

Einführung

- ♦ **Erweiterte Backus-Naur-Form (EBNF)**
 - ist eine Erweiterung der Backus-Naur-Form (BNF) und stellt eine formale Metasprache dar, die zur Beschreibung kontextfreier Grammatiken benutzt wird.

Symbol	Bedeutung
=	ist definiert als
.	Ende einer Produktionsregel
"abc"	das Terminalsymbol abc
Bezeichner	das nichtterminale Symbol
X Y	Alternative (Exklusiv-oder)
[X]	0- oder 1-maliges Auftreten von X
{ X }	0- oder mehrmaliges Auftreten von X
(X Y)	Zusammenfassung: entweder X oder Y

Terminalsymbole: fett hervorgehoben, Großbuchstaben

Einführung in VHDL

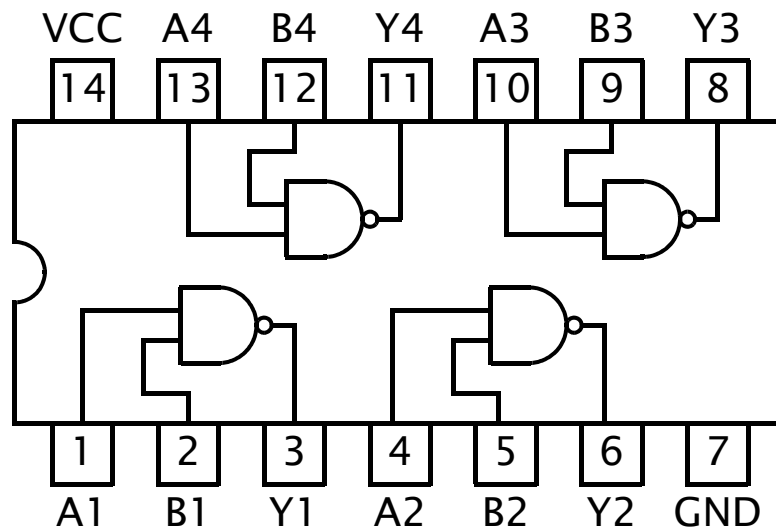
- ♦ VHDL – VHSIC Hardware Description Language
- ♦ VHSIC – Very High Speed Integrated Circuit
- ♦ Die Hardwarebeschreibungssprache VHDL wurde 1987 als IEEE-1076-Standard festgelegt und 1993 erweitert.
- ♦ VHDL ist ursprünglich als Beschreibungs- und Simulations-sprache entwickelt worden.
- ♦ VHDL eignet sich für
 - Spezifikation,
 - Dokumentation und
 - technologieunabhängige Beschreibung
- ♦ digitaler Systeme und Schaltungen auf verschiedenen Abstraktionsebenen

Einführung in VHDL

- ♦ VHDL ist für Entwürfe komplexer Systeme (FPGA-/CPLD-/ASIC-Design) ausgelegt:
 - getrennte Übersetzung einzelner Module;
 - Top-Down-/Bottom-Up-Entwurf;
 - Modularisierung und Hierarchiebildung (Syntaxkonstrukte wie z.B.: ENTITY, COMPONENT, PROCESS, PROCEDURE).
- ♦ Die Beschreibung einer Schaltung in VHDL kann auf unterschiedliche, funktional äquivalente Arten erfolgen. Allerdings bringt das bei einer Schaltungssynthese auch einer VHDL-Beschreibung nicht immer die gleichen Ergebnissen hervor.
- ♦ Aus der Sicht der Synthese enthält VHDL Sprachkonstrukte, die überhaupt nicht (FILE, ASSERT, AFTER) oder nur bedingt (RECORD, REGISTER, WHILE) in Schaltungsstrukturen umgesetzt werden können.

Einführung in VHDL

♦ Einführendes Beispiel am SN7400



```
ARCHITECTURE logic OF SN7400 IS
BEGIN
    Y1 <= NOT(A1 AND B1);
    Y2 <= NOT(A2 AND B2);
    Y3 <= NOT(A3 AND B3);
    Y4 <= NOT(A4 AND B4);
END ARCHITECTURE logic;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

-- Quad 2-Input NAND Gate
ENTITY SN7400 IS
    PORT(A1, A2, A3, A4: IN  std_logic;
         B1, B2, B3, B4: IN  std_logic;
         Y1, Y2, Y3, Y4: OUT std_logic);
END ENTITY SN7400;
```

```
LIBRARY unisim;
USE unisim.vcomponents.ALL;

ARCHITECTURE structure OF SN7400 IS
BEGIN
    g1: NAND2 PORT MAP(Y1, A1, B1);
    g2: NAND2 PORT MAP(Y2, A2, B3);
    g3: NAND2 PORT MAP(Y3, A3, B3);
    g4: NAND2 PORT MAP(Y4, A4, B4);
END ARCHITECTURE structure;
```

Einführung in VHDL

- ♦ Die Schaltungssynthese aus einer VHDL-Beschreibung ist nur unter Einhaltung bestimmter Beschreibungsregeln und eines bestimmten Beschreibungsstils möglich.
- ♦ Eine simulations- und synthesesfähige VHDL-Beschreibung einer Komponente besteht aus:
 - einer Schnittstellenspezifikation (ENTITY);
 - einer oder mehreren Architekturbeschreibungen (ARCHITECTURE);
 - optional einer oder mehreren Konfigurationsvorgaben (CONFIGURATION);
 - optional einer oder mehreren Bibliotheken (PACKAGE).

Grundbegriffe in VHDL

- ◆ Entwurfseinheit (entity)
 - ist ein zusammenhängendes, in sich abgeschlossenes System.
 - ist durch eine Schnittstellenbeschreibung (entity declaration) für die Umgebung sichtbar.
 - entspricht einem Symbol in einer grafischen Schaltungsbeschreibung.
- ◆ Architektur (architecture)
 - enthält die Beschreibung der Funktionalität der modellierten Komponente.
 - Alle simulierbaren/synthetisierbaren Entwurfseinheiten haben eine Architekturbeschreibung.
 - Für eine Komponente können mehrere Architekturen existieren:
 - auf unterschiedlichen Abstraktionsebenen (Systemebene, algorithmische Ebene, Registertransferebene, Logikebene);
 - aus verschiedenen Sichten (Verhalten, Struktur) beschreiben;
 - mit verschiedene Entwurfsalternativen.

Grundbegriffe in VHDL

- ◆ Konfiguration (configuration)
 - legt fest, welche der beschriebenen Architekturen einer bestimmten Schnittstellenspezifikation zugeordnet ist und welche Zuordnungen für möglicherweise verwendete Submodule in der Architektur gelten.
- ◆ Paket (package)
 - ist eine Sammlung gemeinsam genutzter Konstanten, Datentypen, Objekten, datentypspezifischer Operatoren und Unterprogrammen in einem Entwurf.
- ◆ Prozeß (process)
 - ist die Grundausführungseinheit in VHDL;
 - dient als Umgebung für sequentielle, d.h. nacheinander ablaufende Anweisungen;
 - wird zur Modellierung prozeduraler Vorgänge verwendet.

Lexikalische Elemente

- ◆ Kommentare
 - dienen zur besseren Lesbarkeit von VHDL-Quellcode
 - haben keinerlei Bedeutung für die Funktion eines Modells.
Ausnahme: Steueranweisungen für Synthesewerkzeuge, die oft innerhalb eines Kommentars stehen.
 - werden durch den doppelten Bindestrich ("--") eingeleitet und reichen dann bis zum Ende einer Zeile.
 - können zu Beginn einer Zeile oder nach einer VHDL-Anweisung stehen.

Beispiele:

```
-- das ist eine Kommentarzeile  
PC <= PC + 1; -- Inkrementieren des Befehl szählers
```


Lexikalische Elemente

♦ Begrenzungszeichen

- Leerzeichen, Tabulator, Zeilenumbruch

Die Verwendung von Leerzeichen, Tabulatoren und Zeilenumbrüchen dient dann nur der besseren Lesbarkeit des VHDL-Textes durch den Menschen, hat aber keinen Einfluß auf die syntaktische Bedeutung des VHDL-Textes.

- Die Formatierung eines Programms unterstützt die Selbstdokumentation eines Programmtextes.

♦ Einzel- und zusammengesetzte Operatoren und Trennungssymbole:

() | ' . , : ; / * - < = > & +
=> >= <= := /= <> ** --

Lexikalische Elemente

- ◆ Bezeichner (identifizier)
 - sind Namen von Entwurfseinheiten, Objekten, Datentypen, Funktionen, Prozeduren, Instanzen von Komponenten usw..
- ◆ Bei der Wahl von Bezeichnern sind folgende Regeln zu beachten:
 - Bezeichner bestehen aus einer Folge von Buchstaben, Ziffern und einzelnen Unterstrichen
 - das erste Zeichen eines Bezeichners muß ein Buchstabe sein
 - sie dürfen keine Leer- oder Sonderzeichen enthalten
 - der Unterstrich ('_') darf nicht am Anfang oder Ende des Bezeichners und nicht zweimal unmittelbar aufeinanderfolgend verwendet werden
 - Bezeichner dürfen keine reservierten Worte sein
 - Bezeichner sind case-insensitiv

Übersetzungseinheiten und Bibliotheken

- ◆ Übersetzungseinheiten
 - Schnittstellen: ENTITY oder PACKAGE
 - Implementierungen: ARCHITECTURE oder PACKAGE BODY
 - Konfigurationen: CONFIGURATION
 - zusammengefaßt in Projektbibliotheken (z.B. work, std, ieee)
 - Änderungen im Implementierungsteil einer Übersetzungseinheit haben keinen unmittelbaren Einfluß auf andere Übersetzungseinheiten. => Eine Nachübersetzungen ist nicht notwendig.
- ◆ Projektbibliotheken
 - Der Aufbau und die Verwaltung von Projektbibliotheken liegt in der Verantwortung des Simulators oder des Rechnersystems, und ist nicht vom VHDL-Standard vorgeschrieben.
 - Projektbibliotheken werden meistens als eigene Verzeichnisse in einem Dateisystem realisiert.
 - Übersetzungseinheiten in Projektbibliotheken werden über logische Namen (VHDL-Bezeichner) angesprochen.

Projektbibliotheken

- Die Verbindung zwischen einem logischen VHDL-Namen und dem physikalischen Speicherort (einem Verzeichnispfad) ist eine Aufgabe des Simulators/Synthesewerkzeugs und wird mit Hilfe sog. werkzeugspezifischer Konfigurationsdateien festgelegt.
- Auszug aus der werkzeugspezifischen Konfigurationsdatei modelsim.ini (VHDL-Simulator von MentorGraphics Corp.)

```
[Library]
std          = $MODEL_TECH/../../std
ieee         = $MODEL_TECH/../../ieee
vital2000    = $MODEL_TECH/../../vital2000
synopsys     = $MODEL_TECH/../../synopsys
modelsim_lib = $MODEL_TECH/../../modelsim_lib
...
; Xilinx Section
unisim       = $MODEL_TECH/../../xilinx/vhdl/unisim
simprim      = $MODEL_TECH/../../xilinx/vhdl/simprim
...
```

LIBRARY-Klausel

- ♦ Die LIBRARY-Klausel dient dazu, Projektbibliotheken in VHDL-Programmen bekannt zu machen.

```
LIBRARY_Klausel =  
  library Bibliothekenbezeichner { "," Bibliothekenbezeichner } ";" .
```

- Die Bibliotheken STD und WORK sind implizit (d.h. ohne LIBRARY-Klausel) bekannt, und brauchen nicht deklariert zu werden.
 - In der Bibliothek STD werden i.d.R. allgemeine Pakete (z.B. standard, textio) abgelegt.
 - Die Bibliothek WORK dient als sog. Arbeitsbibliothek zum Abspeichern von selbst kompilierten Modellen.
- Andere Bibliotheken müssen explizit durch Angaben von Bibliothekenbezeichnern sichtbar gemacht werden.
- Die LIBRARY-Klausel wird beim Bedarf in jeder Übersetzungseinheit verwendet.
- Sie steht gewöhnlich direkt vor einer Entity, einer Architektur, einer Konfiguration, einem Paket oder vor dem Pakettrumpf.

USE-Klausel

- ♦ Die USE-Klausel dient dazu, nach der Bekanntgabe einer Projektbibliothek mit der LIBRARY-Klausel dort vorhandene Übersetzungseinheiten sichtbar zu machen.

```
USE_Klausel =  
    use selektierter_Bezeichnung { "," selektierter_Bezeichnung } ";" .  
  
selektierter_Bezeichnung =  
    Bibliothekenbezeichner "." [ Paketbezeichner "." ]  
    ( Deklarationsbezeichner | all ) .
```

- Alle Einträge der Standard-Pakete STD.standard und STD.textio sind implizit (d.h. auch ohne USE-Klausel) verfügbar, und brauchen nicht explizit mit einer USE-Klausel sichtbar gemacht zu werden.

```
library IEEE;
```

```
use IEEE.std_logic_1164.all;  
use IEEE.std_logic_unsigned.all;  
use IEEE.std_logic_arith.all;  
use IEEE.std_logic_textio.all;
```

```
library XilinxCoreLib;
```

```
library SIMPRIM;  
use SIMPRIM.VComponents.all;  
  
library UNISIM;  
use UNISIM.VComponents.all;
```

Typkonzept in VHDL

- ♦ VHDL ist streng (stark) typisierte (typgebunde) Sprache:
 - Objekte eines Typs können nur die durch diesen Typ definierten Werte annehmen, und auf diese Objekte sind nur die für diesen Typ definierten Operationen anwendbar.
 - Automatische oder implizite Typkonvertierungen sind weitgehend verboten.
 - Typkonvertierungen müssen explizit mit Konvertierungsfunktionen durchgeführt werden. Bei gleichartigen Typen (numerische Typen oder gleichstrukturierte Reihungstypen) ist eine Cast-Konvertierung in der Form `Typname(Ausdruck)` erlaubt.
 - Das strenge Typkonzept erhöht die Zuverlässigkeit:
 - der Compiler kann bereits zur Übersetzungszeit die meisten Semantik-Fehler finden.
 - das Laufzeitsystem kann auf Semantik-Fehler, die zur Laufzeit/Simulation auftreten, gezielt reagieren.

Typkonzept in VHDL

- ♦ Vier Datentypklassen in VHDL:

- skalare Datentypen (scalare types):

- diskrete Typen:

- » Aufzählungstyp (enumeration)

- » Bereichstyp (range)

- physikalischer Typ (physical)

- Gleitkommatyp (real)

- zusammengesetzte Datentypen (composite types):

- Reihungstyp (array)

- Verbundtyp (record)

Mit zusammengesetzten Typen lassen sich Daten zu komplexen Datentypen organisieren.

- Zugriffstyp (access type)

- Dateityp (file type)

Datentyp

- ♦ Ein Datentyp in VHDL definiert sowohl eine Menge von Werten, die ein Objekt annehmen kann, als auch Operationen, die auf diesen Werten angewendet werden dürfen.

```
Typdefinition =  
    type Bezeichner is  
        Aufzählungstyp | Bereichstyp | Gleitkommatyp | Physikalischer_Typ  
        | Reihungstyp | Verbundtyp | Zugriffstyp | Dateityp ";" .
```

- Die Zuweisung und der Vergleich auf Gleichheit sind zwei Grundoperationen für jeden Datentyp.
- ♦ Eine Untertypdefinition schränkt nur die Wertemenge eines zuvor definierten Datentyps ein, ohne dadurch einen neuen Typ einzuführen.

```
Untertypdefinition = subtype Bezeichner is Bezeichner [ Bereichstyp ] .
```

- Der Untertyp und der (Basis-)Typ sind kompatibel.
- Objekte mit verschiedenen Untertypen des gleichen Basistyps können mit den Operatoren des Basistyps verknüpft werden.

Aufzählungstyp

- ♦ Der Aufzählungstyp definiert eine geordnete Menge von Werten, die durch sog. Aufzählungsliterale (Bezeichner oder Zeichenlitterale) repräsentiert werden. Die Ordnung ist durch die Aufschreibungsreihenfolge vorgegeben.

```
Aufzählungstyp = "(" Literal { "," Literal } ")" .
```

- einige Standard-Aufzählungstypen:

```
-- vordefinierte Typen aus dem standard-Paket:  
type boolean   is (false, true);  
type bit        is ('0', '1');  
type character is (... 256 ASCII-Zeichen ...);
```

- selbstdefinierte Aufzählungstypen:

```
type TOpCode    is (ADD, ADC, SUB, SBC, JMP, MOV, LOAD, STORE);  
  
type TOctValue is ('0', '1', '2', '3', '4', '5', '6', '7');  
  
type TState     is (idle, run, error, send, receive);
```

Aufzählungstyp

- Bezeichner oder Zeichenliterale eines Aufzählungstyps werden implizit mit nicht negativen numerischen Werten von links nach rechts aufsteigend durchnumeriert.
- Das erste (am weitesten links stehende) Element in der Aufzählung hat die Position Null.
- Der Aufzählungstyp zur Modellierung 9-wertiger Logik:

-- vordefinierter Typ aus dem std_logic_1164-Paket:

```
type std_ulogic is ( 'U',  -- Uninitialized      nicht initialisiert
                     'X',  -- Forcing Unknown   stark unbekannt
                     '0',  -- Forcing 0         starke logische 0
                     '1',  -- Forcing 1         starke logische 1
                     'Z',  -- High Impedance    hochohmig, für Busse
                     'W',  -- Weak Unknown     schwach unbekannt
                     'L',  -- Weak 0           schwache logische 0
                     'H',  -- Weak 1           schwache logische 1
                     '-'   -- Don't care       egal
                     );
```

Aufzählungstyp

♦ Grundoperation:

- Neben Zuweisungen (": =", "<=") auch Vergleichsoperationen ("=", "/=", "<", ">", "<=", ">=")
- vier Standard-Funktionen/Attribute:
 - Vorgängerfunktion (PRED) TOpCode' PRED(JMP) = SBC
 - Nachfolgerfunktion (SUCC) TOpCode' SUCC(JMP) = MOV
 - Positionsfunktion (POS) std_ul ogic' POS(' H') = 7
 - Wertfunktion (VAL) std_ul ogic' VAL(4) = ' Z'
 - weitere Attribute ...

Typkonflikte und Auflösung mit Konvertierungsfunktionen:

```
variable var1: bit          := '1';  
variable var2: std_ulogic := '1';
```

```
...  
-- Typfehler  
if var1 = var2 then  
    enable2 := 'Z';  
end if;  
...
```

```
...  
-- mit einer Konvertierungs-  
-- funktion  
if var1 = To_bit(var2) then  
    var2 := 'Z';  
end if;  
...
```

Bereichstyp

- ♦ Der Bereichstyp legt den Wertebereich des Typs durch eine explizite Angabe einer Ober- und Untergrenze fest.

```
Bereichstyp = range Ausdruck ( to | downto ) Ausdruck .
```

- Die Angaben zur Ober- und Untergrenze müssen konstante Ausdrücke sein, also zur Übersetzungszeit berechenbar.
- Mit den Schlüsselwörtern **to** und **downto** lassen sich entsprechend aufsteigende und absteigende Wertebereiche vereinbaren.
- Auch die Integer- und Real-Typen sind als Bereichstypen definiert:

```
-- vordefinierte Typen aus dem standard-Paket:
```

```
type integer is range -2**31 to 2**31 - 1;      -- -2147483648 to +2147483647
```

```
type real    is range -1.0E308 to 1.0E308;      -- -1.0E38 to +1.0E38
```

- Ein ganzzahliger Bereichstyp umfaßt alle Zahlen zwischen Ober- und Untergrenze. Alle Operationen mit Daten dieses Typs sind genau und entsprechen den üblichen arithmetischen Gesetzen.

```
type index is range 15 downto 0;
```

```
type wahrscheinlichkeit is range 0.0 to 1.0;
```

Bereichstyp

- ♦ Mit einer Untertypdefinition kann man die Wertemenge eines zuvor definierten Bereichs- oder Aufzählungstyps einschränkt, ohne dadurch einen neuen Datentyp einzuführen.
 - Operationen können auf Typen und Untertypen in gleicher Weise angewendet werden, sofern keine Bereichsbeschränkungen verletzt werden.

```
-- vordefinierte Typen aus dem standard-Paket:
subtype natural is integer range 0 to integer'high;
subtype positive is integer range 1 to integer'high;

-- vordefinierte Typen aus dem std_logic_1164-Paket:
subtype X01Z is std_ulogic range 'X' to 'Z'; -- ('X','0','1','Z')
subtype UX01 is std_ulogic range 'U' to '1'; -- ('U','X','0','1')

type elektronische_elemente is
    (spule, kondensator, widerstand, leitung, klemme,
     diode, bipolarer_transistor, unipolarer_transistor);
subtype passive_elemente is elektronische_elemente range spule to klemme;
subtype aktive_elemente is elektronische_elemente
    range diode to unipolarer_transistor;
subtype transistor is aktive_elemente
    range bipolarer_transistor to unipolarer_transistor;
```

Reihungstyp

- ♦ Der Reihungstyp ist eine sog. homogene Struktur, die sich aus Komponenten zusammensetzt, die alle vom selben Datentyp sind.
 - Die Definition eines Reihungstyps legt sowohl den Typ der Komponenten als auch den der Indizes fest.

```
Reihungstyp =  
    array "(" Indextyp { "," Indextyp } ")" of Komponententyp .  
  
Indextyp = Bezeichner [ range "<>" ] .
```

- Die Anzahl der Indizes definiert die Dimension eines Reihungstyps.
 - Indextypen müssen diskrete Typen sein: (Aufzählungstypen oder Bereichstypen).
 - Indextypen verschiedener Dimensionen eines Reihungstyps können unterschiedlich sein.

```
type monat is (JAN, FEB, MAR, APR, MAI, JUN, JUL, AUG, SEP, OKT, NOV, DEZ);  
type tag    is (MO, DI, MI, DO, FR, SA, SO);  
type woche is range 1 to 52;  
type kalender is array (tag, woche, monat) of positive;
```

Reihungstyp

- Eindimensionale und zweidimensionale Reihungstypen werden auch oft entsprechend als Vektoren und Matrizen bezeichnet.
- Für mehrdimensionale Reihungstypen wird für gewöhnlich die Bezeichnung Felder verwendet.
- In VHDL unterscheidet man zwischen eingeschränkten und uneingeschränkten (unconstrained array) Reihungstypen.

```
-- vordefinierte Reihungstypen aus dem std_logic_1164-Paket:  
type stdlogic_1d    is array (std_ulogic) of std_ulogic;  
type stdlogic_table is array (std_ulogic, std_ulogic) of std_ulogic;
```

```
-- vordefinierte uneingeschränkte Reihungstypen aus dem std_logic_1164-Paket:  
type std_ulogic_vector is array (natural range <>) of std_ulogic;  
type std_logic_vector  is array (natural range <>) of std_logic;
```

- Bei eingeschränkten Reihungstypen sind die Indizes-Bereiche (also die oberen und unteren Grenzen) aller Objekte des Typs gleich. Somit haben alle Objekte eines solchen Typs die gleiche Größe.

Reihungstyp

- Bei uneingeschränkten Reihungstypen können verschiedene Reihungsobjekte unterschiedliche Grenzen haben:
 - Die Indextypen der Dimensionen sind für alle Objekte des Reihungstyps gleich, aber verschiedene Reihungsobjekte können Indexwerte unterschiedlicher Unterbereiche für den Indexwert einer Dimension haben.
 - Die Bereichsgrenzen eines uneingeschränkten Reihungsobjektes können auch durch Initialisierung festgelegt werden.

```
-- aus dem Package standard
type string          is array (positive range <>) of character;
type bit_vector      is array (natural range <>) of bit;

-- aus dem Package std_logic_arith
type unsigned        is array (natural range <>) of std_logic;
type signed          is array (natural range <>) of std_logic;

constant err_msg:    string(1 to 5) := "ERROR";
constant war_msg:    string := "WARNING";
constant delimiter:  std_logic_vector := "01111110"

signal mem_adr:  std_logic_vector(15 downto 0);
signal acc_reg:  std_logic_vector(7 downto 0);
```

Reihungstyp

- Für Vektoren mit diskreten Komponententypen sind neben "=" und "/=" auch die Relationsoperatoren "<", "<=", ">" und ">=" auf der Grundlage der lexikographische Ordnung definiert.
- Vordefinierte Operationen für den Reihungstypen sind:
 - Indizierung, Aggregatbildung,
 - Ausschnittbildung, Vergleich,
 - Zuweisung und Konkatenation.
- Von Feldern kann man indizierte Komponenten bilden:
 - Bei der Indizierung muß ein Index für jeden Indexbereich des Feldes angegeben werden.
 - Das Laufzeitsystem prüft beim Indizieren während der Simulation, ob die Indizes innerhalb der zulässigen Indexbereiche liegen.
- Von eindimensionalen Feldern können Ausschnitte (slices) gebildet werden. Unter der Bildung eines Ausschnittes eines Feldes versteht man die Einschränkung des Indexbereiches eines eindimensionalen Feldes auf einen Teilbereich.

Objekte in VHDL

- ◆ Konstanten sind Objekte mit festen Werten, die während der Ausführung eines Modells nicht geändert werden können:

```
Konstantendeklaration =  
    constant Bezeichnerliste ":" Typbezeichner [ "!=" Ausdruck ] ";" .  
  
Bezeichnerliste = Bezeichner { "," Bezeichner } .
```

- Eine Konstantendeklaration gibt den Namen der Konstante, ihren Datentyp und optional ihren Initialisierungswert an.
- In VHDL werden sog. offene Konstanten (deferred constants) unterstützt, die nur in Paketen deklariert werden dürfen und deren Wert erst im Pakettrumpf spezifiziert wird.

```
constant delimiter : bit_vector(0 to 7) := "01111110";
```

```
constant tpd       : time      := 20 ns;
```

```
constant size      : natural  := 16;
```

```
constant message   : string   := "Segmentation fault";
```

Objekte in VHDL

- ◆ Variablen sind Objekte mit veränderbaren Werten:

```
Variablendeklaration =  
[ shared ] variable Bezeichnerliste ":" Typbezeichner [ "!=" Ausdruck ] ";" .
```

- Sie dienen zur lokalen Aufbewahrung von temporären Daten und werden hauptsächlich in sequentiellen Bereichen (Prozessen und Unterprogrammen) eingesetzt.
- Das Verhalten von Variablen in VHDL entspricht weitgehend dem von Variablen aus bekannten Programmiersprachen: sie nehmen ihren neuen Wert unmittelbar nach der Zuweisung an.
- In VHDL'93 werden sog. gemeinsame Variablen (shared variables) unterstützt: sie werden im Deklarationsbereich einer Architektur vereinbart, und auf sie darf in Prozessen lesend und schreibend zugegriffen werden (nicht deterministisch, nicht synthetisierbar).

```
variable counter : integer := 0;  
variable delta   : real := 0.1;  
variable adresse : integer range 0 to 16#FFFF#; -- mit 0 initialisiert
```

Objekte in VHDL

- ◆ Signale sind Objekte mit veränderbaren Werten und einem Zeitverhalten.

```
Signaldeklaration =  
  signal Bezeichnerliste ":" Typbezeichner [ Signalart ] [ ":=" Ausdruck ] ";" .  
  
Signalart = register | bus .
```

- Signale dienen zur
 - Verbindung einzelner Komponenten innerhalb eines VHDL-Entwurfes,
 - Modellierung zeitlichen Verhaltens elektronischer Systeme.
- Signale haben ein anderes Verhalten als Variablen: sie bekommen einen neuen Wert erst nach einer gewissen Zeit zugewiesen, z.B. am Ende eines Prozesses oder nach Ablauf einer vorgegebenen Verzögerung.

```
signal counter : integer := 0;  
signal delta   : real := 0.1;  
signal adresse : integer range 0 TO 16#FFFF#;
```

Schnittstellenbeschreibung

- ♦ Die Schnittstellenbeschreibung spezifiziert den Namen einer Entwurfseinheit und gibt deren Schnittstellen an.

```
Schnittstellenbeschreibung =  
  entity Modulbezeichner is  
    [ generic "(" Interfacekonstante { ";"  
      Interfacekonstante } ")" ";" ]  
    [ port      "(" Interfacesignal { ";"  
      Interfacesignal } ")" ";" ]  
    [ Deklarationsbereich ]  
    [ begin  
      Anweisungsbereich ]  
end [ entity ] [ Modulbezeichner ] ";" .
```

- Die Schnittstellenbeschreibung umfaßt vier optionale Definitionsbereiche mit
 - Interfacekonstanten (generic constants),
 - Interfacesignalen (ports),
 - einen Deklarationsbereich mit weiteren Vereinbarungen für gemeinsame Konstanten, Typen, Unterprogramme usw., aber keine Variablen,
 - einem Anweisungsbereich (nicht synthesefähig).

Schnittstellenbeschreibung

- ♦ Interfacekonstanten (sog. generische Konstanten) stellen eine einheitliche Schnittstelle zur Parametrisierung einer Entwurfseinheit dar.

```
Interfacekonstante =  
  [ constant ] Bezeichnerliste ":" [ in ] Typbezeichner [ "!=" Ausdruck ] .
```

- Innerhalb einer Entwurfseinheit verhalten sich Interfacekonstanten wie gewöhnliche Konstanten.
- Außerhalb einer Entwurfseinheit können Interfacekonstanten neue Werte bei der Instanzierung zugewiesen werden.

```
entity SSC_Interface is  
  generic(LENDEF: natural    := 8;      -- default length of data registers  
          RSTDEF: std_logic := '0';    -- reset default value  
          TK_DEF: std_logic := '0';    -- transmitter clock default value  
          TF_DEF: std_logic := '1';    -- transmitter frame default value  
          TD_DEF: std_logic := '0';    -- transmitter data default value  
          RK_DEF: std_logic := '0';    -- receiver clock default value  
          RF_DEF: std_logic := '1';    -- receiver frame default value  
          MSBDEF: boolean   := true);  -- MSB first if true  
  ...  
end SSC_Interface;
```

Schnittstellenbeschreibung

- ♦ Interfacesignale (ports) bilden die eigentlichen Kommunikationsschnittstellen zwischen einer Entwurfseinheit und der Einsatzumgebung durch Angaben von Namen, Typen und Signalflussrichtungen,

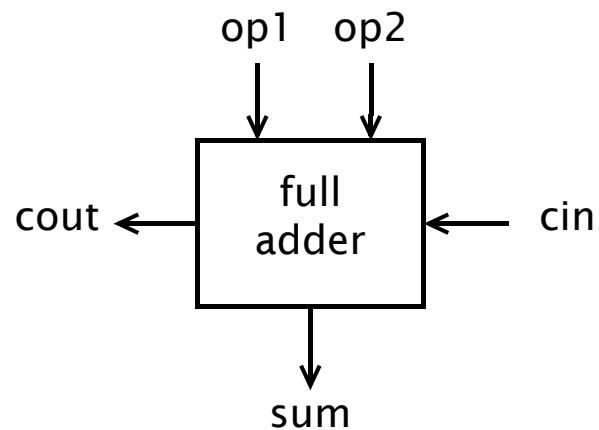
```
Interfacesignal =  
    Signalname ":" [ Modus ] Typbezeichner [ bus ] [ "!=" Ausdruck ] .  
  
Modus = in | out | inout | buffer .
```

- Der Modus bestimmt die Signalflussrichtung durch Ports:

Modus	Eigenschaft	Kommentar
IN	unidirektionaler Eingangsport	Der Datenfluß geht nur in das System hinein
OUT	unidirektionaler Ausgangsport	Der Datenfluß geht nur aus dem System heraus
INOUT	bidirektionaler Port	Der Datenfluß geht in beide Richtungen und kann mehrere Treiber haben. Standardeinstellung, wenn kein Modus angegeben ist.
BUFFER	bidirektionaler Port	Der Datenfluß geht in beide Richtungen, ein Port darf nur einen Treiber haben.

Schnittstellenbeschreibung

♦ 1-Bit-Volladdierer



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

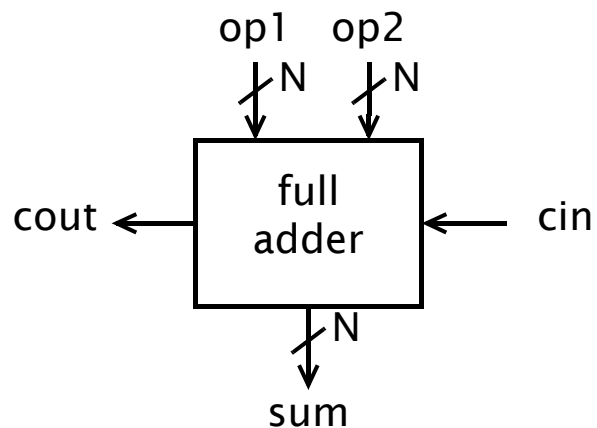
entity full_adder is

    port(cin:  in  std_logic; -- carry input
          op1:  in  std_logic; -- 1. operand
          op2:  in  std_logic; -- 2. operand
          sum:  out std_logic; -- result
          cout: out std_logic  -- carry output
    );

end full_adder;
```

Schnittstellenbeschreibung

♦ N-Bit-Volladdierer



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

entity full_adder_N is

generic(N: natural := 8); -- generische Konstante

port(cin:  in  std_logic;

      op1:  in  std_logic_vector(N-1 downto 0);

      op2:  in  std_logic_vector(N-1 downto 0);

      sum:  out std_logic_vector(N-1 downto 0);

      cout: out std_logic

);

end full_adder_N;
```

Architekturbeschreibung

- ♦ Die Architekturbeschreibung spezifiziert die Beziehung zwischen den Ein- und Ausgängen einer Entwurfseinheit und bestimmt die Struktur, den Datenfluß oder das Verhalten dieser Entwurfseinheit.

```
Architekturbeschreibung =  
  architecture Architekturbezeichner of Modulbezeichner is  
    [ Deklarationsbereich ]  
  begin  
    { [ label ":" ] parallele_Anweisung }  
  end [ architecture ] [ Architekturbezeichner ] ";" .
```

- Im Unterschied zu herkömmlichen, rein sequentiellen Programmiersprachen werden alle Anweisungen innerhalb des Anweisungsbereiches einer Architektur parallel ausgeführt.
- Die Reihenfolge, in der parallele Anweisungen innerhalb des Anweisungsbereiches einer Architektur notiert sind, ist ohne Bedeutung.

Architekturbeschreibung

- ◆ Parallele Anweisungen dienen zur strukturalen Modellierung oder zur Verhaltensmodellierung elektronischer Systeme

```
parallele_Anweisung =  
    einfache_Signalzuweisung  
    | bedingte_Signalzuweisung  
    | selektierte_Signalzuweisung  
    | Blockanweisung  
    | Prozeßanweisung  
    | Prozeduraufruf  
    | Komponenteninstanzierung  
    | Generierungsanweisung .
```

- Die parallele Ausführung dieser Anweisungen läßt sich auf einem Rechner (mit einem Prozessor) nur durch spezielle Simulationsalgorithmen nachbilden.
- Solche Algorithmen setzen parallele Anweisungen in Folgen von sequentiellen Anweisungen um, und zwar so, daß dieser Vorgang für einen Anwender nicht direkt sichtbar sind.

Einfache Signalzuweisung

- ◆ Einfache Signalzuweisungen sind die wichtigsten Elementaranweisungen in VHDL

```
einfache_Signalzuweisung =  
    Signalbezeichner "<=" [ Verzögerungsmodus ] Wellenform ";" .  
  
Verzögerungsmodus =  
    transport | [ reject Ausdruck ] inertial .  
  
Wellenform =  
    Ausdruck [ after Zeitangabe ] { "," Ausdruck after Zeitangabe } .
```

- Sie dient dazu, einem Signal-Objekt eine Wellenform bestehend aus Werte- und Zeitstempel-Paaren zuzuweisen.

```
constant RSTDEF: std_logic := '1';  
constant tpd:    time := 20 ns;  
...  
signal rst: std_logic := RSTDEF;  
signal dly: integer;  
...  
rst <= RSTDEF, not RSTDEF after 5*tpd  
  
dly <= 20 after 2 ns, 7 after 5 ns, -5 after 10 ns;
```

Bedingte Signalzuweisung

- ♦ Bedingte Signalzuweisungen (conditional signal assignments) basieren auf mehreren Zuweisungsalternativen, die jeweils durch Bedingungen gesteuert werden.

```
bedingte_Signalzuweisung =  
  Signalbezeichner "<=" [ guarded ] [ Verzögerungsmodus ]  
                        { Wellenform   when Bedingung else }  
                        Wellenform [ when Bedingung ] ";" .
```

- Das Verhalten einer bedingten Signalzuweisung ähnelt dem einer sequentiellen IF-ELSE-Anweisung, und entspricht technisch einem priorisierten, verketteten Decoder.
- In den einzelnen Bedingungen können unterschiedliche Signale oder Signalkombinationen abgefragt werden.

```
signal clk, hlt, OE: std_logic := '0';  
signal reg, dbus: std_logic_vector(N-1 downto 0);  
...  
dbus <= reg when OE='1' else (others => 'Z');  
  
clk  <= not clk after tpd/2 when hlt='1' else '0' ;
```

Selektierte Signalzuweisung

- ♦ Selektierte Signalzuweisungen (selected signal assignments) basieren auf einer Auswahl aus einer Reihe von gleichberechtigten Alternativen

```
selektierte_Signalzuweisung =  
  with Ausdruck select  
    Signalbezeichner "<=" [ transport ] Wellenform when Alternativen {  
      Wellenform when Alternativen } [ ", "  
      Wellenform when others ] ";" .  
  
Alternativen = Bedingung { "|" Bedingung } .
```

- Das Verhalten einer selektierten Signalzuweisung ähnelt dem einer sequentiellen CASE-Anweisung, und entspricht technisch einer Multiplexer-Struktur.
- Die selektierte Signalzuweisung wird von einem Ausdruck mit diskretem Wertebereich gesteuert.
- Wird der diskrete Wertebereich nicht durch alle aufgelistete Alternativen abgedeckt, so ist die OTHERS-Anweisung notwendig.

Architekturbeschreibung

♦ 1-Bit-Volladdierer

sel			tmp	
cin	op1	op2	cout	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

```
architecture behavioral of full_adder is
```

```
    signal sel: std_logic_vector(1 to 3);
```

```
    signal tmp: std_logic_vector(1 to 2);
```

```
begin
```

```
    sel <= cin & op1 & op2;
```

```
    with sel select
```

```
    tmp <= "00" when "000",  
           "01" when "001",  
           "01" when "010",  
           "10" when "011",  
           "01" when "100",  
           "10" when "101",  
           "10" when "110",  
           "11" when "111",  
           "--" when others;
```

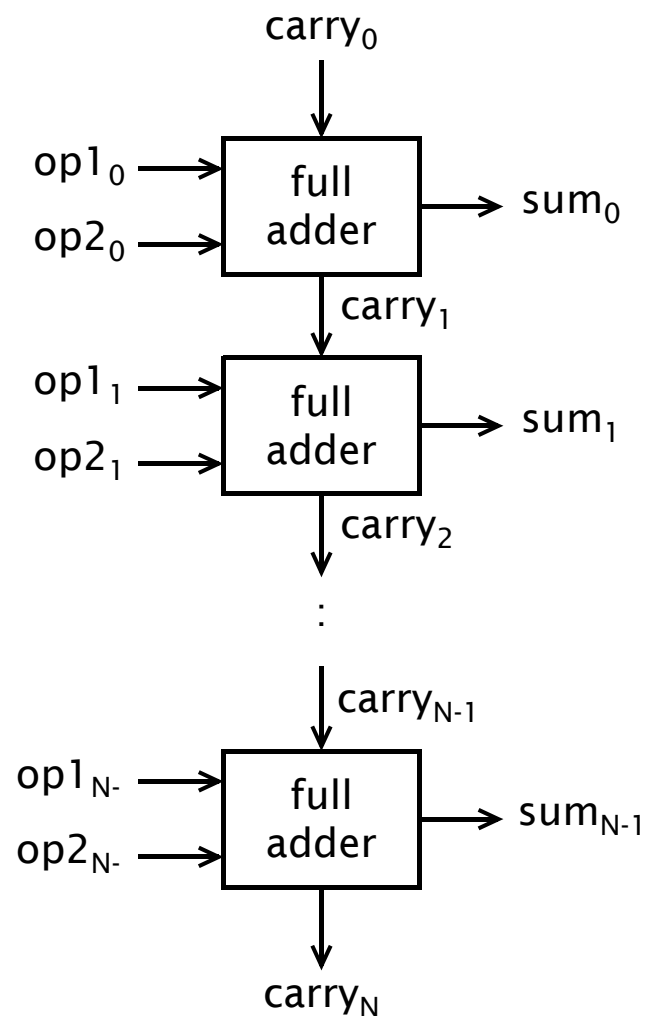
```
    sum  <= tmp(2);
```

```
    cout <= tmp(1);
```

```
end behavioral;
```


Architekturbeschreibung

◆ N-Bit-Volladdierer



```
architecture structure of full_adder_N is
    component full_adder is
        port(cin: in std_logic; -- carry input
              op1: in std_logic; -- 1. operand
              op2: in std_logic; -- 2. operand
              sum: out std_logic; -- result
              cout: out std_logic); -- carry output
    end component;
    signal carry: std_logic_vector(N downto 0);
begin

    carry(0) <= cin;
    e1: for i in 0 to N-1 generate
        u1: full_adder
        port map(
            cin => carry(i),
            op1 => op1(i),
            op2 => op2(i),
            sum => sum(i),
            cout => carry(i+1));
    end generate;
    cout <= carry(N);

end structure;
```

Architekturbeschreibung

- ♦ Design-Regeln zur Modellierung von Schaltnetzen
 - einfache Schaltnetze (z.B.: Multiplexer, Decoder, Basiszellen von Schaltketten)
 - Beschreibung in der Digitaltechnik durch Funktionstabellen
 - tabellarische Beschreibung in VHDL mit bedingten oder selektierten Signalzuweisungen: effizient, kompakt, übersichtlich, eindeutig, leicht modifizierbar, technologieunabhängig
 - algorithmische (Verhaltens-)Beschreibung mit Prozessen und sequentiellen Anweisungen kann manchmal mehrdeutig, unübersichtlich sein.
 - Beschreibung mit booleschen Gleichungen ist meistens unnötig, und stellt bereits das Ergebnis einer Synthese oder Minimierung dar.
 - zusammengesetzte Schaltnetze (Schaltketten, z.B.: arithmetische Schaltketten wie N-Bit-Addierer, N-Bit-Vergleicher)
 - Beschreibung in der Digitaltechnik durch hierarchische Strukturen
 - Beschreibung in VHDL als strukturelle Funktionsbeschreibungen mit Hilfe generischer Komponenten, meistens wird die Vektorlänge als skalierbare Größe gewählt.
 - Instanziierung von Komponenten (als Basiszelle)

Entwurfsmuster

- ♦ 1-aus-4-Multiplexer mit Enable
 - Funktionstabelle

en	s ₁	s ₀	y
0	-	-	0
1	0	0	x ₀
1	0	1	x ₁
1	1	0	x ₂
1	1	1	x ₃

```
signal x: std_logic_vector(0 to 3);  
signal y: std_logic;  
signal t: std_logic;  
signal en: std_logic;
```

```
signal sel: std_logic_vector(1 to 2);  
...  
with sel select  
    t <= x(0) when "00",  
        x(1) when "01",  
        x(2) when "10",  
        x(3) when others;
```

```
signal adr: integer range 0 to 3;  
...  
with adr select  
    t <= x(0) when 0,  
        x(1) when 1,  
        x(2) when 2,  
        x(3) when 3;
```

```
type TState is (S0, S1, S2, S3);  
signal state: TState;  
...  
with state select  
    t <= x(0) when S0,  
        x(1) when S1,  
        x(2) when S2,  
        x(3) when S3;
```

```
y <= t when en='1' else '0';
```

Entwurfsmuster

- ◆ 1-aus-4-Dekoder mit Enable
 - Funktionstabelle

en	s ₁	s ₀	y ₀	y ₁	y ₂	y ₃
0	-	-	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

```
signal y: std_logic_vector(0 to 3);
signal t: std_logic_vector(0 to 3);
signal en: std_logic;

signal sel: std_logic_vector(1 to 2);
...
with sel select
    t <= "1000" when "00",
        "0100" when "01",
        "0010" when "10",
        "0001" when others;

signal adr: integer range 0 to 3;
...
with adr select
    t <= "1000" when 0,
        "0100" when 1,
        "0010" when 2,
        "0001" when 3;

type TState is (S0, S1, S2, S3);
signal state: TState;
...
with state select
    t <= "1000" when S0,
        "0100" when S1,
        "0010" when S2,
        "0001" when S3;

y <= t when en='1' else (others => '0');
```

Prozeßanweisung

- ♦ Prozesse sind nebenläufige (parallele) Anweisungen mit einem sequentiellen Anweisungsbereich.
 - Prozesse dienen zur algorithmischen Verhaltensmodellierung von synchronen und asynchronen Schaltwerken, von Schaltnetzen und zur Beschreibung von Testumgebungen für die Simulation.

```
Prozeßanweisung =  
  process [ "(" Bezeichnerliste ")" ] is  
    [ Deklarationsbereich ]  
  begin  
    sequentielle_Anweisungen  
  end process ";" .
```

- Optionale Sensitivitätsliste gibt Signale aus der Umgebung an, über die ein Prozeß zur Ausführung „angestoßen“ wird.
- Ist die Sensitivitätsliste nicht vorhanden, so ist mindestens eine WAIT-Anweisung im Prozeßrumpf erforderlich.

Prozeßanweisung

- Prozesse kommunizieren mit Hilfe von (Handshake-)Signalen miteinander
=> sorgfältige Planung von Handshake-Mechanismen erforderlich
- Prozesse dürfen nur in Architekturbeschreibungen stehen und können nicht ineinander verschachtelt werden.
- Prozesse bilden eine statische Gruppe und können nicht dynamisch erzeugt und gelöscht werden.
- Ein Prozeß kann (während der Simulation) einen von zwei möglichen Zuständen annehmen:
 - entweder ist der Prozeß aktiv und seine Anweisungen werden gerade abgearbeitet
 - oder der Prozeß ist suspendiert und wartet auf ein für ihn relevantes Ereignis

Sequentielle Anweisungen

- ♦ Sequentielle Anweisungen werden zum strukturierten Programmieren in VHDL auf der Ebene der algorithmischen Verhaltensbeschreibung eingesetzt.

```
sequentielle_Anweisungen =  
  { [ label ":" ] sequentielle_Anweisung } .  
  
sequentielle_Anweisung =  
  Signalzuweisung      | Variablenzuweisung      | Prozeduraufruf  
  | IF_Anweisung        | CASE_Anweisung          | Schleifenanweisung  
  | WAIT_Anweisung       | NULL_Anweisung          | Assertion_Anweisung  
  | NEXT_Anweisung       | EXIT_Anweisung          | RETURN_Anweisung .
```

- Anweisungen in einem Prozeßrumpf werden im allgemeinen nacheinander, also in der Reihenfolge des Aufschreibens ausgeführt.
- Diese Reihenfolge im Ablauf kann mit Steuerflußanweisungen beeinflußt werden:
 - Verzweigungen im Ablauf mit IF-, CASE- oder Schleifenanweisungen,
 - vorzeitiger Abbruch mit RETURN-, EXIT- oder NEXT-Anweisungen,
 - Anhalten mit WAIT-Anweisung bis zum Auftreten eines Ereignisses

IF-Anweisung

- ♦ Die IF-Anweisung ist eine Steuerflußanweisung, die bedingte Verzweigungen in sequentiellen Anweisungsbereichen ermöglicht.

```
IF_Anweisung =  
    if Bedingung then  
        sequentielle_Anweisungen  
    { elsif Bedingung then  
        sequentielle_Anweisungen }  
    [ else  
        sequentielle_Anweisungen ]  
    end if ";" .
```

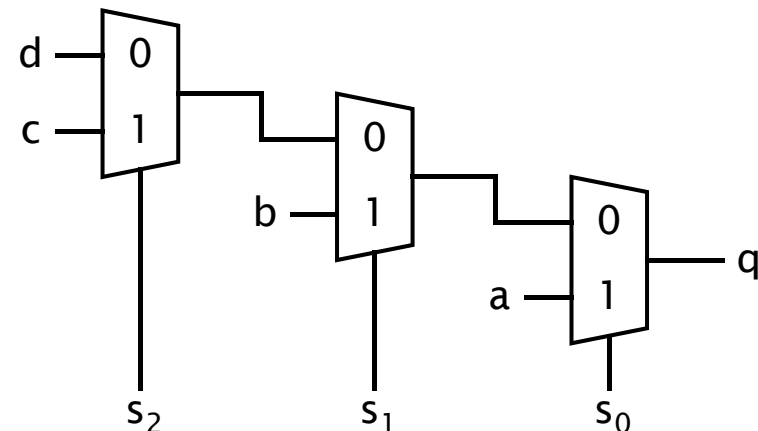
- Mit einer IF-Anweisung kann das Verhalten einer bedingten Signalzuweisung im sequentiellen Bereich nachgebildet werden.
- IF-Anweisungen können ineinander verschachtelt werden.
- Steht am Ende der ELSIF-Kette ein abschließender ELSE-Block, so werden die darin enthaltenen Anweisungen nur dann ausgeführt, wenn weder die Bedingung der (ersten) IF-Anweisung noch aller folgenden ELSIF-Anweisungen erfüllt waren.

Prozeß- mit IF-Anweisungen

♦ Verhalten einer Prioritätskette

- Aus der Reihenfolge in der IF-ELSIF-ELSE-Struktur ergibt eine Prioritätskette mit fester Priorisierungsfolge:
 - Jeder IF- bzw. ELSIF-Block beginnt mit einer Bedingung.
 - Nur wenn diese erfüllt ist, werden die dazugehörigen Anweisungen auch ausgeführt.
 - Ansonsten wird die Bedingung des nächsten ELSIF-Blocks ausgewertet.
- Funktionstabelle und Struktur eines Prioritätsdecoders:

s_0	s_1	s_2	q
1	–	–	a
0	1	–	b
0	0	1	c
0	0	0	d



Prozeß- mit IF-Anweisungen

- ♦ Modellierung eines Prioritätsdecoders mit Hilfe eines Prozesses und einer IF-Anweisung

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

```
entity decoder is
```

```
    port(a: in  std_logic;  
          b: in  std_logic;  
          c: in  std_logic;  
          d: in  std_logic;  
          s: in  std_logic_vector(0 to 2);  
          q: out std_logic);
```

```
end decoder;
```

```
architecture behavioral of decoder is  
begin
```

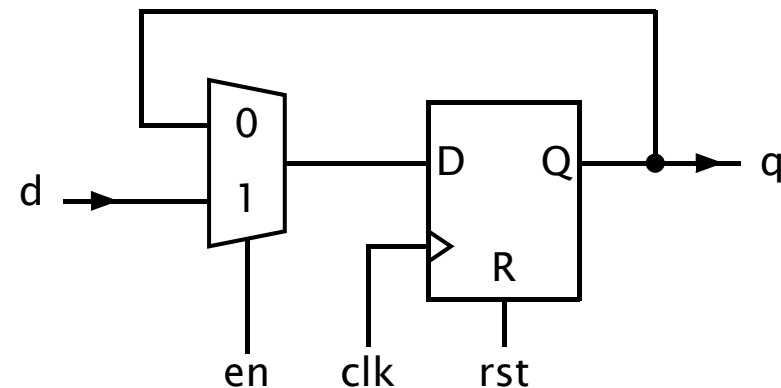
```
    process (s, a, b, c, d) is  
    begin  
        if s(0)='1' then  
            q <= a;  
        elsif s(1)='1' then  
            q <= b;  
        elsif s(2)='1' then  
            q <= c;  
        else  
            q <= d;  
        end if;  
    end process;
```

```
end behavioral;
```

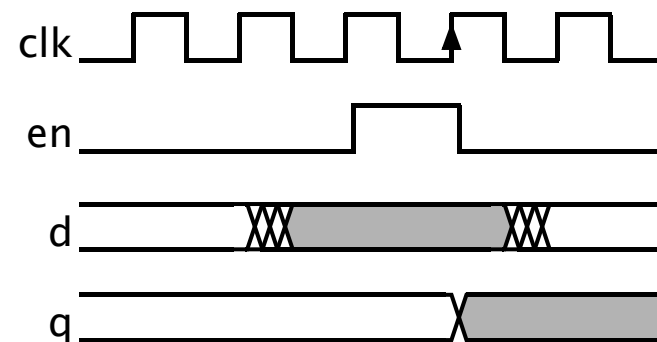
Prozeß- mit IF-Anweisungen

- ♦ Verhalten eines flankengesteuerten Flipflops mit einem asynchronen Rücksetzsignal und einem Enable-Signal
 - Funktionstabelle und Struktur:

rst	clk	en	d	q^{t+1}
0	-	-	-	0
1	↑	0	-	q^t
1	↑	1	0	0
1	↑	1	1	1



- Impulsdiagramm:



Prozeß- mit IF-Anweisungen

- ♦ Modellierung eines flankengesteuerten Flipflops mit einem asynchronen Rücksetzsignal und einem Enable-Signal mit Hilfe eines Prozesses mit IF-Anweisungen

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

entity flipflop is

    generic(RSTDEF: std_logic := '0');
    port(rst: in std_logic;
         clk: in std_logic;
         en: in std_logic;
         d: in std_logic;
         q: out std_logic);

end flipflop;
```

```
architecture behavioral of flipflop is
    signal dff: std_logic;
begin

    q <= dff;

    process (rst, clk) is
    begin
        if rst=RSTDEF then
            dff <= '0';
        elsif rising_edge(clk) then
            if en='1' then
                dff <= d;
            end if;
        end if;
    end process;

end behavioral;
```

Prozeß- mit IF-Anweisungen

♦ Syntheseregel:

- ein Speicherelement wird u.a. immer dann erzeugt, wenn in einem Prozeß ein Signal nur in einigen, aber nicht in allen Zweigen einer IF- oder einer CASE-Anweisung einen Wert zugewiesen bekommt.
- Es hängt vom Beschreibungsstil ab, ob ein flankengesteuertes Flipflop oder ein Latch generiert wird.

```
entity element is
  port(a, b: in std_logic;
        c: out std_logic);
end element;
```

```
process (a, b)
begin
  if a='1' then
    c <= b;
  else
    c <= 'Z';
  end if;
end process;
```

```
process (a, b)
begin
  if a='1' then
    c <= b;
  else
    c <= 'Z';
  end if;
end process;
```

```
process (a)
begin
  if a='1' then
    c <= b;
  else
    c <= 'Z';
  end if;
end process;
```

Design-Muster

- ◆ Design-Muster einer synthesesegerechten Schnittstellenbeschreibung eines parametrisierbaren Registers

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

entity std_register is

    generic(RSTDEF: std_logic := '1';
            LENDEF: natural := 8);

    port(rst:    in  std_logic;    -- reset, RSTDEF active
          clk:   in  std_logic;    -- clock, rising edge active
          swrst: in  std_logic;    -- software reset, RSTDEF active
          en:    in  std_logic;    -- enable, high active

          -- ggf. weitere Steuersignale

          din:   in  std_logic_vector(LENDEF-1 downto 0); -- data input
          q:     out std_logic_vector(LENDEF-1 downto 0)); -- data output

end std_register;
```

Design-Muster

- ◆ Design-Muster einer synthesesegerechten Architekturbeschreibung eines parametrisierbaren Registers

```
architecture behavioral of std_register is
    signal dff: std_logic_vector(LENDEF-1 downto 0);
begin

    q <= dff;

    process (rst, clk) is
    begin
        if rst=RSTDEF then
            dff <= (others => '0');
        elsif rising_edge(clk) then
            if en='1' then
                -- ggf. mit Abfrage weiterer Steuersignale
                dff <= din;
            end if;
            if swrst=RSTDEF then
                dff <= (others => '0');
            end if;
        end if;
    end process;

end behavioral;
```

Design-Muster

- ♦ synchroner Modulo-N-Zähler mit Enable
 - Funktionstabelle

rst	clk	en	cnt ^t	cnt ^{t+1}
0	-	-	-	0
1	↑	0	-	cnt ^t
1	↑	1	=N-1	0
1	↑	1	<N-1	cnt ^t +1

```
constant N: natural := 12;
signal cnt: integer range 0 to N-1;
...
process (rst, clk) begin
    if rst='0' then
        cnt <= 0;
    elsif rising_edge(clk) then
        if en='1' then
            if cnt=N-1 then
                cnt <= 0;
            else
                cnt <= cnt + 1;
            end if;
        end if;
    end if;
end process;
```

```
use ieee.std_logic_unsigned.all;
signal cnt: std_logic_vector(0 to 3);
...
process (rst, clk) begin
    if rst='0' then
        cnt <= (others => '0');
    elsif rising_edge(clk) then
        if en='1' then
            if cnt=N-1 then
                cnt <= (others => '0');
            else
                cnt <= cnt + 1;
            end if;
        end if;
    end if;
end process;
```


Design-Muster

♦ synchroner Frequenzteiler mit Enable

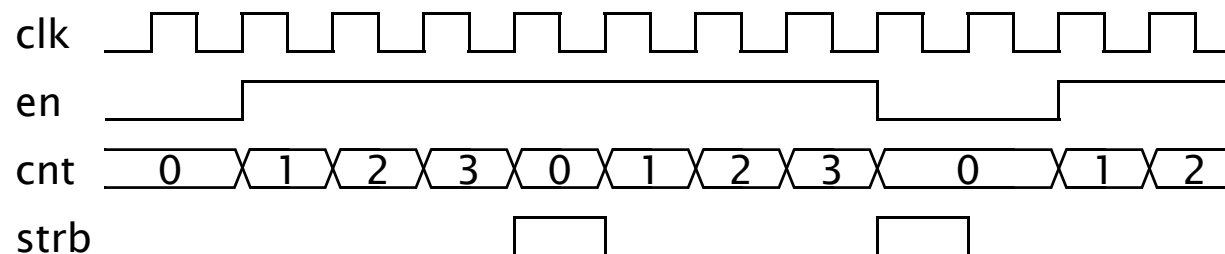
– Funktionstabelle

rst	clk	en	cnt ^t	cnt ^{t+1}	strb
0	-	-	-	0	0
1	↑	0	-	cnt ^t	0
1	↑	1	=N-1	0	1
1	↑	1	<N-1	cnt ^t +1	0

```
constant N: natural := 4;
signal cnt: integer range 0 to N-1;
signal strb: std_logic;
```

```
process (rst, clk) begin
    if rst='0' then
        cnt <= 0;
        strb <= '0';
    elsif rising_edge(clk) then
        strb <= '0';
        if en='1' then
            if cnt=N-1 then
                cnt <= 0;
                strb <= '1';
            else
                cnt <= cnt + 1;
            end if;
        end if;
    end if;
end process;
```

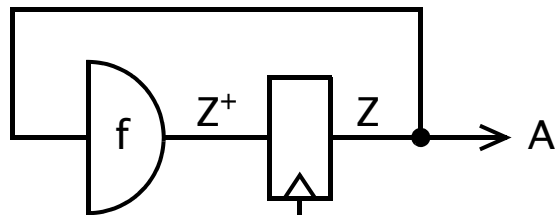
– Impulsdiagramm



Schaltwerke in VHDL

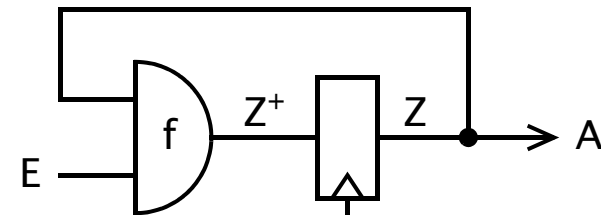
♦ Automatenmodelle

Autonomer Automat



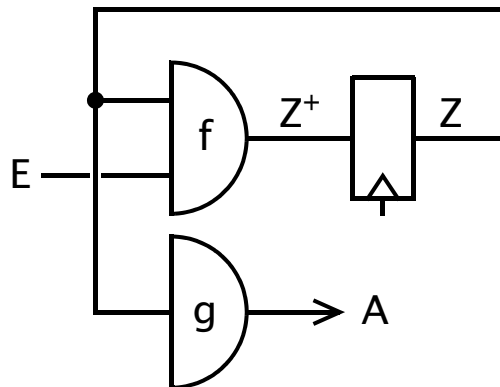
$$Z^+ = f(Z), \quad A = Z$$

Medwedjew-Automat



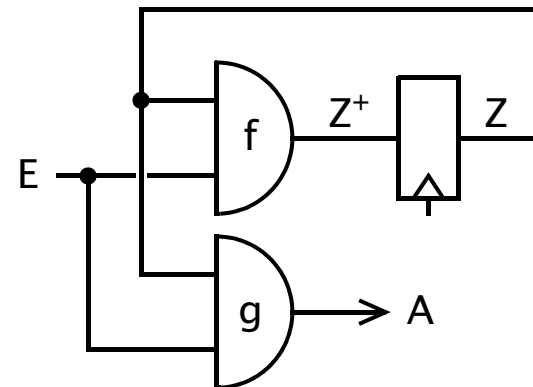
$$Z^+ = f(Z, E), \quad A = Z$$

Moore-Automat



$$Z^+ = f(Z, E), \quad A = g(Z)$$

Mealy-Automat



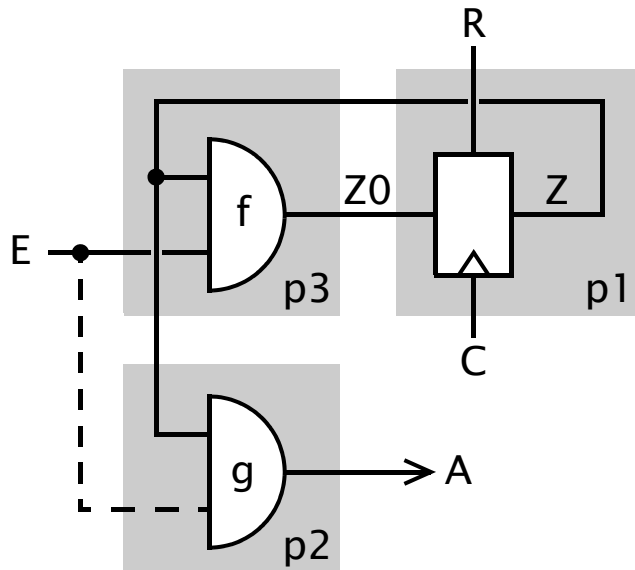
$$Z^+ = f(Z, E), \quad A = g(Z, E)$$

Schaltwerke in VHDL

♦ Mealy- und Moore-Automaten:

– 3-Prozeß-Methode:

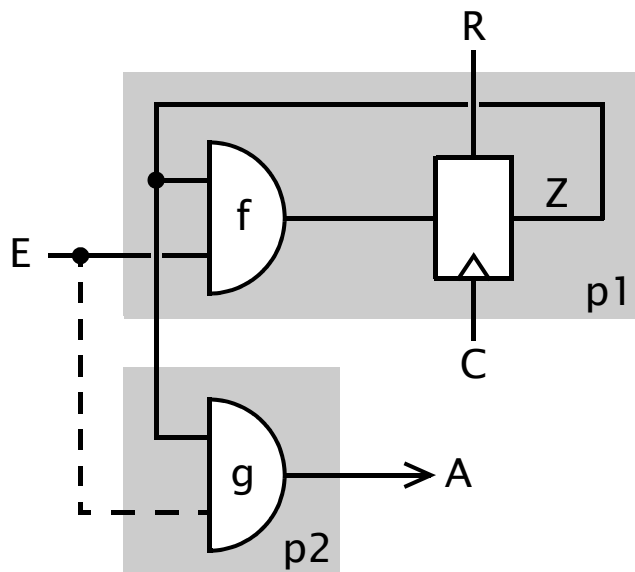
- ein „getakteter“ Prozeß beschreibt das Verhalten des Zustandsregisters
- zwei „ungetaktete“ Prozesse beschreiben das Übergangsschaltnetz und das Ausgangsschaltnetz



```
signal Z, Z0: Zustand;
p1: process (R, C) is -- Zustandsregister
    ...
    Z <= Z0;
    ...
end process;
p2: process (E, Z) is -- Ausgangsschaltnetz
    ...
    if Z=... and E=... then
        A <= ...;
    end if;
    ...
end process;
p3: process (E, Z) is -- Übergangsschaltnetz
    ...
    if Z=... and E=... then
        Z0 <= ...;
    end if;
    ...
end process;
```

Schaltwerke in VHDL

- ◆ Beschreibung von Mealy- und Moore-Automaten:
 - 2-Prozeß-Methode, 1. Variante:
 - ein „getakteter“ Prozeß vereinigt in sich die Beschreibung des Zustandsregisters und die des Übergangsschaltnetzes
 - ein „ungetakteter“ Prozeß beschreibt das Ausgangsschaltnetz



```
signal Z: Zustand;

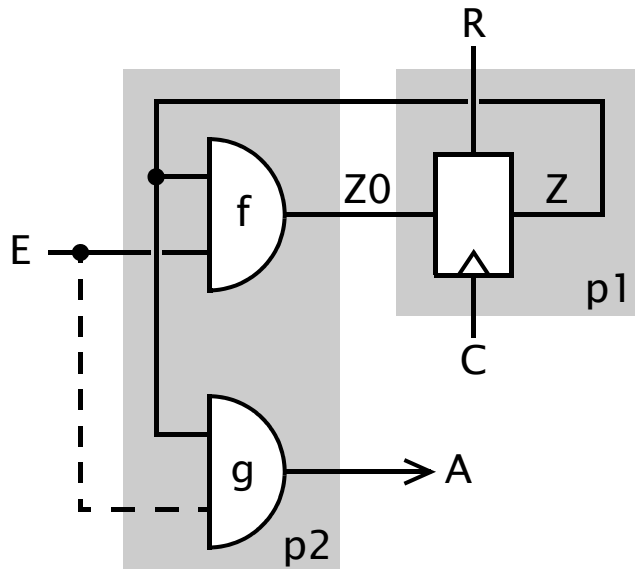
p1: process (R, C) is -- Zustandsregister
    ...
    -- Übergangsschaltnetz

    if Z=... and E=... then
        Z <= ...;
    end if;
    ...
end process;

p2: process (E, Z) is -- Ausgangsschaltnetz
    ...
    if Z=... and E=... then
        A <= ...;
    end if;
    ...
end process;
```

Schaltwerke in VHDL

- ◆ Beschreibung von Mealy- und Moore-Automaten:
 - 2-Prozeß-Methode, 2. Variante:
 - ein „getakteter“ Prozeß beschreibt das Verhalten des Zustandsregisters
 - ein „ungetakteter“ Prozeß vereinigt in sich die Beschreibungen des Übergangsschaltnetzes und des Ausgangsschaltnetzes



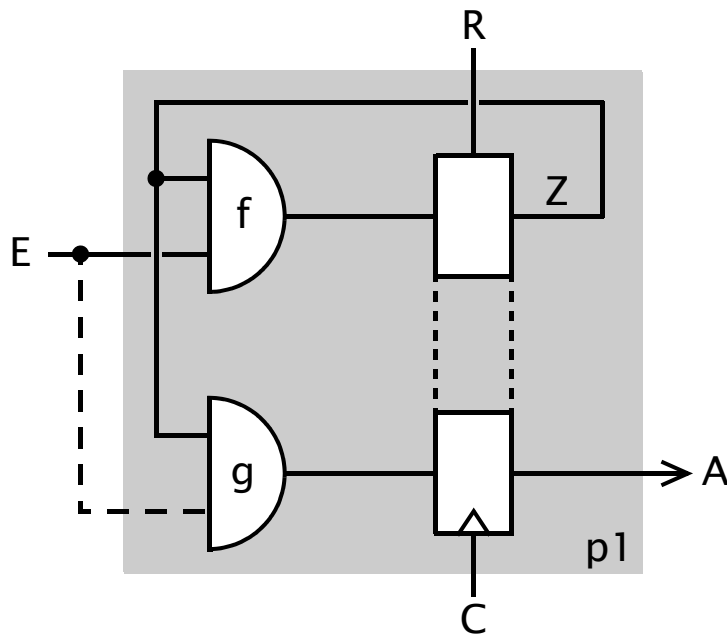
```
signal Z, Z0: Zustand;

p1: process (R, C) is -- Zustandsregister
    ...
    Z <= Z0;
    ...
end process;

p2: process (E, Z) is -- Übergangsschaltnetz
    ... -- Ausgangsschaltnetz
    if Z=... and E=... then
        Z0 <= ...;
        A <= ...;
    end if;
    ...
end process;
```

Schaltwerke in VHDL

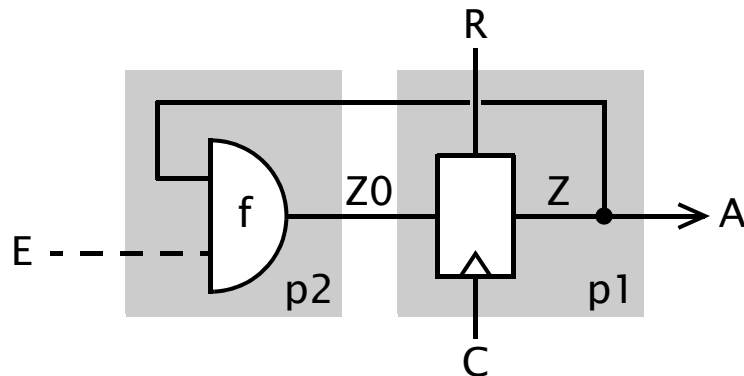
- ◆ Beschreibung von Mealy- und Moore-Automaten:
 - 1-Prozeß-Methode:
 - ein „getakteter“ Prozeß vereinigt in sich die Beschreibungen des Zustandsregisters, des Übergangsschaltnetzes und des Ausgangsschaltnetzes.
 - Ausgangsflipflops beeinflussen das Verhalten des Schaltwerks



```
signal Z: Zustand;  
  
-- Zustandsregister  
-- Übergangsschaltnetz  
-- Ausgangsschaltnetz + Flipflops  
  
p1: process (R, C) is  
    ...  
    if Z=... and E=... then  
        Z <= ...;  
        A <= ...;  
    end if;  
    ...  
end process;
```

Schaltwerke in VHDL

- ◆ Beschreibung von Autonomen und Medwedjew-Automaten:
 - 2-Prozeß-Methode:
 - ein „getakteter“ Prozeß beschreibt das Verhalten des Zustandsregisters
 - ein „ungetakteter“ Prozeß beschreibt das Übergangsschaltnetz



```
signal Z, Z0: Zustand;

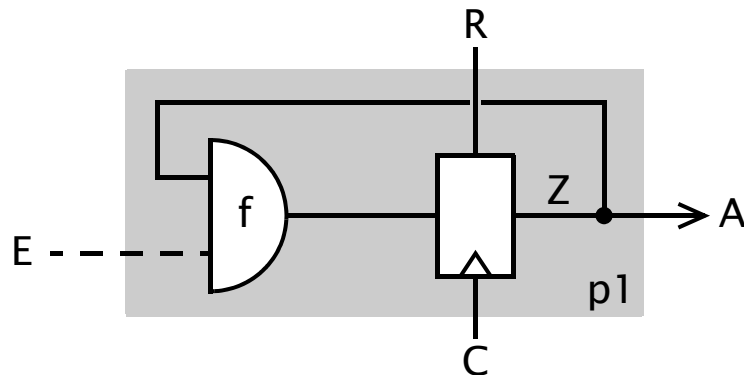
A <= Z;           -- Ausgangsschaltnetz

p1: process (R, C) is -- Zustandsregister
    ...
    Z <= Z0;
    ...
end process;

p2: process (E, Z) is -- Übergangsschaltnetz
    ...
    if Z=... and E=... then
        Z0 <= ...;
    end if;
    ...
end process;
```

Schaltwerke in VHDL

- ◆ Beschreibung von Autonomen und Medwedjew-Automaten:
 - 1-Prozeß-Methode:
 - ein „getakteter“ Prozeß vereinigt in sich die Beschreibungen des Zustandsregisters und des Übergangsschaltnetzes



```
signal Z: Zustand;

A <= Z; -- Ausgangsschaltnetz

-- Zustandsregister
-- Übergangsschaltnetz
p1: process (R, C) is
    ...
    if Z=... and E=... then
        Z <= ...;
    end if;
    ...
end process;
```


CASE-Anweisung

- ♦ Die CASE-Anweisung ist eine Steuerflußanweisung, die Mehrfachverzweigungen im Ablauf ermöglicht
 - Eine CASE-Anweisung besteht aus einem Ausdruck (dem Selektor) und einer Liste von Auswahlanweisungen mit den dazugehörigen sequentiellen Anweisungen.

```
CASE_Anweisung =  
    case Ausdruck is  
        sequentielle_Anweisungen  
    { when Auswahl "=>"  
        sequentielle_Anweisungen }  
    [ when others "=>"  
        sequentielle_Anweisungen ]  
    end case ";" .
```

```
Auswahl = Ausdruck ( ( to | downto ) Ausdruck | { "|" Ausdruck } ) .
```

- Der Typ des Selektors muß entweder ein diskreter Typ oder ein Vektor mit Elementen vom Typ character sein.
- Mit einer CASE-Anweisung kann das Verhalten einer selektierten Signalzuweisung im sequentiellen Bereich nachgebildet werden.

CASE-Anweisung

♦ Laufzeitverhalten

- Während der Laufzeit wählt die CASE-Anweisung zur Ausführung diejenigen sequentiellen Anweisungen aus, die anhand der Auswahlanweisungen zum momentanen Wert des Selektors passen.
- Jeder Wert aus den Auswahlanweisungen darf zu höchstens einem Bereich mit sequentiellen Anweisungen gehören
- Prozeß- mit einer CASE-Anweisung:

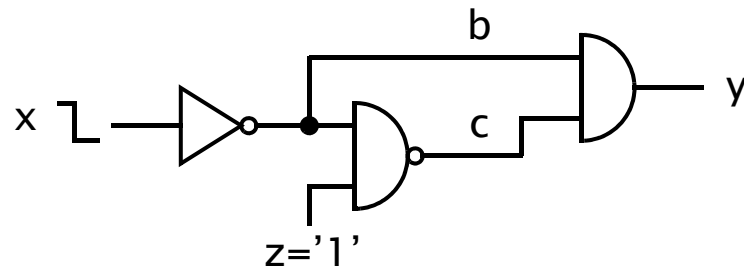
```
-- 1-aus-4-Dekoder
signal sel: std_logic_vector(1 to 2);
signal dec: std_logic_vector(1 to 4);

process (sel) is
BEGIN
    case sel is
        when "00"    => dec <= "0001";
        when "01"    => dec <= "0010";
        when "10"    => dec <= "0100";
        when "11"    => dec <= "1000";
        when others => dec <= "----";
    end case;
end process;
```

Simulationsalgorithmus

♦ Delta-Schritte

```
entity elem is
  port(x, z: in bit;
        y: out bit);
end elem;
```



```
architecture test of elem is
  signal b, c: bit;
begin
  c <= not(z and b);
  y <= c and b;
  b <= not(x);
end test;
```

Zeit [ns]	x	b	c	y
9	1	0	1	0

Simulationsalgorithmus

♦ Unterschied zwischen Variablen und Signalen

```
entity test is
end test;
```

```
architecture verhalten of test is
    signal a: bit := '0';
    signal c: integer := 1;
    signal m: integer := 4;
```

```
begin
```

```
    process(a)
        variable b: integer := 2;
        variable d: integer := 3;
    begin
```

```
(1)    m <= b;
(2)    b := b + d;
(3)    c <= b;
(4)    d := c;
    end process;
```

```
(5)    a <= not(a) after 2 ns;
end verhalten;
```

	a _{SIG}	b _{VAR}	c _{SIG}	d _{VAR}	m _{SIG}
(5)					
(1)					
(2)					
(3)					
(4)					
(5)					
(1)					
(2)					
(3)					
(4)					
(5)					

Hardware/Software-Codesign

```
#define STRT 0x01
#define EOP  0x04

#define ADR 0xFF00
```

```
typedef struct {
    int x, y;
    char scr;
} ass_reg;
```

```
int res;
...
volatile ass_reg *ass = (ass_reg *)ADR;

ass->x = ...;      Parameterübergabe
ass->y = ...;

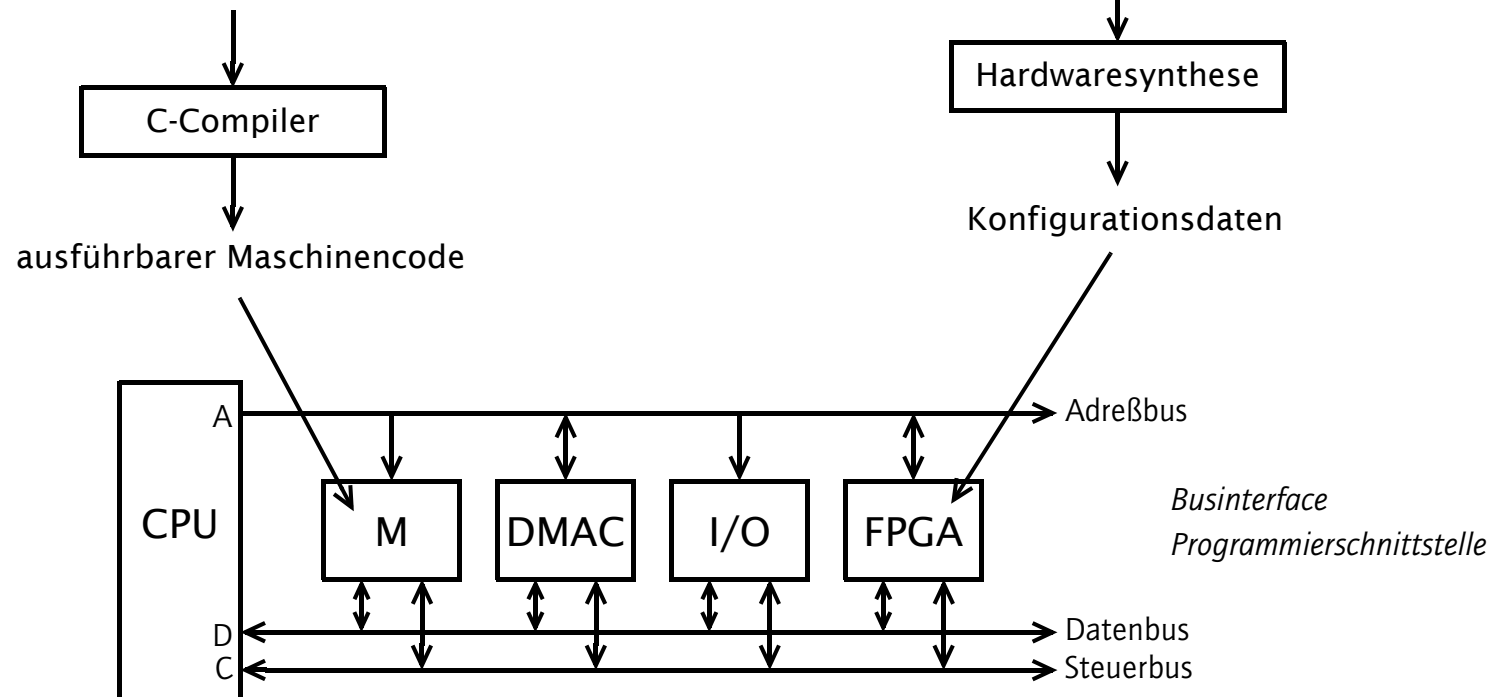
ass->scr = STRT;   Ausführung starten

while (!(ass->scr & EOP)) ;   Synchronisation

res = ass->x;      Ergebnis übernehmen
```

```
int x, y;

void gcd (void) {
    while (x != y)
        if (x < y)
            y = y - x;
        else
            x = x - y;
}
```



Hardware/Software-Codesign

- ◆ Datentypen und Speicherplatzbedarf ermitteln
 - abhängig von Programmiersprache, Compiler, CPU im Zielsystem
 - für C in limits.h definiert

```
#define SCHAR_MAX    127
#define SCHAR_MIN    (-128)
#define UCHAR_MAX    255
```

```
#define INT_MAX       0x7FFF
#define INT_MIN       ((int)0x8000)
#define UINT_MAX      0xFFFFU
```

- mit sizeof() den Speicherplatzbedarf in Bytes bestimmen
- ◆ Unterprogrammaufruf durch Handshaking nachbilden

1. Argumente auf den Stack legen
2. CALL Unterprogramm
das Unterprogramm übernimmt die Kontrolle über die CPU
3. Unterprogramm verarbeitet Argumente
4. Resultate auf den Stack legen
5. RETURN
die aufrufende Umgebung hat die Kontrolle über die CPU zurück erlangt

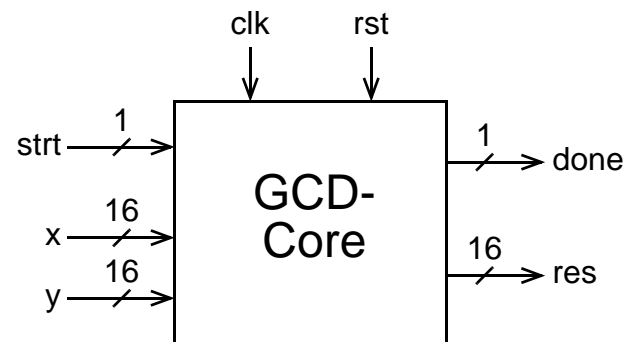
1. Argumente in Registern ablegen
(über die Programmierschnittstelle)
2. Start-Bit/-Kommando setzen
die CPU und der Co-Prozessor arbeiten echt parallel
3. CPU wartet, bis Resultate vorliegen
z.B. durch Interrupt oder Abfrage eines Stausbits (Ready-/Done-Flag)
4. Resultate aus den Registern

Hardware/Software-Codesign

◆ Definition der Signale in der Schnittstelle

```
int x, y; // unsigned
// sizeof(int) = 2
```

```
void gcd (void) {
    while (x != y)
        if (x < y)
            y = y - x;
        else
            x = x - y;
}
```



elektrische Eigenschaften (Pegel, Dauer, Hazard-Freiheit, aktive Flanke) der Steuersignale:

- rst mit Low
- strt und done mit High
- clk mit steigender Flanke

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

```
entity gcd is
    generic(RSTDEF: std_logic := '0');
    port(rst:      in  std_logic;
         clk:      in  std_logic;
         strt:     in  std_logic;
         done:     out std_logic;
         x:        in  unsigned(15 downto 0);
         y:        in  unsigned(15 downto 0);
         res:      out unsigned(15 downto 0));
end gcd;
```

Hardware/Software-Codesign

- ◆ 1. Version des GCD-Prozessors
 - Modellierung des Handshakings an einem ereignisgesteuerten Modell

```
architecture behaviour of gcd is
begin
```

```
    process is
```

```
        variable reg_x: unsigned(x'range);
```

```
        variable reg_y: unsigned(x'range);
```

```
    begin
```

```
    end process;
```

```
end behaviour;
```

```
wait until strt='1';
```

```
done  <= '0';
```

```
reg_x := x;
```

```
reg_y := y;
```

```
while reg_x /= reg_y loop
```

```
    if reg_x<reg_y then
```

```
        reg_y := reg_y - reg_x;
```

```
    else
```

```
        reg_x := reg_x - reg_y;
```

```
    end if;
```

```
end loop;
```

```
wait for 30 ns;
```

```
done <= '1';
```

```
res  <= reg_x;
```


Hardware/Software-Codesign

- ◆ 2. Version des GCD-Prozessors
 - Einführung eines Takt- und Rücksetzsignals

```
architecture behaviour of gcd is
  type TState is (S0, S1);
  signal state: TState;
begin
```

```
  process (rst, clk) is
    variable reg_x: unsigned(x'range);
    variable reg_y: unsigned(x'range);
```

```
  begin
    if rst=RSTDEF then
      reg_x := (others => '0');
      reg_y := (others => '0');
      res   <= (others => '0');
      done  <= '0';
      state <= S0;
    elsif rising_edge(clk) then
```

```
      end if;
    end process;
```

```
end behaviour;
```

```
    case state is
      when S0 =>
        done <= '0';
        if strt='1' then
          reg_x := x;
          reg_y := y;
          state <= S1;
        end if;
      when S1 =>
        while reg_x /= reg_y loop
          if reg_x < reg_y then
            reg_y := reg_y - reg_x;
          else
            reg_x := reg_x - reg_y;
          end if;
        end loop;
        state <= S0;
        done  <= '1';
        res   <= reg_x;
      end case;
```

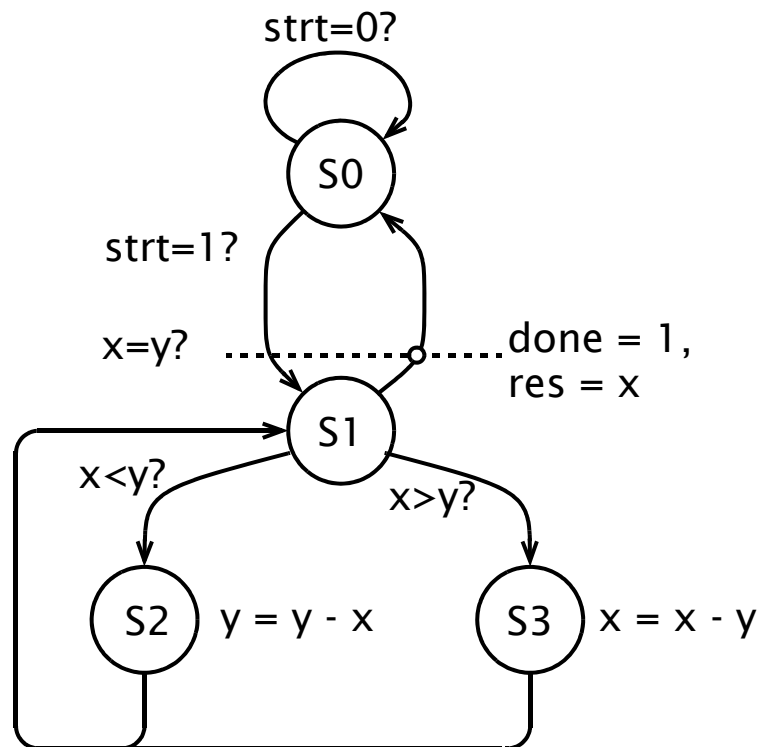
Hardware/Software-Codesign

◆ 3. Version des GCD-Prozessors

- Auflösung von Schleifen
- Übergang auf eine Moore-Zustandsmaschine und Signale

```
int x, y; // unsigned
// sizeof(int) = 2
```



```
void gcd (void) {
    while (x != y)
        if (x < y)
            y = y - x;
        else
            x = x - y;
}
```



◆ 3. Version des GCD-Prozessors

```
architecture behaviour of gcd is
  type TState is (S0, S1, S2, S3);
  signal state: TState;
  signal reg_x: unsigned(x'range);
  signal reg_y: unsigned(x'range);
begin

  res <= reg_x;

  process (rst, clk) begin
    if rst=RSTDEF then
      state <= S0;
      done <= '0';
      reg_x <= (others => '0');
      reg_y <= (others => '0');
    elsif rising_edge(clk) then
      
      
      end if;
    end process;
  end behaviour;
```

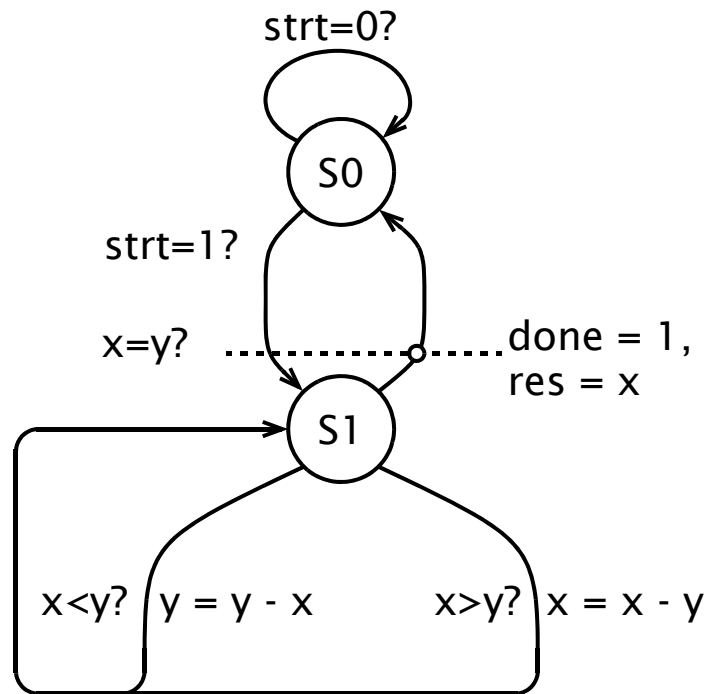
```
case state is
  when S0 =>
    done <= '0';
    if strt='1' then
      reg_x <= x;
      reg_y <= y;
      state <= S1;
    end if;
  when S1 =>
    if reg_x=reg_y then
      done <= '1';
      state <= S0;
    elsif reg_x<reg_y then
      state <= S2;
    else
      state <= S3;
    end if;
  when S2 =>
    reg_y <= reg_y - reg_x;
    state <= S1;
  when S3 =>
    reg_x <= reg_x - reg_y;
    state <= S1;
end case;
```

Hardware/Software-Codesign

- ◆ 4. Version des GCD-Prozessors
 - Realisierung mit einer Mealy-Zustandsmaschine

```
int x, y; // unsigned
// sizeof(int) = 2


void gcd (void) {
    while (x != y)
        if (x < y)
            y = y - x;
        else
            x = x - y;
}
```



◆ 4. Version des GCD-Prozessors

```
architecture behaviour of gcd is
  type TState is (S0, S1, S2, S3);
  signal state: TState;
  signal reg_x: unsigned(x'range);
  signal reg_y: unsigned(x'range);
begin

  res <= reg_x;

  process (rst, clk) begin
    if rst=RSTDEF then
      state <= S0;
      done <= '0';
      reg_x <= (others => '0');
      reg_y <= (others => '0');
    elsif rising_edge(clk) then
      
    end if;
  end process;

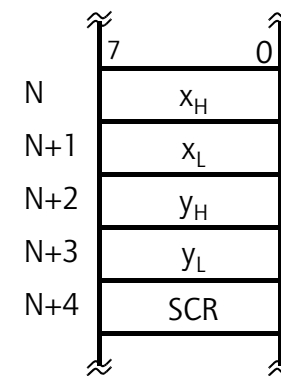
end behaviour;
```

```
case state is
  when S0 =>
    done <= '0';
    if strt='1' then
      reg_x <= x;
      reg_y <= y;
      state <= S1;
    end if;
  when S1 =>
    if reg_x=reg_y then
      done <= '1';
      state <= S0;
    elsif reg_x<reg_y then
      reg_y <= reg_y - reg_x;
      state <= S1;
    else
      reg_x <= reg_x - reg_y;
      state <= S1;
    end if;
end case;
```

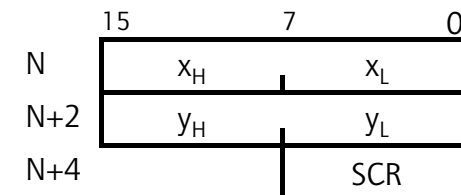
Hardware/Software-Codesign

- ◆ **Programmierschnittstelle**
 - **logisch:** gemeinsamer Adreßraum/-bereich zwischen einem Universalprozessor und einer applikationsspezifischen Schaltung zwecks der Parameterübergabe
 - **physikalisch:** Registersatz in der applikationsspezifischen Schaltung, der für den Universalprozessor sichtbar und somit programmierbar ist:
 - Datenregister für Parameterübergabe: Die Breite der Datenregister ist durch den Speicherplatzbedarf des zugehörigen Datentyps bestimmt.
 - Status-/Steuerregister als Synchronisationsschnittstelle zwischen ASIC und CPU mit Verwaltungsaufgaben.

Anordnung der ASIC-Register aus der Sicht der CPU in einem byteweise organisierten Adreßraum



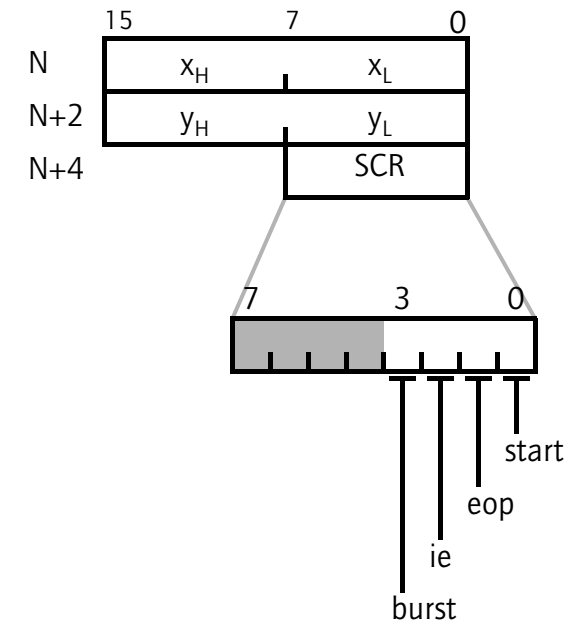
Anordnung der ASIC-Register im ASIC unter Berücksichtigung der Datentypen



Hardware/Software-Codesign

◆ Status-/Steuerregister

- enthält drei/vier Bits mit folgender Bedeutung:
 - Steuerbit *start* zum Starten des in Hardware realisierten Algorithmus
 - Statusbit *eop* (end of processing) zur Anzeige des Endes der Verarbeitung
 - Das Interrupt-Enable-Bit *ie*, zum Sperren/Freigeben von Interrupts, die durch das Bit *eop* ausgelöst werden
 - optional das Steuerbit *burst* zur Aktivierung des Datenübertragungsmodus, sofern eine applikationsspezifische Schaltung selbständig Speicherzugriffe durchführen kann (DMA-Modus): Datenübertragung entweder in mehreren Buszyklen (Blockzyklus, burst mode) oder in nur einem Buszyklus (cycle-steal mode)



- ♦ VHDL-Implementierung einer Programmierschnittstelle
 - folgende Punkte sind zu berücksichtigen:
 - der Universalprozessor darf jeder Zeit auf Register(teile) einer Programmierschnittstelle einer applikationsspezifischen Schaltung lesend und schreibend zugreifen, sofern er die Kontrolle über den Systembus hat.
 - Diese Lese- und Schreibzugriffe dürfen die Arbeit der applikationsspezifischen Schaltung auf keine Weise beeinträchtigen.
 - Kritisch sind vor allem Schreibzugriffe, bei denen der Universalprozessor und die applikationsspezifische Schaltung dasselbe Register(teil) im selben Takt beschreiben wollen. → Konfliktlösung
 - Die Verhaltensbeschreibung von Registern mit Hilfe eines Prozesses koordiniert gleichzeitige Schreibzugriffe auf die Register durch die sequentielle Ausführung der Signalzuweisungen.
 - In VHDL werden Signale, denen ein Wert innerhalb eines Prozesses zugewiesen wird, erst am Ende des Prozesses aktualisiert. Befinden sich in einem Prozeß mehrere verzögerungsfreie Signalzuweisungen, die dasselbe Signal als Ziel haben, so bekommt dieses Signal nach der Ausführung des Prozesses den Wert derjenigen Signalzuweisung zugewiesen, die als letzte ausgeführt wurde.

Hardware/Software-Codesign

- ♦ Businterface applikationsspezifischer Schaltungen (1)
 - Schaltungen mit universellem Businterface (z.B. UART-Bausteine):
 - 8-Bit-Datenbus
 - redundante Steuersignale mit positivem und negativem Pegel aktiv (z.B. IOR und $\overline{\text{IOR}}$)
 - redundante Anwahlsignale (CS und $\overline{\text{CS}}$)
 - Adreßstrobe (AS)
 - Anpassung des Businterfaces an die jeweilige Einsatzumgebung:
 - Daten-/Adreßbusbreite
 - Steuersignale ($\overline{\text{RD}}$ und $\overline{\text{WR}}$ oder nur R/ $\overline{\text{W}}$)
 - synchrones bzw. asynchrones Busprotokoll
 - Synchrone Systemschnittstelle: das ASIC arbeitet mit demselben Takt wie die CPU bzw. das Busprotokoll
 - Asynchrone Systemschnittstelle: das ASIC wird mit einem eigenen Taktgenerator angetrieben, unabhängig von der CPU
 - Aufteilung eines ASIC in ein Businterface und einen ASIC-Kern.

Hardware/Software-Codesign

- ◆ Businterface applikationsspezifischer Schaltungen (2)
 - ASIC-Ansteuerung als Analogie zur Speicheransteuerung

Tabelle: 32kB-RAM – Pinsfunktion

\overline{CS}	\overline{WE}	\overline{OE}	I/O	Mode
H	-	-	High Z	unselected / power down
L	H	L	Data Out	read
L	L	-	Data In	write
L	H	H	High Z	unselected

Speicheransteuerung

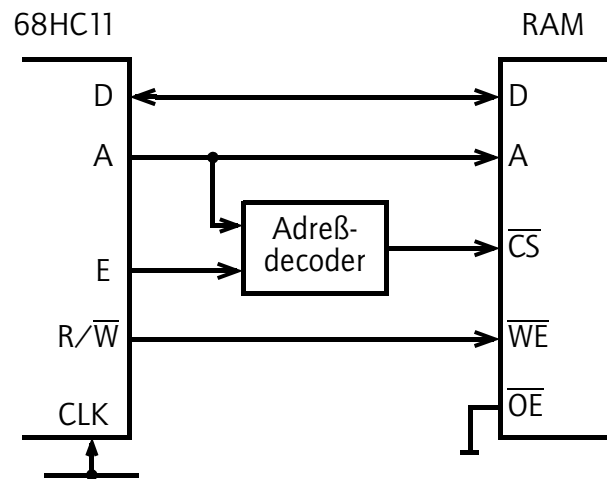
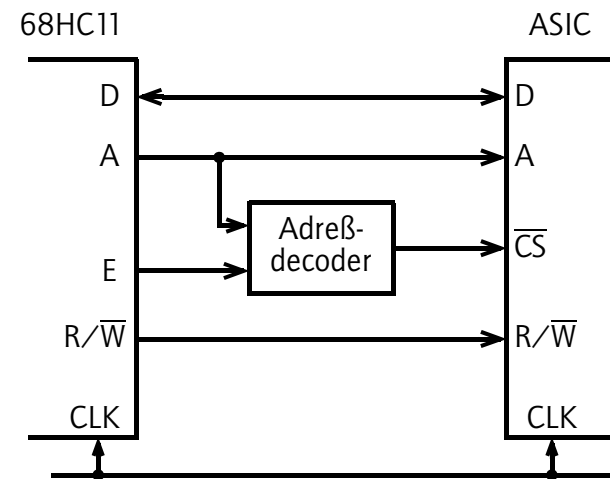


Tabelle: ASIC – Pinsfunktion

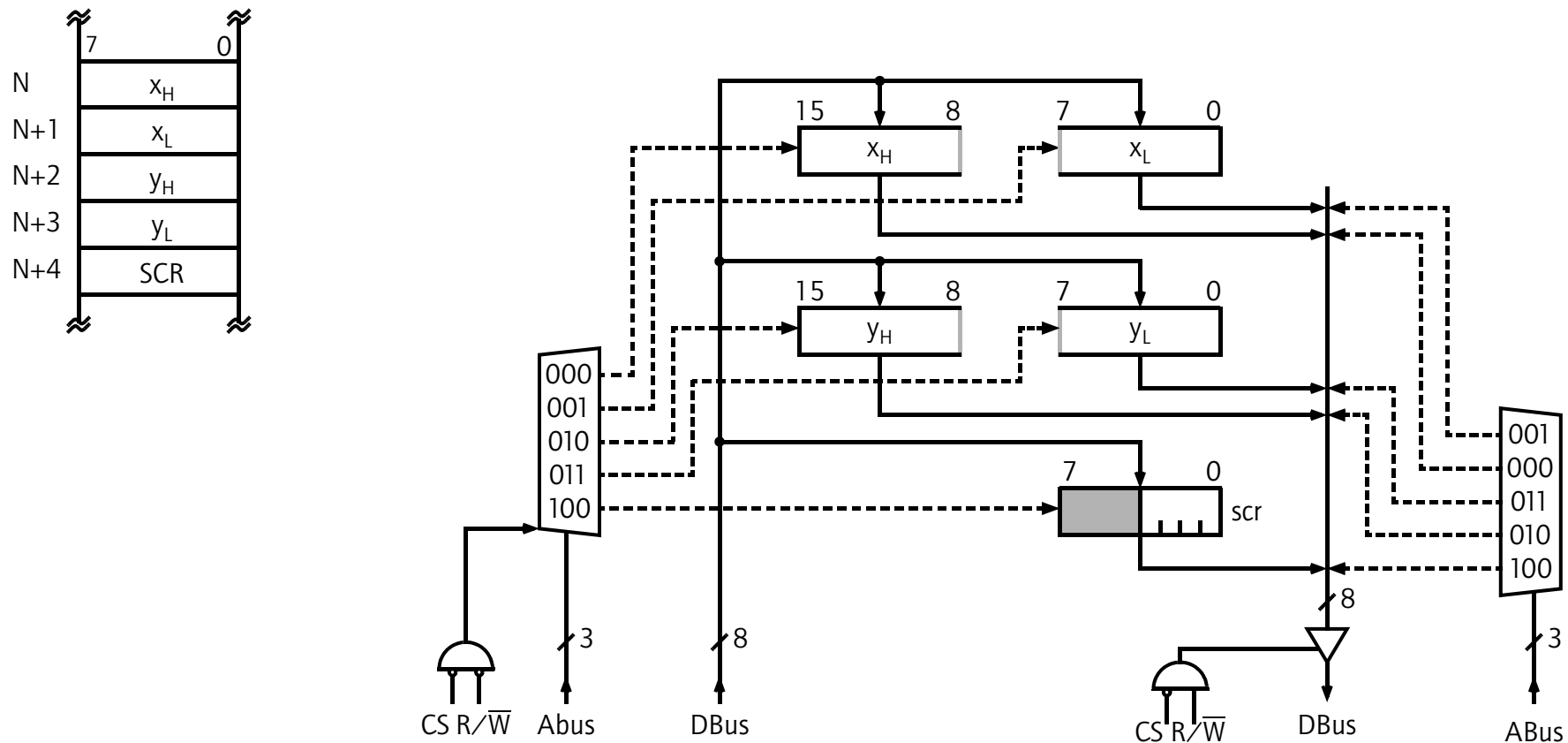
\overline{CS}	R/ \overline{W}	I/O	Mode
H	-	High Z	unselected
L	L	Data In	write
L	H	Data Out	read

ASIC-Ansteuerung



Hardware/Software-Codesign

- ◆ Businterface applikationsspezifischer Schaltungen (3)
 - Synchroner Schnittstelle zum Systembus



Hardware/Software-Codesign

◆ Businterface applikationsspezifischer Schaltungen (3)

```
process (rst, clk) begin
  if rst=RSTDEF then
    strt <= '0';
    ie   <= '0';
    eop  <= '0';
  elsif rising_edge(clk) then
    if done='1' then
      eop <= '1';
      strt <= '0';
    end if;
  end if;
end process;

if cs='0' and rw='0' then
  case abus is
    when "000" => x(15 downto 8) <= unsigned(dbus);
    when "001" => x( 7 downto 0) <= unsigned(dbus);
    when "010" => y(15 downto 8) <= unsigned(dbus);
    when "011" => y( 7 downto 0) <= unsigned(dbus);
    when "100" => strt <= dbus(0);
                  ie   <= dbus(1);
                  eop  <= '0';
    when others => null;
  end case;
end if;
```

```
with abus select
  dout <= std_logic_vector(res(15 downto 8)) when "000",
         std_logic_vector(res( 7 downto 0)) when "001",
         "00000" & eop & ie & strt           when "100",
         "00000000"                          when others;
```

```
dbus <= dout when cs='0' and rw='1' else (others => 'Z');
```

Hardware/Software-Codesign

◆ Erweiterung des GCD-Prozessors um das Businterface

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity asic is
  port(rst:    in  std_logic; -- reset, low active
        clk:   in  std_logic; -- clock, rising edge active
        cs:    in  std_logic; -- chip select, low active
        rw:    in  std_logic; -- write enable, low active
        int:   out std_logic; -- interrupt, low active
        abus:  in  std_logic_vector(2 downto 0); -- address bus
        dbus:  inout std_logic_vector(7 downto 0)); -- data bus
end asic;

architecture behaviour of asic is

  constant RSTDEF: std_logic := '0';

  component gcd
    generic(RSTDEF: std_logic);
    port(rst:    in  std_logic;
          clk:   in  std_logic;
          strt:  in  std_logic;
          done:  out std_logic;
          x:     in  unsigned(15 downto 0);
          y:     in  unsigned(15 downto 0);
          res:   out unsigned(15 downto 0));
  end component;
```

Hardware/Software-Codesign

◆ Erweiterung des GCD-Prozessors um das Businterface

```
signal x:    unsigned(15 downto 0);
signal y:    unsigned(15 downto 0);
signal res:  unsigned(15 downto 0);
signal strt, ie, eop, done: std_logic;
signal dout: std_logic_vector(7 downto 0);
```

begin

```
int <= '0' when ie='1' and eop='1' else 'Z';
```

```
ul: gcd
```

```
generic map(RSTDEF => RSTDEF)
```

```
port map (rst  => rst,
          clk  => clk,
          strt => strt,
          done => done,
          x    => x,
          y    => y,
          res  => res);
```

```
with abus select
```

```
dout <= std_logic_vector(res(15 downto 8)) when "000",
        std_logic_vector(res( 7 downto 0)) when "001",
        "00000" & eop & ie & strt          when "100",
        "00000000"                        when OTHERS;
```

```
dbus <= dout when cs='0' and rw='1' else (others => 'Z');
```

Hardware/Software-Codesign

◆ Erweiterung des GCD-Prozessors um das Businterface

```
process (rst, clk) begin
  if rst=RSTDEF then
    strt <= '0';
    ie   <= '0';
    eop  <= '0';
  elsif rising_edge(clk) then
    if cs='0' and rw='0' then
      case abus is
        when "000" => x(15 downto 8) <= unsigned(dbus);
        when "001" => x( 7 downto 0) <= unsigned(dbus);
        when "010" => y(15 downto 8) <= unsigned(dbus);
        when "011" => y( 7 downto 0) <= unsigned(dbus);
        when "100" => strt <= dbus(0);
                      ie   <= dbus(1);
                      eop  <= '0';
        when others => null;
      end case;
    end if;
    if done='1' then
      eop <= '1';
      strt <= '0';
    end if;
  end if;
end process;

end behaviour;
```

Hardware/Software-Codesign

◆ Testumgebung des GCD-Prozessors mit Businterface

```
entity asic_tb is
    -- empty
end asic_tb;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

architecture behaviour of asic_tb is
    constant RSTDEF: std_logic := '0';
    constant FRQMAX: natural := 50E6;
    constant tpd: time := 1 sec / FRQMAX;

    component asic
        port(rst:    in    std_logic;
             clk:    in    std_logic;
             cs:     in    std_logic;
             rw:     in    std_logic;
             int:    out   std_logic;
             abus:   in    std_logic_vector(2 downto 0);
             dbus:   inout std_logic_vector(7 downto 0));
    end component;

    signal rst: std_logic := RSTDEF;
    signal clk: std_logic := '0';
    signal hlt: std_logic := '0';

    signal cs:    std_logic := '1';
    signal rw:    std_logic := '1';
    signal int:   std_logic := 'H';
    signal abus:  std_logic_vector(2 DOWNTO 0) := (OTHERS => '0');
    signal dbus:  std_logic_vector(7 DOWNTO 0) := (OTHERS => 'Z');
```


Hardware/Software-Codesign

◆ Testumgebung des GCD-Prozessors mit Businterface

```
begin
```

```
rst <= RSTDEF, not RSTDEF after 5*tpd;  
clk <= not clk after tpd/2 when hlt='0' else '0';
```

```
int <= 'H'; -- pull up
```

```
ul: asic  
  port map(rst => rst,  
           clk => clk,  
           cs  => cs,  
           rw  => rw,  
           int => int,  
           abus => abus,  
           dbus => dbus);
```

```
test: process  
  variable x: std_logic_vector(15 downto 0) := X"6738";  
  variable y: std_logic_vector(15 downto 0) := X"C434";  
  variable r: std_logic_vector(15 downto 0);  
  variable s: std_logic_vector( 7 downto 0);  
  
  procedure write(sel: std_logic_vector(2 downto 0);  
                 arg: std_logic_vector(7 downto 0)) is  
    begin  
      dbus <= arg;  
      abus <= sel;  
      cs   <= '0';  
      rw   <= '0';  
      wait until clk'event and clk='1';  
      dbus <= (others => 'Z');  
      cs   <= '1';  
      rw   <= '1';  
      wait until clk'event AND clk='1';  
    end procedure write;
```

◆ Testumgebung des GCD-Prozessors mit Businterface

```
procedure read(sel: in std_logic_vector(2 downto 0);
               arg: out std_logic_vector(7 downto 0)) is
begin
    abus <= sel;
    cs   <= '0';
    rw   <= '1';
    wait until clk'event and clk='1';
    arg  := dbus;
    cs   <= '1';
    rw   <= '1';
    wait until clk'event and clk='1';
end procedure read;

begin
    wait until clk'event and clk='1' and rst=NOT RSTDEF;
    write("000", x(15 downto 8));
    write("001", x( 7 downto 0));
    write("010", y(15 downto 8));
    write("011", y( 7 downto 0));
    write("100", "00000001");

    s := (others => '0');
    while s(2)='0' loop
        read("100", s);
    end loop;

    read("000", r(15 downto 8));
    read("001", r( 7 downto 0));
    hlt <= '1';
    wait;
end process;

end architecture behaviour;
```

Hardware/Software-Codesign

♦ Fallstudie: Selection-Sort-Algorithmus

- Synthese eines applikationsspezifischen Co-Prozessors, der selbstständig Daten im Speicher verarbeiten kann und mit dem Pico-Blaze über seine IO-Schnittstelle kommuniziert

```
// Anzahl der zu sortierenden Zeichen
unsigned int n;
```

```
// Zeiger auf das zu sortierende Feld
unsigned char *a;
```

```
void sort(void) {
    int i, j, min, t;
    for (i=0; i<n-1; i++) {
        min = i;
        for (j=i+1; j<n; j++) {
            if (a[j] < a[min]) {
                min = j;
            }
        }
        t = a[min];
        a[min] = a[i];
        a[i] = t;
    }
}
```

```
-- positive: range 0 to integer'high
-- natural:   range 1 to integer'high
```

```
procedure sort(a: inout string;
               n: positive) is
    variable tmp: character;
    variable min: natural;
begin
    for i in 0 to n-2 loop
        min := i;
        for j in i+1 to n-1 loop
            if a(j) < a(min) then
                min := j;
            end if;
        end loop;
        tmp := a(min);
        a(min) := a(i);
        a(i) := tmp;
    end loop;
end procedure;
```

Hardware/Software-Codesign

- ◆ Testumgebung mit repräsentativen Testfällen
 - Referenzdaten: sortiertes Datenfeld
 - Testdaten: unsortiertes, sortiertes und umgekehrt sortiertes Datenfeld

```
constant N: natural := 20;
type string is array(natural range <>) of character;

constant g: string(0 to N-1) := "1449BEKMMPQTZceffrvz";

variable a: string(0 to N-1) := "P91fQeZB4KvTMcrEfzM4";
variable b: string(0 to N-1) := "1449BEKMMPQTZceffrvz";
variable c: string(0 to N-1) := "zvrffecZTQPMKEB9441";

...

sort(a, n);
assert a=g severity error;
sort(b, n);
assert b=g severity error;
sort(c, n);
assert c=g severity error;
```

Hardware/Software-Codesign

- ♦ alle Änderungen/Transformationen/Optimierungen des Algorithmus müssen semantikerhaltend sein
- ♦ hardware-unabhängige und code-verbessernde Optimierungen
 - Anzahl „teurer“ Operationen reduzieren
 - Speicherzugriffe sind i.a. nicht parallelisierbar
 - mehrere Takte, ggf. Hilfsvariablen einführen
 - Schleifeninvariante Ausdrücke vor dem Schleifenkopf auswerten
 - die Auswertung $a(\min)$ in der IF-Abfrage wird durch eine einfache Registerzuweisung ersetzt
 - unnötige Abläufe mit Sperrvariablen/-flags verhindern
 - der (unnötige) Austausch der Elemente im Speicher findet auch dann statt, wenn beide Elemente gleich gross sind
 - Eigenschaften von RAM-Blöcken in FPGA nutzen
 - synchrone Lesezugriffe, (Pipeline-)Register am RAM-Blockausgang

Hardware/Software-Codesign

♦ hardware-unabhängige und code-verbessernde Optimierungen

```
procedure sort(a: inout string; n: positive) is
  variable tmp: character;
  variable min: natural;
  variable x, y: character;      -- Hilfsvariable
  variable swp: boolean;         -- Hilfsvariable
begin
  for i in 0 to n-2 loop
    min := i;
    y := a(i);
    tmp := y;
    swp := false;
    for j in i+1 to n-1 loop
      x := a(j);
      if x < tmp then
        swp := true;
        min := j;
        tmp := x;
      end if;
    end loop;
    if swp then
      a(min) := y;
      a(i) := tmp;
    end if;
  end loop;
end procedure;
```

Hardware/Software-Codesign

- ♦ FOR-Schleifen durch WHILE-Schleifen ersetzen
 - explizite Schleifenzähler einführen
 - Wertebereiche berücksichtigen

```
for i in 0 to n-2 loop
  ...
```

```
  for j in i+1 to n-1 loop
    ...
  end loop;
```

```
  ...
```

```
end loop;
```

```
variable i: natural range 0 to n-1;
variable j: natural range 1 to n;
```

```
i := 0;
while i <= n-2 loop
  ...
```

```
    j := i+1;
    while j <= n-1 loop
      ...
      j := j + 1;
    end loop;
```

```
    ...
```

```
    i := i + 1;
  end loop;
```

Hardware/Software-Codesign

◆ FOR-Schleifen durch WHILE-Schleifen ersetzen

```
procedure sort ...
```

```
variable x, y: character;  
variable tmp: character;  
variable swp: boolean;  
variable min: natural range 0 to n-1;  
variable i: natural range 0 to n-1;  
variable j: natural range 1 to n;
```

```
begin
```

```
for i in 0 to n-2 loop  
  ...  
  for j in i+1 to n-1 loop  
    ...  
  end loop;  
  ...  
end loop;
```

```
end procedure;
```

```
i := 0;  
while i <= n-2 loop  
  min := i;  
  y := a(i);  
  tmp := y;  
  swp := false;  
  j := i+1;  
  while j <= n-1 loop  
    x := a(j);  
    if x < tmp then  
      swp := true;  
      min := j;  
      tmp := x;  
    end if;  
    j := j + 1;  
  end loop;  
  if swp then  
    a(min) := y;  
    a(i) := tmp;  
  end if;  
  i := i + 1;  
end loop;
```


Hardware/Software-Codesign

- ◆ WHILE-Schleifen durch Zustandsmaschinen ersetzen
 - Initialisierungszustand
 - Ausführungszustand-/zustände (Schleifenrumpf)

```
variable i: natural range 0 to n-1;

i := 0;
while i <= n-2 loop
    ...

    i := i + 1;
end loop;
```

```
type TState is (S0, S1);
variable state: TState := S0;

variable i: natural range 0 to n-1;

case state is
    -- Initialisierungszustand
    when S0 =>
        i      := 0;
        state := S1;

    -- Ausführungszustand
    when S1 =>
        if i <= n-2 then

            ...

            i := i + 1;
            state := S1;
        else
            state := S2;
        end if;
end case;
```

Hardware/Software-Codesign

◆ WHILE-Schleifen durch Zustandsmaschinen ersetzen

```
procedure sort ...
```

```
type TState is (S0, S1, S2);  
variable state: TState := S0;  
-- die restlichen Variablen-  
-- deklarationen bleiben unverändert
```

```
begin  
loop
```

```
case state is  
when S0 =>  
i := 0;  
state := S1;  
when S1 =>  
if i <= n-2 then  
min := i;  
y := a(i);  
tmp := y;  
swp := false;  
j := i+1;  
state := S2;  
else  
return;  
end if;
```

```
when S2 =>  
if j <= n-1 then  
x := a(j);  
if x < tmp then  
swp := true;  
min := j;  
tmp := x;  
end if;  
j := j + 1;  
state := S2;  
else  
if swp then  
a(min) := y;  
a(i) := tmp;  
end if;  
i := i + 1;  
state := S1;  
end if;  
end case;
```

```
end loop;  
end procedure;
```

Hardware/Software-Codesign

- ♦ Gemeinsame Ausdrücke mehrfach verwenden, ggf. Hilfsvariablen einführen, z.B. $m := n - 1$, ggf. Bedingungen anpassen
aus $(i \leq n-2)$ wird $(i < n-1)$ und schliesslich $(i < m)$
- ♦ Lese- und Schreibzugriffe auf den RAM-Block sind synchron, und der RAM-Block hat ein Ausgangsregister (Pipeline-Effekt)
 - Zuweisung $x := a(j)$ läuft in zwei Takten ab
 - 1. Takt : der Inhalt der adressierten Speicherzelle $a(j)$ wird im Ausgangsregister des RAM-Blocks abgelegt, und
 - 2. Takt: das Datum aus dem Ausgangsregister wird ins Zielregister (hier in die Zielvariable x) übernommen
 - die folgende Folge von Operationen $x := a(j)$; IF $x < y$ THEN ... kann somit nicht in einem Zustand ausgeführt werden. Hier muss ein Zustand dazwischen liegen.

Hardware/Software-Codesign

◆ Gemeinsame Ausdrücke und Pipeline-Effekt berücksichtigen

```
procedure sort ...
```

```
    type TState is (S0, S1, S2, S3);
```

```
    variable state: TState := S0;
```

```
    variable m: natural range 0 to n-1;
```

```
    -- die restlichen Variablen-
```

```
    -- deklarationen bleiben unverändert
```

```
begin
```

```
    loop
```

```
        case state is
```

```
            when S0 =>
```

```
                i      := 0;
```

```
                m      := n - 1;
```

```
                state := S1;
```

```
            when S1 =>
```

```
                if i < m then
```

```
                    min := i;
```

```
                    y   := a(i);
```

```
                    tmp := y;
```

```
                    swp := false;
```

```
                    j   := i+1;
```

```
                    state := S2;
```

```
                else
```

```
                    return;
```

```
                end if;
```

```
            when S2 =>
```

```
                if j <= m then
```

```
                    x := a(j);
```

```
                    state := S3;
```

```
                else
```

```
                    if swp then
```

```
                        a(min) := y;
```

```
                        a(i)   := tmp;
```

```
                    end if;
```

```
                    i := i + 1;
```

```
                    state := S1;
```

```
                end if;
```

```
            when S3 =>
```

```
                if x < tmp then
```

```
                    swp := true;
```

```
                    min := j;
```

```
                    tmp := x;
```

```
                end if;
```

```
                j := j + 1;
```

```
                state := S2;
```

```
            end case;
```

```
        end loop;
```

```
    end procedure;
```

Hardware/Software-Codesign

- ◆ Pipeline-Register ausnutzen
- ◆ Unter der Annahme, dass mit x das Ausgangsregister des RAM-Blocks bezeichnet wird, ergeben sich einige Änderungen in der Notation, und auf die Variable (das explizite Register) x kann ganz verzichtet werden
- ◆ Dadurch wird die Zuweisung $y := a(i)$ in zwei separate Zuweisungen $x := a(i)$ und $y := x$ aufgeteilt, die in zwei Takten ablaufen müssen.
=> Einfügen eines neuen (Zwischen-)Zustands oder eines sog. Sperrflags, das, nach dem es gesetzt ist, verhindert, dass Zuweisungen mehrmals ausgeführt werden.

Hardware/Software-Codesign

```
procedure sort ...
  -- Ausgangsregister des RAM-Blocks
  variable x: character;
  variable flg: boolean; -- Sperrflag
  -- die restlichen Variablen-
  -- deklarationen bleiben unverändert

begin
  loop
    case state is
      when S0 =>
        i      := 0;
        m      := n - 1;
        state := S1;
      when S1 =>
        if i < m then
          min := i;
          x   := a(i);
          flg := false;
          swp := false;
          j   := i+1;
          state := S2;
        else
          return;
        end if;
      end if;
```

```
    when S2 =>
      if not flg then
        flg := true;
        y   := x;
        tmp := x;
      end if;
      if j <= m then
        x := a(j);
        state := S3;
      else
        if swp then
          a(min) := y;
          a(i)   := tmp;
        end if;
        i := i + 1;
        state := S1;
      end if;
    when S3 =>
      if x < tmp then
        swp := true;
        min := j;
        tmp := x;
      end if;
      j := j + 1;
      state := S2;
    end case;
  end loop;
end procedure;
```

Hardware/Software-Codesign

- ◆ Definition der Signale in der Schnittstelle und Übergang auf ein synchrones (getaktetes) System

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity selectcore is
  generic(RSTDEF: std_logic := '1');
  port(rst:  in  std_logic;  -- reset, RSTDEF active
       clk:  in  std_logic;  -- clock, rising edge active

       -- handshake signals
       strt: in  std_logic;  -- start bit, high active
       done: out std_logic;  -- done bit, high active
       ptr:  in  std_logic_vector(10 downto 0); -- pointer to vector
       len:  in  std_logic_vector( 7 downto 0); -- length of vector

       -- interface to RAM block
       WEB:  out std_logic;  -- Port B Write Enable Output, high active
       ENB:  out std_logic;  -- Port B RAM Enable, high active
       ADB:  out std_logic_vector(10 downto 0); -- Port B 11-bit Address Output
       DIB:  in  std_logic_vector( 7 downto 0); -- Port B 8-bit Data Input
       DOB:  out std_logic_vector( 7 downto 0)); -- Port B 8-bit Data Output
end selectcore;
```

Hardware/Software-Codesign

- ♦ Datentypen und Speicherplatzbedarf
 - Erweiterung um Hilfssignale zur gemeinsamen Nutzung von Ressourcen

```
architecture verhalten of selectcore is
```

```
    type TState is (S0, S1, S2, S3, S4);  
    signal state, state0: TState;
```

```
    signal swp, swp0: std_logic;  
    signal flg, flg0: std_logic;  
    signal d, d0:      std_logic;  
    signal i, i0, i1:  std_logic_vector(7 downto 0);  
    signal j, j0:      std_logic_vector(8 downto 0);  
    signal m, m0:      std_logic_vector(7 downto 0);  
    signal y, y0:      std_logic_vector(7 downto 0);  
    signal tmp, tmp0:  std_logic_vector(7 downto 0);  
    signal min, min0:  std_logic_vector(7 downto 0);  
    signal ofs:        std_logic_vector(7 downto 0);
```

```
begin
```

```
    done <= d;  
    ADR  <= ptr + ofs;  
    i1   <= i + 1;
```


Hardware/Software-Codesign

◆ Registertransferbeschreibung mit der 2-Prozeßmethode

```
signal state, state0: TState;  
signal min,   min0:   std_logic_vector(7 downto 0);
```

```
reg: process (rst, clk) is  
begin  
    if rst=RSTDEF then  
        state <= S0;  
        min   <= (others => '0');  
    elsif rising_edge(clk) then  
        state <= state0;  
        min   <= min0;  
    end if;  
end process;  
  
fsm: process (state, min, i, m, ... ) is  
begin  
    state0 <= state;  
    min0   <= min;  
    case state is  
        when S1 =>  
            min0 <= i;  
            ...  
        when S2 =>  
            ofs <= min;  
            ...  
    end case;  
end process;
```

Hardware/Software-Codesign

- ◆ Variablen werden zu Registern, die mit Hilfe von Signalen in einem getakteten Prozeß modelliert sind

```
reg: process (rst, clk) is
begin
    if rst=RSTDEF then
        state <= S0;
        i      <= (others => '0');
        j      <= (others => '0');
        m      <= (others => '0');
        y      <= (others => '0');
        tmp    <= (others => '0');
        min    <= (others => '0');
        d      <= '0';
        flg    <= '0';
        swp    <= '0';
    elsif rising_edge(clk) then
        state <= state0;
        i      <= i0;
        j      <= j0;
        m      <= m0;
        y      <= y0;
        tmp    <= tmp0;
        min    <= min0;
        d      <= d0;
        flg    <= flg0;
        swp    <= swp0;
    end if;
end process;
```

◆ Übergangs-/Ausgangsschaltnetz

```
fsm: process (state, strt, len, i, i1, j, d, m, y, tmp, min, flg, swp, dib) is
begin
    state0 <= state;
    i0      <= i;
    j0      <= j;
    m0      <= m;
    y0      <= y;
    tmp0    <= tmp;
    min0    <= min;
    d0      <= d;
    flg0    <= flg;
    swp0    <= swp;

    ofs     <= i;    -- default (OTHERS => '0');
    WEB     <= '0';
    ENB     <= '0';
    DOB     <= tmp;  -- default (OTHERS => '0');
    case state is
        when S0 =>
            if strt='1' then
                d0      <= '0';
                i0      <= (others => '0');
                m0      <= len - 1;
                state0 <= S1;
            end if;
```

Hardware/Software-Codesign

♦ Übergangs-/Ausgangsschaltnetz (Fortsetzung)

```
when S2 =>
  if flg='0' then
    flg0 <= '1';
    y0    <= DIB;
    tmp0  <= DIB;
  end if;
  if j<=m then
    ofs    <= j(ofs'range);
    ENB    <= '1';
    state0 <= S3;
  else
    if swp='1' then
      ofs    <= min;
      DOB    <= y;
      ENB    <= '1';
      WEB    <= '1';
      state0 <= S4;
    else
      i0     <= i1;
      state0 <= S1;
    end if;
  end if;
end if;
```

Hardware/Software-Codesign

♦ Übergangs-/Ausgangsschaltnetz (Fortsetzung)

```
    when S3 =>
        if DIB<tmp then
            swp0    <= '1';
            min0    <= j(min0'range);
            tmp0    <= DIB;
        end if;
        j0        <= j + 1;
        state0 <= S2;
    when S4 =>
        -- ofs      <= i;
        -- DOB      <= tmp;
        ENB        <= '1';
        WEB        <= '1';
        i0         <= i1;
        state0 <= S1;
    end case;

end process;

end verhalten;
```

Hardware/Software-Codesign

◆ Schnittstelle zum PicoBlaze

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

entity selectsort is
  generic(RSTDEF: std_logic := '1');
  port(rst:    in  std_logic;  -- reset, RSTDEF active
       clk:    in  std_logic;  -- clock, rising edge active

       -- interface to PicoBlaze
       rsel:   in  std_logic_vector(7 downto 0); -- register select
       din:    in  std_logic_vector(7 downto 0); -- data input
       dout:   out std_logic_vector(7 downto 0); -- data output
       ena:    in  std_logic;  -- enable, high active
       wre:    in  std_logic;  -- write strobe, high active

       -- interface to RAM block through port B
       WEB:    out std_logic;  -- Port B Write Enable Output, high active
       ENB:    out std_logic;  -- Port B RAM Enable, high active
       ADDR_B: out std_logic_vector(10 downto 0); -- Port B 11-bit Address Output
       DIB:    in  std_logic_vector(7 downto 0);  -- Port B 8-bit Data Input
       DOB:    out std_logic_vector(7 downto 0)); -- Port B 8-bit Data Output
end selectsort;
```

◆ Programmierschnittstelle

```
process (rst, clk) is
begin
  if rst=RSTDEF then
    len  <= (others => '0');
    ptr  <= (others => '0');
  elsif rising_edge(clk) then
    if ena='1' and wre='1' then
      case rsel(1 downto 0) is
        when "01"    => ptr( 7 downto 0) <= din;
        when "10"    => ptr(10 downto 8) <= din(2 downto 0);
        when "11"    => len <= din;
        when others  => null;
      end case;
    end if;
  end if;
end process;

strt <= din(0) when rsel(1 downto 0)="00" and ena='1' and wre='1' else '0';

with rsel(1 downto 0) select
dout <= "0000000" & done when "00",
      ptr(7 downto 0)  when "01",
      "00000" & ptr(10 downto 8) when "10",
      len              when others;
```

◆ Instantiierung der Komponente

```
u1: selectcore
  generic map(RSTDEF => RSTDEF)
  port map(rst  => rst,
           clk  => clk,
           strt => strt,
           done => done,
           ptr  => ptr,
           len  => len,
           WEB  => WEB,
           ENB  => ENB,
           ADR  => ADDR_B,
           DIB  => DIB,
           DOB  => DOB);
```