technische universität
dortmund

fakultät für informatik
informatik 12

3. Mikroarchitekturen

Peter Marwedel
Informatik 12
TU Dortmund

2012/04/20

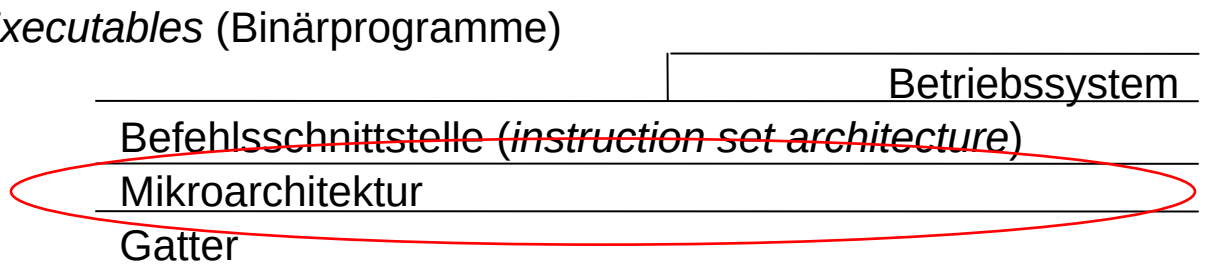
Diese Folien enthalten Graphiken mit Nutzungseinschränkungen. Das Kopieren der Graphiken ist im Allgemeinen nicht erlaubt.

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)*.



Gegenstand des Kurses RA
- Definitionen von „Rechnerarchitektur” -

Def. (nach Stone): *The study of computer architecture is the study of the* ***organization and interconnection of components*** *of computer systems. Computer architects construct computers from* ***basic building blocks*** *such as memories, arithmetic units and buses.*

From these building blocks the computer architect can construct ***anyone of a number of different types of computers****, ranging from the smallest hand-held pocket calculator to the largest ultra-fast super computer. The functional behaviour of the components of one computer are similar to that of any other computer, whether it be ultra-small or ultra-fast.*

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.

Datentyp `boolean`

`boolean` ist vordefiniert:

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

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

Datentypen `bit` und `bit_vector`

`bit` ist vordefiniert durch

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

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

`bit_vector` ist vordefiniert durch

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

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

Beispiel:

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

`downto`: absteigender Indexbereich.

In dieser Vorlesung: untere Indexgrenze meist = 0

Der Datentyp bit_vector (2)

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.

Weiterhin: Konkatenation:

"01010101" & '0'
"01010101" & "01010101"
'1' & '0'

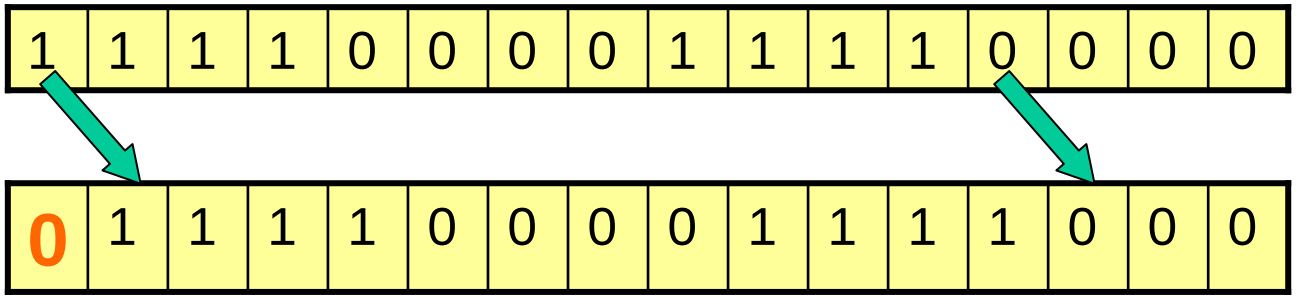
3.2 Realisierung elementarer Datentypen
3.2.1 Operationen auf Bitvektoren

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

3.2.1.1 Schiebeoperationen

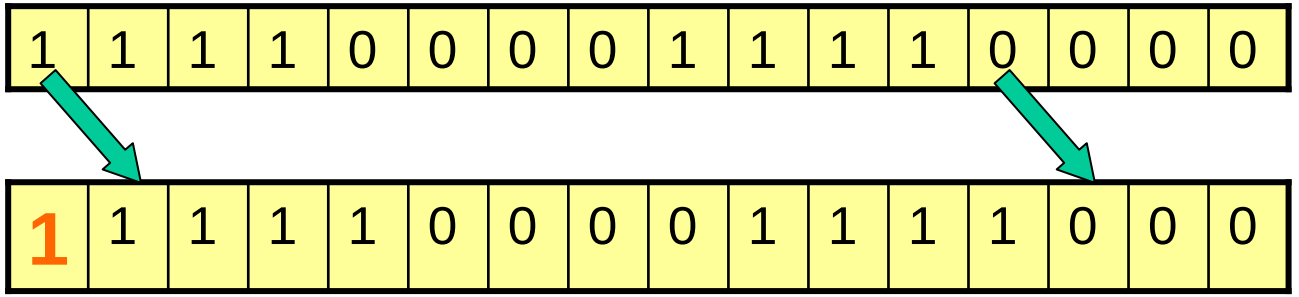
Hier: *shift right logical, shift left logical, shift right arithmetical, shift left arithmetical*:

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



Schiebeoperationen

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



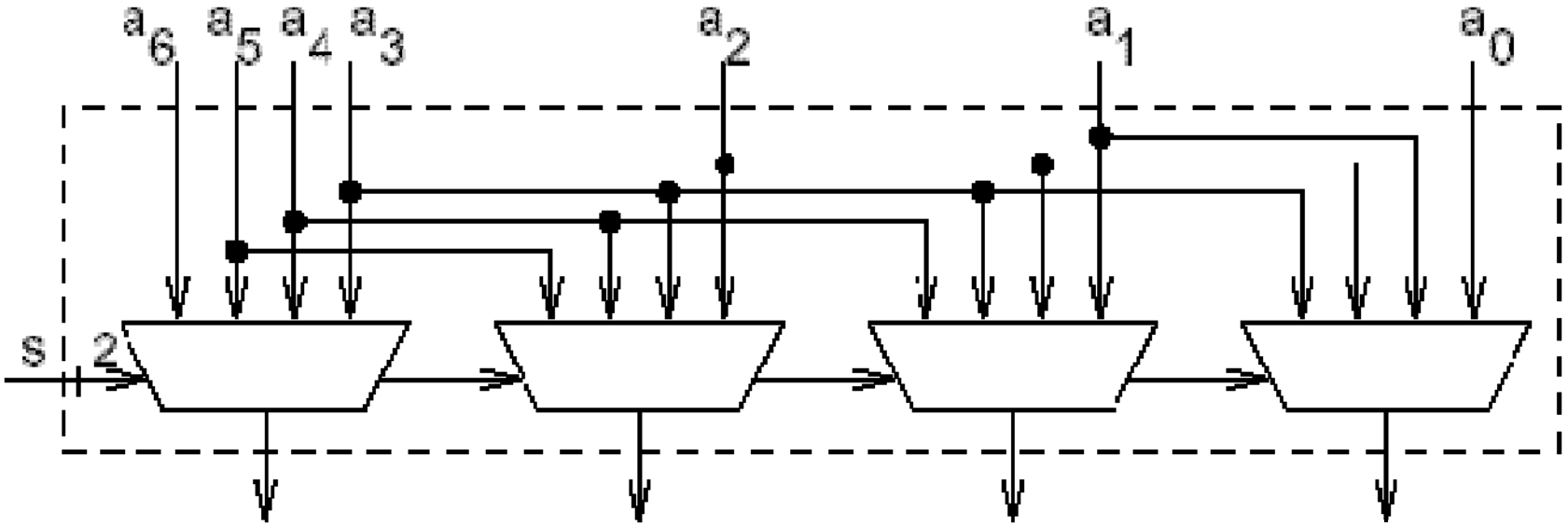

```
sll(a) = a (a'left-1 downto 0) & '0'
```

```
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.

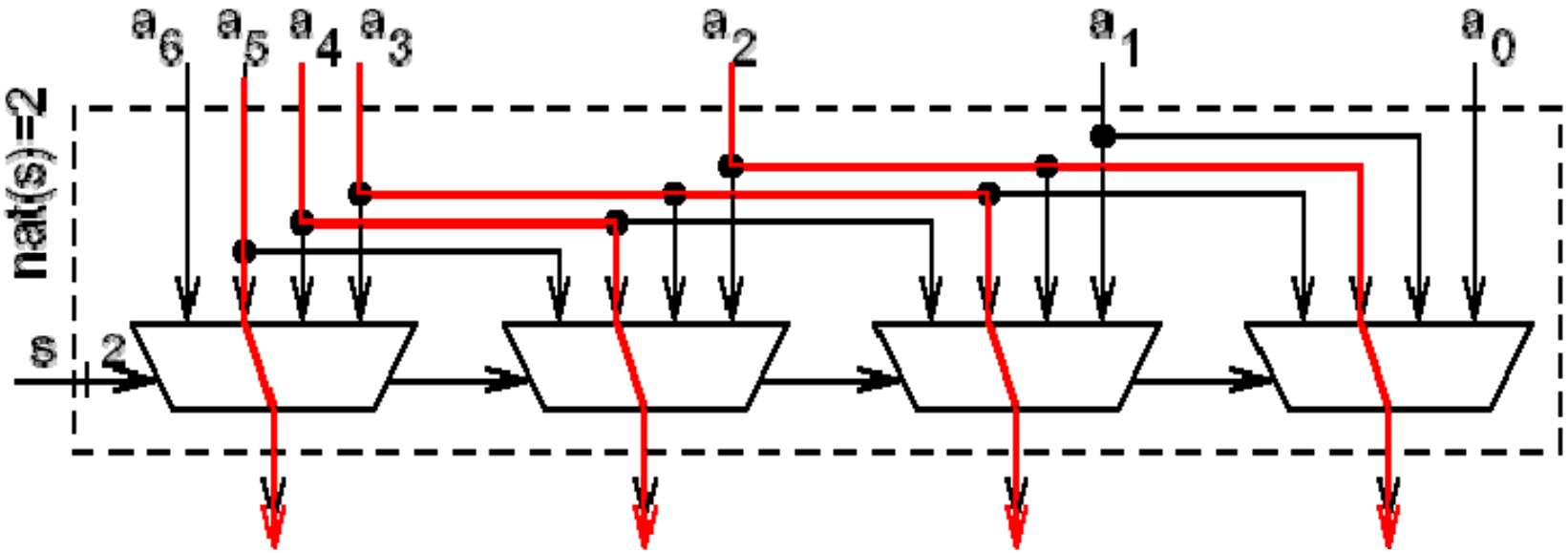
Realisierung der Operationen auf Bitvektoren: Schieben

- 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:

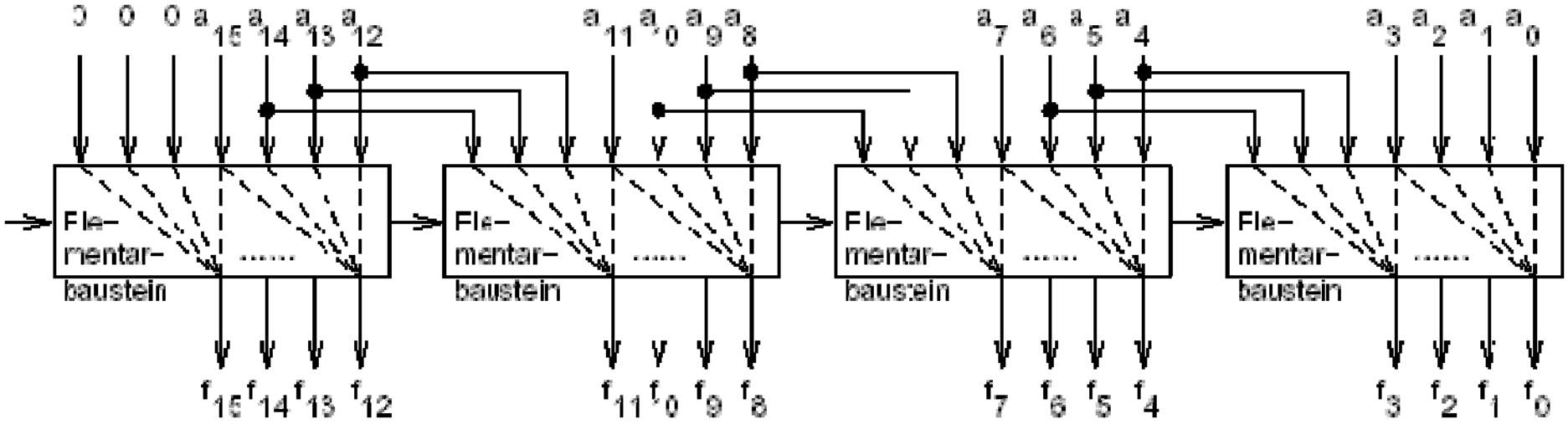


Realisierung der Operationen auf Bitvektoren: Schieben

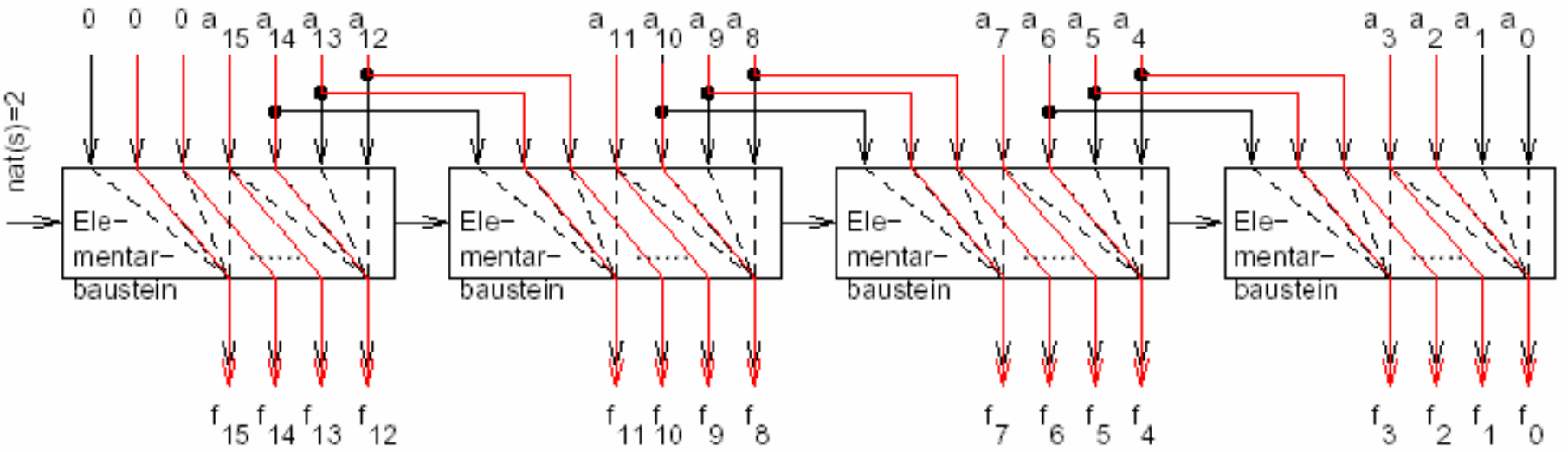
Elementarbaustein, 4 Ausgänge, Schieben: 0-3 Stellen:



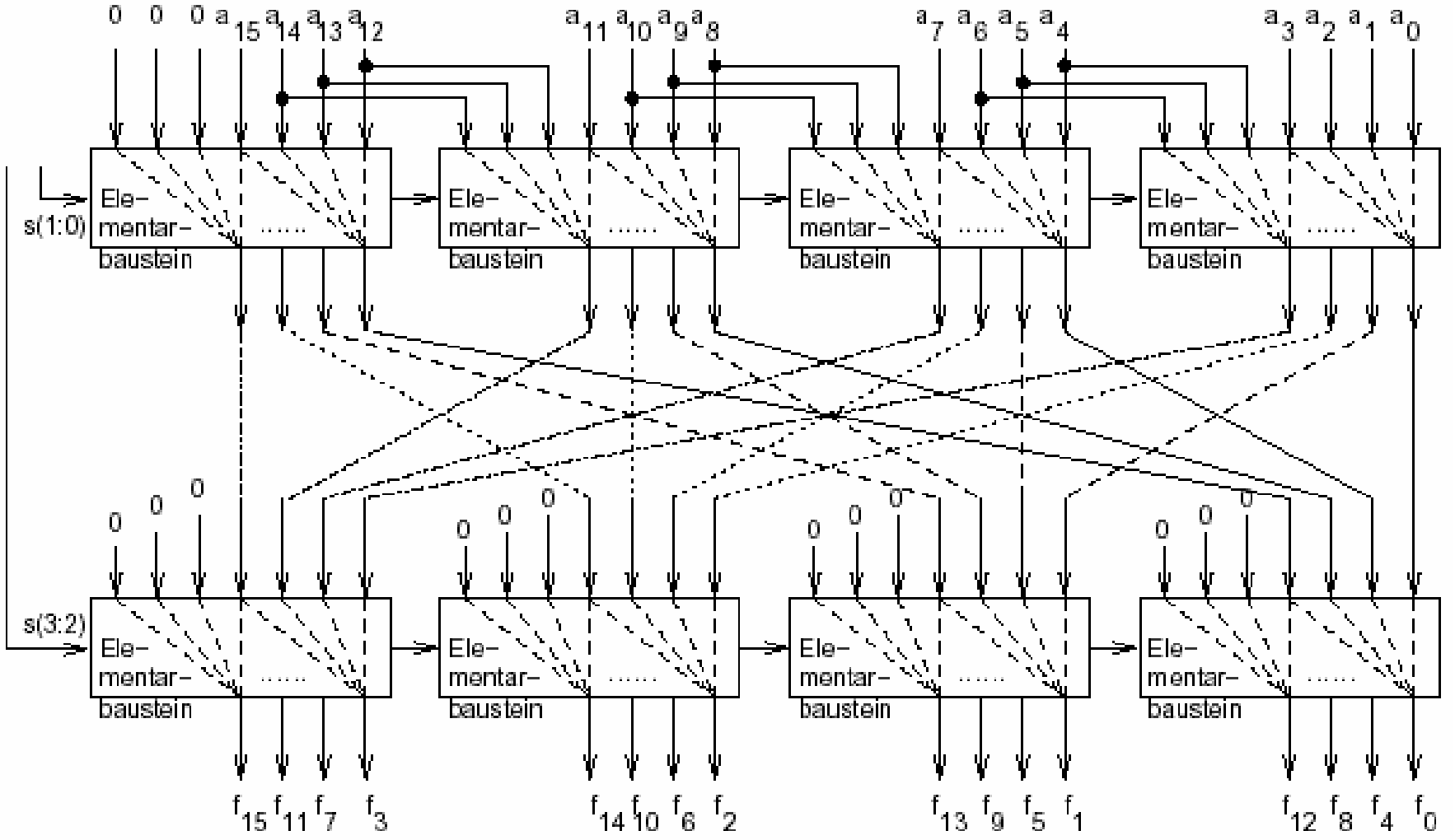
Barrelshifter mit 16 Ausgängen



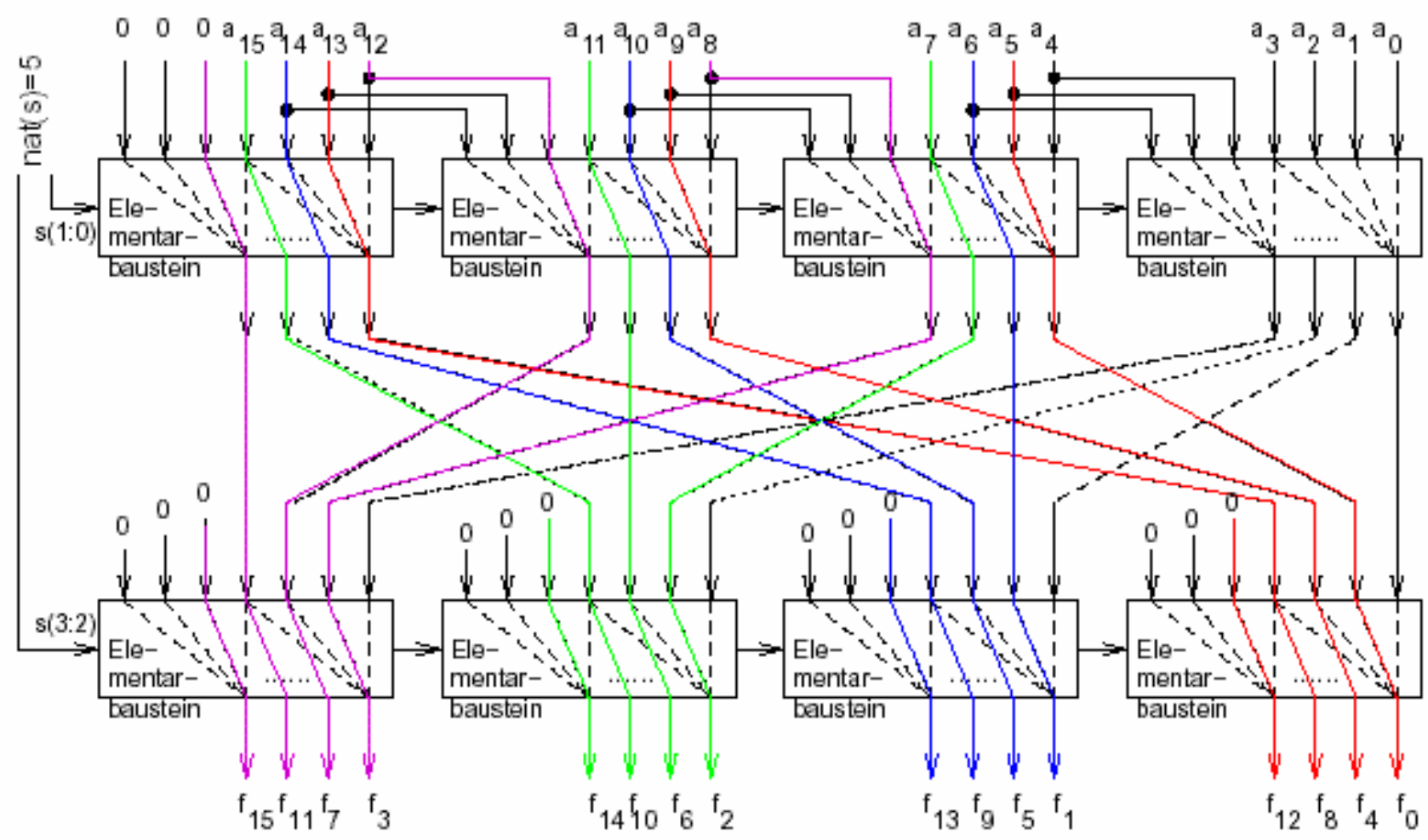
Barrelshifter mit 16 Ausgängen



Barrelshifter für das Schieben um 0 bis 15 Stellen:



Barrelshifter für das Schieben um 0 bis 15 Stellen:



Weitere Information zu *Barrel Shiftern*

- Anwendung: u.a. Normalisierung von Gleitkommazahlen
- Simulation:
  http://tams-www.informatik.uni-hamburg.de/applets/hades/webdemos/10-gates/60-barrel/shifter8.jnlp

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

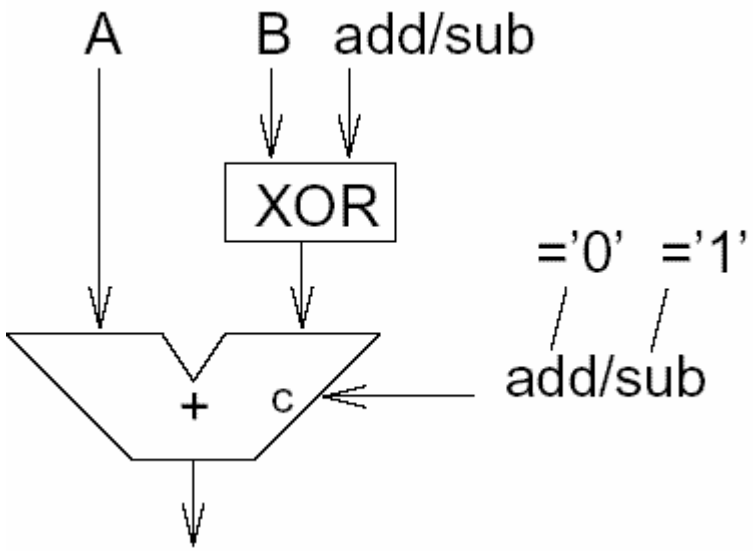
Ergebnis: Summe natürlicher Zahlen, soweit dies bei fester Datenwortlänge möglich ist.

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 $A-B = A+ \neg B+1$

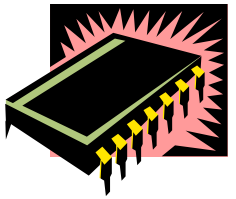


Die Subtraktion kann also einfach auf die Addition zurückgeführt werden.

(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)

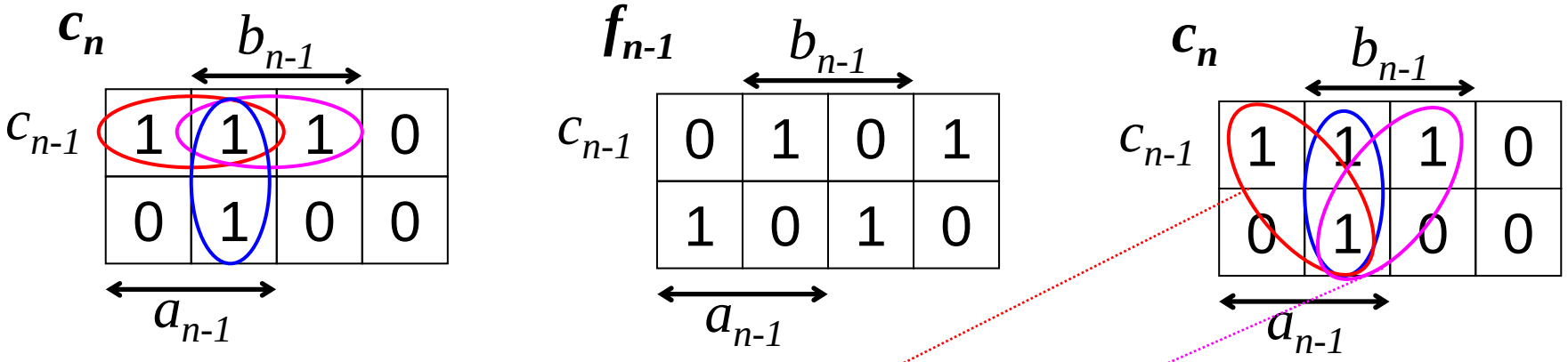
$$= (a_{n-1} b_{n-1}) \vee ((a_{n-1} \vee b_{n-1})\, c_{n-1})$$

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

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

Überläufe

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



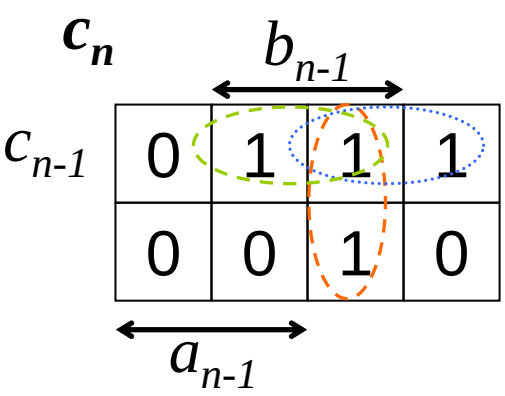
$$c_n = (a_{n-1} b_{n-1}) \vee (a_{n-1}\, \overline{f_{n-1}}) \vee (b_{n-1}\, \overline{f_{n-1}})$$

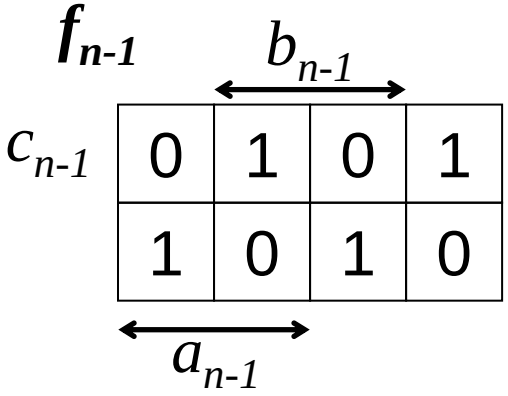
$\overline{f_{n-1}}$ extern bekannt:

(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 $cf\_ = c_n$:

$$a<b \Leftrightarrow (a-b)<0 \Leftrightarrow cf\_ = \text{'1'}$$
$$a \geq b \Leftrightarrow \neg((a-b)<0) \Leftrightarrow cf\_ = \text{'0'}$$
$$a>b \Leftrightarrow ((a-b) \geq 0) \wedge (a-b) \neq 0 \Leftrightarrow (cf\_ = \text{'0'}) \wedge (zf = \text{'0'})$$
$$a \leq b \Leftrightarrow \neg(a>b) \Leftrightarrow (cf\_ = \text{'1'}) \vee (zf = \text{'1'})$$

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

Zusammenfassung

Realisierung elementarer (Maschinen-) Datentypen

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