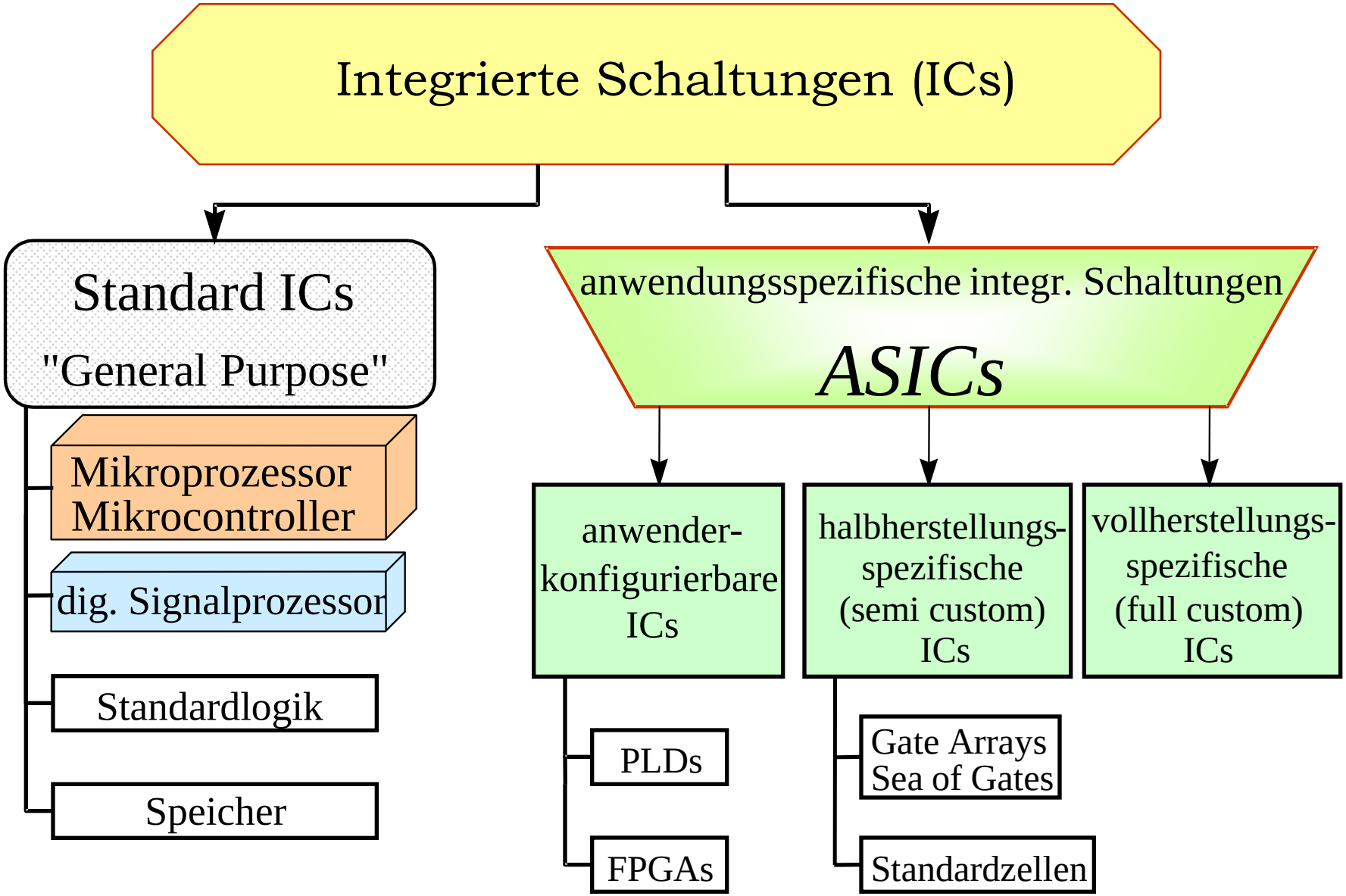
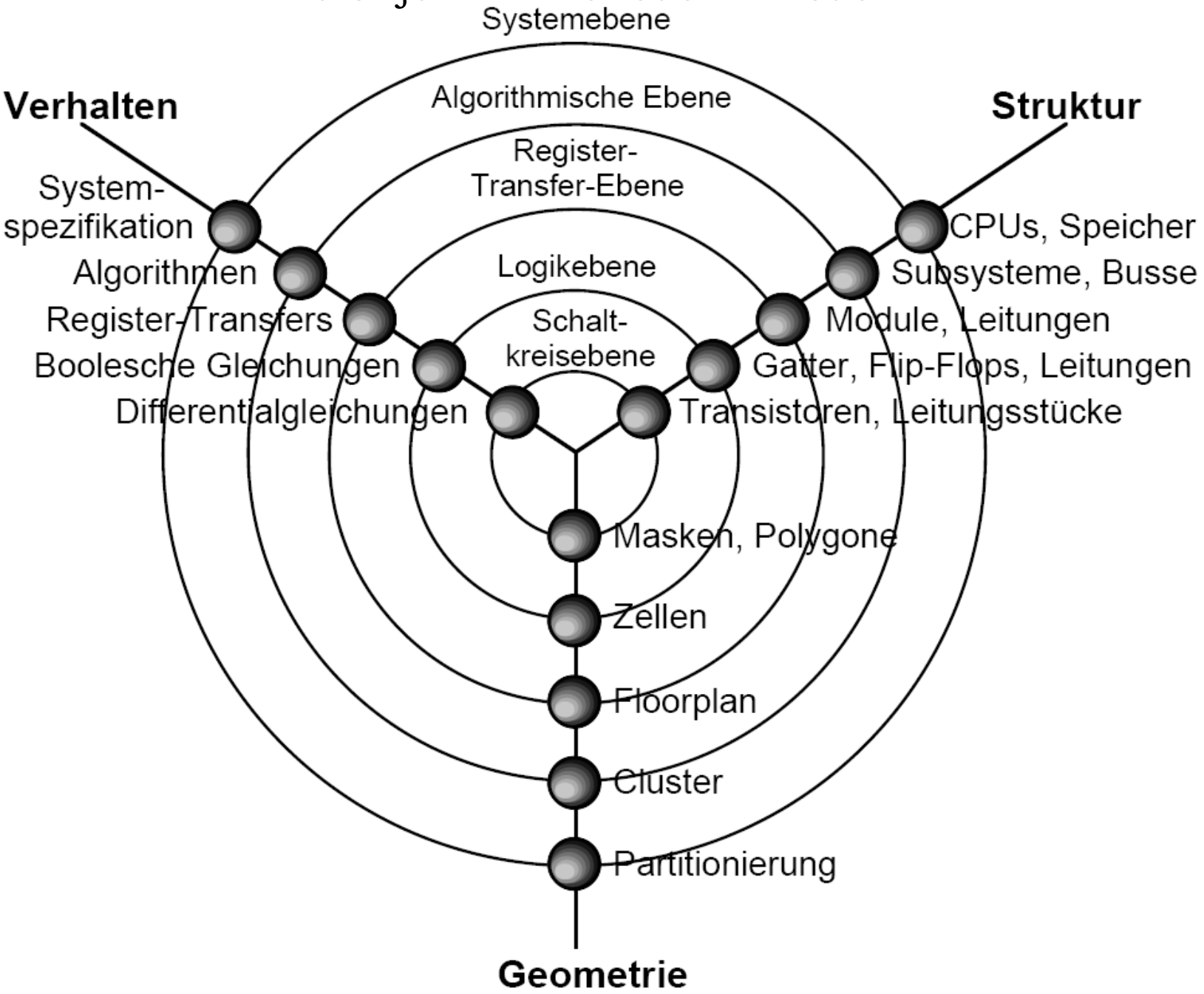

Integrierte Schaltungen (ICs)
Standard ICs
"General Purpose"
Mikroprozessor
Mikrocontroller
dig. Signalprozessor
Standardlogik
Speicher
anwendungsspezifische integr. Schaltungen
ASICs
anwender-
konfigurierbare
ICs
halbherstellungs-
spezifische
(semi custom)
ICs
vollherstellungs-
spezifische
(full custom)
ICs
PLDs
FPGAs
Gate Arrays
Sea of Gates
Standardzellen

Das Gajski-Walker oder Y-Modell



Historische Entwicklung von VHDL

Die Anfänge von VHDL reichen bis in die frühen achtziger Jahre zurück.

Im **VHSIC**-Programm des (Department of Defense, **DoD**) wurde in den USA nach einer Sprache zur Dokumentation elektronischer Systeme gesucht. Es sollten die enormen Kosten für **Wartung und Nachbesserung** bei militärischen Systemen reduziert werden.

Ziel:
- Klare Beschreibung komplexer Schaltungen sowie die Austauschbarkeit von Modellen zwischen verschiedenen Entwicklungsgruppen

Start:
- Juli 1983: Intermetrics, IBM und Texas Instruments bekommen den Auftrag, die neue Sprache zu entwickeln. Sie soll sich an die im militärischen Bereich gebräuchliche Programmiersprache ADA anlehnen.
- August 1985: Eine erste Version - VHDL V7.2
- Februar 1986: Übergabe zur Standardisierung an IEEE
- Dezember 1987: IEEE l076-1987 wird zum ersten und bislang einzigen Standard für Hardwarebeschreibungssprachen. Der Standard definiert allerdings nur die Syntax und Semantik der Sprache selbst, nicht jedoch ihre Anwendung bzw. ein einheitliches Vorgehen bei der Anwendung.
- ab September 1988: Alle Elektronik-Zulieferer des DoD müssen VHDL-Beschreibungen ihrer Komponenten und Subkomponenten bereitstellen. Ebenso sind VHDL-Modelle von Testumgebungen mitzuliefern.

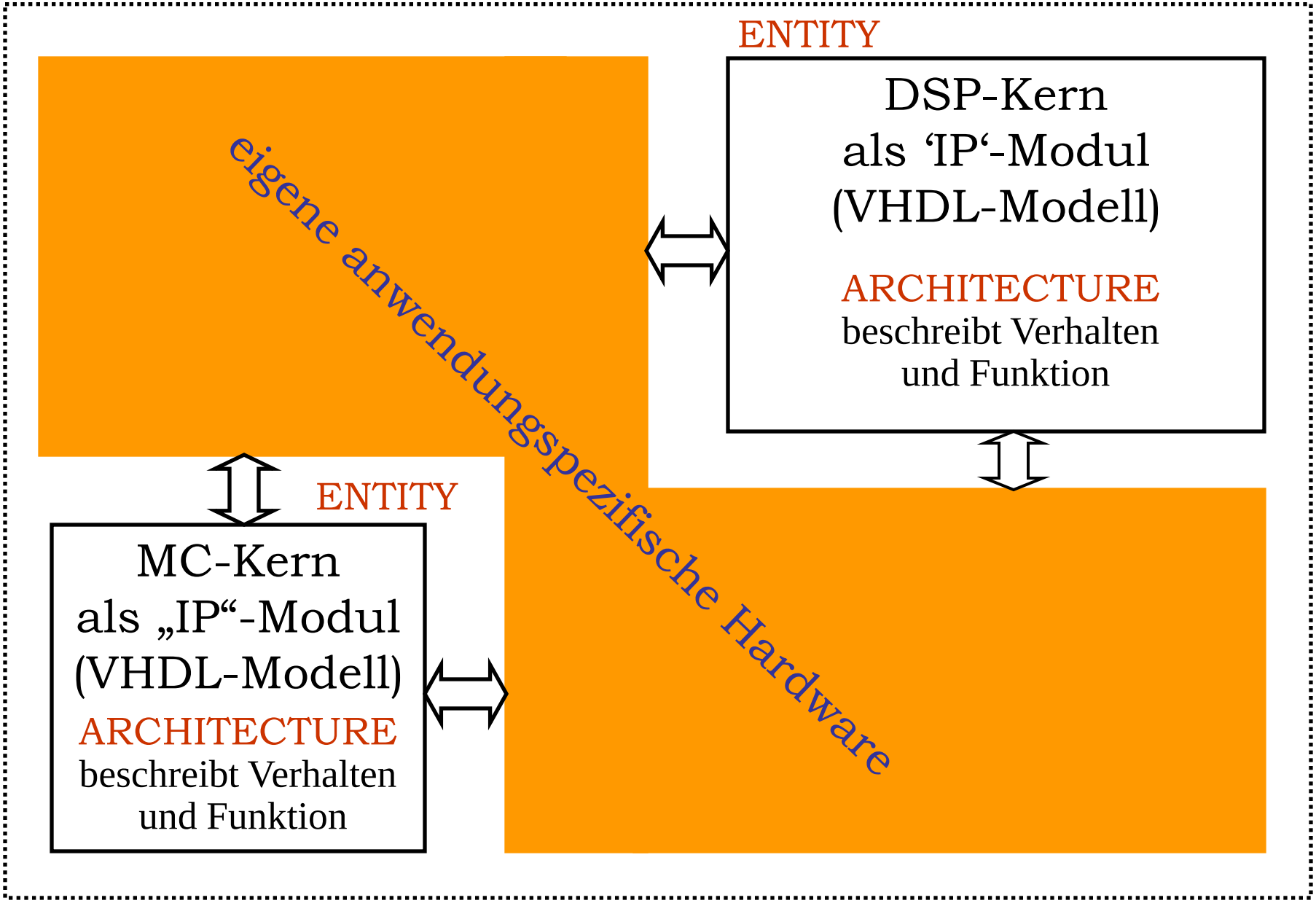
Nach IEEE-Richtlinien muss ein Standard alle fünf Jahre überarbeitet werden. Daher wurde im Zeitraum 1992-1993 die Version IEEE 1076-1993 erarbeitet. Seit Beginn der neunziger Jahre hat sich VHDL weltweit etabliert.

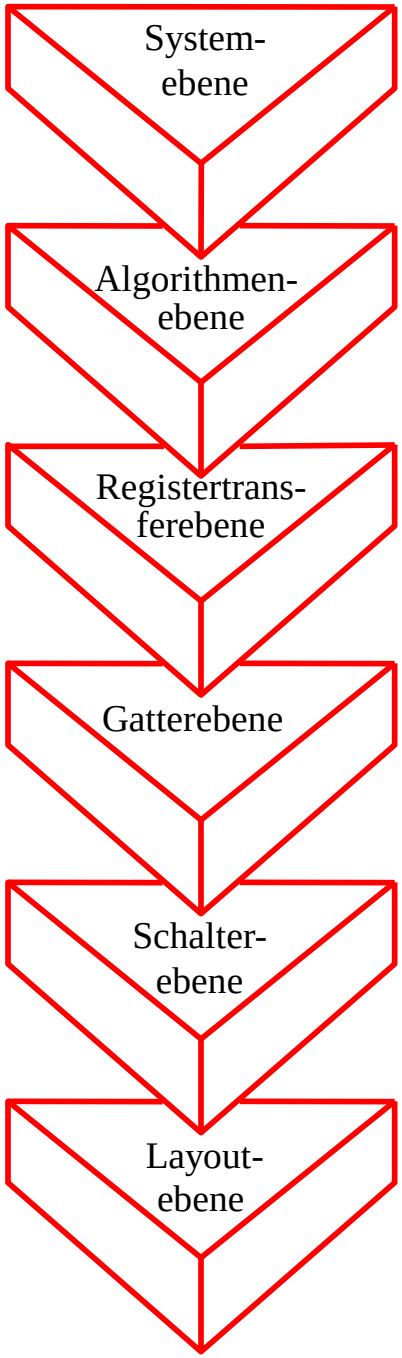
Der **1076-l993**-Standard entstand bereits weitgehend losgelöst vom **DoD** unter internationaler Beteiligung. Europa wirkte unter anderem über **ESPRIT** daran mit. Auch Asien - vor allem Japan - hat VHDL akzeptiert.

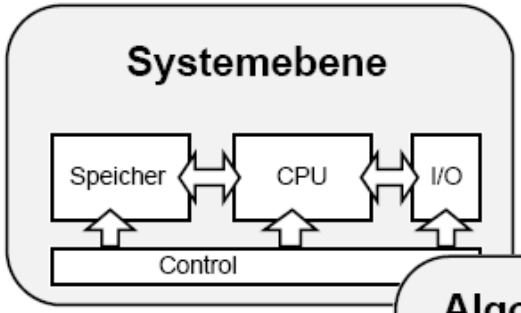
Konkurrenz besteht heute praktisch nur noch durch „Verilog HDL“, weil der führende CAE-Hersteller Cadence diese HDL in seine Werkzeuge eingebaut hat. Während Verilog lange Zeit als „Cadence-proprietär“ war, ist auch diese HDL mittlerweile ein offener Standard, der in den meisten CAE-Werkzeugen für den Entwurf integrierter Schaltungen gleichberechtigt neben VHDL zur Verfügung steht.

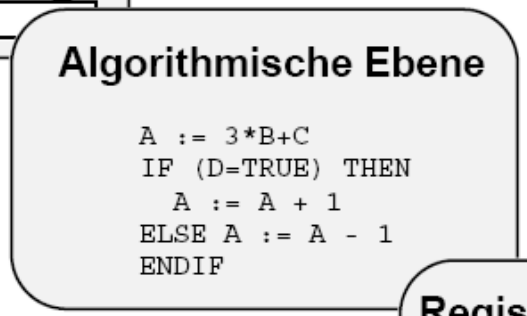
VHDL **V**HSIC **H**ardware **D**escription Language; **VHSIC** **V**ery **H**igh **S**peed **I**ntegrated **C**ircuit
IEEE **I**nstitute of **E**lectrical and **E**lectronics **E**ngineers; **CAE** **C**omputer **A**ided **E**ngineering

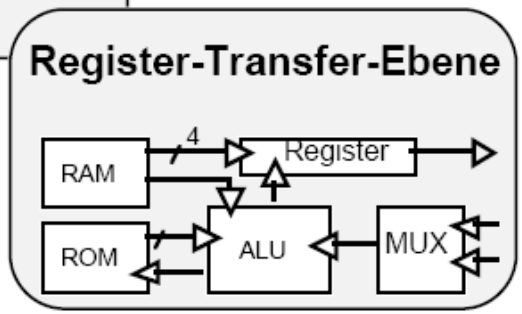
Konzept zum Aufbau komplexer Systeme auf Silizium

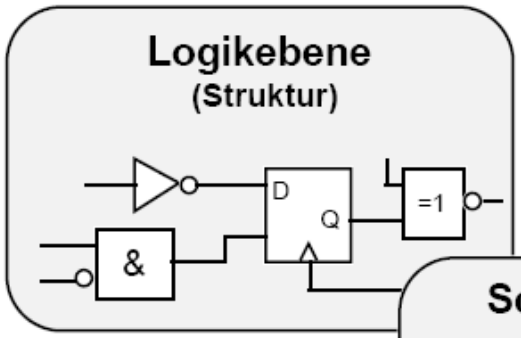


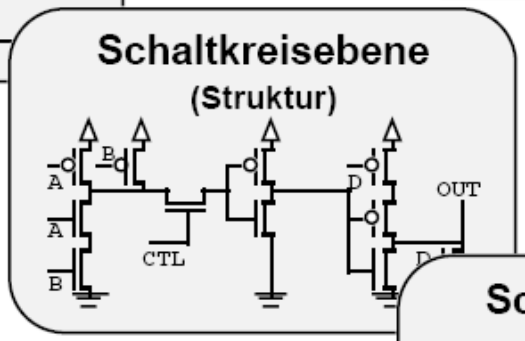

System-
ebene
Algorithmen-
ebene
Registertrans-
ferebene
Gatterebene
Schalter-
ebene
Layout-
ebene

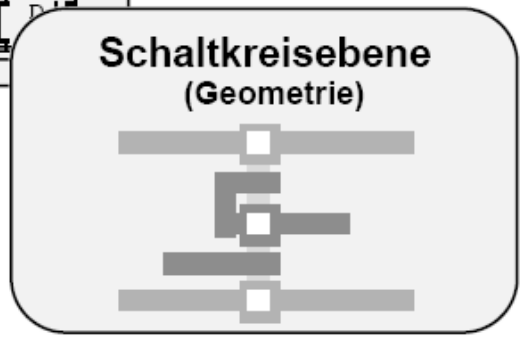

Systemebene
Speicher
CPU
I/O
Control

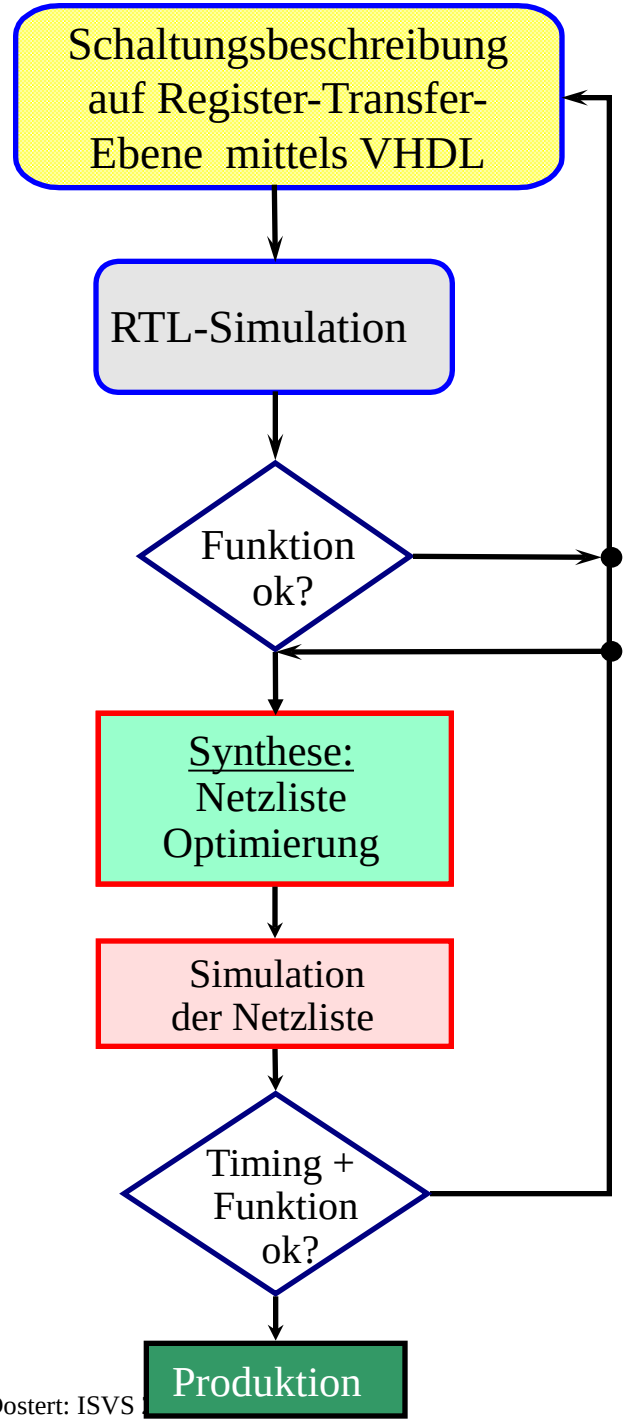

Algorithmische Ebene
A := 3*B+C
IF (D=TRUE) THEN
A := A + 1
ELSE A := A - 1
ENDIF

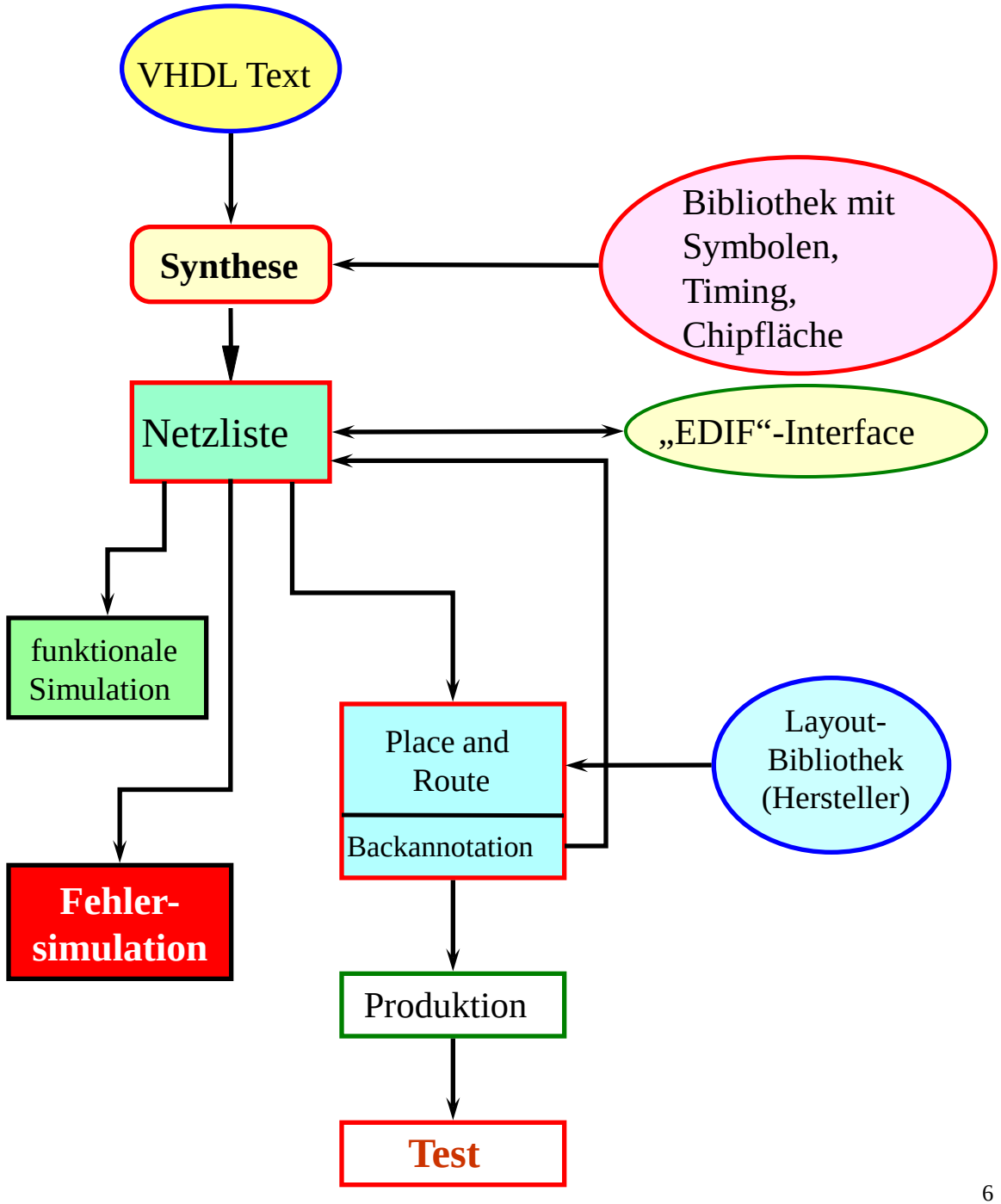

Register-Transfer-Ebene
RAM
4
Register
ROM
ALU
MUX


Logikebene
(Struktur)
D
Q
=1
&


Schaltkreisebene
(Struktur)
A
B
A
B
CTL
D
OUT


Schaltkreisebene
(Geometrie)


Schaltungsbeschreibung auf Register-Transfer-Ebene mittels VHDL
RTL-Simulation
Funktion ok?
Synthese: Netzliste Optimierung
Simulation der Netzliste
Timing + Funktion ok?
Produktion


VHDL Text
Synthese
Bibliothek mit Symbolen, Timing, Chipfläche
Netzliste
„EDIF“-Interface
funktionale Simulation
Fehler-simulation
Place and Route
Backannotation
Layout-Bibliothek (Hersteller)
Produktion
Test

Download kostenloser Entwicklungsumgebungen für CPLDs und FPGAs ALTERA Quartus und LOGIC2

http://dl.altera.com/?edition=lite&platform=windows&download_manager=direct

Der Link führt zum „Download“–Bereich, von wo die

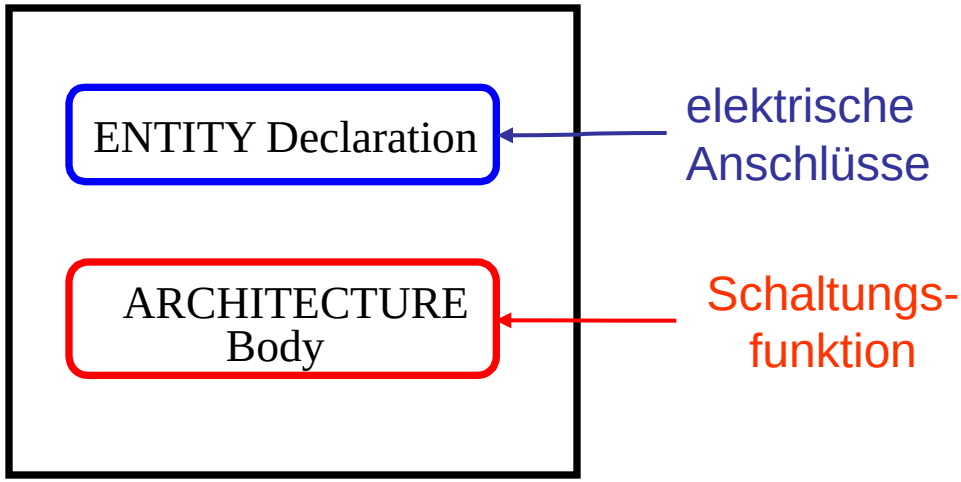
Quartus Prime Light Edition, Release 15.1

im Umfang von ca. 5,2 GB heruntergeladen werden kann. Es wird keine Lizenz benötigt.

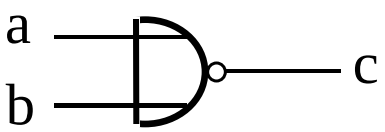
Das Design-Tool LOGIC2 steht kostenlos und ohne besondere Lizenzierung unter http://www.logic2.de/
zum Download zur Verfügung (Umfang ca. 16MB). Eine ausführliche Installations- und Bedienungsanleitung ist enthalten.

Die Grundelemente eines VHDL–Modells: Entity und Architecture

Skript: S. 202-252



Beispiel: NOR-Gatter



a	b	c
0	0	1
1	0	0
0	1	0
1	1	0

```
ENTITY NOR_Gate IS
PORT (a, b:    IN  bit;  -- Signale an den
      c:       OUT bit); -- Ein- und Ausgängen
END NOR_Gate;
ARCHITECTURE Verknuepfung OF NOR_Gate IS
BEGIN
c <='1' WHEN a = '0' AND b = '0' ELSE
    '0' WHEN a = '0' AND b = '1' ELSE
    '0' WHEN a = '1' AND b = '0' ELSE
    '0' WHEN a = '1' AND b = '1' ELSE
    '0'
END Verknuepfung
```

Alternativ

```
BEGIN
c <= a NOR b
END Verknuepfung
```

Auszug aus dem VHDL-Sprachumfang

- Portdeklaration (**PORT**)

- Objekte:
  - Konstanten (**CONSTANT**)
  - Variablen (**VARIABLE**)
  - Signale (**SIGNAL**)
- Komponentendeklaration (**COMPONENT**)
- Typdeklaration (**TYPE**)
- Prozesse (**PROCESS**)
- Anweisungen:
  - **IF**
  - **CASE**
  - bedingte Zuweisung mit **SELECT**
  - Schleifen:
    - **FOR**
    - **WHILE**
  - **WAIT**
  - **GENERATE**
  - **BLOCK**

- Bibliotheken (**LIBRARY**)
- Packages (**PACKAGE**)

Portdeklaration

Syntax:

```
PORT (port_name: Modus Typ;
      port_name: Modus Typ
     );
```

Beispiel:

```
PORT (a: IN      BOOLEAN;
      b: IN      INTEGER RANGE 0 TO 31;
      c: INOUT   BIT;
      e: OUT     STD_LOGIC;
      f: OUT     STD_LOGIC_VECTOR (3 DOWNTO 0)
     );
```

Deklaration von Konstanten und Variablen

Syntax:

```
CONSTANT Konstantenname: Typ := Wert;
```

Beispiele:

```
CONSTANT a: BOOLEAN   := TRUE;
CONSTANT b: INTEGER   := 31;
CONSTANT c: BIT_VECTOR (3 DOWNTO 0) := "0000";
CONSTANT d: STD_LOGIC := 'Z';
CONSTANT e: STD_LOGIC_VECTOR (3 DOWNTO 0) := "0-0-";
```

Syntax:

```
VARIABLE var_name: Typ;
```

Beispiele:

```
VARIABLE a: BOOLEAN;
VARIABLE b: INTEGER RANGE 0 TO 31;
VARIABLE c: BIT;
VARIABLE d: BIT_VECTOR (3 DOWNTO 0);
VARIABLE e: STD_LOGIC;
VARIABLE f: STD_LOGIC_VECTOR (0 TO 3);
```

Deklaration und Besonderheiten von Signalen

Syntax:

```
SIGNAL sig_name: Typ;
```

Beispiele:

```
SIGNAL a: BOOLEAN <= true;          --Defaultwert: true
SIGNAL b: BOOLEAN;                  --Defaultwert: false
SIGNAL c: INTEGER RANGE 0 TO 31;    --Defaultwert: 0
SIGNAL d: BIT;                      --Defaultwert: '0'
SIGNAL e: BIT_VECTOR (3 DOWNTO 0); --Defaultwert:'0','0','0','0'
SIGNAL f: STD_LOGIC;                --Defaultwert: '0'
```

Signale dienen dazu, spezielle Eigenschaften elektronischer Schaltungen zu modellieren. Änderungen von Signalwerten können zeitlich verzögert zugewiesen werden, um Laufzeiten von Hardwarekomponenten nachzubilden.
Verschiedene Module können auch auf ein und dasselbe Signal schreiben.
Diese Eigenschaft gestattet die Modellierung von Bussystemen.

Signalzuweisung: `sig_name <= Wert;`

Variablenzuweisung: `var_name := Wert;`

Durch die Symbole `:=` oder `=:` bzw. `=>` oder `<=` werden Wertzuweisungen an Variablen und an Signale unterschieden.

Komponentendeklaration und -instantiierung

Die Deklaration macht eine Komponente mit ihren Ein- und Ausgangsports bekannt. Sie kann z.B. im Deklarationsteil einer **Architecture**, eines **Blocks** oder einer **Package** stehen.

Syntax:

```
COMPONENT Komponentenname
  Portdeklarationen;
END COMPONENT;
```

Komponenteninstantiierung

Bei der Instantiierung werden die bereits bekannt gemachten Ports der Komponente verdrahtet. Dies geschieht durch Zuweisen von Signalnamen an die Ports.

Syntax:

```
Name: Komponentenname
   PORT MAP (
       port_name => sig_name,
       port_name => sig_name
             );
```

Typdefinition

Bevor in einem VHDL–Modell mit einem Objekt gearbeitet werden kann, muss festgelegt werden, welche Werte das Objekt annehmen darf. Zu diesem Zweck werden **Datentypen** deklariert, mit deren Hilfe die Wertebereiche definiert werden. VHDL verfügt nur über sehr wenige, vordefinierte Datentypen, wie z.B. **real** oder **integer**. Es gibt jedoch umfangreiche Möglichkeiten, benutzerdefinierte Datentypen zu erzeugen. Man kann von vordeklarierten Typen sogenannte Untertypen ableiten. Untertypen sind im einfachsten Fall im Wertebereich eingeschränkte Grundtypen. Eine Ableitung weiterer Untertypen aus einem Untertyp ist nicht möglich.

Beispiel:

```
TYPE Aufzaehlungstyp IS (a, b, c, d, e);
SUBTYPE S_Aufzaehlungstyp IS Aufzaehlungstyp RANGE b TO d;
```

Prozesse

Prozesse stehen in der **Architecture** und laufen in Abhängigkeit von Signalereignissen wie z.B. Taktvorderflanken ab.
Ein Prozess wird nur dann aufgerufen, wenn sich **eines** der in seiner „**Sensitivity–Liste**“ aufgeführten Signale ändert.
Prozesse sind generell nebenläufig.
Komponenten sind in Prozessen nicht erlaubt. Lokale Variablen dürfen aber dort deklariert und unmittelbar verwendet werden.

Syntax:

```
Name:                                          -- optional
  PROCESS (Sensitivity-List)
    Deklarationen
  BEGIN
    Anweisungen
  END PROCESS;
```

Beispiel:

```
PROCESS (clock)
BEGIN
  IF (clock'event AND clock='1') THEN -- Taktvorderflanke
    ...
  END IF;
END PROCESS;
```

Einfache Anweisungen mit IF und CASE

Syntax:

```
IF Bedingung THEN Anweisung;
   ELSIF Bedingung THEN Anweisung
   ELSE Anweisung
END IF;
```

Syntax:

```
CASE Ausdruck IS
  WHEN wert   => Anweisung;
  WHEN wert   => Anweisung;
  WHEN OTHERS => Anweisung;
END CASE;
```

Beispiel:

```
CASE sel IS
  WHEN 0 | 1 | 2 => z <= b;
  WHEN 3 to 10   => z <= c;
  WHEN OTHERS    => z <= d;
END CASE;
```

Während die Case–Anweisung in Abhängigkeit von einem Ausdruck entscheidet, können mit der **bedingten Signalzuweisung** einem Signal in übersichtlicher Weise mehrere Werte in Abhängigkeit von mehreren Bedingungen zugewiesen werden.

Fall 1

Syntax:

```
Name:                                   -- optional
       sig_name <= Wert WHEN Bedingung ELSE
       Wert WHEN Bedingung ELSE
       Wert;
```

Beispiel:

```
z <= a WHEN (x>10) ELSE
b WHEN (x>5)  ELSE
c;
```

Fall 2

Syntax:

```
Name:                                   -- optional
       WITH Ausdruck SELECT
       sig_name <= Wert WHEN Bedingung,
              Wert WHEN Bedingung,
              Wert WHEN OTHERS;
```

Beispiel:

```
WITH sel SELECT
      z <= a WHEN 0 | 1 | 2,
            b WHEN 3 TO 10,
            c WHEN OTHERS;
```

Schleifen: FOR, WHILE und die WAIT-Anweisung

Syntax:

```
Name:          -- optional
  FOR i IN Untergrenze TO Obergrenze LOOP
    Anweisungen
  END LOOP;
```

Beispiel:

```
FOR i IN 1 TO 10 LOOP
  a(i):=i*i;
END LOOP
```

Syntax:

```
WHILE Bedingung LOOP
        Anweisung
END LOOP;
```

Beispiel:

```
i:=0;
WHILE (i<10) LOOP
  s<=i;
  i:=i+1;
END LOOP;
```

WAIT kann an jeder Stelle eines Programms auftreten. Es ist jedoch zu beachten, dass dieser Befehl **nicht synthetisiert** werden kann. **WAIT** wird vorwiegend beim Erstellen von **“Testbenches“** benutzt, wodurch ein sequentieller Ablauf mit klar definierter Geschwindigkeit erzielt wird.

Beispiele:

```
WAIT ON a, b;          -- (wartet bis sich a oder b ändert)
WAIT UNTIL x>10;       -- (wartet bis x>10 ist)
WAIT FOR 10 ns;        -- (stoppt das Programm für 10 ns)
WAIT;                  -- (Endlosschleife)
```

LOOP-Anweisung

Iterationsschleifen, d.h. mehrfach zu durchlaufende Anweisungsblökke, können mittels der LOOP-Anweisung realisiert werden. Dabei existieren die folgenden drei Alternativen: FOR-Schleife, WHILE-Schleife und Endlosschleife:

```
[loop_label :] FOR range LOOP
   ...
   ...      -- sequentielle Anweisungen
   ...
END LOOP [loop_label] ;

[loop_label :] WHILE condition LOOP
   ...
   ...      -- sequentielle Anweisungen
   ...
END LOOP [loop_label] ;

[loop_label :] LOOP
   ...
   ...      -- sequentielle Anweisungen
   ...
END LOOP [loop_label] ;
```

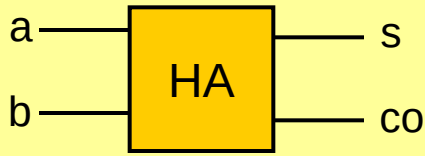
Generate Anweisung

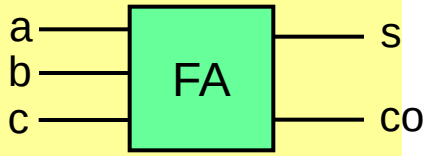
Syntax:

```
Name:          FOR i IN Untergrenze TO Obergrenze GENERATE
                     Anweisungen
               END GENERATE;

               IF Bedingung GENERATE
                     Anweisungen
               END GENERATE;
```

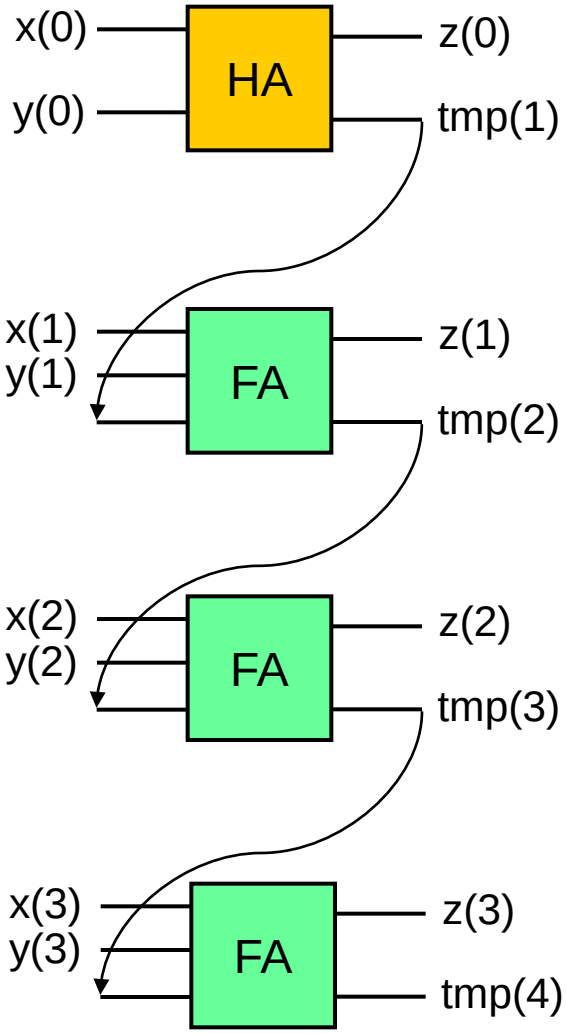
Beispiel: Vierstelliger Ripple-Carry Addierer

```
      g0: FOR i IN 0 TO 3 GENERATE
      g1: IF i=0 GENERATE
            ha: half_adder PORT MAP (
                a => x(i), b => y(i),
                s => z(i), co => tmp(i+1)
                  );
          END GENERATE;
     g2: IF i> = 1 AND i<=3 GENERATE
            fa: full_adder PORT MAP (
                a => x(i), b => y(i), c => tmp(i),
                s => z(i), co => tmp(i+1)
                  );
          END GENERATE;
   END GENERATE;
```





Ausführung des Beispiels



Bibliotheken und die USE-Anweisung

Compilierte VHDL-Modelle werden in einer **Library** für nachfolgende Simulationen oder für die Weiterverwendung in anderen Modellen gespeichert.

Der Vorteil von **Bibliotheken** besteht darin, daß bereits verfügbare Schaltungs-entwürfe wiederverwendet werden können.

Ohne Deklaration steht immer ***.std** zur Verfügung. Hier sind allgemeine Packages gespeichert. Nach korrektem Compilieren eines VHDL-Quellcodes wird das Ergebnis in einer Bibliothek mit dem Default-Namen **work** abgelegt.

Syntax:

```
LIBRARY library_name_1 {,library_name_2...};
```

Syntax:

```
USE library_name.all;
USE library_name.element_name;
USE library_name.package_name.all;
USE library_name.package_name.element_name;
```

Beispiel:

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
```

Auch ohne USE–Anweisung sind alle Elemente von *.std immer sichtbar

Package

Eine **Package** enthält Anweisungen wie z.B. Konstanten- und Typdeklarationen, die in mehreren darunterliegenden VHDL–Modellen gebraucht werden. In einer **Package** wird häufig der zu verwendende Logiktyp mit allen korrespondierenden Operatoren definiert.
Mit der **Package** können mehrfach benötigte Deklarationen durch *einmalige* Eingabe in verschiedene Projekte eingebunden werden. Ein Ändern solcher globaler Informationen führt mit minimalem Aufwand zu einer Neukonfiguration eines Modells. Durch Ändern der Bitbreite von Bussen, Multiplexern, Addieren oder Multiplizierern läßt sich ein Entwurf in erheblichem Umfang verändern und somit schnell und übersichtlich unterschiedlichen Anforderungen anpassen.

Syntax:

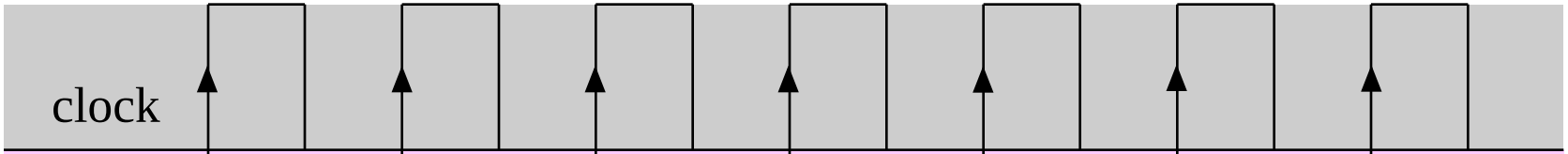
```
PACKAGE definitionen IS
  CONSTANT zeit: TIME := 1 ns;
  CONSTANT a: INTEGER:= 0;
  SUBTYPE int_subtype IS INTEGER RANGE -16 TO 15; --32 Elemente
  TYPE int IS ARRAY (3 DOWNTO 0) OF int_subtype; -- Matrix 4x32
END definitionen;
```

$$\text{vector} \; \{0 \; \cdot \; \cdot \; \cdot \; 31\}$$

$$\text{Matrix} \begin{matrix} 3,0 & \cdot & \cdot & \cdot & 3,31 \\ 2,0 & \cdot & \cdot & \cdot & 2,31 \\ 1,0 & \cdot & \cdot & \cdot & 1,31 \\ 0,0 & \cdot & \cdot & \cdot & 0,31 \end{matrix}$$

Unterschiede beim Gebrauch von Variablen und Signalen

```
PROCESS (clock)
VARIABLE y1, y2, y3: INTEGER;
BEGIN
  IF (clock'event AND clock='1') THEN   -- Taktvorderflanke
    y1:=x;                              -- x wird von außen vorgegeben
    y2:=x+1;
    y3:=y2;
  END IF;
END PROCESS;
```

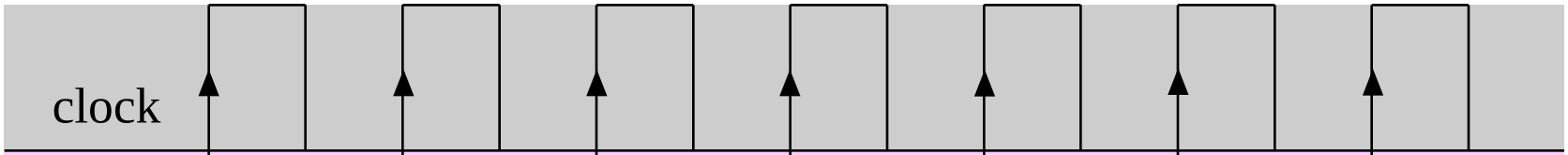


x	1	2	3	4	5	6	7
y1	1	2	3	4	5	6	7
Y2	2	3	4	5	6	7	8
Y3	2	3	4	5	6	7	8

Zeitverlauf bei Verwendung von Variablen

Unterschiede beim Gebrauch von Variablen und Signalen

```
SIGNAL x, y1, y2, y3: INTEGER;
...
PROCESS (clock)
BEGIN
  IF (clock'event AND clock='1') THEN   -- Taktvorderflanke
    y1 <= x;                            -- x wird von außen vorgegeben
    y2 <= x+1;
    y3 <= y2;
  END IF;
END PROCESS
```



x	1	2	3	4	5	6	7
y1	'U'	1	2	3	4	5	6
y2	'U'	2	3	4	5	6	7
y3	'U'	'U'	2	3	4	5	6

Verlauf bei Verwendung von Signalen ('U' steht für nicht initialisiert, d. unbekannt)

Die Datentypen **std_ulogic** und **std_logic**

Die VHDL–Bibliothek **ieee.std_logic_1164** enthält die Datentypdeklarationen:

```
type std_ulogic IS ('U','X','0','l','Z','W','L','H','-');
type std_logic IS ('U','X','0','l','Z','W','L','H','-');
```

Wert	Bedeutung	Kommentar
'U'	nicht initialisiert	nicht initialisiertes Signal im Simulator
'X'	**undefiniert**	**mehr als ein aktiver Signaltreiber ⇒ Buskonflikt liegt vor**
'0'	starke logische '0'	entspricht einem auf Null liegenden Signal der Breite 1bit
'1'	starke logische 'l'	entspricht einem auf Eins liegenden Signal der Breite 1bit
'Z'	hochohmig	Tri-State Ausgang
'W'	**schwach unbekannt**	**Buskonflikt zwischen 'L'- und 'H'-Pegel**
'L'	schwache logische '0'	z.B. Ausgang mit Pull-Down-Widerstand (Open Source)
'H'	schwache logische '1'	z.B. Ausgang mit Pull-Up-Widerstand (Open Drain)
'-'	don't care	ohne Bedeutung => kann für Minimierung verwendet werden

Auflösungsfunktion für den Datentyp **std_logic**

(Aus Gründen der Übersicht wurden die Anführungszeichen in der Tabelle weggelassen)

Signal 1 →

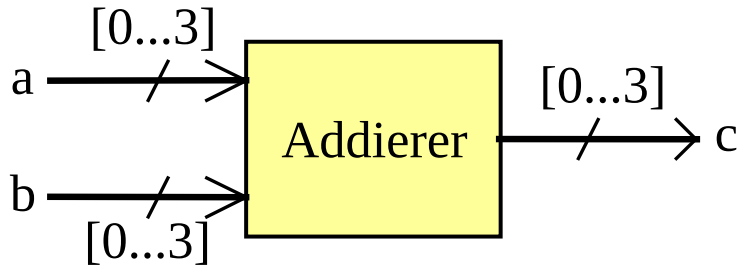
Signal 2 ↓

	U	X	0	1	Z	W	L	H	-
U	**U**	**U**	**U**	**U**	**U**	**U**	**U**	**U**	**U**
X	**U**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**
0	**U**	**X**	**0**	**X**	**0**	**0**	**0**	**0**	**X**
1	**U**	**X**	**X**	**1**	**1**	**1**	**1**	**1**	**X**
Z	**U**	**X**	**0**	**1**	**Z**	**W**	**L**	**H**	**X**
W	**U**	**X**	**0**	**1**	**W**	**W**	**W**	**W**	**X**
L	**U**	**X**	**0**	**1**	**L**	**W**	**L**	**W**	**X**
H	**U**	**X**	**0**	**1**	**H**	**W**	**W**	**H**	**X**
-	**U**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**

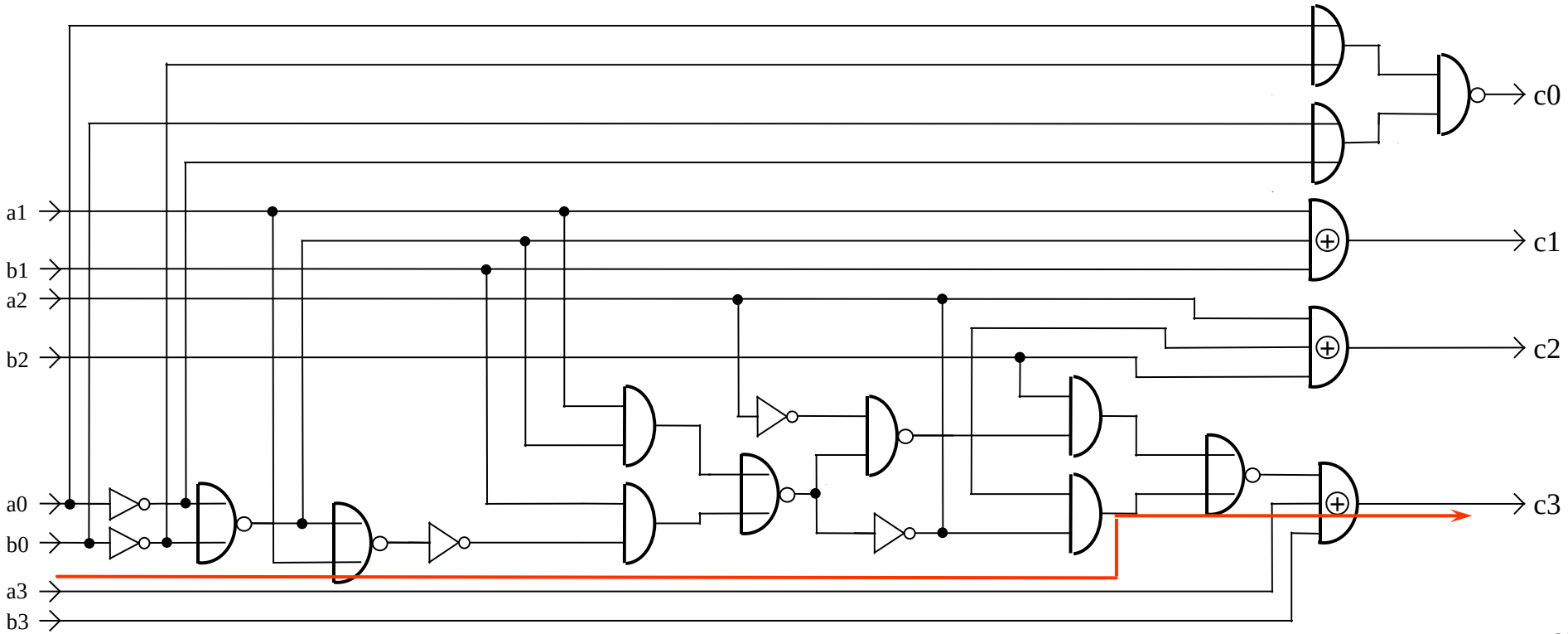
jeweiliges ‘Auflösungsergebnis‘ bei Kollision

4 bit-Addierer

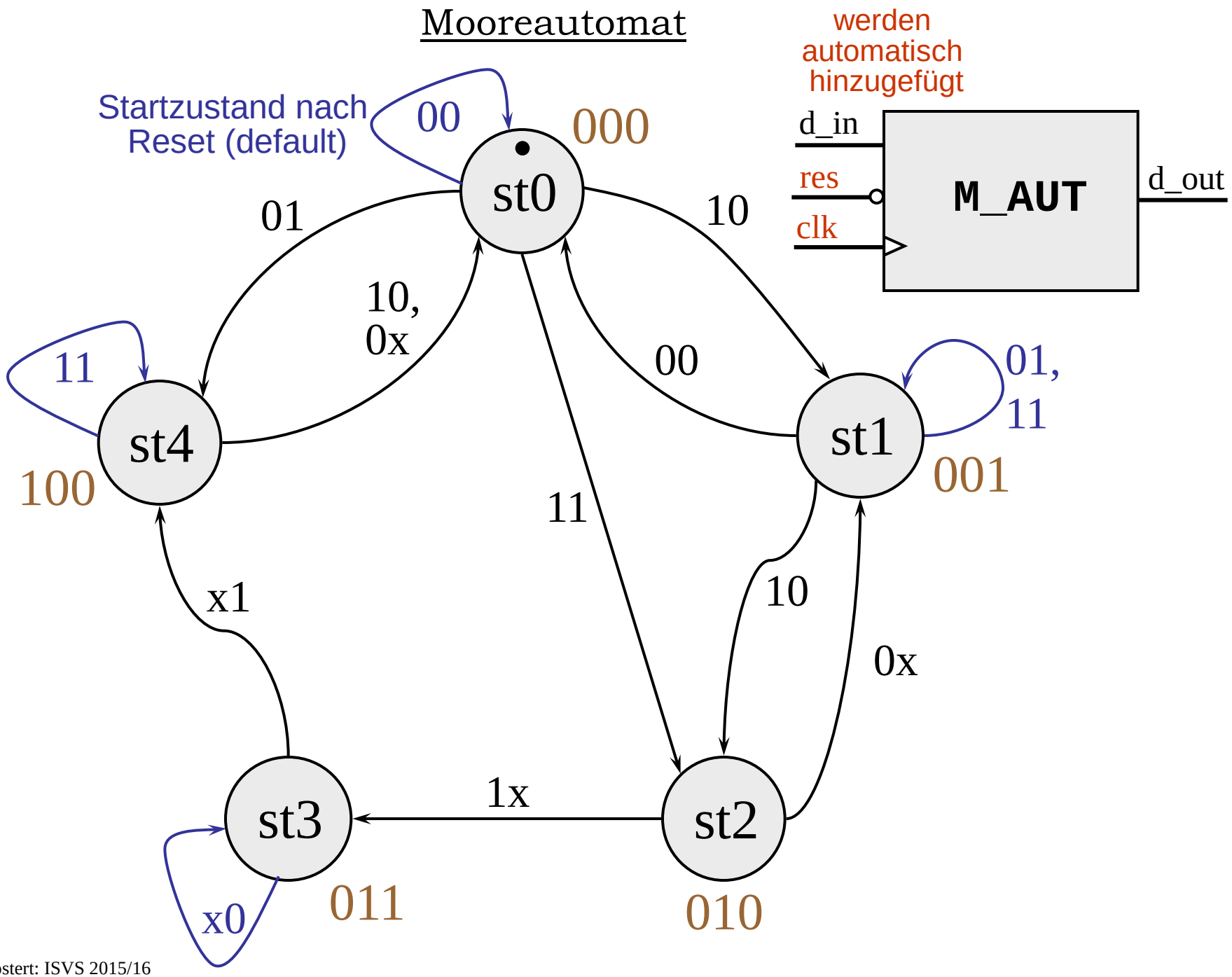
```
ENTITY Adder IS
PORT (a, b:   IN INTEGER RANGE 0 TO 15;
      c:      OUT INTEGER RANGE 0 TO 15)
END Adder;

ARCHITECTURE Addition OF Adder IS
BEGIN
c <= a + b;
END Addition;
```



(Carry nicht dargestellt)



Mooreautomat



Beschreibung mit VHDL

```
library IEEE;
use ieee.std_logic_1164.all;

entity M_AUT is
port (clk, res: in std_logic;
d_in: in std_logic_vector (1 downto 0);
d_out: out std_logic_vector (2 downto 0)
);
end M_AUT;
architecture FUNCT of M_AUT is
        type zustands_werte is (st0, st1, st2, st3, st4);
        signal akt_zust, folg_zust: zustands_werte;
        begin

-- Synchronisation mit dem Takt (Mooreverhalten)
Zust_Reg: process (clk, res))
            begin
             if (reset = '0') then
             akt_zust <= st0;
              elsif (clk ='1' and clock'event) then
              akt_zust <= folg_zust;
            end if;
end process Zust_Reg;
```

```
    -- Kombinatorik der FSM

    FSM: process (akt_zust, d_in)
begin
case akt_zust is
when st0 =>
    case d_in is
  when ”00” => folg_zust <= st0;
  when ”01” => folg_zust <= st4;
  when ”10” => folg_zust <= st1;
  when ”11” => folg_zust <= st2;
  when others => null;
end case;
   when st1 =>
   case d_in is
  when ”00” => folg_zust <= st0;
  when ”10” => folg_zust <= st2;
  when others => folg_zust <= st1;
end case;
   when st2 =>
   case data_in is
  when ”00” => folg_zust <= st1;
  when ”01” => folg_zust <= st1;
  when ”10” => folg_zust <= st3;
  when ”11” => folg_zust <= st3;
  when others => null;
end case;
```

```
when st3 =>
case d_in is
 when ”01” => folg_zust <= st4;
 when ”11” => folg_zust <= st4;
 when others => folg_zust <= st3;
end case;
when st4 =>
case d_in is
 when ”11” => folg_zust <= st4;
 when others => folg_zust <= st0;
end case;
 when others => folg_zust <= st0;
end case;
end process FSM;

-- Moore-Ausgabewerte nur von akt_zust abhängig

AUSGABE: process (akt_zust)
begin
case akt_zust is
 when st0    => d_out <= ”000”;
 when st1    => d_out <= ”001”;
 when st2    => d_out <= ”010”;
 when st3    => d_out <= ”011”;
 when st4    => d_out <= ”100”;
 when others => d_out <= ”000”;
end case;
end process AUSGABE;
end FUNCT;
```

Was ist eine 'Testbench'?

```
ENTITY TEST IS
END TEST;

ARCHITECTURE TESTBENCH OF TEST IS
SIGNAL      <Signalliste>; -- alle Schnittstellensignale der
                           -- zu untersuchenden Entity
COMPONENT <Komponentenname -- identisch mit Entityname der
                           -- zu untersuchenden Entity
      <Port-Liste>         -- identisch mit der Port-Liste
                           -- der zu untersuchenden Entity
END COMPONENT;
FOR ALL: <Komponentenname> USE ENTITY work.<Entityname>
                           (<Architekturname>);
BEGIN
          -- nebenläufige Signalzuweisungen an die Stimuli
          -- mit Angabe des Zeitpunkts der Signalübergänge

NAME: <Komponentenname>
PORT MAP (< Liste der angeschlossenen aktuellen Signale>);

END TESTBENCH;
```

LOOP-Anweisung

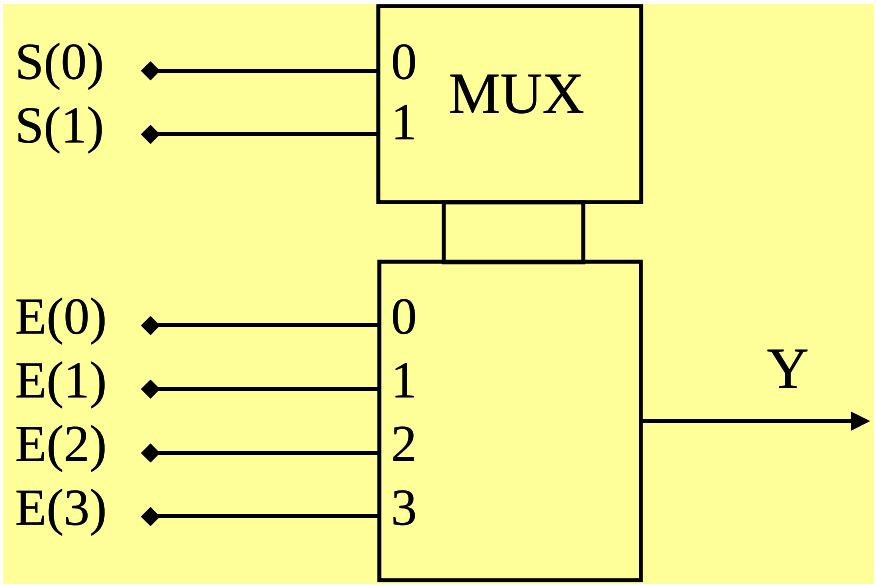
Iterationsschleifen, d.h. mehrfach zu durchlaufende Anweisungsblökke, können mittels der LOOP-Anweisung realisiert werden. Dabei existieren die folgenden drei Alternativen: FOR-Schleife, WHILE-Schleife und Endlosschleife:

```
[loop_label :] FOR range LOOP
   ...
   ...      -- sequentielle Anweisungen
   ...
END LOOP [loop_label] ;

[loop_label :] WHILE condition LOOP
   ...
   ...      -- sequentielle Anweisungen
   ...
END LOOP [loop_label] ;

[loop_label :] LOOP
   ...
   ...      -- sequentielle Anweisungen
   ...
END LOOP [loop_label] ;
```

```
LIBRARY IEEE;                                   -- allgemeines Beispiel
USE IEEE.STD_LOGIC_1164.ALL;                    -- für eine Testbench
ENTITY testbench IS                             -- keine Portdeklaration
END testbench;
ARCHITECTURE simulate OF testbench IS
COMPONENT testdesign
PORT (x: IN  INTEGER;
      y: OUT INTEGER
     );
END COMPONENT;
SIGNAL eingang, ausgang: INTEGER;
BEGIN
  TEST: testdesign PORT MAP(eingang, ausgang);
PROCESS
BEGIN
  LOOP                      -- Endlosschleife = Takt mit 25MHz (40ns)
        eingang <= 1;
        WAIT FOR 20 ns;
        eingang <=0;
        WAIT FOR 20 ns;
  END LOOP;
END PROCESS;
END simulate;
CONFIGURATION test_simulate OF testbench IS
  FOR simulate
    FOR TEST: testdesign USE ENTITY work.testdesign(testdesign_arch);
    END FOR;
  END FOR;
END test_simulate;
```

Testbencherstellung am Beispiel eines 4-zu-1-Multiplexers

```
entity MUX4X1 is
port (S:in bit_vector(1 downto 0); --2 Select-Eingänge S(0),S(1)
      E: in bit_vector(3 downto 0);-- 4-bit Eingangsdaten E
      Y: out bit);                 -- 1-bit-Ausgang Y
end MUX4X1;


 architecture VERHALTEN of MUX4X1 is
 begin
   with S select          -- Auswahlsignal
        Y <=   E(0) when "00",
               E(1) when "01",
               E(2) when "10",
               E(3) when "11";

 end VERHALTEN;
```



Testbench für den 4-zu-1-Multiplexer

```
entity TEST is
end TEST;
architecture VERHALTEN of TEST is
signal S1: bit_vector(l downto 0);
signal E1: bit_vector(3 downto 0);
signal Yl: bit;
component MUX4X1 is
              port(S: in bit_vector(l downto 0);
                   E: in bit_vector(3 downto 0);
                   Y: out bit);
end component;

for all: MUX4X1
use entity work.MUX4X1(VERHALTEN); -- Modell aus der Bibliothek "work"
 begin
      E1 <= "1010", "0101" after 400 ns;
      S1 <= "00", "01" after 100 ns, "10" after 200 ns,
     "11" after 300 ns, "00" after 400 ns, "01" after 500 ns,
     "10" after 600 ns, "11" after 700 ns, "00" after 800 ns;
C1:   MUX4X1 port map(S1, E1, Y1); -- Anschließen der Signale unter
end VERHALTEN;                      -- Beachtung der Reihenfolge
```

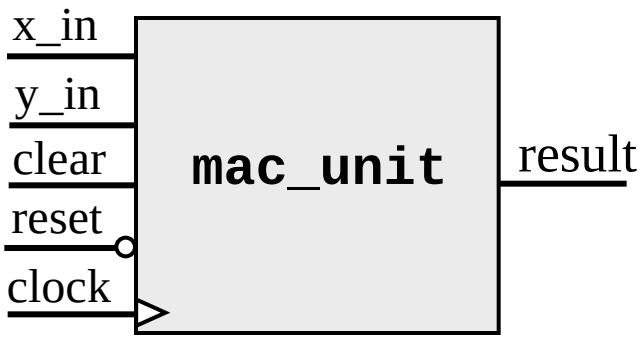
Testbench-Prozesse für nichtperiodische und periodische Stimuli

```
STIMULI: process              -- nichtperiodische Stimuli
    begin
      E(1)<='0','l'after 100 ns,'0'after 200 ns,'l'after 300 ns;
      E(2) <='0', '1' after 200 ns;
    wait;                     -- keine weiteren Signaländerungen

end process STIMULI;
```

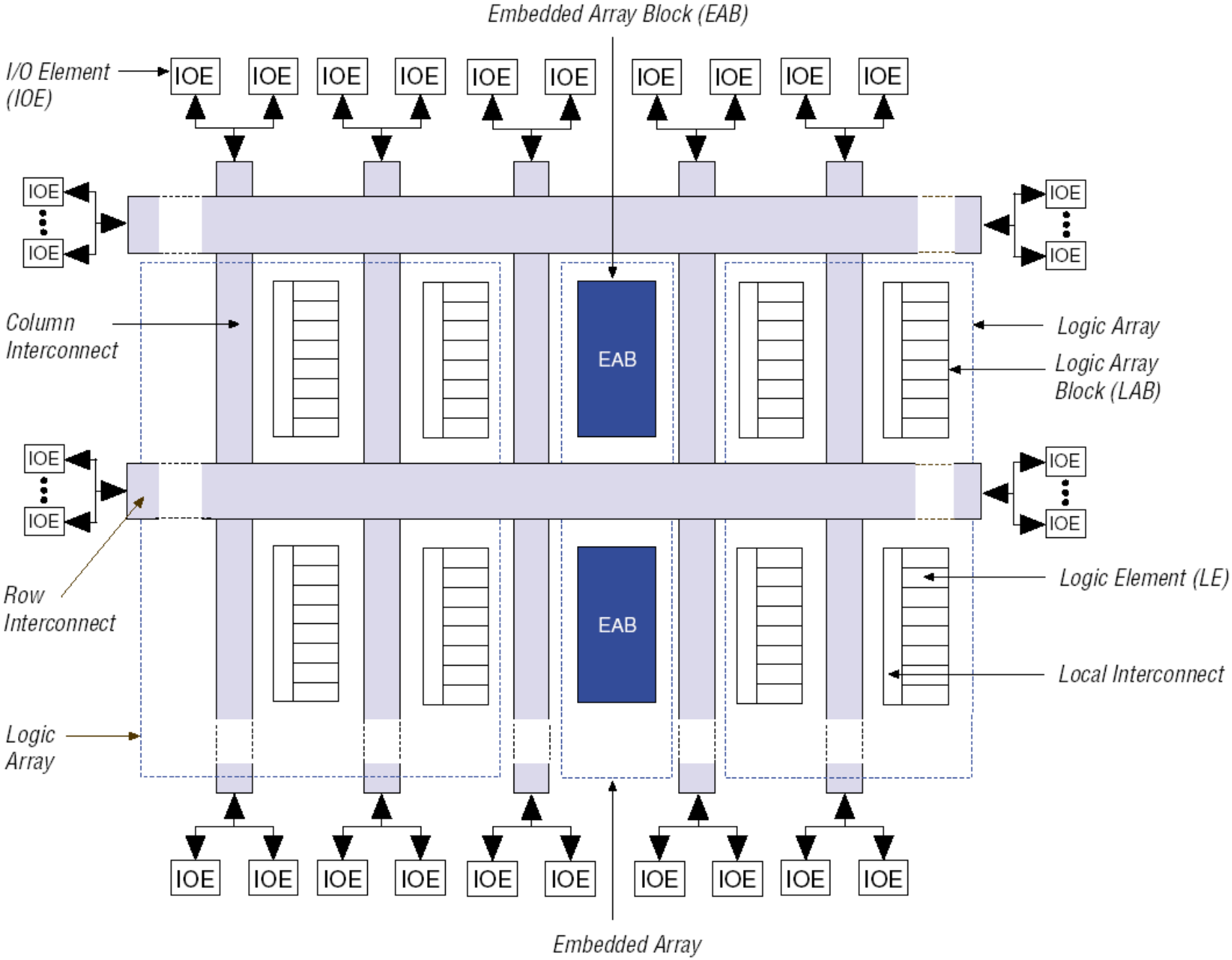
```
TAKTGEN: process
                  begin              -- periodischer Stimulus
                  E(0) <= '0';
                  wait for 50 ns;
                  E(0) <= '1';
                  wait for 50 ns;

end process TAKTGEN;
```

VHDL-Beschreibung einer MAC-Unit

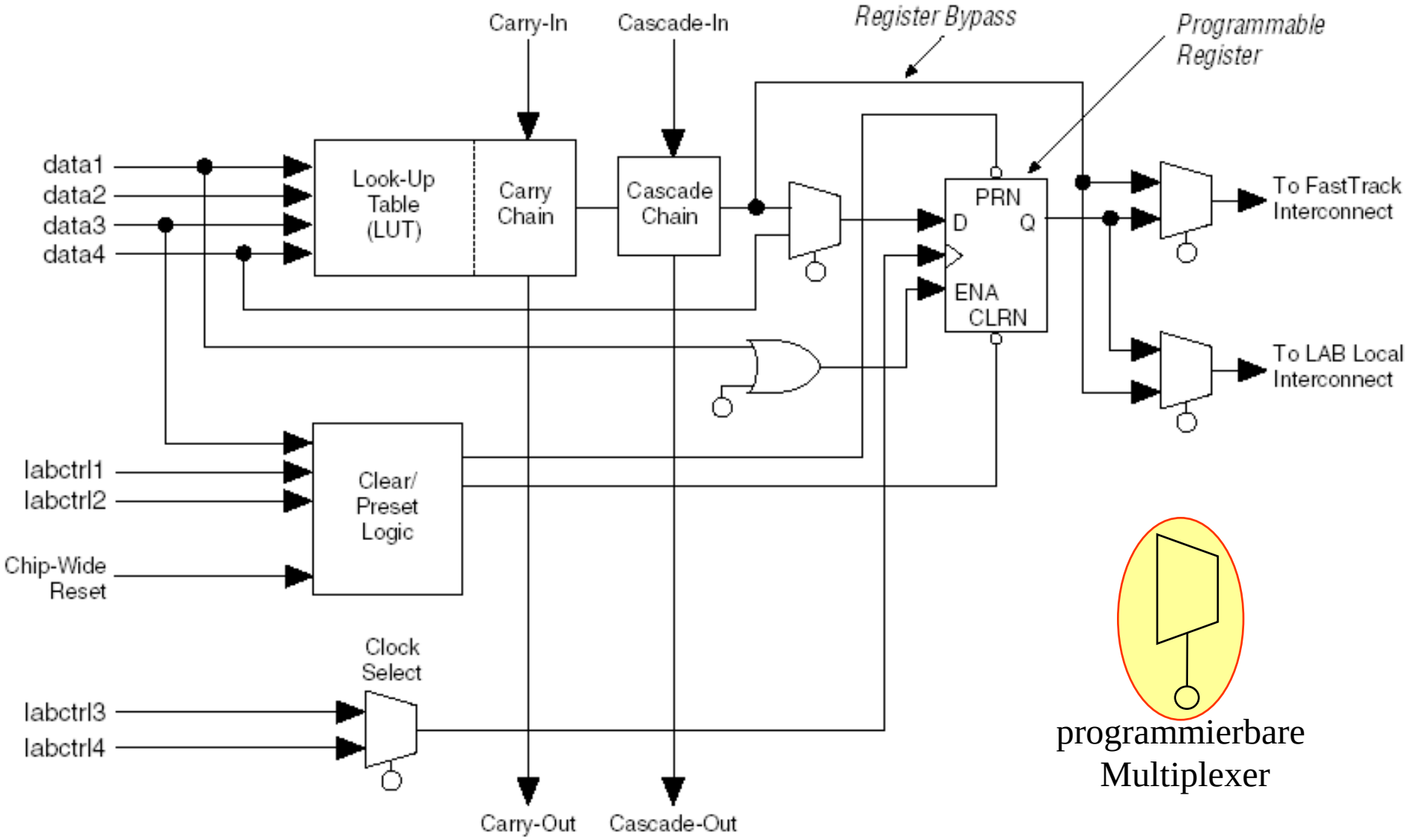
```
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_arith.all;
ENTITY mac IS
PORT (clock, reset, clear : IN bit;
      x_in, y_in: IN integer range -128 to 127;  -- 8bit
      result: OUT integer range -32768 to 32767  -- 16bit
      );
END mac;

ARCHITECTURE mac_arch OF mac IS
SIGNAL int_bus:integer range -32768 to 32767;
BEGIN
   PROCESS (clock, reset)
    BEGIN
     IF reset = '0' THEN
       int_bus <= 0;
       ELSIF clock'event AND clock= '1' THEN
       IF clear = '1'THEN – normale Operation
       int_bus <= int_bus + x_in * y_in;
       ELSE  -- sonst z.B. 1. Korrelationswert
       int_bus <= x_in * y_in;
       END IF;
    END IF;
  END PROCESS;
   result <= int_bus;
END mac_arch;
```



Grundstruktur von Altera-FPGAs

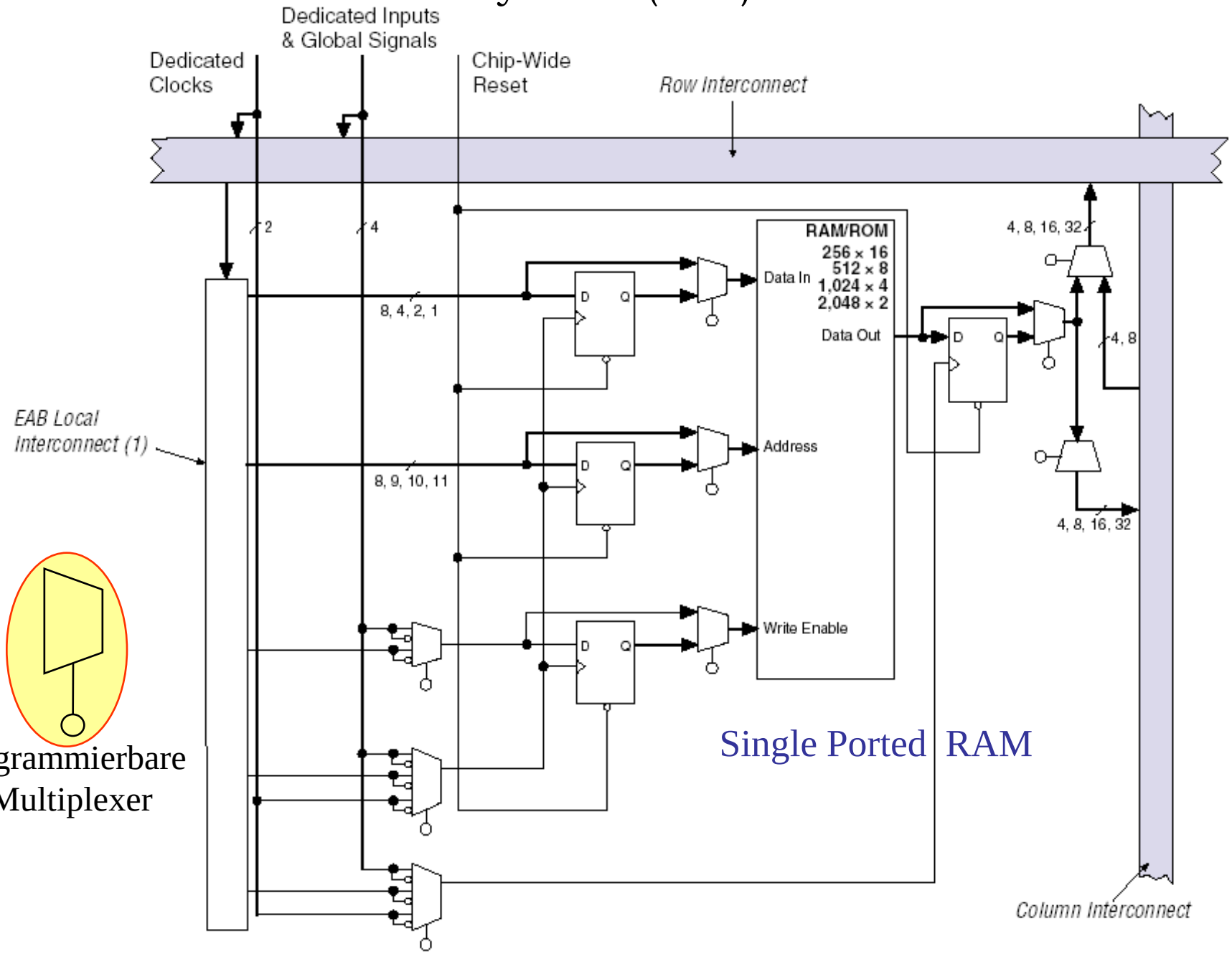


Aufbau eines Logik-Elements (LE) in den LABs von ACEX-FPGAs



programmierbare Multiplexer

Embedded Array Block (EAB) in einem FPGA

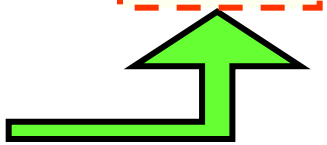


programmierbare Multiplexer

Single Ported RAM

Typ: Cyclone III

Feature	EP3C5	EP3C10	EP3C16	EP3C25	EP3C40	EP3C55	EP3C80	EP3C120
Logic Elements	5,136	10,320	15,408	24,624	39,600	55,856	81,264	119,088
Memory (Kbits)	414	414	504	594	1,134	2,340	2,745	3,888
Multipliers	23	23	56	66	126	156	244	288
PLLs	2	2	4	4	4	4	4	4
Global Clock Networks	10	10	20	20	20	20	20	20



Einsatz in Forschungsprojekten des IIIT

Prinzip der FPGA-Programmierung mittels statischer RAM-Zellen und bidirektionalen Schaltern