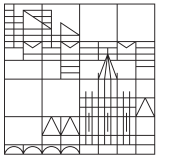
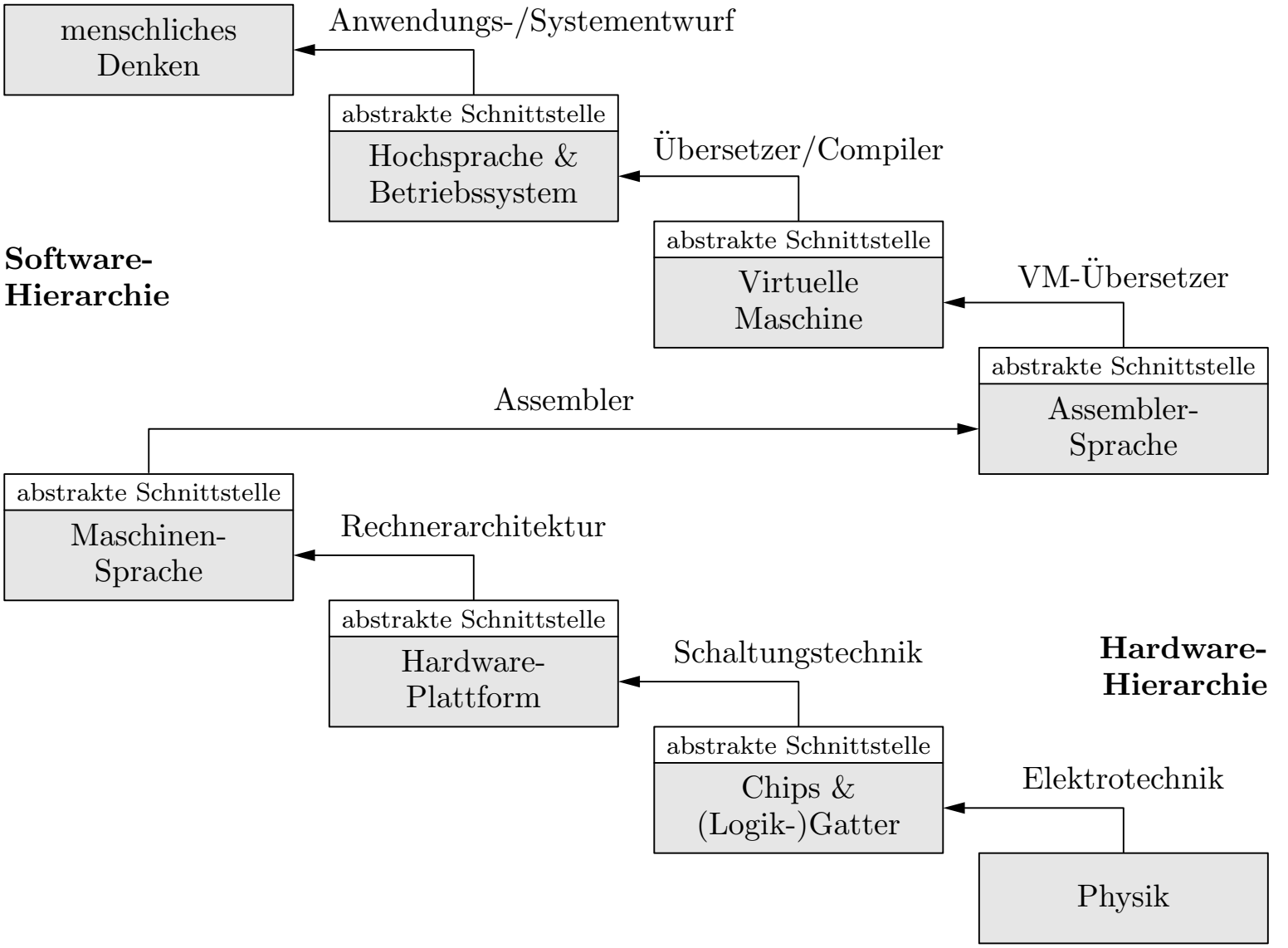
Universität Konstanz



Rechnersysteme und -netze
Kapitel 3
Binärarithmetik und ihre Implementierung

Bastian Goldlücke
Universität Konstanz
WS 2019/20

Rechnersysteme: Plan der Vorlesung



Erinnerung: Schaltungstechnik I

- **Boolesche Algebra / Schaltalgebra**
  - Definitionen, Operatoren, Schreibweisen etc.
  - Wahrheitstafeln und Boolesche Funktionen
  - Disjunktive und konjunktive Normalform
  - Vollständige Operationenmengen
- **Gatterlogik**
  - Elementare (Logik-)Gatter, Schaltzeichen für Gatter
  - Zusammengesetzte Gatter: Schnittstelle und Implementierung
  - Bitmustererkennung, Datenflußsteuerung, Multiplexer und Dekodierer
- **Implementierung durch Schaltkreise**
  - Elektrotechnische Grundlagen, Schalter, Transistoren etc.
  - Integrierte Schaltungen und ihre Herstellung

Erinnerung: Schaltungstechnik II

- **Minimierung Boolescher Formeln**
  - Transformationen mit Hilfe von Äquivalenzen
  - Das Minimierungsverfahren von Karnaugh–Veitch (Karnaugh–Veitch-Diagramme)
  - Das Minimierungsverfahren von Quine–McCluskey (Methode der Primimplikanten, Petricks Algorithmus)
  - Programmierbare Logikarrays (programmable logic arrays, PLAs) (Implementierung von Disjunktionen von Konjunktionen von Literalen)

- **Hardware-Beschreibungssprache** (hardware description language, HDL)
  - Beschreibung der Zusammenschaltung von Gattern (Schnittstelle und Implementierung)
  - Simulationsumgebung für Gatterschaltungen (Schaltungstest auf Grundlage einer Spezifikation)

Inhalt

Inhalt

Erinnerung: Zahlendarstellung/Stellenwertsystem

- Im Unterschied z.B. zur römischen Zahlschrift, die eine **additive Zählschrift** mit ergänzender Regel für die subtraktive Schreibung bestimmter Zahlen ist, benutzen wir zur Darstellung i.a. ein Stellenwertsystem.

Römische Zahlschrift:

Zeichen	I	V	X	L	C	D	M
Wert	1	5	10	50	100	500	1000

Beispiele: `MMXVII` $\widehat{=}$ 2017, `MCMLXVI` $\widehat{=}$ 1966, `MDCCXCIV` $\widehat{=}$ 1794.

- Ein **Stellenwertsystem**, **Positionssystem** oder **polyadisches Zahlensystem** ist ein Zahlensystem, bei dem die (additive) Wertigkeit eines Symbols von seiner Position, der **Stelle**, abhängt. Beispiele (Basis 10):

$$742 = 7 \cdot 100 + 4 \cdot 10 + 2 \cdot 1 = 7 \cdot 10^2 + 4 \cdot 10^1 + 2 \cdot 10^0$$

$$0.358 = 3 \cdot \tfrac{1}{10} + 5 \cdot \tfrac{1}{100} + 8 \cdot \tfrac{1}{1000} = 3 \cdot 10^{-1} + 5 \cdot 10^{-2} + 8 \cdot 10^{-3}$$

- Entscheidend für die Entwicklung eines Stellenwertsystems war die Einführung der Null „0“ als Lückenzeichen (für nicht besetzte Stellen). Erst dies machte die Weiterentwicklung der Mathematik möglich.

Erinnerung: Zahlensysteme

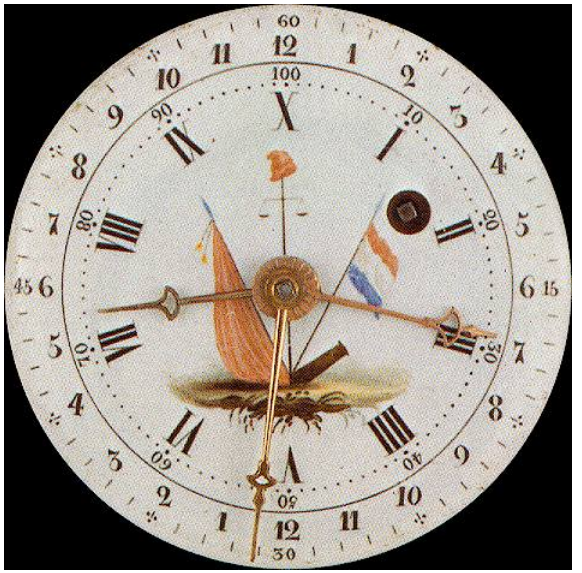
- Im Alltag verwenden wir im wesentlichen zwei Zahlensysteme:
  - Das **Dezimalsystem** (Basis 10) für die Darstellung
    z.B. von Anzahlen, Längen, Gewichten, Preisen etc.
  - Das **Sexagesimalsystem** (Basis 60) für die Darstellung von Zeiten:
    Eine Stunde hat 60 Minuten, eine Minute hat 60 Sekunden.

    Das Sexagesimalsystem wird in der Mathematik (Geometrie)
    und der Geodäsie (Erdvermessung) auch für Winkel verwendet:
    Ein Winkelgrad hat 60 Winkelminuten (oder Bogenminuten),
    eine Winkelminute hat 60 Winkelsekunden (oder Bogensekunden).

- In früheren Zeiten war auch das **Duodezimalsystem** (Basis 12) anzutreffen:
  Ein Dutzend sind 12 Einheiten, ein Gros hat 12 Dutzend,
  ein Maß (oder Großgros) hat 12 Gros. (Auch: Ein Tag hat $24 = 2 \cdot 12$ Stunden.)

- Vermutliche Gründe für die Verwendung dieser Systeme:
  Wir haben 10 Finger; 12 ist die kleinste Zahl, die durch 1, 2, 3 und 4 teilbar ist;
  60 die kleinste Zahl, die durch 1, 2, 3, 4, 5 und 6 teilbar ist (ohne Rest).

Erinnerung: Zahlensysteme



www.wikipedia.org

io9.gizmodo.com

- Zeit wurde manchmal auch im Dezimalsystem dargestellt, sogenannte **Dezimalzeit**.

- In China wurde die Dezimalzeit lange parallel zur Duodezimalzeit benutzt:

  Ein Tag (Mitternacht bis Mitternacht) wurde sowohl in 12 Doppelstunden (*shi chén*) als auch in 10 *shi* / 100 *ké* unterteilt. Jedes *ké* wurde in 100 *fen*, jedes *fen* wiederum in 100 *miao* unterteilt.

- In der Zeit der französischen Revolution, ab 1793, wurde für kurze Zeit versucht, das Zeitsystem auf ein Dezimalsystem umzustellen:

  Ein Tag sollte 10 Stunden haben, eine Stunde 100 Minuten, eine Minute 100 Sekunden.

  Links: Zwei Uhren aus dieser Zeit.

Quelle: Wikipedia

Erinnerung: Zahlensysteme

- Im Prinzip kann jede beliebige Zahl als Basis eines Zahlensystems gewählt werden.

- Das **Duodezimalsystem** und das **Sexagesimalsystem** haben den Vorteil, daß (einfache) Brüche oft leichter in Stellenschreibweise darzustellen sind.

Bruch	$1/2$	$1/3$	$1/4$	$1/5$	$1/6$	$1/8$
Dezimalsystem	0.5	0.333 . . .	0.25	0.2	0.1666 . . .	0.125
Duodezimalsystem	0.6	0.4	0.3	0.2497 . . .	0.2	0.15
Sexagesimalsystem	$0.d_{30}$	$0.d_{20}$	$0.d_{15}$	$0.d_{12}$	$0.d_{10}$	$0.7d_{30}$

- Das in der Rechnertechnik verwendete **Binärsystem** (Basis 2) hat den Vorteil der kleinstmöglichen Zahl an Ziffernzeichen, nämlich nur zwei: 0 und 1.

- Da wir im folgenden mit verschiedenen Zahlensystemen (verschiedenen Basen) arbeiten werden, brauchen wir eine Notation, wie eine Zahl zu interpretieren ist.

  Wir werden die Konvention benutzen, daß die Basis des Zahlensystems als Subskript rechts der Zahl angegeben wird (wenn dies Subskript fehlt: Basis 10):

  $342_{12} = 482_{10} = 742_8 = 2122_6 = 13202_4 = 122212_3 = 111100010_2.$

Erinnerung: Zahlensysteme

- Die Zahl 742 kann auch in einem anderen Zahlensystem als einem mit Basis 10 interpretiert werden, z.B. in **Oktalsystem** (Basis 8):

$$742_8 = 7 \cdot 8^2 + 4 \cdot 8 + 2 \cdot 1 = 448 + 32 + 2 = 482_{10}.$$

  Deswegen ist die explizite Angabe der Basis wichtig, wenn nicht schon durch den Zusammenhang klar ist, welche Basis benutzt wird.

- Wenn eine Zahl in einem Zahlensystem mit Basis b interpretiert werden soll, so sind die gültigen **Ziffernzeichen** (oder kurz **Ziffern**) $0, \ldots, b-1$. Dagegen ist b keine gültige Ziffer im Zahlensystem mit der Basis b. Stattdessen:

$$8_8 = 8 \cdot 8^0 = 1 \cdot 8^1 + 0 \cdot 8^0 = 10_8.$$

- In der Rechnertechnik wird auch das **Hexadezimalsystem** (Basis 16) häufig verwendet. Die Hexadezimalziffern sind

$$0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}, \mathtt{E}, \mathtt{F}.$$

  D.h., $\mathtt{A}_{16} = 10_{10}$, $\mathtt{B}_{16} = 11_{10}$, $\mathtt{C}_{16} = 12_{10}$, $\mathtt{D}_{16} = 13_{10}$, $\mathtt{E}_{16} = 14_{10}$, $\mathtt{F}_{16} = 15_{10}$.

Erinnerung: Zahlensysteme

- Allgemein gilt für die Interpretation einer Zahl im einem Zahlensystem mit Basis b und Ziffern $d_k, \ldots, d_1, d_0$, mit $k \geq 0$ und $\forall i, 0 \leq i \leq k : \ 0 \leq d_i < b$:

$$[d_k \ldots d_1 d_0]_b \ = \ d_k \cdot b^k \ + \ \ldots \ + \ d_1 \cdot b^1 \ + \ d_0 \cdot b^0.$$

  Erinnerung: $b^1 = b$, $b^0 = 1$.

  Bemerkung: Das Zeichen „$\forall$“ bedeutet „für alle“.
  Der Ausdruck „$\forall i, 0 \leq i \leq k : \ 0 \leq d_i < b$“ ist daher zu lesen als
  „Für alle i, die zwischen 0 (einschließlich) und k (einschließlich) liegen,
  gilt, daß d_i zwischen 0 (einschließlich) und b (ausschließlich) liegt.“

- Einige Programmiersprachen unterstützen die Angabe von Zahlen in bestimmten anderen Zahlensystemen (außer der Standardbasis 10), meist im **Oktalsystem** (Basis 8) und im **Hexadezimalsystem** (Basis 16).

  Als Beispiel betrachten wir die Programmiersprache C (Angabe der Konstante n):

Basis b	C-Syntax	Beispiel
8	`0`n	`0742`
16	`0x`n	`0xCAFE`

Erinnerung: Addition und Subtraktion

	4	6	7	8	5	3	9	9
+	2_1	8	0_1	7	1	2_1	3_1	4
=	7	4	8	5	6	6	3	3

	7	4	8	5	6	6	3	3
−	2_1	8	0_1	7	1	2_1	3_1	4
=	4	6	7	8	5	3	9	9

- Aus der Grundschule ist bekannt, daß man Addition und Subtraktion mit Zahlen in einem Stellenwertsystem sehr leicht **stellenweise** ausführen kann.
- Einziges Problem ist die Behandlung des **Stellenüberlaufs**, der auftritt, wenn die Summe der Ziffernzeichen die Basis des Zahlensystems (hier: Basis 10) erreicht oder überschreitet bzw. des **Stellenunterlaufs**, der auftritt, wenn die Ziffer des Subtrahenden (unten) die Ziffer des Minuenden (oben) übersteigt.
- In diesem Fall entsteht ein **Übertrag** (carry), der bei der Subtraktion umgangssprachlich auch „Borgen“ von der nächsten Stelle genannt wird.
- Diese Rechenschemata sind nicht nur im Dezimalsystem (Basis 10), sondern im Prinzip in jedem Zahlensystem analog anwendbar.

  Wir werden sie speziell für das Binärsystem (Basis 2) betrachten.

Erinnerung: Multiplikation und Division

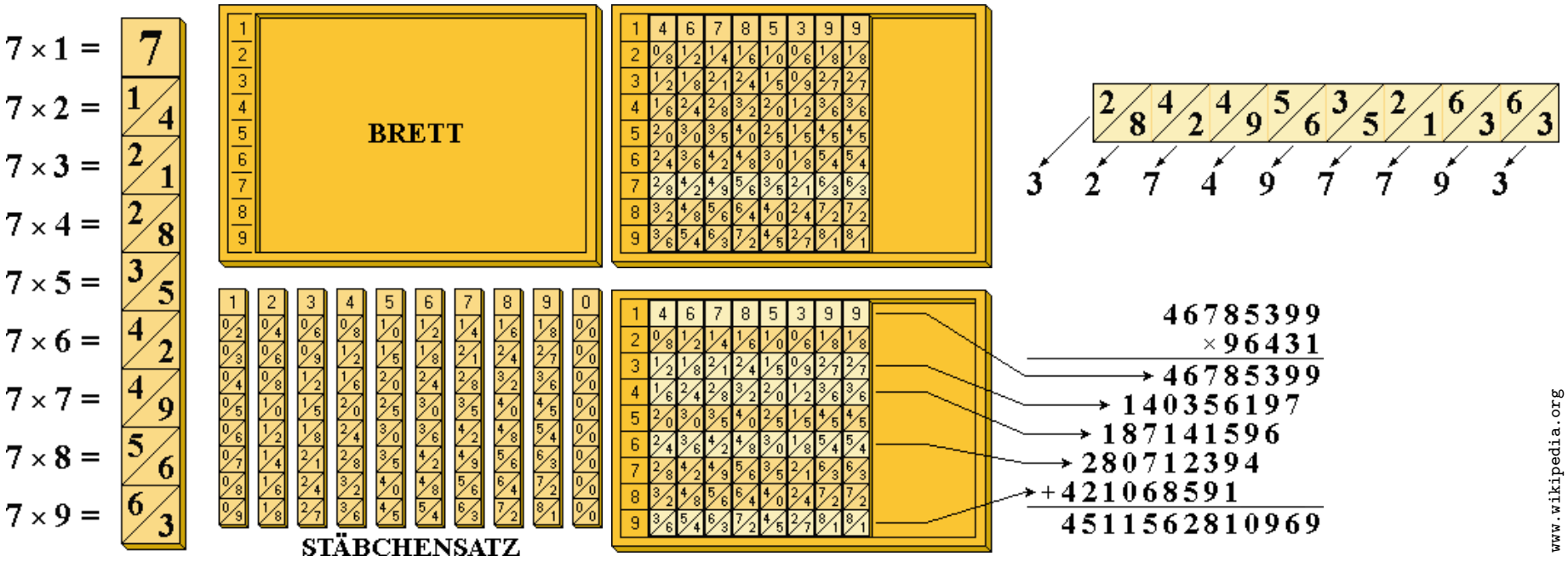
	46785399	
×	96431	
	46785399	1
+	1403561970	3
+	187141596	4
+	28071239400	6
+	4210685910	9
=	4511562810969	

	4511562810969	
÷	46785399	=
−	4210685910	9
=	300876900	
−	280712394	6
=	20164506	9
−	18714159	6
=	1450347	36
−	1403561	97
=	467853	999
−	467853	99 1
=	0	

- Bekannt aus der Grundschule: Die Multiplikation wird auf eine **Summe von Stellenprodukten** zurückgeführt.

- Bei der Division besteht das Hauptproblem darin, den richtigen Stellenfaktor zu bestimmen. Dies geschieht meist über eine Schätzung, anschließendes Ausprobieren, und gegebenenfalls eine Korrektur der Schätzung.

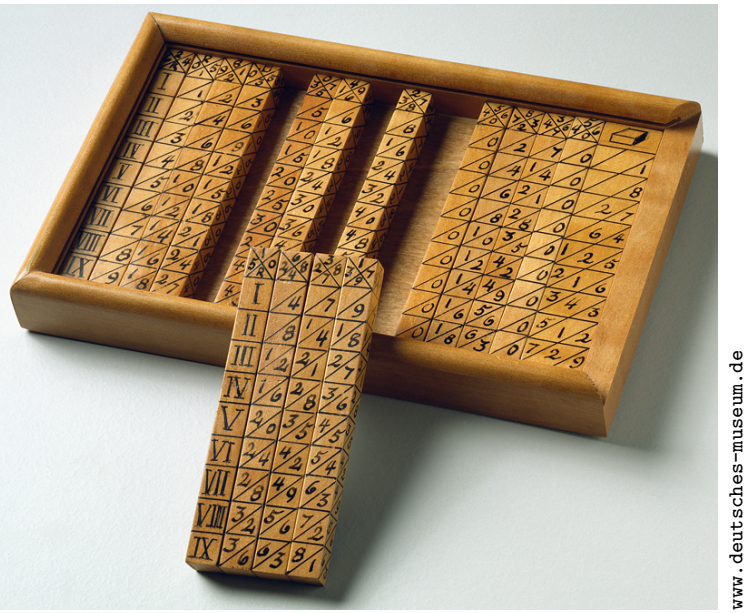
Rechenhilfsmittel (Basis 10)



- Mit sogenannten **Napierschen Rechenstäbchen** [John Napier 1617] wird bei der Multiplikation und Division das Bestimmen der Stellenprodukte erleichtert.
- Man erkennt hier, daß eines der Hauptprobleme die Zahl der Ziffernzeichen ist, durch die so viele verschiedene Stellenprodukte möglich sind, wie es Ziffern gibt.
- **Binärarithmetik** (nur zwei Ziffern: 0 und 1) vermeidet dieses Problem:
  Es gibt nur zwei mögliche Stellenprodukte: 0 und den Faktor selbst.

Mechanische Rechner: Basis 10

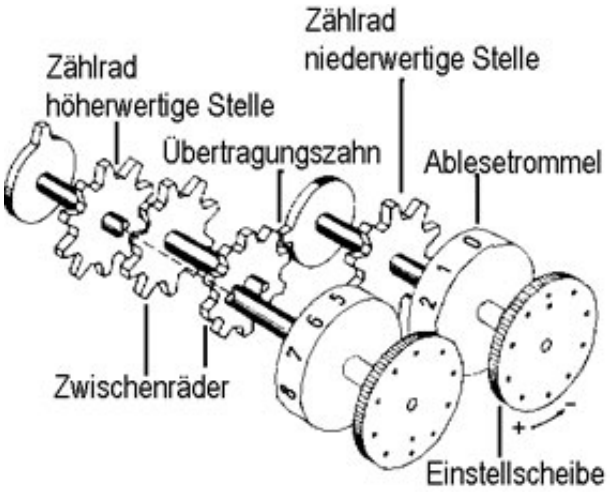
www.arithmeum.uni-bonn.de



www.deutsches-museum.de

- Die „**Rechenuhr**“ [Wilhelm Schickard 1623] war der erste Rechner.
- Mit der „Rechenuhr“ konnte man Addieren und Subtrahieren (Unterschieden durch die Drehrichtung der Einstellräder).
- Die „Rechenuhr“ erleichterte das Multiplizieren und Dividieren, da in sie ein Äquivalent der **Napierschen Rechenstäbchen** [John Napier 1617] eingebaut war (allerdings auf Zylindern statt auf Stäbchen).

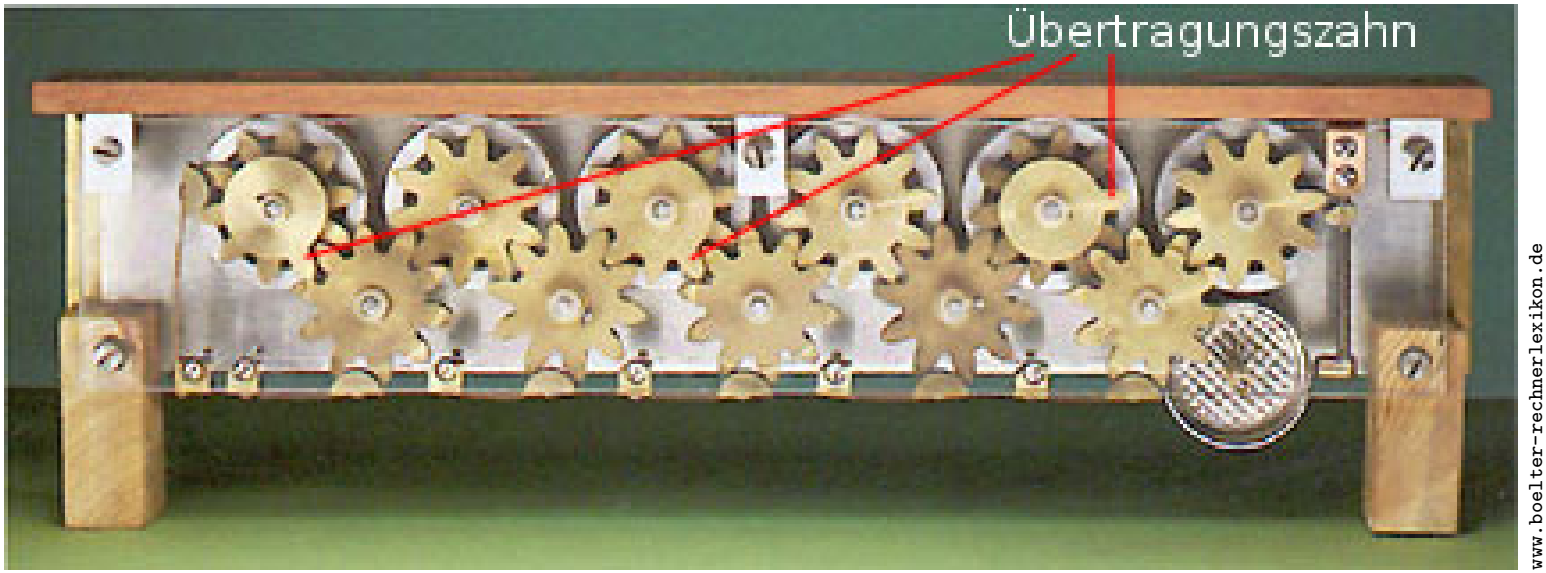
Mechanische Rechner: Übertrag in Basis 10



www.boelter-rechnerlexikon.de

Entscheidend war die Idee eines Übertragungszahns (einzahniges Zahnrad, „verstümmeltes Rädchen“ in einem Brief Wilhelm Schickards an Johannes Kepler), um in einer Addition/Subtraktion einen Übertrag auf die nächste Stelle zu behandeln.

Die „Pascaline“ [Blaise Pascal 1642] benutzte eine ähnliche Idee.



www.boelter-rechnerlexikon.de

Inhalt

Binärarithmetik

- Die beiden Ziffern des **Binärsystems** (auch: **Dualsystem**), nämlich 0 und 1, können direkt durch Schalter implementiert werden.

Zifferndarstellung:

Schalterzustand	Binärziffer
offen	0
geschlossen	1

Analog zur Logik:

Schalterzustand	Wahrheitswert
offen	0 (falsch)
geschlossen	1 (wahr)

- Das Prinzip der Zahlendarstellungen ist direkt anwendbar:

$$101010_2 = 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 42_{10}$$

- Wie bereits bemerkt, ist eine übliche Bezeichung für eine einzelne Binärstelle das Wort „**Bit**“ (Zusammenziehung aus “***bi***nary digi***t***”).

- Einige Berechnungen sind im Binärsystem sehr einfach:
  - Test auf gerade oder ungerade Zahl: Prüfe geringwertigste Stelle.
  - Multiplikation mit einer Zweierpotenz (später mehr dazu).

Mathematischer Exkurs: Summen von Zweierpotenzen

Die folgende Formel ist sehr wichtig für den Umgang mit Binärzahlen.

- Es gilt für **Summen von Zweierpotenzen** die allgemeine Formel:

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1.$$

- Beweis durch **vollständige Induktion**:
  - Induktionsanker ($n = 0$):

$$\sum_{i=0}^{0} 2^i = 2^0 = 1 = 2 - 1 = 2^1 - 1.$$

  - Induktionsannahme:

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1 \qquad \text{gilt bis } n.$$

  - Induktionsschritt ($n \to n+1$):

    ($*$): Verwendung der Induktionsannahme

$$\sum_{i=0}^{n+1} 2^i = 2^{n+1} + \sum_{i=0}^{n} 2^i \overset{(*)}{=} 2^{n+1} + 2^{n+1} - 1 = 2 \cdot 2^{n+1} - 1 = 2^{n+2} - 1.$$

Binärarithmetik: Addition

- Die **Addition** ist die einfachste und am häufigsten verwendete Operation in einer arithmetisch-logischen Einheit (arithmetic logic unit, ALU).

  Bemerkung: Die Addition wird nicht nur für arithmetische Operationen benutzt, sondern z.B. auch dann, wenn der sogenannte Programmzähler (program counter, PC, manchmal auch instruction counter, IC) erhöht wird, um die Adresse der nächsten Programmanweisung zu bestimmen.

- Die Regeln für die Addition zweier Bits x und y lassen sich direkt in Wahrheitstafeln übersetzen:

$$\begin{array}{rr} & 0 \\ + & 0 \\ \hline = & 0 \end{array} \qquad \begin{array}{rr} & 1 \\ + & 0 \\ \hline = & 1 \end{array} \qquad \begin{array}{rr} & 0 \\ + & 1 \\ \hline = & 1 \end{array} \qquad \begin{array}{rr} & 1 \\ + & {\scriptstyle 1}\,1 \\ \hline = & 0 \end{array}$$

x	y	s	c
0	0	0	0
1	0	1	0
0	1	1	0
1	1	0	1

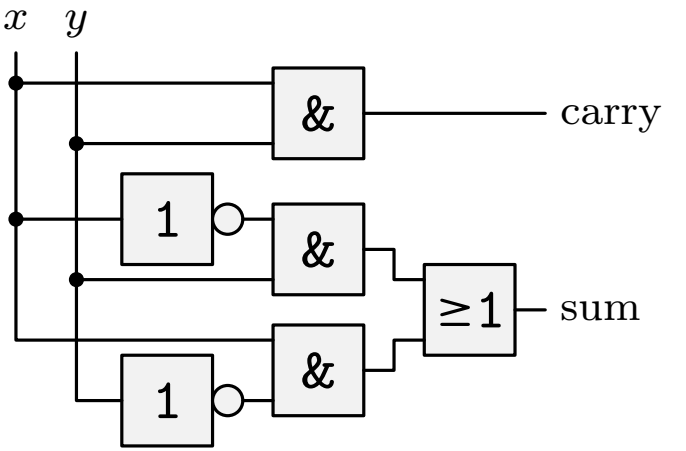
- Ein Ein-Bit-Addierer benötigt zwei Ausgänge: Die **Summe** (sum, s) und den **Übertrag** (carry, c).

- Die bitweise Addition kann offenbar durch (Logik-)Gatter implementiert werden.

Halbaddierer

- Während der Übertrag durch ein einfaches logisches Und (AND-Gatter) implementiert werden kann, ist die Summe das exklusive Oder der beiden Eingänge, das z.B. so dargestellt werden kann: $c = x \wedge y$ und $s = (x \wedge \overline{y}) \vee (y \wedge \overline{x})$.

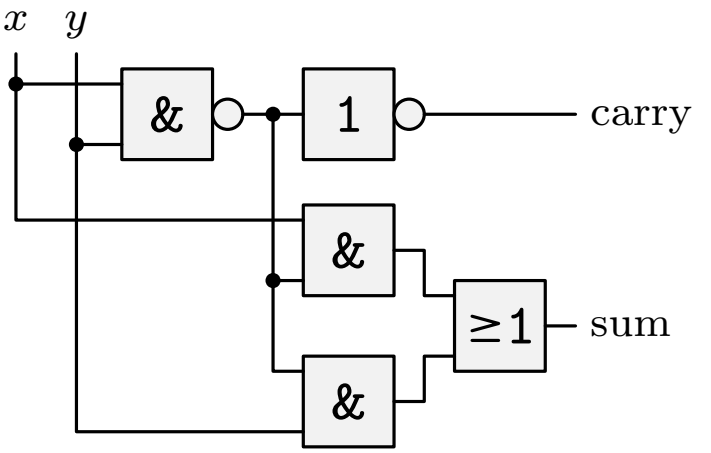
x	y	s	c
0	0	0	0
1	0	1	0
0	1	1	0
1	1	0	1



- Verknüpfung der beiden Funktionen erlaubt eine Implementierung mit weniger Gattern:

$$s = (x \wedge \overline{(x \wedge y)}) \vee (y \wedge \overline{(x \wedge y)})$$

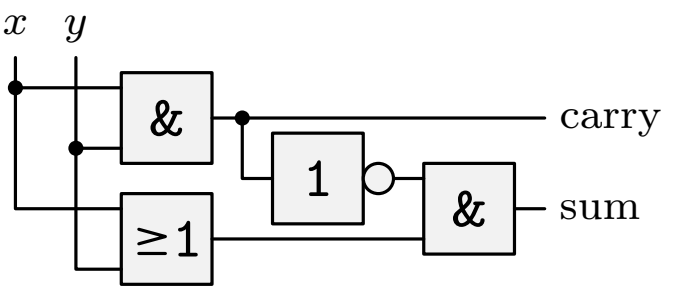
(Ableitbar über Distributivität, De Morgan, Komplementarität, Neutralität.)



Halbaddierer

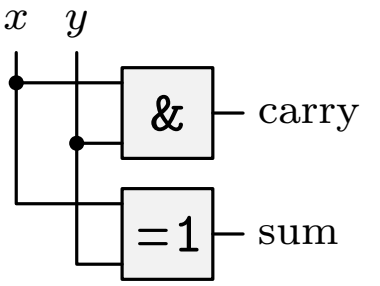
- Ausnutzen der Booleschen Gesetze zur Vereinfachung (Assoziativität und Kommutativität nicht explizit angegeben):

$$
\begin{aligned}
s &= (x \wedge \overline{y}) \vee (y \wedge \overline{x}) \\
&\overset{\text{Distributivität}}{=} ((x \wedge \overline{y}) \vee y) \wedge ((x \wedge \overline{y}) \vee \overline{x}) \\
&\overset{\text{Distributivität}}{=} (x \vee y) \wedge (\overline{y} \vee y) \wedge (x \vee \overline{x}) \wedge (\overline{y} \vee \overline{x}) \\
&\overset{\text{Komplementarität}}{=} (x \vee y) \wedge 1 \wedge 1 \wedge (\overline{y} \vee \overline{x}) \\
&\overset{\text{Neutralität}}{=} (x \vee y) \wedge (\overline{y} \vee \overline{x}) \\
&\overset{\text{De Morgan}}{=} (x \vee y) \wedge \overline{(y \wedge x)}
\end{aligned}
$$



- Verwenden eines expliziten XOR-Gatters:

$$
\begin{aligned}
s &= (x \wedge \overline{y}) \vee (\overline{x} \wedge y) \\
&= x \oplus y
\end{aligned}
$$



(„Optimierter“ Halbaddierer)

Volladdierer

- Um eine Addition nicht nur für ein Bit, sondern für n Bits, $n \geq 2$, zu implementieren, braucht man einen **Addierer mit drei Eingängen**.

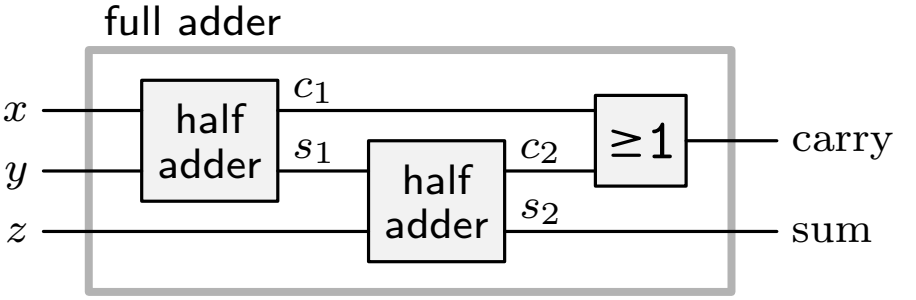
- Beispiel (binäre 4-Bit-Addition):

$$\begin{array}{rcccc} & 1 & 0 & 1 & 1 \\ + & {\scriptstyle 1}\,0_1 & 1_1 & 1_1 & 1 \\ \hline = & 0 & 0 & 1 & 0 \end{array}$$

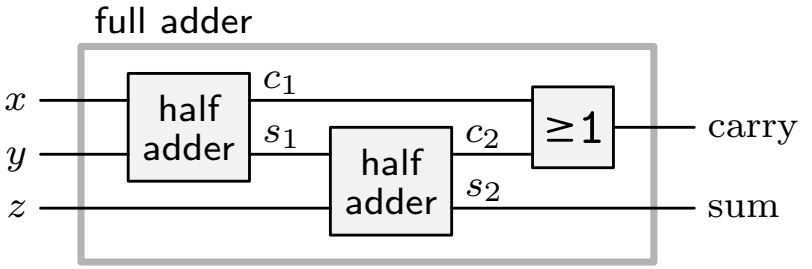
- Ein **Volladdierer** berücksichtigt den Übertrag einer vorangehenden Addition.

x	y	c_{in}	s	c_{out}
0	0	0	0	0
1	0	0	1	0
0	1	0	1	0
1	1	0	0	1
0	0	1	1	0
1	0	1	0	1
0	1	1	0	1
1	1	1	1	1

Man beachte, daß ein Volladdierer die Funktion $s = (x + y) + c_{\text{in}}$ berechnet. Er wird daher am einfachsten aus zwei Halbaddierern zusammengesetzt (mit $z = c_{\text{in}}$).

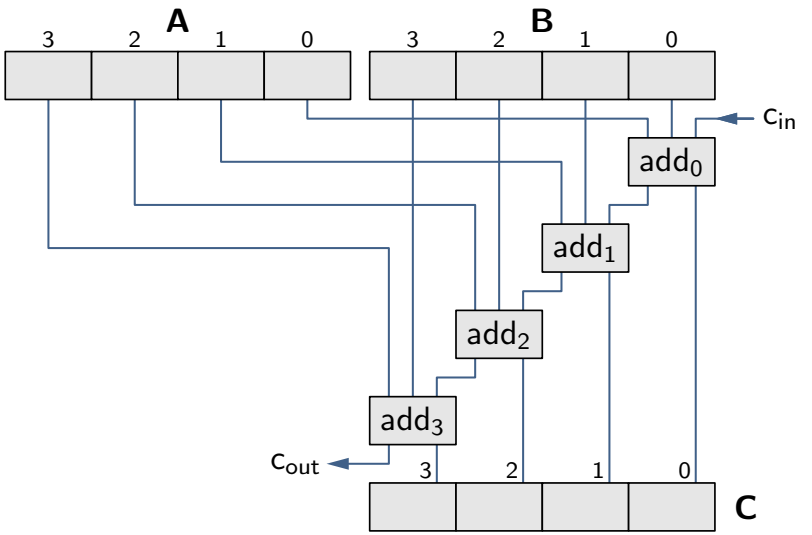


n-Bit-Übertragskette-Addierer



Prüfung der Funktion des Volladdierers:

$$s_1 = x \oplus y, \qquad c_1 = x \wedge y,$$
$$s_2 = z \oplus s_1, \qquad c_2 = z \wedge s_1,$$
$$\text{sum} = s_2, \qquad \text{carry} = c_1 \vee c_2.$$



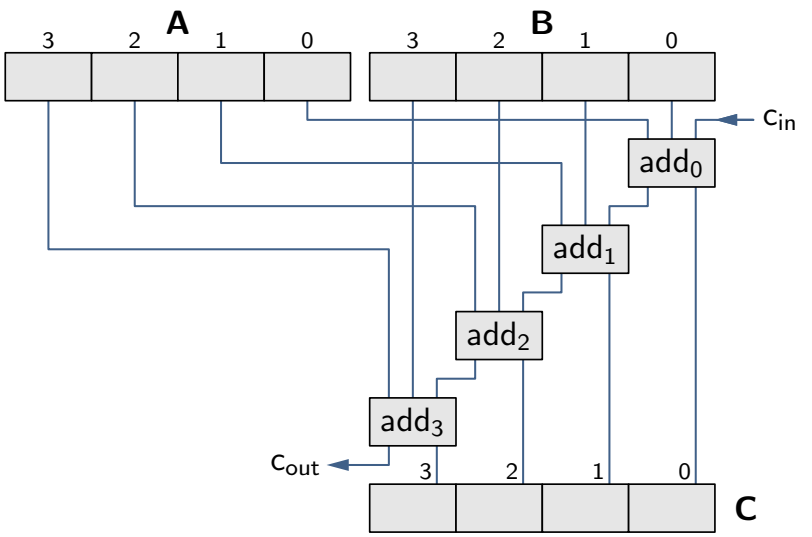
Ein **n-Bit-Übertragskette-Addierer** (carry ripple adder) für C = A + B kann leicht aus n Volladdierern zusammengesetzt werden.

Bemerkung: Der Addierer add_0 ist hier ebenfalls als Volladdierer ausgelegt (aus Gründen, die in Kürze dargelegt werden). Im Prinzip wäre an dieser Stelle ein Halbaddierer ausreichend.

- Ein n-Bit-Übertragskette-Addierer hat den jedoch Nachteil, daß sich ein Übertrag wellenartig durch die Addiererkette ausbreitet (carry ripple).

n-Bit-Übertragskette-Addierer

- Im Bauplan des n-Bit-Übertragskette-Addierers ist deutlich zu sehen, daß der Volladdierer add_k erst dann anfangen kann, seine Ausgabe zu berechnen, nachdem der Volladdierer add_{k-1} seine Berechnung abgeschlossen hat (insbesondere seinen Übertrag c_{out} berechnet hat).
- Die Überträge werden also **sequentiell** berechnet; sie pflanzen sich wellenartig durch die Addiererkette fort. (Daher der Name "carry ripple adder".)

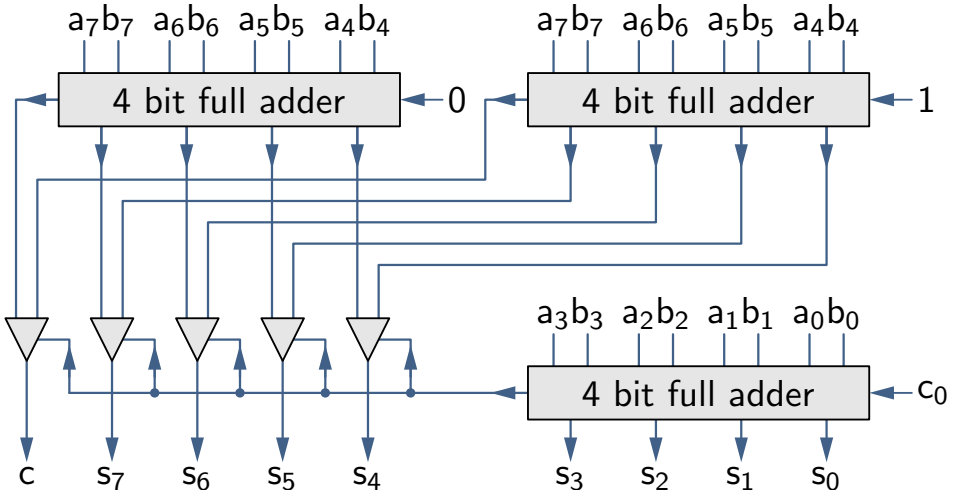


- Zwar sind Addierer recht einfache Schaltkreise und führen daher nur zu relativ kurzen Verzögerungen.
- Aber die Addition ist eine so wichtige Operation, daß jede Möglichkeit, sie so schnell wie möglich auszuführen, genutzt werden sollte.
- Ziel: Verkürzung der Signallaufzeit durch **parallele Berechnung**.

Optimierung: n-Bit-Übertragsauswahl-Addierer

- In einem **n-Bit-Übertragsauswahl-Addierer** (carry select adder) werden die Summen des unteren Halbwortes (Bits 0 bis $\frac{n}{2} - 1$) und des oberen Halbwortes (Bits $\frac{n}{2} - 1$ bis n) **parallel** berechnet.

- Da der Wert des Übertrags aus dem unteren Halbwort noch nicht bekannt ist, wenn die Summenbildung für das obere Halbwort beginnt, wird diese Summe zweimal, in zwei getrennten Schaltungen, berechnet, wobei die eine Schaltung $c_{in} = 0$ und die andere $c_{in} = 1$ annimmt (um beide möglichen Ergebnisse zu erhalten).



- Wenn der Übertrag des unteren Halbworts berechnet ist, wird er über einen Multiplexer zur Auswahl des richtigen oberen Halbwortes benutzt.
- Beachte: die beiden oberen 4-Bit-Addierer addieren die gleichen Bits, gehen aber von verschiedenen Überträgen aus.

Optimierung: n-Bit-Übertragsauswahl-Addierer

- Ein Übertragsauswahl-Addierer arbeitet durch die halbe Addiererkettenlänge fast doppelt so schnell wie ein Übertragungskette-Addierer
(nicht ganz doppelt wegen der zusätzlichen Laufzeit durch den Multiplexer.)

- Ein Übertragsauswahl-Addierer benötigt etwas über 50% zusätzliche Schaltung (Addierer für oberes Halbwort doppelt, Multiplexer zusätzlich).

z	x	y	o
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

- Der 2-Bit-Multiplexer schaltet abhängig von einer Steuerleitung z entweder den Eingang x oder den Eingang y auf seinen Ausgang.
Minimierungsergebnis:

$$o = (y \wedge z) \vee (x \wedge \overline{z}).$$

- Im Übertragsauswahl-Addierer ist die Steuerleitung z der Übertrag aus der Addition des unteren Halbwortes
($z = c_{\text{out}}$ des unteren Addierers).

- Bei längeren Binärzahlen wird das Prinzip des Übertragsauswahl-Addierers rekursiv angewandt (d.h., die Halbwörter werden ihrerseits zerlegt).

Darstellung negativer Zahlen

- Im Dezimalsystem zeigt ein vorangestelltes **Minuszeichen** an, daß eine Zahl negativ ist, während die Zahl selbst unverändert bleibt.

- Wünschenswerte Eigenschaften einer Darstellung **negativer Binärzahlen**:
  - Keine Einführung eines neuen Zeichens, damit weiterhin allein mit 0 und 1 gerechnet werden kann.
  - Einfacher Test, ob eine Zahl Null, positiv oder negativ ist.
  - Einfache Negation (Bilden der negierten Zahl mit gleichem Betrag) und einfache Subtraktion (z.B. über Addition der Negation).
  - Wenn möglich, keine Festlegung der Stellenzahl, z.B. auf n Binärstellen.

- Wir prüfen drei “naheliegende” Möglichkeiten, negative Zahlen darzustellen:
  - **Betrag und Vorzeichen**: Ein Bit ist ein Vorzeichen (z.B. $0 \mathrel{\widehat{=}} +$, $1 \mathrel{\widehat{=}} -$).
  - **Einerkomplement**: Alle n Bits einer Zahl werden invertiert ($0 \leftrightarrow 1$), um das Vorzeichen der Zahl zu wechseln.
  - **Zweierkomplement**: Zum Einerkomplement wird 1 addiert, um das Vorzeichen der Zahl zu wechseln.

Darstellung negativer Zahlen

Wert		dezimale Interpretation			
bin.	hex.	vzl.	BuV	1k	2k
0000	0	0	0	0	0
0001	1	1	1	1	1
0010	2	2	2	2	2
0011	3	3	3	3	3
0100	4	4	4	4	4
0101	5	5	5	5	5
0110	6	6	6	6	6
0111	7	7	7	7	7
1000	8	8	-0	-7	-8
1001	9	9	-1	-6	-7
1010	A	10	-2	-5	-6
1011	B	11	-3	-4	-5
1100	C	12	-4	-3	-4
1101	D	13	-5	-2	-3
1110	E	14	-6	-1	-2
1111	F	15	-7	-0	-1

Beispiel zur Darstellung negativer Binärzahlen

- Vergleich der Interpretation einer 4-Bit-Zahl (auch “nibble” genannt) als vorzeichenloser Wert („vzl.“), als Betrag und Vorzeichen („BuV“), im Einerkomplement („1k“) und im Zweierkomplement („2k“).
- Man beachte das Auftreten einer Zahl „-0“ (verschieden von 0!) in der Darstellung als Betrag und Vorzeichen und im Einerkomplement.
- Man beachte weiter den größeren, aber asymmetrischen Zahlenbereich beim Zweierkomplement (-8 bis $+7$ statt nur -7 bis $+7$).

Negative Zahlen: Betrag und Vorzeichen

- In der Darstellung mit Betrag und Vorzeichen
  ist ein **Vorzeichenbit** ausgezeichnet (i.a. das höchstwertigste Bit).

  Damit dieses Bit leicht identifiziert und verarbeitet werden kann,
  sollten alle Zahlen die gleiche Stellenzahl aufweisen.

- Anderenfalls: Bei einer Erweiterung der interpretierten Stellen einer Zahl muß
  dieses Vorzeichenbit identifiziert und ggf. an eine andere Stelle versetzt werden.
  $\rightarrow$ zusätzliche Steuerlogik nötig.

- Beim Einer- und Zweierkomplement muß dagegen nur mit dem Wert
  des höchstwertigsten Bits erweitert werden, um den Wert zu erhalten.

Beispiel:

	binär	BuV	1k	2k
4-Bit-Zahl	1010	-2	-5	-6
Erweiterung mit 0	01010	$+10$	$+10$	$+10$
Erweiterung mit 1	11010	-10	-5	-6

- Folglich ist die Darstellung durch Betrag und Vorzeichen weniger geeignet,
  da oft Zahlen unterschiedlicher Stellenzahl verarbeitet werden müssen.

Negative Zahlen: Einerkomplement

- Das Einerkomplement scheint eine sehr einfache Subtraktion zu erlauben, nämlich Komplementbildung gefolgt von einer das Vorzeichen ignorierenden Addition.

  Beispiel:

$$\begin{array}{rcccc} & 1 & 0 & 1 & 1 \\ + & 0 & 0_1 & 1_1 & 1 \\ \hline = & 1 & 1 & 1 & 0 \end{array} \quad \begin{array}{l} (-4)_{10} \\ (+3)_{10} \\ \\ (-1)_{10} \end{array}$$

- Dies gilt allerdings nur, solange es nicht zu einem Nulldurchgang kommt.

  Gegenbeispiel:

$$\begin{array}{rcccc} & 1 & 0 & 1 & 1 \\ + \ {\scriptstyle 1} & 0_1 & 1_1 & 1 & 0 \\ \hline = & 0 & 0 & 0 & 1 \end{array} \quad \begin{array}{l} (-4)_{10} \\ (+6)_{10} \\ \\ (+1)_{10} \end{array}$$

$$-4 \xrightarrow{1} -3 \xrightarrow{2} -2 \xrightarrow{3} -1 \xrightarrow{4} -0 \xrightarrow{5} 0 \xrightarrow{6} 1$$

  Bei einem Nulldurchgang (beim Hochzählen) ist das Ergebnis um 1 zu klein, was auch durch einen Übertrag in der höchstwertigsten Stelle angezeigt wird.
  Man muß daher den Übertrag der obersten Stelle zur untersten addieren.

- Grund dieses Effekts ist die Existenz einer -0, die außerdem den Nachteil hat, daß sie die Behandlung der Null als wohl wichtigster Zahl verkompliziert (speziell: Test auf 0). Außerdem: unnötige Redundanz.

Negative Zahlen: Zweierkomplement

- Die geschilderten Probleme werden durch das Zweierkomplement vermieden.

- Subtraktion kann durch Addition des Komplementes ausgeführt werden, wobei das Vorzeichen einfach ignoriert wird (wie beim Einerkomplement). Aber: Beide betrachteten Situationen führen hier zu keinen Problemen.

Beispiele:

$$\begin{array}{rcccc} & 1 & 1 & 0 & 0 \\ + & 0 & 0_1 & 1_1 & 1 \\ \hline = & 1 & 1 & 1 & 1 \end{array} \qquad \begin{array}{c} (-4)_{10} \\ (+3)_{10} \\ \\ (-1)_{10} \end{array} \qquad \begin{array}{rcccc} & 1 & 1 & 0 & 0 \\ + \; 1 & 0_1 & 1 & 1 & 0 \\ \hline = & 0 & 0 & 1 & 0 \end{array} \qquad \begin{array}{c} (-4)_{10} \\ (+6)_{10} \\ \\ (+2)_{10} \end{array}$$

- Formal kann das Zweierkomplement so definiert werden:

$$[d_{n-1} d_{n-2} \ldots d_0]_{2K} \qquad \text{wird interpretiert als} \qquad d_{n-1} \cdot (-2^{n-1}) + \sum_{i=0}^{n-2} d_i \cdot 2^i .$$

Beispiel: $[1\ 000\ 1000]_{2K} = -2^7 + 2^3 = -128 + 8 = -120$

- Diese Darstellung negativer Zahlen ist sehr gut geeignet, um Schaltungen zur Berechnung der Subtraktion zu entwerfen. Sie hat aber auch Vorteile bei der Implementierung der Multiplikation.

Negative Zahlen: Zweierkomplement

- **Formale Definition des Zweierkomplementes:**

$$[d_{n-1}d_{n-2}\ldots d_0]_{2K} \qquad \text{wird interpretiert als} \qquad d_{n-1}\cdot(-2^{n-1}) + \sum_{i=0}^{n-2} d_i \cdot 2^i.$$

- In Zweierkomplementdarstellung umfaßt der Wertebereich n-stelliger Zahlen folglich die Werte $-2^{n-1}, \ldots, 0, \ldots, 2^{n-1}-1$ (wegen $\sum_{i=0}^{k} 2^i = 2^{k+1}-1$), z.B.

Stellenzahl n	kleinster darstellbarer Wert	größter darstellbarer Wert
8	-128_{10}	$+127_{10}$
16	-32768_{10}	$+32767_{10}$
32	-2147483648_{10}	$+2147483647_{10}$
64	$-9223372036854775808_{10}$	$+9223372036854775807_{10}$

- Eigenschaften:
  - Null: Nur eine Darstellung, nämlich $[0\ldots0]_{2K}$.
  - Negation: Negiere jedes Bit ($0 \leftrightarrow 1$), dann addiere 1.
  - Addition: Vorzeichen braucht nicht beachtet zu werden.
  - Subtraktion: $A - B = A + (-B)$.

Zweierkomplement und Negation

Negationsregel des Zweierkomplementes:
Negiere jedes Bit ($0 \leftrightarrow 1$), dann addiere 1.

Warum funktioniert diese Negationsregel?

- Gegeben sei: $A = a_{n-1}a_{n-2}\ldots a_1a_0$.
  Zweierkomplement: $-A = \overline{a}_{n-1}\overline{a}_{n-2}\ldots \overline{a}_1\overline{a}_0 + 1$.

Erinnerung:

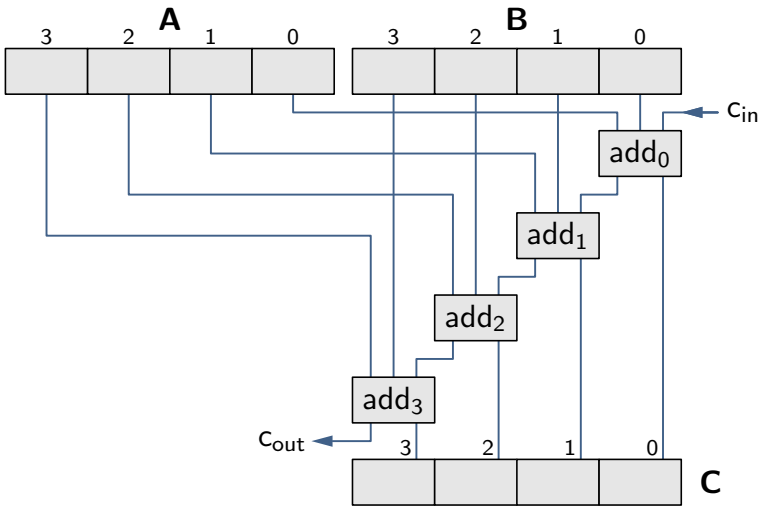
$$\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$$

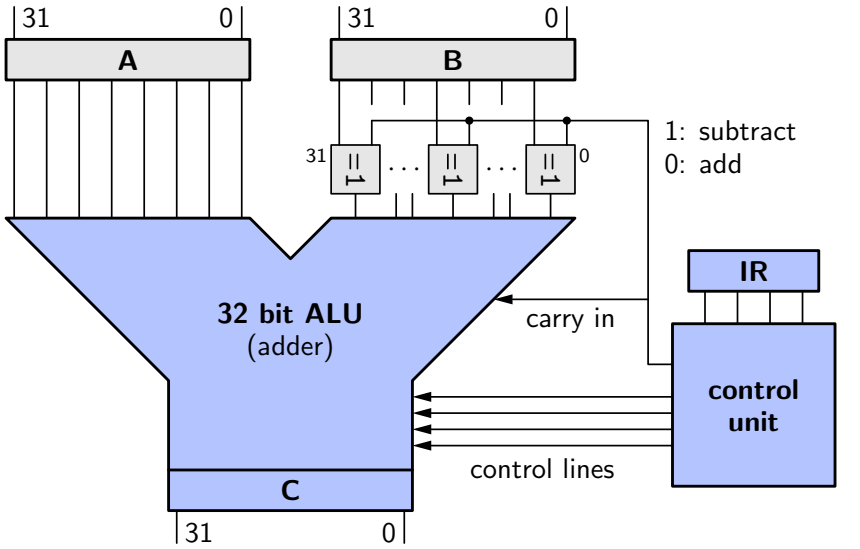
- Berechne $A - A = A + (-A)$, um zu prüfen, dass tatsächlich 0 herauskommt:

$$\begin{aligned}
&A + (-A)\\
&= \underbrace{a_{n-1}\cdot(-2^{n-1}) + \sum_{i=0}^{n-2} a_i\cdot 2^i}_{A} + \underbrace{\overline{a}_{n-1}\cdot(-2^{n-1}) + \sum_{i=0}^{n-2} \overline{a}_i\cdot 2^i + 1}_{-A}\\
&= \underbrace{(a_{n-1} + \overline{a}_{n-1})}_{=1}\cdot(-2^{n-1}) + \sum_{i=0}^{n-2} \underbrace{(a_i + \overline{a}_i)}_{=1}\cdot 2^i + 1\\
&= -2^{n-1} + \sum_{i=0}^{n-2} 2^i + 1 = -2^{n-1} + (2^{n-1} - 1) + 1 = 0.
\end{aligned}$$

Zweierkomplement: n-Bit-Addierer/Subtrahierer

Idee: $A - B = A + (-B)$





- Ein **n-Bit-Subtrahierer** besteht aus einem n-Bit-Addierer und Gattern, die die Negationsregel implementieren: Negiere jedes Bit ($0 \leftrightarrow 1$), dann addiere 1.
- Ein XOR-Gatter kann als steuerbarer Negierer dienen:
- Übertragseingabe Addierer add_0: Addition: $c_{in} = 0$, Subtraktion: $c_{in} = 1$.

X	N	$X \oplus N$
x	0	x
x	1	$\overline{x}$

Binärarithmetik: Über-/Unterlauf

- Bei einer festen Stellenzahl n (feste Anzahl von Bits) kann eine Addition (oder eine Subtraktion) zu einem **Überlauf** oder einem **Unterlauf** führen: Das Ergebnis der Rechnung liegt außerhalb des Bereichs $-2^{n-1} \ldots 2^{n-1} - 1$.

- Beispiele: ($n = 4$, Zahlenbereich $-8 \ldots 7$, kein Über-/Unterlauf)

	0	0	1	0	$(+2)_{10}$		0	1	0	1	$(+5)_{10}$		1	0	1	1	$(-5)_{10}$
+	1	0	0	1	$(-7)_{10}$	+	1	1	1	0	$(-2)_{10}$	+	1	1	1	0	$(-2)_{10}$
=	1	0	1	1	$(-5)_{10}$	=	0	0	1	1	$(+3)_{10}$	=	1	0	0	1	$(-7)_{10}$

- Beispiele: (Über-/Unterlauf, Ergebnis unbrauchbar)

	0	1	1	1	$(+7)_{10}$		1	0	1	0	$(-6)_{10}$		0	1	1	1	$(+7)_{10}$
+	0	1	1	1	$(+7)_{10}$	+	1	1	0	0	$(-4)_{10}$	+	0	0	0	1	$(+1)_{10}$
=	1	1	1	0	$(-2)_{10}$	=	0	1	1	0	$(+6)_{10}$	=	1	0	0	0	$(-8)_{10}$

- **Über-/Unterlaufregel:** Ein Überlauf (bzw. Unterlauf) tritt auf, wenn beide Operanden positives (bzw. negatives) Vorzeichen haben, aber das Ergebnis ein negatives (bzw. positives) Vorzeichen hat.

Multiplikation mit Potenzen der Basis

- Eine Multiplikation mit einer Potenz der Basis b des Zahlensystems ist einfach:

  Eine Multiplikation mit b^k verschiebt die Ziffern um k Stellen nach links
  (d.h. Verschiebung in Richtung auf die höherwertigen Stellen);
  die freiwerdenden niederwertigen Stellen werden auf Null gesetzt.

  allgemein:
  $$[d_{n-1} \ldots d_0]_b \cdot b^k = [d_{n-1} \ldots d_0 \underbrace{0 \ldots 0}_{k \text{ Nullen}}]_b$$

  Beispiele:
  $$482_{10} \cdot 10^2 = 48200_{10}$$
  $$10101_2 \cdot 2^3 = 10101000_2$$

- Analog gilt für die Division durch eine Potenz der Basis b:

  Eine Division durch b^k verschiebt die Ziffern um k Stellen nach rechts
  (d.h. Verschiebung in Richtung auf die niederwertigen Stellen).

  allgemein:
  $$[d_{n-1} \ldots d_0]_b \div b^k = [d_{n-1} \ldots d_k]_b$$

  Beispiele:
  $$482_{10} \div 10^2 = 4_{10}$$
  $$10101_2 \div 2^3 = 10_2$$

  (Aber Achtung: Dies liefert nur den ganzzahligen Teil der Division!)

Multiplikation mit Potenzen der Basis

- Man beachte, daß die Division durch b^k äquivalent ist zur Multiplikation mit b^{-k}.

  allgemein: $[d_{n-1} \ldots d_0]_b \cdot b^{-k} = [d_{n-1} \ldots d_k]_b$

- Für den Rest der Division einer Zahl durch eine Potenz der Basis b gilt:

  Der Rest einer Division (modulo) durch b^k sind die letzten k Stellen der Zahl.

  allgemein: $[d_{n-1} \ldots d_0]_b \mod b^k = [d_{k-1} \ldots d_0]_b$

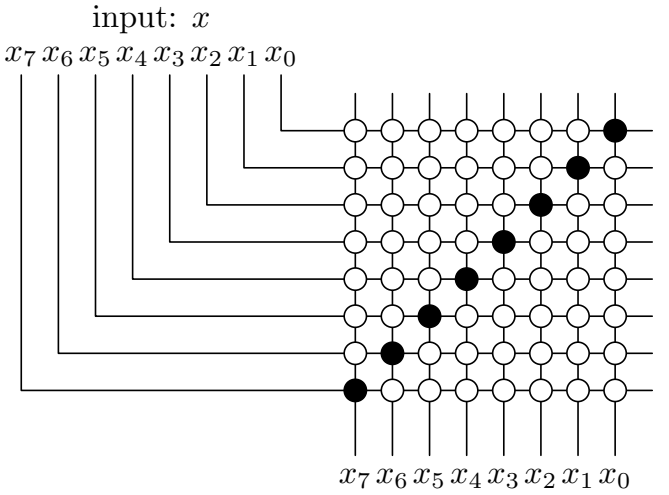
  Beispiele:
  $$\begin{aligned} 482_{10} \mod 10^2 &= 82_{10} \\ 10101_2 \mod 2^3 &= 101_2 \end{aligned}$$

- Für $b = 2$ (Binärsystem) entsprechen Multiplikation und Division mit 2^k dem Verschieben der Bits einer Zahl um k Stellen nach links bzw. rechts.

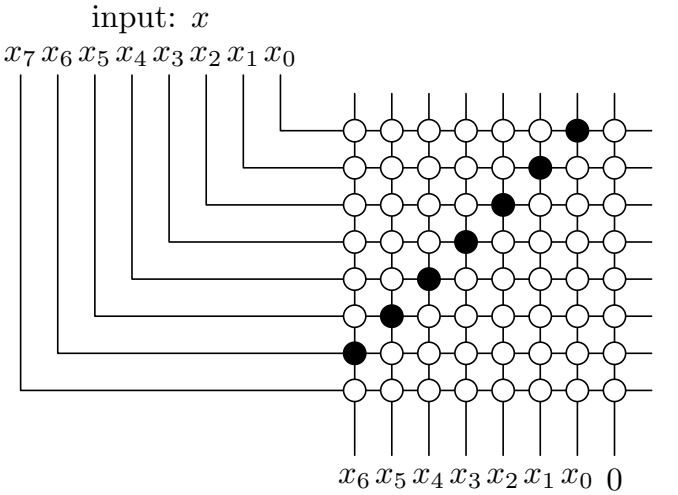
- Dieses Verschieben kann leicht durch einen **Kreuzungspunktschalter** (crosspoint switch oder crossbar switch) implementiert werden.

  Kreuzungspunktschalter bestehen aus einer $n \times n$ Matrix von Schaltern, durch die gekreuzte Leiterbahnen gezielt verbunden werden können.

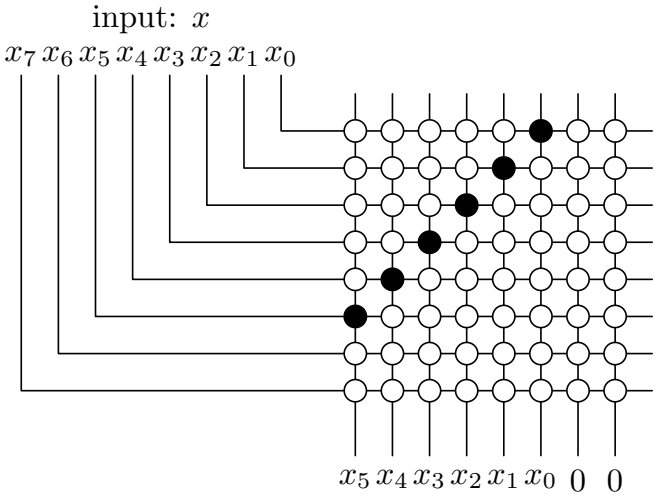
Multiplikation mit Zweierpotenzen: Bit-Schieben



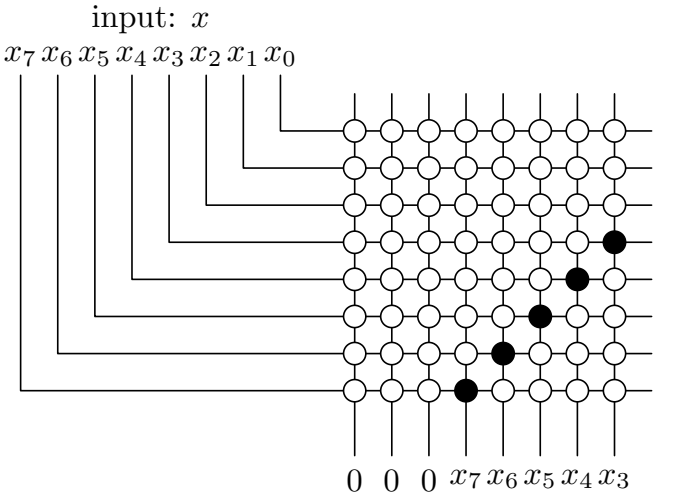
multiplication by 1 output: x



multiplication by 2 output: x <<= 1



multiplication by 4 output x <<= 2



division by 8 output x >>= 3

Allgemeine binäre Multiplikation

- Im Prinzip könnte eine Multiplikation $a \cdot b$ ausgeführt werden, indem man $(b-1)$-mal den Wert von a zu a hinzuaddiert.

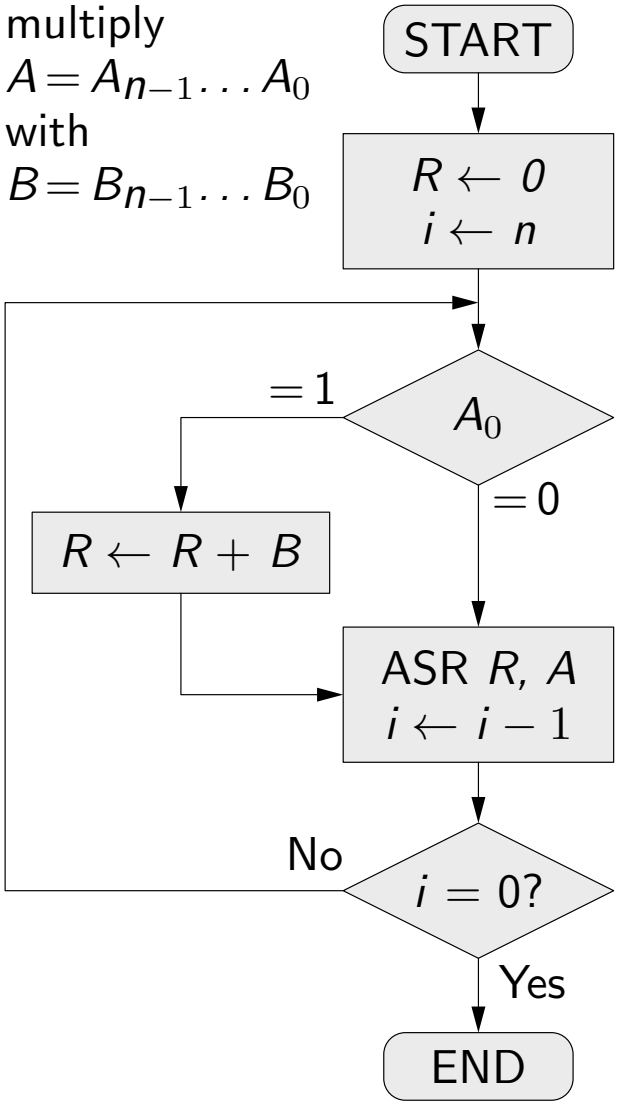
  Diese Methode ist jedoch offensichtlich höchst ineffizient und nicht zu empfehlen.

- Stattdessen wird der Grundschulalgorithmus (siehe frühere Erinnerung) des **Addierens von Stellenprodukten** im Binärsystem angewandt. Die Stellenprodukte werden durch **Bit-Schieben** berechnet.

				1	1	0	1			$(13)_{10}$
$\times$				1	0	1	0			$(10)_{10}$
				1	0	1	0		1	$(10)_{10}$
$+$			0	0	0	0			0	$(0)_{10}$
$+$		1	0	1	0				1	$(40)_{10}$
$+$	1	0	1	0					1	$(80)_{10}$
$=$	1	0	0	0	0	0	1	0		$(130)_{10}$

- Es können nur zwei Stellenprodukte auftreten: 0 und der Multiplikator.
  Die Multiplikation zweier n-Bit-Zahlen ergibt eine $2n$-Bit-Zahl.

Multiplikation: Standardalgorithmus



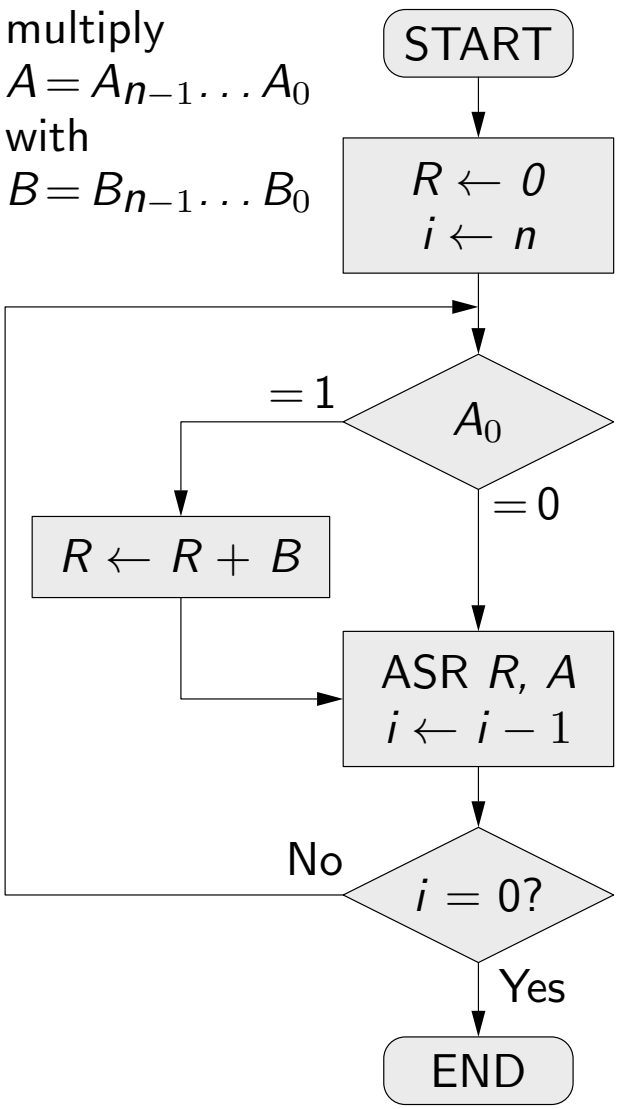
- **Standard-Multiplikationsalgorithmus**:
  Es wird der aus der Grundschule bekannte Multiplikationsalgorithmus, der ein Produkt über eine Summe von Stellenprodukten berechnet, im Binärsystem ausgeführt.

- Ist die k-te Stelle des Faktors A Eins ($A_k = 1$), so wird der Faktor B, multipliziert mit dem Stellenwert 2^k, auf das Ergebnis aufaddiert.

- Da stets nur n Stellen des Ergebnisses angesprochen werden, obwohl am Ende $2n$ Stellen entstehen, werden R und A im höher- bzw. niederwertigen Teil eines $2n$-Bit-Registers abgelegt.

- Die Multiplikation mit 2^k wird durch ein bitweises Rechtsschieben des späteren Ergebnisses R erreicht. Dadurch muß stets nur A_0 getestet werden (least significant bit, LSB).

Multiplikation: Standardalgorithmus



- Implementierung in Java:

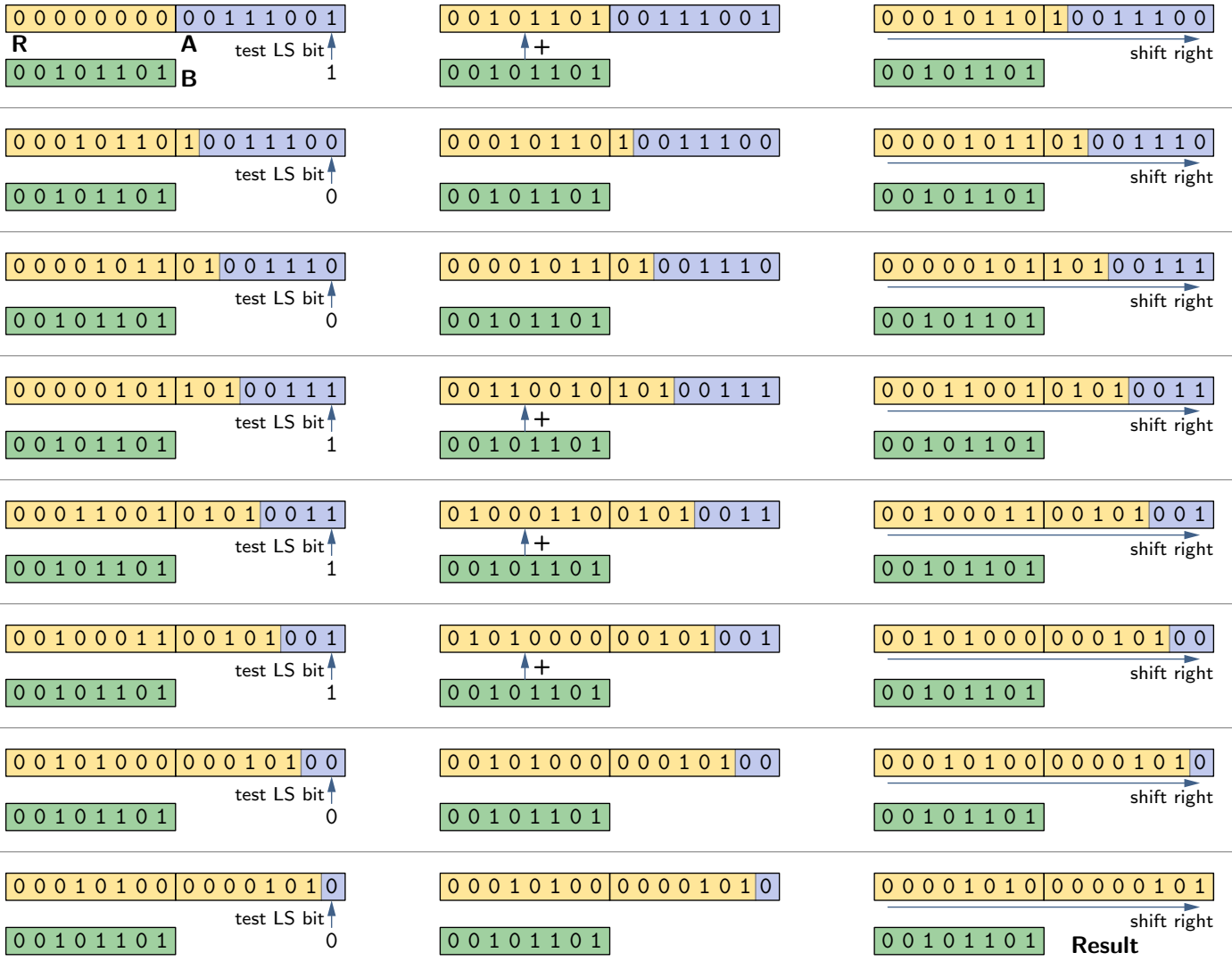
  Multiplikation $a \cdot b$ zweier 16-Bit-Zahlen (32-Bit-Ergebnis) unter Verwendung von Bit-Schieben („<<“, „>>“), Bit-Testen („&“) und Addition („+“).

```
static int multiply (int a, int b) {
    int i;
    b = b << 16;
    for (i = 16; --i >= 0; ) {
        if ((a & 1) != 0)
            a = a + b;
        a = a >> 1;
    }
    return a;
}
```

- Beachte: Allgemein liefert die Multiplikation zweier n-Bit-Zahlen ein $2n$-Bit-Ergebnis.

- Beachte: in der trickreichen Java-Implementation werden die oberen 16 Bit der Variable `a` verwendet, um `R` zu speichern. Die Implementation klappt also nur, wenn die oberen 16 Bit von `a` beim Aufruf alle `0` sind. Besser: bei Start auf `0` setzen, oder Fehlermeldung.

Multiplikation: Standardalgorithmus (8 Bit)



Multiplikation: Negative Zahlen

- Für die Addition und Subtraktion funktioniert die Darstellung negativer Zahlen über das **Zweierkomplement** sehr gut.
  Wie steht es mit der Multiplikation?

- Beispiel: (Zahlen als Zweierkomplementdarstellung interpretiert)

				0	1	1	1		$(+7)_{10}$
$\times$				1	0	1	1		$(-5)_{10}$
				1	0	1	1	1	
$+$			1	0	1	1		1	
$+$		1	0	1	1			1	
$+$	0	0	0	0				0	
$=$	0	1	0	0	1	1	0	1	$(77)_{10}$

- Man beachte: Da die beiden Operanden je 4 Bit haben, muß das Ergebnis 8 Bit haben, daher die führende 0 im Ergebnis.

- Dieses Multiplikationsschema schlägt fehlt, wenn einer der Operanden negativ ist. (Beachte: 1011 vorzeichenlos interpretiert ist 11_{10}.)

Multiplikation: Negative Zahlen

- Was ist der Grund für das Scheitern des Multiplikationsschemas?

- Die Multiplikation zweier n-Bit Zahlen erfordert die Addition von $2n$-Bit Zahlen (da das Ergebnis $2n$ Bit hat).

- Es passiert hier also tatsächlich (implizit) folgendes:

	0	0	0	0	0	1	1	1		$(+7)_{10}$
$\times$	0	0	0	0	1	0	1	1		$(-5)_{10}$
	0	0	0	0	1	0	1	1	1	$(11)_{10}$
$+$	0	0	0	1	0	1	1	0	1	$(22)_{10}$
$+$	0	0	1	0	1	1	0	0	1	$(44)_{10}$
$+$	0	0	0	0	0	0	0	0	0	$(0)_{10}$
$=$	0	1	0	0	1	1	0	1		$(77)_{10}$

- Das heißt: Interpretiert als 8-Bit-Zahlen sind die verschobenen Stellenprodukte positive Zahlen (und nicht negative wie erwartet).

- Deshalb: Stellenprodukte auf 8 Bit erweitern, so daß sie negativ werden.

Multiplikation: Negative Zahlen

- Um eine im Zweierkomplement interpretierte Zahl von n auf m Bit, $m > n$, zu **erweitern**, müssen die höherwertigen Stellen passend aufgefüllt werden: Die neuen $m - n$ Stellen bekommen den **Wert des Vorzeichenbits**.

  Bemerkung: Obwohl es im Zweierkomplement eigentlich kein explizites Vorzeichenbit gibt, beginnen doch alle positiven Zahlen mit 0 und alle negativen Zahlen mit 1.
  Das führende Bit (also d_{n-1}) kann folglich als eine Art Vorzeichenbit angesehen werden.

- Beispiel: $-5_{10} = [\underbrace{1}_{\text{„Vorzeichen“}}011]_{2K} = [\underbrace{1111}_{\text{neue Stellen}}1011]_{2K} = -5_{10}$

- 4-Bit-Multiplikation (durch Erweiterung auf 8 Bit korrigiert):

	0	0	0	0	0	1	1	1		$(+7)_{10}$
$\times$	1	1	1	1	1	0	1	1		$(-5)_{10}$
	1	1	1	1	1	0	1	1	1	$(-5)_{10}$
$+$	1	1	1	1	0	1	1	0	1	$(-10)_{10}$
$+$	1	1	1	0	1	1	0	0	1	$(-20)_{10}$
$+$	0	0	0	0	0	0	0	0	0	$(0)_{10}$
$=$	1	1	0	1	1	1	0	1		$(-35)_{10}$

Zweierkomplement und Stellenerweiterung

Erweiterungsregel des Zweierkomplementes:
Zur Erweiterung einer n-Bit-Binärzahl auf eine m-Bit-Binärzahl, $m > n$,
stelle der n-Bit-Binärzahl $(m - n)$-mal das führende Bit voran.

Warum funktioniert diese Erweiterungsregel?

- Wir betrachten der Einfachheit halber eine Erweiterung von n auf $n + 1$ Bit.
  Die Verallgemeinerung auf m Bit ergibt sich durch einfache Induktion.

- Im Zweierkomplement gilt: $[a_{n-1}a_{n-2}\ldots a_0]_{2K} = a_{n-1} \cdot (-2^{n-1}) + \sum_{i=0}^{n-2} a_i \cdot 2^i$.

  - **1. Fall:** $a_{n-1} = 0$

$$[0\, a_{n-2}\ldots a_0]_{2K} = \sum_{i=0}^{n-2} a_i \cdot 2^i = [0\,0\, a_{n-2}\ldots a_0]_{2K}.$$

  - **2. Fall:** $a_{n-1} = 1$

$$\begin{aligned}[1\, a_{n-2}\ldots a_0]_{2K} &= -2^{n-1} + \sum_{i=0}^{n-2} a_i \cdot 2^i \\ &= -2^n + 2^{n-1} + \sum_{i=0}^{n-2} a_i \cdot 2^i = [1\,1\, a_{n-2}\ldots a_0]_{2K}.\end{aligned}$$

Multiplikation: Negative Zahlen

- Im Fall einer Multiplikation $\langle$positive Zahl$\rangle \cdot \langle$negative Zahl$\rangle$
  konnten wir den Standardalgorithmus (der sonst in diesem Fall fehlschlägt)
  durch Erweiterung der Stellenprodukte von n auf $2n$ Stellen korrigieren.

- Aber wie sieht es bei einer Multiplikation $\langle$negative Zahl$\rangle \cdot \langle$positive Zahl$\rangle$
  oder gar einer Multiplikation $\langle$negative Zahl$\rangle \cdot \langle$negative Zahl$\rangle$ aus, also bei
  einer Multiplikation, bei der der Faktor, dessen Stellen durchlaufen werden,
  um die Stellenprodukte zu bilden, negativ ist?

  $\Rightarrow$ Übungsaufgabe, Blatt 7

  Diese Übungsaufgabe macht deutlich,
  daß eine Erweiterung der Stellenprodukte allein nicht immer ausreicht.

- Es ist daher angebracht, nach einer anderen Methode zu suchen,
  die mit negativen Zahlen und speziell den bei Erweiterungen negativer Zahlen
  auf mehr Stellen auftretenden führenden Einsen besser umgehen kann.

- Ein solches Verfahren ist **Booths Algorithmus**.

  [Andrew Donald Booth 1951]

Multiplikation: Booths Algorithmus

- Die Kernidee von **Booths Algorithmus** ist es, den ersten Faktor im Produkt $a \cdot b$ in der Form
$$a = (a_1 - a_2) + (a_3 - a_4) + \cdots + (a_{k-1} - a_k)$$
zu zerlegen, wobei die Zahlen a_i Zweierpotenzen mit streng monoton fallenden Exponenten sind. Dass dies immer geht, sehen wir demnächst.

- Die Multiplikation $a \cdot b$ wird dabei umgeformt zu
$$a \cdot b = a_1 b - a_2 b + a_3 b - a_4 b + \cdots + a_{k-1} b - a_k b.$$

- Warum ist das nützlich? Eine Differenz von Zweierpotenzen entspricht einer Binärzahl mit genau einer Kette von Einsen (siehe folgende Folie). Alle Einserketten in der Zahl a werden daher als Differenzen von Zweierpotenzen codiert.

- Vorteile gegenüber dem normalen Multiplikationsalgorithmus daher: Eine einzelne Einserkette führt in Booths Algorithmus nicht zu so vielen Additionen, wie die Kette lang ist (so wie beim Standardalgorithmus), sondern nur zu höchstens zwei.

- Wegen dieser Eigenschaft erlaubt Booths Algorithms eine effiziente Multiplikation von Binärzahlen im Zweierkomplement (Vorzeichen brauchen nicht beachtet zu werden).

Multiplikation: Booths Algorithmus

- Wegen $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$ gilt natürlich auch (Summe muß nicht bei 0 anfangen):

$$\sum_{i=k}^{n} 2^i \;=\; 2^k \sum_{i=0}^{n-k} 2^i \;=\; 2^k \cdot (2^{n-k+1} - 1) \;=\; 2^{n+1} - 2^k.$$

- In der Binärdarstellung von Zahlen gilt daher allgemein:

$$[\ldots 0 \overbrace{1}^{n\text{-te Stelle}} \underbrace{1 \ldots 1}_{n-k+1 \text{ Einsen}} \overbrace{1}^{k\text{-te Stelle}} 0 \ldots 0]_2$$

$$= [\ldots \mathbf{1} \overbrace{0}^{n\text{-te Stelle}} \underbrace{0 \ldots 0}_{n-k+1 \text{ Nullen}} \overbrace{0}^{k\text{-te Stelle}} 0 \ldots 0]_2 - [\ldots 0 \overbrace{0}^{n\text{-te Stelle}} \underbrace{0 \ldots 0}_{n-k \text{ Nullen und eine Eins}} \overbrace{\mathbf{1}}^{k\text{-te Stelle}} 0 \ldots 0]_2,$$

- Booths Algorithmus zerlegt einen Faktor folglich in **Abschnitte von Einsen** statt jede Eins einzeln zu behandeln. Er benötigt daher nur so viele Additionen, wie es Wechsel zwischen 0 und 1 und umgekehrt in dem Faktor *A* gibt.

- Der Standardalgorithmus benötigt so viele Additionen, wie es Einsen gibt.

Multiplikation: Booths Algorithmus

- Wie man sich leicht klar macht, gilt sogar allgemein, d.h. für jede Basis b:

$$\sum_{i=k}^{n} (b-1) \cdot b^i \;=\; b^{n+1} - b^k.$$

- Folglich gilt in der Zahlendarstellung mit Basis b allgemein:

$$[\ldots 0 \overbrace{(b-1)}^{n\text{-te Stelle}} \underbrace{(b-1) \ \ldots \ (b-1)}_{(n-k+1)\text{-mal } (b-1)} \overbrace{(b-1)}^{k\text{-te Stelle}} 0 \ \ldots \ 0]_b$$

$$= [\ldots \mathbf{1} \underbrace{0 \ 0 \ \ldots \ 0}_{n-k+1 \text{ Nullen}} 0 \ \ldots \ 0]_b - [\ldots 0 \underbrace{0 \ 0 \ \ldots \ 0}_{n-k \text{ Nullen und eine Eins}} \mathbf{1} \ 0 \ \ldots \ 0]_b,$$

Beispiele:

$$99900_{10} = 100000_{10} - 100_{10}$$
$$77770_8 = 100000_8 - 10_8$$

- Die vorher betrachtete Regel für das Binärsystem ergibt sich aus dieser allgemeinen Regel als Spezialfall für $b = 2$, folglich mit $b - 1 = 1$.

Multiplikation: Booths Algorithmus

- 4-Bit-Multiplikation (mit Booths Algorithmus analog zu Grundschulalgorithmus):

	0	0	0	0	0	1	1	1		$(+7)_{10}$
$\times$	1	1	1	1	1	0	1	1		$(-5)_{10}$
$-$	1	1	1	1	1	0	1	1	1	$(-5)_{10}$
$+$	0	0	0	0	0	1	0	1	1	$(5)_{10}$
$+$	0	0	0	0	0	0	0	0	1	$(0)_{10}$
$+$	0	0	0	0	0	0	0	0	1	$(0)_{10}$
$+$	1	1	0	1	1	0	0	0	0	$(-40)_{10}$
$=$	1	1	0	1	1	1	0	1		$(-35)_{10}$

- Der erste Faktor 7 hat nur eine Einserkette. Als Differenz von Zweierpotenzen ist diese darstellbar als $7 = 2^3 - 2^0$.

- An der letzten Eins der Einserkette der 7 (entsprechend dem Exponenten 0) muß daher -5, subtrahiert werden (rote Zeile), was durch Addieren des Zweierkomplementes von -5, also durch Addieren von 5 erreicht wird.

  (Beachte: Die rote Zeile ist bei der abschließenden Addition der Stellenprodukte zu ignorieren. Sie dient nur der Illustration und wird durch die nachfolgende schwarze Zeile ersetzt.)

Multiplikation: Booths Algorithmus

- 4-Bit-Multiplikation (mit Booths Algorithmus analog zu Grundschulalgorithmus):

	0	0	0	0	0	1	1	1		$(+7)_{10}$
$\times$	1	1	1	1	1	0	1	1		$(-5)_{10}$
$-$	1	1	1	1	1	0	1	1	1	$(-5)_{10}$
$+$	0	0	0	0	0	1	0	1	1	$(5)_{10}$
$+$	0	0	0	0	0	0	0	0	1	$(0)_{10}$
$+$	0	0	0	0	0	0	0	0	1	$(0)_{10}$
$+$	1	1	0	1	1	0	0	0	0	$(-40)_{10}$
$=$	1	1	0	1	1	1	0	1		$(-35)_{10}$

- An der Null vor der ersten Eins der Einserkette der 7 (entsprechend dem Exponenten 3) muß der andere Faktor, also -5, multipliziert mit 2^3 (d.h. verschoben um drei Stellen) addiert werden.

  (Beachte: Die rote Zeile ist bei der abschließenden Addition der Stellenprodukte zu ignorieren. Sie dient nur der Illustration und wird durch die nachfolgende schwarze Zeile ersetzt.)

- Man beachte: Der Standardalgorithmus braucht drei Additionen (eine für jede Eins in der Darstellung der 7).

Multiplikation: Booths Algorithmus

- 4-Bit-Multiplikation (mit Booths Algorithmus analog zu Grundschulalgorithmus):

	1	1	1	1	1	0	1	1		$(-5)_{10}$
$\times$	0	0	0	0	0	1	1	1		$(+7)_{10}$
$-$	0	0	0	0	0	1	1	1	1	$(7)_{10}$
$+$	1	1	1	1	1	0	0	1	1	$(-7)_{10}$
$+$	0	0	0	0	0	0	0	0	1	$(0)_{10}$
$+$	0	0	0	1	1	1	0	0	0	$(28)_{10}$
$-$	0	0	1	1	1	0	0	0	1	$(56)_{10}$
$+$	1	1	0	0	1	0	0	0	1	$(-56)_{10}$
$=$	1	1	0	1	1	1	0	1		$(-35)_{10}$

- Wir kehren die Reihenfolge der Multiplikation um, und zeigen, dass es immer noch klappt. Die Zahl -5 hat in der Darstellung als Zweierkomplement $[11111011]_{2K}$ zwei Einserketten.

- Für die rechte Einserkette subtrahieren wir zuerst $7 \cdot 2^0$ und addieren $7 \cdot 2^2$, analog zu vorher. Die Subtraktion erfolgt wieder durch Addition des Zweierkomplements.

- Schliesslich wird für die führende Einserkette erst einmal $7 \cdot 2^3$ genau so subtrahiert.

Multiplikation: Booths Algorithmus

- 4-Bit-Multiplikation (mit Booths Algorithmus analog zu Grundschulalgorithmus):

	1	1	1	1	1	0	1	1		$(-5)_{10}$
$\times$	0	0	0	0	0	1	1	1		$(+7)_{10}$
$-$	0	0	0	0	0	1	1	1	1	$(7)_{10}$
$+$	1	1	1	1	1	0	0	1	1	$(-7)_{10}$
$+$	0	0	0	0	0	0	0	0	1	$(0)_{10}$
$+$	0	0	0	1	1	1	0	0	0	$(28)_{10}$
$-$	0	0	1	1	1	0	0	0	1	$(56)_{10}$
$+$	1	1	0	0	1	0	0	0	1	$(-56)_{10}$
$=$	1	1	0	1	1	1	0	1		$(-35)_{10}$

- Der Trick ist nun, dass bei einer führenden Eins, also einem negativen ersten Faktor, kein positiver Summand mehr mit addiert wird.

- Warum funktioniert das? Das Zweierkomplement $[11111011]_{2K}$ codiert die Zahl

$$-2^7 + [1111011]_2 = -2^7 + (2^7 - 2^3) + (2^2 - 2^0) = -2^3 + 2^2 - 2^0.$$

  siehe Folie 33.

- Wir haben an dieser Stelle also bereits die korrekte Multiplikation durchgeführt und sind fertig!

Multiplikation: Booths Algorithmus

- 4-Bit-Multiplikation (mit Booths Algorithmus analog zu Grundschulalgorithmus):

	1	1	1	1	1	0	1	1		$(-5)_{10}$
$\times$	0	0	0	0	0	1	1	1		$(+7)_{10}$
$-$	0	0	0	0	0	1	1	1	1	$(7)_{10}$
$+$	1	1	1	1	1	0	0	1	1	$(-7)_{10}$
$+$	0	0	0	0	0	0	0	0	1	$(0)_{10}$
$+$	0	0	0	1	1	1	0	0	0	$(28)_{10}$
$-$	0	0	1	1	1	0	0	0	1	$(56)_{10}$
$+$	1	1	0	0	1	0	0	0	1	$(-56)_{10}$
$=$	1	1	0	1	1	1	0	1		$(-35)_{10}$

- Man beachte, dass es nicht wichtig ist, an welcher Stelle das Zweierkomplement "abgeschnitten" wird. Im Extremfall, dass die -5 tatsächlich nur als 4-Bit Zahl $[1011]_{2K}$ codiert ist, hat man ebenfalls

$$[1011]_{2K} = -2^3 + [011]_2 = -2^3 + (2^2 - 2^0),$$

  also den gleichen effektiven Faktor.

- Auf einen allgemeineren mathematischen Beweis verzichten wir hier ausnahmsweise, das Beispiel möge genügen.

Multiplikation: Booths Algorithmus

- Booths Algorithmus benutzt eine Operation ASR (*Arithmetic Shift Right*), die fast wie ein einfaches Rechtsschieben von Bits funktioniert.

  Unterschied: Das Vorzeichenbit wird erhalten.

- Im Gegensatz dazu steht die Operation LSR (*Logical Shift Right*), die die neu entstehende Stelle Null setzt.

- Beispiele:

$$1011 \overset{\text{ASR}}{\Longrightarrow} 1101 \qquad 0011 \overset{\text{ASR}}{\Longrightarrow} 0001$$

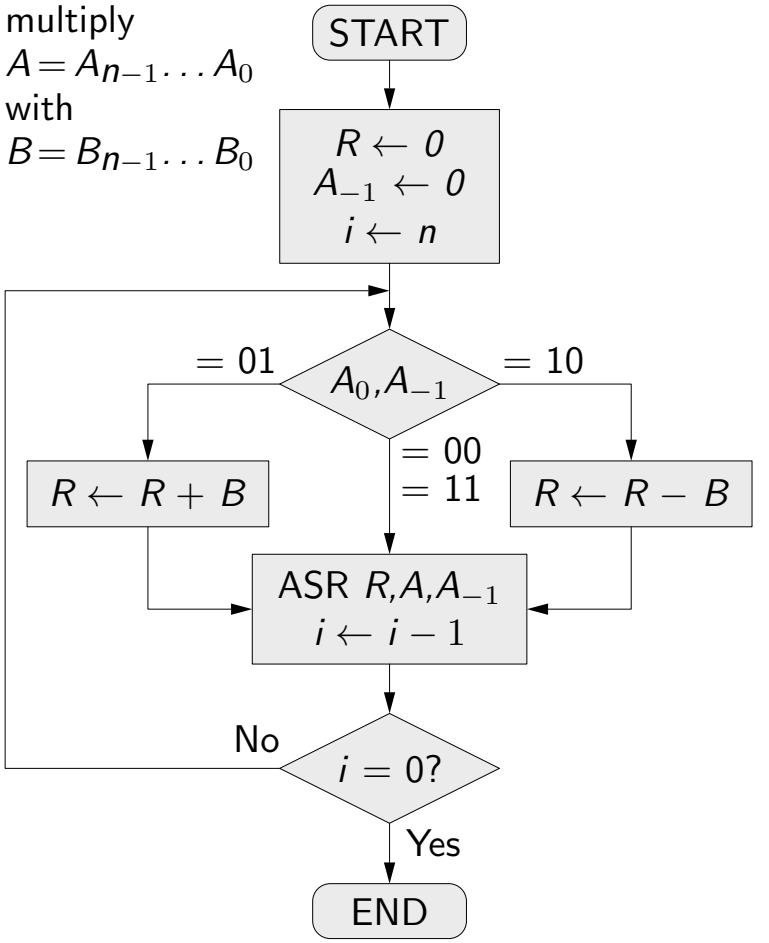
$$1011 \overset{\text{LSR}}{\Longrightarrow} 0101 \qquad 0011 \overset{\text{LSR}}{\Longrightarrow} 0001$$

- Beachte: Beim Linksschieben von Bits werden ASL (*Arithmetic Shift Left*) und LSL (*Logical Shift Left*) nicht unterschieden.
  (Gleiche Operation: In beiden Fällen wird das unterste Bit 0 gesetzt.)

- In Booths Algorithmus wird ASR auf drei „verbundene“ Register angewandt:

$$\underbrace{1001}_{R} \frown \underbrace{0010}_{A} \frown \underbrace{1}_{A_{-1}} \overset{\text{ASR } R, A, A_{-1}}{\Longrightarrow} \underbrace{1101}_{R} \frown \underbrace{1001}_{A} \frown \underbrace{0}_{A_{-1}}$$

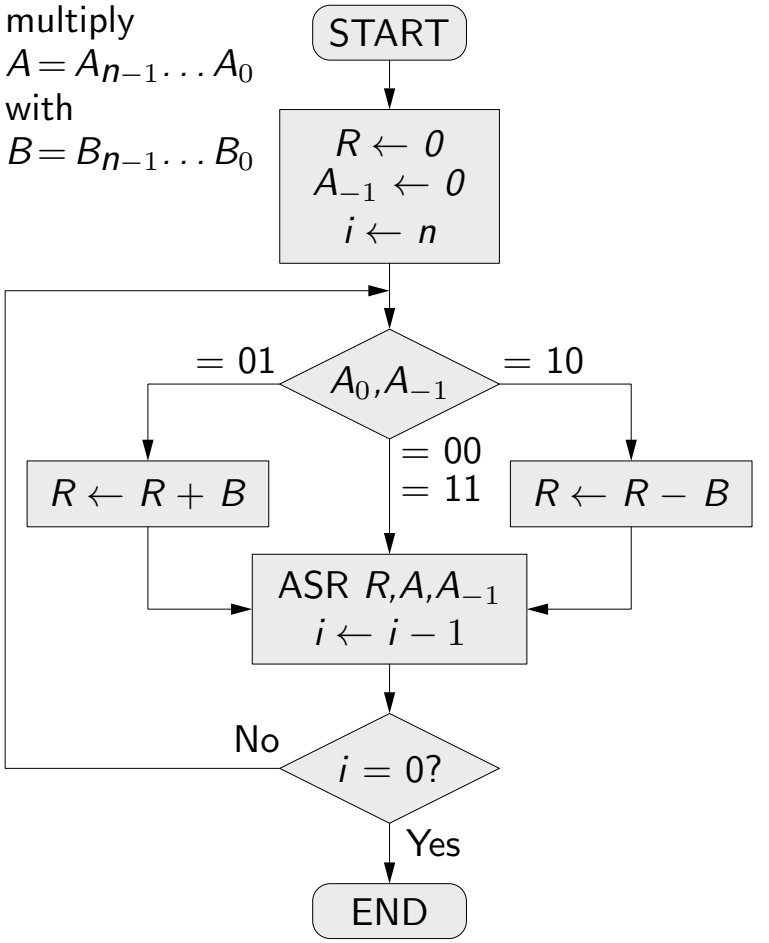
Multiplikation: Booths Algorithmus



[Andrew Donald Booth 1951]

- Man beachte, wie Booths Algorithmus bei einem Übergang $0 \rightarrow 1$ (also $A_0 A_{-1} = 10$) den rechten Zweig (Subtraktion) und bei einem Übergang $1 \rightarrow 0$ (also $A_0 A_{-1} = 01$) den linken Zweig (Addition) wählt. Dies implementiert das besprochene Schema.
- Booths Algorithmus ist besonders bei Multiplikationen mit (kleinen) negativen Zahlen günstig, da die führenden Einsen mit Hilfe nur einer Subtraktion behandelt werden. (Keine zusätzliche Addition zum Abschluß, da kein Übergang $1 \rightarrow 0$ mehr.)
- Booths Algorithmus ist aber nicht immer besser als das Standardverfahren: Bei alternierenden Nullen und Einsen braucht Booths Algorithmus doppelt so viele Additionen.

Multiplikation: Booths Algorithmus



[Andrew Donald Booth 1951]

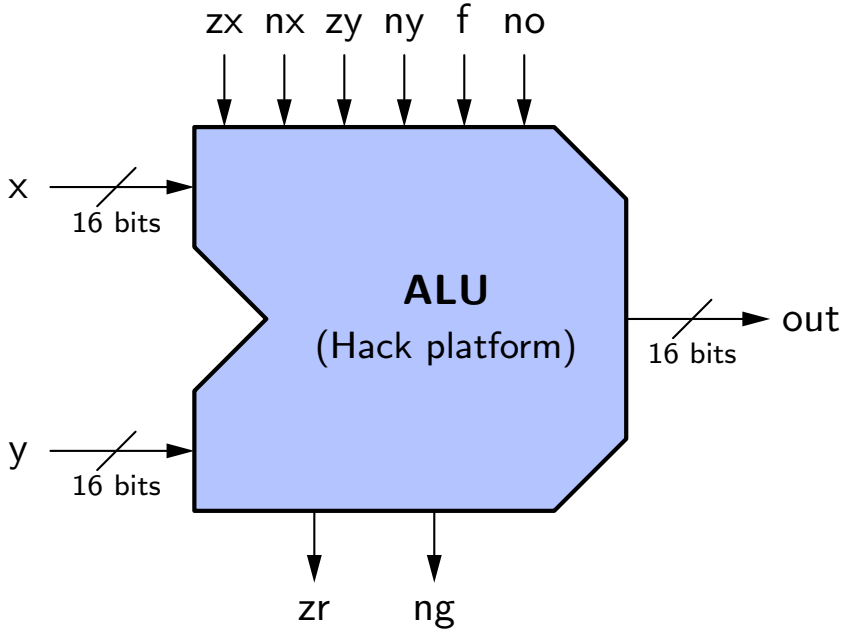
Beispiel: $A = 1011_{2K} = -5_{10}$
$B = 0111_{2K} = +7_{10}$

i	R	A	A_{-1}	B	Bemerkung
4	0000	1011	0	0111	$A_0A_{-1} = 10$
4	1001	1011	0	0111	$R \leftarrow R - B$
4	1100	1101	1	0111	ASR R,A,A_{-1}
3	1100	1101	1	0111	$A_0A_{-1} = 11$
3	1110	0110	1	0111	ASR R,A,A_{-1}
2	1110	0110	1	0111	$A_0A_{-1} = 01$
2	0101	0110	1	0111	$R \leftarrow R + B$
2	0010	1011	0	0111	ASR R,A,A_{-1}
1	0010	1011	0	0111	$A_0A_{-1} = 10$
1	1011	1011	0	0111	$R \leftarrow R - B$
1	1101	1101	1	0111	ASR R,A,A_{-1}
0	1101	1101	1	0111	END

Ergebnis: $RA = 1101\ 1101_{2K} = -35_{10}$

Inhalt

Hack-Architektur: Arithmetisch-Logische Einheit



ALU der Hack-Platform
(16 Bit)

Die Ausgaben zr und ng der ALU zeigen Eigenschaften des berechneten Ergebnisses an:

Es ist zr $= 1$, wenn out $= 0$.
(zero flag)

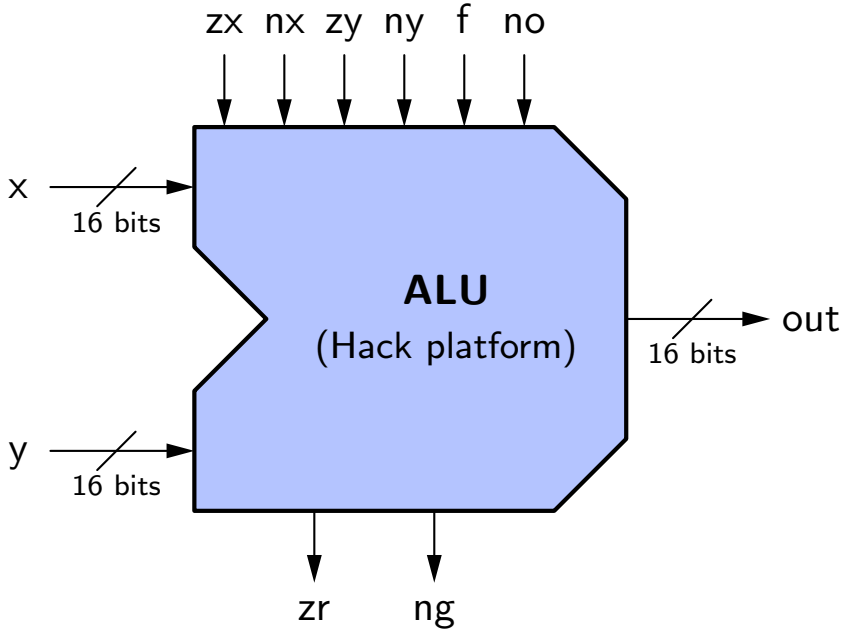
Es ist ng $= 1$, wenn out < 0.
(negative flag)

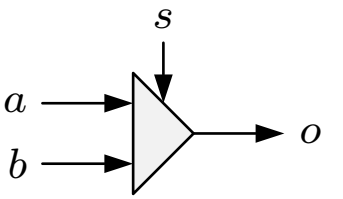
Über-/Unterlauf wird ignoriert.

Die Eingaben zx, nx, zy, ny, f und no kodieren die auszuführende Operation:

zx	Setze Eingabe x $= 0$.	nx	Bilde Einerkomplement der Eingabe x.
zy	Setze Eingabe y $= 0$.	ny	Bilde Einerkomplement der Eingabe y.
f	Wählt zwischen Addition und bitweisem Und als Operation.		
no	Bilde Einerkomplement der Ausgabe out.		

Hack-Architektur: Arithmetisch-Logische Einheit





$s = 0$

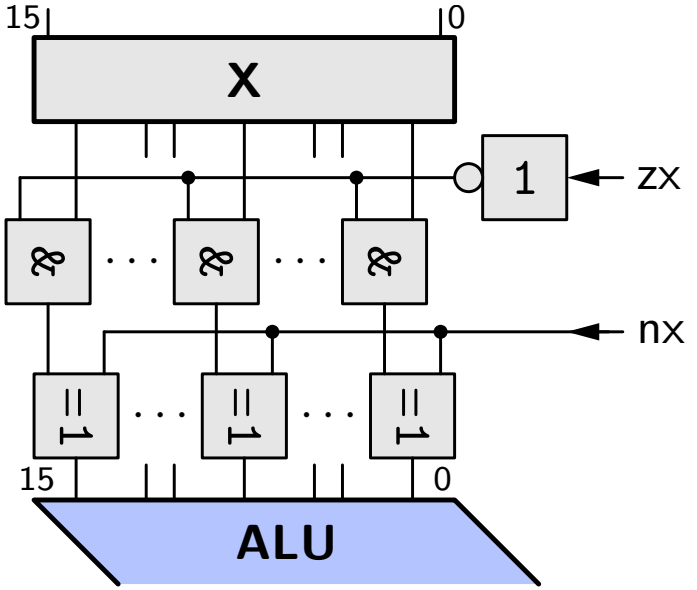
a	b	o
0	0	0
1	0	1
0	1	0
1	1	1

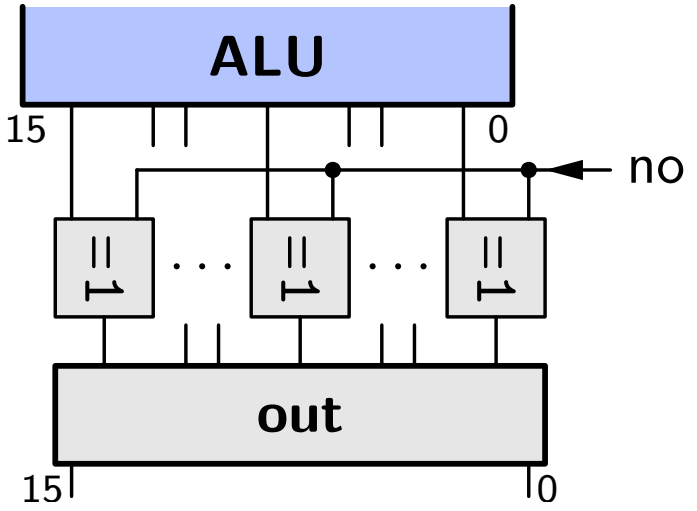
$s = 1$

a	b	o
0	0	0
1	0	0
0	1	1
1	1	1

- Die Steuerbits sind genauso zu interpretieren, wie man auch die Auswahlvariable eines Multiplexers als Wahl zwischen zwei Wahrheitstafeln sehen kann.
- Da es 6 Steuerbits gibt, kann im Prinzip zwischen $2^6 = 64$ Wahrheitstafeln gewählt werden, die bestimmen, wie sich die Ausgabe out aus den Eingaben x und y ergibt.
- Von diesen 64 sind nur 18 relevant, die einfache, sinnvolle Funktionen darstellen, die sich außerdem leicht in Vor- und Nachverarbeitung zerlegen lassen.

Hack-Architektur: Arithmetisch-Logische Einheit





- Die **Steuerbits** dienen als Eingaben für Steuergatter, die die zugeordnete Operation bewirken.

  Links sind die Steuerbits zx, nx und no als Beispiele gezeigt.

- Das negierte Steuerbit zx wird konjunktiv mit den Bits der Eingabe X verknüpft. Es wird also ausgenutzt, daß

$$x_i \wedge 0 = 0 \quad \text{(falls zx} = 1\text{)},$$
$$x_i \wedge 1 = x_i \quad \text{(falls zx} = 0\text{)}.$$

- Die Steuerbits nx bzw. no werden über ein exklusives Oder mit den Bits der Eingabe X bzw. der Ausgabe out verknüpft. Es wird also ausgenutzt, daß z.B.

$$x_i \oplus 0 = x_i \quad \text{(falls nx} = 0\text{)},$$
$$x_i \oplus 1 = \overline{x_i} \quad \text{(falls nx} = 1\text{)}.$$

Hack-Architektur: Arithmetisch-Logische Einheit

These bits instruct how to preset the input x		These bits instruct how to preset the input y		This bit selects between + and &	This bit instructs how to postset the output out	Resulting ALU output
zx	nx	zy	ny	f	no	out=
if zx then x = 0	if nx then x = ~x	if zy then y = 0	if ny then y = ~y	if f then out = x+y else out = x&y	if no then out = ~out	f(x,y) =
1	0	1	0	1	0	0
1	1	1	1	1	1	1
1	1	1	0	1	0	-1
0	0	1	1	0	0	x
1	1	0	0	0	0	y
0	0	1	1	0	1	~x
1	1	0	0	0	1	~y
0	0	1	1	1	1	-x
1	1	0	0	1	1	-y
0	1	1	1	1	1	x+1
1	1	0	1	1	1	y+1
0	0	1	1	1	0	x-1
1	1	0	0	1	0	y-1
0	0	0	0	1	0	x+y
0	1	0	0	1	1	x-y
0	0	0	1	1	1	y-x
0	0	0	0	0	0	x&y
0	1	0	1	0	1	x|y

Hack-Architektur: Arithmetisch-Logische Einheit

zx	nx	zy	ny	f	no	out=
1	0	1	0	1	0	0
1	1	1	1	1	1	1
1	1	1	0	1	0	-1
0	0	1	1	0	0	x
1	1	0	0	0	0	y
0	0	1	1	0	1	~x
1	1	0	0	0	1	~y
0	0	1	1	1	1	-x
1	1	0	0	1	1	-y
0	1	1	1	1	1	x+1
1	1	0	1	1	1	y+1
0	0	1	1	1	0	x-1
1	1	0	0	1	0	y-1
0	0	0	0	1	0	x+y
0	1	0	0	1	1	x-y
0	0	0	1	1	1	y-x
0	0	0	0	0	0	x&y
0	1	0	1	0	1	x|y

- **Bemerkungen zur Notation:**
  In dieser Tabelle bezeichnet das Zeichen „~“ das Bilden des Einerkomplementes.
- In dem Buch “The Elements of Computing Systems: Building a Modern Computer from First Principles” von Noam Nisan & Shimon Schocken, dem diese Vorlesung folgt, bezeichnet das Zeichen „!“ das Einerkomplement.
- Dies steht jedoch im Widerspruch zu der Konvention in vielen Programmiersprachen (z.B. C/C++, C#, Java), in denen das Zeichen „~“ das Einerkomplement bezeichnet.
- In diesen Programmiersprachen bezeichnet das Zeichen „!“ stattdessen die logische Negation: Die Zahl 0 wird als falsch, jede Zahl $\neq 0$ als wahr interpretiert; der Operator „!“ kehrt diese Interpretation um (entspricht „$\neg$“).

Hack-Architektur: Arithmetisch-Logische Einheit

zx	nx	zy	ny	f	no	out=
1	0	1	0	1	0	0
1	1	1	1	1	1	1
1	1	1	0	1	0	-1
0	0	1	1	0	0	x
1	1	0	0	0	0	y
0	0	1	1	0	1	~x
1	1	0	0	0	1	~y
0	0	1	1	1	1	-x
1	1	0	0	1	1	-y
0	1	1	1	1	1	x+1
1	1	0	1	1	1	y+1
0	0	1	1	1	0	x-1
1	1	0	0	1	0	y-1
0	0	0	0	1	0	x+y
0	1	0	0	1	1	x-y
0	0	0	1	1	1	y-x
0	0	0	0	0	0	x&y
0	1	0	1	0	1	x|y

- **Bemerkungen zur Notation:**
  Es ist $\texttt{x\&y} \mathrel{\widehat{=}} \texttt{x} \wedge \texttt{y}$ (bitweise)
  und $\texttt{x|y} \mathrel{\widehat{=}} \texttt{x} \vee \texttt{y}$ (bitweise).

- Der Grund für die Verwendung dieser beiden Zeichen ist, daß in vielen Programmiersprachen,
  (z.B. C/C++, C# und Java), das Et-Zeichen (nach dem lateinischen „et“ für „und“), auch kaufmännisches Und-Zeichen genannt (engl.: ampersand), also das Zeichen „&“, zur Bezeichnung der bitweisen Konjunktion, und das Zeichen „|“ zur Bezeichnung der bitweisen Disjunktion verwendet wird.

- Hier besteht leider die Gefahr der Verwechslung mit dem logischen Zeichen „|“ (Sheffer-Strich), das die negierte Konjunktion bezeichnet.

  In diesen Folien sind die beiden Zeichen „|“ und „|“ typographisch unterschieden.

Hack-Architektur: Arithmetisch-Logische Einheit

zx	nx	zy	ny	f	no	out=
1	0	1	0	1	0	0
1	1	1	1	1	1	1
1	1	1	0	1	0	-1
0	0	1	1	0	0	x
1	1	0	0	0	0	y
0	0	1	1	0	1	~x
1	1	0	0	0	1	~y
0	0	1	1	1	1	-x
1	1	0	0	1	1	-y
0	1	1	1	1	1	x+1
1	1	0	1	1	1	y+1
0	0	1	1	1	0	x-1
1	1	0	0	1	0	y-1
0	0	0	0	1	0	x+y
0	1	0	0	1	1	x-y
0	0	0	1	1	1	y-x
0	0	0	0	0	0	x&y
0	1	0	1	0	1	x|y

- Die Ausgabe `0` wird berechnet als `0+0` (klar), die Ausgabe `1` aber als `~(~0+~0)` und die Ausgabe `-1` als `~0+0` („~“ bezeichnet das Einerkomplement).

  Grund: Im Zweierkomplement ist `~0` = `-1`, folglich ist `~0+0` = `-1` und `~0+~0` = `-2`, weiter ist im Zweierkomplement `~-2` = `1`.

- Die Ausgabe `x` wird berechnet als `x&~0`, die Ausgabe `y` als `~0&y` (da `~0` = `11...1`), analog die Ausgabe `~x` als `~x&~0` und die Ausgabe `~y` als `~0&~y`.

- Die Ausgabe `-x` wird berechnet als `~(x+~0)`, die Ausgabe `-y` als `~(~0+y)`.

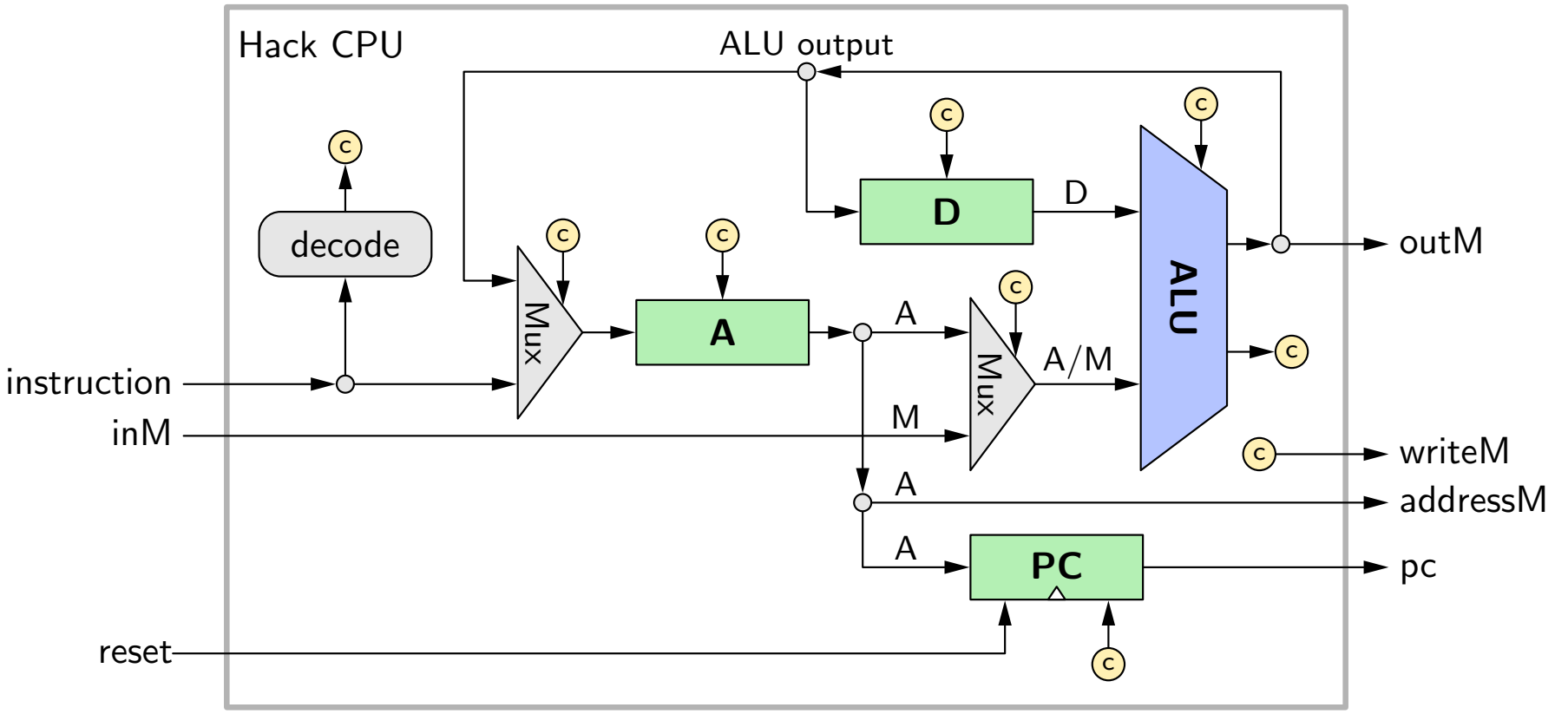
  Grund: Im Zweierkomplement ist `-z` = `~z+1`, mit `z` = `x-1` folglich `-(x-1)` = `~(x-1)+1`, also `-x` = `~(x-1)` (für `z` = `y-1` analog).

Hack-Architektur: Arithmetisch-Logische Einheit

zx	nx	zy	ny	f	no	out=
1	0	1	0	1	0	0
1	1	1	1	1	1	1
1	1	1	0	1	0	-1
0	0	1	1	0	0	x
1	1	0	0	0	0	y
0	0	1	1	0	1	~x
1	1	0	0	0	1	~y
0	0	1	1	1	1	-x
1	1	0	0	1	1	-y
0	1	1	1	1	1	x+1
1	1	0	1	1	1	y+1
0	0	1	1	1	0	x-1
1	1	0	0	1	0	y-1
0	0	0	0	1	0	x+y
0	1	0	0	1	1	x-y
0	0	0	1	1	1	y-x
0	0	0	0	0	0	x&y
0	1	0	1	0	1	x|y

- Erklärung der übrigen Funktionen:
  Übungsaufgabe, Blatt 8

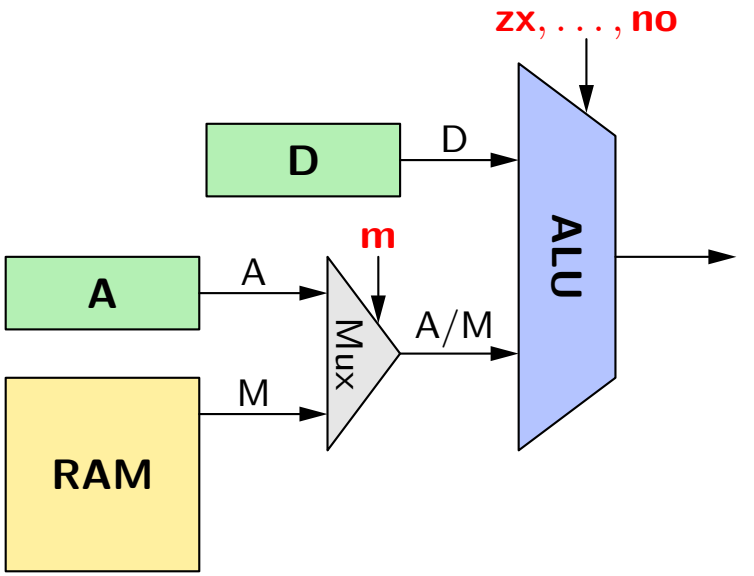
Hack-Architektur: Prozessor (CPU)



- Nur **Daten- und Adreßpfade** sind gezeigt (d.h., Verbindungen, die Daten und Adressen transportieren), nicht jedoch die **Steuerlogik**, mit Ausnahme der Ein- und Ausgaben von Steuerbits, die durch (c) markiert sind.

Hack-Architektur: ALU-Steuerung

Berechnung wenn m = 0	zx	nx	zy	ny	f	no	Berechnung wenn m = 1
0	1	0	1	0	1	0	
1	1	1	1	1	1	1	
-1	1	1	1	0	1	0	
D	0	0	1	1	0	0	
A	1	1	0	0	0	0	M
~D	0	0	1	1	0	1	
~A	1	1	0	0	0	1	~M
-D	0	0	1	1	1	1	
-A	1	1	0	0	1	1	-M
D+1	0	1	1	1	1	1	
A+1	1	1	0	1	1	1	M+1
D-1	0	0	1	1	1	0	
A-1	1	1	0	0	1	0	M-1
D+A	0	0	0	0	1	0	D+M
D-A	0	1	0	0	1	1	D-M
A-D	0	0	0	1	1	1	M-D
D&A	0	0	0	0	0	0	D&M
D|A	0	1	0	1	0	1	D|M



~D, ~A und ~M bezeichnen das Einerkomplement des Register D, des Registers A bzw. einer Speicherzelle M.

D&A und D&M bzw. D|A und D|M berechnen ein bitweises Und bzw. ein bitweises Oder des Registers D mit dem Register A bzw. einer Speicherzelle M.

Festkomma- und Fließ-/Gleitkommazahlen

- Mit dem Rechnen mit ganzen Zahlen hat man implizit bereits die Möglichkeit, auch mit Kommazahlen zu rechnen: Man interpretiert einfach die höherwertigen Stellen der Zahl als Vorkommateil und die niederwertigen Stellen, ab einem bestimmten, festzulegenden Punkt, als Nachkommastellen und erhält **Festkommazahlen**.

- Festkommazahlen werden in Finanzprogrammen, aber auch Textsatzprogrammen wie TEX und LATEX sowie ihren Hilfsprogrammen Metafont und Metapost verwendet (dort z.B. 16 Bit Vorkomma, 16 Bit Nachkomma). Außerdem: Bitcoin.

- **Fließkommazahlen** (auch: **Gleitkommazahlen**), die größere Zahlenbereiche erlauben, werden meist auf Grundlage der Zahlendarstellung IEEE 754 definiert.

- Eine Gleitkommazahl wird dargestellt als $x = s \cdot m \cdot b^e$ mit
  - Vorzeichen s (1 Bit),
  - Mantisse m (p Bits, z.B. $p = 23$ (single) oder $p = 52$ (double precision)),
  - Basis b (bei normalisierten Gleitkommazahlen nach IEEE 754 ist $b = 2$),
  - Exponent e (r Bits, z.B. $r = 8$ (single) oder $r = 11$ (double precision)).

Ausblick

- Unser **Addierer** ist sehr einfach: keine Parallelisierung.
  Es lohnt sich aber sehr, Addierer zu optimieren.

- Unsere **arithmetisch-logische Einheit** ist ebenfalls sehr einfach:
  - keine Multiplikation oder Division, nur Addition und Subtraktion,
  - keine Fließ-/Gleitkommazahlen,
  - keine höheren mathematischen Operationen;
  - alle diese Operationen müssen in Software implementiert werden.

- In anderen Rechnern werden Fließ-/Gleitkommanzahloperationen ausgeführt
  - in Software (Softwareemulation statt Hardware),
  - in einem gesonderten Prozessor oder einem besonderen Teil des Prozessors, der Fließkommaeinheit (floating point unit, FPU),
  - in speziellen, oft massiv parallelen Zusatzprozessoren, z.B. Graphikprozessoren (graphics processing unit, GPU).

Zusammenfassung

- **Arithmetik** (im wesentlichen Erinnerung an Bekanntes)
  - Zahlensysteme (Basis 10, andere Basen als 10)
  - Addition und Subtraktion, Übertrag
  - Multiplikation und Division, Stellenprodukte
  - Implementierung durch mechanische Rechner

- **Arithmetisch-logische Einheit** (Arithmetic Logic Unit, ALU)
  - Binärarithmetik (Rechnen im Zahlensystem mit Basis 2)
  - Halbaddierer und Volladdierer
  - n-Bit-Addierer (mit Übertragskette und Übertragsauswahl)
  - Darstellung negativer Zahlen
  - Multiplikation: Bit-Schieben, Standardalgorithmus
  - Multiplikation: negative Zahlen, Booths Algorithmus
  - Hack-Architektur (arithmetisch-logische Einheit, Prozessor)