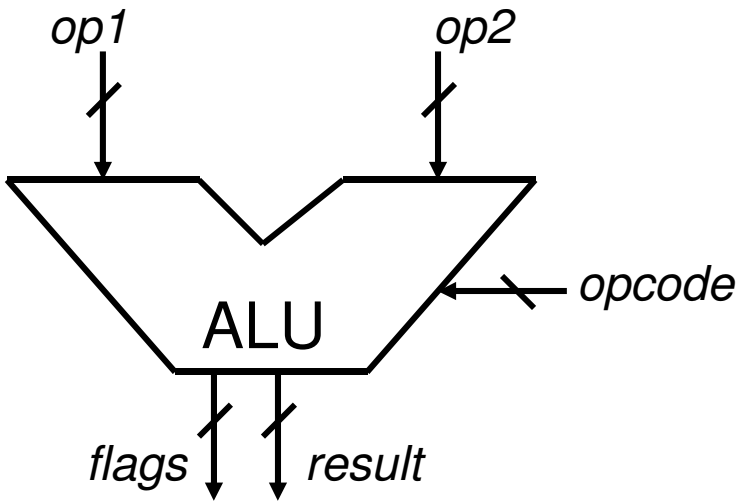
3.4 Arithmetische und logische Operationen -- die ALU

Zur Vorlesung Rechenanlagen

SS 2019

Die ALU

Die meisten Befehle betreffen arithmetische (+,-,…) oder logische (und, oder,…) Verknüpfungen von Maschinenworten. Hinzu kommen noch Vergleiche und Schiebeoperationen. Die Einheit, die im Rechner all diese Funktionen bereitstellt, nennt man **ALU** (arithmetical logical unit). Sie hat nach aussen folgende Schnittstelle:



Die ALU ff

Die Anschlüsse haben folgende Aufgaben:

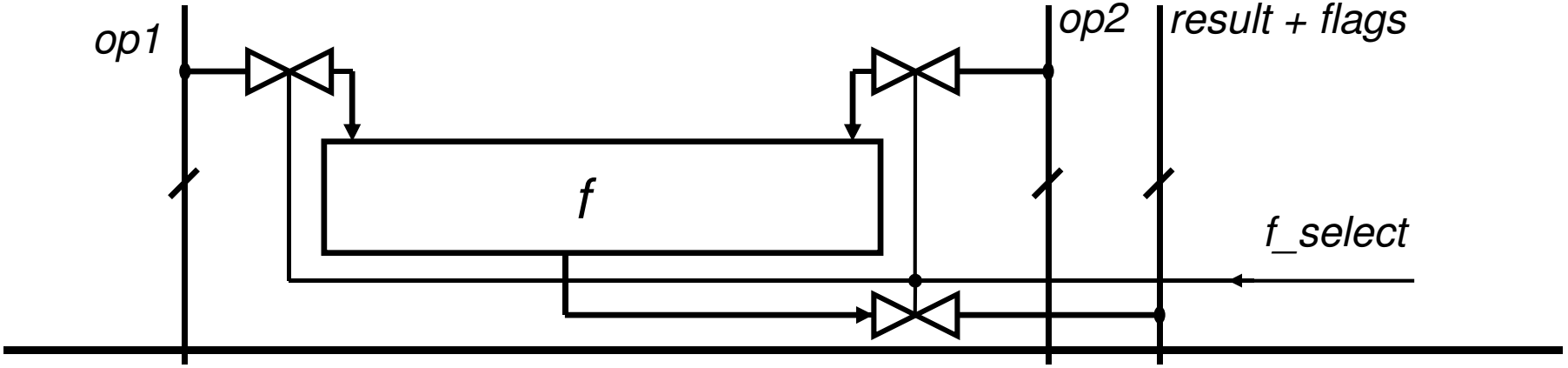
- ***op1,2***: Operanden der auszuführenden Operation
- ***result***: Resultatswort zur Operation
- ***opcode***: Auswählen der auszuführenden Operation
- ***flags***: Statusanzeigen (Fehler, Vergleichsergebnisse,…)

Wir wollen nun versuchen, diesen Baustein möglichst systematisch zu konstruieren:

Funktionsscheiben

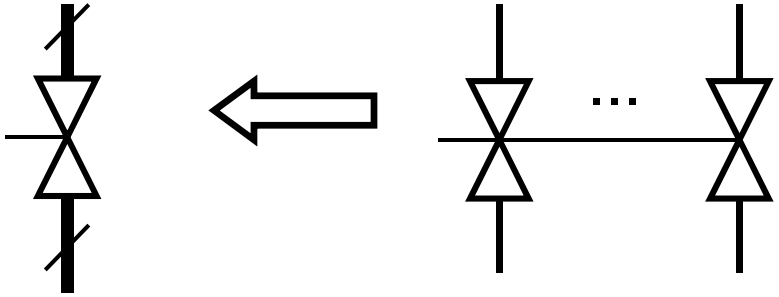
Der erste Schritt besteht darin, dass wir zu jeder Operation einen Schaltkreis bauen müssen, der diese Operation als Schaltfunktion implementiert. Wir nennen einen solchen Baustein daher auch eine **Funktionsscheibe** (function slice) der ALU.

Die Einbettung in die ALU selbst ist dann sehr einfach, wenn wir eine Steuerleitung *f_select* für die Operation *f* aus dem Operationscode decodiert haben, die *1* ist, falls die Operation *f* ausgeführt werden soll:

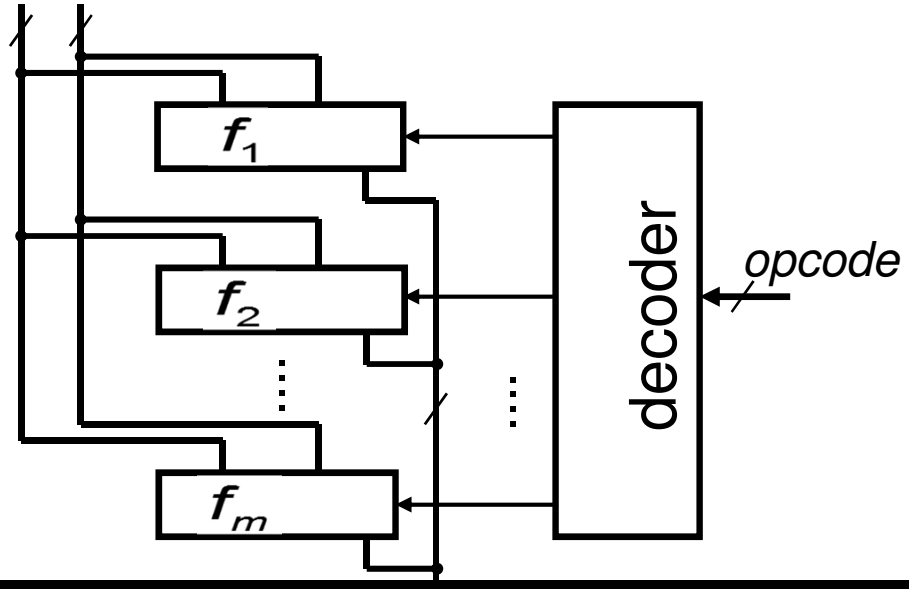


Funktionsscheiben ff

Die Schalter in obiger Zeichnung sind in Wortbreite zu lesen, d.h.



Integriert man diese Ansteuerung in die Scheibe zu *f*, so erhält man eine sehr einfache Grundstruktur der ALU:



Wir müssen uns also nur noch um die einzelnen Funktionen kümmern!

Funktionsscheiben ff

Bemerkung:

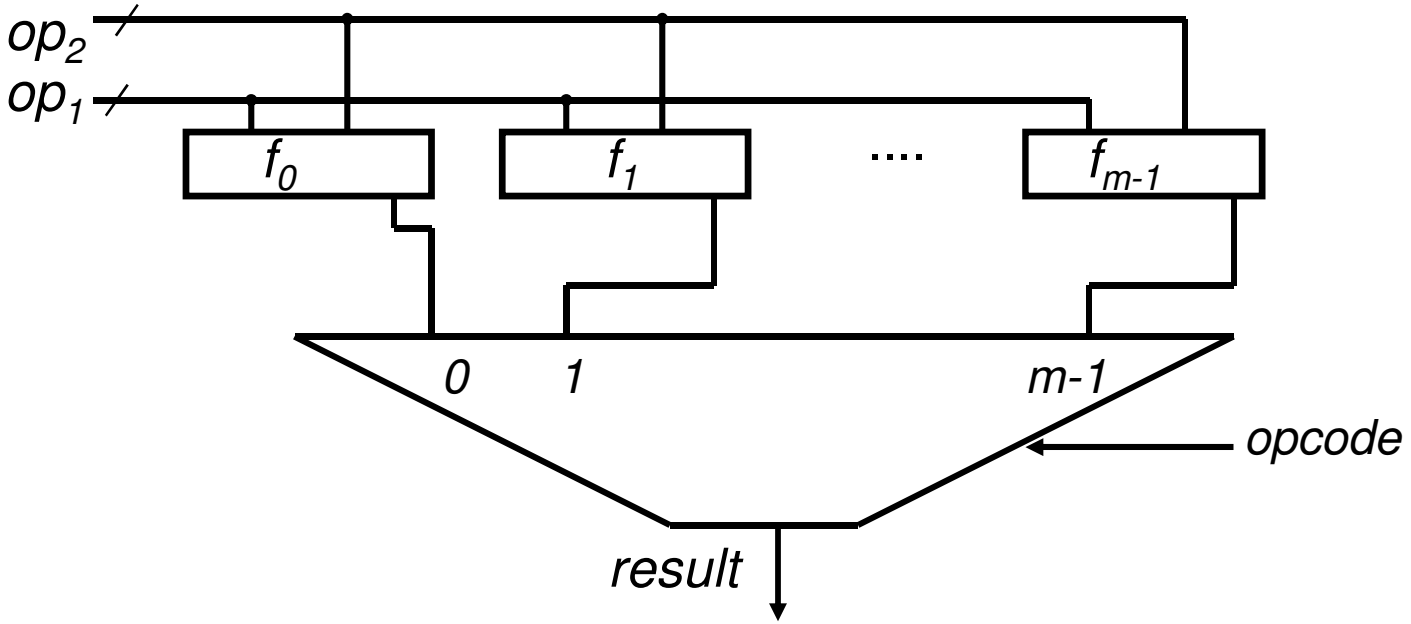
Die Schalter, mit denen die Operanden angelegt werden, scheinen überflüssig zu sein, da ja eine Funktionsscheibe auch dann rechnen kann, wenn ihr Ergebnis nicht benutzt wird.

Aber: Schaltvorgänge in Funktionsscheiben kosten Energie, d.h. rechnen Funktionsscheiben Ergebnisse aus, die nicht gebraucht werden, verbraucht die Schaltung unnötig Energie. Dies kann zu Kühlproblemen (Umschaltvorgänge erzeugen Wärme) oder niedrigen Akku-Laufzeiten führen. Sowohl im High Performance Bereich und erst recht im mobilen Bereich ist man an einer Minimierung der Verlustleistung interessiert.

Funktionsscheiben ff

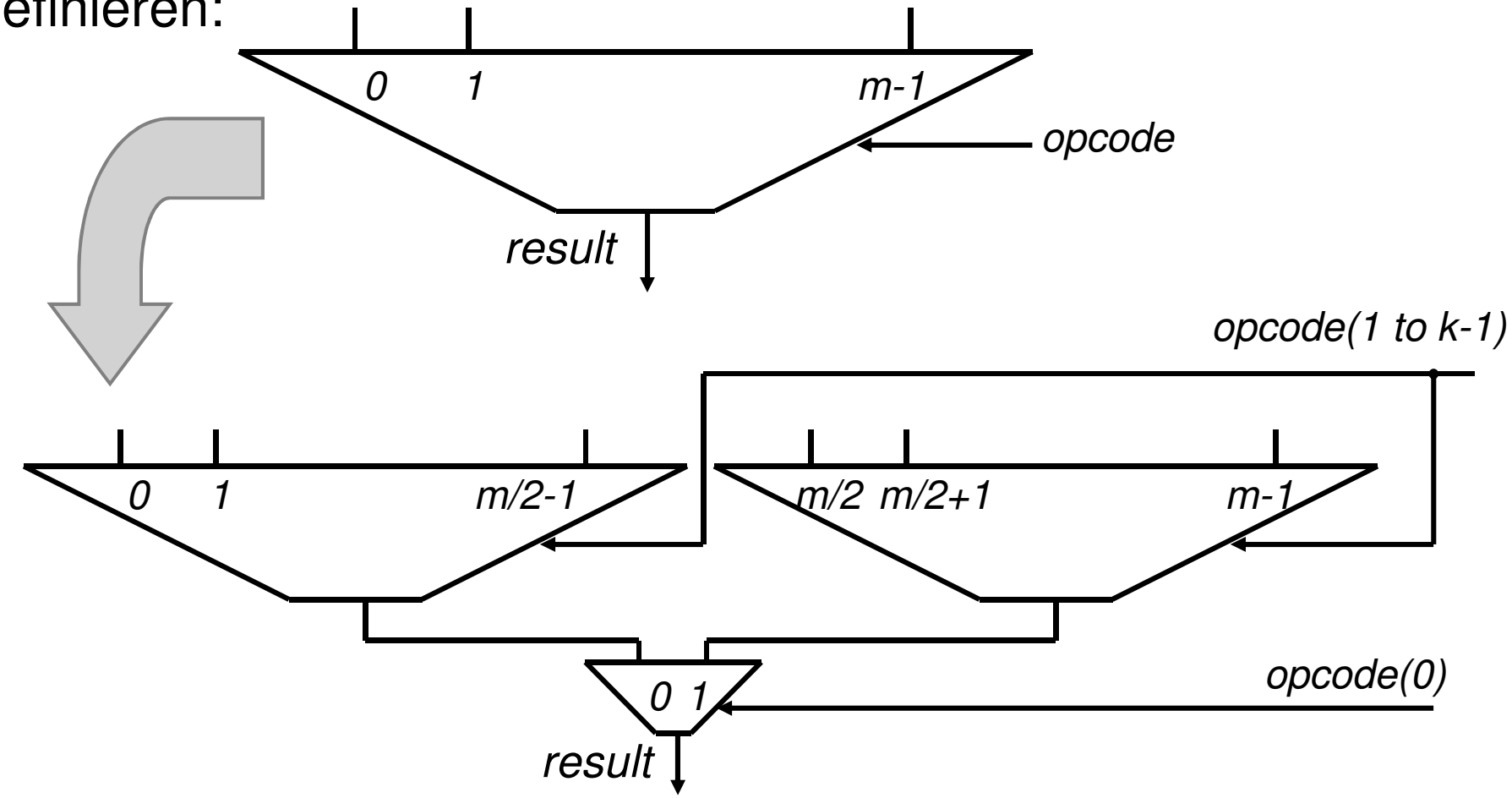
Bemerkung:

Die Resultatsleitungen sind in unserem Schema aufgelöste Signale, da sie mit Schaltern angesteuert werden. In Technologien, die nur unaufgelöste Signale zulassen (z.B. FPGAs), kann man statt dieses Schemas eine **Auswahlschaltung** benutzen:



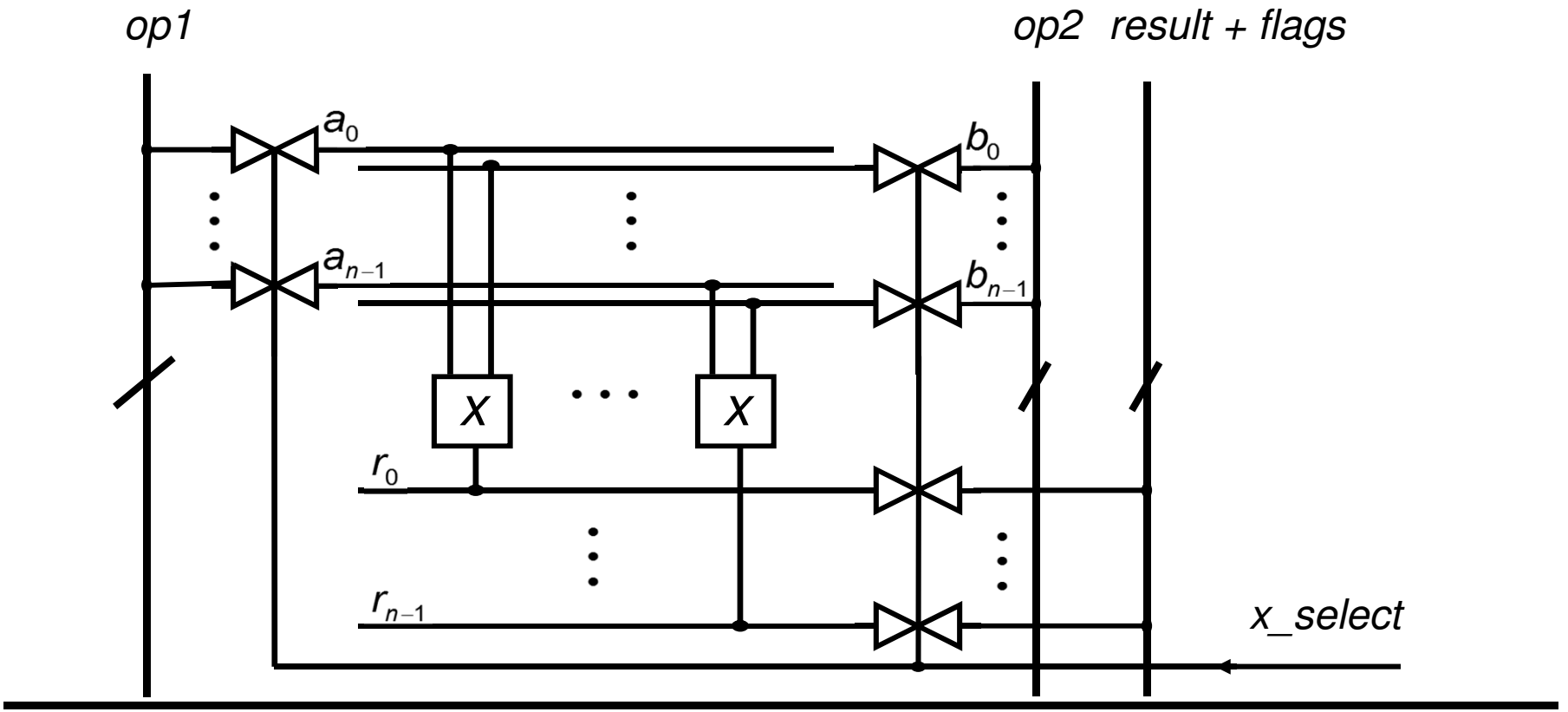
Funktionsscheiben ff

Die Auswahlschaltung kann man generisch in Wortbreite und Zahl der Operationen (Opcode-Breite) leicht rekursiv definieren:



Logikoperationen

Am leichtesten sind wohl die Funktionsscheiben für einfache Logikoperationen zu konstruieren, da diese lediglich komponentenweise boolesche Operationen auf den Bits eines Maschinenworts ausführen müssen:



Addition und Subtraktion

Wir haben schon die Addition von vorzeichenlosen Zahlen betrachtet, d.h. wir wissen, wie eine Funktionsscheibe **ADDU** aussieht, die für zwei Worte *a, b* der Länge *n* als Eingänge und ein Wort *s* der Länge *n* als Ausgang, sowie ein Flag *ov* als weiteren Ausgang die folgende Funktion *addu* realisiert:

$$addu : \mathbf{B}^{2n} \mapsto \mathbf{B}^{n+1}, \text{ mit}$$

$$addu(a,b) = (ov,s) \Leftrightarrow 2^n ov + u(s) = u(a) + u(b)$$

Da $u(s) \leq 2^n - 1$, zeigt das Flag *ov* offenbar an, dass die Summe nicht mehr in einem Maschinenwort dargestellt werden kann, da sie größer als $2^n - 1$ ist. Wir nennen dieses Flag auch die **Überlaufanzeige** (overflow flag).

Addition ganzer Zahlen

Bis jetzt können wir aber nur unsigned numbers addieren.

Problem:

Gegeben sei eine brauchbare Schaltung zur Addition von vorzeichenlosen Zahlen in Darstellung *u*.

Kann man daraus Additionsschaltungen für ganze Zahlen im Einer- oder Zweierkomplement oder nach Betrag und Vorzeichen gewinnen?

Dazu müssen wir wissen, wie diese Darstellungen zusammenhängen. In den Übungen klären wir:

$$(1) \quad b_2(a) = b_1(a) - a_0$$

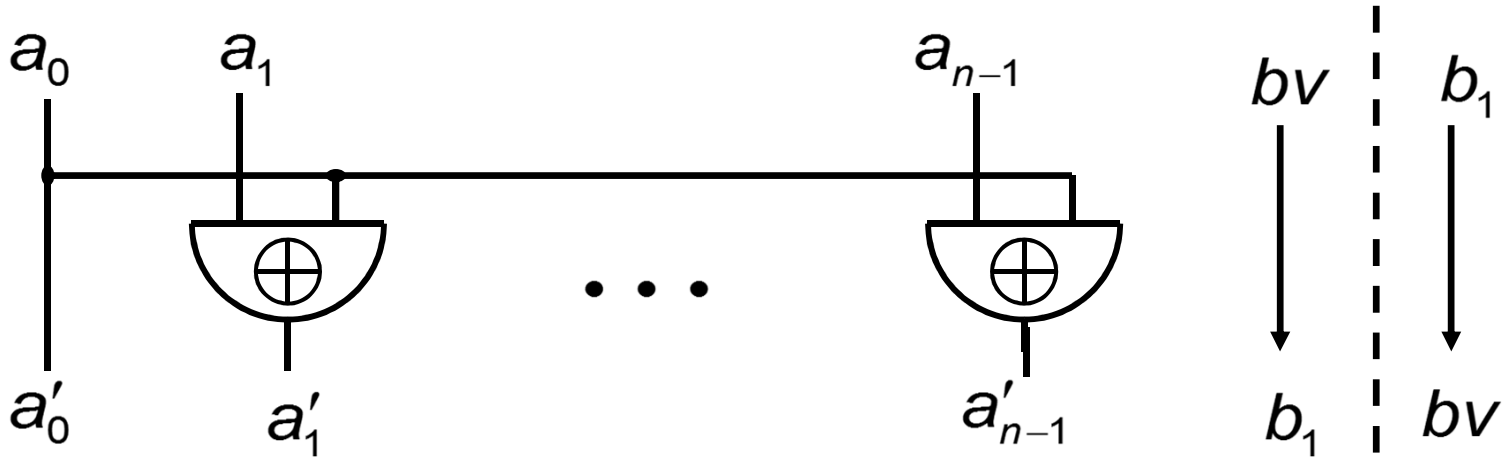
$$(2) \quad b_1(\overline{a}) = -b_1(a)$$

$$(3) \quad b_1(a_0, a_0 \oplus a_1, \ldots, a_0 \oplus a_{n-1}) = bv(a)$$

$$bv(a_0, a_0 \oplus a_1, \ldots, a_0 \oplus a_{n-1}) = b_1(a)$$

Konvertierer

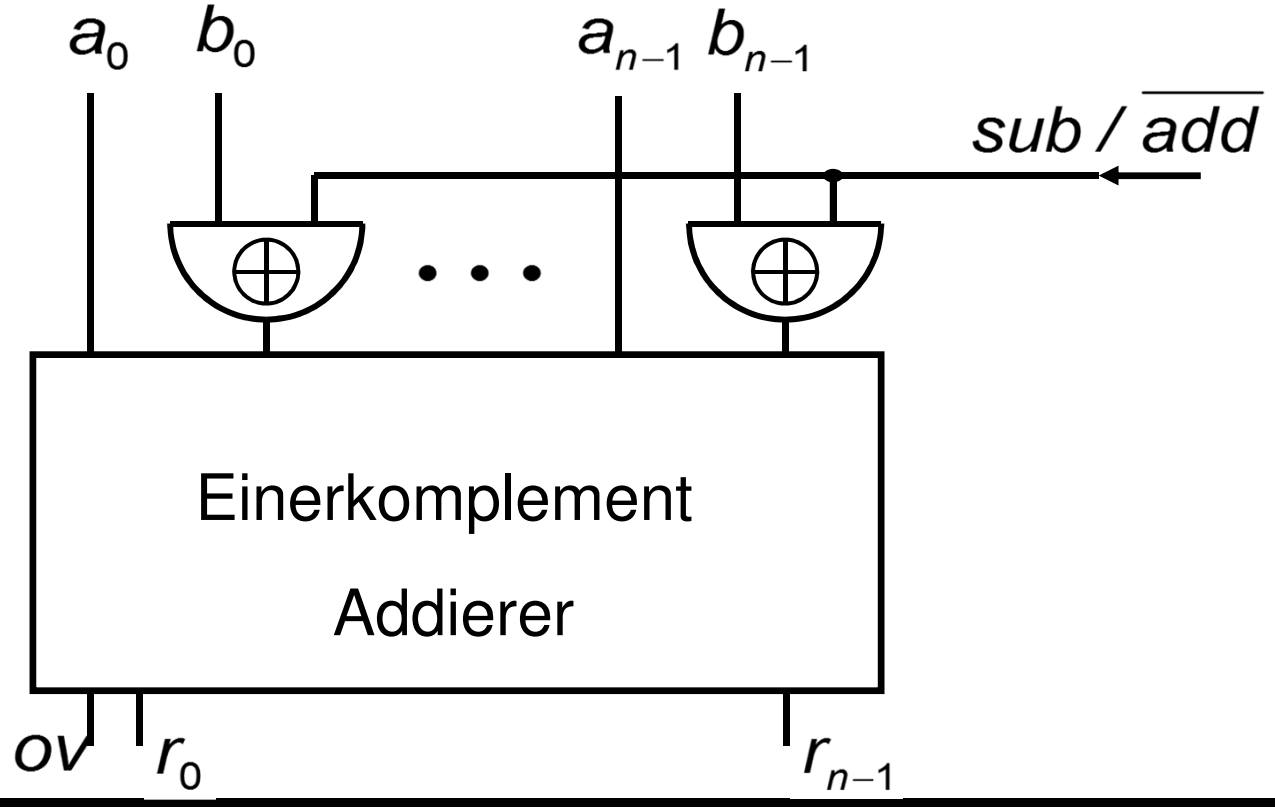
Damit können wir in einfacher Weise zwischen Einerkomplement und Betrag und Vorzeichen konvertieren:



Ebenso zwischen Zweier- und Einerkomplement, wenn man leicht eine *1* dazuaddieren oder subtrahieren kann.

Subtraktion

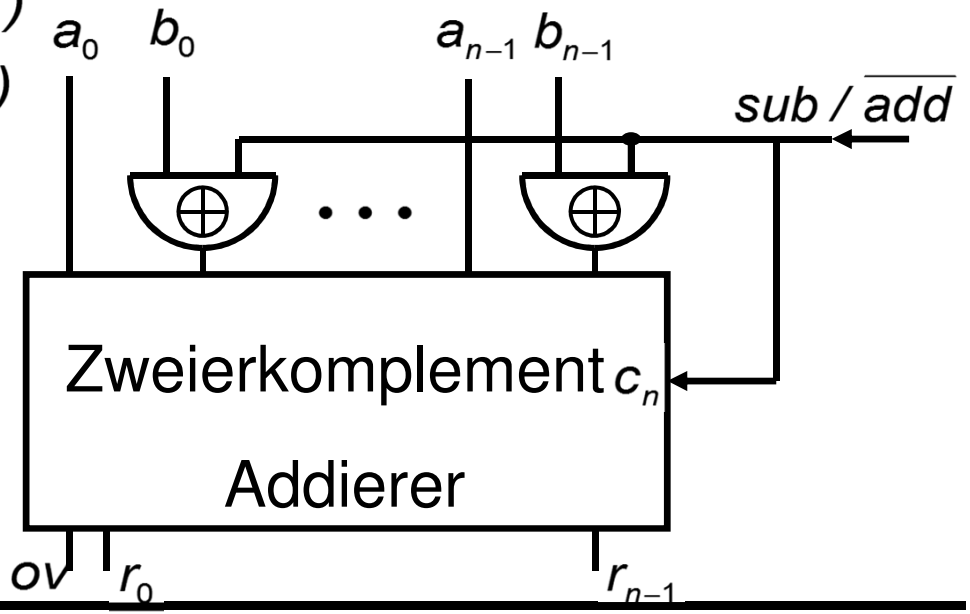
Die Subtraktion erhalten wir mit der Regel (2) für das Einerkomplement in einfacher Weise, wenn wir einen Addierer für Einerkomplementdarstellung haben: Wir brauchen bei einer Subtraktion nur die Bits des Subtrahenden zu invertieren:



Subtraktion

Mit den Regeln (1) und (2) kann man aber auch für das Zweierkomplement in einfacher Weise subtrahieren, wenn wir einen Addierer haben, der die Addition einer 1 zusätzlich erlaubt (z.B. beim Csu-Adder geschenkt !)

$$
\begin{aligned}
-b_2(x) &= -(b_1(x) - x_0) && (1)\\
&= -b_1(x) + x_0\\
&= b_1(\overline{x}) + x_0 && (2)\\
&= b_2(\overline{x}) + \overline{x}_0 + x_0 && (1)\\
&= b_2(\overline{x}) + 1
\end{aligned}
$$



Zurückführung auf unsigned Adder

Zu klären bleibt also nur noch

- wie man die Addition im 1er, 2er Komplement auf den Fall der vorzeichenlosen Zahlen zurückführt, und
- wie man Überläufe erkennt.

Der Zusammenhang zwischen unsigned und Zweierkomplementdarstellung ist

$$b_2(a) = -a_0 2^{n-1} + 2^{n-1} \sum_{i=1}^{n-1} a_i 2^{-i}$$

$$= -a_0 2^n + a_0 2^{n-1} + 2^{n-1} \sum_{i=1}^{n-1} a_i 2^{-i}$$

$$= -a_0 2^n + 2^{n-1} \sum_{i=0}^{n-1} a_i 2^{-i}$$

also $\boxed{b_2(a) = -a_0 2^n + u(a)}$

Zurückführung auf unsigned Adder

Nehmen wir nun einfach einmal einen unsigned Adder her, der zu zwei Worten *x* und *y* ein Wort *s* und ein Überlaufbit *c* zurückliefert mit

$$2^n c + u(s) = u(x) + u(y)$$

Dann gilt mit $b_2(a) + a_0 2^n = u(a)$

$$2^n c + b_2(s) + s_0 2^n = b_2(x) + x_0 2^n + b_2(y) + y_0 2^n$$

also

$$b_2(s) = b_2(x) + b_2(y) + (x_0 + y_0 - s_0 - c)2^n$$

Dies ist ja schon fast die Summe, bis auf den „Schmutz“

$(x_0 + y_0 - s_0 - c)2^n$ Das schauen wir uns genauer an!

Diskussion -- Fall 1

Wir nehmen an, dass $x_0 + y_0 = 0$

also **beide Operanden nicht negativ.**

Dann ist

$u(x) + u(y) \leq 2^{n-1} - 1 + 2^{n-1} - 1 = 2^n - 2$ und damit $\boxed{c = 0}$

Ist nun aber

$$(x_0 + y_0 - s_0 - c)2^n = -s_0 2^n \neq 0$$

Dann ist $u(x) + u(y) \geq 2^{n-1}$

$$\Rightarrow b_2(x) + x_0 2^n + b_2(y) + y_0 2^n \geq 2^{n-1}$$

$$\Rightarrow b_2(x) + b_2(y) \geq 2^{n-1} \quad \Rightarrow \textbf{Überlauf}$$

Das Ergebnis ist also korrekt, wenn kein Überlauf vorliegt.

Diskussion -- Fall 2

Wir nehmen an, dass $x_0 + y_0 = 1$

d.h. **beide Operanden haben verschiedenes Vorzeichen.**

Dann ist für den einlaufenden Übertrag c_1 im Addierer:

$2c + s_0 = x_0 + y_0 + c_1$ und da $c_1 \in \{0,1\}$

ist also

$$1 = x_0 + y_0 \leq 2c + s_0 \leq x_0 + y_0 + 1 = 2$$

Dann ist aber schon $c + s_0 = 1$

Also ist

$$(x_0 + y_0 - s_0 - c)2^n = (1 - (s_0 + c))2^n$$
$$= 0$$

Das Ergebnis ist also immer korrekt.

Diskussion -- Fall 3

Wir nehmen an, dass $x_0 + y_0 = 2$

also **beide Operanden negativ.**

Dann ist

$$u(x) + u(y) \geq 2^{n-1} + 2^{n-1} = 2^n$$ und damit $\boxed{c = 1}$

Ist nun aber $(x_0 + y_0 - s_0 - c)2^n = (1 - s_0)2^n \neq 0$

Dann ist $u(x) + u(y) \leq 2^n + 2^{n-1}(0 + \sum_{i=1}^{n-1} s_i 2^i)$

$$\leq 2^n + 2^{n-1} - 1$$

$$\Longrightarrow b_2(x) + x_0 2^n + b_2(y) + y_0 2^n \leq 2^n + 2^{n-1} - 1$$

$$\Longrightarrow b_2(x) + b_2(y) \leq 2^n + 2^{n-1} - 1 - 2^{n+1} = -2^{n-1} - 1$$

$\Rightarrow$ **Überlauf**

Das Ergebnis ist also korrekt, wenn kein Überlauf vorliegt.

Überlaufbedingungen

Wir erhalten also in allen 3 Fällen das korrekte Ergebnis, wenn kein Überlauf auftritt. Damit gilt also

Satz

Seien $x, y, s \in \boldsymbol{B}^n$ und $c \in \boldsymbol{B}$ mit $2^n c + u_n(s) = u_n(x) + u_n(y)$

dann gilt

$$b_{2,n}(s) = b_{2,n}(x) + b_{2,n}(y) \text{ und } b_{2,n}(s) \in [-2^{n-1} : 2^{n-1} - 1]$$

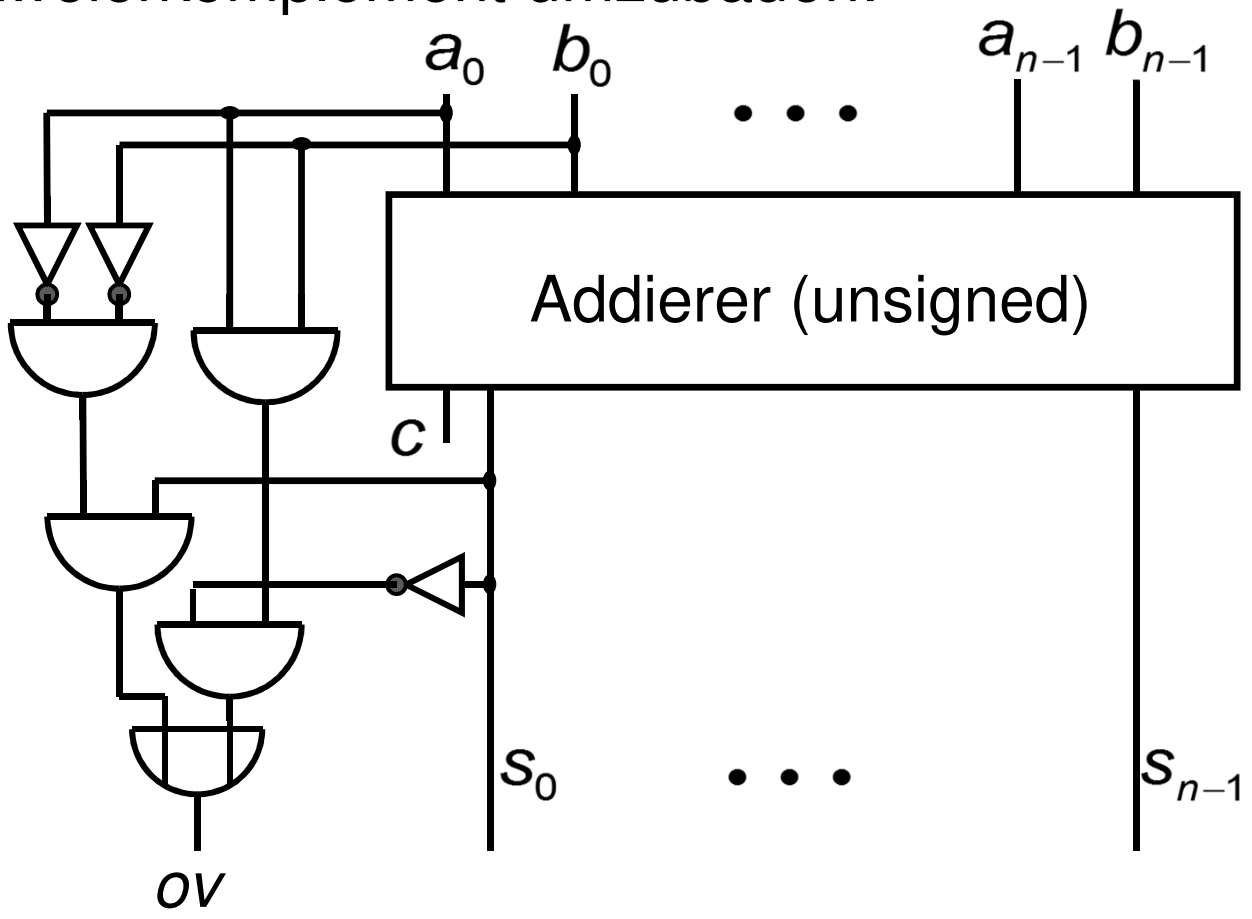
$$\Leftrightarrow$$

$$\overline{x}_0 \overline{y}_0 s_0 \vee x_0 y_0 \overline{s}_0 = 0$$

Beweis: durch die Diskussion schon geführt.

Zweierkomplement Addierer

Es ist also sehr leicht, einen unsigned Adder für das Zweierkomplement umzubauen:



Die Multiplikation

Auch hier können wir auf unsere Grundschulkenntnisse zurückgreifen. Es ist

$$u(a)\cdot u(b) = u(a)\cdot 2^{n-1}\sum_{i=0}^{n-1} b_i 2^{-i}$$

$$= \sum_{i=0}^{n-1} \underbrace{b_i 2^{n-1-i} u(a)}_{p_i}$$

Wir müssen also lediglich die *n* **Partialprodukte**

$$p_i := b_i 2^{n-1-i} u(a)$$

aufaddieren. Diese Produkte sind leicht zu berechnen, da die Multiplikation mit einer Zweierpotenz gerade einem Linksshift entspricht, sofern keine führenden Stellen herausgeschoben werden, d.h.

Shift und Multiplikation mit 2

Ist $a_0 = \cdots = a_{j-1} = 0$

Dann gilt

$$
\begin{aligned}
u(shl(a,j)) &= u(a_j,\ldots,a_{n-j-1},0,\ldots,0) \\
&= 2^{n-1}(0 + \sum_{i=0}^{n-j-1} a_{i+j}2^{-i} + 0) \\
&= 2^{n-1}(\sum_{i=0}^{j-1} 0 + \sum_{i=j}^{n-1} a_i 2^j 2^{-i}) \\
&= 2^{n-1}(\sum_{i=0}^{j-1} 2^j 0 + \sum_{i=j}^{n-1} a_i 2^j 2^{-i}) \\
&= 2^j 2^{n-1}(\sum_{i=0}^{j-1} 0 + \sum_{i=j}^{n-1} a_i 2^{-i}) \\
&= 2^j u(a)
\end{aligned}
$$

Multiplikation

Problem: Wir können beim Berechnen eines Partialproduktes selbst oder aber bei der Addition einen Überlauf erhalten, denn das Produkt von zwei *n*-bit Zahlen kann so groß werden wie

$$(2^n - 1)^2 = 2^{2n} - 2^{n+1} + 1$$

Es wird also ein 2*n*-bit Wort benötigt, um das Produkt exakt zu berechnen. Da man dies häufig auch möchte, betrachten wir also zunächst folgende

Aufgabe: Realisiere die Funktion

$$prod : \mathbf{B}^n \times \mathbf{B}^n \mapsto \mathbf{B}^{2n}$$

mit $u_{2n}(prod(a,b)) = u_n(a) \cdot u_n(b)$

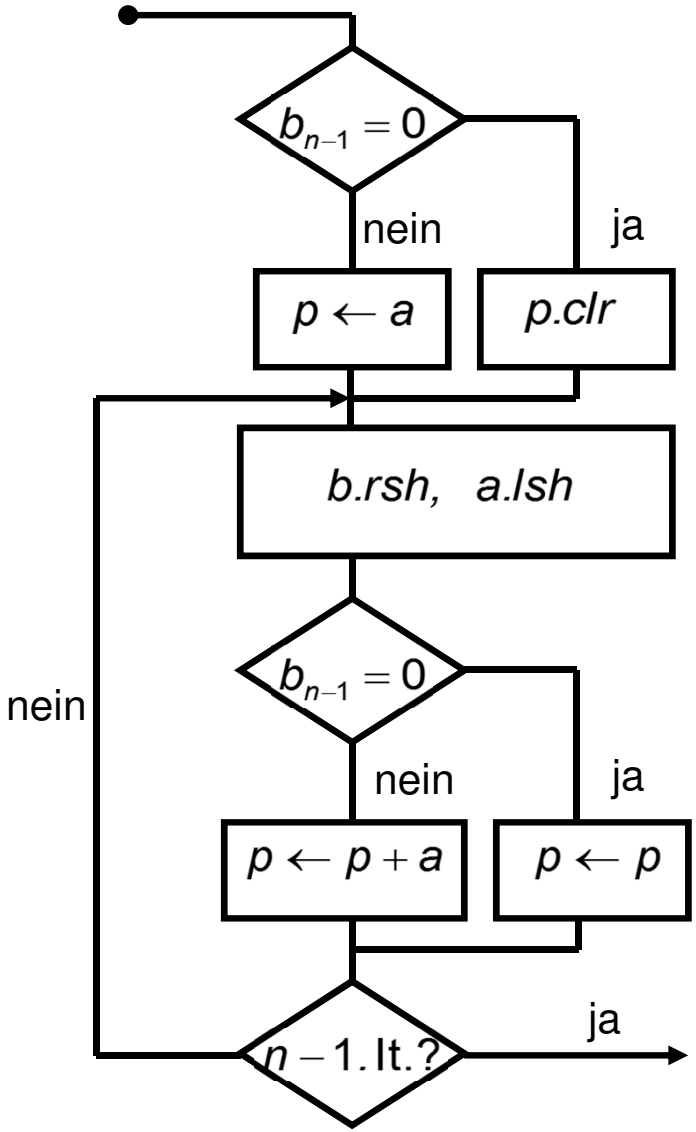
Multiplikation ff

Idee: Der einfachste Ansatz besteht nun darin, dass man einfach die Partialprodukte in *2n*-bit breiten Registern über einer 2*n*-bit ALU, die die Addition bereitstellt, iterativ aufsummiert. Baut man die Register auch noch als Schieberegister, dann kommt man ohne einen Shifter aus, weil nur jeweils um *1* geschoben werden muß. Solche einfachen Multiplizierwerke nennt man auch **Shift&Add** Multiplizierer.

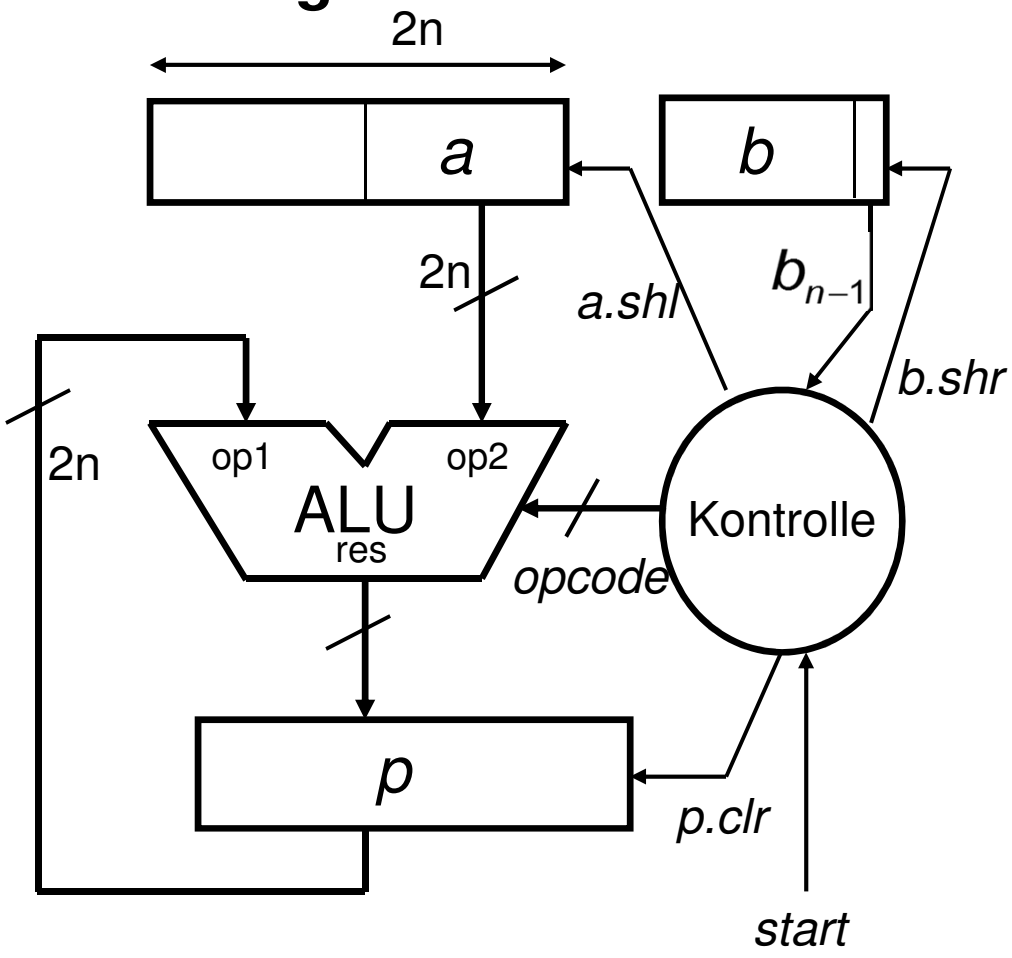
Wir nehmen also an, dass *a,b* in Schieberegistern der Breite 2*n* stehen, die mit den Steuerleitungen *b.rsh, a.lsh* um jeweils ein Bit nach rechts bzw. links geschoben werden können.

Die Partialprodukte summieren wir in einem 2*n* Register *p* auf. Dies ergibt folgendes Ablaufschema:

Shift&Add Multiplikation



Umsetzung:



Shift&Add Multiplikation ff

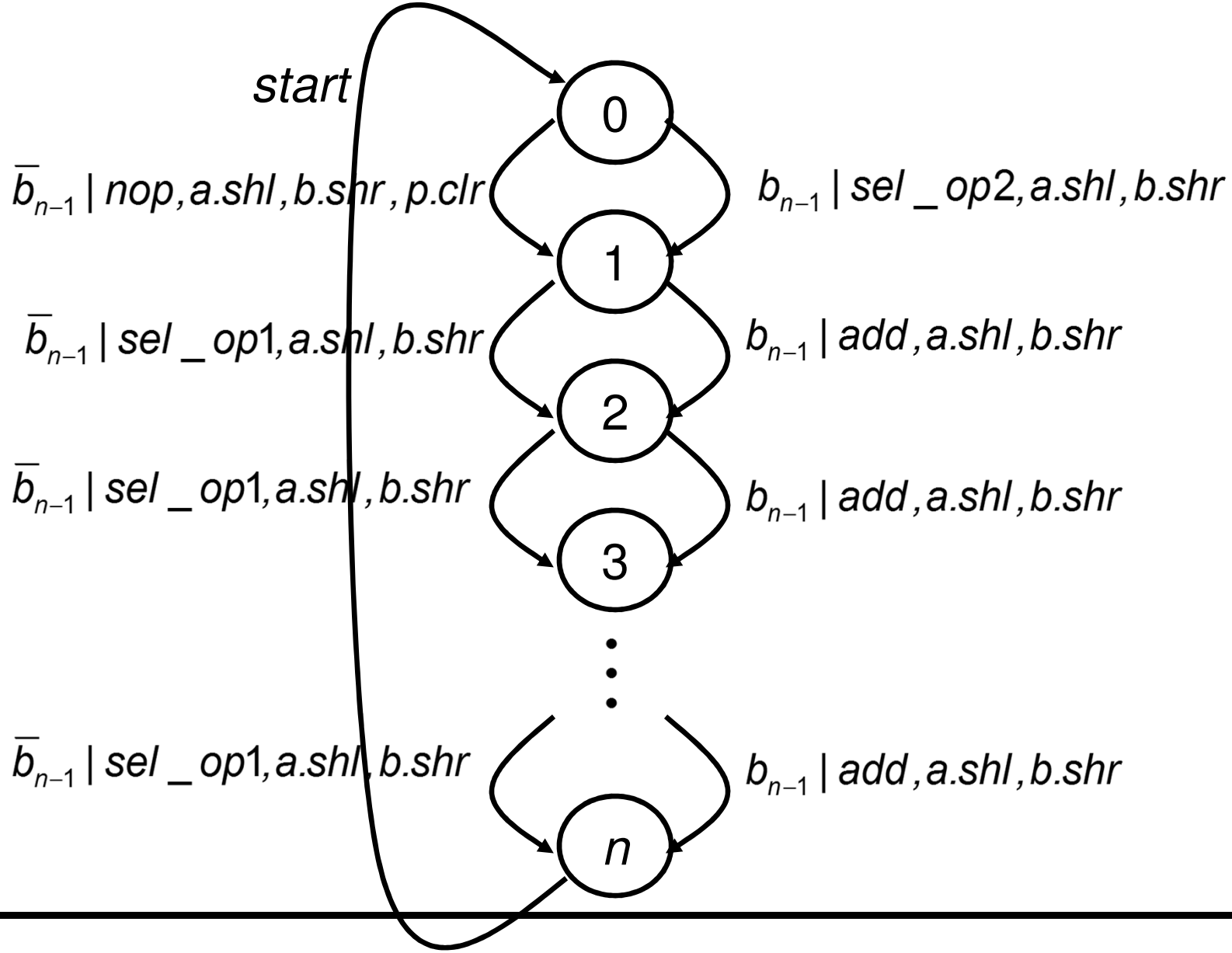
Wir stellen fest, dass wir für *b* sogar nur ein *n*-bit Register brauchen. Es bleibt noch der Entwurf der Kontrolle, die den Kontrollfluß aus unserem Diagramm durch eine entsprechende Folge von Steuersignalen erzeugen muss.

Folgende Zeichen sind zu erzeugen:

- die Steuersignale der Register (a.shl, b.shr, p.clr)
- die Opcodes der ALU, und zwar
  - *add* : Addiere (realisiert *p := p + a*)
  - *sel_op1* : *res := op1* (realisiert *p := p*)
  - *sel_op2* : *res := op2* (realisiert *p := a*)
  - *nop :* keine Operation

Damit hat die Kontrolle folgendes Aussehen:

Shift&Add: Die Kontrolle



Shift&Add Multiplikation ff

Die Initialisierung entspricht dem Übergang von Zustand *0* nach *1*, die Schleife den Übergängen von *1* bis *n-1*.

Man kann dieses Verfahren noch auf viele Weisen verbessern, aber es hat im Grunde den selben Nachteil wie die serielle Addition, es dauert *n* Taktperioden.

Wie erhält man eine schnellere Schaltung?

Man kann, wie bei der Addition auch, zunächst die sequentielle Schaltung „abrollen“. Dies würde zu einer Schaltung der Tiefe *n*T(Addierer)* führen (vgl. Ripple Carry Adder).

Es geht aber besser:

Carry Save Addition

Wir sind nicht gezwungen, jedes Zwischenergebnis

$$\sum_{i=j}^{n-1} \underbrace{b_i 2^{n-1-i} u(a)}_{p_i}$$

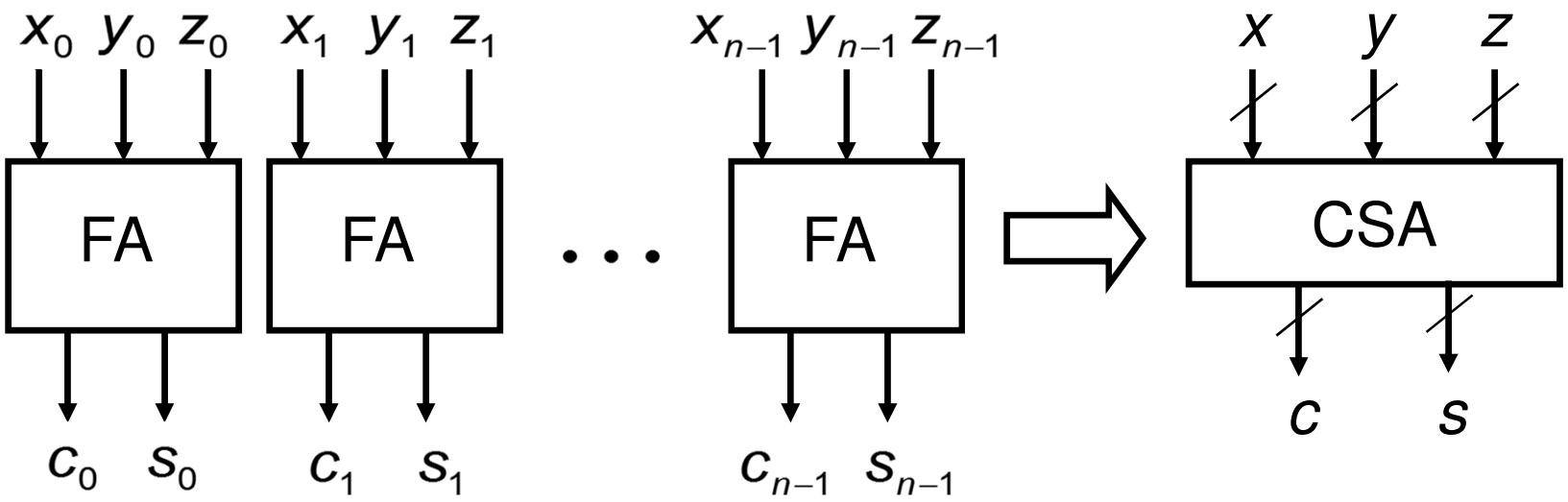
beim Akkumulieren der Partialprodukte durch ein einziges Maschinenwort darzustellen!

Beobachtungen:

• ein Volladdierer kann die Summe von *3* Bits durch eine zweistellige Binärzahl darstellen *(2c + s = x + y + z)*

• dieses Prinzip kann man durch folgende Schaltung, einen sog. **Carry Save Adder** auf *n*-bit Zahlen verallgemeinern:

Carry Save Adder



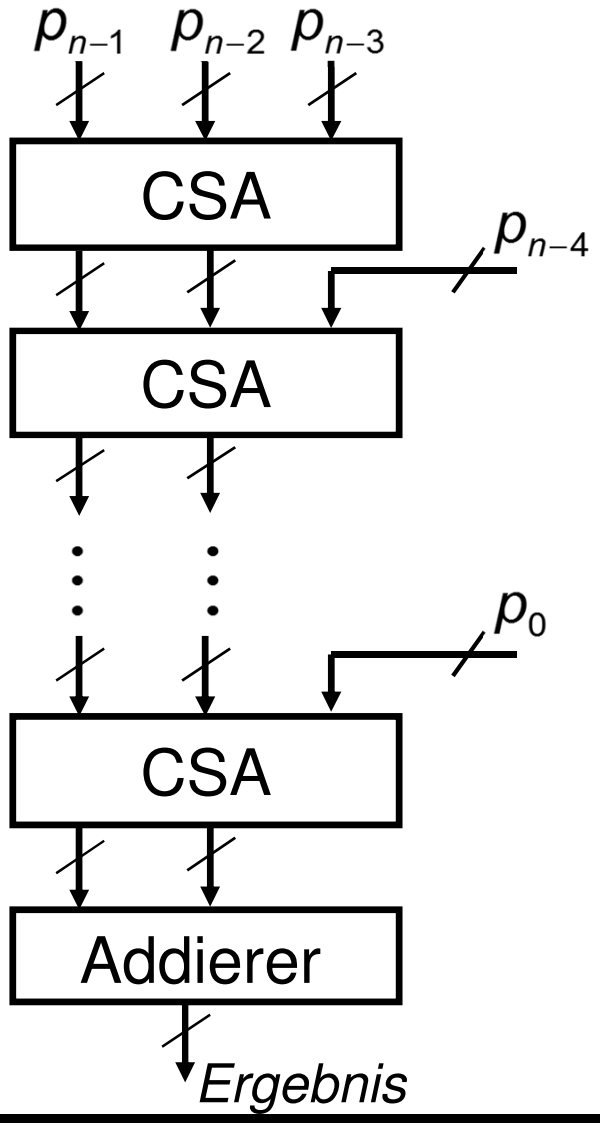
Für diese einfache Nebeneinanderschaltung von Voll-addierern gilt:

$$u(x) + u(y) + u(z) = 2^{n-1} \sum_{i=0}^{n-1} (x_i + y_i + z_i) 2^{-i}$$

$$= 2^{n-1} \sum_{i=0}^{n-1} (2c_i + s_i) 2^{-i} \qquad = 2u(c) + u(s)$$

Wir nennen sie einen Carry Save Adder der Breite *n*.

CSA Multiplizierer

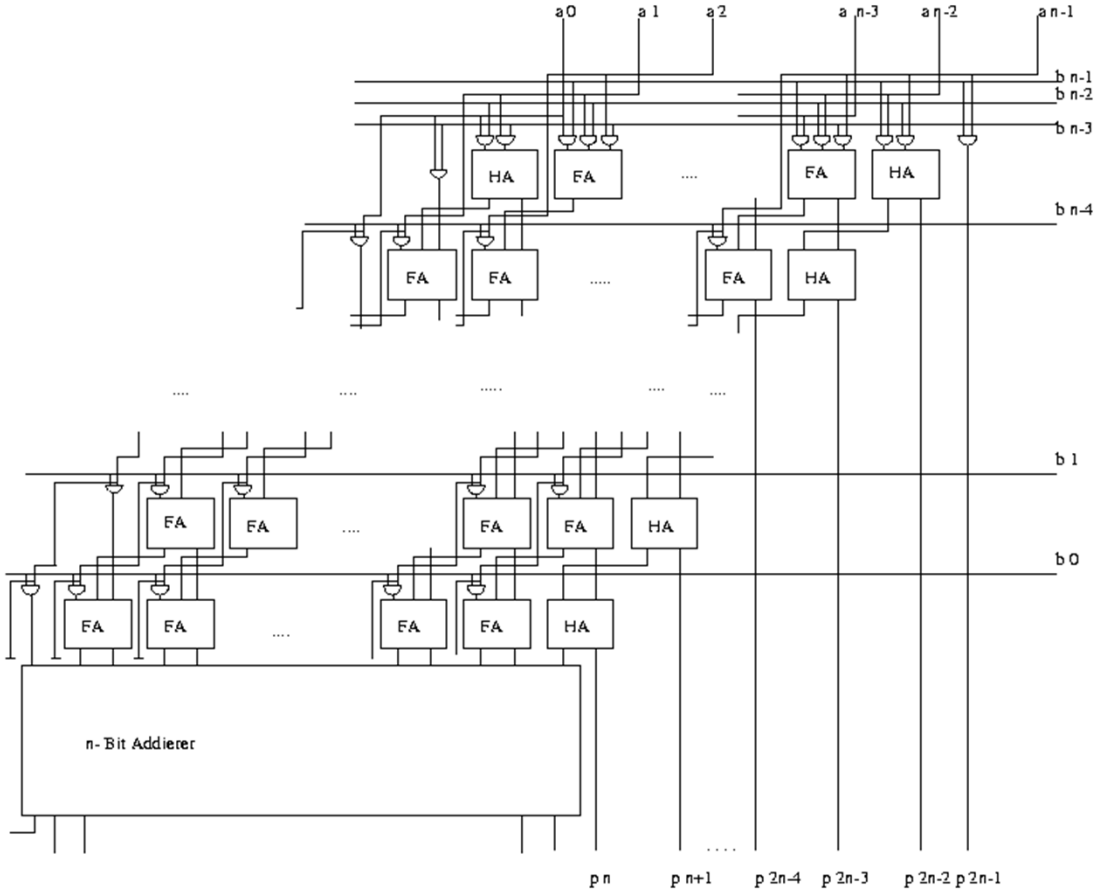


Wir können also leicht in Tiefe *(n-2)T(FA) + T(Addierer)* alle Partialprodukte aufsummieren, indem wir zunächst nur Carry Save Addierer der Breite *2n* einsetzen.

Die Schaltung hat Breite *2n*, allerdings benötigt man diese Breite nicht überall. Man kann sie noch weiter optimieren, indem man ausnutzt, dass die Partialprodukte nur Breite *n* haben, und überflüssige Volladdierer weglässt bzw. durch Halbaddierer ersetzt, wenn man nur *2* Bit addiert.

CSA Multiplizierer ff

Nimmt man die Erzeugung der Partialprodukte ebenfalls noch in die so optimierte Schaltung hinein, erhält man einen einfachen Parallelmultiplizierer nach folgender Skizze:



Wallace tree

Man kann sogar in logarithmischer Tiefe multiplizieren, indem man die Partialprodukte nicht nacheinander in Carry Save Addern aufaddiert, sondern parallel erzeugt und dann in einem „Baum“ von Carry Save Addern auf zwei Zahlen reduziert. Ein solches Schema heißt **Wallace tree**:

1. Stufe: reduziere alle $p_0, p_1, \ldots, p_{n-1}$ zu $z_0^{(1)}, z_1^{(1)}, \ldots, z_{2\lfloor \frac{n}{3} \rfloor + (n - 3\lfloor \frac{n}{3} \rfloor - 1)}^{(1)}$

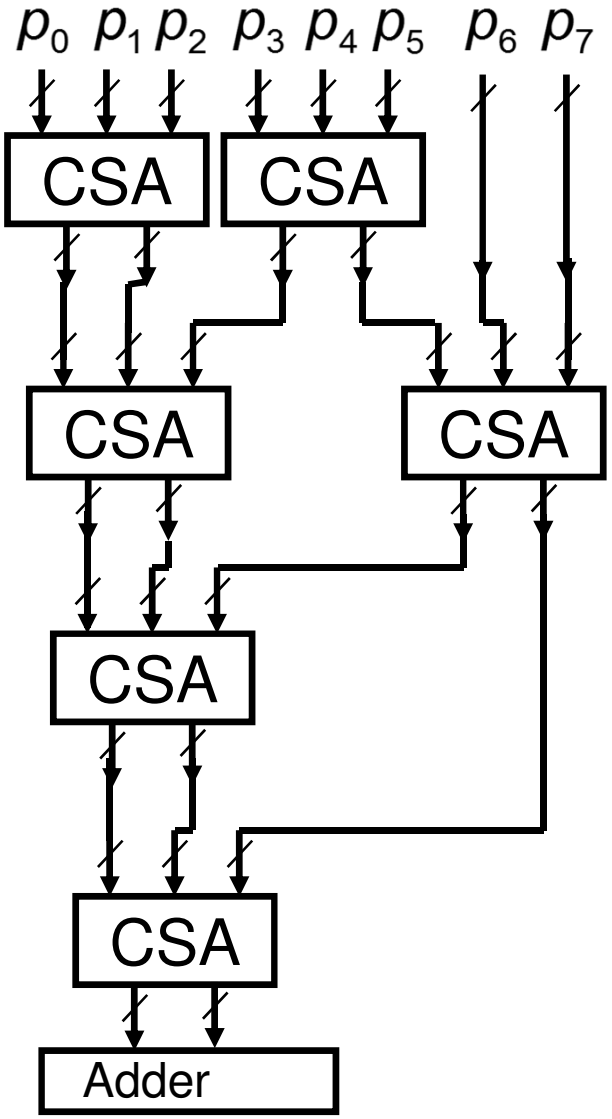
2. Stufe: reduziere alle $z_i^{(1)}$ zu $z_0^{(2)}, z_1^{(2)}, \ldots, z_{2\left\lfloor \frac{2\lfloor \frac{n}{3} \rfloor + (n - 3\lfloor \frac{n}{3} \rfloor - 1)}{3} \right\rfloor + \cdots}^{(2)}$

… Auf jeder Stufe sinkt die Zahl der Zwischenergebnisse um den Faktor *~3/2*, d.h. nach etwa

$$\sim \log_{\frac{3}{2}} n$$

Stufen brauchen wir nur noch einen schnellen Addierer.

Beispiel: 8 - Bit Wallace tree



Ausblick:

Es gibt noch eine ganze Fülle von Techniken zur schnellen Multiplikation, deren Behandlung den Rahmen dieser einführenden Veranstaltung sprengt. Schaltkreise dazu sind recht groß und nicht immer leicht systematisch zu konstruieren. Daher nimmt man die Multiplikation in der Regel aus der ALU heraus und realisiert sie in einer extra Funktionseinheit.

Ähnliches gilt für die Division: Auch auf die Division lässt sich die Schulmethode übertragen (sukzessive Subtraktion eines (binären) Vielfachen des Divisors). Allerdings werden hier die Verfahren recht trickreich, will man schnell dividieren.

(trickreich → fehleranfällig ?!? vgl. Pentium Bug)