

3 Entwurfsmethoden und Hardware-Beschreibungssprachen

- Entwurfsmethoden und ihre Eigenschaften
- VHDL
 - Der Aufbau von VHDL-Systemen
 - Datentypen, Signale, Konstanten und Attribute
 - Operatoren und numeric_std-Funktionen
 - Nebenläufigkeit in VHDL, Prozesse und erweiterte Signalzuweisungen
 - Beispiele, Tipps, "Tricks & Pitfalls"
 - Strukturelle Beschreibung und Simulation
 - Effizienter VHDL-Entwurf für FPGA-Architekturen

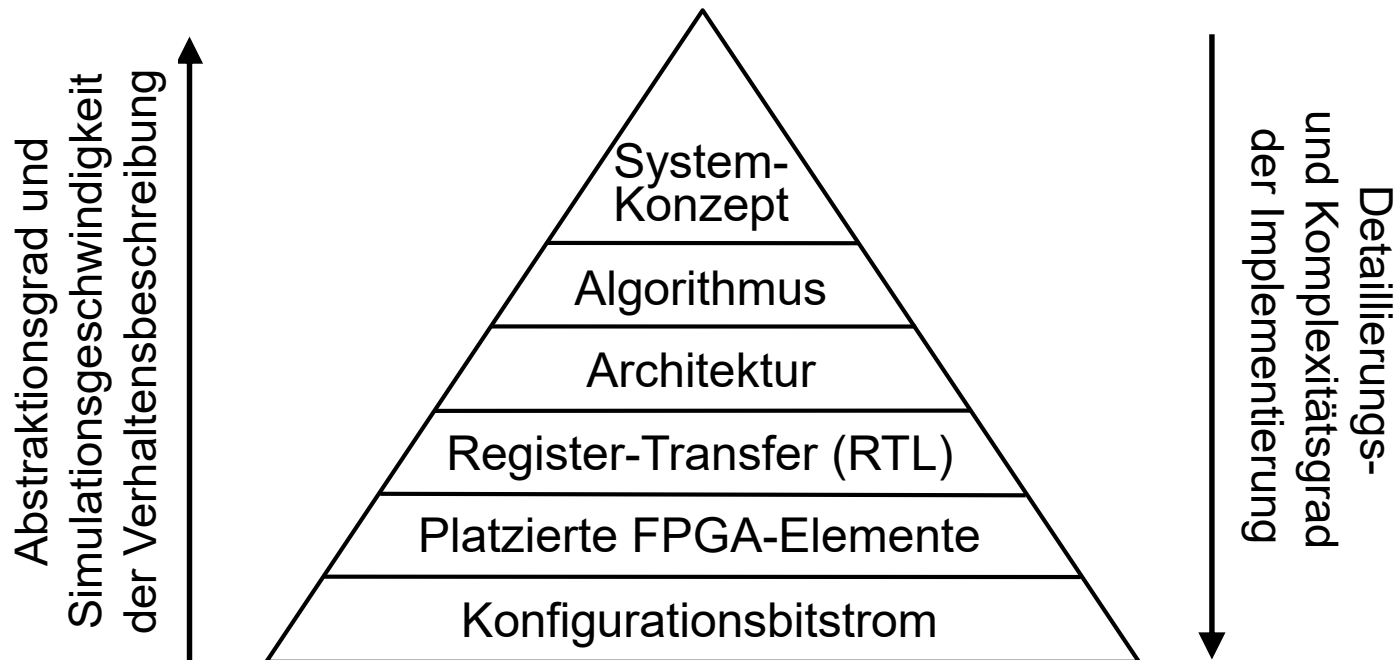
- **Ursprünglich**, wurden die Schaltungen für FPGAs mit Hilfe sogenannter **Schematic-Entry-Tools** aufgenommen. Die geringe Komplexität früherer FPGA-Bausteine erlaubte sogar eine Handoptimierung zwecks höherer Performance
- Diese Vorgehensweise besitzt signifikante Nachteile (mangelnde Abstraktion, Aufwand, etc.) und wurde sukzessive durch neue Methoden ersetzt
- Heute werden zumeist **Top-Down Design-Strategien** verwendet, bei denen Hardware-Beschreibungssprachen (**Hardware-Description Languages, HDLs**) eingesetzt werden
- Auch Beschreibungen in höheren Sprachen (SystemC, Chisel, SystemVerilog) lassen sich für FPGA-Technologien übersetzen

Vorteile HDL-gestützter Design-Methoden

- erhöhte Produktivität und verkürzte Entwicklungszeiten
- verringerte Non-Recurring-Engineering- (NRE) Kosten
- Design-Reuse wird ermöglicht
- erhöhte Flexibilität in Bezug auf Design-Modifikationen
- schnellere Exploration alternativer Architekturen und Technologie-Libraries
- ermöglicht den Einsatz von Synthese-Werkzeugen und so die schnellere und einfachere Exploration des Entwurfsraums hinsichtlich
 - Fläche
 - Timing
 - Verlustleistungsaufnahme
- bessere und leichtere Design-Verifikation

Ebenen der Verhaltensbeschreibung (FPGAs)

- Eine Top-Down-Entwurfsmethodik überführt ein HDL-Modell der Hardware von einer hohen Abstraktionsebene (System oder Algorithmus) hinab zur bausteinspezifischen Netzliste mit fertig konfigurierten und verdrahteten FPGA-Architekturelementen



Hauptfunktion einer HDL

- Eine *Hardware Description Language* ist eine formale Sprache, mit der logischen Funktionen für ihre Abbildung in eine bestimmte Hardware, z.B. FPGA oder ASIC, beschrieben werden können
- Die wichtigsten HDLs sind
 - VHDL (Very High Speed Integrated Circuit HDL)
 - Verilog
- VHDL unterstützt zwei Beschreibungsarten
 - eine abstrakte, funktionale Beschreibung der Funktion
 - eine strukturelle Beschreibung der Hardware
- Das Verhalten der Hardware kann auf unterschiedlichen Abstraktionsebenen beschrieben werden
 - Auf einer hohen Ebene mit wenig Detailinformationen und einer hohen Abstraktion von Hardware
 - Auf einer geringen Abstraktionsebene mit mehr Detailinformationen

Geschichte von VHDL I

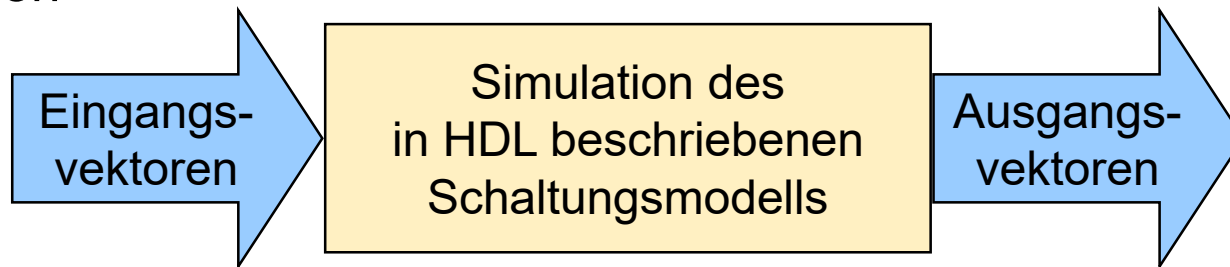
- 1980 US-Verteidigungsministerium verwendet eine Standard HDL um Schaltungsdesign selbst dokumentierend zu machen, einen gemeinsamen Design-Flow zu verfolgen und bei neuen Technologien alte Schaltungsentwürfe wieder verwenden zu können
- 1983 IBM, Texas Instruments, Intermetrics und das US-Verteidigungsministerium schließen sich zusammen, um VHDL und entsprechende Simulations-Tools zu entwickeln
- 1987 Das US Verteidigungsministerium fordert, dass alle digitalen Schaltungen, die für das Verteidigungsministerium neu entwickelt werden, in VHDL beschrieben werden müssen.
Der IEEE verabschiedet VHDL als Standard 1076
- 1993 Der Standard VHDL wird durch IEEE 1164 zu IEEE 1076-1993 (VHDL'93) erweitert. Diese Version ist die am weitesten verbreitete

Geschichte von VHDL II

- 1996 Kommerzielle Synthese- und Simulations-Tools werden verfügbar. Weitere Pakete zur Synthese werden dem Standard hinzugefügt
- 1999 Erweiterung für analoge und gemischte Schaltungen (VHDL-AMS)
- 2002 Einführung geschützter Typen und Änderungen für mehr Flexibilität (VHDL-2002: aktueller Standard bei Mainstream-Simulationswerkzeuge)
- 2006 Mehr Operatoren, größere Flexibilität, Integration von Schnittstellen zu C/C++. Alles für eine effizientere Nutzung auf Systemebene (VHDL-2006, Draft 3.0)
- 2009 Veröffentlichung von VHDL 4.0 (VHDL-2008): Ressourcen für IP-Schutz, neue Typen für Fest- und Gleitkommazahlen, Korrekturen und Verbesserung bzgl. der Neuerungen aus VHDL 3.0. Integriert alle bereits vorhandenen sog. *std-Packages*. Dauerte einige Jahre, bis Standard von den Tool-Herstellern komplett unterstützt wurde

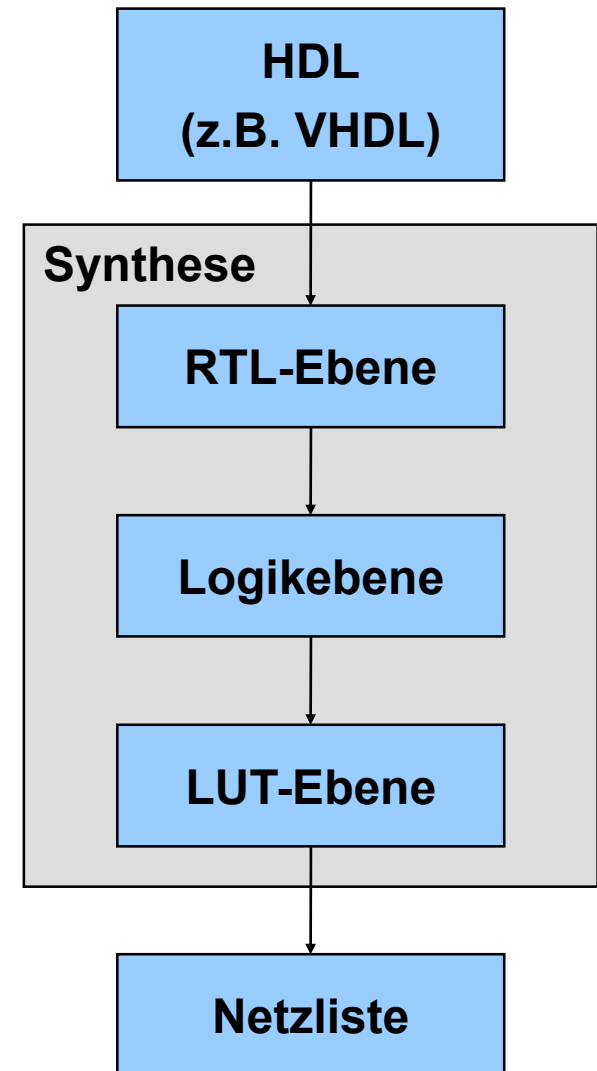
Design Automation Tools: Simulation

- Simulatoren dienen der funktionalen Verifikation einer Schaltungsbeschreibung auf verschiedenen Abstraktionsebenen
- Es kann sowohl das logische als auch das zeitliche Verhalten simuliert werden
- Zur Simulation des zeitlichen Verhaltens müssen:
 - entweder vor der Synthese Zeitinformationen im HDL-Programm integriert werden
 - oder nach der Synthese werden diese Informationen aus den Zellbibliotheken der FPGA- bzw. ASIC-Hersteller bezogen
- Zeitliche Simulationen sollten nur nach der Synthese durchgeführt werden



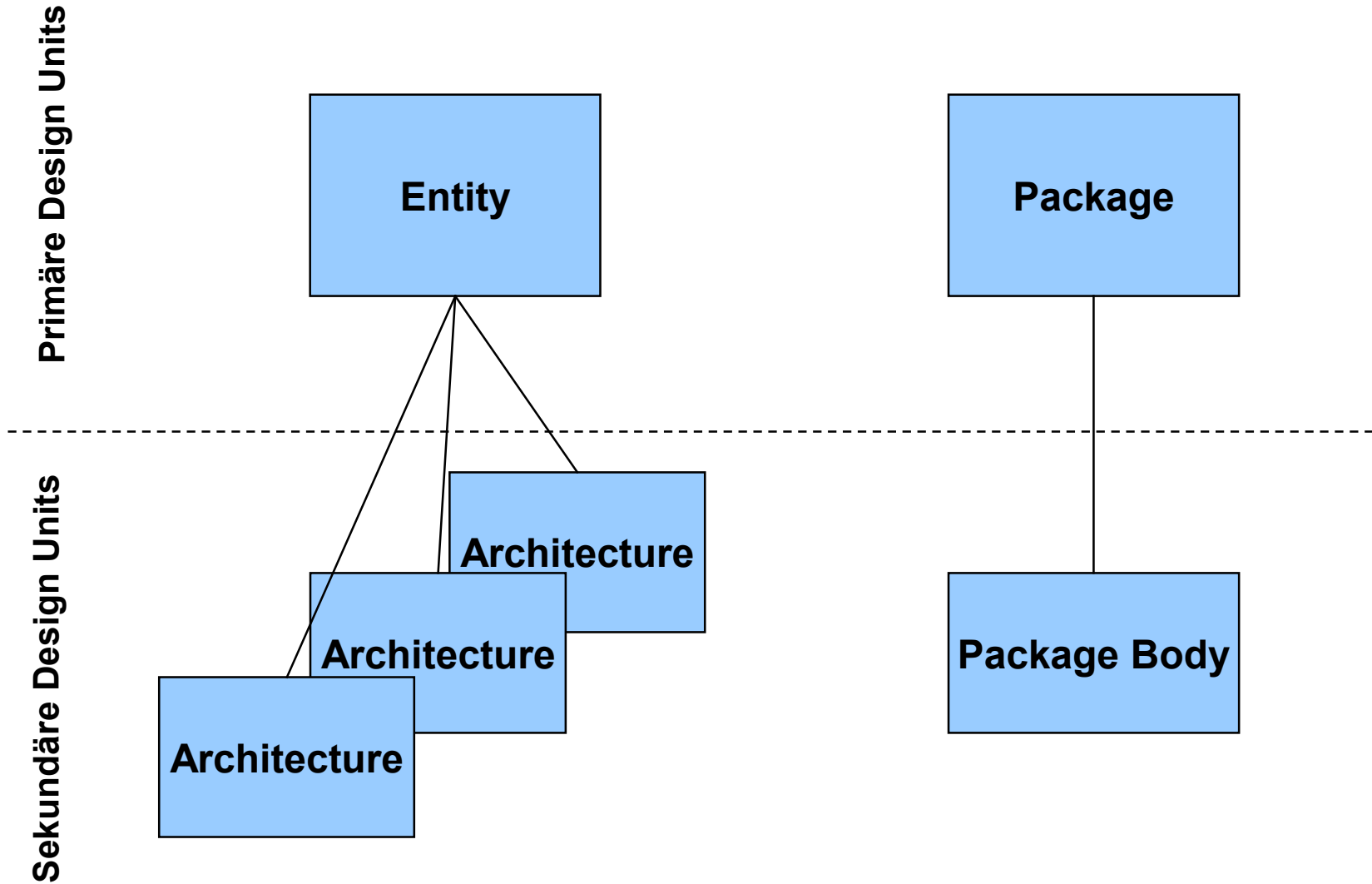
Design Automation Tools: Synthese

- Synthese ist das Überführen einer HDL-Beschreibung über verschiedene Zwischenstufen in eine Netzliste, die die FPGA-Konfiguration beschreibt. Auf jeder Zwischenstufe werden logische Optimierungstechniken angewendet
- Einige Sprachkonstrukte, die in der Simulation sinnvoll sein können (z.B. zeitliches Verhalten, Fließkommazahlen oder Operationen zur Bearbeitung von Dateien) werden von den Synthese-Tools nicht unterstützt

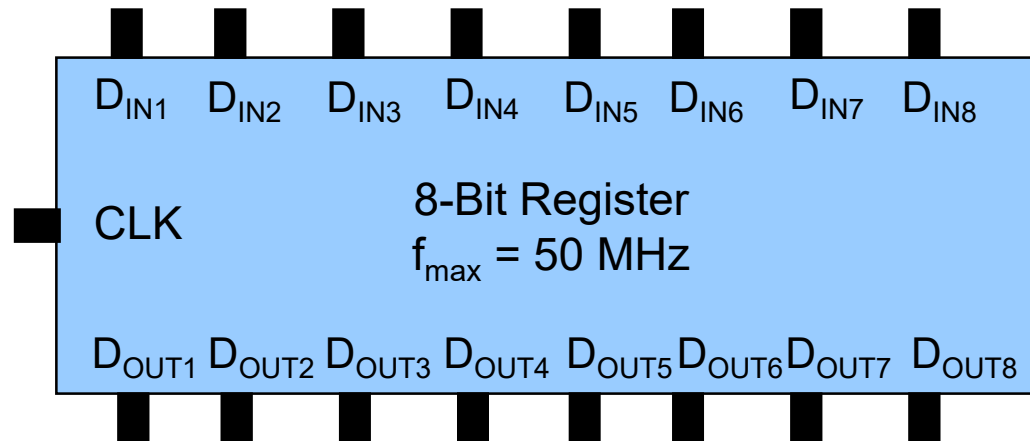


Der Aufbau von VHDL-Systemen

Der Aufbau von VHDL-Systemen – Design Units



Design Unit: Entity



Eine *Entity* besteht aus

- **Parametern**, welche die Systemstruktur betreffen (z.B. Breite eines Busses, max. Takt-Frequenz)
- **Verbindungen**, die Informationen in das System und aus dem System transferieren

```
entity n_bit_register is
    generic (
        width : integer := 8;
        fmax   : integer := 50
    );
    port (
        D_IN  : IN bit_vector(0 to width-1);
        D_OUT : OUT bit_vector(0 to width-1);
        CLK   : IN bit
    );
end entity n_bit_register;
```

Design Unit: Architecture

- Eine *Entity* (z.B. x86 Prozessor) kann mehrere *Architectures* besitzen (z.B. AMD, INTEL, CYRIX), die alle das selbe Interface besitzen

```
entity x86 is
    port(
        ...
    );
end entity;
```

```
architecture AMD of x86 is
    ...
end architecture AMD;
```

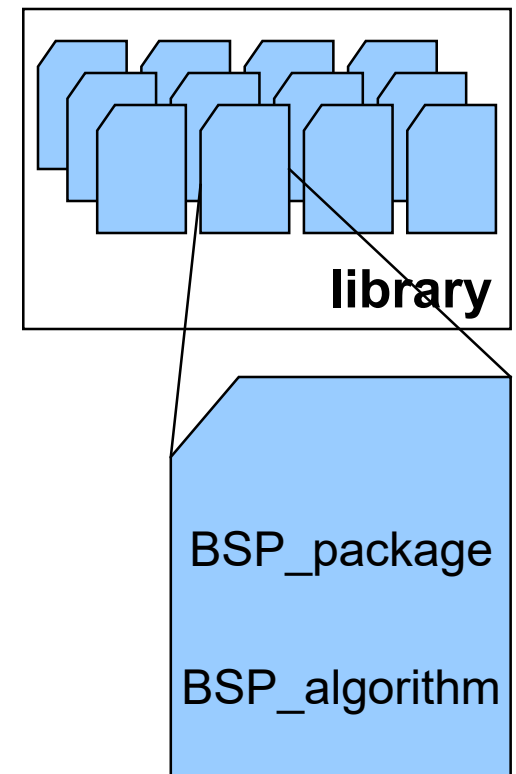
```
architecture INTEL of x86 is
    ...
end architecture INTEL;
```

```
architecture CYRIX of x86 is
    ...
end architecture CYRIX;
```

Design Unit: Package

- *Packages* enthalten Elemente, z.B. Typen- und Konstantendeklarationen, *Entities*, Funktionen, etc., die nicht zum VHDL-Standard gehören.

```
library BSP_Lib;  
use BSP_lib.BSP_package.BSP_algorithm;  
  
entity Beispiel is  
    ...  
end entity Beispiel;  
  
architecture a of Beispiel is  
    ....  
    BSP_algorithm(...);  
end architecture a;
```



Vordefinierte Packages: Beispiele

STD_LOGIC_1164

- Erweiterung des VHDL-Standards um IEEE Standard 1164
- Wichtigstes Element: Multi-Value-Logic (std_logic)

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

NUMERIC_STD

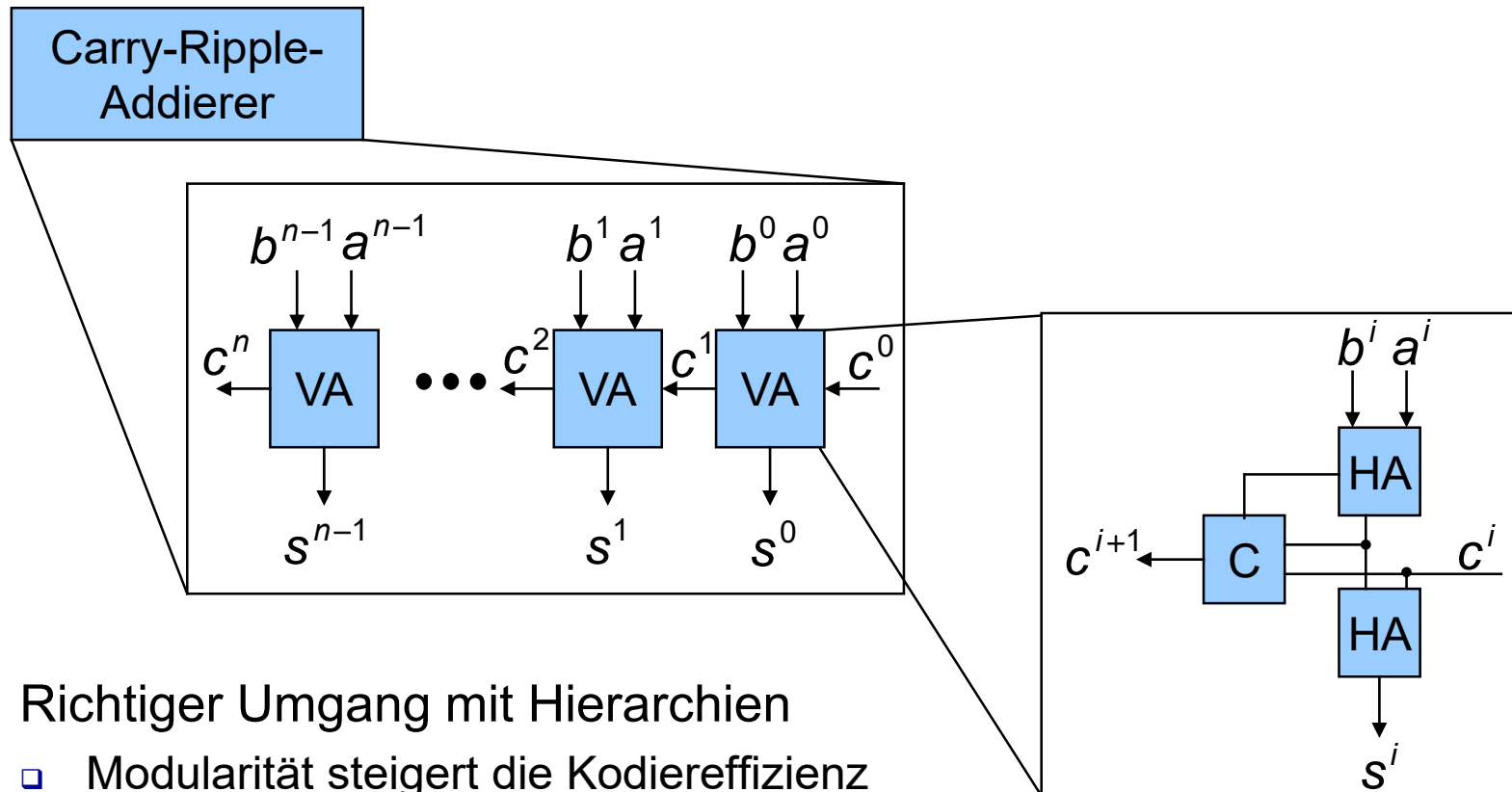
- Deklaration von Vektoren von std_logic
 - vorzeichenlos (unsigned)
 - vorzeichenbehaftet (signed)
- Definition von arithmetische, logische und Schiebeoperatoren
- Definition von Funktionen für die Datentypkonvertierung

```
library IEEE;  
use IEEE.numeric_std.all;
```

Funktionale vs. strukturelle Beschreibung

- **Funktionale Beschreibung** (*behavioral description*):
Was soll das System machen?
- **Strukturelle Beschreibung** (*structural description*):
Wie soll diese Funktion erreicht werden? Wie ist die Struktur des Systems?
- *Coding for synthesis vs. coding for simulation*
 - Einige Elemente von VHDL sind nicht synthetisierbar
 - Eine Beschreibungsart, die für eine effiziente Simulation sorgt, mag zu einer ineffizienten HW-Architektur führen
 - Komplexe Hierarchien in strukturellen Beschreibungen mögen in die Hardware resultieren, die gewünscht wird, stellen allerdings hohe Ansprüche an die Verifikationsumgebung
→ Trade-off zwischen Granularität und Verifikationsgeschwindigkeit

Hierarchie in VHDL-Beschreibungen



- Richtiger Umgang mit Hierarchien
 - Modularität steigert die Kodiereffizienz
 - Spezifikation
 - Verifikation
 - Aber: Nicht bis zur Elementaraussage hinuntergehen!

Datentypen, Signale, Konstanten und Attribute

Datentypen in VHDL I (skalare Standardtypen)

- VHDL ist eine stark typisierte Sprache!
 - Keine implizite Datentypkonvertierung
 - Keine automatische Breitenanpassung
- Skalare (und synthetisierbare) Typen
 - `boolean`: `false` & `true`
 - `integer`
 - Rang: -2^{31} bis $2^{31}-1$
 - Dezimal ist Standard, sonst wird das Zahlenformat angegeben.
Zum Beispiel: `16#cafe#`, `2#011101#`
 - `bit`: die beiden logischen Werte '0' & '1'
 - Aufzählungstypen: für eigens definierte Zustandsmengen
 - unabhängig von technischen Codierungen
 - erhöhen die Lesbarkeit der Beschreibung
 - `type` `wetter_t` `is` (`REGEN`, `SONNE`, `NEBEL`);
 - `type` `fsm_state_t` `is` (`IDLE`, `RUN`, `WAIT`, `COPY`);

Datentypen in VHDL II (skalare Standardtypen)

- Skalare Typen, die ohne Weiteres nicht synthetisierbar sind
 - `character`
 - Entspricht dem ISO 8859-1 Zeichensatz
 - Die Zeichen werden in Hochkommas eingeschlossen, 'a'...'z'
 - `real` (Fließkommazahlen)
 - $-1.0e + 38$ bis $+1.0e + 38$
 - physikalische Maßeinheiten (z.B. `time`)
 - Die Einheiten fungieren als Umrechnungsfaktoren
 - Modellierung von Verzögerungen. Bsp.: `C <= ... after 2 ns;`
- Untertypen, die ebenfalls zum Standard gehören
 - `positive`, alle integer von 1 bis $2^{31}-1$; `natural`, alle von 0 bis $2^{31}-1$
- Andere nicht-synthetisierbare Typen
 - `access`: Zeiger-Typ (wird oft für die Verhaltensmodellierung größerer Speicher verwendet)
 - `file`: für I/O Zwecke in Testbenches

Datentypen in VHDL III (Multi-Value logic)

- Der IEEE 1164 Standard ist im externen *package* `std_logic_1164` definiert und muss explizit eingebunden werden

```
library IEEE;
use IEEE.std_logic_1164.all;
```
- Die Datentypen `std_logic` und `std_ulogic` enthalten mehr Zustände als der Standard-Typ `bit`, um Konflikte durch mehrere Treiber darzustellen & aufzulösen
 - Auflösungsfunktionen gibt es nur bei `std_logic`!
- Für Synthese und Simulation besser geeignet als der Typ `bit`

```
TYPE std_logic is
('U', -- Uninitialized
 'X', -- 0/1? (Konflikt)
 '0', -- 0
 '1', -- 1
 'Z', -- hohe Impedanz
 'W', -- L/H? (Konflikt)
 'L', -- schwache 0 (für
      -- Pull-Down)
 'H', -- schwache 1 (für
      -- Pull-Up)
 '_ ' -- don't care
);
```

Datentypen in VHDL IV (komplexe Typen: Arrays)

- Komplexe Datentypen, die eine reguläre Struktur haben. Diese bestehen aus Elementen des gleichen Datentyps
- Die Größe eines *Arrays* wird durch ein Intervall (`range`) bestimmt. Hierbei muss auf die Richtung (`to/downto`) geachtet werden!
 - beschränkt
 - `type vierzahlen is array (3 downto 0) of integer;`
 - unbeschränkt
 - `type bit_vector is array (natural range <>) of bit;`
 - Bei unbeschränkten Intervallen muss bei Deklaration des Objektes oder des Untertyps das konkrete Intervall definiert werden!
 - `subtype dreibits is bit_vector(0 to 2);`
 - Mehrdimensionale Arrays sind möglich aber nur bedingt empfehlenswert (Code-Lesbarkeit, Simulation- & Syntheseeffizienz)
 - Anstelle der Standard-Typen, `integer` und `bit_vector`, werden für die HW-Beschreibung die Typen `std_logic_vector` und `signed/unsigned` verwendet (`array of std_logic`).

Datentypen in VHDL V (komplexe Typen: Records)

- Records fassen Elemente unterschiedlicher Typen zusammen

- Skalare
- Eigene Typen oder Untertypen (auch komplexe)

- Hauptnutzung: Bildung abstrakter Datenmodelle

```
➤ type MEM_stat_t is  
    record  
        VALID          : boolean;  
        START_ADDR     : std_logic_vector(7 downto 0);  
        NOFWORDS       : integer range 0 to 2**BUS_WIDTH_C-1;  
    end record;
```

- Die Dereferenzierung erfolgt über die Namen der einzelnen Felder

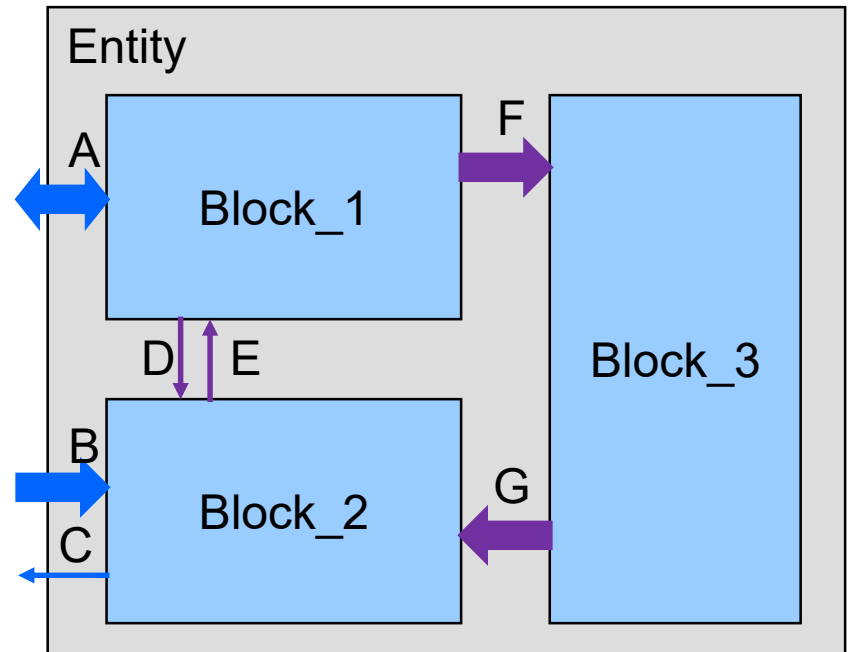
```
➤ -- Bei einem Objekt (z.B. RAM_st) des Typs MEM_stat_t  
RAM_st.VALID      := true;  
RAM_st.BASE_ADDR  := "00010000";  
RAM_st.NOFWORDS   := 0;
```

Bezeichner (*identifizier*)

- *Identifizier* ist der Name von Signalen, *Entities*, Konstanten, etc.
- Regeln
 - Ein *Identifizier* muss in eine Zeile passen
 - Ein *Identifizier* muss mit einem Buchstaben anfangen
 - Ein *Identifizier* kann aus Buchstaben, Zahlen und Unterstrichen bestehen
 - Es dürfen nicht zwei Unterstriche hintereinander, bzw. Unterstriche am Anfang oder Ende eines *Identifiziers* stehen
 - Ein *Identifizier* darf keine Leerzeichen enthalten
 - *Identifizier* sind nicht case-sensitive
 - Reservierte Worte dürfen nicht als *Identifizier* verwendet werden
- Beispiele für gültige *Identifizier*
 - `Multiplex32_nxt`
 - `MEM_ADDR_WIDTH_C`

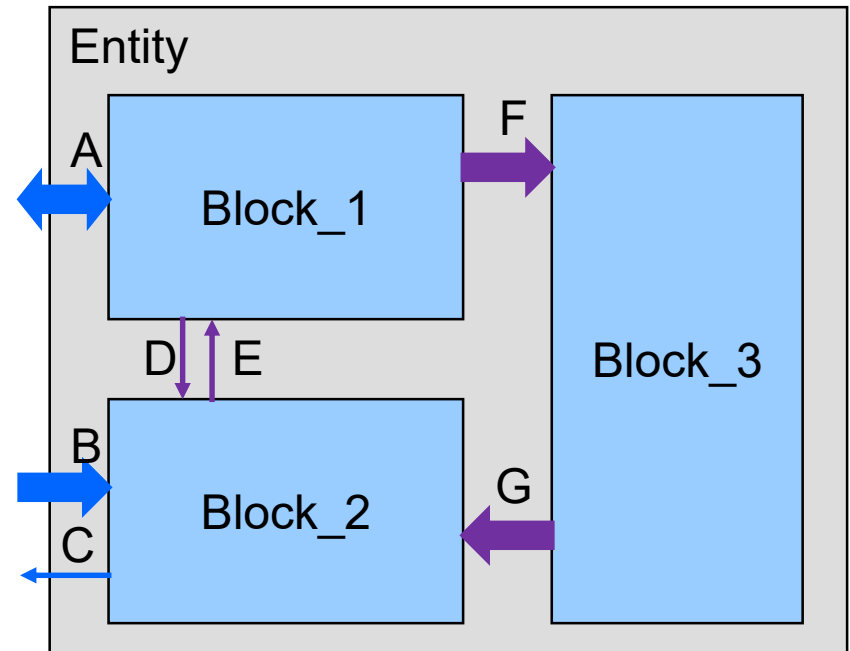
Signale in VHDL: Varianten

- Zwei Varianten abhängig von ihrer Funktion
 - Anbindungen mit der "Umwelt" werden als `port` deklariert (im Bsp. A, B, C)
 - Anbindungen zwischen internen Blöcken werden als `signal` deklariert (im Bsp. D, E, F, G)
- Signale können von jedem Typ sein
 - Für `ports` werden allerdings die Typen `std_(u)logic` bzw. `std_(u)logic_vector` empfohlen



Signale in VHDL: ports

- Die `ports` werden in der Entity deklariert
- In die Deklaration gehören
 - der Name (*identifier*)
 - die Richtung (*mode*)
 - der Typ
- Achtung: Innerhalb der Entity können ihre Ausgangsports nicht gelesen werden!
 - Hilfssignal erforderlich



➤ **Entity** Beispiel **is**

port(

A : **INOUT** std_logic_vector(7 downto 0);

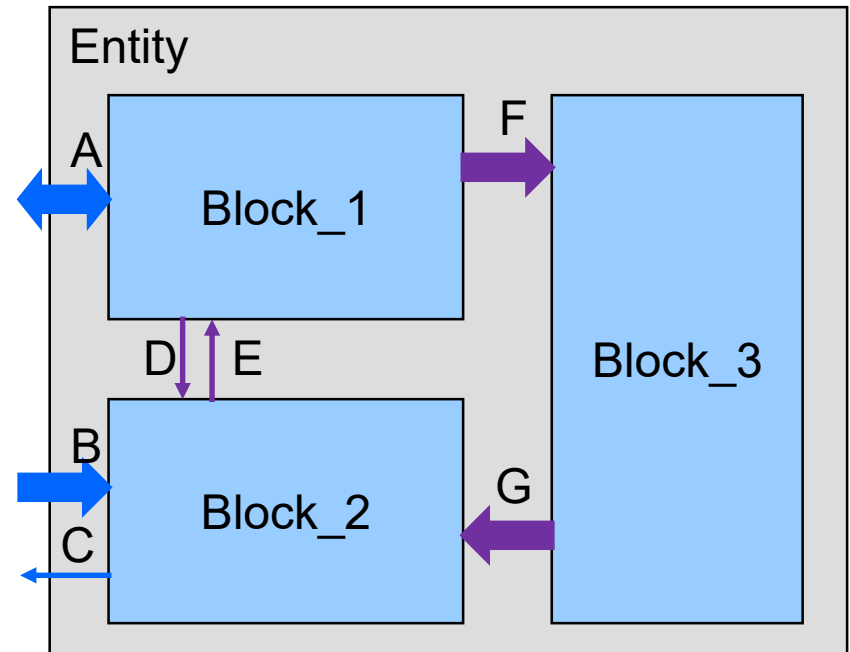
B : **IN** std_logic_vector(7 downto 0);

C : **OUT** std_logic);

end Beispiel;

Signale in VHDL: interne Signale

- Interne Signale dienen zur Kommunikation zwischen Blöcken einer Architektur
- Die Deklaration eines Signals erfolgt in einer *Architecture*, einem Block oder einem Unterprogramm
- Syntax
signal <name>: <daten_typ>;



Architecture arch_1 of Beispiel is
 signal F, G: std_logic_vector(7 **downto** 0);
 signal D, E: std_logic;
begin -- hier fangen die Anweisungen der Architektur an
 . . .
end arch_1;

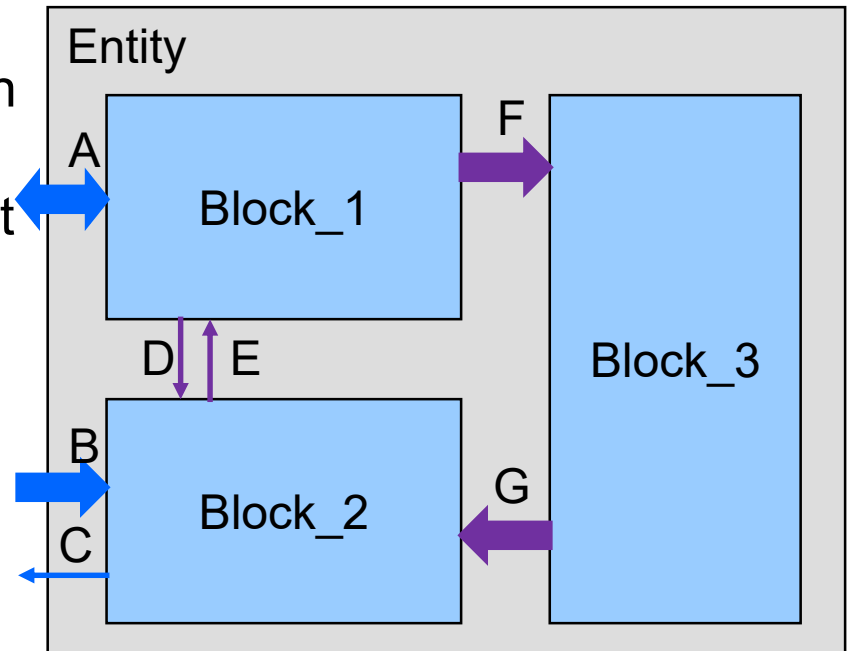
Kommentare werden mit '--' eingeleitet

Hierarchie in VHDL: Einführung

- Um Einheiten in einer Architektur einbinden zu können, muss es ein Platzhalter für sie geben: Ein *component* wird hierfür verwendet
- Syntax

```
component component_name  
  generic (generic_list);  
  port     (port_list);  
end component;
```

```
Architecture arch_1 of Beispiel is  
  component Block_3  
    port (COEFF: IN std_logic_vector(15 downto 0);  
          RES:  OUT std_logic_vector(15 downto 0));  
    . . . -- Deklaration von Signalen und mehr Komponenten  
begin  
  U1: Block_3 port map (COEFF => F, RES => G);  
  . . .  
end arch_1;
```



Signale: Sichtbarkeit

- Die Sichtbarkeit von Signalen hängt von dem Ort der Deklaration ab
 - Ein Signal, das als *Port* in einer *Entity* deklariert wurde, ist in allen *Architectures*, die der *Entity* zugeordnet sind, sichtbar.
 - Ein Signal, das in einer *Architecture* deklariert wird, ist nur in dieser *Architecture* sichtbar.
 - Ein Signal, das in einem Block einer *Architecture* deklariert wird, ist nur in diesem Block sichtbar.
 - Ein Signal, das in einem *Package* deklariert wurde, ist in allen Design-Units sichtbar, die dieses *Package* benutzen.
 - Solche globale Signale sind nicht synthetisierbar, sondern eignen sich nur für die rein funktionale Modellierung. Daher sollten Signale in *Packages* für die HW-Beschreibung nicht deklariert werden!
- Für die HW-Beschreibung wird empfohlen, Signale **nur** im entsprechenden Abschnitt einer *architecture* zu deklarieren! Ausnahmsweise können Signale in Funktionen (Unterprogramme) deklariert und verwendet werden.

Signale: Zuweisungen

- Einem Signal kann ein anderes Signal, eine Variable oder ein fester Wert (Konstante) zugewiesen werden
- Signalzuweisungen erfolgen durch das Symbol **<=**
- Typ und Arraybreite müssen auf beiden Seiten zueinander passen
 - Keine automatische Typkonvertierung
 - Keine automatische Anpassung der Vektorlänge
- Beispiele

```
    signal a, b: std_logic_vector(7 downto 0);
    signal c: integer range 0 to 255;
begin
    a <= b;
    a <= "00100111"; -- alternativ auch a <= x"27";
    b(7 downto 4) <= a(7 downto 4);
    c <= 32;
    b(0) <= '1';
    a <= (others => '1');
    b <= a(3 downto 0) & a(7 downto 4);
```

Konstanten

- Die Werte von Konstanten (`constant`) werden bei der Übersetzung des VHDL-Codes seitens des Synthese-Tools festgelegt
- Vorteile
 - Erhöhen die Lesbarkeit des Codes
 - Module sind portabler für andere Projekte, da sie leichter angepasst werden können
- Konstanten können in einem *Package*, einer *Architecture*, einem Block (*procedure/function*) oder einem Prozess deklariert werden
- Mögliche Funktionen für Konstanten
 - Spezifikation der Größe von komplexen Objekten
 - Kontrolle von Loop-Zählern
 - Definition von Modul-Parametern
- Beispiel
 - `constant loopNumber : integer := 4;`

Attribute

- Attribute werden dafür benutzt, Informationen aus Signalen, Typen oder anderen Objekten verwenden zu können
- Zwei Klassen von attributen: “1076 Standard” und “Custom”
- Beispiele

```
architecture behavior of shifter is
    signal Qreg, Data : std_logic_vector(31 downto 0);
begin
    reg: process(rst,clk)
    begin
        if rst = '1' then    -- Async reset
            Qreg <= (others => '0');
        elsif clk = '1' and clk'event and clk'last_value = '0' then
            Qreg <= Data(Data'left-1 downto Data'right) & Data(Data'left);
        end if;
    end process;
end behavior;
```


Attribute

T is an enumeration, integer, floating or physical type or subtype

T'LEFT is the leftmost value of type T. (Largest if downto)

T'HIGH is the highest value of type T.

T'ASCENDING is boolean true if range of T defined with to .

T'VALUE(X) is a value of type T converted from the string X.

T'VAL(X) is the value of discrete type T at integer position X.

T'PRED(X) is the value of discrete type T that is the predecessor of X.

T'RIGHTOF(X) is the value of discrete type T that is right of X.

Examples:

```
type bit_array is array (1 to 5) of bit;
variable L: integer := bit_array'left; -- L has a value of 1
type state_type is (Init, Hold, Strobe, Read, Idle);
variable P: integer := state_type'pos(Read); -- P has the value of 3
```

Attribute

A is an array signal, variable, constant, type or subtype

A'LEFT(N) is the leftmost subscript of dimension N of array A.

A'RIGHT(N) is the rightmost subscript of dimension N of array A.

A'HIGH(N) is the highest subscript of dimension N of array A.

A'LOW(N) is the lowest subscript of dimension N of array A.

A'RANGE(N) is the range of dimension N of A.

A'REVERSE_RANGE(N) is the REVERSE_RANGE of dimension N of array A.

A'LENGTH(N) is the number of elements of dimension N of array A.

A'ASCENDING(N) is boolean true if dimension N of array A defined with to .

Example:

```
type bit_array is array (15 downto 0) of bit;
variable I: integer := bit_array'left(bit_array'range); -- I has the value 15
```

Attribute

S is a signal

S'STABLE is true if no event is occurring on signal S.

S'QUIET is true if signal S is quiet. (no event this simulation cycle)

S'TRANSACTION is a bit signal, the inverse of previous value each cycle S is active.

S'ACTIVE is true if signal S is active during current simulation cycle.

S'LAST_ACTIVE is the time since signal S was last active.

S'DRIVING is false only if the current driver of S is a null transaction.

Example:

```
wait until Clk = '1' and Clk'event and Clk'last_value = '0';
```

Custom Attribute

```
generic (  
    mem_style_g  : string  := "auto"; -- implementation style of memory cells  
        -- auto | block | distributed | pipe_distributed | block_power1 | block_power2  
    addr_width_g : natural := 6; -- address width (bit) = Log2(reg_count_g)  
    data_width_g : natural := 64 -- data width (bit)  
);
```

```
type      MEM_TYPE is array (((2**addr_width_g) - 1) downto 0) of  
          STD_LOGIC_VECTOR((data_width_g - 1) downto 0);  
  
signal    MEMORY : MEM_TYPE;  
  
attribute ram_style : string;  
attribute ram_style of MEMORY : signal is mem_style_g;
```

Operatoren und numeric_std Funktionen

Operatoren: Einleitung

- Operatoren dienen dazu, ein oder zwei Operanden zu transformieren.
- Die Operanden eines Operators müssen vom selben Datentyp sein
 - Aber, durch Überladen von Operatoren kann ein Operator auch auf Operanden unterschiedlichen Datentyps arbeiten (*packages*)
- Es gibt folgende Klassen von Operatoren
 - Logische Operatoren
 - Relationale Operatoren
 - Schiebe-Operatoren
 - Numerische Operatoren (Additive, Unäre, Multiplikation/Division)
 - Sonstige Operatoren (Verknüpfung)
- Aus praktischen Gründen wird davon ausgegangen, dass die *Packages* `std_logic_1164` und `numeric_std` geladen werden!

Logische Operatoren

- Vorhandene Operatoren

- and/nand, or/nor, xor/xnor, not

- Kompatible Datentypen

- Standard VHDL: `bit`, `bit_vector`, `boolean`
 - `std_logic_1164`: `std_ulogic[_vector]`, `std_logic[_vector]`

- Nutzungsregeln

- Die Operanden und das Ergebnis müssen zum selben Datentyp gehören und ggf. die selbe Breite haben
 - Ausnahme: `std_logic` und `std_ulogic` (kein `_vector`!)
 - Bei Vektoren wird die logische Operation bitweise angewandt

- Beispiele

```
signal a, b, c: std_logic_vector(7 downto 0);
signal wr_en: boolean;
begin
    a      <= b and not(c);
    wr_en <= true;
```

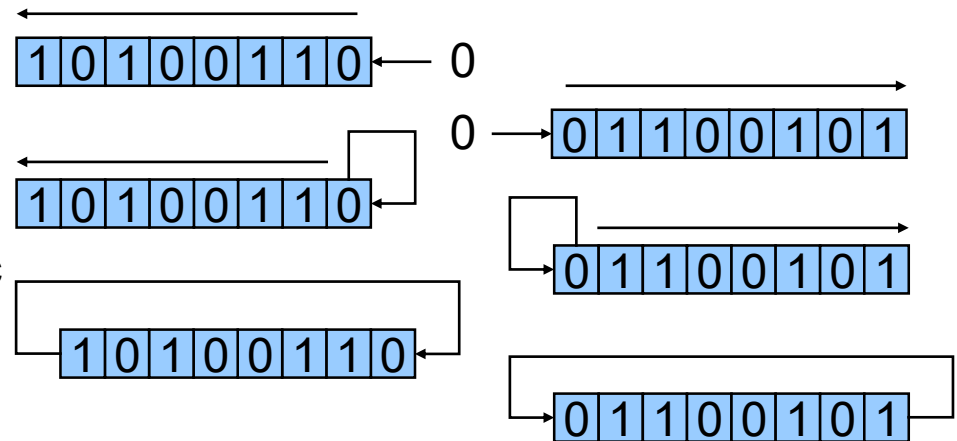
Relationale Operatoren

- Vergleich von 2 Operanden des selben Typs. Ergebnis: `boolean`
- Vorhandene Operatoren
 - gleich `=`
 - ungleich `/=`
 - kleiner `<`
 - größer `>`
 - größer gleich `>=`
 - kleiner gleich `<=`
- Kompatible Datentypen
 - Standard VHDL: `boolean`, `bit`, `character`, `integer`, `real`, `time`, `string` und `bit_vector`
 - `numeric_std`: `unsigned`, `signed` (~~`std_logic_vector`~~ (!))
 - `unsigned` & `signed` dürfen mit `natural` bzw. `integer` verglichen werden
- Hinweis
 - Wenn `bit_vector`en unterschiedlicher Länge verglichen werden, werden sie linksbündig verglichen(!). Bsp.: `(1011 < 110)` gibt ein `true` aus

Schiebe Operatoren

■ Vorhandene Operatoren

- `sll` shift left logical
- `srl` shift right logical
- `sla` *shift left arithmetic*
- `sra` shift right arithmetic
- `rol` rotate left logical
- `ror` rotate right logical



■ Datentypen & Syntax (VHDL-Standard)

`<ziel> <= <quelle> <operator> <Schiebpos>`

- `<ziel>` und `<quelle>` müssen gleiche Typ und Breite haben
 - Nur `bit_vector` oder `arrays` boole'scher Elemente
- `<Schiebpos>` vom Typ `integer` (bei `<0` umgekehrte Richtung)
- `my_bitvec <= your_bitvec sra 3;`

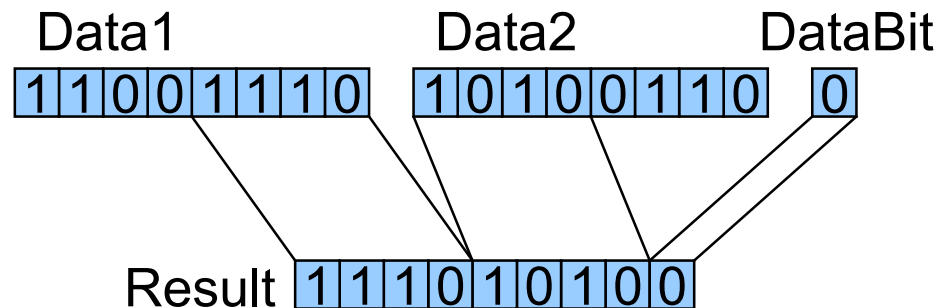
- In `numeric_std` werden `sll`, `srl`, `rol` und `ror` für die Nutzung mit `un-/signed` überladen. Besser: Nutzung vordefinierter Funktionen.

Numerische Operatoren

- Vorhandene Operatoren:
 - Addition (+), Subtraktion (-), Multiplikation (*), Division (/), Modulo (MOD), Rest (REM), Exponent(**) und Absolutwert(ABS)
 - $A \text{ rem } B = A - B * \text{trunc}(A/B)$ (A/B wird gegen Null gerundet)
 - $A \text{ mod } B = A - B * \lfloor A/B \rfloor$ (A/B wird gegen $-\infty$ gerundet)
- Kompatible Datentypen:
 - Standard VHDL: `integer`, `real`, `time`
 - `numeric_std`: `signed`, `unsigned` (~~`std_logic_vector`~~ (!))
- Nutzungsregeln
 - Alle Operatoren und das Ergebnis müssen vom selben Typ sein
 - Ausnahmen: `integer <op> signed` → `signed`
`natural <op> unsigned` → `unsigned`
- Nicht alle numerischen Operatoren sind synthetisierbar! Teilweise werden nur Sonderfälle abgedeckt (z.B. Division nur wenn Quotient eine zweier Potenz als Konstante ist → Schiebeoperation)

Verknüpfungsoperator

- Mit dem Verknüpfungsoperator „&“ können eindimensionale *Arrays* und skalare Elemente desselben Datentyps zu einem *Array* dieses Datentyps kombiniert werden
 - Die Länge des Zielobjekts muss der Summe der Längen aller Operanden entsprechen
- **signal** Data1, Data2 : std_logic_vector (7 **downto** 0);
signal DataBit : std_logic;
signal Result : std_logic_vector (8 **downto** 0);
...
Result <= (Data1(3 **downto** 0) & Data2(7 **downto** 4) & DataBit);



Operatoren: Präzedenz und Kombination

- Die zur Verfügung stehenden Operatoren dürfen kombiniert werden, vorausgesetzt alle Typ- und Breitevorgaben werden eingehalten

Operatorklasse						
Logisch	and	or	nand	nor	xor	xnor
Relational	=	/=	<	<=	>	>=
Schieben	sll	srl	sla	sra	rol	ror
Additiv	+	-	&			
Unär	+	-				
Multiplikativ	*	/	mod	rem		
Sonstiges	**	abs	not			



- Bei gleicher Klasse wird die Anweisung von links nach rechts "gelesen"
- Zur besseren Lesbarkeit wird die Nutzung von Klammern empfohlen.
- Beispiel

`not X & Y xor Z rol 1` \leftrightarrow `((not X) & Y) xor (Z rol 1)`

Vordefinierte Funktionen von `numeric_std`

- Einige Operatoren des Standard-packages sind für unsigned/signed Typen nicht kompatibel
 - `numeric_std` hat Funktionen für das Schieben und Breitenänderungen
- Funktionen
 - `shift_left`, `shift_right`, `rotate_left`, `rotate_right`, `resize`
- Syntax
 - `<ergebnis> <= <funktion>(<quelle>, <parameter>);`
 - `<quelle>` ist vom Typ unsigned oder signed
 - `<parameter>` (Typ: natural)
 - `<ergebnis>` hat denselben Typ wie `<quelle>`
 - Bei *resize* hat `<ergebnis>` die Breite `<parameter>`
 - Bei signed in *shift_right* und *resize* wird das Vorzeichen berücksichtigt
- Beispiele

```
your_byte <= rotate_left(my_byte, 4); -- nibble swap
my_halfword <= resize(my_byte, 16);    -- copy byte to 16-bit signal
```

Konvertierungsfunktionen aus numeric_std

- Notwendigkeit
 - Nur wenige Standard-Operatoren sind für std_logic/std_logic_vector gültig, obwohl diese Typen "de facto" Standards sind.
 - VHDL ist eine stark typisierte Sprache
- Vorhandene Funktionen

Syntax	Quelle (Typ oder Typen)	Ergebnistyp
signed(<quelle>)	unsigned oder std_logic_vector	signed
unsigned(<quelle>)	signed oder std_logic_vector	unsigned
std_logic_vector(<quelle>)	unsigned oder signed	std_logic_vector
to_integer(<quelle>)	unsigned oder signed	integer
to_unsigned(<quelle>, <breite>)	natural	unsigned
to_signed(<quelle>, <breite>)	integer	signed

- Beispiele

```
eine_zahl    <= to_integer(call_police);    -- z.B. "110" wird zu -2
vier_stellen <= to_unsigned(eine_sechs, 4); -- z.B. 6 wird zu "0110"
wdata_slv    <= std_logic_vector(to_unsigned(wdata, wdata_slv'length));
```

Nebenläufigkeit in VHDL, Prozesse und erweiterte Signalzuweisungen

Nebenläufigkeit: Einleitung

- Hardware-Systeme arbeiten nicht sequentiell, sondern bestehen aus parallel arbeitenden, d.h. nebenläufigen, Teilsystemen
- Diese nebenläufigen Teilsysteme können entweder
 - eigenständige Einheiten sein (entities), die instanziiert werden können, um eine hierarchische Beschreibung des Systems zu erhalten,
 - sequentiell ausgeführte Prozesse darstellen oder
 - Anweisungen, die sich direkt im Körper der Architektur befinden
- Die Reihenfolge einzelner Anweisungen auf Architekturebene, Instanziierungen oder Prozesse hat daher keinen Einfluss auf das Verhalten

```
> architecture behave of simpel is  
    signal a, b, c, z1, z2: std_logic;  
begin  
    z1 <= a and b;          z2 <= c or z1;  
    z2 <= c or z1;          z1 <= a and b;  
end behave;
```


Nebenläufigkeit: Beispiel

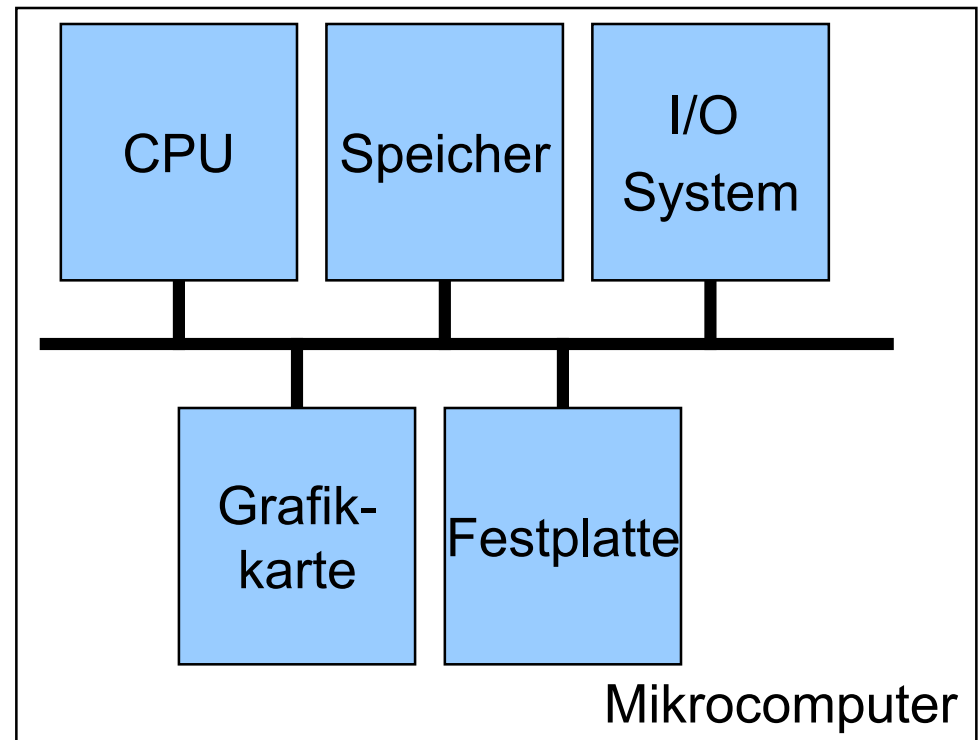
```
architecture mixed of MikroComputer is
  signal dataBus: std_logic_vector(13 downto 0);
begin

  CPU_i: CPU
    port map (
      data => dataBus,
      ...);

  Speicher: process (sList)
begin
    ...
  end process Speicher;

  -- Zuweisungen für I/O
  dataIO <= dataBus
    when data_en = '1'
    else (others => 'Z');
  dataBus_r <= dataIO;

end architecture mixed;
```



Prozess: Einleitung

- Prozesse dienen der Modellierung des Verhaltens eines Systems
- Sie sind eine Liste sequentieller Anweisungen (vgl. „normale“ Programmiersprachen)
- Syntax

```
name: process (sensitivity_list)  
    declarations  
begin  
    sequential_statements  
end process name;
```

- name: Prozessname (optional, aber empfohlen)
- sensitivity_list: hier werden **alle** Signale eingetragen, deren Wertänderung die Ausführung des Prozesses bewirkt
- declarations: Hier werden Variablen und Konstanten deklariert
- sequential_statements: Alle sequentiell auszuführende Prozess-Anweisungen stehen zwischen `begin` und `end process`

Prozess: Beispiel

```
architecture RTL of clippixel is
    signal signalIn, signalOut: integer range 0 to 255;
    . . .
begin
    . . .
    clipping : process (signalIn)
        constant threshold : integer := 20;
    begin
        signalOut <= signalIn;           -- Standard-Wert
        if signalIn < threshold then
            signalOut <= threshold;
        end if;
    end process clipping;
    . . .
end RTL;
```

Prozess: Ausführung und Sensitivity-List

- Ein Prozess wird **in der Simulation** nur dann ausgeführt, wenn sich ein Signal, das in der *Sensitivity-List* des Prozesses steht, ändert
- Achtung: bei der Synthese spielt die *Sensitivity-List* meistens keine Rolle!
 - Kombinatorische Prozesse werden so synthetisiert, als ob alle Signale, auf die **lesend** zugegriffen wird, in der *Sensitivity-List* stehen würden
 - Wenn ein Signal vergessen wird, kann das zu Unterschiede zwischen Simulation und Synthese führen!

■ Beispiel

```
signal a, b, c : integer;  
...  
process (a)  
begin  
    c <= a + b;  
end;
```



	t ₀	t ₁	t ₂	t ₃
a :	1	2	2	0
b :	1	1	2	2
c (sim) :	2	3	3	2
c (syn) :	2	3	4	2

Prozess: Wait-Anweisung

- Die Ausführung des Prozesses wird unterbrochen bis die in der wait-Anweisung angegebene Bedingung erfüllt wird
- Arten der wait-Anweisungen
 - Warten bis eine gewisse System-Zeit vorüber ist (`wait for`)
 - Warten bis ein bestimmtes Ereignis eintritt (`wait until`)
 - Warten bis sich ein Signal ändert (`wait on`)
 - Eine "wait on" Anweisung mit mehreren Signalen am Anfang des Prozesses entspricht dem Verhalten einer Sensitivity-List
- Wait-Anweisungen sind im Allgemeinen nicht synthetisierbar. Daher sollten sie **ausschließlich für Simulationsmodelle oder Testbenches** verwendet werden!

```
> rst_gen : PROCESS    -- Leere Sensitivity-List (sofort ausführen)
BEGIN
    reset <= '1';      -- Aktiviere den Reset
    WAIT FOR 100 ns;   -- Warte bis 100 ns der Simulation vergangen sind
    reset <= '0';      -- Deaktiviere den Reset
    WAIT;              -- Warte unendlich
END PROCESS umr_rst_gen;
```

Prozess: Signalverhalten

- Eigenschaften
 - Signale dürfen in Prozessen **nicht** deklariert werden
 - Signalzuweisungen sind **erst am Ende des Prozesses wirksam**
 - Bei mehreren Zuweisungen zu einem Signal **ist nur die letzte gültig**
- Konsequenz: Nutzungsempfehlungen
 - **Default-Werte** sollten den Signalen in kombinatorischen Prozessen am Prozessanfang zugewiesen werden
 - Signale, denen in einem kombinatorischen Prozess ein Wert zugewiesen wird, sollten **nicht im selben Prozess "gelesen"** werden

```
process (a, c)
begin
  c <= 2 + a;
  b <= 1;
  if (c = 2) then
    b <= 4;
    c <= 3;
  end if;
end process;
```

t_0 **Zu $t_1 \rightarrow$**

a :	1	0	0	0	0
b :	1	1	4	1	4
c :	3	2	3	2	3

...

Simulationsabbruch

Prozess: Variablen

- Variablen sind **keine** Signale, obwohl Operationen und Zuweisungen mit bzw. von/zu Signalen möglich sind, und zwar mit denselben Datentypbeschränkungen wie bei Signalen
- Eigenschaften
 - Variablen sind auf Prozesse beschränkt und auch nur in dem Prozess sichtbar. Die Deklaration erfolgt mit dem Schlüsselwort `variable` im Deklarationsteil eines Prozesses
 - Variablen ändern ihren Wert sofort nach der Zuweisung
 - Zuweisungen zu Variablen erfolgen mit dem Symbol `:=`
 - Variablen sollten **nur** verwendet werden, um den Wert eines komplexen Ausdrucks, der mehrfach gebraucht wird, zu speichern

```
process ( ... )  
    variable logic_expr: boolean := false; -- Standard-Wert (optional)  
begin  
    logic_expr := not(X1) and (Y0 nor Z); -- := weil Variable links  
    sign1 <= logic_expr;                 -- sign1/2 sind Signale  
    sign2 <= logic_expr and cond;        -- cond auch (oder Ports)  
end process;
```

Prozess: Konstrukte

- In VHDL gibt es 4 Strukturen, mit denen die Ausführung von Anweisungen im Prozess gesteuert werden kann
 - Bedingte Ausführung von Anweisungen (if...then)
 - Bedingte Ausführung von Anweisungen mit Alternativen (if...then...else bzw. if...then...elsif)
 - Case-Anweisungen
 - Loops: wiederholte Ausführung einer oder mehrerer Anweisungen (while...loop bzw. for...loop)

```
ExProc:process(sList)
begin
    if cond1 then
        ...
        case cond2 is
            when val1 => ...
            when val2 => ...
            when others =>
                for i in 1 to 4 loop
                    ...
                end loop;
            end case;
        else -- not(cond1)
            while cond3 loop
                ...
            end loop;
            ...
        end if;
    end process ExProc;
```


Prozess: If-else-Anweisungen

- Bei If-else-Anweisungen werden boolesche Ausdrücke analysiert bis einer den Wert "true" ergibt. Die darunter geschriebene Anweisungen werden dann ausgeführt
- Es ist erlaubt mehrere If-Anweisungen zu verschachteln
- Syntax

```
if condition then  
    sequential statements  
[elsif condition then  
    sequential statements ]  
[else  
    sequential statements ]  
end if;
```

- Beispiel (Prozessauszug, wo sigA Teil der Sensitivity-List ist)

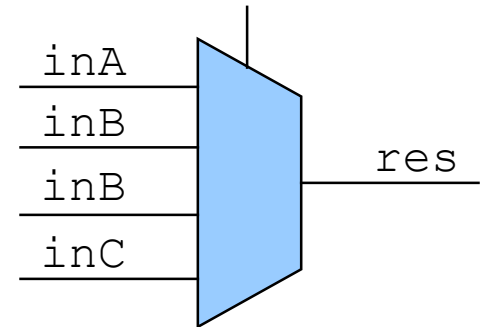
```
if sigA = '1' then                -- wenn sigA den Wert '1' hat  
    sigC <= "01";                -- bekommt sigC den Wert "01"  
else                             -- sonst ist sigA '0','X','Z',...  
    sigC <= "11";                -- dann erhält sigC den Wert "11"  
end if;  -- d.h., sigC(1) <= not(sigA) und sigC(0) <= '1'
```

Prozess: Case-Anweisungen

- If-Anweisungen mit mehreren Optionen und Verschachtelungen sind schnell unleserlich. Case-Anweisungen schaffen hier Ordnung

- Beispiel

```
□ Multiplexer: process (sel, inA, inB, inC)          sel(1:0)
begin
    case sel(1 downto 0) is
        when "00"          => res <= inA;
        when "01"|"10"    => res <= inB;
        when "11"          => res <= inC;
        when others      => res <= 'X';
    end case;
end process Multiplexer;
```



- Jede Möglichkeit muss eindeutig sein (keine Überlappungen)
- Auch wenn *when others* nicht verwendet wird, müssen alle Fälle abgedeckt werden (bei std_[u]logic sind auch X,Z,... mögliche Fälle!)
- case-Konstrukte eignen sich besonders für die Beschreibung von Endzustandsautomaten (FSM), optimal mit eigenem type

Prozesse: Loop-Anweisungen I

■ Syntax

```
[ loop_label :]iteration_scheme loop
  sequential statements
  [next [label] [when condition];
  [exit [label] [when condition];
end loop [loop_label];
```

■ Es gibt 3 mögliche Iterationsschemata

- normale Schleife (keine *iteration_scheme*). Wird durch *exit* verlassen
- while-loop: *iteration_scheme* ::= **while** *condition*
 - Wenn *condition* nicht erfüllt wird, wird die Schleife verlassen
- for-loop: *iteration_scheme* ::= **for** *identifizier* **in** *range*
 - Der *identifizier* wird automatisch deklariert und ist nur in der Schleife verwendbar
 - Die *range* muss aus integer bestehen und beim "Kompilieren" fest sein

■ Loops sind nur unter bestimmten Bedingungen synthetisierbar!

Prozesse: Loop-Anweisungen II

- Es wird empfohlen, auf Loop-Anweisungen in HW-Beschreibungen zu verzichten, da sie nur bedingt zu einer effizienten Hardware führen und eine hohe VHDL-Expertise erfordern
 - Ausnahmebeispiel: Bei der Wertzuweisung von/zu Arrays kann vom Einsatz eines for-loops profitiert werden

■ Beispiel

```
signal DataBus : std_logic_vector(DataBusWidth-1 downto 0);
signal ones    : integer range 0 to DataBusWidth;
begin
    ...
    CountOnes: process(DataBus)
        variable NumOfOnes : integer range 0 to DataBusWidth;
    begin
        NumOfOnes := 0;
        for Cntr in DataBus'range loop    -- Dank range, portabler Code
            next when DataBus(Cntr) = '0';
            NumOfOnes := NumOfOnes + 1;
        end loop;
        ones <= NumOfOnes;
    end process;
```

Erweiterte Signalzuweisungen

- Signalzuweisungen oder logische Funktionen müssen nicht unbedingt in einem Prozess beschrieben werden → zu aufwändig
- Stattdessen können sie direkt in der Architektur als nebenläufige Zuweisungen geschrieben werden → sog. vereinfachte Prozesse
- Vereinfachte Prozesse gibt es für
 - einfache Operatoren
 - bedingte Zuweisungen
 - ausgewählte Zuweisungen

```
signal a,b,c,d : std_logic;
begin -- Anfang der Architektur
  gate1: process (a,b)
  begin
    d <= a and b;
  end process gate1;
  gate2: process (d,c)
  begin
    e <= c or d;
  end process gate2;
```



```
signal a,b,c,d : std_logic;
begin -- Anfang der Architektur

  d <= a and b;
  e <= c or d;
  -- Gleiches Verhalten und HW
```

Bedingte Zuweisungen

■ Syntax

```
target <= expr1 when cond1 else  
      [expr2 when cond2 else]-- beliebig erweiterbar  
      exprN when others;
```

- Dem Signal *target* können andere Signale oder nebenläufige Ausdrücke zugewiesen werden (keine Verschachtelung erlaubt!)

```
architecture proc of example is  
  signal a, b, Z: std_logic;  
  signal x: unsigned(3 downto 0);  
begin  
  Sel: process (a, b, x)  
  begin  
    if (x = "1111") then  
      Z <= a;  
    elsif (x > "1000") then  
      Z <= b;  
    else  
      Z <= '0';  
    end if;  
  end process Sel;
```

=

```
Z <= a when (x="1111") else  
      b when (x>"1000") else  
      '0' when others;
```

Ausgewählte Zuweisungen

■ Syntax

```
with sel_expr select
    target <= src1 when choice1,
    [src2 when choice2,] -- beliebig erweiterbar
    srcN when others;
```

- Dem Signal *target* wird eine Quelle zugewiesen abhängig vom Wert des Auswahlausdrucks *sel_expr*

```
architecture proc of example is
    signal a, b, Z: std_logic;
    signal x: integer range 0 to 15;
begin
    Sel: process (a, b, x)
        case x is
            when 15 =>
                Z <= a;
            when 8 to 14 =>
                Z <= b;
            when others =>
                Z <= '0';
        end case;
    end process Sel;
```

=

```
with x select
    Z <= a when 15,
    b when 8 to 14,
    '0' when others;
```

Beispiele, Tipps, Tricks & Pitfalls

Beispiel: taktflankengesteuertes D-Flipflop

- Bei der VHDL-Beschreibung taktflankengesteuerter Elemente ist auf die Syntax und auf die Sensitivity-List besonders zu achten
- Beispiel: D-FF mit asynchronem Reset

```
dff1: process (clk, reset)
begin
    if reset='0' then
        Q <= (others => '0'); -- Einfacher bei breite Vektoren
    elsif clk='1' and clk'event then
        Q <= D;                -- Q & D haben denselben Datentyp
    end if;
end process dff1;
```

- Der Eingang ist nicht in der Sensitivity-List, denn nur wenn sich der Reset oder der Takt ändern, soll der Prozess ausgeführt werden
- Der Ausdruck (clk='1' and clk'event) bezeichnet die Taktflanke ('1' bei der steigenden, '0' bei der fallenden)
 - Alternativ kann der Ausdruck rising_edge(clk) bzw. falling_edge(clk) verwendet werden. Keine Konsequenzen bei HW, aber die Simulation ist konsistenter mit der Hardware

Beispiel: taktflankengesteuertes D-Flipflop II

- Beispiel: D-FF mit *enable*

```
dff2: process (clk)
begin
    if clk='1' and clk'event then
        if dff_en = '1' then
            Q <= D;
        end if;
    end if;
end process dff1;
```

- In diesem Beispiel reagiert der Prozess ausschließlich auf eine Änderung im Taktsignal, da kein Reset vorgesehen ist
- Nur wenn im Moment der steigenden Taktflanke das Enable-Signal aktiv ist, erhält Q den Wert von D, sonst behält Q seinen Wert, ungeachtet dessen, was mit D passiert
 - Die Beschreibung entspricht somit dem Verhalten eines taktflankengesteuertes D-FF mit Enable

VHDL für FPGA-Entwurf: Nutzungsrichtlinien

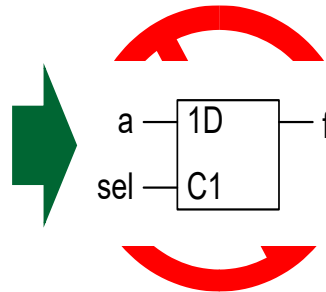
- In einem **Schaltwerk** sollten die **Speicherelemente** und das **Schaltnetz** in getrennten Prozessen geschrieben werden
 - Bessere Lesbarkeit, einfachere Fehleranalyse
- Im Bezug auf **kombinatorische** Prozesse
 - Alle Signale, die einen Wert zugewiesen bekommen (Ausgangssignale), müssen unter jeder möglichen Eingangsbedingung einen Wert erhalten
 - Sonst werden ungewollt pegelgesteuerte Speicherelemente (Latch) erzeugt! Abhilfe: Immer **Default-Zuweisungen** am Prozessbeginn
 - Kein Ausgangssignal sollte im Prozess gelesen werden
 - Bei Unvorsichtigkeit besteht die Gefahr einer kombinatorischen Rückkopplung
- Im Bezug auf **sequenzielle** Prozesse
 - Sequenzielle Prozesse dürfen nur von einem **einzigen** Takt und evtl. auch von asynchronen Reset/Set-Signalen abhängig sein
 - Lokale Taktsignale sollen **nie** erzeugt bzw. verwendet werden
 - *Enables* erfüllen mit Sicherheit ebenfalls den gewünschten Zweck

Tricks & Pitfalls: Unvollständige if/case-Konstrukte

- Sind if/else bzw. case-Konstrukte unvollständig, werden die Ausgangswerte für die nicht spezifizierten Zustände gehalten
 - pegelgesteuerte Flip-Flops (Latches) werden synthetisiert, die in einem FPGA-Entwurf meistens Probleme verursachen
- Entweder wird dem Ausgangssignal ein spezifikationskonformer Wert gegeben, oder, wie bei *std_logic* möglich, ein *don't care*
- Beispiel

```
signal sel : std_logic;
signal a   : std_logic;
signal f   : std_logic;

begin
  process (sel, a)
  begin
    if (sel = '1') then
      f <= a;
    end if;
  end process;
```

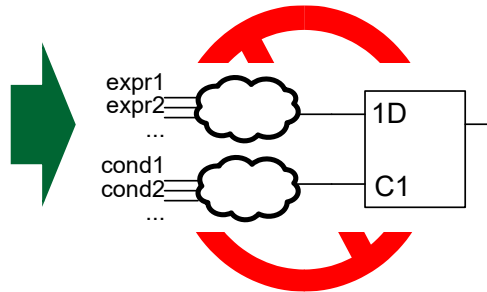


```
process (sel, a)
begin
  f <= '-'; ← Default-Wert
  if (sel = '1') then
    f <= a;
  end if;
end process;
```

Tricks & Pitfalls: Halten von Signalen

- Wenn bestimmte Zustände zu einer Änderung des zu speichernden Werts führen sollen und sonst der Wert gespeichert werden soll:
 - außerhalb des sequenziellen Prozesses dem zukünftigen Signalwert den aktuellen Signalwert standardmäßig zuweisen
 - oder mit einem erzeugten *Enable*-Signal eine Übernahme des neuen Wertes steuern
- Beispiel

```
begin
  process (SensList)
  begin
    if cond1 then
      f <= expr1;
    elsif cond2 then
      f <= expr2;
    ... -- kein else?
    ... -- f <= f?
    end if;
  end process;
```

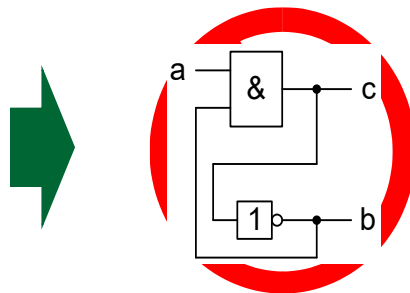


```
begin
  process (f, SensList)
  begin
    f_next <= f;
    if cond1 then
      f_next <= expr1;
    elsif cond2 then
      f_next <= expr2;
    end if;
  end process;
  process (clk)
  begin
    if rising_edge(clk) then
      f <= f_next;
    end if;
  end process;
```

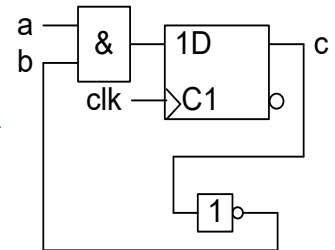
Tricks & Pitfalls: kombinatorische Rückkopplungen

- Können entstehen, wenn das Verhalten von Signalen in Prozessen nicht richtig interpretiert werden
 - Ausgangssignale wurden im selben Prozess gelesen
 - Unter bestimmten Bedingungen kommt es dann zu einer Änderung des ursprünglich als Eingang interpretierten Signals, was bei komplexen Prozessen schon unbemerkt passieren kann
- Hardware-Oszillatoren können dadurch entstehen
- Auftrennen mit Flipflops erforderlich!

```
process (a, b)
begin
  ...
  c <= a and b;
end process;
process (c)
begin
  ...
  b <= not c;
end process;
```



```
process (clk)
begin
  if rising_edge(clk) then
    c <= a and b;
  end if;
end process;
process (c)
begin
  b <= not c;
end process;
```



Strukturelle Beschreibung und Simulation

Strukturelle Beschreibung: Einleitung

- Die strukturelle VHDL-Beschreibung eines Systems definiert, wie ein System aufgebaut ist und wie die Komponenten und/oder Prozesse miteinander und mit der Umwelt verbunden sind
- Strukturelle Beschreibung erlaubt mehrere Hierarchieebenen
 - Eine Entity kann aus mehreren Komponenten bestehen, die wiederum strukturell oder verhaltensmäßig beschrieben worden sind
- In einer rein strukturellen Beschreibung gibt es keine Verhaltensbeschreibungen
 - Keine Prozesse
 - Keine komplexen nebenläufigen Ausdrücke
 - Die tiefste Hierarchieebene besteht aus technologischen Primitiven
- Der ideale VHDL-Entwurf ist eine Mischung aus struktureller (Top- und Subsystem-Ebene) und funktionaler Beschreibung

Strukturelle Beschreibung: Komponentendeklaration

- Um Entities in einer Architektur einbinden zu können, muss es ein Platzhalter für sie geben: Eine *component* wird hierfür verwendet. Die Deklaration befindet sich

- entweder im deklarativen Abschnitt der instanzierenden Architektur
- oder in einem eingebundenen *Package*

- Syntax

```
component component_name
    generic (generic_list);
    port      (port_list);
end component [component_name]; --Name hier optional
```

- Eine *component* sollte denselben Namen tragen wie die entsprechende *entity* und über dieselben Schnittstellen (im Namen und Datentyp) verfügen
 - Nur so ist eine automatische Zuordnung seitens des Tools möglich

Strukturelle Beschreibung: Komponenteninstanziierung

- In der *architecture* werden die deklarierten Komponenten instanziiert und über Signale mit den restlichen Bestandteilen der Architektur verbunden
 - Bei passender Richtung (und Typ und Breite) ist eine direkte Verbindung mit den Ports der instanziiierenden *entity* erlaubt
- Ein *component* darf mehrmals instanziiert werden
- Syntax

```
instance_label: component_name
    [generic map (generic_map_aspect)] -- Kein ';' !
    [port map (port_map_aspect)];
```

- Ein *map_aspect* ist eine mit Kommata getrennte Liste, wo den *component*-Schnittstellen bestimmte in der instanziiierenden *Architecture* sichtbare Objekte zugewiesen werden
- Werden den *generics* keine Werte zugewiesen, gelten die bei der Entity-Deklaration angegebenen Werte

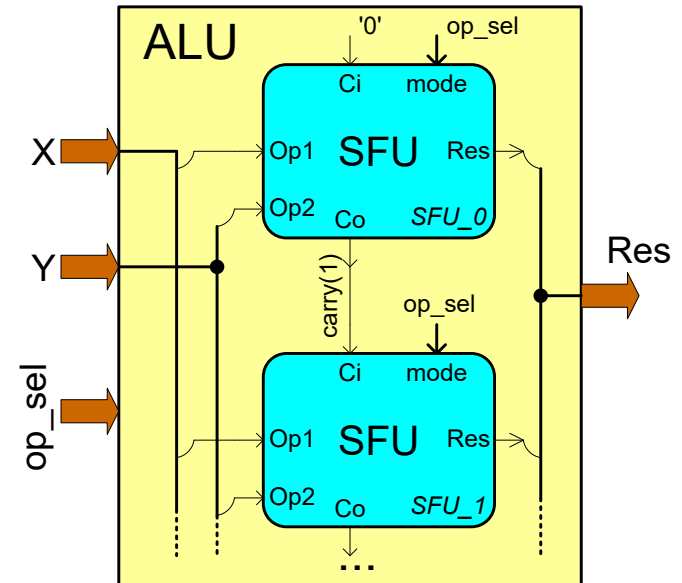
Strukturelle Beschreibung: Beispiel

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
entity ALU is
  port (
    X, Y      : in  std_logic_vector(7 downto 0);
    Res       : out std_logic_vector(7 downto 0);
    op_sel    : in  std_logic_vector(3 downto 0));
end ALU;
architecture struct of ALU is
  component SFU is
    port (
      Op1, Op2, Ci : in  std_logic;
      mode         : in  std_logic_vector(3 downto 0);
      Res, Co      : out std_logic);
  end component;
  constant LOW: std_logic := '0';
  signal carry: std_logic_vector(X'range);
begin
  SFU_0: SFU port map (X(0), Y(0), LOW, op_sel, Res(0), carry(1));
  SFU_1: SFU port map (Res=>Res(1), Ci=>carry(1), Op1=>X(1),
    Op2=>Y(1), Co=>carry(2), mode=>op_sel);
  ...
end struct;

```

Die Instanzbezeichnung muss eindeutig sein !



Zuweisung über Position

Zuweisung über Name

Strukturelle Beschreibung: *generate*-Ausdrücke

- Die sog. *generate*-Ausdrücke erhöhen die Lesbarkeit und die Wiederverwendbarkeit einer VHDL-Modulbeschreibung
 - Bei hoher HW-Redundanz, z.B. viele Instanzen eines Moduls, kann ein *for-generate*-Konstrukt die Lesbarkeit erhöhen
 - Bei hochparametrisierten Beschreibungen, z.B. wenn die Top-Ebene einer Systembeschreibung mehrerer Implementierungsalternativen beinhalten soll, ist ein *if-generate* Konstrukt unerlässlich
- Beispiel (Bezug auf das ALU/SFU-Beispiel)

```
carry(0) <= LOW;
SFU_gen: for i in X'range generate --X'range == 7 downto 0
  SFUlsb_gen: if i/=X'length-1 generate
    SFU: SFU port map (Op1 => X(i), Op2 => Y(i), Ci => carry(i),
      mode => op_sel, Res => Res(i), Co => carry(i+1));
  end generate SFUlsb_gen;
  SFUmsb_gen: if i=X'length-1 generate--(noch) kein else generate erlaubt
    SFU: SFU port map (Op1 => X(i), Op2 => Y(i), Ci => carry(i),
      mode => op_sel, Res => Res(i), Co => open);
  end generate SFUmsb_gen;                                -- open == Ausgangsport offen
end generate SFU_gen;
```

Strukturelle Beschreibung: configuration

- Eine sog. *configuration* ist erforderlich
 - Beim Vorhandensein mehrerer Architekturen für ein *entity*, da sonst die Synthese- bzw. Simulationstools sich irgendeine Architektur aussuchen
 - Bei Unterschieden zwischen Komponenten und den entsprechenden *entities*, da sonst die Werkzeuge keine Hierarchie aufbauen können
- Die *configuration* kann nach der *architecture* oder in einer separaten Datei definiert werden
- Werden nur bei komplexen Systemhierarchien mit mehreren Abstraktionsebenen oder Realisierungsalternativen gebraucht
- Beispiel

```
configuration ALU_cfg1 of ALU is
  for struct -- Name der zu konfigurierenden Architektur
    for ALL: SFU -- Wenn instanzspezifisch, kein ALL (SFU_0)
      use ENTITY work.SFU(behavioral); -- library.ent(arch)
    end for;
  end for;
end configuration;
```

Simulation: Einleitung

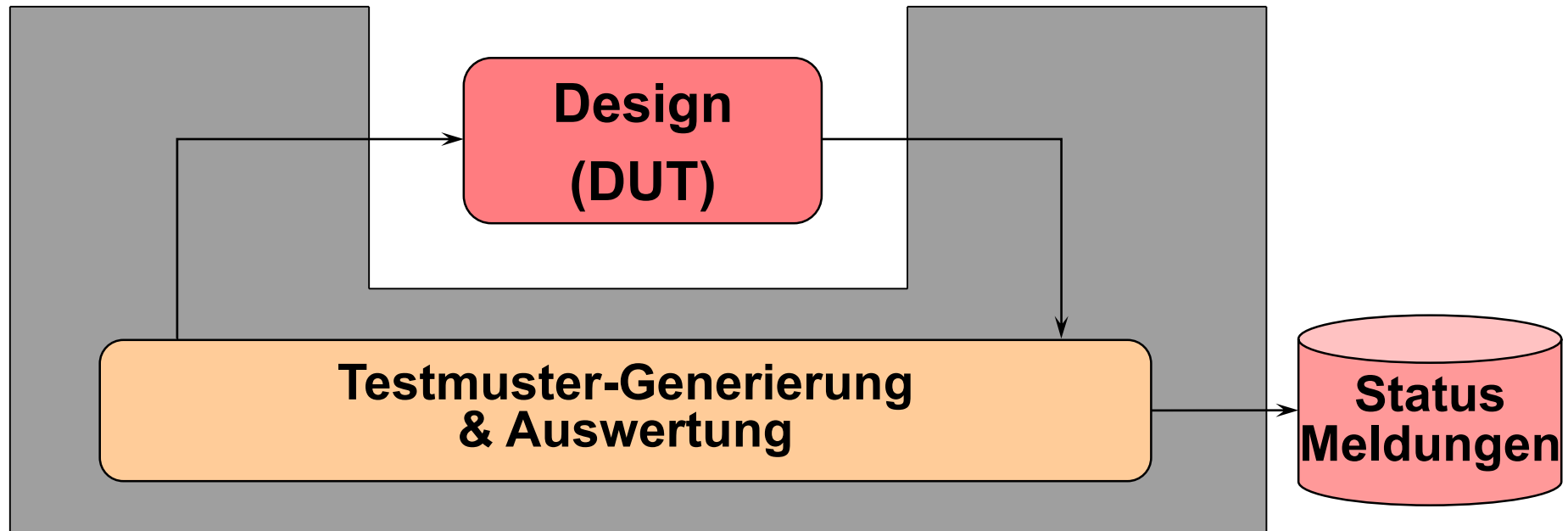
- Eine verhaltensorientierte VHDL-Beschreibung lässt sich mit einem Simulationswerkzeug kompilieren und relativ effizient simulieren
- Um ein in VHDL beschriebenes System simulieren zu können, ist eine sog. *Testbench* erforderlich, die das zu testende System instanziiert und mit geeigneten Testvektoren versorgt
 - Einige FPGA-Entwicklungsumgebungen erstellen automatisch die Testbench als VHDL-Datei, die bereits die Instanziierung des DUT (*Design under Test*) und evtl. die Takterzeugung beinhaltet
- Die Simulation kann auf unterschiedlicher Art durchgeführt werden
 - Nur die *Funktionalität* der Beschreibung wird überprüft, ohne auf die technologiespezifische Implementierung zu achten
 - Die *Timing*-Eigenschaften der fertig synthetisierten Schaltung werden in die Simulation einbezogen → zeitaufwändiger

Simulation: Testbench

- Eigenschaften einer VHDL-Testbench
 - Die Entity hat keine Ein- oder Ausgänge
 - In der Architektur werden das zu verifizierende (Sub-)System und sonstige erforderliche Modelle (z.B. Speichermodelle) instanziiert
 - Der VHDL-Code einer Testbench muss sich nicht an irgendwelche synthesespezifische Beschreibungsrichtlinien halten
 - File I/O und print-Ausgaben sind möglich (Package textio)
 - wait-Anweisungen dürfen verwendet werden
 - Nachteil: Der Implementierungsaufwand einer Testbench kann u.a. aufgrund der strengen VHDL-Syntax erheblich sein
- Verifikationsstrategien
 - Testvektoren: externes Programm erstellt eine Datei mit Eingangs- und erwarteten Ausgangsdaten, mit denen die Simulation verglichen wird
 - Selbstcheckende verhaltensorientente Testbench-Beschreibung
 - Anbindung über Simulationstoolabhängige FLIs (*Foreign Language Interface*) mit einer in C geschriebenen Testumgebung

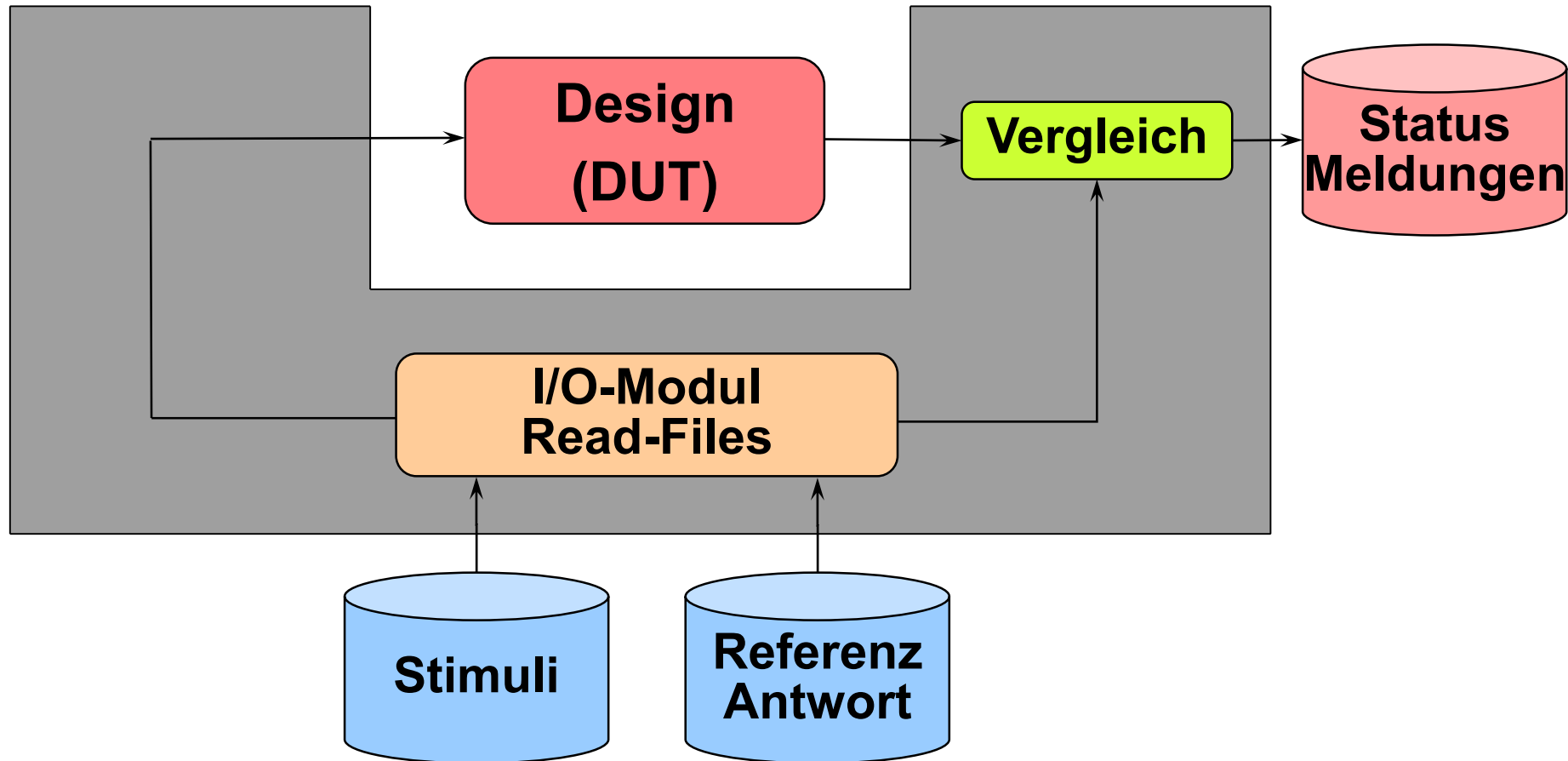
Simulation: Testbench mit interner Referenz

- Erzeugung der Stimuli und Auswertung in VHDL codiert



Simulation: Testbench mit externer Referenz

- Stimuli und Referenzdaten aus Datei gelesen
- Stimuli- und Referenzerzeugung: externe Tools (Script o.ä.)



Simulation: Einfache Stimuli Erzeugung

```
entity my_tb is
end entity my_tb;

architecture tb of my_tb is
  component adder is
    port (
      clk : in  std_logic;
      a   : in  std_logic_vector(7 downto 0);
      b   : in  std_logic_vector(7 downto 0);
      sum : out std_logic_vector(7 downto 0)
    );
  end component adder;

  signal sys_clk : std_logic := 0;
  signal sys_a   : std_logic_vector(7 downto 0);
  signal sys_b   : std_logic_vector(7 downto 0);
  signal sys_sum : std_logic_vector(7 downto 0);

begin

  sys_clk <= not sys_clk after 5 ns;

  STIMULI: process
  begin
    sys_a <= "00001001";
    sys_b <= "01001001";
    wait for 11 ns
    assert (sys_sum = "01010010") report "Simulation error" severity failure;
    assert false report "Simulation stop" severity failure;
  end process;

  DUT : adder
    port map (clk => sys_clk, a => sys_a, b => sys_b, sum => sys_sum);

end architecture tb;
```

**Typischer clock-Signal
Generator**

Stimuli Zuweisung

Instanzierung DUT

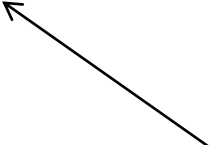
Simulation: Stimuli Erzeugung aus Datei

```
entity my_tb is
end entity my_tb;

architecture tb of my_tb is
  component adder is
    port (
      clk : in  std_logic;
      a   : in  std_logic_vector(7 downto 0);
      b   : in  std_logic_vector(7 downto 0);
      sum : out std_logic_vector(7 downto 0)
    );
  end component adder;
  signal sys_clk : std_logic := 0;
  signal sys_a, sys_b, sys_sum, ref_sum : std_logic_vector(7 downto 0);
  file in_file: TEXT open read_mode is "stimuli.txt";
begin
  sys_clk <= not sys_clk after 5 ns;
  FILEIO: process
    variable in_line: line
    variable in_str: string(7 downto 0);
  begin
    wait for 1 ns
    while not endfile(in_file) loop
      readline(in_file, in_line);
      read(in_line, in_str);
      sys_a <= to_std_logic_vector(in_str);
      sys_b <= to_std_logic_vector(in_str);
      ref_sum <= to_std_logic_vector(in_str);
      wait for 10 ns
      assert (sys_sum /= ref_sum) report "Simulation error" severity failure;
      assert sys_sum = ref_sum report "Simulation correct" severity note;
    end loop;
  end process;
  DUT : adder
    port map (clk => sys_clk, a => sys_a, b => sys_b, sum => sys_sum);
end architecture tb;
```

stimuli.txt

```
000000010
000001010
000001100
...
```



Unterprogramme in VHDL: Funktionen und Prozeduren

- Wie bei SW-Programmiersprachen, ist in VHDL die Deklaration und Benutzung von Unterprogrammen möglich
- Eigenschaften
 - Die Ausführung erfolgt sequenziell
 - Funktionen geben einen Wert zurück (Prozeduren nicht)
 - Orte der Deklaration
 - In Packages
 - Im deklarativen Teil des Packages werden nur die Prototypen der Unterprogramme deklariert (analog zu einem Header-File in C)
 - Im Körper des Pakets werden die Unterprogramme beschrieben
 - In Architekturen werden die Unterprogramme direkt beschrieben
- Nutzungshinweise
 - Unterprogramme sind hauptsächlich nur bei der Beschreibung reiner Simulationsmodelle oder bei Test-Umgebungen empfehlenswert
 - In der HW-Beschreibung sollten Funktionen nur da definiert werden, wo VHDL an ihre Grenzen stößt

Unterprogramme in VHDL: Beispiele

```
function parity(D: std_logic_vector) return std_logic is  
    variable result: std_logic := '0';  
    begin  
        for i in D'range loop  
            result := result xor D(i);  
        end loop;  
    return result;  
end parity;
```

-- *parity* wird dann in einem process oder in einer nebenläufige Anweisung verwendet
-- Eine Kombination (und Rekursion) ist möglich, solange die Typen stimmen
vector_parity <= parity(input_vector); --vector_parity hat den Typ std_logic

-- Aufgrund der evtl. Signalkonflikte, sollten Procedures nur als Debug-Möglichkeit
-- verwendet werden (z.B., ASSERTs unter bestimmten Bedingungen)

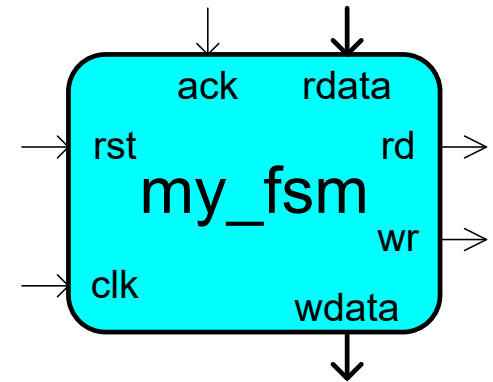
```
PROCEDURE output_note(str : IN string; err : IN boolean := false) IS  
BEGIN  
    IF (err) THEN  
        ASSERT false REPORT str SEVERITY ERROR;  
    ELSE  
        ASSERT false REPORT str SEVERITY NOTE;  
    END IF;  
END output_note;
```

Effizienter VHDL-Entwurf für FPGA-Architekturen

VHDL-Beispiele für FPGAs: FSMs (I)

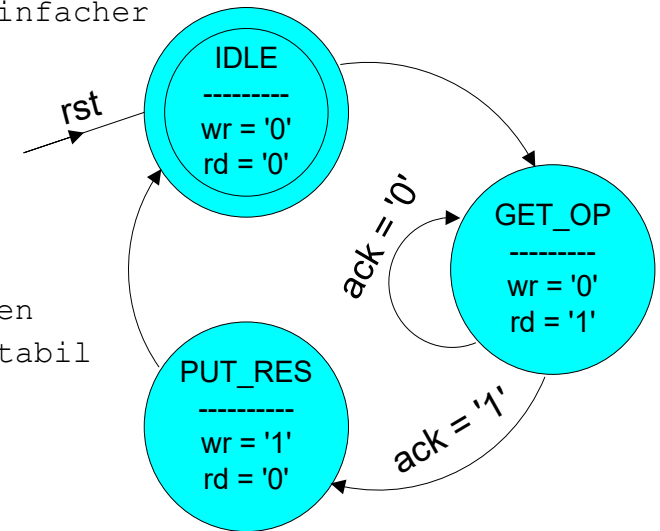
```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
entity my_fsm is
    generic (DATA_WIDTH : positive := 8);
    port (
        rst, clk : in  std_logic;
        rd, wr    : out std_logic;
        ack       : in  std_logic;
        rdata     : in  std_logic_vector(DATA_WIDTH-1 downto 0);
        wdata     : out std_logic_vector(DATA_WIDTH-1 downto 0));
end my_fsm;
```

```
architecture beh of my_fsm is
    type fsm_t is (IDLE, GET_OP, PUT_RES);
    signal stat, stat_nxt: fsm_t;
    signal data_r, data_nxt: std_logic_vector(wdata'range);
begin
    seq: process(clk, rst)
    begin
        if rst='1' then
            stat    <= IDLE;
            data_r <= (others => '0');
        elsif rising_edge(clk) then
            stat    <= stat_nxt;
            data_r <= data_nxt;
        end if;
    end process seq;
```



VHDL-Beispiele für FPGAs: FSMs (II)

```
fsm_comb: process (stat, ack, rdata, data_r)
begin
  stat_nxt <= stat;    -- Wenn nicht anders gewollt, bleibt man im selben Zustand
  data_nxt <= data_r;  -- Register behält standardmäßig seinen Wert
  rd      <= '0';      -- Dank der Spezifizierung der Default-Werte, sind die
  wr      <= '0';      -- ...Zustandsbeschreibungen einfacher
  wdata   <= (others => '-');
  case stat is
    when IDLE => -- Start- und Wartezustand
      stat_nxt <= GET_OP;
    when GET_OP =>
      rd      <= '1';
      data_nxt <= rdata; -- FF wird immer überschrieben
      if ack='1' then -- erst hier ist das Datum stabil
        stat_nxt <= PUT_RES;
      end if;
    when PUT_RES =>
      wr      <= '1';
      wdata   <= data_r; -- Oder etwas anderes in Funktion von data_r
      stat_nxt <= IDLE;
    when others => null; -- nicht zwingend, da alle Zustände kodiert wurden
  end process comb;
end beh;
```



VHDL-Beispiele für FPGAs: Schieberegister (FFs)

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity shift_reg is
  generic (TAPS: integer := 32);
  port (
    clk   : in   std_logic;
    shift : in   std_logic;
    d_in  : in   std_logic;
    d_out : out std_logic_vector(TAPS-1 downto 0));
end entity shift_reg;

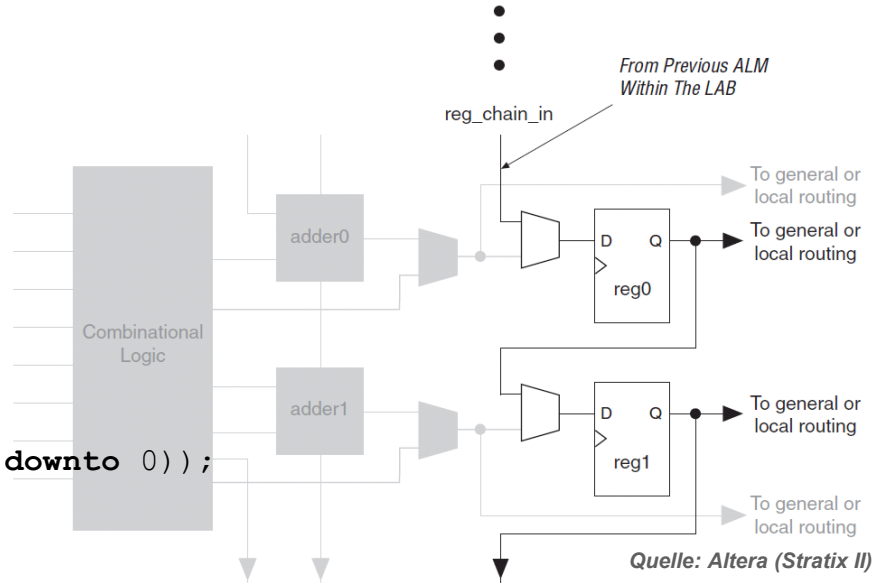
architecture FFs of shift_reg is
  signal shift_ff: std_logic_vector(TAPS-1 downto 0) := (others => '1');
  -- Wenn Initialisierung in der Signal-Deklaration, nur bei FPGAs !

begin
  process (clk)
  begin
    if rising_edge(clk) then
      if shift = '1' then
        shift_ff <= d_in & shift_ff(TAPS-1 downto 1); --1-Mal nach rechts schieben
      end if;
    end if;
  end process;

  d_out <= shift_ff; --Der parallele Lesevorgang erzwingt die Nutzung von Flip-Flops
end FFs;

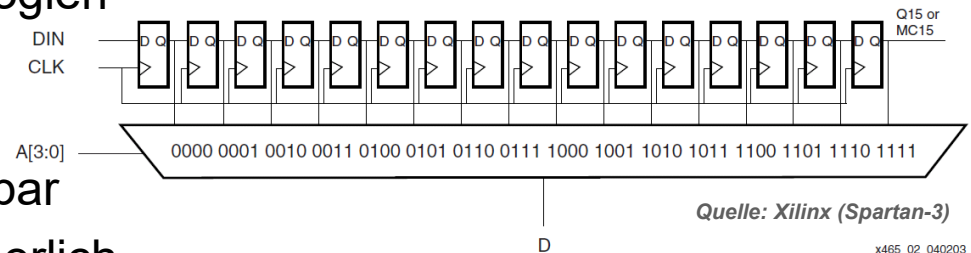
```

Quelle: Altera (Stratix II)



Schieberegister-Implementierung (FFs vs. LUT/RAM)

- Eigenschaften eines Schieberegisters mit FFs
 - Evtl. hoher Bedarf einer "teuren" Ressource
 - Überdimensionierte Flexibilität: werden alle Positionen zeitgleich gebraucht?
- Lösung I: LUTs als Schieberegister
 - Nicht bei allen FPGA-Technologien
 - Asynchroner Lesevorgang mit LUT-Eingängen
 - nur 1 Position pro Takt lesbar
 - Zusätzlicher FF evtl. erforderlich
- Lösung II: dedizierte Speicherblöcke
 - Wenn Tiefe x Breite zur Größe (und Verhalten zur Beschreibung) passt
 - Der Lesevorgang ist hier synchron (evtl. unerwünscht)
- strenge VHDL-Richtlinien für beide Lösungen
 - alternativ: technologieabhängige Instanziierung von Primitiven ☹



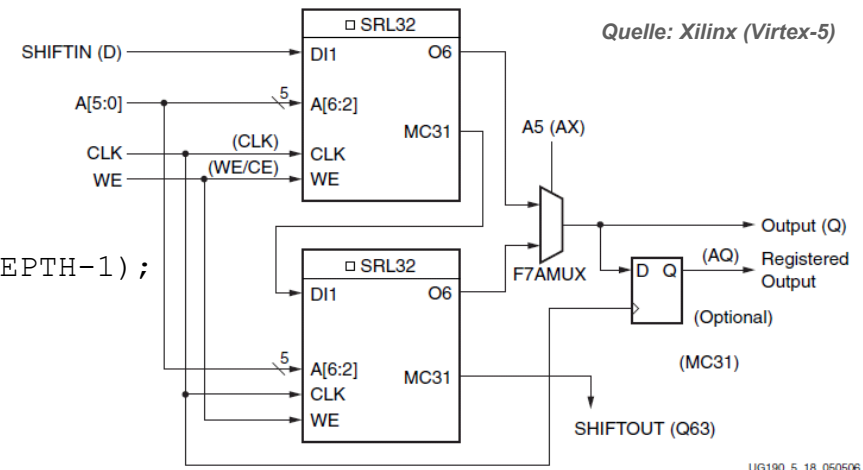
VHDL-Beispiele für FPGAs: Schieberegister (LUTs)

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

entity shift_reg is
  generic (MAX_DEPTH: integer := 64);
  port (
    taps : in  std_logic_vector(BITS(MAX_DEPTH-1)-1 downto 0);
    clk  : in  std_logic;
    shift : in  std_logic;
    d_in  : in  std_logic;
    d_out : out std_logic);
end entity shift_reg;
```

```
architecture LUTs of shift_reg is
  signal shift_ff: std_logic_vector(0 to MAX_DEPTH-1);
begin
  process (clk)
  begin
    if rising_edge(clk) then
      if shift = '1' then
        shift_ff <= d_in & shift_ff(0 to MAX_DEPTH-2);
      end if;
    end if;
  end process;
  d_out <= shift_ff(to_integer(unsigned(taps)));
end LUTs;
```

```
FUNCTION BITS(n : natural) RETURN natural IS
BEGIN
  IF n = 1 THEN RETURN 1;
  ELSE RETURN (1+BITS(n/2));
  END IF;
END FUNCTION BITS;
```



VHDL-Beispiele für FPGAs: Speicherblöcke - Einleitung

- In den meisten aktuellen FPGAs gibt es dedizierte Speicherblöcke mit unterschiedlichen Größen und erlaubten Modi
- Die FPGA-Hersteller empfehlen die Nutzung ihrer "Core-Generatoren" für die Erstellung fertig kodierter VHDL-Komponenten
 - Wenig portabel, da FPGA- und Tool-abhängig
 - Die Simulation erfordert die Kompilierung zusätzlicher Bibliotheken
- Lösung → Generische VHDL-Beschreibung
- Kodierungshinweise
 - Das Verhalten des Leseports während eines Schreibvorgangs soll so "locker" angegeben wie von der Spezifikation erlaubt (*don't cares*)
 - Bei Problemen: Die erlaubte Beschreibungsart wird in der Dokumentation der Synthese-SW für VHDL und Verilog angegeben
 - Synthese-Tools fügen "Weiterleitungslogik" zwischen 2 Ports hinzu, was bei unterschiedlichen Taktdomänen zu Fehlfunktionen führt
 - Lösung: Nutzung von sog. VHDL-Attributen (z.B. `syn_ramstyle`)

VHDL-Beispiele für FPGAs: Speicherblöcke (Bsp. 1)

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
entity mem_rw is
    generic (AWIDTH : positive := 9;
            DWIDTH : positive := 64);
    port (clk      : IN  std_logic;
          we       : IN  std_logic;
          addr     : IN  std_logic_vector(AWIDTH-1 downto 0);
          wdata    : IN  std_logic_vector(DWIDTH-1 downto 0);
          rdata    : OUT std_logic_vector(DWIDTH-1 downto 0));
end entity mem_rw;
architecture old_data of mem_rw is
    type ram_t IS ARRAY(0 to 2**AWIDTH-1) of std_logic_vector(DWIDTH-1 downto 0);
    signal ram: ram_t;
begin
    process (clk)
    begin
        if clk'event and clk='1' then
            rdata <= ram(to_integer(unsigned(addr))); -- Es wird das alte Datum gelesen
            if we = '1' then
                ram(to_integer(unsigned(addr))) <= wdata;
                -- rdata <= (others => '-') oder <= wdata?? -- Welches Leseverhalten
                -- wird beim Schreiben erwünscht?
            end if;
        end if;
    end process;
end old_data;
```

VHDL-Beispiele für FPGAs: Speicherblöcke (Bsp. 2)

```
entity mem_r_w is
    generic (AWIDTH : positive := 9; DWIDTH : positive := 64);
    port (clk, we      : IN  std_logic;
          waddr, raddr : IN  std_logic_vector(AWIDTH-1 downto 0);
          wdata        : IN  std_logic_vector(DWIDTH-1 downto 0);
          rdata        : OUT std_logic_vector(DWIDTH-1 downto 0));
end entity mem_r_w;
architecture forward of mem_r_w is
    signal raddr_reg: std_logic_vector(raddr'range);
    type ram_t IS ARRAY(0 to 2**AWIDTH-1) of std_logic_vector(wdata'range);
    signal ram: ram_t;
begin
    process (clk)
    begin
        if clk'event and clk='1' then
            raddr_reg <= raddr; -- Diese Register werden ins Speicherblock "optimiert"
            if we = '1' then
                ram(to_integer(unsigned(waddr))) <= wdata;
            end if;
        end if;
    end process; -- Durch das Registern der Leseadresse wird das asynchrone Lesen...
    rdata <= ram(to_integer(unsigned(raddr_reg))); --...in ein synchrones "umgewandelt"
end forward;
-- Bei Übereinstimmung der Adressen erscheint an rdata das neue Datum (nach dem Takt)
-- Evtl. addiert die SW zusätzliche HW, was mit einem Attribut verhindert werden kann
-- attribute ramstyle: string; -- Deklaration des Attributs (nur 1x pro Arch.)
-- attribute ramstyle of ram: signal is "no_rw_check"; -- Spezifikation (1xpro Signal)
```

VHDL-Beispiele für FPGAs: Speicherblöcke (Bsp. 3 I)

- Die meisten der dedizierten Speicherblöcke unterstützen *echtes* dual-port Verhalten (true dual-port), ggf. mit verschiedenen Takten
 - Das sog. "mixed read-during-write" Verhalten ist nicht spezifiziert!
 - Eine Simulation würde das "richtige" Datum liefern
 - Die Hardware wird evtl. ein ungültiges Datum liefern (Mischung von alten und neuen Daten)
 - Nicht alle Technologien unterstützen es (VHDL weniger universell)
- Beispiel

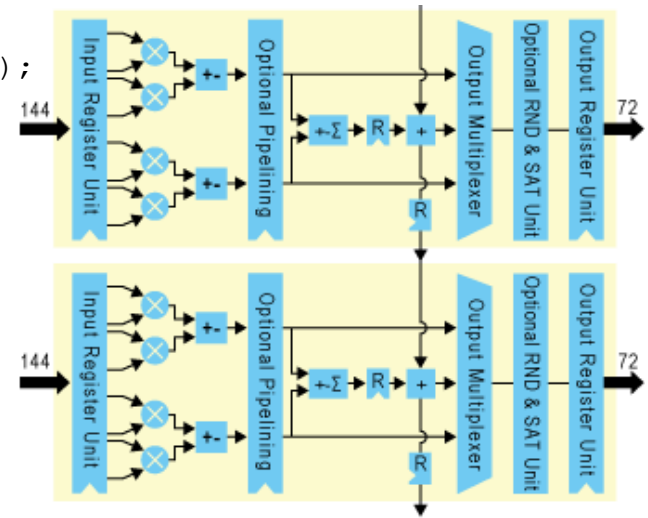
```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
entity mem_rw_rw is
    generic (AWIDTH : positive := 9;
            DWIDTH : positive := 64);
    port (clk_a, clk_b      : IN  std_logic;
          we_a, we_b       : IN  std_logic;
          addr_a, addr_b   : IN  std_logic_vector(AWIDTH-1 downto 0);
          wdata_a, wdata_b : IN  std_logic_vector(DWIDTH-1 downto 0);
          rdata_a, rdata_b : OUT std_logic_vector(DWIDTH-1 downto 0));
end entity mem_rw_rw; -- Fortsetzung auf nächster Seite
```

VHDL-Beispiele für FPGAs: Speicherblöcke (Bsp. 3 II)

```
architecture fpga of ram_test IS
    type ram_t IS ARRAY(0 to 2**AWIDTH-1) of std_logic_vector(DWIDTH-1 downto 0);
    signal ram: ram_t;
begin
    process (clk_a)
    begin
        if clk_a'event and clk_a='1' then
            rdata_a <= ram(to_integer(unsigned(addr_a)));
            if we_a = '1' then
                ram(to_integer(unsigned(addr_a))) <= wdata_a;
                rdata_a <= wdata_a;
            end if;
        end if;
    end process;
    process (clk_b)
    begin
        if clk_b'event and clk_b='1' then
            rdata_b <= ram(to_integer(unsigned(addr_b)));
            if we_b = '1' then
                ram(to_integer(unsigned(addr_b))) <= wdata_b;
                rdata_b <= wdata_b;
            end if;
        end if;
    end process;
end fpga;
-- Das Signal ram wird in 2 Prozessen gleichzeitig geschrieben. std_logic_vector
-- erlaubt dies als Typ. Aber das hier ist nur eine Ausnahme!!
```


VHDL-Beispiele für FPGAs: DSPs

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
ENTITY mult_dsp IS
    PORT (clk          : IN  std_logic;
          a, b         : IN  std_logic_vector(7 DOWNTO 0);
          rst_n        : IN  std_logic;
          accum_out    : OUT std_logic_vector(15 DOWNTO 0));
END mult_dsp;
ARCHITECTURE rtl OF mult_dsp IS
    SIGNAL a_reg, b_reg : signed(7 DOWNTO 0);
    SIGNAL adder_out    : signed(15 DOWNTO 0);
BEGIN
    PROCESS (clk, rst_n)
    BEGIN
        IF (rst_n = '1') THEN
            a_reg    <= (OTHERS => '0');
            b_reg    <= (OTHERS => '0');
            adder_out <= (OTHERS => '0');
        ELSIF (clk'event AND clk = '1') THEN
            a_reg    <= signed(a);
            b_reg    <= signed(b);
            adder_out <= adder_out + a_reg*b_reg; -- Wenn der DSP eine Multiplikation und
        END IF; -- Akkumulation bei der Bit-Breiten unterstützt, erkennt die Synthese-SW
    END PROCESS; -- die Operation und instantziiert in der Synthese den DSP
    accum_out <= std_logic_vector(adder_out);
END rtl;
```



Quelle: Altera (Stratix)

Einschränkungen bei VHDL für RAMs & DSPs

- Die dedizierte HW-Blöcke in FPGAs unterstützen viele, oft komplexe Betriebsmodi
 - Diese lassen sich in VHDL beschreiben, werden aber von den Synthese-Tools oft nur begrenzt erkannt
- RAM-Blöcke
 - unterstützen gemischte Wortbreiten, z.B. Port A 64-bit, Port B 32-bit
 - extra-Logik für diesen Zweck wird daher eingespart
 - verfügen meistens über Ports für *byte write enable*
 - lassen sich in manchen FPGA-Familien direkt als FIFOs mit integrierten Zustandssignalen verwenden
- DSPs
 - integrieren Rundung- & Saturierungslogik
- Für eine kleine und schnelle HW, ist die Nutzung toolspezifischer Core-Generatoren erforderlich
 - Nur dann empfehlenswert, wenn das Projekt dies fordert