3. Mikroarchitekturen

Peter Marwedel
Informatik 12
TU Dortmund

2014/05/02

Gegenüberstellung der Definitionen

Programmierschnittstelle	Interner Aufbau
Externe Rechnerarchitektur	Interne Rechnerarchitektur
Architektur	Mikroarchitektur
Rechnerarchitektur	Rechnerorganisation

Die externe Rechnerarchitektur definiert

- Programmier- oder Befehlssatzschnittstelle
- engl. *instruction set architecture, (ISA)*
- eine (reale) Rechenmaschine bzw.
- ein *application program interface (API)*.

Executables (Binärprogramme)

Betriebssystem

Befehlsschnittstelle (*instruction set architecture*)

Mikroarchitektur

Gatter

3.1 Notation: VHDL-Datentypen

Vorteile einer Hardware-Beschreibungssprache (engl. *hardware description language*, HDL):

- präzise Spezifikation,
- Möglichkeit der Simulation,
- Kommunikation zwischen Entwicklern,
- automatisierte Erzeugung/ Überprüfung von Designs,
- Dokumentation des Arbeitsergebnisses,
- Beschleunigung des Entwurfsprozesses.

Verbreitet: VHDL *(VHSIC Hardware Description Language)*
VHSIC = *very high speed integrated circuit*.
Unterscheidung zwischen Folgen von Bits und deren Interpretion als Wert in einem anderen Bereich.

Datentypen `integer` und `natural`

`integer` und dafür übliche arithmetische Operationen **`+, <`** usw. sind vordefiniert.

Ableitung mittels subtype-Definitionen:

```
subtype natural is integer range 0 to integer'high;

subtype positive is integer range 1 to integer'high;
```

Abgeleitete Typen sind zuweisungskompatibel.

Obere Grenzen des darstellbaren Bereichs sind implementationsabhängig.

`integer'high` = obere Grenze des darstellbaren Zahlenbereichs.

Datentypen `boolean, bit`

`boolean` ist vordefiniert:

```
type boolean is (False, True);
```

`boolean` wird also durch Aufzählung der möglichen Werte definiert.

`bit` ist vordefiniert durch

```
    type bit is ('0', '1');
```

Literale des Typs **bit** werden durch einfache Anführungszeichen

bezeichnet.

Datentyp bit_vector

bit_vector ist vordefiniert durch

```
type bit_vector is array (natural range <>) of bit;
```

<> = ausgelassener Index eines *unconstrained array*,
(Grenzen durch einen Bereich der natürlichen Zahlen später festzulegen).

Beispiel:

```
variable instruction : bit_vector (31 downto 0);
```

downto: absteigender Indexbereich.

In dieser Vorlesung: untere Indexgrenze meist = 0

Literale in doppelten Anführungszeichen:

```
"01010101"
```

Für **bit** und **bit_vector** sind die üblichen logischen Operationen (**AND** usw.) vordefiniert.

a'left ist der am weitesten links stehende Index eines Vektors.

Konkatenation

Konkatenation:

"01010101" & ’0’

"01010101" & "01010101"

’1’ & ’0’

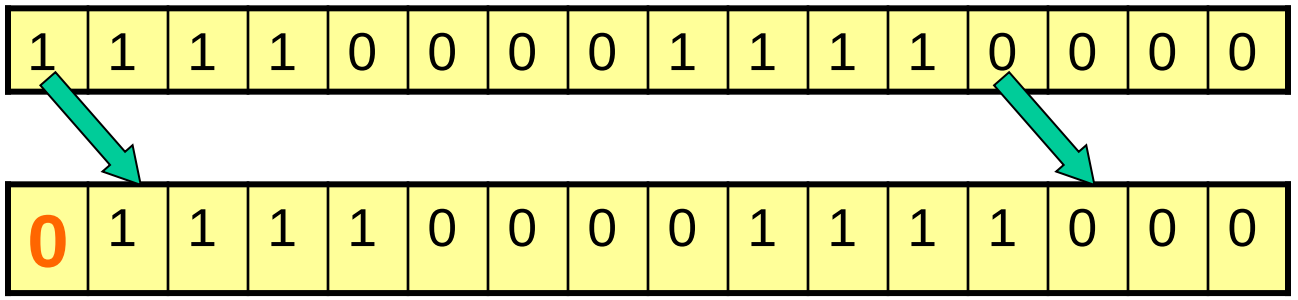
3.2 Realisierung elementarer Datentypen
3.2.1 Operationen auf Bitvektoren

Die meisten Befehlssätze unterstützen Bitvektor-Operationen auf begrenzter Länge (häufig: ein Wort).

3.2.1.1 Schiebeoperationen

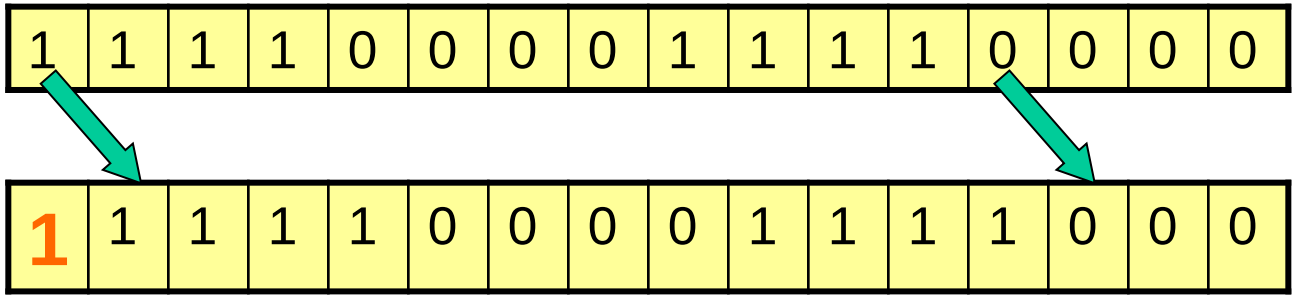
Hier: *shift right logical, shift left logical, shift right arithmetical, shift left arithmetical*:

```
srl(a) = '0' & a (a'left downto 1)
```



Schiebeoperationen

```
sra(a) = a (a'left) & a (a'left downto 1)
```



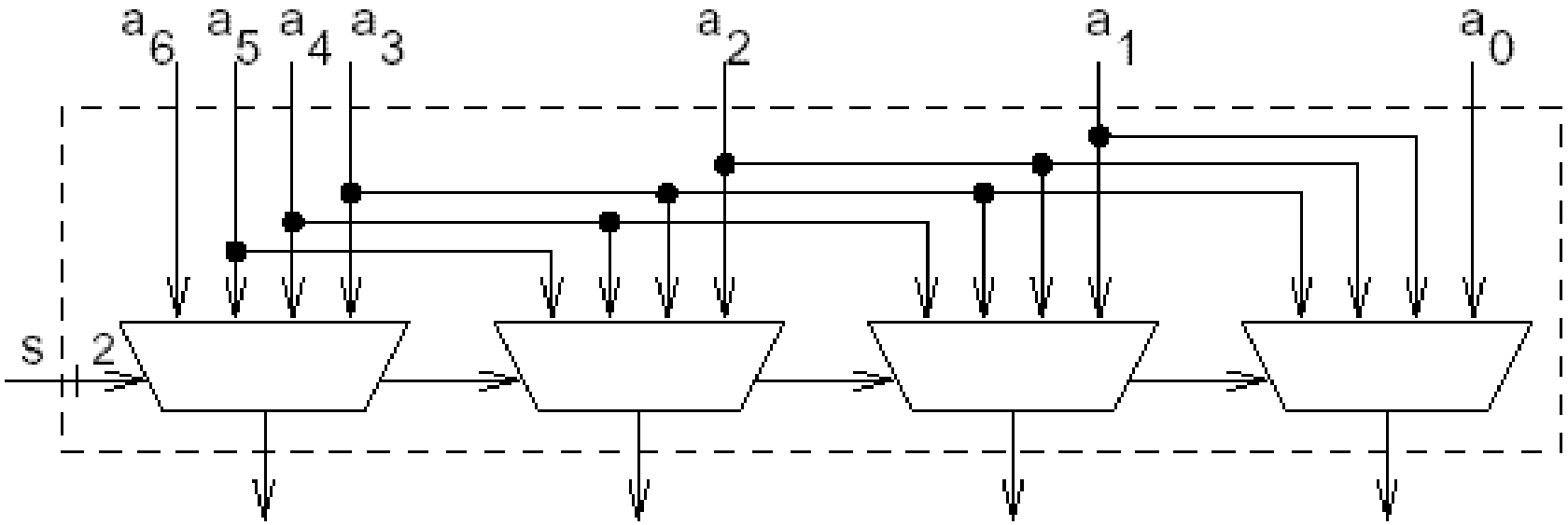

```
sll(a) = a (a'left-1 downto 0) & '0'
```

```
sla(a) = a (a'left)& a(a'left-2 downto 0)&'0'
```

Erklärung von Schiebeoperationen um n Stellen durch n-maliges Schieben um 1 Stelle.

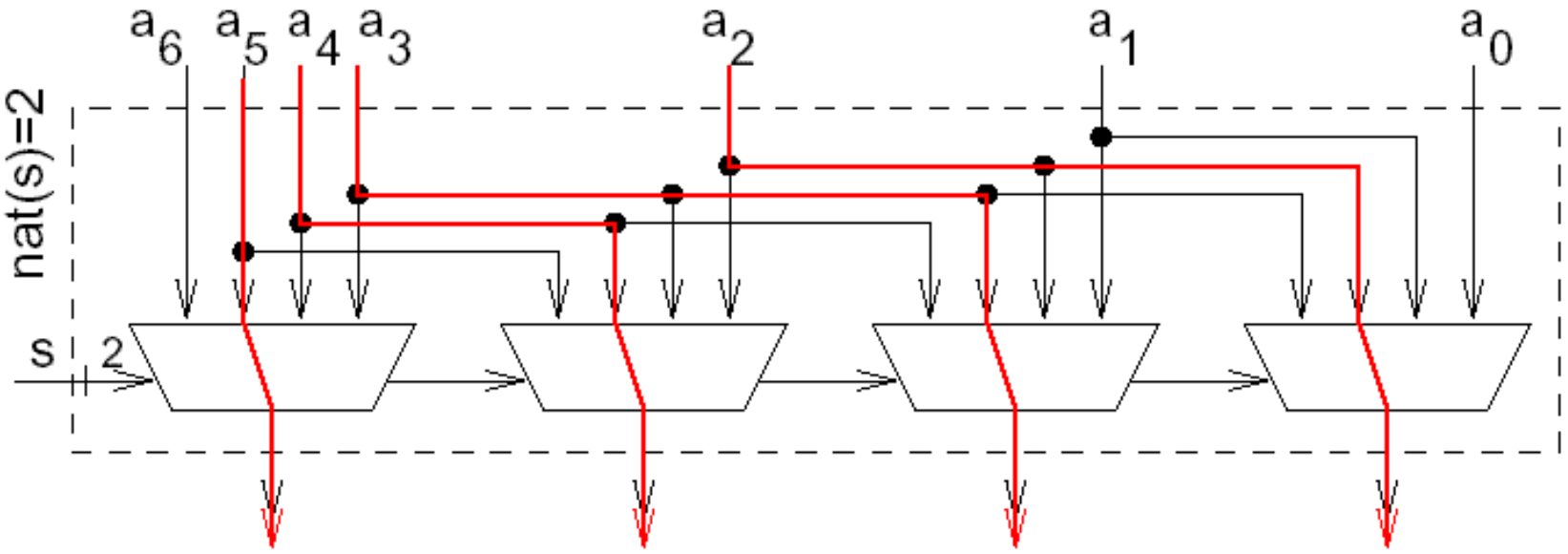
Realisierung der Operationen auf Bitvektoren: Schieben

- Realisierung mit ***barrel shiftern*.**
- Schaltskizze soll ähnlich wie das Schieben eines Fasses aussehen können (nie selbst gesehen)
- Elementarbausteine, z.B. 4 Ausgänge, Schieben: 0-3 Stellen:



Realisierung der Operationen auf Bitvektoren: Schieben

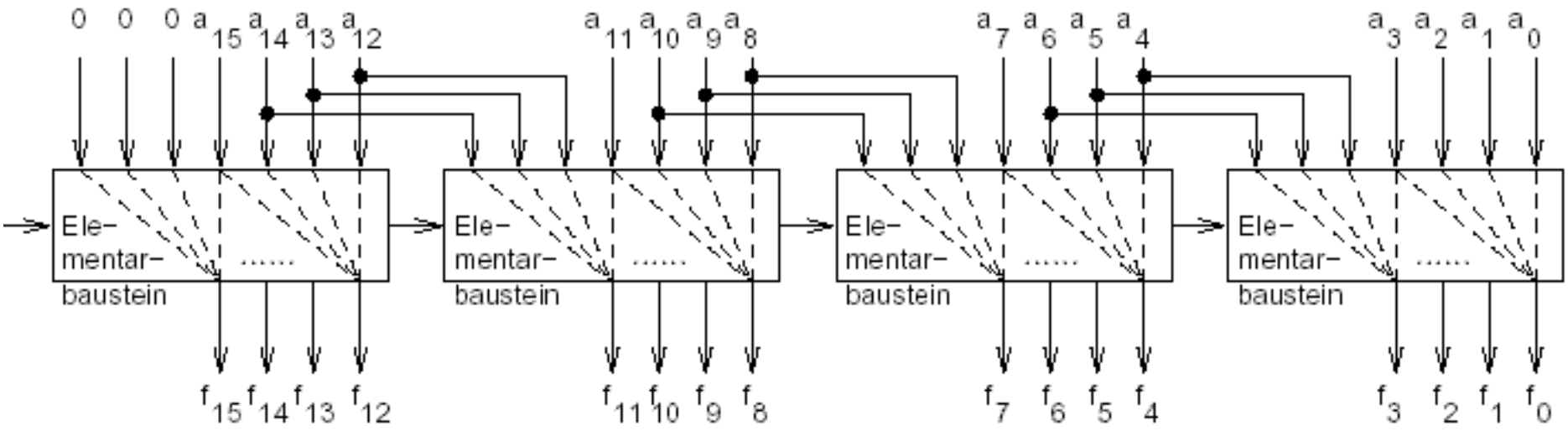
Elementarbaustein, 4 Ausgänge, Schieben: 0-3 Stellen:



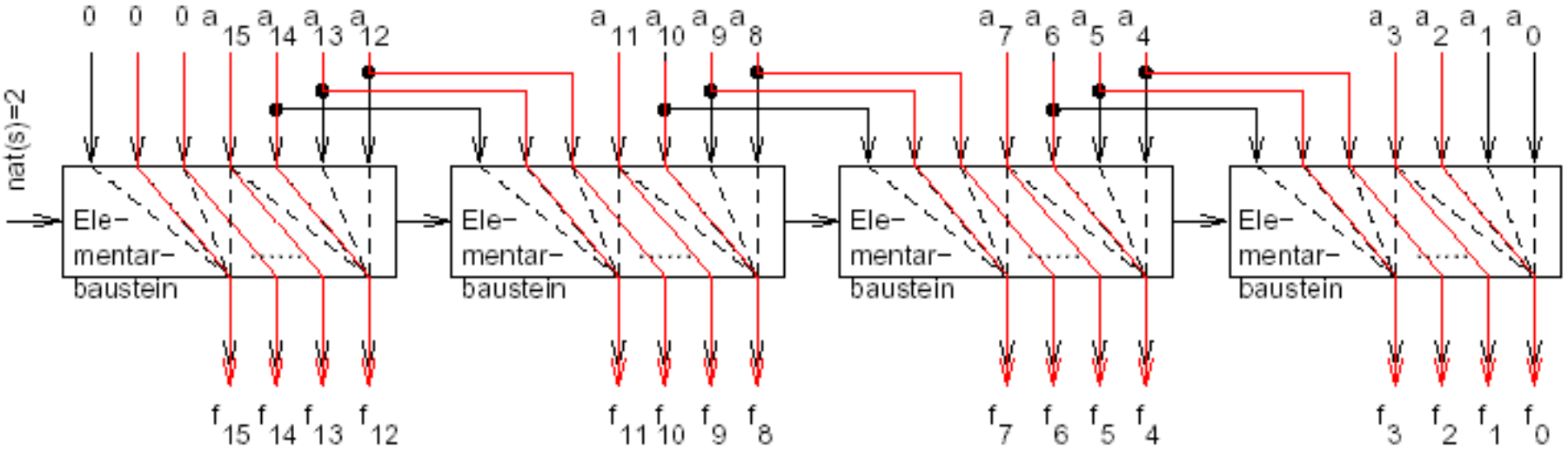
Anwendung: u.a. Normalisierung von Gleitkommazahlen

Simulation: http://tams-www.informatik.uni-hamburg.de/applets/hades/webdemos/10-gates/60-barrel/shifter8.jnlp

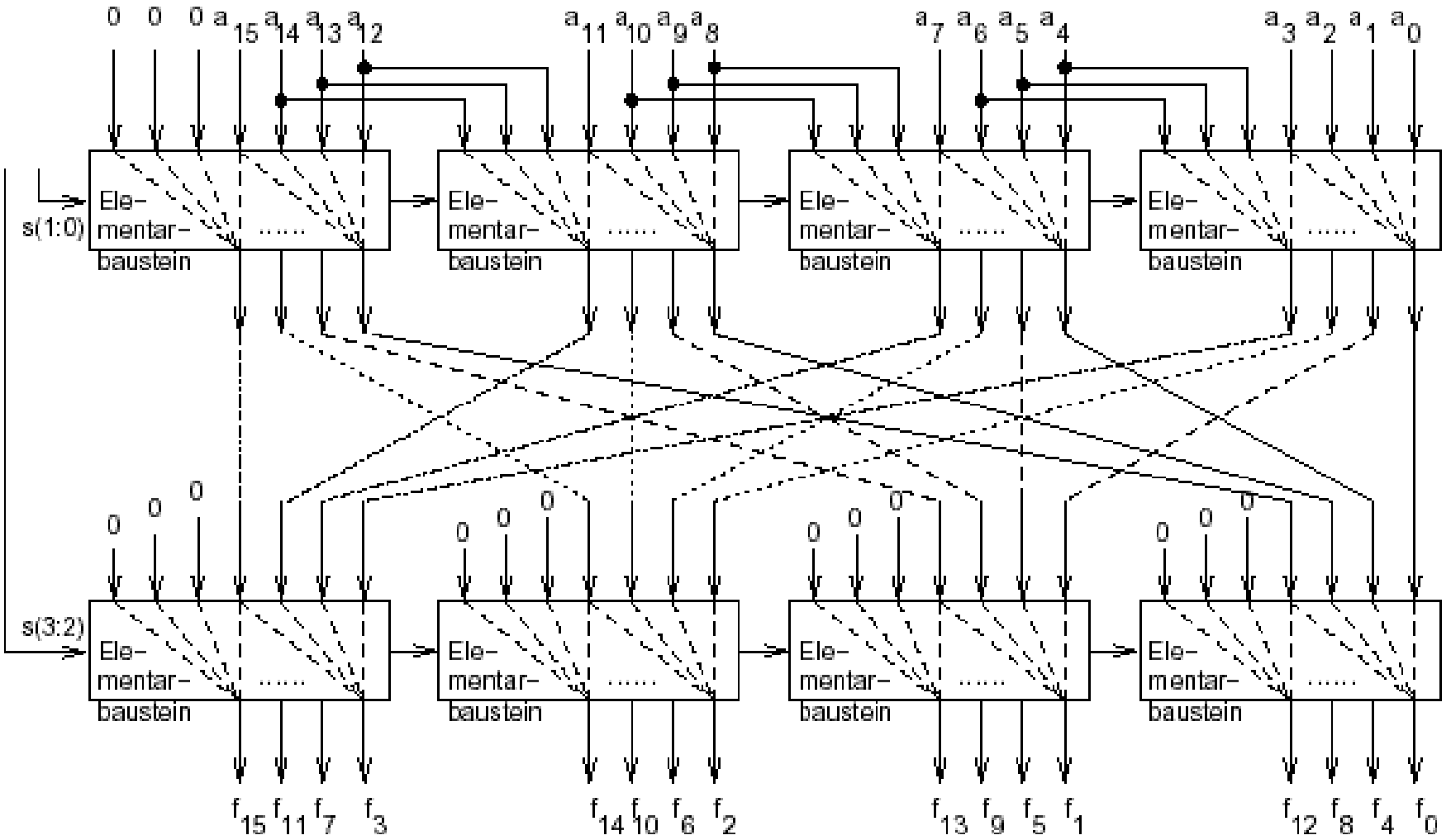
Barrelshifter mit 16 Ausgängen



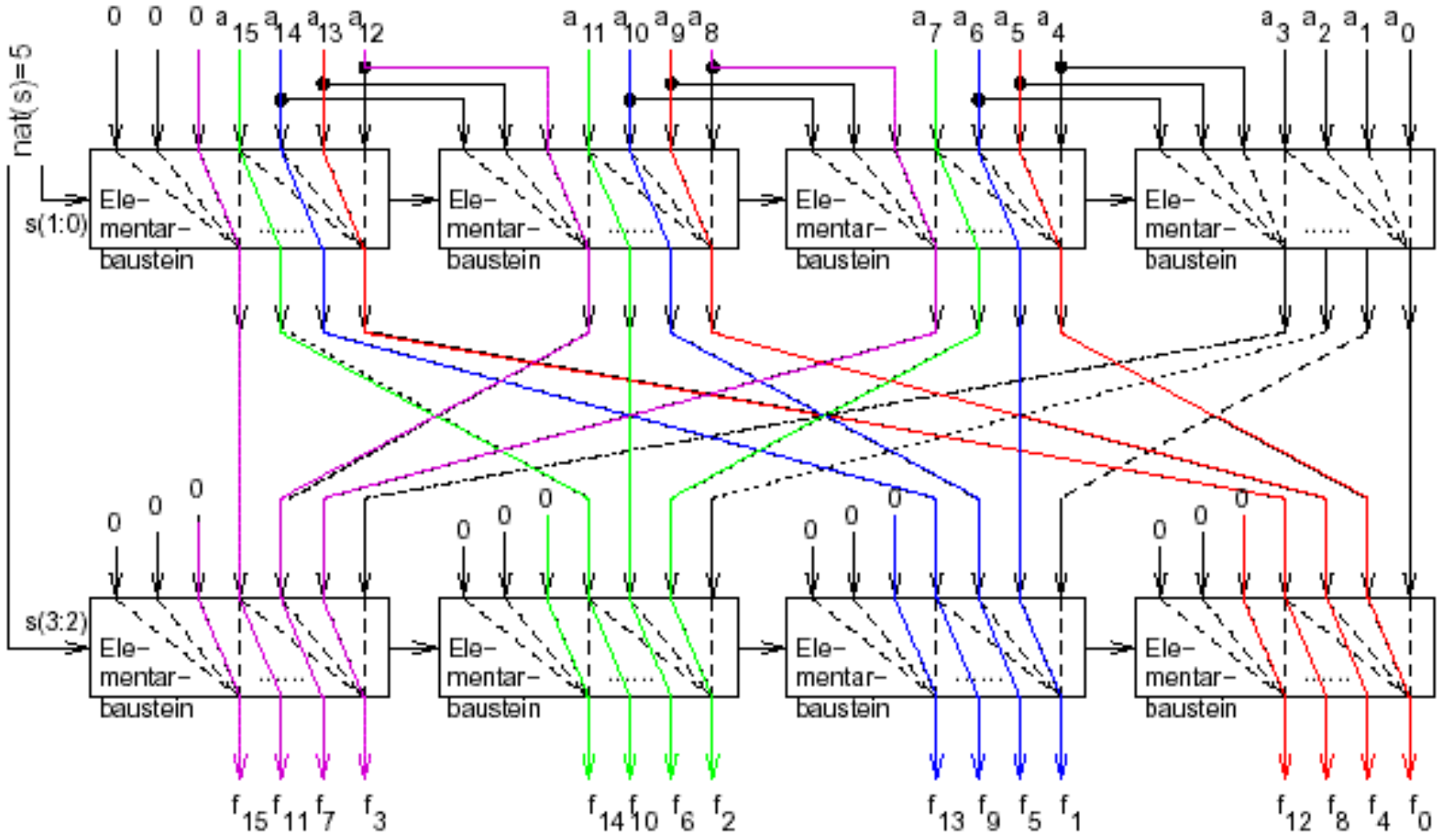
Barrelshifter mit 16 Ausgängen



Barrelshifter für das Schieben um 0 bis 15 Stellen:



Barrelshifter für das Schieben um 0 bis 15 Stellen:



3.2.2 Natürliche Zahlen

Übliche Interpretation von Bitvektoren als natürliche Zahlen:

$$\texttt{nat(a)} = \sum_{i=0}^{a'left} a_i \cdot 2^i$$

Beispiel:
`nat("1000") =` 8.

Addition

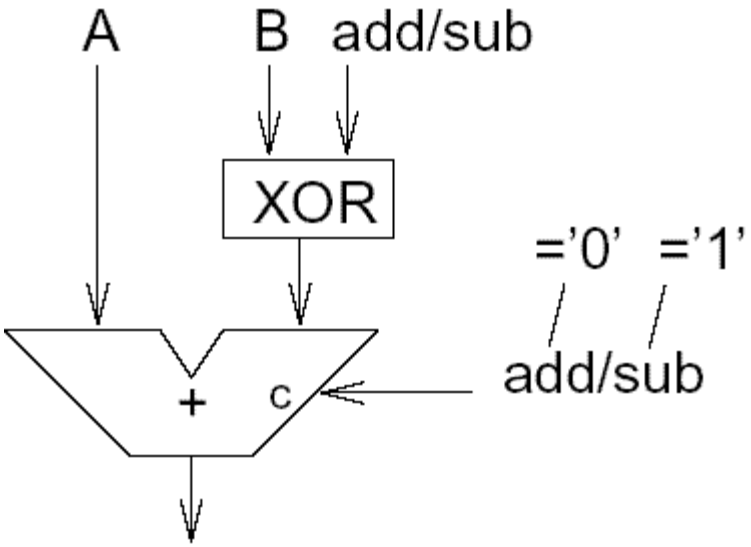
Ergebnis: Summe natürlicher Zahlen, soweit dies bei fester Datenwortlänge möglich ist.

Bezeichnungen:

- Argumente durch Bitvektoren $a = (a_{n-1} , \ldots , a_0)$ und $b = (b_{n-1} , \ldots , b_0)$ repräsentiert.
- Ergebnis durch $f = (f_{n-1} , \ldots , f_0)$ dargestellt
- $\forall\ i , 0 \leq i \leq n : c_i :$ Übertrag in die Stelle i hinein

Subtraktion

Für das Zweierkomplement gilt:
$-B = \neg B + 1$, also $A - B = A + \neg B + 1$



Die Subtraktion kann also einfach auf die Addition zurückgeführt werden.

(siehe arithmetisch/logische Einheiten (ALUs)).

Beispiel einer arithmetisch/logischen Einheit: ”74181”

Steuersignale S3,S2,S1,S0 und M.

Steuerkode S3	S2	S1	S0	$M =' 1'$ (logische Op.)	$M =' 0'$ (arithm. Op.) $c_0 =' 0'$	$M =' 0'$ $c_0 =' 1'$
'0'	'0'	'0'	'0'	$F := \overline{A}$	$F := A$	$F := A + 1$
'0'	'0'	'0'	'1'	$F := \overline{A \vee B}$	$F := A \vee B$	$F := (A \vee B) + 1$
'0'	'0'	'1'	'0'	$F := \overline{A} \wedge B$	$F := A \vee \overline{B}$	$F := (A \vee \overline{B}) + 1$
'0'	'0'	'1'	'1'	$F := 0$	$F := -1_{10}$	$F := 0$
.	.	.	.	...	$F := ...$	$F := ...$
'0'	'1'	'1'	'0'	$F := A \neq B$	$F := A - B - 1$	$F := A - B$
.	.	.	.	...	$F := ...$	$F := ...$
'1'	'0'	'0'	'1'	$F := A = B$	$F := A + B$	$F := A + B + 1$
.	.	.	.	...	$F := ...$	$F := ...$
'1'	'1'	'1'	'0'	$F := A \vee B$	$F := (A \vee \overline{B}) + A$	$F := (A \vee \overline{B}) + A + 1$
'1'	'1'	'1'	'1'	$F := A$	$F := A - 1$	$F := A$

Überläufe

Wir möchten gerne wissen: wann ist

(1) $\text{nat}(a) + \text{nat}(b) \geq 2^n$?

(2^n = erste nicht mehr durch f darstellbare Zahl)

Aus (1) folgt:

(2) $$\sum_{i=0}^{n-1} a_i * 2^i + \sum_{i=0}^{n-1} b_i * 2^i \geq 2^n$$

Ungleichung erfüllt, wenn bei der Addition ein Übertrag in die Stelle n hinein entsteht.

Bei der Addition **natürlicher** Zahlen ist der **Überlauf** c_f gleich dem **Übertrag** c_n in die nächste Stelle.

Überläufe

Für c_n gilt:

$$cf_+ = c_n = (a_{n-1}b_{n-1}) \vee ((a_{n-1} \textbf{ xor } b_{n-1})c_{n-1})$$

(**xor** wegen des 1. Terms durch $\vee$ ersetzbar)

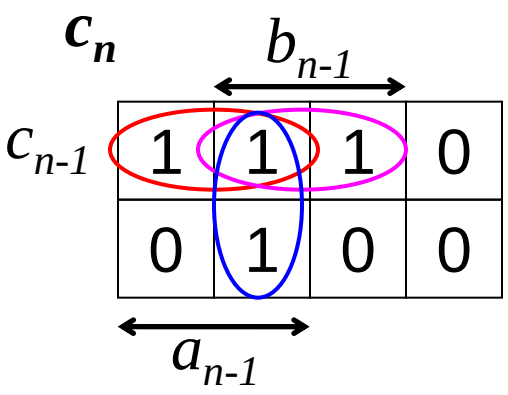
$$= (a_{n-1}b_{n-1}) \vee ((a_{n-1} \vee b_{n-1})\, c_{n-1})$$

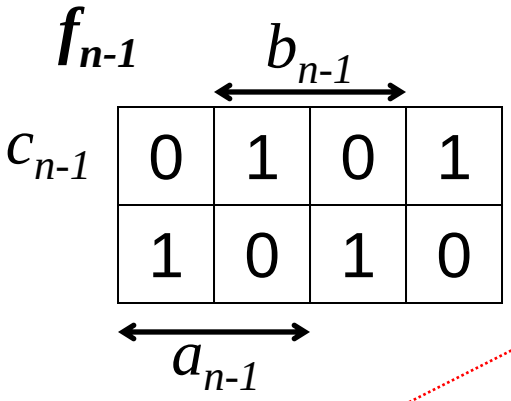
$$= (a_{n-1}b_{n-1}) \vee (a_{n-1}\, c_{n-1}) \vee (b_{n-1}c_{n-1})$$

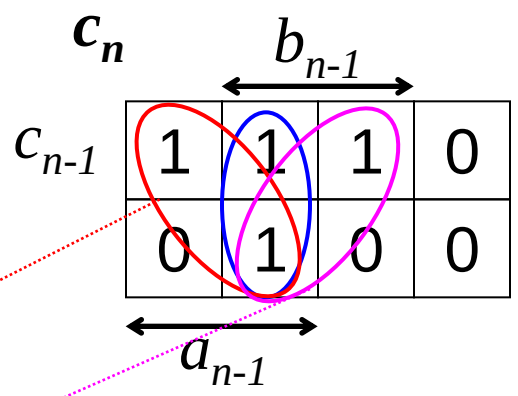
c_{n-1} meist nur intern verfügbar
→ Ersatz durch verfügbaren Wert!

Überläufe

$$c_n = (a_{n-1} b_{n-1}) \vee (a_{n-1}\, c_{n-1}) \vee (b_{n-1} c_{n-1})$$







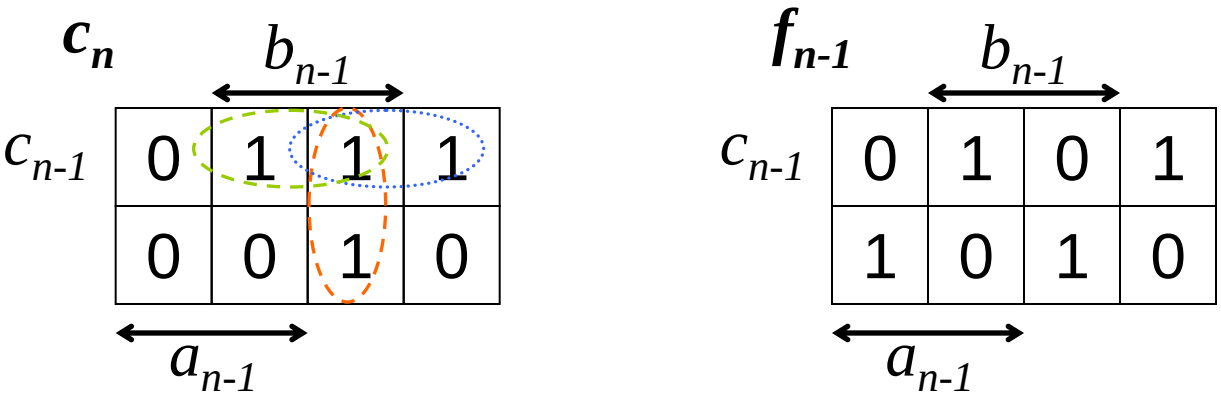
$$c_n = (a_{n-1} b_{n-1}) \vee (a_{n-1}\, \overline{f_{n-1}}) \vee (b_{n-1}\, \overline{f_{n-1}})$$

$\overline{f_{n-1}}$ extern bekannt:

(z.B. Abfrage <0 für Zweierkomplementinterpretation).

Subtraktion

Differenz natürlicher Zahlen, soweit darstellbar.
Überläufe: Der Überlauf $cf\_$ ist gleich dem 'Borgebit' c_n in die erste nicht mehr darstellbare Stelle:



Gemäß linkem Teil: $cf\_ = (\overline{a_{n-1}}b_{n-1}) \vee (\overline{a_{n-1}}c_{n-1}) \vee (b_{n-1}c_{n-1})$

Statt internem c_{n-1} extern bekanntes f_{n-1} verwenden!

Aus KV-Diagramm: $cf\_ = (\overline{a_{n-1}}b_{n-1}) \vee (\overline{a_{n-1}}f_{n-1}) \vee (b_{n-1}f_{n-1})$

Größenvergleich

Der Größenvergleich basiert häufig auf den Inhalten der *Condition-Code* Register (*flags*) nach Ausführen einer Subtraktion.

Dazu gehören typischerweise:

carryflag c_n (=*cf*_ nach einer Subtraktion)

zeroflag zf : zf ='1' $\Leftrightarrow$ $\forall i \in [0..n\text{-}1]$: f_i='0'

signflag sf : sf ='1' $\Leftrightarrow$ $f_{n\text{-}1}$='1'

Berechnung der Relationen anhand der *flag*-Register

Berechnung der Vergleichsergebnisse bei Interpretation als natürliche Zahlen nach einer Subtraktion aus zf und $cf\_ = c_n$:

$a<b \Leftrightarrow (a-b)<0 \Leftrightarrow cf\_ = '1'$

$a \geq b \Leftrightarrow \neg((a-b)<0) \Leftrightarrow cf\_ = '0'$

$a>b \Leftrightarrow ((a-b) \geq 0) \wedge (a-b) \neq 0 \Leftrightarrow (cf\_ = '0') \wedge (zf = '0')$

$a \leq b \Leftrightarrow \neg(a>b) \Leftrightarrow (cf\_ = '1') \vee (zf = '1')$

Typische Anwendung bei x86- & ARM-Architektur; nicht so bei der MIPS-Architektur.

ARM instruction set tests [ARM]	
Unsigned higher or same	C set
Unsigned lower	C clear
Unsigned Higher	C set and Z clear
Unsigned Lower or Same	C clear or Z set
Not Equal	Z clear

ARM (+6502): C='0' unter der „Borge“-Bedingung. Gegenüber Mehrzahl der Maschinen invertiert.

Multiplikation

→ Siehe Rechnerstrukturen,
Wallace-Tree-Multiplizierer

3.2 Realisierung elementarer Datentypen
3.2.3 Ganze Zahlen

Interpretation der Bitvektoren

Annahme:
Darstellung im **Zweierkomplement**, Interpretation mittels **`int`**:

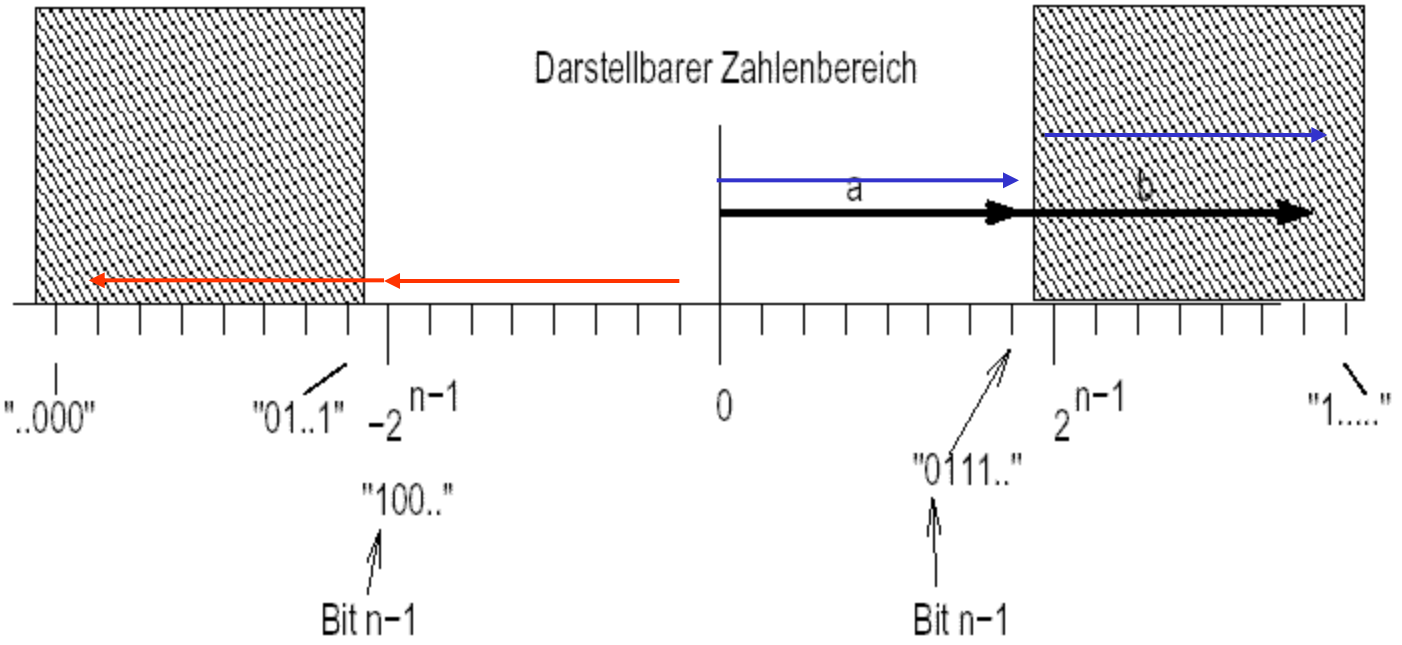
$$\texttt{int(a)} = -2^{a'left} \cdot a_{a'left} + \sum_{i=0}^{a'left-1} a_i \cdot 2^i$$

Beispiele:

```
-int("1000") = -8.
-int("1001") = -7.
```

Überläufe bei der Addition

Mittels $a_{n-1}..a_0$ bzw. $b_{n-1}..b_0$ darstellbar:



Überläufe: wenn beide Operanden das gleiche und das Ergebnis das entgegen gesetzte Vorzeichen haben:

Überlauf

a_{n-1}	b_{n-1}	Überlauf unter der Bedingung
0	0	$f_{n-1} = 1$
0	1	nicht möglich
1	0	nicht möglich
1	1	$f_{n-1} = 0$

Auch für **`int(a)`** = -2^{n-1} bzw. **`int(b)`** = -2^{n-1}. Es gilt also:

$$overflow\_add(a,b) = (a_{n-1} \equiv b_{n-1}) \wedge (a_{n-1} \textbf{ xor } f_{n-1})$$

overflow_add in vielen Rechnern in weiterem Condition-Code-Register (overflow_flag, ov) gespeichert .

Reaktion mittels *branch if_overflow*-Befehls.

Keine Unterscheidung zwischen **integer** und **natural** beim *add*-Befehl, beim *branch*-Befehl richtigen Datentyp wählen!

Sättigungsarithmetik: Datentypen beim *add*-Befehl bekannt.

Subtraktion

a_{n-1}	b_{n-1}	Überlauf unter der Bedingung
0	0	nicht möglich
0	1	$f_{n-1} = 1$
1	0	$f_{n-1} = 0$
1	1	nicht möglich

Überlauf, wenn beide Operanden entgegengesetztes Vorzeichen haben und das Ergebnis f_{n-1} ... f_0 ein anderes Vorzeichen als der erste Operand hat:

$$overflow\_sub(a,b)=(a_{n-1} \textbf{ xor } b_{n-1}) \wedge (a_{n-1} \textbf{ xor } f_{n-1})$$

Größenvergleich

Für Zahlen in Zweierkomplementdarstellung:

wegen: $overflow\_sub(a,b) = (a_{n-1} \texttt{ xor } b_{n-1}) \wedge (a_{n-1} \texttt{ xor } sf)$
$= (\overline{a_{n-1}} \wedge b_{n-1} \wedge sf) \vee (a_{n-1} \wedge \overline{b_{n-1}} \wedge \overline{sf})$

folgt:

$sf = \texttt{'1'} \rightarrow overflow\_sub = \overline{a_{n-1}} \wedge b_{n-1}$
$sf = \texttt{'0'} \rightarrow overflow\_sub = a_{n-1} \wedge \overline{b_{n-1}}$

ARM instruction set tests [ARM]	
Signed greater Than or Equal	*N set and V set, or N clear and V clear (N=V)*
Signed Less Than	*N set and V clear, or N clear and V set (N != V)*
Signed Greater Than	*Z Clear , and either N set and V set, or N clear and V clear (Z=0, N=V)*
Signed Less Than or Equal	*Z set, or N set and V clear, or N clear and V set (Z=1, N != V)*

☞:

overflow_sub	sf	Kommentar	Ergebnis
0	0	Kein Überlauf, F positiv, $0 \le a\text{-}b \le 2^{n-1}\text{-}1$	$a \ge b$
0	1	Kein Überlauf, F negativ, $-2^{n-1} \le a\text{-}b < 0$	$a < b$
1	0	$(a_{n-1}=1) \wedge (b_{n-1}=0)$: a negativ, b positiv	$a < b$
1	1	$(a_{n-1}=0) \wedge (b_{n-1}=1)$: a positiv, b negativ	$a > b$

Daraus ergibt sich:

$a<b \Leftrightarrow (overflow\_sub \texttt{ xor } sf)$
$a \le b \Leftrightarrow (a<b) \vee (a=b) \Leftrightarrow (overflow\_sub \texttt{ xor } sf) \vee zf$
$a>b \Leftrightarrow \neg(a \le b) \Leftrightarrow \neg((overflow\_sub \texttt{ xor } sf) \vee zf)$
$a \ge b \Leftrightarrow \neg(a<b) \Leftrightarrow overflow\_sub \equiv sf$

Anwendung

Codeerzeugung für Vergleiche:

z.B. Intel-Prozessoren (im Prinzip):

```
sub a,b    # Setzen von Flag-Registern
bgt ziel   # Prüfen der Flag-Register
```

Größenvergleiche ≠ Subtraktion und Vorzeichentest:

$a<b \Leftrightarrow (overflow\_sub$ **xor** $sf)$

Beispiel: *Predicated execution* @ ARM-Prozessor

Opcode (31:28)	*Mnemonic Extension*	*Meaning*	*Status flag state*
0001	NE	*Not Equal*	…
0010	CS/HS	*Carry Set/Unsigned Higher or Same*	…
0011	CC/LO	*Carry Clear/Unsigned Lower*	…
0100	MI	*Minus/Negative*	…
0101	PL	*Plus/Positive or Zero*	…
0110	VS	*Overflow*	V set
1000	HI	*Unsigned Higher*	…
1001	LS	*Unsigned Lower or Same*	…
…	…	…	…

Multiplikation

Vorgehen analog zu natürlichen Zahlen erfordert Sonderbehandlung des Vorzeichens. Wird beim Algorithmus von Booth vermieden.

Grundidee: für eine Kette von Einsen nur 2 Additionen erforderlich:

`i`: Position der am weitesten rechts stehenden 1

`j`: die Position der am weitesten links stehenden 1:

```
              j  i
A * int("0001111000")
```

```
         j  i
int("0001111111") = 2^(j+1) − 1.
int("0000000111") = 2^i − 1.
int("0001111000") = (2^(j+1) − 1 − 2^i + 1)
```

Also:

```
              j  i
A * int("0001111000") = A * (2^(j+1) − 2^i)
```

Beispiel:

```
A = int("0011") = 3; B = int("0110") = 6 ;
n = 4; A * 6 = A * int("0110") = A * (2^3 − 2^1)
```

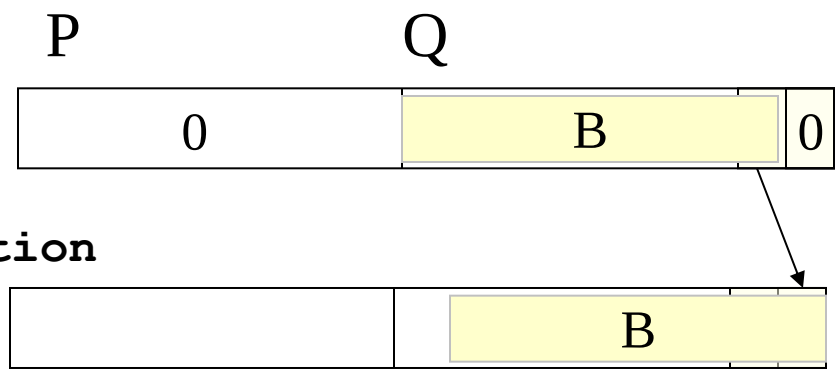
−A für die volle Bitvektorlänge des Ergebnisses: −A =int("11111101")

Schritte des Rechenvorgangs:

B	Aktion		Ergebnis
			"00000000"
"0110"	keine	+	"00000000"
			"00000000"
"0110"	−2*A	+	"11111010"
			"11111010"
"0110"	keine	+	"00000000"
			"11111010"
"0110"	+8*A	+	"00011000"
			"00010010"

Booth-Algorithmus

```
FUNCTION Booth(A,B: IN bit_vector) RETURN bit_vector IS
       CONSTANT n : natural := A'LENGTH;
       VARIABLE P : bit_vector(n-1 DOWNTO 0):=(OTHERS => '0');
       VARIABLE Q : bit_vector(n DOWNTO 0) :=(OTHERS => '0');
BEGIN
 Q(n DOWNTO 1) := B;
 FOR i IN 0 TO n-1 LOOP
   CASE Q(1 DOWNTO 0) IS
       WHEN "10" => P := P - A;
       WHEN "01" => P := P + A;
       WHEN OTHERS => -- keine Aktion
   END CASE;
   P & Q := sra (P & Q);
  END LOOP;
  RETURN P(n-2 DOWNTO 0) & Q(n DOWNTO 1);
END Booth;
```



Einzelschritte der Prozedur

Opr.	P	Q	Kommentar
	0000	00000	
	0000	01100	Q(1 DOWNTO 0) = "00"
sra	0000	00110	Q(1 DOWNTO 0) = "10"
-A	1101	00110	- 1 Bit vom LSB entfernt =-2*A
sra	1110	10011	Q(1 DOWNTO 0) = "11"
sra	1111	01001	Q(1 DOWNTO 0) = “01"
+A	0010	01001	"1111"+"0011"="0010"; +8*A
sra	0001	00100	

LSB des Ergebnisses

MSB des Ergebnisses

```
FOR i IN 0 TO n-1 LOOP
 CASE Q(1 DOWNTO 0) IS
  WHEN "10" => P := P - A;
  WHEN "01" => P := P + A;
  WHEN OTHERS => --
 END CASE;
 P & Q := sra (P & Q);
```

Ergebnis: *int*(“0010010“) =18.

Wegen ’-A’: P & Q in 2k-Darstellung u. arithmetisches Schieben.

Korrekte Interpretation von B ?

Nachweise für B´*length*=4; Seien b_0 bis b_3 die Elemente von B.

Jeder einzelne Schritt: stellengerechte Multiplikation von A mit (b_{i-1} -b_i);

$$\begin{aligned} \text{Booth}(A, B) = \; & A * (b_{-1} - b_0) * 2^0 + \\ & A * (b_0 - b_1) * 2^1 + \\ & A * (b_1 - b_2) * 2^2 + \\ & A * (b_2 - b_3) * 2^3 \end{aligned}$$

Es gilt: $-b_i \times 2^i + b_i \times 2^{i+1} = b_i \times 2^i$

➔: $Booth(A, B) = A \times (b_0 \times 2^0 + b_1 \times 2^1 + b_2 \times 2^2 - b_3 \times 2^3) = A \times int(B)$

Trick der korrekten Behandlung von B als Integer-Zahl:
”vergessen” , bei einer ’1’ im Vorzeichen von B die eigentlich übliche Behandlung am linken Rand einer Folge von Einsen vorzunehmen.

Booth-Algorithmus
Sonderfall: A = kleinste darstellbare Zahl

Falls **A** die kleinste darstellbare Zahl ist,

dann ist **−A** nicht in n Bit darstellbar.

Beispiel:

(-8)*(-8) = 64 = int("0100 0000")

Dieser Wert kann vom obigen Booth-Algorithmus nicht geliefert werden, da aufgrund des abschließenden Schiebeschritts **P(n−1)=P(n−2)** ist.

Ausweg: Verlängern von **P** um ein Bit.

Verbesserungen des Booth-Algorithmus:

Einzelne Nullen bzw. Einsen sollen wie beim Standard-verfahren nur eine Operation erzeugen.

Bei isolierter 1 im Bit i wird $2^{i+1} - 2^i = 2^i$ addiert.

Bei isolierter 0 im Bit i wird $2^i - 2^{i+1} = -2^i$ addiert.

Übergang auf die Betrachtung eines Fensters von 3 Bit und Verschiebung um jeweils $\pm$2 Bit; abhängig vom Muster im Fenster Addition von $\pm 2 * A$, A (siehe Hayes)

Ignorieren von Folgen gleicher Ziffern

Multiplikation

Ergebnis: Produkt ganzer Zahlen, soweit bei fester Datenwortlänge möglich.

Doppelt-langes Ergebnis?

Integer in VHDL

Interpretation von Bitvektoren als **integer**
(falls der Zahlenbereich ausreicht):

```
function int (a: bit_vector) return
  integer is -- Annahme: a'left > 0
constant t : natural := (2 ** (a'left));
  begin
    if a(a'left) = '0' - - positive Zahl
    then return nat (a)
    else return(nat(a(a'left-1 downto 0))-t)
    end if;
  end int;
```

2^n im Datentyp **integer** evtl. nicht darstellbar.

Es kann helfen, $-(2^{n-1}) + nat(a(a'left\text{-}1 \text{ downto } 0)) - (2^{n-1})$ zu berechnen.

Zusammenfassung

Realisierung elementarer (Maschinen-) Datentypen

- Operationen auf Bitvektoren
  - *Barrel shifter*
- Natürliche Zahlen
  - Konvertierung Bitvektor $\rightarrow$ natürliche Zahl
  - Basisbausteine: ALUs
  - Erkennung von Überläufen
  - Parallele Multiplikation
- Ganze Zahlen
  - Konvertierung Bitvektor $\rightarrow$ ganze Zahl
  - Erkennung von Überläufen