_0)}^{1+p+m=n\text{ Bit}}$$

- **Mantisse** aus Vorzeichen s und **normalisiertem Betrag** $a = 1.f$ im Bereich $1.00\ldots00$ bis $1.11\ldots11$ ohne die führende Eins.
- **Exponent** e mit konstantem **Bias** $b = 2^{p-1} - 1 \geq 0$
- Darstellbarer Zahlenbereich: $\pm 2^{1-b}, \ldots, (2 - 2^{-m}) \times 2^b$
- Zwischen 2^{e-b} und 2^{e-b+1} gibt es 2^m Gleitkommazahlen, deren Abstand von e abhängt.

IEEE 754

Standardisierte Formate für die Gleitkommazahlendarstellung

Genauigkeit		*single*	*double*	*quad*
Gesamtbreite	n [Bit]	32	64	128
davon:				
Mantisse	m	23	52	112
Exponent	p	8	11	15
Vorzeichen		1	1	1
Bias	b	127	1 023	16 383
Minimum (Betrag)	$\lvert z_{\text{min}} \rvert$	2^{-126} $\approx 10^{-38}$	2^{-1022} $\approx 10^{-308}$	2^{-16382} $\approx 10^{-4932}$
Maximum (Betrag)	$\lvert z_{\text{max}} \rvert$	$\approx 10^{38}$ $(2 - 2^{-23}) \times 2^{127}$	$\approx 10^{308}$ $(2 - 2^{-52}) \times 2^{1023}$	$\approx 10^{4932}$ $(2 - 2^{-112}) \times 2^{16383}$
gültige Dezimalstellen		7.22	15.95	34.02

Kodierung besonderer Zahlen

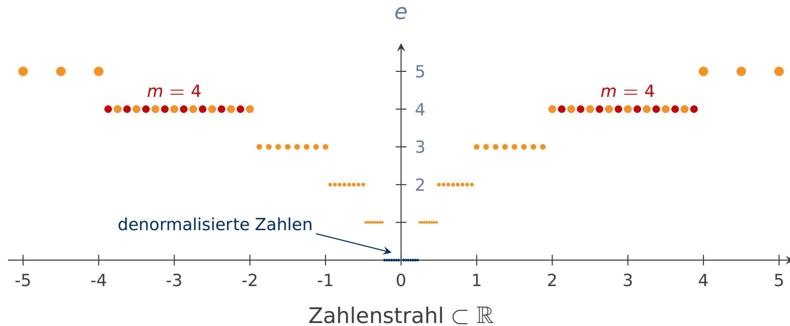
IEEE 754 definiert Spezialfälle, die mit $e = \mathbf{0}$ oder $e = \mathbf{1}$ kodiert werden:

		Kodierung		
Zahl	Bezeichnung	e	f	s
$z = +0$	*positive zero*	$\mathbf{0}$	0	0
$z = -0$	*negative zero*	$\mathbf{0}$	0	1
$z = +\infty$	*positive infinity*	$\mathbf{1}$	0	0
$z = -\infty$	*negative infinity*	$\mathbf{1}$	0	1
$z = \text{NaN}$	*not a number*	$\mathbf{1}$	$\neq 0$	$\mathbf{d}$
$z = (-1)^s \times 0.f \times 2^{1-b}$	*denormalized number*	$\mathbf{0}$	$\neq 0$	$\{0, 1\}$

Visualisierung

$$z = (-1)^s \times 1.f \times 2^{e-b}$$

Beispiele für $p = 3$, $m = 3$ $\Rightarrow b = 2^{3-1} - 1 = 3$



Ausnahmesituationen

Überlauf, wenn nach Normalisierung für $z : e \geq e_{\max} = \mathbf{1}$

- Ausgabe von $+\infty$, falls $z > 0$; $-\infty$, falls $z < 0$
- Auch bei Division durch Null $\pm x : 0 = \pm\infty$ (falls $x \neq 0$)
- Rechenregeln für ∞:
  $\infty \pm x = \infty$ (falls $x \neq \mp\infty$), $\infty \cdot x = \pm\infty$ (falls $x \neq 0$)

Unbestimmtes Ergebnis

- $\infty \cdot 0 = \text{NaN}$, $0 : 0 = \text{NaN}$, $\infty - \infty = \text{NaN}$
- Für alle Operationen mit NaN gilt: $F(x, \text{NaN}) = \text{NaN}$

Unterlauf, wenn nach Normalisierung für $z : e = 0$

- Ausgabe einer **denormalisierten** Darstellung von z
- Ausgabe von $z = 0$ (*"flushing to zero"*)

Multiplikation von Gleitkommazahlen

Faktoren: $(-1)^s \times a \times 2^{\alpha-\mathrm{bias}}$ und $(-1)^t \times b \times 2^{\beta-\mathrm{bias}}$

1. **Multipliziere Mantissen** als Festkommazahlen: $y = a \times b$
   $a = 1.f_a$ und $b = 1.f_b$ haben $m+1$ Stellen $\Rightarrow$ y hat $2m+2$ Stellen
2. **Addiere Exponenten** $\gamma = \alpha + \beta - \mathrm{bias}$
3. **Berechne Vorzeichen** $u = s \oplus t$
4. **Normalisiere Produkt** $(-1)^u \times y \times 2^{\gamma-\mathrm{bias}}$
   i. Falls $y \geq 2$, schiebe y um ein Bit nach rechts und erhöhe γ um 1.
   ii. Setze $y = 1.f_y = 1.(y_{2m-1}, y_{2m-2}, \ldots, y_m)_2$ mit Rundung.
5. **Behandle Ausnahmesituationen**
   i. Überlauf, falls $\gamma \geq e_{\max} = 2^p - 1$ $\Rightarrow$ Rückgabe $\pm\infty$ (abh. von u)
   ii. Unterlauf, falls $\gamma \leq e_{\min} = 0$ $\Rightarrow$ Denormalisierung
   iii. Zero, falls $y = 0$ $\Rightarrow$ Rückgabe ± 0 (abh. von u)

Addition von Gleitkommazahlen

Summanden: $(-1)^s \times a \times 2^{\alpha-\mathrm{bias}}$ und $(-1)^t \times b \times 2^{\beta-\mathrm{bias}}$

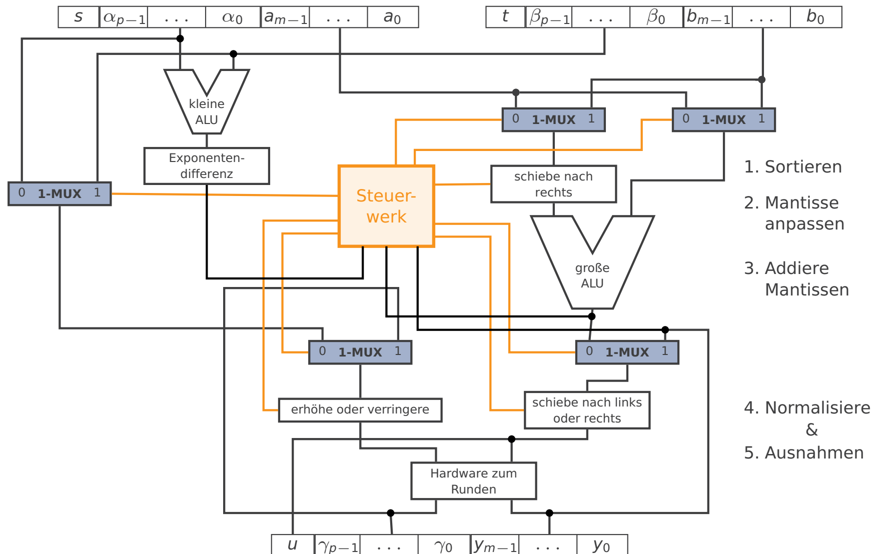
1. **Sortiere** die Summanden, sodass $\alpha \leq \beta = \gamma$
2. **Bitposition der Mantisse anpassen:** Bestimme a', sodass

$$(-1)^s \times a \times 2^{\alpha-\mathrm{bias}} = (-1)^s \times a' \times 2^{\gamma-\mathrm{bias}}$$

   durch Rechtsschieben von a um $\beta - \alpha$ Bits.

3. **Addiere Mantissen**
   i. Falls nötig, bilde Zweierkomplement von a' oder b (abh. von s und t)
   ii. Festkomma-Addition $y = a' + b$
   iii. Falls $y < 0$, setze $u = 1$ und bilde Zweierkomplement von y
4. **Normalisiere Summe** $(-1)^u \times y \times 2^{\gamma-\mathrm{bias}}$
   i. Falls $y \geq 2$, schiebe y nach rechts und erhöhe γ um 1.
   ii. Solange $y < 1$, schiebe y nach links und verringere γ um 1.
5. **Behandle Ausnahmesituationen:** Überlauf, Unterlauf, $y = 0$

Skizze eines Gleitkomma-Addierwerks



Syllabus – Wintersemester 2020/21

07.10.20	1.	Einführung
14.10.20	2.	Kombinatorische Logik I
21.10.20	3.	Kombinatorische Logik II
28.10.20	4.	Sequenzielle Logik I
04.11.20	5.	Sequenzielle Logik II
11.11.20	6.	Arithmetik I
18.11.20	7.	Arithmetik II
25.11.20	8.	Befehlssatzarchitektur (ARM) I
02.12.20	9.	Befehlssatzarchitektur (ARM) II $\xrightarrow{\text{danach}}$ **in**day students virtual edition
09.12.20	10.	Prozessorarchitekturen
16.12.20	11.	Ein-/Ausgabe
13.01.21	12.	Speicher
20.01.21	13.	Leistung
27.01.21		**Klausur (1. Termin)**