Systementwurf mit VHDL

Sommersemester 2017

Sergei Sawitzki

saw@fh-wedel.de

FH Wedel (University of Applied Sciences)

Systementwurf mit VHDL: Vorlesungs-Gesamtübersicht

Systementwurf mit VHDL: Gliederung der 1. Vorlesung

1. Einleitung

Lernziele

- Ergänzung und Erweiterung der Grundlagen aus zurückliegenden Lehrveranstaltungen (Digitaltechnik 1 und Digitaltechnik 2, Informationstechnik, Rechnerstrukturen)
- Einsicht in Methoden der Beschreibung, des Entwurfs und der Implementierung komplexer digitaler Systeme
- Fähigkeit, digitale Systeme in Hardwarebeschreibungssprachen zu entwerfen (am Beispiel von VHDL)
- Basis für weiterführende Eigenstudien und Vorbereitung auf den Einsatz im Beruf

Kontaktdaten und Einordnung

Zeitplan: Sommer, 1 Vorlesung pro Woche, donnerstags 09:30–10:45 Uhr, Seminarraum 8

Präsentationsform: Beamer, Handouts, Tafel

Kontakt: Sergei Sawitzki, Zimmer 208 (Altbau, im 2. OG), Telefon: (04103)-8048-37, e-Mail: saw@fh-wedel.de

Sprechstunde: Mittwochs 13:00–14:30, sowie nach Vereinbarung

Prüfung: schriftlich (Klausur, Teil des Moduls „Systementwurf mit VHDL“)

Kreditpunkte: 2 (von 5 für das Gesamtmodul)

Fortsetzung: VHDL-Workshop, 2. Hälfte des laufenden Semesters

Zeitraster

April 2017

					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30

Mai 2017

1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31				

Juni 2017

			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30		

Juli 2017

					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31						

Achtung ⚠

Keine Vorlesungen am 18.05.2017 und 25.05.2017!

Feedback und Arbeitsweise

- jederzeit mündlich oder schriftlich (Postfach im Sekretariat), am besten zeitnah an der entsprechenden Vorlesung, damit es noch in der gleichen Vorlesungsperiode berücksichtigt werden kann
- Lehrevaluation am Ende der Vorlesungsperiode

Wie immer gilt: Es ist unser **gemeinsames Anliegen**, das Beste aus dieser Vorlesung zu machen!

Wichtig ⚠

Die Inhalte der Vorlesung sind stufenweise aufeinander abgestimmt und setzen solides Wissen aus der Vorlesung „Digitaltechnik 1“ und „Digitaltechnik 2“ voraus. Daher:

- Die Grundlagen gleich am Anfang wiederholen
- Am Ball bleiben und Unklarheiten zeitnah beseitigen
- Die Inhalte im Selbststudium nacharbeiten

Voraussetzungen aus den zurückliegenden Semestern

. . . insbesondere aus Digitaltechnik 1 und 2:

- Schaltalgebra, Schaltfunktionen, Arbeiten mit schaltalgebraischen Ausdrücken
- Darstellung und Vereinfachung von Schaltfunktionen
- Darstellung, Analyse und Realisierung von Schaltnetzen und Schaltwerken
- Digitale Grundschaltungen
- Speicherelemente
- Schaltwerke
- Zeitverhalten digitaler Schaltungen

Literatur

G. Lehmann, B. Wunder, M. Selz: *Schaltungsdesign mit VHDL*, Franzis' Verlag, 1994

J. Reichardt, B. Schwarz: *VHDL-Synthese: Entwurf digitaler Schaltungen und Systeme*, 6. Auflage, Oldenbourg Verlag, 2012

P. J. Ashenden: *Designer's Guide to VHDL*, 3rd edition, Morgan Kaufmann Publishers, 2008

P. J. Ashenden, J. Lewis: *VHDL-2008 Just the New Stuff*, Elsevier Inc., 2008

Language Reference Manual

And last but not least:

Google und 

(mit kritischer Reflexion!)

Bezeichnungen und Konventionen

Die Vorlesungsunterlagen sind in Kapitel, Abschnitte und Unterabschnitte gegliedert (3 Gliederungsstufen, wobei nicht bei jedem Thema die maximale Gliederungstiefe genutzt wird). Zur Orientierung sind diese im Folienkopf angegeben. Bei einzelnen Präsentationen wird zusätzlich die Lage der aktuellen Seite innerhalb des Abschnitts sowie innerhalb der Gesamtpräsentation als Navigationsleiste angezeigt (diese Anzeige entfällt bei Druck der gesamten Vorlesungsunterlagen aus einer Datei).

Definitionen

sind eingerahmt und farblich hervorgehoben.

Fremdsprachige Begriffe erscheinen in der *Kursivschrift*.

Personennamen sind in Kapitälchen gesetzt (gilt nicht für das Titelblatt, das Literaturverzeichnis und die Maßeinheiten).

Gesetze, Sätze, Lemmata u. ä.

sind eingerahmt und farblich hervorgehoben.

Verfügbarkeit von Vorlesungsunterlagen

Die Präsentationen erscheinen am Tag der Vorlesung auf dem Handout-Server im Unterverzeichnis „`Sawitzki/VHDL`". Zu jeder Vorlesung gibt es

- eine Datei „`vhdl_v<n>.pdf`" (`n` ≙ Vorlesungsnummer) mit Präsentation zum Ansehen (z. B. Adobe Reader ➠ CTRL+L für Vollbildschirmmodus)
- eine Datei „`vhdl_v<n>_print.pdf`" im Unterverzeichnis „`Druckvorlage_4_auf_1`" mit Handout zum Ausdrucken (4 Folien pro Seite)
- eine Datei „`vhdl_v<n>.pdf`" im Unterverzeichnis „`Druckvorlage_A4`" ohne Overlays/Übergänge (für eigenhändige Gestaltung der Druckausgabe, z. B. 2 Folien pro Seite, skaliert, gespiegelt usw.)

Unterlagen aus dem vergangenen Semester

Alle „Systementwurf mit VHDL“-Vorlesungen des letzten Semesters (inklusive Aufgabenlösungen) sind in der Datei „`vhdl_alt.pdf`" unter „`Sawitzki/VHDL`" zu finden.

Überlegenswert: Grundsätzlich können bei AStA die Handouts aus dem letzten Semester käuflich erworben werden.

Zu beachten: Die Präsentationen werden von Semester zu Semester geringfügig angepasst.

➥ Vergessen Sie nicht, die eventuellen Änderungen seitenweise zu ergänzen.

Für konstruktive Verbesserungshinweise (auch wenn es dabei nur um einfache Tippfehler geht) bin ich immer dankbar!

Beispiele und Synthese-Werkzeuge

Die in der Vorlesung besprochenen VHDL-Beispiele sind auf dem Handout-Server im Unterverzeichnis „`VHDL/Beispiele`" zu finden. Als Entwurfssystem wird Quartus Prime (Intel Corp.) eingesetzt. Diese steht als „Lite Edition“ (mit eingeschränktem Funktionsumfang) zum kostenfreien Download unter `www.altera.com`

`Products -> Quartus Prime -> Donwload`

bereit. Von gleicher Quelle kann man ebenfalls kostenfrei Dokumentation herunterladen bzw. einen Online-Tutorial absolvieren. Eine lizenzierte Vollversion der Vorgänger-Software (Quartus II) ist auf mehreren Arbeitsplätzen im CAE-Labor (Zi. 209) installiert und darf (sofern nicht durch andere Lehrveranstaltungen besetzt) jederzeit genutzt werden. Bei Fragen und Problemen wenden Sie sich bitte an Herrn Timm Bostelmann (Zi. 215).

Wichtig ⚠

Regelmäßiges Nacharbeiten von Beispielen aus der Vorlesung ist für einen erfolgreichen Einstieg in VHDL sehr hilfreich.

Simulationssoftware

Innerhalb des Quartus-II-Packets stehen Ihnen unter anderem Werkzeuge zur Simulation und Synthese von VHDL-Beschreibungen zur Verfügung. Als VHDL-Simulator benutzt Quartus-II das Produkt ModelSim der Firma Mentor Graphics. Dieser kann unter `www.altera.com` unter gleichem Pfad wie Quartus-Software

`Products -> Quartus Prime -> Download`

heruntergeladen werden. In der Vorlesung wird Bezug auf diese Produkte genommen (und auch das VHDL-Praktikum baut darauf auf).

In der zur Zeit verfügbaren Version von Quartus-II werden alle VHDL-Standards von 1987 bis 2008 unterstützt, einstellbar über Projekt-Optionen.

Was ist VHDL?

VHDL steht für VHSIC *Hardware Description Language* (wobei VHSIC wiederum für *Very High Speed Integrated* Circuit steht), also eine Hardwarebeschreibungssprache für integrierte Schaltkreise sehr hoher Verarbeitungsgeschwindigkeit *(„What's in a name?“)*.

Hardwarebeschreibungssprachen sind Sprachen für Modellierung, Entwurf und Synthese von Hardware (vergleiche mit gewöhnlichen Programmiersprachen):

Hardware	Software
HDL	höhere Programmiersprache
Schaltplan	Assembler

Wesentlicher Unterschied zur Software: Explizite Einbeziehung des Zeitbegriffs sowie Darstellung von physikalisch-strukturellen Zusammenhängen, z. B. Verzögerungszeiten, Entwurfshierarchie, Unterscheidung zwischen nebenläufigen und sequentiellen Wertzuweisungen.

Kurzverzeichnis von HDL

- VHDL
- Verilog-HDL (mittlerweile erweitert auf System-Verilog)
- ABEL-HDL (heute kaum noch im Einsatz)
- AHDL (*Altera Hardware Description Language*, wenig verbreitet)
- EDIF (*Electronic Design Interchange Format*, nur Netzlisten)
- ...

Bibliotheken und Erweiterungen gewöhnlicher Programmiersprachen zur Beschreibung von Hardware:

- SystemC (basiert auf C++)
- Lava (basiert auf Haskell)
- Handel-C (basiert auf C)
- ...

Wesentliche Merkmale von Hardwarebeschreibungssprachen

- Explizite Einbeziehung des Zeitbegriffs
- Modellierung paralleler Abläufe
- Beschreibungsmittel für verschiedene Abstraktionsebenen
- Datentypen und Strukturen mit einem direkten Hardware-Abbild
- Simulations-orientierte Beschreibungsmittel
- Synthese-orientierte Beschreibungsmittel
- Meistens: Umfangreiche Bibliotheken mit Basiselementen und -funktionen (auch herstellerspezifisch)
- Zielstellung: Darstellung eines Simulationsmodells und/oder eines synthesefähigen Modells der zu entwerfenden Hardware (Simulation erfordert einen HDL-fähigen Simulator)

Einsatzgebiete von Hardwarebeschreibungssprachen

- Modellierung und Simulation von Schaltungen und Systemen
- Entwurfsautomatisierung
- Hardware-Synthese
- Test und Verifikation
- Entwurfsdokumentation
- Entwurfsverwaltung und Entwurfs-Datenaustausch
- Steigerung der Effizienz beim Hardware-Entwurf
- Wiederverwendung von Entwürfen
- Migration von Entwürfen zwischen Entwurfswerkzeugen, Herstellungstechnologien und Schaltkreisfamilien
- Standardisierung und Vereinheitlichung der Entwurfsmethodik
- ...

Historische Entwicklung

1980–82	Erste Diskussionen innnerhalb des VHSIC Projektes (DoD, US-Verteidigungsministerium)
1983	Entwicklungsauftrag durch DoD
1985	Erste offizielle VHDL-Version
1986	Erste kommerzielle Werkzeuge mit VHDL-Unterstützung
1987	Standardisierung durch IEEE, Istitute of Electrical and Electronics Engineers (IEEE 1076-1987, als VHDL'87 bekannt)
seit 1990	Schnelle Verbreitung im akademischen und industriellen Bereich
1993	Überarbeitung des Standards (IEEE 1076-1993, VHDL'93)
2000	Revision des Standards (IEEE 1076-2000)
2002	Eine weitere Revision (IEEE 1076-2002)
2004	Standardisierung durch IEC, International Electrotechnical Commission (IEC 61691-1-1)
2008	Vorläufig letzte Version des Standards (IEEE 1076-2008, VHDL-2008)
01/2009	Letzte Revision von VHDL-2008

VHDL vs. Verilog-HDL

	VHDL	Verilog-HDL
Erste Version	1985	1984
Angelehnt an	Ada	C
Ursprung	DoD	Industrie
Ausdrucksstärke	Ungefähr gleich	
Werkzeugunterstützung	Alle nennenswerten CAD-Werkzeuge	
Synthesefähigkeit	Teilmenge aller Sprachkonstrukte	
Größte Verbreitung	Europa	USA
Standard	IEEE 1076-2008	IEEE 1364-2001

➥ Keine deutlichen Vor- oder Nachteile. Die meisten kommerziellen Systeme erlauben mittlerweile gleichzeitige Verwendung von VHDL und Verilog-Komponenten in einem Entwurf.

Besonderheiten beim VHDL-Einsatz

Wichtig ⚠

In VHDL wird nicht programmiert, in VHDL wird Hardware beschrieben.

- Erfolgreiche Arbeit mit VHDL erfordert eine grundlegende Veränderung der Denkweise im Vergleich zum Softwareentwurf
- Bei der Erzeugung (insbesondere) synthesefähiger Modelle hat jede VHDL-Anweisung ein physikalisches Abbild (Leitung, Register, Latch, Multiplexer, . . .)
- Das Vertauschen von zwei „harmlosen“ Zeilen in Quellcode kann unter Umständen schwerwiegende Folgen haben
- Die Benutzung zweier semantisch scheinbar äquivalenten und bei funktionaler Simulation verhaltensgleichen Beschreibungen kann bei der Synthese zwei völlig unterschiedliche Schaltungen erzeugen (Zeitverhalten, Gatteranzahl, Leistungsaufnahme, . . .)

➥ WYWIWYG, *What You Write Is What You Get!*

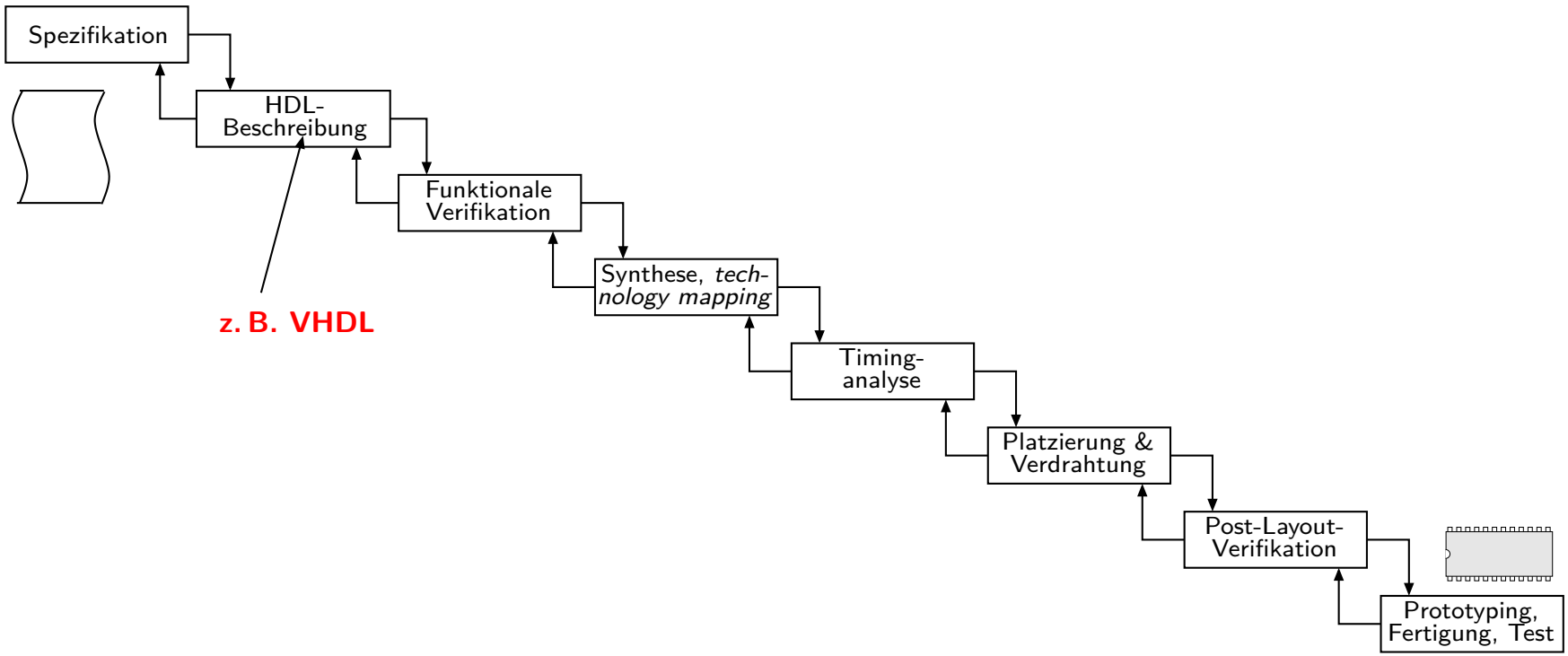
Stand der Sprachstandardisierung

- VHDL'87 ist reichlich überholt, wird von den Entwurfswerkzeugen nur aus Kompatibilitätsgründen unterstützt
- Die größte Revision erfolgte mit dem Übergang zu VHDL'93, danach erfolgten mit VHDL'2000 und VHDL'2002 weitestgehend kosmetische Veränderungen
- In VHDL'2008 wurden Erweiterungen (keine Streichungen!) vorgenommen, die allerdings eher für einen erfahrenen Entwerfer von Interesse sind, diese sind mittlerweile auch in den meisten CAD-Softwareprodukten für digitalen Entwurf umgesetzt
- Absolute Mehrheit der Entwürfe ist nach wie vor VHDL'93-konform

➥ Alle weiteren Betrachtungen gültig für VHDL ab IEEE 1076-1993, auf Unterschiede zum VHDL'87 wird nicht mehr eingegangen, dafür werden einige nützliche Optionen von VHDL'2008 einbezogen

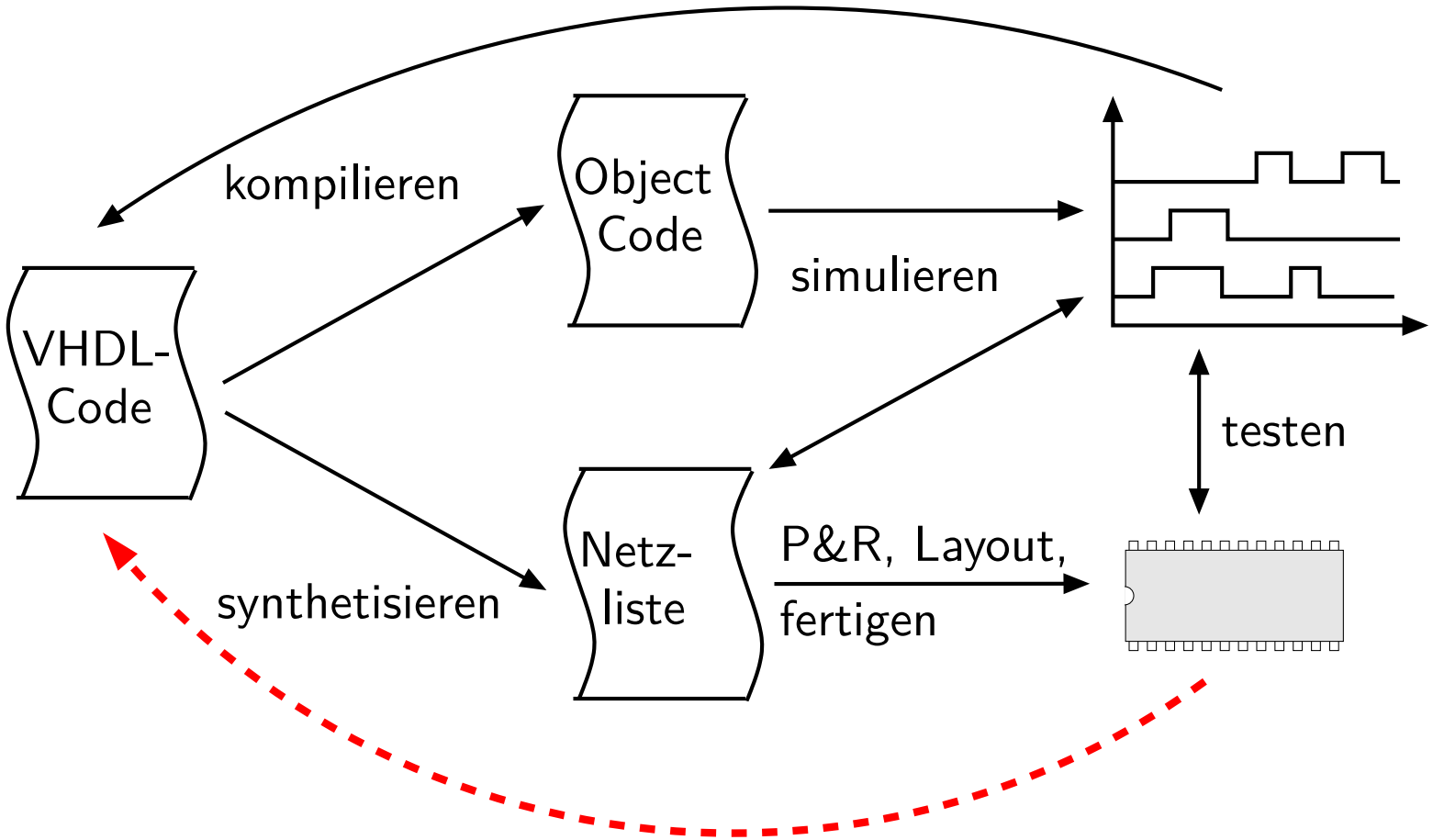
Einordnung von VHDL in den allgemeinen Entwurfsprozess

Technologische Schritte beim Top-Down-Entwurf



➥ Vereinfachtes Modell (Wasserfallmodell der ASIC-Entwicklung)

Einsatz von VHDL und grundsätzliche Abläufe beim VHDL-Entwurf



Zusammenfassung

- Kurze Vorstellung der Vorlesungsinhalte, Aufbau, Kontaktdaten und organisatorischer Ablauf
- Vorstellung der Literaturquellen
- Zusammenfassung der benötigten Voraussetzungen aus zurückliegenden Lehrveranstaltungen
- VHDL im Entwurfsprozess digitaler Systeme

Systementwurf mit VHDL: Vorlesungs-Gesamtübersicht

Systementwurf mit VHDL: Gliederung der 2. Vorlesung

2. Basiskonzepte
   - 2.1. Modellaufbau
   - 2.2. Zeichenvorrat und lexikalische Elemente
   - 2.3. Sprachkonstrukte
   - 2.4. Objekte
   - 2.5. Entwurfseinheiten

2. Basiskonzepte

Basisstruktur

HDL beschreiben Modelle realer Hardware. Mit Simulation können das Verhalten und die Eigenschaften dieser Modelle untersucht werden, mit Synthese (sowie einer Reihe weiterer Entwurfsschritte) werden die Modelle in physikalisch herstellbare Hardware-Strukturen überführt. Bestandteile eines VHDL-Modells:

ENTITY: Schnittstellenbeschreibung des Modells, Modellparameter

ARCHITECTURE: Funktionsbeschreibung (Verhalten, Struktur oder gemischt)

CONFIGURATION (optional, bei einigen EDA-Werkzeugen Pflicht): Zuordnung von ARCHITECTURE zu ENTITY (sowie zu Submodulen), weitere Modellparameter

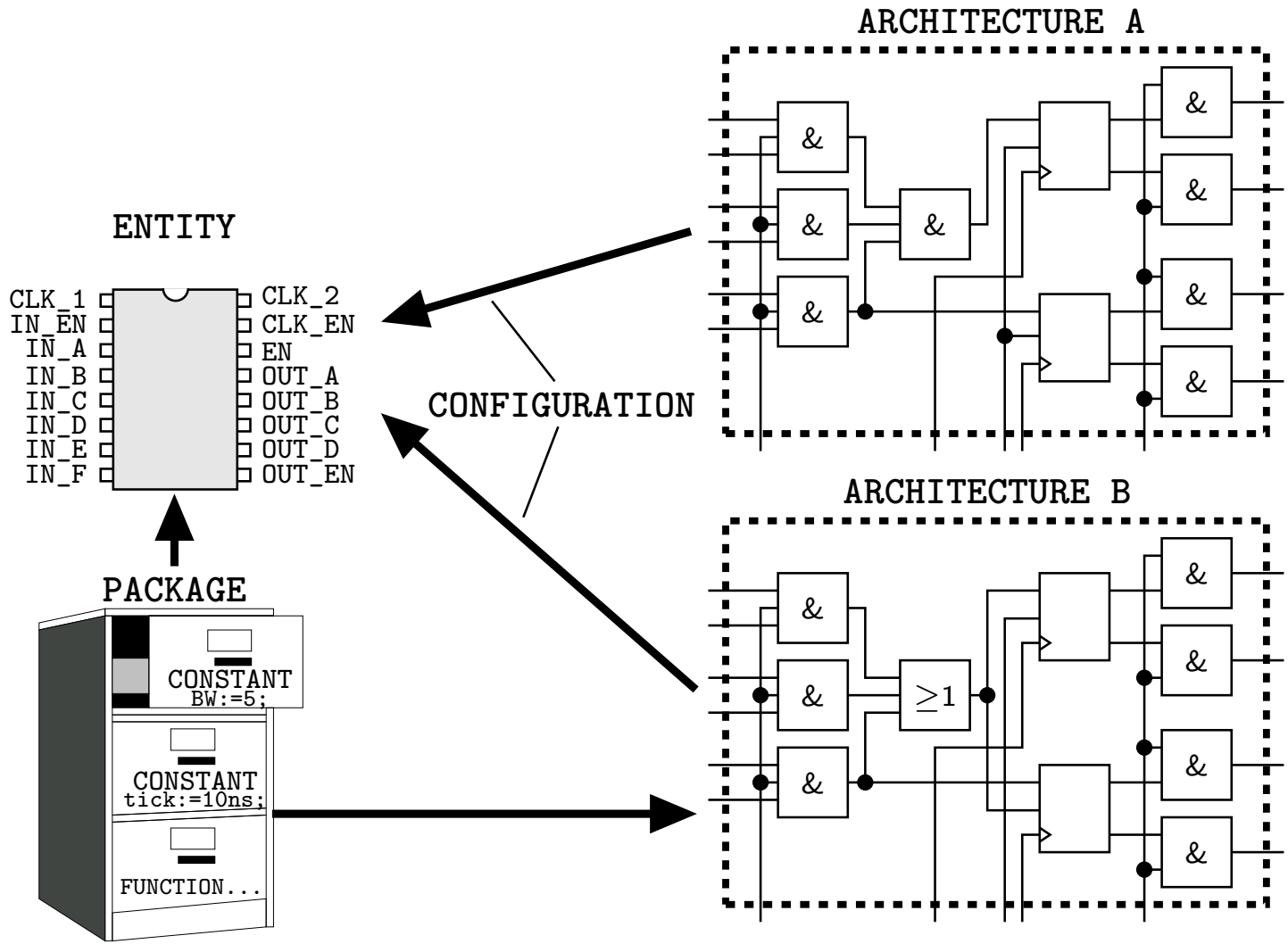
PACKAGE (optional): Deklarationen von Typen, Funktionen, Prozeduren, Submodulen und Konstanten, „VHDL-Bibliothek“

Einige Konventionen

- In VHDL wird nicht zwischen Groß- und Kleinschreibung unterschieden *(VHDL is not case-sensitive)*, die Lesbarkeit erhöht sich dennoch enorm, wenn man die Schlüsselwörter komplett in GROSSBUCHSTABEN und Objektnamen in Kleinbuchstaben setzt (bzw. umgekehrt)
- Solche und ähnliche Regeln werden oft unternehmensintern oder auch fachübergreifend als *Design Style Guidelines* oder *Coding Style Guidelines* zusammengefasst und strikt befolgt:
  - Namensgebung von Objekten- und Entwurfseinheiten
  - Verzeichnisstruktur bei komplexeren Entwürfen
  - Maximale Schachtelungstiefe bei Kontrollstrukturen
  - Zwingende Verwendung bzw. Vermeidung bestimmter Sprachkonstrukte
  - ...

Erste Schlüsselwörter

ENTITY, ARCHITECTURE, CONFIGURATION und PACKAGE



Begriff der Entwurfseinheit

Entwurfseinheiten *(design units)*

ENTITY, ARCHITECTURE, CONFIGURATION und PACKAGE sind Entwurfseinheiten. Ein vollständiges VHDL-Modell besteht aus mindestens zwei Einheiten (eine ENTITY und eine dazugehörende ARCHITECTURE).

PACKAGE kann optional um einen PACKAGE BODY erweitert werden (separate Entwurfseinheit). Im PACKAGE BODY können gleiche Sprachkonstrukte wie im PACKAGE verwendet werden, zusätzlich werden die deklarierten Prozeduren und Funktionen implementiert (vergleichbar mit Header- und Library-Dateien in C).

- Bei einer Änderung der Funktionsimplementierung im PACKAGE BODY müssen alle Entwurfseinheiten, die diese Funktion nutzen, nicht neu übersetzt werden (wenn die Deklaration im PACKAGE gleich bleibt).

Bestandteile eines VHDL-Entwurfs

Entwurfseinheiten: Eine Entwurfseinheit entspricht nicht immer einer physikalischen Projektdatei, es können mehrere Einheiten in einer Datei zusammengefasst werden (aber nicht umgekehrt!)

Libraries: Library ist kein Package, sondern übersetzter VHDL-Code (Objekt-Code) eines Entwurfs, der für die Simulation oder Synthese benötigt wird. Für den aktuellen Entwurf wird automatisch eine Library mit dem symbolischen Namen `WORK` angelegt. Ein Entwurf darf andere Libraries (Resource-Libraries) einbeziehen (Wiederverwendung).

In VHDL-Beschreibungen werden nur logische Bezeichnungen für Entwurfseinheiten und Libraries benutzt (absolute Pfadangaben und physikalische Dateien nur bei der Datei Ein-/Ausgabe). Den Bezug zwischen VHDL-Bezeichnung und der physikalischen Lokation des zugehörigen Objektes stellt die Konfiguration der CAD-Werkzeuge her.

Eine einfache VHDL-Beschreibung

Eine Beispiel-ENTITY:

```
ENTITY und_gatter IS
  PORT (in_a, in_b : IN bit;
        out_c      : OUT bit);
END ENTITY und_gatter;
```

Eine Beispiel-ARCHITECTURE:

```
ARCHITECTURE arch1 OF und_gatter IS
BEGIN
  out_c <= in_a AND in_b;
END ARCHITECTURE arch1;
```

- Ein kompletter Entwurf (ein Analogon von „Hello world!“ in C). Eine Konfiguration ist nicht notwendig, da es nur eine ARCHITECTURE von `und_gatter` gibt (logische Verbindung wird über den gleichen Bezeichner hergestellt).

Eine weitere ARCHITECTURE

Die angegebene ARCHITECTURE `arch1` verwendet den im Sprachumfang enthaltenen `AND`-Operator. Es gibt auch viele andere Möglichkeiten für die Umsetzung gleicher Funktionalität:

```
ARCHITECTURE arch2 OF und_gatter IS
BEGIN
    out_c <= '1' WHEN in_a & in_b = "11" ELSE '0';
END ARCHITECTURE arch2;
```

- `<=` steht für die Zuweisung des Wertes, der sich aus der Auswertung des Ausdrucks rechts vom Gleichheitszeichen ergibt, an das Signal links vom Kleiner-als-Zeichen
- `&` ist der Verkettungs-Operator *(concatenation)*, er verbindet zwei Elemente zu einem durch einfaches Aneinanderhängen (hier entsteht aus zwei einzelnen Bits ein 2-Bit-Wort)
- `WHEN-ELSE`-Konstrukt setzt das Konzept der **bedingten Signalzuweisung** *(conditional signal assignment)* um.

Eine einfache CONFIGURATION

Zwei ARCHITECTURE-Beschreibungen für eine ENTITY erfordern eine CONFIGURATION (beim Weglassen ist das Verhalten systemabhängig: *default*-Bindung, d. h. es wird die letzte übersetzte Architektur mit gleichem Bezeichner hinter „`OF`“ verwendet, bei einigen EDA-Werkzeugen aber auch Warnung oder Fehlermeldung):

```
CONFIGURATION und_gatter_conf OF und_gatter IS
FOR arch1
END FOR;
END CONFIGURATION und_gatter_conf;
```

- Einfachste Form der Konfiguration, sogenannte Blockkonfigurationsanweisung.
- Statt „`arch1`" kann auch „`arch2`" eingesetzt werden, es wird jeweils die in der CONFIGURATION stehende Architektur ausgewählt.

Ein PACKAGE-Beispiel

Modellierung des Zeitverhaltens:

```
PACKAGE zeiten IS
 CONSTANT verz_1 : time := 10 ns;
 CONSTANT verz_2 : time := 15 ns;
END PACKAGE zeiten;

LIBRARY WORK; -- kann entfallen, WORK ist immer sichtbar
USE WORK.zeiten.ALL; -- alle Objekte einbinden

ARCHITECTURE arch1 OF und_gatter IS
BEGIN
  out_c <= in_a AND in_b AFTER verz_1;
END ARCHITECTURE arch1;
```

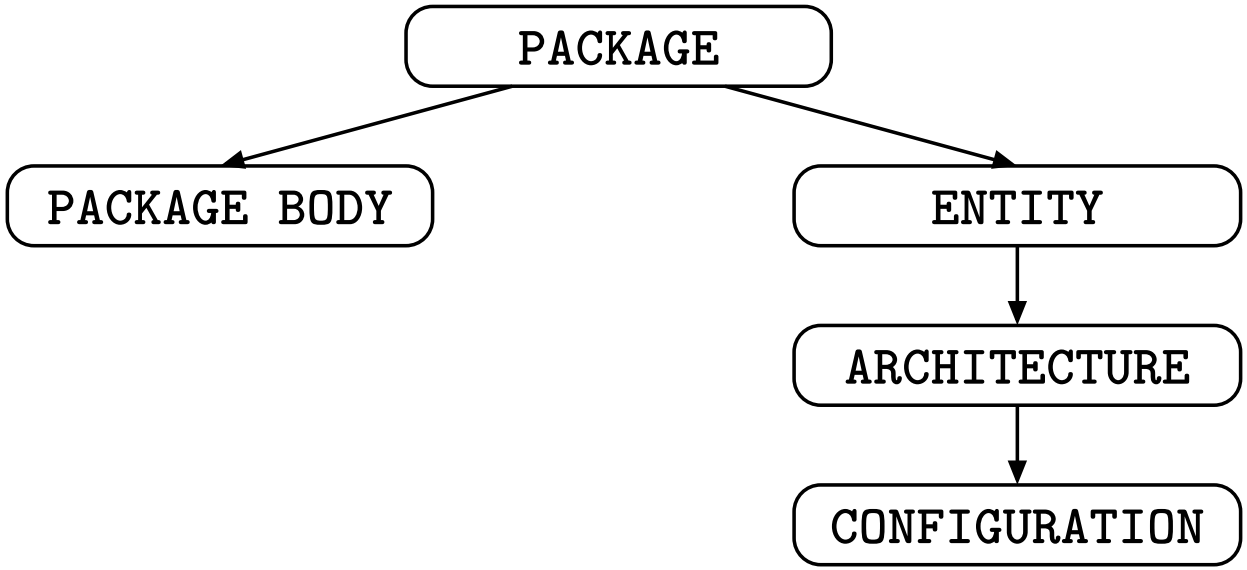
- Jede Entwurfseinheit muss das PACKAGE mit `LIBRARY` und `USE` einzeln einbinden!
- Bei Änderung der Zeiten wird nur das PACKAGE editiert.

Erweiterung um PACKAGE BODY

```
PACKAGE zeiten IS
  CONSTANT verz_1 : time; -- keine Wertzuweisung!
  CONSTANT verz_2 : time; -- keine Wertzuweisung!
END PACKAGE zeiten;

PACKAGE BODY zeiten IS
  CONSTANT verz_1 : time := 10 ns; -- Wert ist hier
  CONSTANT verz_2 : time := 15 ns; -- und hier!
END PACKAGE BODY zeiten;
```

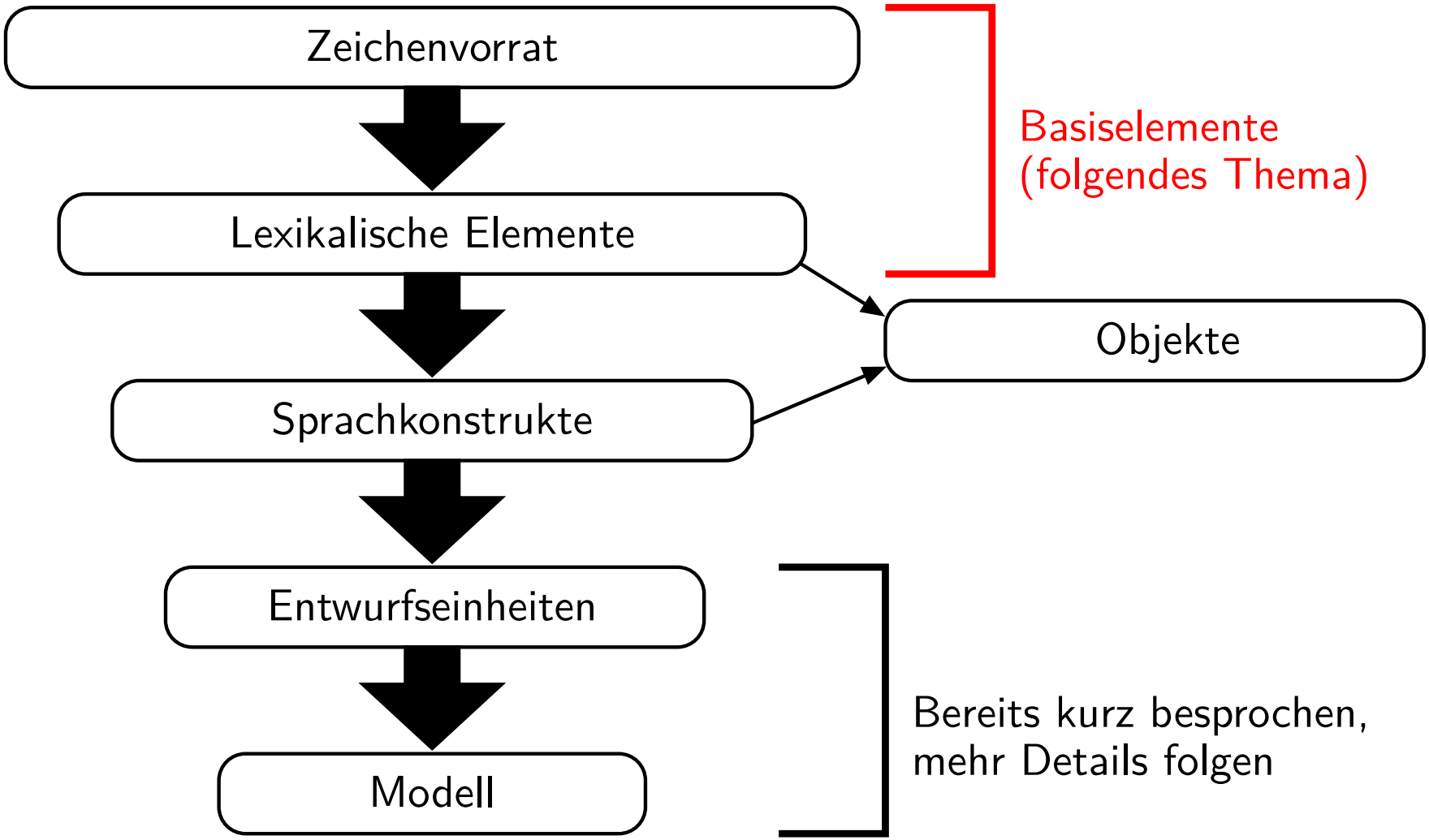
- Legt man den Wert im PACKAGE fest, so müssen alle Entwurfseinheiten (die das PACKAGE nutzen) neu übersetzt werden!
- Bei Zuweisung im PACKAGE BODY wird nur diese eine Einheit neu übersetzt.

Abhängigkeiten beim Übersetzen



- Einige Entwurfswerkzeuge erkennen die Reihenfolge automatisch, bei anderen müssen die Entwurfseinheiten im Projekt entsprechend angeordnet werden
- Für die Simulations- und Syntheseergebnisse sind die Auslagerung der Deklarationen in `PACKAGE BODY` bzw. ein Verzicht darauf nicht relevant (das Ergebnis bleibt gleich)

Globaler Sprachaufbau



Zeichensatz

```
TYPE character IS (
  NUL, SOH, STX, ETX, EOT, ENQ, ACK, BEL, BS,  HT,  LF,
  VT,  FF,  CR,  SO,  SI,  DLE, DC1, DC2, DC3, DC4, NAK,
  SYN, ETB, CAN, EM,  SUB, ESC, FSP, GSP, RSP, USP,
  ' ', '!', '"', '#', '$', '%', '&', ''', '(', ')', '*',
  '+', ',', '-', '.', '/', '0', '1', '2', '3', '4', '5',
  '6', '7', '8', '9', ':', ';', '<', '=', '>', '?', '@',
  'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K',
  'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V',
  'W', 'X', 'Y', 'Z', '[', '\', ']', '^', '_', '`', 'a',
  'b', 'c', 'd', 'e', 'e', 'g', 'h', 'i', 'j', 'k', 'l',
  'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
  'x', 'y', 'z', '{', '|', '}', '~', DEL);
```

Sowie eine Reihe weiterer Sonderzeichen (ISO-8859-1)

Kommentare und Bezeichner

- Kommentare werden an beliebiger Stelle in der Zeile mit `"--"` eingeleitet und enden mit der Zeile
- Bezeichner *(identifier)*:
  - Buchstaben, Ziffern, **einzelne** Unterstriche, keine Leer- und Sonderzeichen
  - Keine Unterscheidung von Groß- und Kleinschreibung
  - Erstes Zeichen muss ein Buchstabe sein
  - Keine Unterstriche am Anfang und am Ende
  - Keine Schlüsselwörter (reservierte Wörter), z. B. `ENTITY`
  - Erweiterte Bezeichner *(extended identifier)*: `"\...\"`, (zwischen Schrägstrichen) haben die genannten Einschränkungen nicht

Beispiele: `Input_1`, `Input2`, `inPUt_23`, `\12__in\`, `\ALU#1\`.

Delimited comments

Neuerung in VHDL-2008 *(delimited comments)*: Kommentare können sich auch über mehrere Zeilen erstrecken, wenn sie statt "--" mit "/*" eingeleitet und mit "*/" abgeschlossen werden. Auch Mischen verschiedener Kommentar-Typen ist möglich (jedoch Vorsicht bei der Verschachtelung):

```
-- Ein Kommentar
-- Noch ein Kommentar /* mit einer Zusatzbemerkung */
-- Noch mehr Kommentare /* mit
-- Abschluss auf einer neuen Zeile */
/* Das ist -- ohne Probleme -- moeglich */
-- Das fuehrt zu /* einer
Fehlermeldung (warum?) */
/* Auch das ist
ENTITY gatter IS /* Deklaration */
problematisch  (warum?) */
```

Reservierte Wörter

ABS	ACCESS	AFTER	ALIAS	ALL
AND	ARCHITECTURE	ARRAY	ASSERT	ATTRIBUTE
BEGIN	BLOCK	BODY	BUFFER	BUS
CASE	COMPONENT	CONFIGURAION	CONSTANT	DISCONNECT
DOWNTO	ELSE	ELSIF	END	ENTITY
EXIT	FILE	FOR	FUNCTION	GENERATE
GENERIC	GROUP	GUARDED	IF	IMPURE
IN	INERTIAL	INOUT	IS	LABEL
LIBRARY	LINKAGE	LITERAL	LOOP	MAP
MOD	NAND	NEW	NEXT	NOR
NOT	NULL	OF	ON	OPEN
OR	OTHERS	OUT	PACKAGE	PORT
PROCESS	POSTPONED	PROCEDURE	PURE	RANGE
RECORD	REGISTER	REJECT	REM	REPORT
RETURN	ROL	ROR	SELECT	SEVERITY
SHARED	SIGNAL	SLA	SLL	SRA
SRL	SUBTYPE	THEN	TO	TRANSPORT
TYPE	UNAFFECTED	UNITS	UNTIL	USE
WAIT	VARIABLE	WHEN	WHILE	WITH
XNOR	XOR			

Reservierte Wörter, die erst mit VHDL-2002 und VHDL-2008 eingeführt wurden

ASSUME	ASSUME_GUARANTEE	CONTEXT
COVER	DEFAULT	FAIRNESS
FORCE	PARAMETER	PROPERTY
PROTECTED	RELEASE	RESTRICT
RESTRICT_GUARANTEE	SEQUENCE	STRONG
VMODE	VPROP	VUNIT

Diese sollten aus Portabilitätsgründen nicht mehr als Bezeichner verwendet werden, auch wenn Übersetzer für ältere Sprachversionen dies zulassen.

Weitere Zeichen und Zeichenketten, die eine besondere Bedeutung haben:

```
"   #   &   '   (   )   *   +   -   ,   .   /   :   ;   <   =   >   ?   @
[   ]   `   |   =>  **  :=  /=  >=  <=  <>  ??  ?=  ?/=  ?>
?<  ?>=  ?<=  <<  >>
```

Zusammenfassung

- Modellaufbau
- Zeichenvorrat
- Lexikalische Elemente

Aufgaben zum Selbststudium (1)

Am Ende der Vorlesung gibt es meistens Aufgaben, die zur Festigung des vermittelten Stoffes dienen. Diese zu Lösen ist **keine Pflicht** und Lösungen werden auch nicht eingesammelt, sie sind eigens zum Zwecke der Selbstkontrolle da. Die Beispiellösungen erscheinen ca. 1 Woche nach der entsprechenden Vorlesung auf dem Handout-Server im Unterverzeichnis „`Aufgabenloesungen`" als „`vhdl_v<n>_loesungen.pdf`" (`n` ≙ Vorlesungsnummer).

1. Sind folgende VHDL-Beschreibungen zulässig?

```
ENTITY empty_entity IS
END empty_entity;
```

```
ENTITY and IS
  PORT (in_a, in_b : IN bit;
        out_c      : OUT bit);
END ENTITY and;
```

Aufgaben zum Selbststudium (2)

2. Erweitern Sie das Modell des UND-Gatters so, dass zwei unabhängige UND-Gatter entstehen, von denen eines die Verzögerungszeit `verz_1` und das andere die Verzögerungszeit `verz_2` aufweist. Legen Sie für `PACKAGE` und `PACKAGE BODY` zwei getrennte Dateien an. Verändern sie die Werte von Verzögerungszeiten und beobachten Sie, dass die neuen Werte in der Simulation sichtbar werden, sobald lediglich das `PACKAGE BODY` neu übersetzt wird. Führen sie das gleiche Experiment für `PACKAGE` ohne `PACKAGE BODY` durch.
3. Welche der folgenden Zeichenketten sind keine gültigen VHDL-Bezeichner und warum?

```
_clk                12_uhr                rst_signal
NAND                ein_signal            ein_signal_
\beze__ichner\      \ein bezeichner\      \12_uhr\
```

Aufgabenlösungen zur 2. Vorlesung

Modellaufbau und Bezeichner

1. Die linke ENTITY-Beschreibung ist zuässig, die rechte dagegen nicht (Schlüsselwort als Bezeichner).
2. Der entsprechende VHDL-Quellkode ist auf dem Handout-Server unter

   `VHDL/Aufgabenloesungen/VHDL/vhdl_v2_und_gatter_2.vhd`

   zu finden.
3. Nachfolgend sind die ungültigen Bezeichner aufgeführt:
   - `_clk` beginnt mit einem Unterstrich
   - `12_uhr` beginnt mit einer Ziffer
   - `NAND` ist ein Schlüsselwort
   - `ein_signal_` endet mit einem Unterstrich

Systementwurf mit VHDL: Vorlesungs-Gesamtübersicht

Systementwurf mit VHDL: Gliederung der 3. Vorlesung

Größen *(literals)*

- Numerische abstrakte Größen
- Numerische physikalische Größen
- Zeichen
- Zeichenketten
- Bit-Ketten

Größen dienen zur Darstellung von Werten bzw. Inhalten von Objekten (Variablen, Signalen usw.)

Numerische abstrakte Größen sind einheitslose Größen, die Zahlen repräsentieren:

- Ganze Zahlen *(integers)*
- Reelle Zahlen *(reals)*

Beispiele von numerischen abstrakten Größenangaben

Ganze Zahlen:

- Ziffernfolgen ohne Dezimalpunkt
- Optional: Exponentialschreibweise mit nichtnegativen ganzzahligen Exponenten

```
10   123   6758   34E6   677e9   12E+5   11e+4 12e00
```

Reelle Zahlen:

- Ziffernfolgen mit Dezimalpunkt
- Optional: Exponentialschreibweise mit ganzzahligen Exponenten

```
0.0   12.3   3.1415   3.4E6   6.77e9   12.0E+12   11.3e-4
445.554e+10
```

Zur Erhöhung der Lesbarkeit dürfen **innerhalb** der Zahl Unterstriche verwendet werden.

Größen in Exponentialdarstellung

Die numerischen abstrakten Größen können zur Basis $B = 2 \ldots 16$ dargestellt werden:

`basis#integerwert[.integerwert]#[exponent]`

Dabei werden Ziffern 10 bis 15 als „`a`" bzw. „`A`" bis „`f`" bzw. „`F`" dargestellt. Beispiele:

```
2   23   0023   2E3   2e3   23e+3   2.3   2.3e3   2.3E-3
2.345e-6   2#110111#   3#21022#   8#76004#e3
16#ADF#E-3   14#AAB#   12#AB#E2   2#1011_1100_1111_1111#
```

- Der Exponent bezieht sich immer auf die Basisangabe (10 ohne explizite Basisangabe), d. h. die Zeichenketten
  `2#1#e10   16#4#E2   1024   10#1024#e+00`
  repräsentieren alle dieselbe Zahl!
- Die Basis selbst ist immer eine Dezimalzahl (das bedeutet z. B. „`11`" statt „`B`"), der Exponent hat die Form `e[+,-]natural` bzw. `E[+,-]natural` (`natural` ist eine natürliche Zahl).

Weitere Größen

Numerische physikalische Größen sind einheitsbehaftete Größen, bestehend aus einer numerischen abstrakten Größe und einer Einheitsangabe (erlaubte Einheiten sind in der Typdeklaration festgelegt, man kann auch eigene Einheiten definieren und verwenden). Beispiele (vom Typ `time`):

```
2 sec, 0.2 ns, 5.3e3 fs, 2.3E-3 ms
```

Zeichengrößen *(characters)* sind einzelne Zeichen in Hochkommata (Achtung: Groß- und Kleinschreibung sind in diesem Spezialfall zu unterscheiden!) Beispiele:

```
'A', 'a', '3', '$', '*', '?', ' '
```

(auch das Leerzeichen ist ein Zeichen).

Zeichenketten

Zeichenketten-Größen *(strings)* sind beliebig lange (bis zum Zeilenende) Ketten von Einzelzeichen in Anführungszeichen. Auch Zeichenketten in Zeichenketten sind möglich (doppelte Anführungszeichen). Beispiele:

```
"A", "ABcC", "Ab  CD", "" (leere Zeichenkette),
"Ein ""String"" in einem String"
```

Besonderheiten bei Zeichenketten:

- Groß- und Kleinschreibung ist wichtig
- `'A'` ist nicht identisch mit `"A"` (und beide ihrerseits nicht identisch mit `"'A'"`)
- Soll sich eine Zeichenkette über mehrere Zeilen erstrecken, muss sie mit Hilfe des „&"-Operators zusammengefügt werden:

```
"Eine Zeichenkette beginnt auf einer Zeile "
& "und endet auf der naechsten Zeile!"
```

Bitketten

Bit-Ketten-Größen *(bit strings)* sind Zeichenketten aus Ziffern `"0"` bis `"9"` und Buchstaben `"a"` bis `"f"` (bzw. `"A"` bis `"F"`) in Anführungszeichen, die einen binären, oktalen, dezimalen oder hexadezimalen Zahlenwert darstellen. Die Basis wird wie folgt gekennzeichnet:

`b` bzw. `B`	binär	`d` bzw. `D`	dezimal (nur ab VHDL-2008)
`o` bzw. `O`	oktal	`x` bzw. `X`	hexadezimal

Beispiele:

```
"1010101010101", b"1010_1010_1111_1111", X"CAFF10FF",
d"12345222345", o"345232677", D"345232677"
```

Unabhängig von der Basis werden diese Angaben intern als Bitketten (Folgen von „1“en und „0“en) interpretiert. Auch bei Bitketten sind zur Erhöhung der Lesbarkeit Unterstriche **innerhalb** der Zeichenketten erlaubt. Ohne Basisangabe wird die Bitkette als binär interpretiert.

Besonderheiten bei Bitketten

- Ist die Basis dezimal, so entspricht die Länge der Bitkette der minimalen Anzahl der Bits, die für die Darstellung der Größe zur Basis 2 notwendig sind
- Ist die Basis oktal bzw. hexadezimal, so wird die Länge der Bitkette als jeweils 3 Bit pro Ziffer bzw. 4 Bit pro Ziffer berechnet
- Im Sinne der Basis „ungültige“ Zeichen werden entsprechend dem letzten Punkt vervielfältigt (nicht bei dezimal und Unterstrich)

Folgerung:

```
o"265" = b"010_110_101"
x"B5"  = b"1011_0101"
```

Beide Bitketten sind nicht identisch, auch wenn bei vorzeichenloser Konvertierung in das dezimale Zahlensystem in beiden Fällen derselbe Wert entsteht!

➥ Wichtig bei Zuweisung von Bitketten an Objekte (z. B. Signale), da die Bitbreite dabei überprüft wird.

Einige Neuerungen in VHDL-2008

Nur für binäre, oktale und hexadezimale Bitketten:

- Optionale explizite Längenangabe bei Bitketten, z. B. `8o"265"` entspricht `b"1011_0101"` (das oberste Bit wird weggelassen) bzw. `12o"265"` entspricht `b"0000_1011_0101"` (die „fehlenden“ Bitstellen werden mit „0“ gefüllt). Weglassen signifikanter Bitstellen (explizite Längenangabe ist zu kurz) ist verboten (Fehlermeldung).
- Optionale explizite Kennzeichnung der Interpretation des MSB: `UB`, `UO` und `UX` bzw. `SB`, `SO` und `SX` stehen für vorzeichenlose *(unsigned)* bzw. vorzeichenbehaftete *(signed)* Zahlen. Wichtig insbesondere bei expliziten Längenangaben, damit das Vorzeichen korrekt behandelt werden kann, z. B.

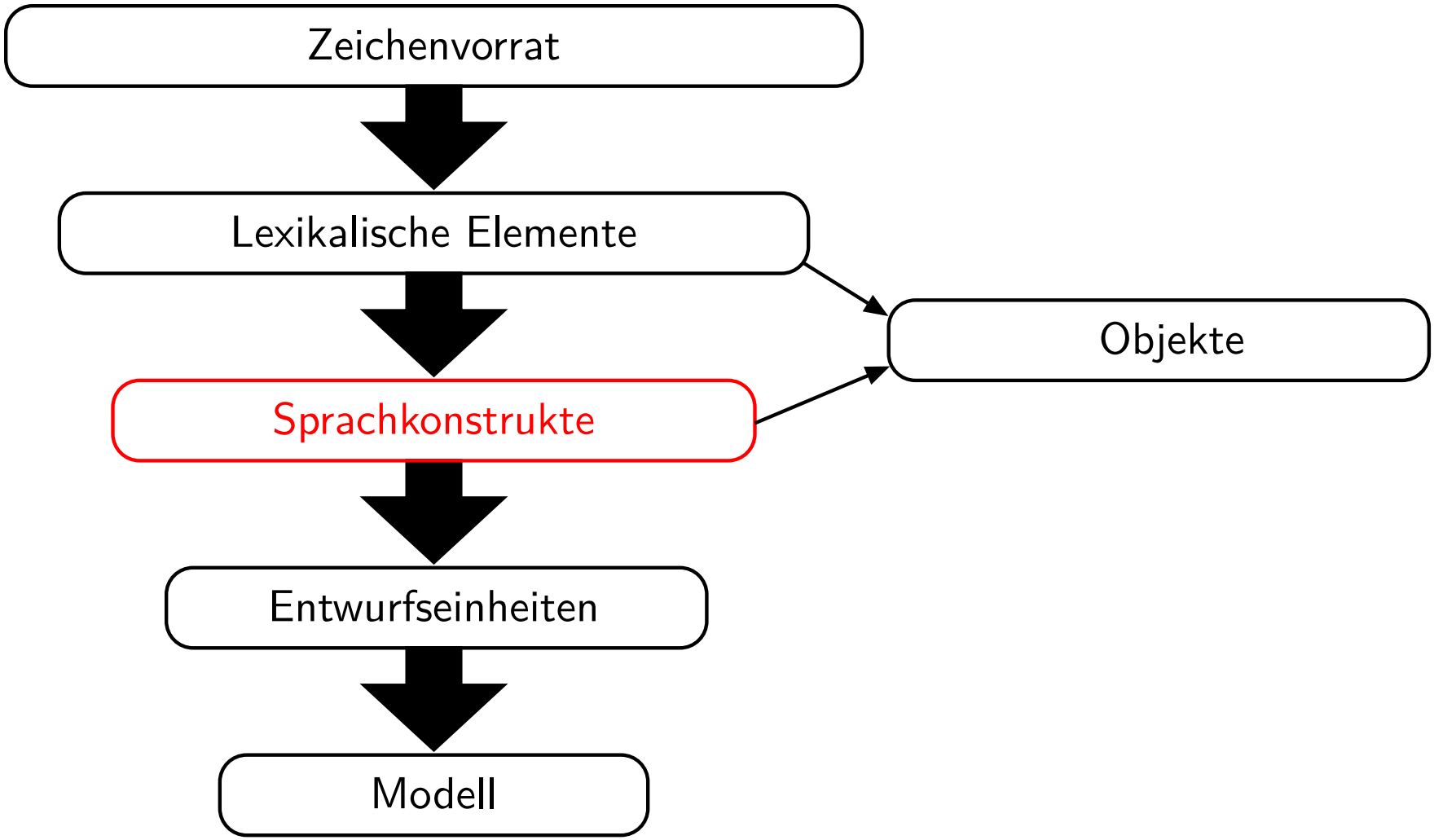
```
10SX"65" = b"00_0110_0101" 10UX"65" = b"00_0110_0101"
10SX"85" = b"11_1000_0101" 10UX"85" = b"00_1000_0101"
```

Basiselemente: Zusammenfassung

- Verschiedene Arten von Größenangaben werden in den Modellen bei Wertezuweisungen, Konstantendefinitionen, Vergleichen usw. eingesetzt.
- Häufige Fehlerquellen
  - Inkorrekte Syntax bei Größenangaben
  - Verwendung von reservierten Wörtern als Bezeichner (die vom Übersetzer erzeugten Fehlermeldungen sind bei dieser Art von Fehler nicht immer hilfreich)
  - Verwechslung von Zeichen und Zeichenketten

➥ Fehler bei Basiselementen werden glücklicherweise bereits beim Übersetzen des Modells erkannt

Einordnung


Zeichenvorrat
Lexikalische Elemente
Objekte
Sprachkonstrukte
Entwurfseinheiten
Modell

Arten von Sprachkonstrukten

Sprachkonstrukte sind Kombinationen von lexikalischen Elementen, die eine syntaktische Bedeutung haben:

Primitive: Möglichkeiten zur Darstellung eines Wertes (einzelne Operanden oder Ausdrücke aus Operanden und Operatoren). Eine Primitive kann selbst wieder als ein Operand dienen.

Befehle (Anweisungen): Kombinationen von Schlüsselwörtern, die eine bestimmte Funktion beschreiben (Signalzuweisungen, Schleifen, Verzweigungen usw.)

Syntaktische Rahmen: Kombinationen von Schlüsselwörtern, die andere Sprachkonstrukte (Funktionen, Entwurfseinheiten, Submodule usw.) einbetten, z. B.

```
ENTITY xyz
...
END ENTITY xyz;
```

Operanden und Operatoren

Als Operanden kommen explizite Größenangaben, Bezeichner, Funktionsaufrufe u. ä. in Frage.

Eine Auswahl von Operatoren (zeilenweise sortiert nach absteigender Priorität):

`**, ABS, NOT`	Potenzieren, Betrag, Negation
`*, /, MOD, REM`	Multiplikation und Division
`+, -`	Vorzeichen
`+, -, &`	Addition, Subtraktion, Verkettung
`=, /=, <, <=, >, >=`	Vergleiche
`AND, NAND, OR, NOR, XOR, XNOR`	Logische Verknüpfungen

Operatoren gleicher Prioritätsklasse werden in einem Ausdruck in der Reihenfolge des Auftretens bearbeitet, d. h. linksassoziativ (Änderung der Reihenfolge durch Klammerung möglich)

Befehle (Anweisungen)

- Deklarationen
  - Typdeklarationen
  - Objektdeklarationen (Signale, Variable, ...)
  - Schnittstellendeklarationen (siehe den Abschnitt `PORT(...)` bei `ENTITY`)
  - Komponentendeklarationen
  - Funktions- und Prozedurdeklarationen
- Sequentielle Anweisungen (werden sequentiell simuliert und meistens als Schaltnetze synthetisiert)
- Nebenläufige Anweisungen (werden als pseudoparallel simuliert und als parallel arbeitende Hardware-Module synthetisiert)
- Konfigurationsbefehle (Konfiguration von Modellen, nicht zu verwechseln mit `CONFIGURATION`)

Sprachelemente am Beispiel von und_gatter

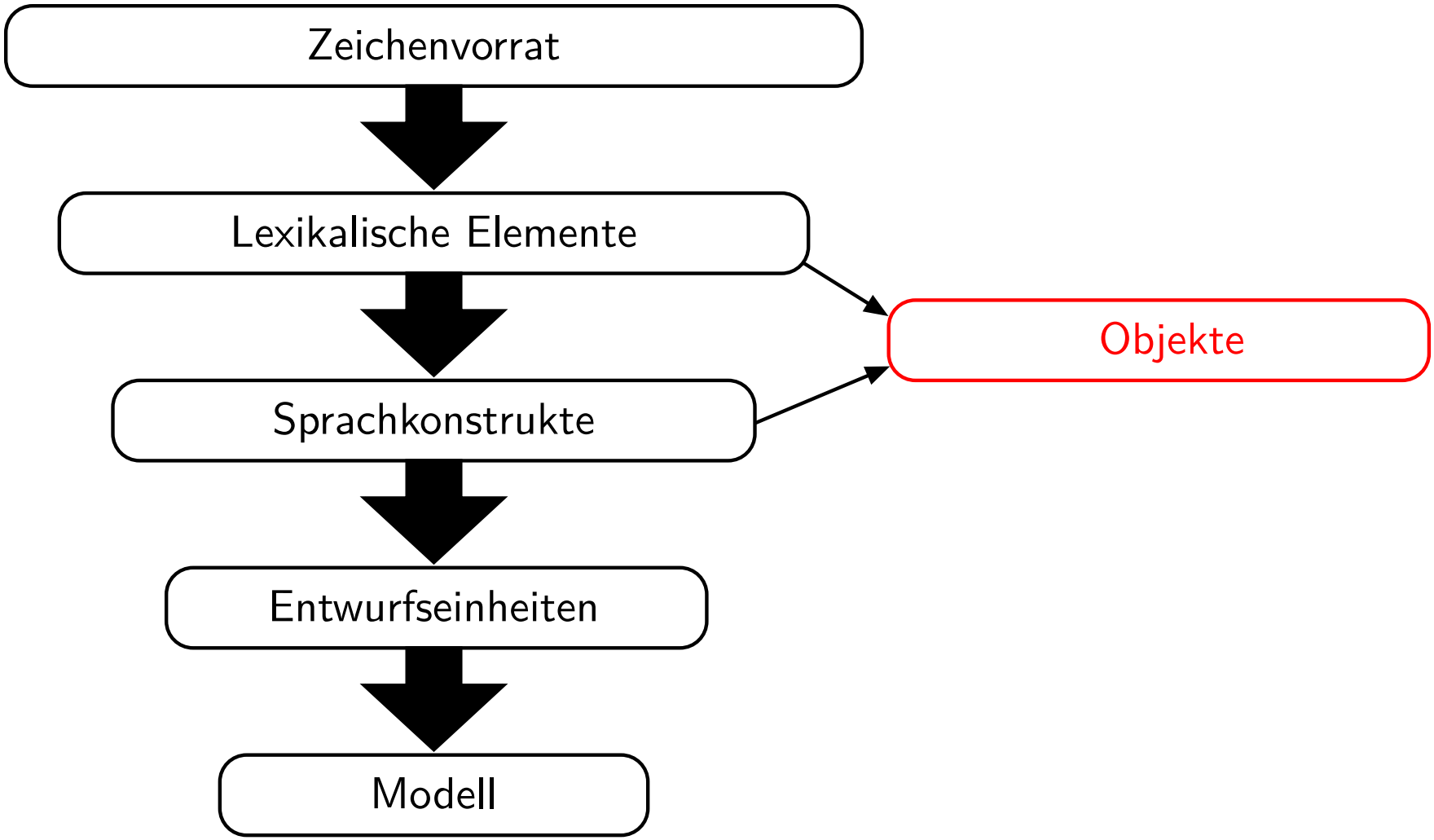
```
1 ARCHITECTURE arch1 OF und_gatter IS
2 BEGIN
3   out_c <= in_a AND in_b;
4 END ARCHITECTURE arch1;
```

Zeilen 1 und 4 stellen einen syntaktischen Rahmen der Entwurfseinheit `ARCHITECTURE` dar. Zeile 3 ist ein Befehl (Signalzuweisung <=, nicht zu verwechseln mit Vergleich), der die Primitive `out_c`, `in_a` und `in_b` (Bezeichner, Operanden) sowie die Primitive `AND` (Operator) nutzt.

Warum ist diese Unterscheidung wichtig? ⚠

Nicht alle Konstrukte können in allen syntaktischen Rahmen angewandt werden! Ohne Verständnis der syntaktischen Strukturierung einer Beschreibung wird ein Entwerfer viele Fehlermeldungen beim Übersetzen nicht interpretieren können.

Einordnung


Zeichenvorrat
Lexikalische Elemente
Sprachkonstrukte
Entwurfseinheiten
Modell
Objekte

Objektgruppen

Wichtig ⚠

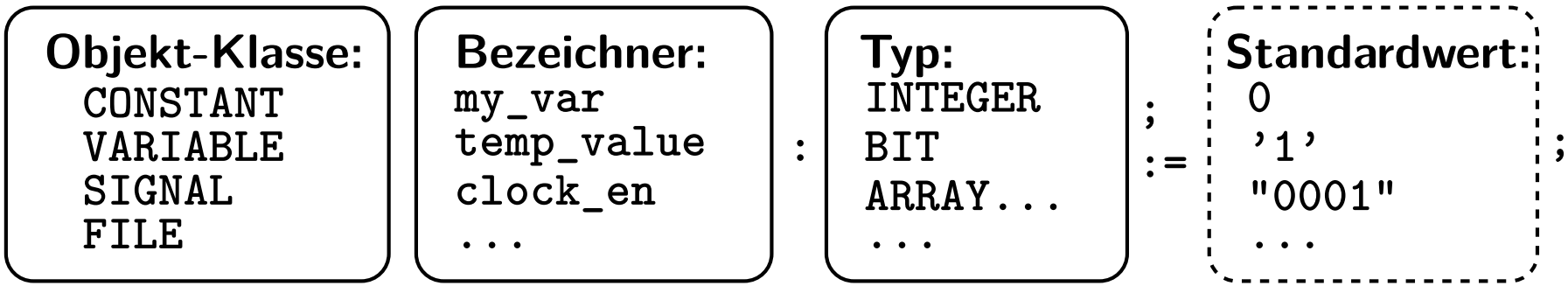
VHDL verwaltet Daten in Form von **Objekten**. Objekt in VHDL ist nur eine Bezeichnung der internen Datenverwaltungseinheit und ist nicht im Sinne eines objektorientierten Ansatzes zu verstehen!

Objekte in VHDL können in folgende Gruppen (Klassen) eingeteilt werden:

- Konstanten
- Variable
- Signale
- Dateien

Für Objekte in VHDL besteht **Deklarationspflicht** (Angabe der Gruppe, des Bezeichners, des Datentyps und eines optionalen Standard- bzw. Initialisierungswertes, *default value*)! Als Typ kommen vordefinierte oder benutzerdefinierte (vor der ersten Objekt-Deklaration) Typen in Frage. **VHDL ist streng typisiert!**

Deklaration von Objekten



Beispiele:

```
CONSTANT verz_1    : time := 3 ns;
CONSTANT zaehler   : integer := 0;
VARIABLE a1        : integer := 2;
VARIABLE a2, a3    : bit;
SIGNAL   n_2       : bit_vector (0 TO 15) := X"00FF";
SIGNAL   clk_en    : bit;
ALIAS    verz_x    : time IS verz_1;
```

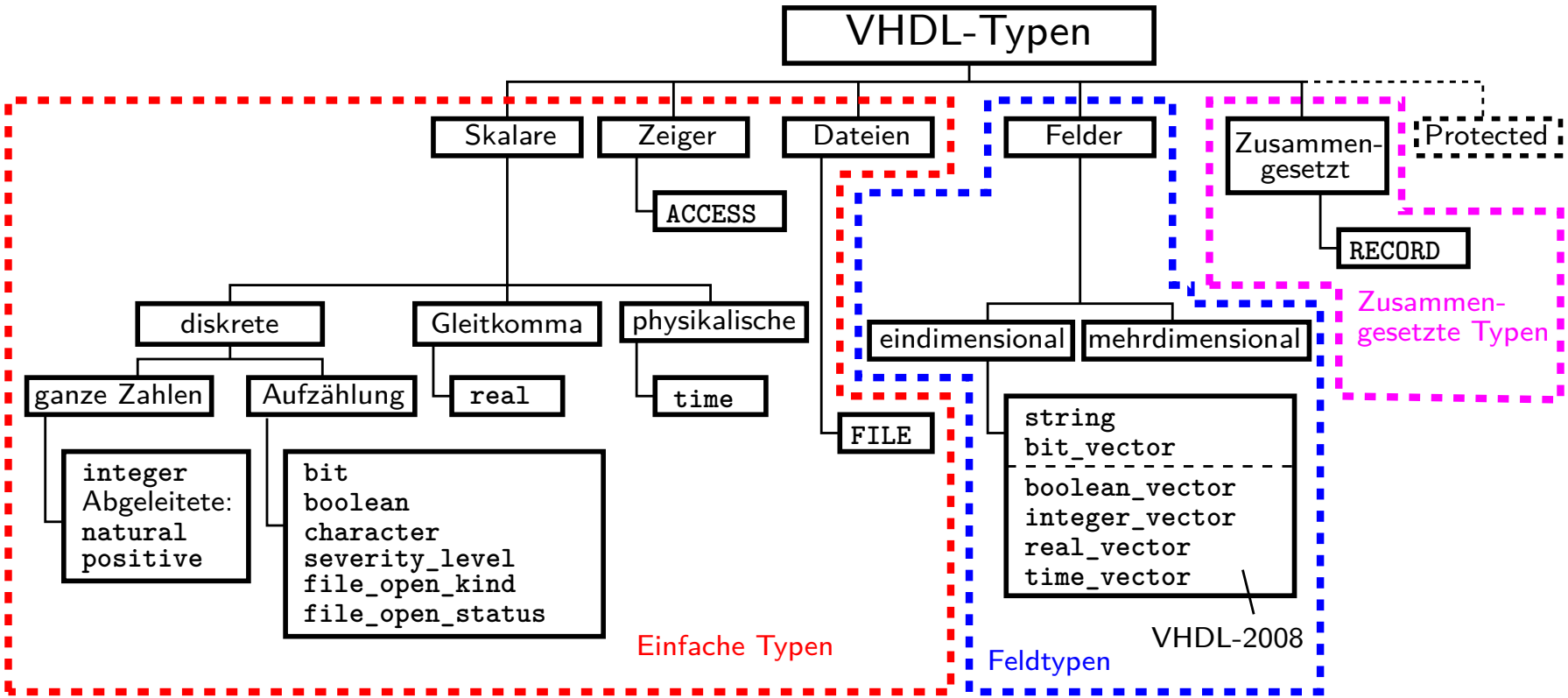
Als Objekt-Klasse kann auch ein `ALIAS` stehen (eine andere Bezeichnung für ein bestehendes Objekt).

Einteilung

- Einfache Typen
  - Aufzählltypen
  - Ganzzahlige
  - Gleitkommatypen
  - Physikalische
  - Zeiger
  - Dateien
  - Abgeleitete
- Feldtypen
  - Eindimensionale
  - Mehrdimensionale
  - Abgeleitete
- Zusammengesetzte Typen

Übersicht

Systematische Einteilung der Typen in VHDL (mit Beispielen von vordefinierten Typen):



Deklaration von neuen Typen

```
TYPE bezeichner IS definition;
```

Der `definition`-Teil ist abhängig von der Art des Typs, der deklariert wird (z. B. Aufzählung, Unterbereich eines vorhandenen Typs usw.)

Beispiel zur strengen Typisierung:

```
TYPE apfel IS RANGE 0 TO 100;
TYPE birne IS RANGE 0 TO 100;
...
SIGNAL a : apfel;
SIGNAL b : birne;
...
a <= b; -- illegal!
```

➥ Erzeugt eine Fehlermeldung (obwohl die Typen inhaltlich vollkommen identisch sind)

Aufzähltypen und ganzzahlige Typen

Aufzähltypen *(enumeration types)*, eine endliche nichtleere Menge von vordefinierten festen Bezeichnern (keine Zahlen!):

```
TYPE abc IS ('A', 'B', 'C', 'a', 'b', 'c');
TYPE wochentage IS (mo, di, mi, do, fr, sa, so);
```

Vordefinierte Aufzählypen (`standard`-Package):

```
TYPE boolean IS (false, true);
TYPE bit IS ('0', '1');
TYPE character IS (...); -- bereits besprochen
TYPE severity_level IS (note, warning, error,
                                        failure);
```

Ganzzahlige Typen *(integer types)*, Angaben der Wertebereiche von ganzen Zahlen, der Typ `integer` ist vordefiniert:

```
TYPE dezimal IS RANGE 0 TO 9;
CONSTANT max_amp : INTEGER := 5;
TYPE amplitude IS RANGE -max_amp TO max_amp;
```

Gleitkomma und physikalische Typen

Gleitkomma Typen *(real types, floating point types)*, Angaben der Wertebereiche von rationalen Zahlen, der Typ `real` ist vordefiniert:

```
TYPE messbereich IS RANGE -2.0 TO 2.0;
TYPE alpha IS RANGE 0.0 to 5.6;
```

Physikalische Typen, ganzzahliger oder Gleitkomma-Wert kombiniert mit einer Einheit (auch Ableitung weiterer Untereinheiten möglich), der Typ `time` ist vordefiniert:

```
TYPE abstand IS RANGE -1e8 TO 1e8
  UNITS mm;
    cm = 10 mm;
    m = 100 cm;
    km = 1000 m;
END UNITS abstand;
TYPE widerstand IS RANGE 0 TO 1e+9 UNITS ohm;
END UNITS widerstand;
```

Operationen mit Objekten physikalischer Typen

Das Ergebnis ist vom gleichen physikalischen Typ:

- `ABS`, Betragsbildung
- `MOD`, `REM`, Rest der ganzzahligen Division (nur in VHDL-2008)
- Division und Multiplikation mit einer Konstanten

Divisionen und Multiplikationen von zwei Objekten physikalischer Typen sind nicht erlaubt, da das Ergebnis meistens nicht dem gleichen Typ zugeordnet werden kann.

➥ Eine Multiplikation von zwei Abständen (siehe Definition auf der letzten Seite) erzeugt eine Größe, die von der Dimension her einer Fläche entspricht.

Abgeleitete Typen und Feldtypen

Zeiger und Dateien werden später behandelt.

Abgeleitete einfache Typen *(subtypes)*, Einschränkung des Wertebereiches bestehender Typen (aber nicht von bereits abgeleiteten Typen), `natural` und `positive` sind vordefiniert:

```
SUBTYPE ein_byte IS integer RANGE 0 TO 255;
SUBTYPE natural IS integer RANGE 0 TO integer'HIGH;
SUBTYPE positive IS integer RANGE 1 to integer'HIGH;
```

Eindimensionale Feldtypen (*one-dimensional arrays*, Vektoren), Zusammenfassung mehrerer Objekte eines Typs zu einem. Die Größe kann sowohl eingeschränkt *(constrained array)* als auch uneingeschränkt *(unconstrained array)* sein, `string` und `bit_vector` sind vordefiniert:

```
TYPE ein_vektor IS ARRAY (0 TO 127) OF integer;
```

Mehrdimensionale und abgeleitete Feldtypen

Mehrdimensionale Feldtypen (*multi-dimensional arrays*, Vektoren), Zusammenfassung mehrerer eindimensionaler Felder. Jedes Feld darf einen anderen Basistyp und Dimensionierung haben:

```
TYPE int_matrix IS ARRAY (positive RANGE 4 TO 10,
     natural RANGE 9 DOWNTO 0) OF integer;
TYPE drei_d IS ARRAY (integer RANGE 0 TO 99,
     integer RANGE 0 TO 99,
     integer RANGE 0 TO 99) OF int_matrix;
```

Abgeleitete Feldtypen, Einschränkung des Indexbereichs bestehender Feldtypen (aber nicht von bereits abgeleiteten Feldtypen):

```
SUBTYPE halbwort IS bit_vector (1 TO 16);
SUBTYPE nachname IS string (1 TO 20);
```

Zusammengesetzte Typen

Zusammengesetzte Typen *(records)*, Zusammenfassung mehrerer Objekte gleicher oder unterschiedlicher Typen zu einem:

```
TYPE komplexe_zahl IS RECORD
  realteil      : real;
  imaginaerteil : real;
END RECORD;

TYPE monate IS (Jan, Feb, Mar, Apr, Mai, Jun,
                Jul, Aug, Sep, Okt, Nov, Dez);
SUBTYPE tage IS integer RANGE 1 TO 31;

TYPE datum IS RECORD
  jahr  : natural;
  monat : monate;
  tag   : tage;
END RECORD;
```

Vorteile von Aufzählungen, abgeleiteten Typen und Einschränkungen der Wertebereiche

- bessere Lesbarkeit
- Hilfe bei der Fehlersuche (Überschreitungen der Wertebereiche sind eine sehr häufige Fehlerquelle):
  - Adressberechnung (Anzahl der Speicherstellen $\leq 2^{Adressbreite}$)
  - Ereigniszähler (modulo $< 2^{Bitbreite}$)
  - ...
- Reduzierung der Komplexität bei der Synthese: Ist die exakte Bitbreite nicht festgelegt (z. B. erste Komplexitätsabschätzungen, Pin-Belegungen unbekannt usw.), so erzeugt ein Synthese-Werkzeug bei einem `integer`-Objekt in der Regel Strukturen für 32 Bit

Typumwandlung

- durch die strenge Typisierung müssen Operanden bei einer Operation der jeweiligen Typvorgabe entsprechen.
- oft ist eine Typumwandlung erforderlich
- für ganzzahlige und Gleitkommatypen sind Umwandlungsfunktionen vordefiniert (`integer()`, `real()` usw.)
- meistens sind Umwandlungsfunktionen in Bibliotheken zusammengefasst (dort, wo die entsprechenden Typen auch deklariert sind)
- Konvertierung ist oft sehr lästig, durch strenge Typisierung können aber viele Fehlerquellen eliminiert werden
- nicht alle Operationen sind für alle Typen definiert, auch wenn man es „nach eigenem Ermessen“ voraussetzen würde (z. B. Addition von Bitvektoren)

Zulässige Deklarationsorte

- Typen: in ENTITY (nach `PORT()`, Deklarationsteil), in ARCHITECTURE (vor `BEGIN`, Deklarationsteil), in PACKAGE und PACKAGE BODY, in BLOCK-, PROCESS-, FUNCTION- und PROCEDURE-Deklarationsteil (dazu später mehr)
- Konstanten: wie Typen
- Variable: im PROCESS-, FUNCTION- und PROCEDURE-Deklarationsteil
- Signale: im ENTITY-, ARCHITECTURE- und BLOCK-Deklarationsteil sowie in PACKAGE
- Ganzzahlige Laufvariable in `FOR`–Schleifen müssen nicht deklariert werden (einzige Ausnahme von der Deklarationspflicht)

Ansprechen von Objekten

- Einfache Typen über die Namensangabe:

```
CONSTANT verz_1 : time := 1 ns;
SIGNAL a_temp : bit;
...
a_temp <= '0' AFTER verz_1;
```

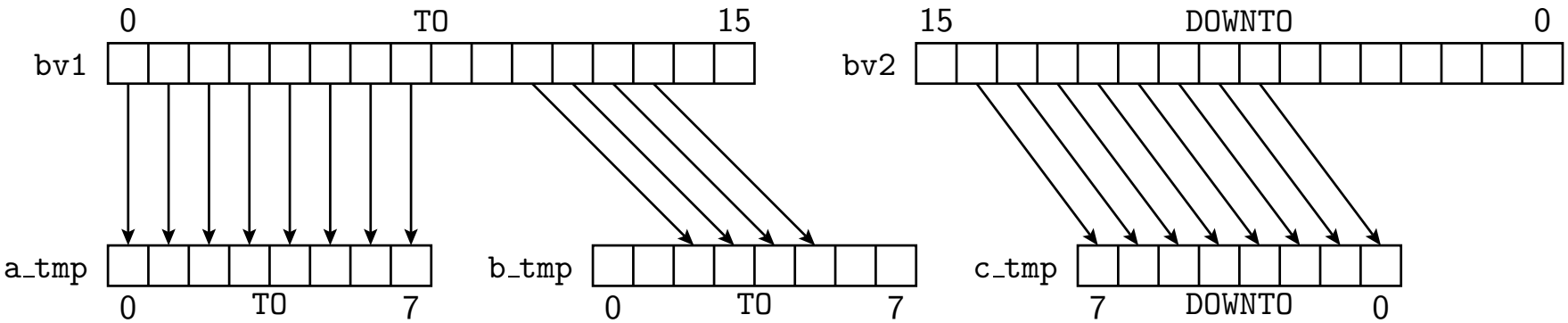
- Einzelelemente von Feldtypen über Namen und Index (oder Indizes)

```
TYPE bit_matrix IS ARRAY (0 TO 4, 0 TO 4) OF bit;
SIGNAL bv : bit_vector (0 TO 5);
SIGNAL bm : bit_matrix;
SIGNAL a_temp, b_temp : bit;
...
a_temp <= bv(2);
b_temp <= bm(2,2);
```

Auswahl der Unterbereiche

Bei Objekten von Feldtypen können auch Unterbereiche selektiv angesprochen werden:

```
SIGNAL bv1           : bit_vector (0 TO 15);
SIGNAL bv2           : bit_vector (15 DOWNTO 0);
SIGNAL a_tmp, b_tmp  : bit_vector (0 TO 7);
SIGNAL c_tmp         : bit_vector (7 DOWNTO 0);
...
a_tmp <= bv1(0 TO 7);
b_tmp(2 TO 5) <= bv1 (10 TO 13);
c_tmp <= bv2(14 DOWNTO 7);
```



Ansprechen von Objekten von zusammengesetzten Typen

erfolgt über den Namen und die Referenz auf die Bezeichnung in der Typdeklaration (durch Punkt getrennt):

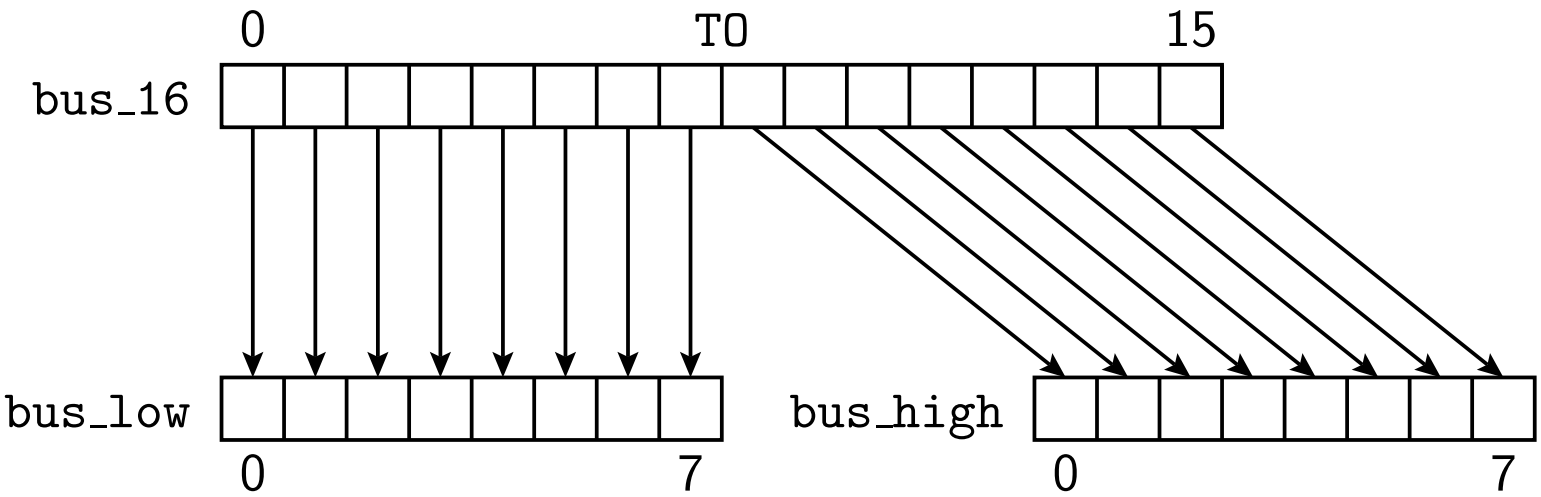
```
TYPE komplexe_zahl IS RECORD
  realteil      : real;
  imaginaerteil : real;
END RECORD;
SIGNAL a_tmp, b_tmp   : komplexe_zahl;
SIGNAL r_teil, i_teil : real;
...
b_tmp <= a_tmp;
r_teil <= a_tmp.realteil;
i_teil <= a_tmp.imaginaerteil;
```

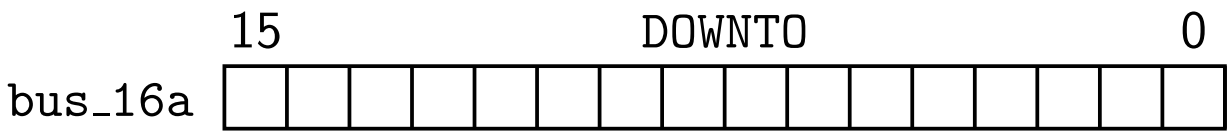
Hier kann z. B. auch ein `ALIAS` sinnvoll eingesetzt werden:

```
ALIAS re IS a_tmp.realteil;
```

Beispiel zur Nutzung von ALIAS

```
SIGNAL bus_16  : bit_vector (0 TO 15);
ALIAS bus_16a  : bit_vector (15 DOWNTO 0) IS bus_16;
ALIAS bus_low  : bit_vector (0 TO 7) IS bus_16 (0 TO 7);
ALIAS bus_high : bit_vector (0 TO 7) IS bus_16 (8 TO 15);
```





Ein weiteres Beispiel zur Nutzung von ALIAS

Objekt mit gleichem Bezeichner aber unterschiedlichen Werten in zwei verschiedenen Packages:

```
PACKAGE ext_pack1 IS
CONSTANT verz_t : time
    := 3 ns;
...
END PACKAGE ext_pack1;
```

```
PACKAGE ext_pack2 IS
CONSTANT verz_t : time
    := 4 ns;
...
END PACKAGE ext_pack2;
```

Werden beide Packages von einer Entwurfseinheit genutzt, so muss die Konstante immer mit vollen Package-Namen referenziert werden:

```
a <= work.ext_pack1.verz_t;
b <= work.ext_pack2.verz_t;
```

Abhilfe mit

```
ALIAS verz_t1 IS  work.ext_pack1.verz_t;
ALIAS verz_t2 IS  work.ext_pack2.verz_t;
```

Zusammenfassung

- Lexikalische elemente: Größen, Zeichen- und Bitketten
- Primitive, Befehle und syntaktische Rahmen
- Objektdeklaration
- Datentypen

Aufgaben zum Selbststudium

1. Definieren sie einen zusammengesetzten Typ, mit dem Informationen für einen Personen-Eintrag im Studentenverzeichnis dargestellt werden können.
2. Erzeugen Sie einen Entwurf, in dem zwei verschiedenen Elementen des im Punkt 1 definierten Typs bestimmte Werte zugewiesen werden.

Aufgabenlösungen zur 3. Vorlesung

Zusammengesetzte Typen in VHDL

Der entsprechende VHDL-Quellkode ist auf dem Handout-Server unter

`VHDL/Aufgabenloesungen/VHDL/vhdl_v3_record.vhd`

zu finden.

Systementwurf mit VHDL: Vorlesungs-Gesamtübersicht

Systementwurf mit VHDL: Gliederung der 4. Vorlesung

Ansprechen von Feldtypen-Objekten über Aggregate

- *positional association*, über Position des Elementes im Feld
- *named association*, über Index des Elementes im Feld

```
TYPE int_vector IS ARRAY (0 TO 7) OF integer;
SIGNAL a : int_vector;
...
a <= (2,4,6,4,0,0,2,3); -- positional
a <= (0 TO 3 => 4, 4 TO 7 => 3); -- named
a <= (0 TO 2 => 3, 3 | 7 => 4, OTHERS => 0); -- named
a(0 TO 3) <= (0, 1, 0, 1); -- slice und positional
a(4 TO 7) <= (1, 1, 1, 1);
```

`OTHERS` bezeichnet alle nicht spezifizierten Elemente, „`|`" trennt Elemente mit gleichem Wert. *named* und *positional* dürfen nicht in einer Zuweisung kombiniert werden, `OTHERS` darf bei beiden Arten verwendet werden.

Aggregate bei Objekten von zusammengesetzten Typen

```
TYPE komplexe_zahl IS RECORD
  realteil      : real;
  imaginaerteil : real;
END RECORD;

SIGNAL a_tmp, b_tmp, c_tmp, d_tmp : komplexe_zahl;
SIGNAL r_teil, i_teil             : real;
...
a_tmp <= (3.4, 2.5); -- positional
b_tmp <= (realteil => 3.4,
          imaginaerteil => 2.5); -- named
c_tmp <= (3.4, OTHERS => 2.5);   -- positional
d_tmp <= (a_tmp.realteil, a_tmp.imaginaerteil); -- d = a
```

Informationen über Objekte und Typen

können über **Attribute** abgefragt werden:

Typbezogene Attribute gehören zu den einfachen Datentypen (z. B. Wertebereich)

Feldbezogene Attribute gehören zu den Felddatentypen (z. B. Größe des Arrays)

Signalbezogene Attribute gehören zu den Signalen (Details in den späteren Vorlesungen)

Allgemeine Attribute liefern Informationen über den Namen der oder den Pfad zu einer Entwurfseinheit in einem Modell.

```
TYPE drei_bit IS RANGE 0 TO 7;
VARIABLE a : drei_bit;
...
a := drei_bit'LEFT; -- a = 0;
a := drei_bit'RIGHT; -- a = 7;
```

Typbezogene Attribute

`t'BASE`	Basistyp des Prefixtyps `t`
`t'LEFT`	linke Grenze des Prefixtyps `t`
`t'RIGHT`	rechte Grenze des Prefixtyps `t`
`t'HIGH`	obere Grenze des Prefixtyps `t`
`t'LOW`	untere Grenze des Prefixtyps `t`
`t'POS(x)`	Position (Index) des Elementes `x` im Prefixtyp `t`
`t'VAL(x)`	Wert des Elementes an Position `x` im Prefixtyp `t`
`t'SUCC(x)`	Nachfolger von `x` im Prefixtyp `t`
`t'PRED(y)`	Vorgänger von `x` im Prefixtyp `t`
`t'LEFTOF(x)`	Element links von `x` im Prefixtyp `t`
`t'RIGHTOF(x)`	Element rechts von `x` im Prefixtyp `t`
`t'ASCENDING`	`true` wenn `t` steigend indiziert ist
`t'IMAGE(x)`	Wert von `x` wird zur Zeichenkette `t`
`t'VALUE(x)`	Zeichenkette `x` wird zum Wert des Typs `t`

Beispiele für typbezogene Attribute

```
TYPE drei_bit is RANGE 0 TO 7;
TYPE drei_rbit is RANGE 7 DOWNTO 0;
TYPE wochentage IS (mo, di, mi, do, fr, sa, so);
VARIABLE a : drei_bit;   VARIABLE b : drei_rbit;
VARIABLE c : wochentage; VARIABLE d : boolean;
VARIABLE i : integer;
...
a := drei_bit'LEFT;         -- a = 0
a := drei_bit'RIGHT;        -- a = 7
b := drei_rbit'LEFT;        -- b = 7
b := drei_rbit'RIGHT;       -- b = 0
a := drei_bit'HIGH;         -- a = 7
b := drei_rbit'HIGH;        -- b = 7
i := wochentage'POS(mi);    -- i = 2
c := wochentage'VAL(3);     -- c = do
d := drei_rbit'ASCENDING; -- d = false
```

Feldbezogene Attribute

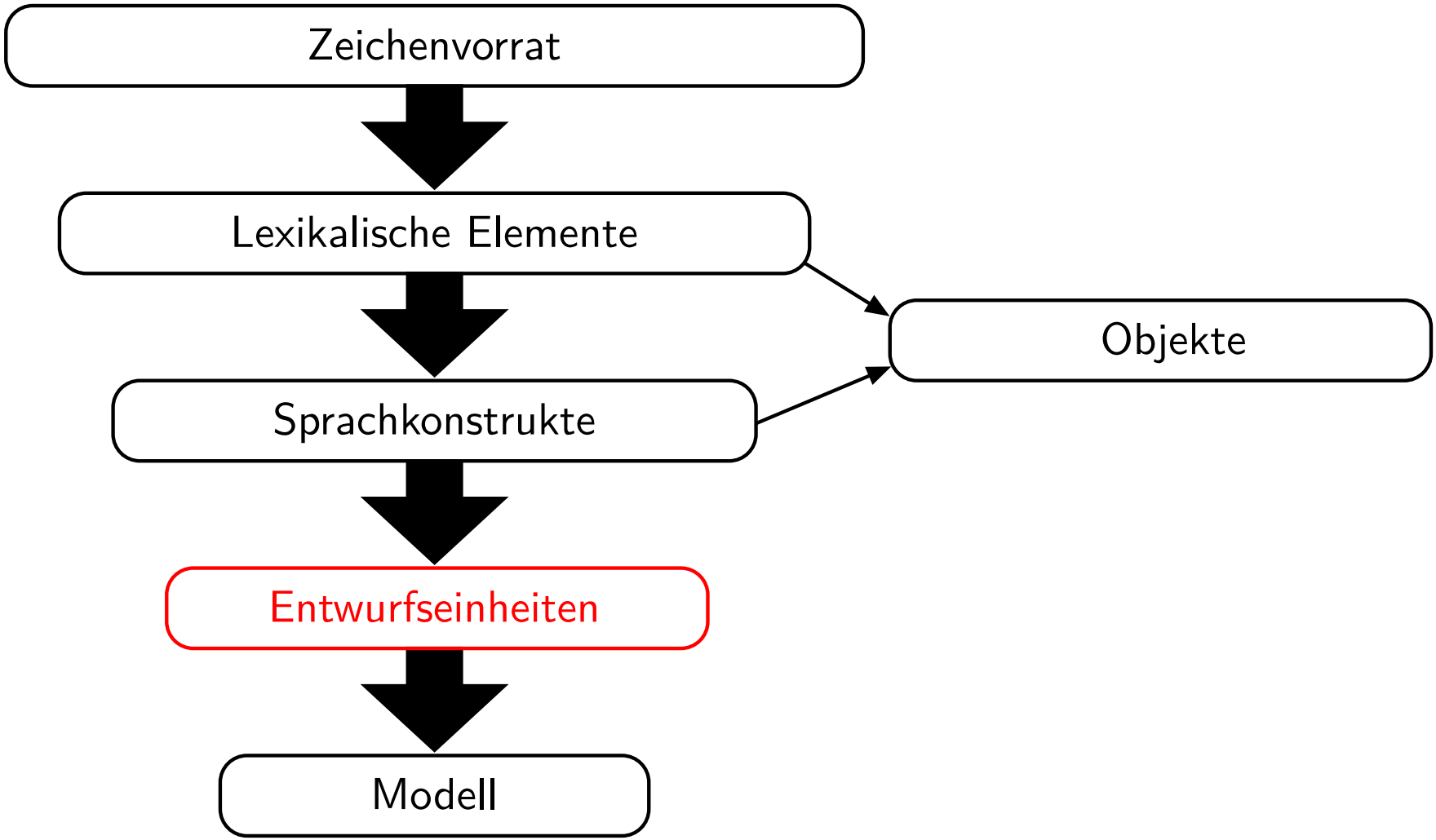
`a'LEFT(n)`	linke Grenze der *n*. Dimension von a
`a'RIGHT(n)`	rechte Grenze der *n*. Dimension von a
`a'HIGH(n)`	obere Grenze der *n*. Dimension von a
`a'LOW(n)`	untere Grenze der *n*. Dimension von a
`a'LENGTH(n)`	Bereichslänge der *n*. Dimension von a
`a'RANGE(n)`	Bereich der *n*. Dimension von a
`a'REVERSE_RANGE(n)`	Bereich der *n*. Dimension von a in umgekehrter Reihenfolge
`a'ASCENDING(n)`	`true` wenn a in der *n*. Dimension steigend indiziert ist

In VHDL-2008 wurden einige wenige neue Attribute eingeführt (hier nicht betrachtet).

Beispiele für feldbezogene Attribute

```
TYPE zwei_d IS ARRAY (1 TO 4, 31 DOWNTO 0) of boolean;
VARIABLE a : integer := 0;
VARIABLE b : boolean;
...
a := zwei_d'LEFT(1);                -- a = 1
a := zwei_d'LOW(1);                 -- a = 1
a := zwei_d'RIGHT(2);               -- a = 0
a := zwei_d'HIGH(2);                -- a = 31
a := zwei_d'LENGTH(1);              -- a = 4
a := zwei_d'LENGTH(2);              -- a = 32
b := zwei_d'ASCENDING(1);           -- b = true
b := zwei_d'ASCENDING(2);           -- b = false
```

Einordnung


Zeichenvorrat
Lexikalische Elemente
Objekte
Sprachkonstrukte
Entwurfseinheiten
Modell

Darstellungskonventionen

- „`[Konstrukt]`“ bedeutet, dass `Konstrukt` optional ist
- „`{Konstrukt}`“ bedeutet, dass `Konstrukt` beliebig oft wiederholt werden darf (auch 0 Mal!)
- Einige Darstellungsmöglichkeiten sind bewusst weggelassen (teilweise aus Platz- und Zeitgründen, teilweise weil sie für die Synthese hinderlich sind)

➥ Eine umfassende Darstellung aller Möglichkeiten mit exakter Syntax findet man in LRM *(Language Reference Manual)*, in den Hilfedateien der Entwurfswerkzeuge bzw. in

Peter J. Ashenden, *Designer's Guide to VHDL*, 3rd edition, Morgan Kaufmann Publishers 2008

Schnittstellenbeschreibung eines Modells

```
ENTITY entity_name IS
  [generics]
  [port declaration]
  [local declarations] -- Deklarationsteil
[BEGIN
  passive statements]
END [ENTITY] [entity_name];
```

Generics: Möglichkeiten der Modellparametrisierung

Port declaration: Eigentliche Schnittstellendefinition

Local declarations: USE-Anweisungen, Typendeklarationen, Aliases, Konstanten, Unterprogramme, Attribute

Passive statements: Prozeduren, Prozesse usw. ohne Signalzuweisung (selten benutzt)

Generics und Portdeklaration

Generics:

```
GENERIC (generic_list : type [:= Initialwert]
         {; generic_list : type [:= Initialwert]});
```

Portdeklaration:

```
PORT  (port_list : port_mode type [:= Initialwert]
    {; port_list : port_mode type [:= Initialwert]});
```

`port_mode` deklariert den Zugriffsmodus der entsprechenden Ports:

`IN`	Eingang (nur lesen)
`OUT`	Ausgang (nur schreiben, seit VHDL-2008 auch lesen)
`INOUT`	Bidirektional (lesen und schreiben)
`BUFFER`	beliebig lesen, nur von einer Quelle (VHDL'93) schreiben (Nutzung nicht empfehlenswert)

Eine Portdeklaration ist eine implizite Signaldeklaration, alle Portbezeichner sind im entsprechenden Modell als Signale sichtbar.

Eine ENTITY mit lokalen Deklarationen

```
ENTITY ROM_example IS
PORT (addr : IN  bit_vector(14 DOWNTO 0);
      data : OUT bit_vector(7 DOWNTO 0);
      enable : IN bit);
SUBTYPE rom_data_byte IS bit_vector (7 DOWNTO 0);
TYPE rom_content_array IS
    ARRAY (0 TO 2**15-1) OF rom_data_byte;
CONSTANT rom_content : rom_content_array :=
  (X"43", X"4F", X"51", -- ADD R15, R5, R1
   X"31", X"43",        -- LOAD R1, $43
   ... );
END ENTITY ROM_example;
```

Die Konstante `rom_content` kann z. B. in der entsprechenden ARCHITECTURE-Beschreibung zur Initialisierung des Speichers genutzt werden.

Ein komplexeres Beispiel

```
ENTITY und_gatter IS
GENERIC (verz_in : time := 3.5 ns;
         verz_out: time := 4 ns);
PORT (in_a, in_b : IN  bit;
      out_c      : OUT bit);
TYPE tristate IS ('0', '1', 'Z');
BEGIN
ASSERT ((in_a ='1') AND (in_b = '1')) REPORT
  "Ausgang ist 0" SEVERITY note;
END ENTITY und_gatter;
```

Anmerkung: `ASSERT` beschreibt eine sogenannte **Assertion**, eine Bedingung, die vom Simulator überprüft wird und bei deren **Nichterfüllung** bestimmte Aktionen unternommen werden können (eine Meldung ausgeben, Simulation stoppen).

Anmerkungen zu den Port-Modi

- Der Modus `INOUT` soll nur dann verwendet werden, wenn die entsprechenden Leitungen tatsächlich bidirektional betrieben werden. Bei der Synthese solcher Ports werden Tristate-Buffer erzeugt, d. h. die Zielhardware muss diese auch bereitstellen können! Das ist meistens nur bei den E/A-Pins realisierbar (oder erwünscht), nicht jedoch bei internen Verbindungen. Wenn die Funktionalität es nicht zwingend vorschreibt, kann eine bidirektionale Verbindung in zwei unidirektionale Verbindungen aufgetrennt werden.
- Der Modus `BUFFER` kann bei der Simulation und Synthese zu Problemen führen und sollte daher vermieden werden (z. B. ein internes Signal erzeugen und Werte über dieses austauschen). Das Problem ist mit VHDL-2008 beseitigt (aber noch nicht von allen CAD-Werkzeugen umgesetzt).

Die eigentliche Funktionalität eines Modells

```
ARCHITECTURE architecture_name OF entity_name IS
  [local declarations] -- Deklarationsteil
BEGIN
  statements
END [ARCHITECTURE] [architecture_name];
```

Deklarationsteil: USE-Anweisungen, Typendeklarationen, Aliases, Konstanten, **Signale,** Unterprogramme, Attribute, Definition von Unterprogrammen und Attributen

Statements: Nebenläufige Anweisungen (Prozesse, Prozedurrufe, Signalzuweisungen, Komponenteninstanziierungen, ...) und sequentielle Anweisungen (Schleifen, Verzweigungen, Prozedurrufe, ...)

Ein Beispiel

```
ARCHITECTURE arch1 OF und_gatter IS
  SIGNAL temp_a : bit;
BEGIN
  temp_a <= in_a AND in_b;
  out_c <= temp_a AFTER verz_out;
  ASSERT (temp_a = '0') REPORT
    "Ausgang ist 1" SEVERITY note;
END ARCHITECTURE arch1;
```

Bei der Synthese wird `temp_a` zur direkten Verbindung zwischen dem Ausgang des UND-Gatters und dem Port `out_c`, da es selbst keine logische Verknüpfung und keine Speicherfunktion realisiert. Der `statements`-Teil der ARCHITECTURE kann sehr umfangreich sein und mit sehr vielen verschiedenen Ausdrucksmitteln realisiert werden, auf die in einem späteren Vorlesungsabschnitt eingegangen wird.

Parametrisierung und Instantiierung

CONFIGURATION stellt einzelne Modellparameter des Modells ein und ordnet den Schnittstellen entsprechende Implementierungen zu:

```
CONFIGURATION configuration_name OF entity_name IS
  [USE statements]
  [ATTRIBUTE assignments]
  [configuration statements]
END [CONFIGURATION] [configuration_name];
```

USE statements: generelle USE-Anweisungen

ATTRIBUTE statements: Deklaration und Definition benutzerdefinierter Attribute

Configuration statements: Modell- und blockspezifische Konfigurationsangaben `FOR...USE`, z. B. Komponenteninstantiierung

CONFIGURATION für ein einfaches Modell

Zuordnung der ARCHITECTURE zu ENTITY:

```
CONFIGURATION configuration_name OF entity_name IS
  FOR architecture_name
    [block configuration statements]
    [component configuration statements]
  END FOR;
END [CONFIGURATION] [configuration_name];
```

Block and component configuration statements können zusätzlich angegeben werden, falls die Architektur weitere Blöcke oder Komponenten enthält:

```
FOR inst_name_1 {, inst_name_x} : component_name
  USE ENTITY entity_name [(architecture_name)]
   [GENERIC MAP (...)]
   [PORT MAP (...)];
END FOR;
```

Ein Beispiel

```
ENTITY und_gatter IS
  GENERIC (verz_t : time);
  PORT (in_a, in_b : IN bit;
        out_c      : OUT bit);
END ENTITY und_gatter;

ARCHITECTURE arch_1 OF und_gatter IS
...

CONFIGURATION conf_und OF und_gatter
  FOR arch_1
  END FOR;
END CONFIGURATION und_gatter;
```

Komplexere Beispiele von Konfigurationen werden beim Thema „Strukturale Modellierung“ besprochen.

Default binding

Verhalten bei (teilweise) fehlender CONFIGURATION (*default binding*):

- gibt es nur eine ARCHITECTURE mit zur ENTITY passender Bezeichnung, so wird diese eingesetzt
- gibt es mehrere ARCHITECTURE mit zur ENTITY passender Bezeichnung, so wird meistens die zuletzt übersetzte eingesetzt (systemabhängig)
- gibt es keine ARCHITECTURE mit zur ENTITY passender Bezeichnung, so wird eine Fehlermeldung erzeugt (keine *default binding* möglich)

Das gilt auch dann, wenn die Konfigurationsanweisung nur für eine oder mehrere Komponenten fehlt. Im letzten Fall (keine *default binding* möglich), erfolgt jedoch keine Fehlermeldung, sondern die Komponente wird ungebunden (*unbound*, als Platzhalter) gelassen (kann auch explizit mit `USE OPEN` erfolgen).

Deklarationen

PACKAGE fasst Objekte zusammen, die an mehreren Stellen im Entwurf benötigt werden oder von mehreren Entwürfen verwendet werden sollen (nur Deklarationen):

```
PACKAGE package_name IS
  [USE statements]
  [Declarations]
  [Definitions]
END [PACKAGE] [package_name];
```

USE statements: generelle USE-Anweisungen

Declarations: Deklarationen von Typen, Untertypen, Aliases, Konstanten, Signalen, Dateien, Komponenten, Unterprogrammen, Attributen (z. B. `ATTRIBUTE kodierung : bit_vector;`)

Definitions: Nur Attribute, z. B. `ATTRIBUTE kodierung OF zustand_0 : LITERAL IS b"0000";`

PACKAGE BODY

PACKAGE BODY beschreibt die Implementierung von Objekten, die vom PACKAGE bereitgestellt werden:

```
PACKAGE BODY package_name IS
  [USE statements]
  [Declarations]
  [Definitions]
END [PACKAGE BODY] [package_name];
```

USE statements: generelle USE-Anweisungen

Declarations: Deklarationen von Typen, Untertypen, Aliases, Konstanten, Signalen, Dateien, Komponenten, Unterprogrammen

Definitions: Implementierung von Unterprogrammen (Code)

Ein Beispiel

```
PACKAGE dies_und_das IS
  TYPE tristate IS ('0', '1', 'Z');
  CONSTANT verz_1 : time;
  FUNCTION a_pl_b (a, b : integer) RETURN integer;
END PACKAGE dies_und_das;

PACKAGE BODY dies_und_das IS
  CONSTANT verz_1 : time := 3.2 ns;
  FUNCTION a_pl_b (a, b : integer) RETURN integer IS
    VARIABLE temp : integer := 0;
    BEGIN
      temp := a + b;
      RETURN temp;
    END a_pl_b;
END PACKAGE BODY dies_und_das;
```

Fest verfügbare Packages

Einige Packages gehören zum festen Lieferumfang eines jeden VHDL-Systems:

standard: Einige gebräuchliche Datentypen und Operatoren. Muss nicht mit `LIBRARY`- und `USE`-Anweisungen sichtbar gemacht werden, da folgende Anweisungen für jede Entwurfseinheit automatisch (implizit) ausgeführt werden:

```
LIBRARY std, work;
USE std.standard.all;
```

IEEE 1164: Mehrwertige Logik inklusive entsprechender Funktionen. Wird mit `USE ieee.std_logic_1164.all;` eingebunden.

textio: Funktionen zum formatierten Arbeiten mit Dateien (wird später bei der Erläuterung von Testumgebungen näher betrachtet). Einbindung mit `USE std.textio.all;`

Auszug aus dem Standard-Package

```
TYPE boolean IS (false, true);
TYPE bit     IS ('0', '1');
TYPE character IS (...); -- siehe Zeichenvorrat
TYPE severity_level IS (note, warning, error, failure);
TYPE integer  IS RANGE ...; -- rechnerabhaengig
TYPE real  IS RANGE ...; -- rechnerabhaengig
SUBTYPE natural IS integer RANGE 0 TO integer'HIGH;
SUBTYPE positive IS integer RANGE 1 TO integer'HIGH;
TYPE time IS RANGE ... -- rechnerabhaengig
  UNITS fs; ps = 1000 fs; ns = 1000 ps; us = 1000 ns;
            ms = 1000 us; sec = 1000 ms; min = 60 sec;
            hr = 60 min; END UNITS;
TYPE string IS ARRAY (positive RANGE <>) OF character;
TYPE bit_vector IS ARRAY (natural RANGE <>) OF bit;
```

Standard-Funktionen

Funktionen, die auf den Typen aus dem Standard-Package definiert sind:

- Logische Verknüpfungsoperatoren für einzelne Bits und Bitvektoren: NOT, OR, AND, NOR, NAND, XOR, XNOR (seit VHDL'93)
- Vergleichsoperatoren
- Mathematische Operatoren für `integer` und `real`
- Multiplikation und Division für gemischte Operanden (`time` und `integer` bzw. `time` und `real`)
- Verkettungsoperator „&“ für die Datentypen `character`, `string`, `bit` und `bit_vector`
- Funktion `now` (aktuelle Simulationszeit)

Mehrwertige Logik mit IEEE 1164

- Der im `standard`-Package definierte Datentyp Bit lässt nur zwei mögliche Werte eines Objektes zu: '0' und '1' (klassische zweiwertige Logik)
- Bei der Simulation des realen Verhaltens von Signalen sind viele andere Objektwerte denkbar. Einige Zeit lang wurde dieser Situation durch herstellerspezifische Erweiterungen (Packages, Simulationsbibliotheken) Rechnung getragen

➥ Einführung eines Standards für mehrwertige Logik: IEEE 1164, Erweiterung des Wertevorrates um folgende Werte: Undefiniert, hochohmig, uninitialisiert, don't care, schwache 0, schwache 1, schwaches Undefiniert. Ergebnis: `std_logic_1164`-Package (9-wertige Logik)

Definition der Basisdatentypen

```
TYPE std_ulogic IS
  ( 'U',    -- Uninitialized, uninitialisiert
    'X',    -- Unknown, undefiniert
    '0',    -- Forcing 0, starke 0
    '1',    -- Forcing 1, starke 1
    'Z',    -- High impedance, hochohmig
    'W',    -- Weak unknown, schwaches undefiniert
    'L',    -- Weak 0, schwache 0
    'H',    -- Weak 1, schwache 1
    '-' );  -- Don't care
TYPE std_ulogic_vector IS ARRAY (natural RANGE <>) OF
                                                  std_ulogic;
FUNCTION resolved (s : std_ulogic_vector) RETURN std_ulogic;
TYPE std_logic IS resolved std_ulogic;
TYPE std_logic_vector IS ARRAY (natural RANGE <>) OF std_logic;
SUBTYPE X01 IS resolved std_ulogic RANGE 'X' TO '1';
SUBTYPE X01Z IS resolved std_ulogic RANGE 'X' TO 'Z';
SUBTYPE UX01 IS resolved std_ulogic RANGE 'U' TO '1';
SUBTYPE UX01Z IS resolved std_ulogic RANGE 'U' TO 'Z';
```

Physikalische Entsprechung

'0', '1', 'X'	„starke Werte“; Technologien, die „High“ und „Low“ aktiv treiben (z. B. CMOS)
'L', 'H', 'W'	„schwache“ Werte; Technologien, die Ausgangsstufen schwach treiben (z. B. NMOS oder PMOS mit Widerstand als Lastelement)
'Z'	tristate-Ausgänge
'U' und '–'	nicht initialisierte Signale und *don't cares*
X01	nur starke Werte
X01Z	nur starke und hochohmige Werte
UX01	nur starke und nicht initialisierte Werte
UX01Z	nur starke, nicht initialisierte und hochohmige Werte

Bei der Definition wird die Funktion `resolved` gebraucht:

```
FUNCTION resolved (s : std_ulogic_vector)
                                    RETURN std_ulogic;
```

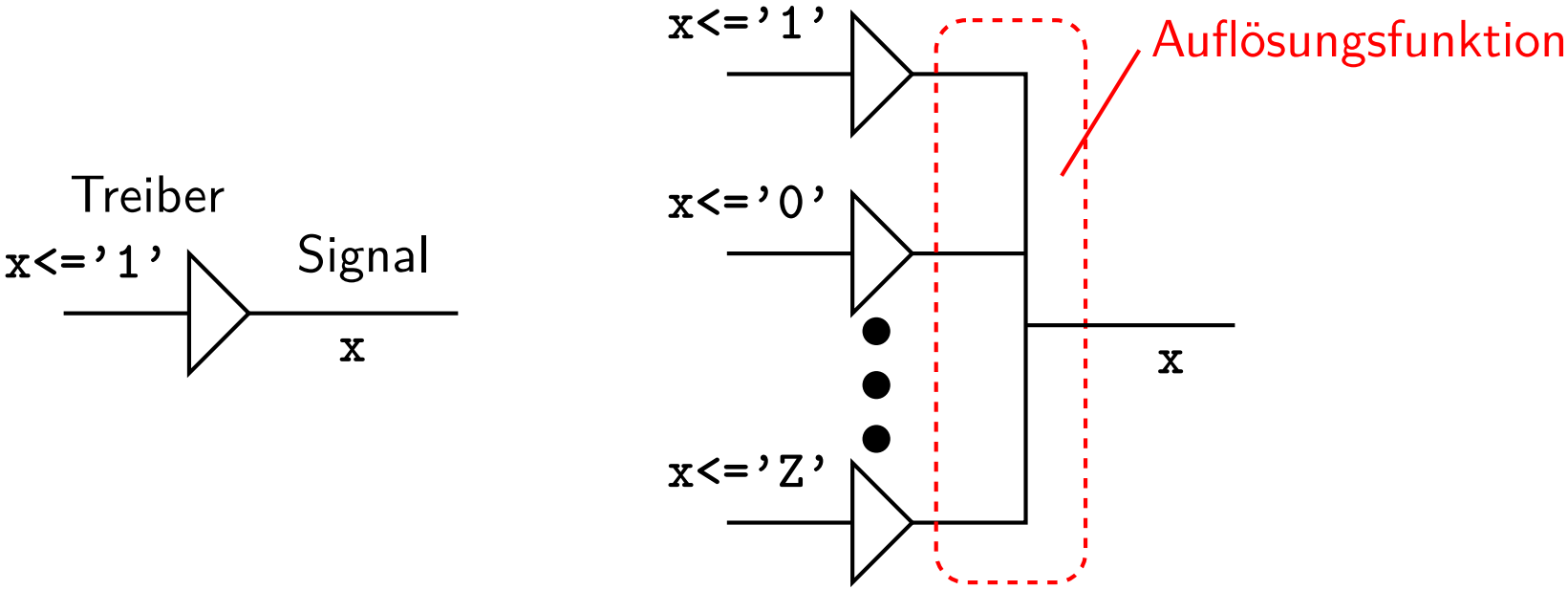
Signale, Treiber und Auflösungsfunktionen

Signale werden modellintern durch **Treiber** gesteuert. Ein Treiber ist eine Quelle im Modell, die eine Wertzuweisung an ein Signal vornimmt. Es sind dabei zwei Varianten möglich:

- Ein Signal hat nur einen Treiber. In diesem Fall wird das Signal immer den durch diesen Treiber erzeugten Wert zugewiesen bekommen.
- Ein Signal hat (gewollt oder ungewollt) mehrere Treiber. Der Wert des Signals wird durch Überlagerung der Treiberwerte mittels einer **Auflösungsfunktion** *(resolution function)* ermittelt. Ist für den Datentyp des Signals keine Auflösungsfunktion definiert, so wird eine Fehlermeldung erzeugt. Datentypen mit bzw. ohne Auflösungsfunktion heißen entsprechend **aufgelöst** *(resolved)* bzw. **unaufgelöst** *(unresolved)*.

`resolved` (Definition von `std_logic`) ist eine Auflösungsfunktion.

Konzept der Treiber und Auflösungsfunktionen



Prinzipiell können alle Datentypen aufgelöst deklariert werden, aus Effizienzgründen wird jedoch darauf verzichtet (der Simulator ruft intern die Auflösungsfunktion bei jeder Zuweisung eines Wertes an ein Objekt vom aufgelösten Typ auf).

- Hat ein Signal vom Typ z. B. `bit` mehrere Treiber, wird eine Fehlermeldung erzeugt (unaufgelöster Typ).

Auflösungsfunktion in IEEE 1164 (Konstantentabelle)

```
TYPE stdlogic_table IS ARRAY (std_ulogic, std_ulogic) OF
                                                  std_logic;

CONSTANT resolution_table : stdlogic_table := (
-- ---------------------------------------------------
-- | U    X    0    1    Z    W    L    H    -       |   |
-- ---------------------------------------------------
   ('U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U'), -- | U |
   ('U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X'), -- | X |
   ('U', 'X', '0', 'X', '0', '0', '0', '0', 'X'), -- | 0 |
   ('U', 'X', 'X', '1', '1', '1', '1', '1', 'X'), -- | 1 |
   ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X'), -- | Z |
   ('U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X'), -- | W |
   ('U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X'), -- | L |
   ('U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X'), -- | H |
   ('U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X'));-- | - |
```

Auflösungsfunktion in IEEE 1164 (eigentliche Funktion)

```
FUNCTION resolved (s : std_ulogic_vector)
                    RETURN std_ulogic IS
  VARIABLE result : std_ulogic := 'Z';
                     -- weakest state default
BEGIN
  IF (s'LENGTH = 1) THEN
    RETURN s(s'LOW); -- single driver
  ELSE
    FOR i IN s'RANGE LOOP
      result := resolution_table(result, s(i));
    END LOOP;
  END IF;
  RETURN result;
END resolved;
```

Definitionsmöglichkeiten für Objekte mit aufgelösten Typen

- Aufgelösten Untertyp eines unaufgelösten Typs deklarieren, Objekt mit diesem Untertyp deklarieren:

```
SUBTYPE resolved_type_name
        IS resolution_funktion_name
           unresolved_type_name;
SIGNAL resolved_signal_name : resolved_type_name;
```

- Bei der Objektdeklaration einen unaufgelösten Typ verwenden und die Auflösungsfunktion mit angeben (diese muss natürlich definiert sein, z. B. in einem Package):

```
SIGNAL resolved_signal_name :
           resolution_funktion_name
           unresolved_type_name;
```

Weitere Bestandteile des IEEE 1164 Packages

- Logische Verknüpfungsoperatoren für mehrwertige Logik: NOT, OR, AND, NOR, NAND, XOR
- Konvertierungsfunktionen
  - Zwischen den Typen innerhalb des Package
  - Zwischen den Typen des Packages und `bit` sowie `bit_vector`

  `To_bit()`, `To_bitvector()`, `To_StdULogicVector()`, ...
- Zustands- und Ereignisdetektion:

`rising_edge(SIGNAL s : std_ulogic) RETURN boolean;`
(`true`, wenn steigende Flanke an `s` vorliegt),
`falling_edge(SIGNAL s: std_ulogic) RETURN boolean;`
(`true`, wenn fallende Flanke an `s` vorliegt),
`Is_X(s : std_(u)logic(_vector)) RETURN boolean;`
(`true`, wenn ein Wert undefiniert ist)

Operanden am Beispiel der Invertierungsfunktion

```
TYPE stdlogic_1d IS ARRAY (std_ulogic) OF std_ulogic;

CONSTANT not_table: stdlogic_1d :=
-- ---------------------------------------------
-- | U    X    0    1    Z    W    L    H    -  |
-- ---------------------------------------------
   ('U', 'X', '1', '0', 'X', 'X', '1', '0', 'X');

FUNCTION "NOT"  (l : std_ulogic)
                    RETURN UX01 IS
BEGIN
  RETURN (not_table(l));
END "NOT";
```

Ein Beispiel für Is_X()-Funktion

```
FUNCTION Is_X  (s : std_ulogic) RETURN boolean IS
BEGIN
  CASE s IS
    WHEN 'U' | 'X' | 'Z' | 'W' | '-' => RETURN true;
    WHEN OTHERS => NULL;
  END CASE;
  RETURN false;
END Is_X;
```

Weitere Is_X-Funktionen:

```
FUNCTION Is_X  (s : std_ulogic_vector)
                    RETURN boolean IS
...
FUNCTION Is_X  (s : std_logic_vector)
                    RETURN boolean IS
...
```

Weitere IEEE-Packages

IEEE-Bibliothek liefert eine Anzahl weiterer Packages, die viele Entwurfsaufgaben vereinfachen:

`math_real`	Operationen mit reellen Zahlen
`math_complex`	Operationen mit komplexen Zahlen
`numeric_bit`	Mathematische Operationen mit Standard-Bitvektoren
`numeric_std`	Mathematische Operationen mit Bitvektoren (mehrwertige Logik)
`numeric_bit_unsigned`	Wie oben, jedoch vorzeichenlos (nur ab VHDL-2008)
`numeric_std_unsigned`	Wie oben, jedoch vorzeichenlos (nur ab VHDL-2008)
`fixed_generic_pkg`	Mathematische Operationen mit Festkomma-Zahlen (nicht zu verwechseln mit ganzen Zahlen!)
`float_generic_pkg`	Mathematische Operationen mit Gleitkomma-Zahlen

Zusammenfassung

- Aggregate und Attribute
- Ein tieferer Einblick in die Entwurfseinheiten
- Standard-Packages
- Mehrwertige Logik und Auflösungsfunktionen
- Implementierung von Auflösungsfunktion und einiger anderer Funktionen bei `std_logic`

Aufgaben zum Selbststudium

1. Beschreiben Sie in VHDL einen 2:1 Multiplexer. Dieser soll nur Ports von Typ `std_logic` besitzen. Modellieren Sie eine Verzögerungszeit, die zwischen einer Veränderung eines Eingangssignalwertes und der Sichtbarkeit ihrer Auswirkung am Ausgang vergeht. Falls mindestens einer der Eingangswerte undefiniert oder nicht initialisiert ist, soll der Simulator eine entsprechende Warnung erzeugen.
2. Modifizieren Sie das Modell des UND-Gatters so, dass es nur mit aufgelösten Typen `std_logic` arbeitet. Führen Sie einen dritten Eingang ein, der bei der Belegung '1' der Eingang `in_a` auf den Ausgang `out_c` durchschaltet und bei allen anderen Belegungen die UND-Verknüpfung der beiden Eingänge am Ausgang `out_c` realisiert. Vermeiden Sie dabei (trotz der Auflösungsmöglichkeit) mehrfache Treiber am `out_c`.

Aufgabenlösungen zur 4. Vorlesung

Aufgelöste Datentypen

Der entsprechende VHDL-Quellkode ist auf dem Handout-Server unter

`VHDL/Aufgabenloesungen/VHDL/vhdl_v4_mux2_to_1.vhd`

und

`VHDL/Aufgabenloesungen/VHDL/vhdl_v4_logik.vhd`

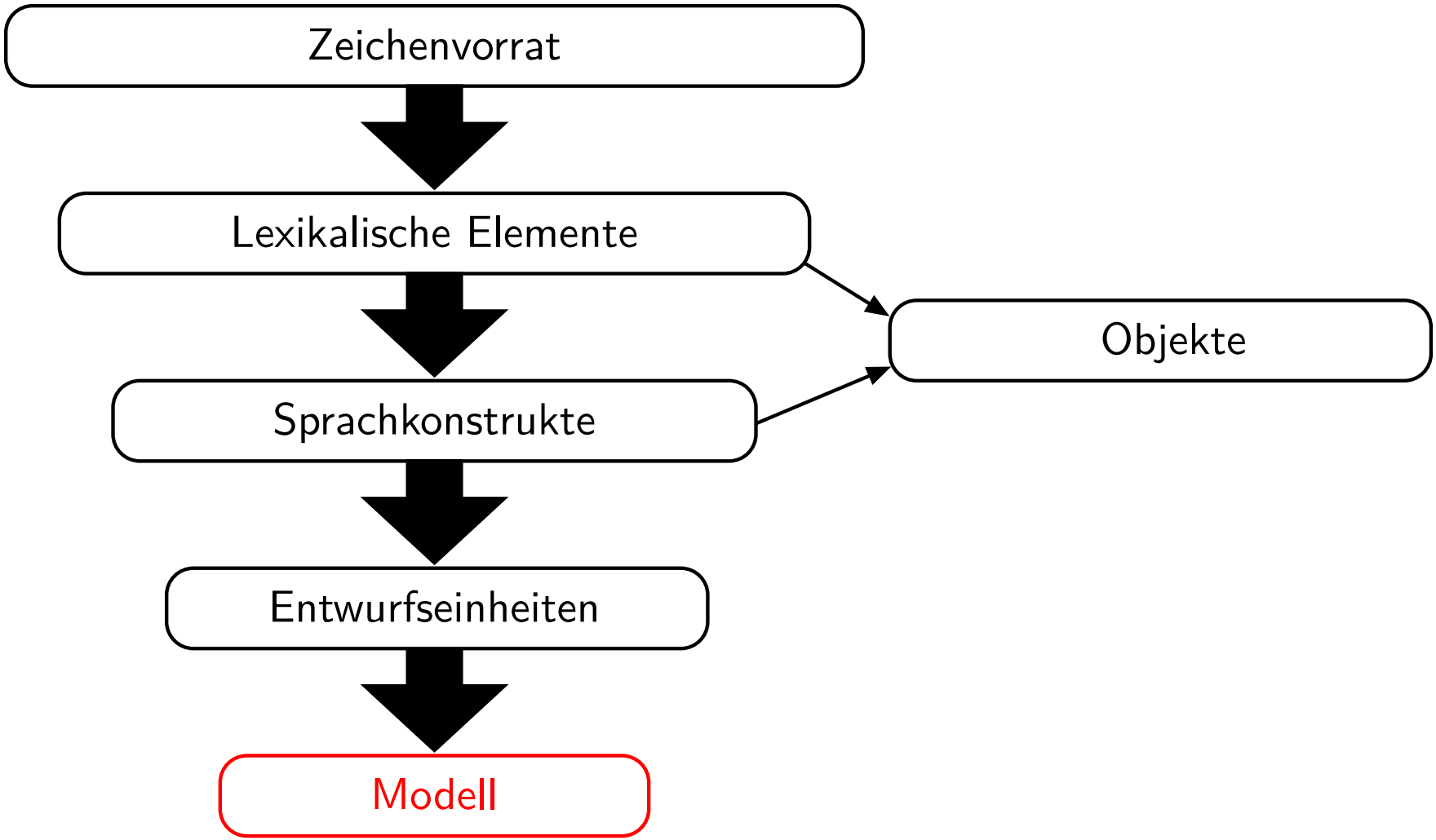
zu finden.

Systementwurf mit VHDL: Vorlesungs-Gesamtübersicht

Systementwurf mit VHDL: Gliederung der 5. Vorlesung

3. Modellierungstechniken

Einordnung


Zeichenvorrat
Lexikalische Elemente
Objekte
Sprachkonstrukte
Entwurfseinheiten
Modell

Einteilung

VHDL liefert sehr umfangreiche Beschreibungsmöglichkeiten für die Spezifikation, Modellierung und Synthese von Schaltungen und Systemen. Abhängig von der Aufgabenstellung, dem erwarteten Ergebnis sowie anderen Randbedingungen erweisen sich bestimmte VHDL-Beschreibungen für jeden spezifischen Fall als besser oder schlechter geeignet. Verschiedene Arten von VHDL-Beschreibungen werden in folgende Klassen zusammengefasst, die man **Modellierungstechniken** (auch Modellierungsstile, Codierungsstile) nennt:

- Strukturale Modellierung
- Funktionale Modellierung
- Verhaltensmodellierung
- Hierarchische Modellierung

Wesentliche Merkmale

- Aufbau des Modells aus einfacheren Submodellen (Blöcken, Komponenten), Nachbildung der Struktur
- Der ARCHITECTURE-Teil besteht aus zwei Hauptabschnitten:
  - Definitionen: Einbindung der Bibliotheken, Deklaration von Komponenten, Signalen, Konstanten, Typen
  - Netzliste: Instanziierung von Komponenten, Herstellung der Verbindungen zwischen den Komponenten (sowie zur Außenwelt)
- Beschreibung entspricht meistens tatsächlicher/gewünschter Struktur der physikalischen Hardware
- Vollsynthetisierbar (vorausgesetzt, die Komponenten liegen in synthesefähiger Form vor)

Beispiele von Basisblöcken für ein strukturales Modell

```
ENTITY und_gatter IS
  PORT (in_a, in_b : IN bit;
        out_c      : OUT bit);
END ENTITY und_gatter;
ARCHITECTURE arch OF und_gatter IS
BEGIN
  out_c <= in_a AND in_b;
END ARCHITECTURE arch;
ENTITY oder_gatter IS
  PORT (in_a, in_b : IN bit;
        out_c      : OUT bit);
END ENTITY oder_gatter;
ARCHITECTURE arch OF oder_gatter IS
BEGIN
  out_c <= in_a OR in_b;
END ARCHITECTURE arch;
```

Modell mit Basisblöcken und_gatter und oder_gatter

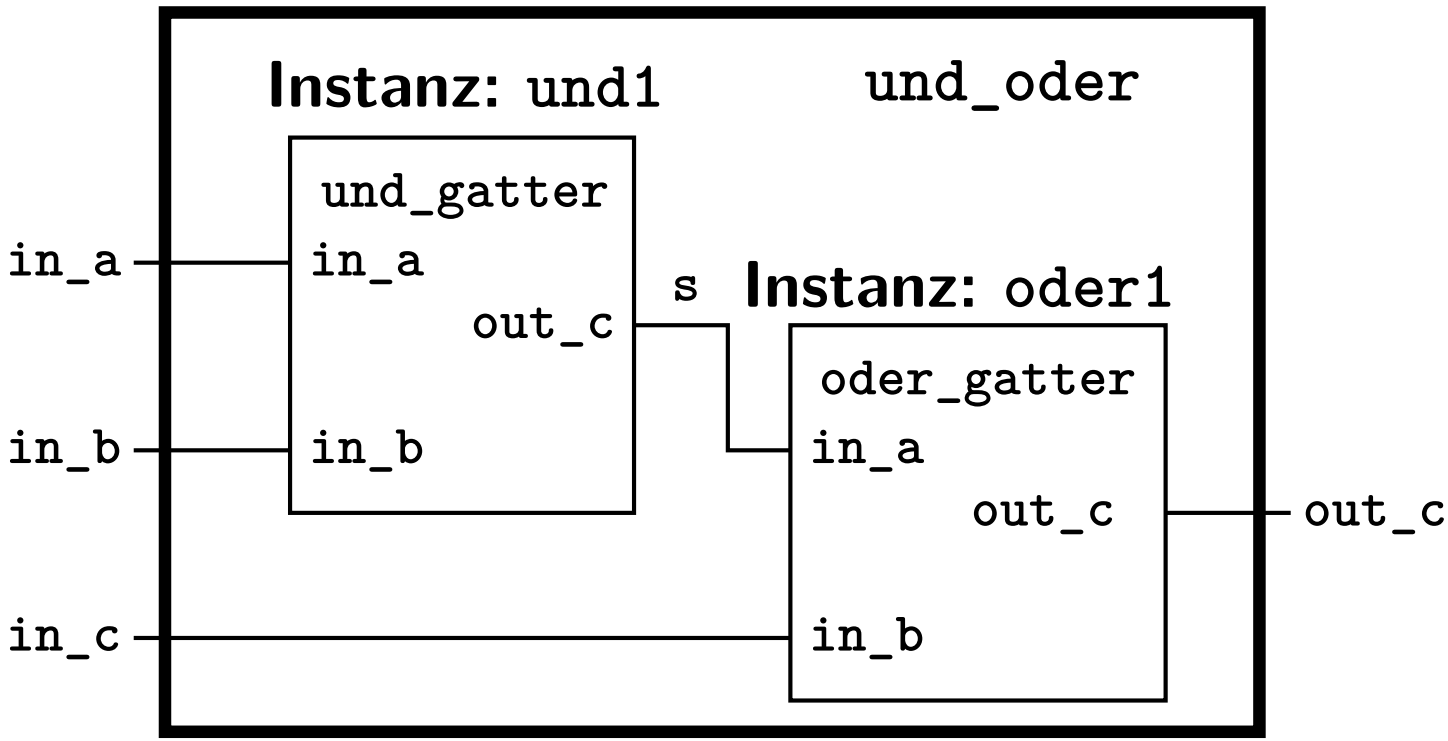
```
ENTITY und_oder IS
  PORT (in_a, in_b, in_c : IN bit;
        out_c       : OUT bit);
END ENTITY und_oder;
ARCHITECTURE struktur OF und_oder IS
  COMPONENT und_gatter
    PORT(in_a, in_b : IN bit;
         out_c      : OUT bit);
  END COMPONENT;
  COMPONENT oder_gatter
    PORT(in_a, in_b : IN bit;
         out_c      : OUT bit);
  END COMPONENT;
  SIGNAL s : bit;
BEGIN
  und1  : und_gatter PORT MAP(in_a => in_a, in_b => in_b,
                                            out_c => s);
  oder1 : oder_gatter PORT MAP(in_a => s, in_b => in_c,
END ARCHITECTURE struktur;                  out_c => out_c);
```

Passende CONFIGURATION

Im angegebenen Beispiel gibt es je eine ARCHITECTURE zu jeder ENTITY (bzw. Komponente), so dass die Standardbindung über die Bezeichner problemlos möglich ist. Für eine saubere Modellierung empfiehlt sich dennoch zusätzlich eine Konfiguration zu erzeugen:

```
CONFIGURATION und_oder_config OF und_oder IS
  FOR struktur
    FOR ALL : und_gatter
      USE ENTITY work.und_gatter(arch);
    END FOR;
    FOR ALL : oder_gatter
      USE ENTITY work.oder_gatter(arch);
    END FOR;
  END FOR;
END CONFIGURATION und_oder_config;
```

Graphische Darstellung der modellierten Struktur



und_gatter und oder_gatter sind Komponenten (COMPONENT), d. h. allgemeine Schnittstellenbeschreibungen, Platzhalter, Sockel. und1 und oder1 sind Instanzen dieser Komponenten. Eine Komponente darf mehrere Instanzen haben.

Konfiguration von Komponenten

- Mit Hilfe von CONFIGURATION-Entwurfseinheit (bereits gezeigt)
- USE–Anweisung direkt in der ARCHITECTURE-Entwurfseinheit:

```
ARCHITECTURE struktur OF und_oder IS
  COMPONENT und_gatter
    PORT(in_a, in_b : IN bit;
         out_c      : OUT bit);
  END COMPONENT;
  ...
  SIGNAL s : bit;
  FOR und1 : und_gatter USE ENTITY work.und_gatter(arch);
  FOR oder1 : oder_gatter USE ENTITY
BEGIN                                  work.oder_gatter(arch);
  und1  : und_gatter PORT MAP(in_a => in_a, in_b => in_b,
                                                 out_c => s);
  oder1 : oder_gatter PORT MAP(in_a => s, in_b => in_c,
END ARCHITECTURE struktur;                      out_c => out_c);
```

Direkte Instanziierung

Seit VHDL'93 ist eine direkte Instanziierung *(direct instantiation)* möglich, bei der eine explizite Komponenten-Deklaration entfallen kann:

```
ENTITY und_oder IS
  PORT (in_a, in_b, in_c : IN bit;
        out_c       : OUT bit);
END ENTITY und_oder;
ARCHITECTURE struktur OF und_oder IS
  SIGNAL s : bit;
BEGIN
  und1  : ENTITY work.und_gatter(arch)
      PORT MAP(in_a => in_a, in_b => in_b, out_c => s);
  oder1 : ENTITY work.oder_gatter(arch)
      PORT MAP(in_a => s, in_b => in_c, out_c => out_c);
END ARCHITECTURE struktur;
```

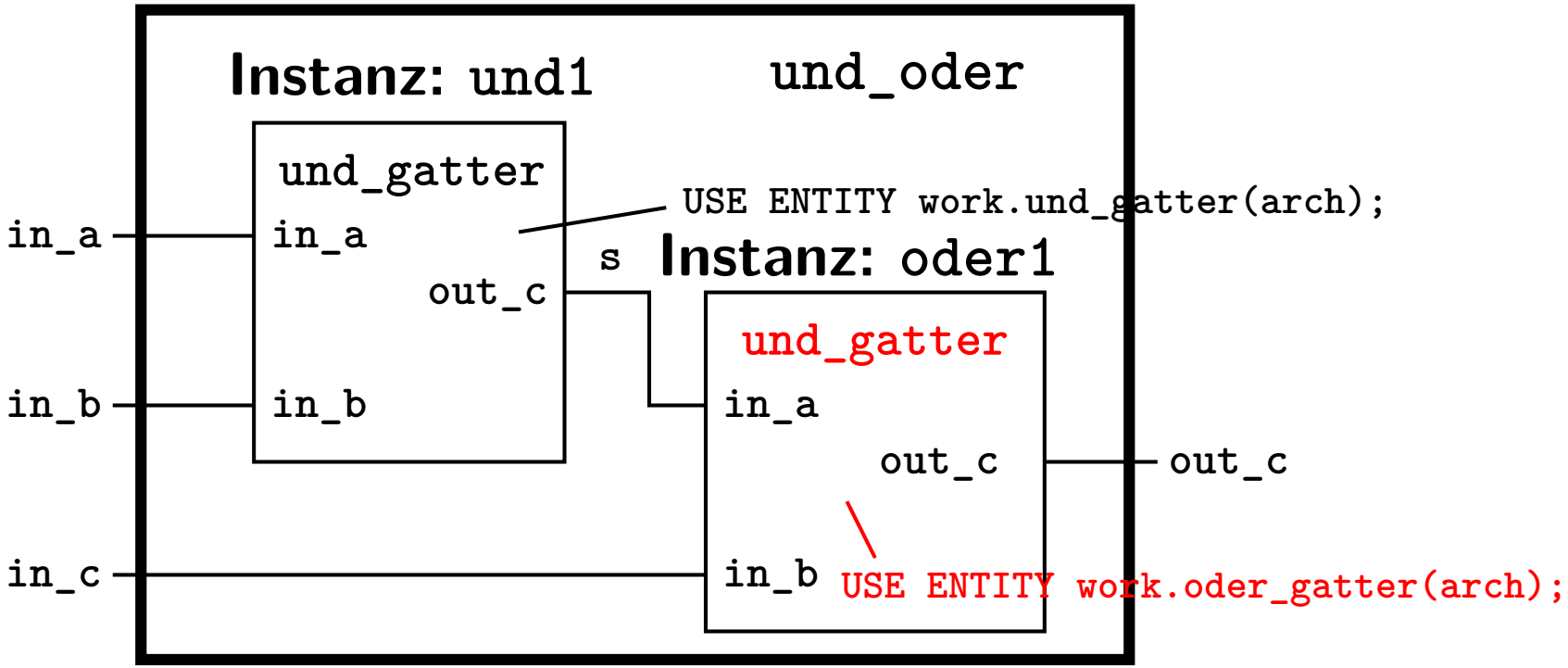
Die Voraussetzung ist ebenfalls, dass und_gatter sowie oder_gatter ins Projekt eingebunden sind (übersetzt, in der WORK-Library abgelegt).

Eine weitere Beschreibungsmöglichkeit

Nicht elegant, aber gültig, da Komponentenschnittstellen gleich sind:

```
ENTITY und_oder IS
  PORT (in_a, in_b, in_c : IN bit;
        out_c       : OUT bit);
END ENTITY und_oder;
ARCHITECTURE struktur OF und_oder IS
  COMPONENT und_gatter
    PORT(in_a, in_b : IN bit;
         out_c      : OUT bit);
  END COMPONENT;
  SIGNAL s : bit;
  FOR und1 : und_gatter USE ENTITY work.und_gatter(arch);
  FOR oder1 : und_gatter USE ENTITY work.oder_gatter(arch);
BEGIN
  und1  : und_gatter PORT MAP(in_a => in_a, in_b => in_b,
                                                out_c => s);
  oder1 : und_gatter PORT MAP(in_a => s, in_b => in_c,
END ARCHITECTURE struktur;                      out_c => out_c);
```

Graphische Darstellung des letzten Modells



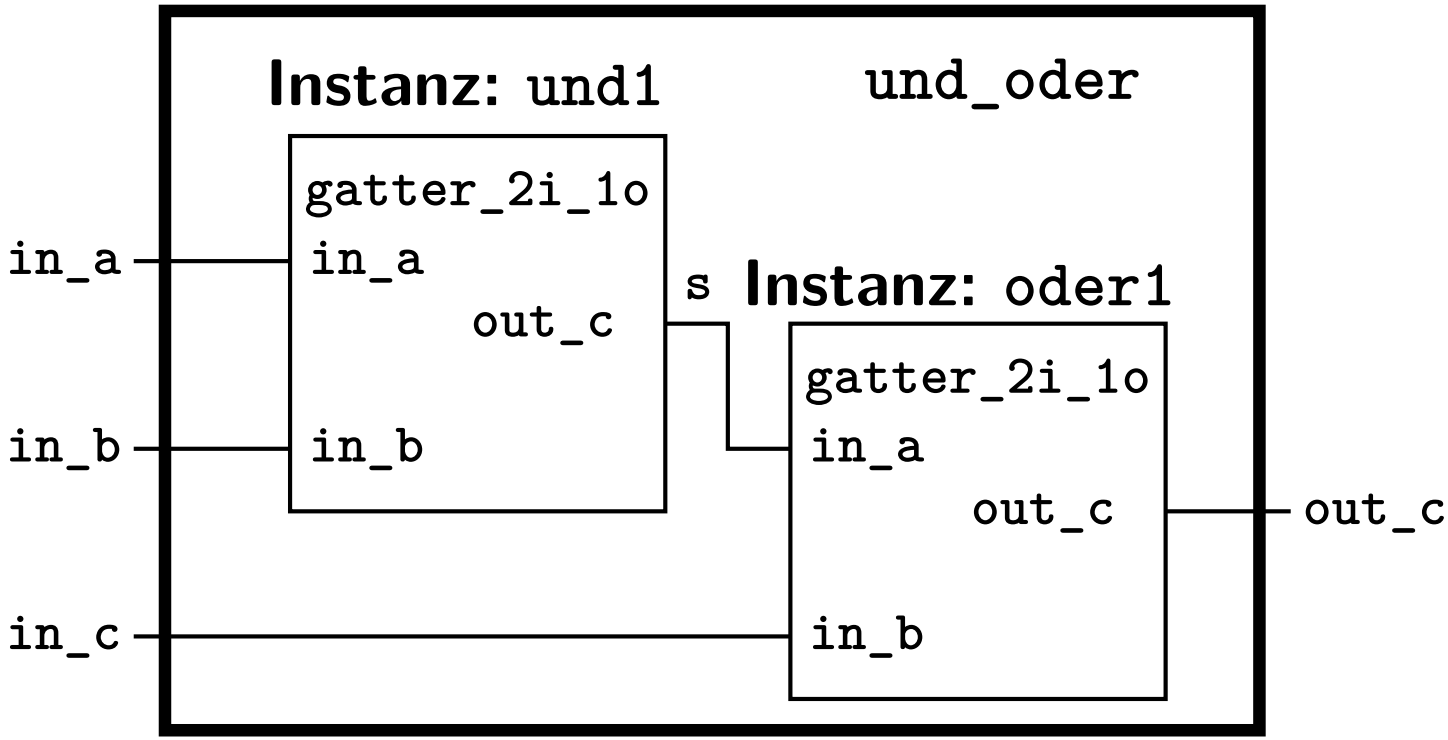
Komponenten sind gleich (Instanzen unterscheiden sich durch Namen und `PORT MAP`), bei der Konfiguration kann jedoch beim `oder1` die Implementierung einer ODER-Verknüpfung eingesetzt werden (obwohl der Name der Komponente etwas anderes vermuten lässt).

Konsistente Namensgebung der Komponenten

Zur Vermeidung der Verwirrung sollte man „COMPONENT und_gatter" in „COMPONENT gatter_2i_1o" umbenennen:

```
ENTITY und_oder IS
  PORT (in_a, in_b, in_c : IN bit;
        out_c       : OUT bit);
END ENTITY und_oder;
ARCHITECTURE struktur OF und_oder IS
  COMPONENT gatter_2i_1o
    PORT(in_a, in_b : IN bit;
         out_c      : OUT bit);
  END COMPONENT;
  SIGNAL s : bit;
  FOR und1 : gatter_2i_1o USE ENTITY work.und_gatter(arch);
  FOR oder1 : gatter_2i_1o USE ENTITY work.oder_gatter(arch);
BEGIN
  und1  : gatter_2i_1o PORT MAP(in_a => in_a, in_b => in_b,
                                out_c => s);
  oder1 : gatter_2i_1o PORT MAP(in_a => s, in_b => in_c,
END ARCHITECTURE struktur;                    out_c => out_c);
```

Graphische Darstellung nach der letzten Änderung



Prinzipiell kann jedes Modell mit passender Schnittstelle (2 Eingänge und 1 Ausgang jeweils von Typ `bit`) als Instanz der Komponente `gatter_2i_1o` eingesetzt werden.

- ➥ Komponente ist ein Platzhalter zur Abstraktion der Implementierung eines Teils des strukturalen Modells.

COMPONENT-Konstrukt

In den letzten Beispielen wurden bereits mehrfach **Komponenten** verwendet. Diese werden mit der `COMPONENT`-Konstruktion beschrieben:

```
COMPONENT component_name
  [GENERIC ( param_1 {, param_n} :
                type_name [:= def_value]
  { ; further_generic_declarations} ); ]
  [ PORT (
      port_declarations); ]
END COMPONENT;
```

Port-Beschreibung erfolgt mit gleicher Syntax wie bei ENTITY (entsprechend auch die Modi IN, OUT, INOUT, BUFFER);

Die Implementierung einer Komponente ist durch entsprechendes ENTITY-ARCHITECTURE-Paar beschrieben (kann bei Bedarf durch CONFIGURATION ergänzt werden).

Benutzung von Komponenten

- Deklaration erfolgt im Deklarationsteil der ARCHITECTURE (zusammen mit Signalen, Typen usw.)
- Instantiierung erfolgt im Definitionsteil der ARCHITECTURE (nach `BEGIN`):

```
instance_name : component_name
  [ GENERIC MAP (...) ]
  [ PORT MAP (...)];
```

- Konfiguration erfolgt entweder in einer separaten CONFIGURATION-Einheit oder in der ARCHITECTURE mit Hilfe der `FOR-USE`-Anweisungen (im Deklarationsteil)
- Sollen bestimmte Instanzen nicht belegt werden, kann das mit der Konstruktion `FOR instance_name : component_name USE OPEN;` erreicht werden (Probleme bei der Synthese).

Begriffssystematik bei Signalzuordnungen

Formals, *locals* und *actuals*

Ports und Generics in der ENTITY werden als *formals* bezeichnet. Entsprechend sind Ports und Generics in der COMPONENT *locals* und die lokale Signale in der ARCHITECTURE *actuals*.

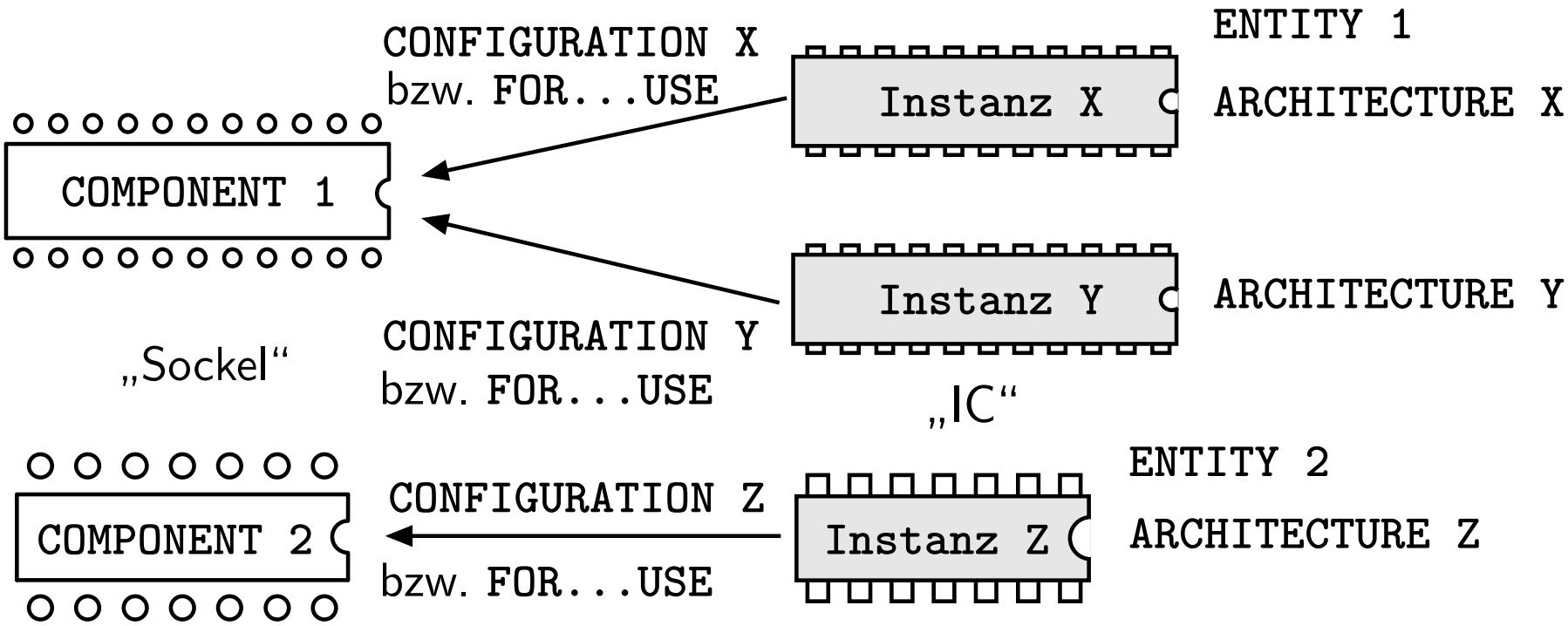
Die Komponenteninstantiierung ordnet den *locals* die *actuals* zu. Das erfolgt in der `GENERIC MAP` und `PORT MAP` auf eine der beiden Arten:

- Positionell *(positional association)*:
  `PORT/GENERIC MAP (actual_1, ..., actual_n);`
- Namentlich (*named association*, wie in allen bisherigen Beispielen):

```
PORT/GENERIC MAP (local_1 => actual_1,
                  ...
                  local_2 => actual_n);
```

Ein `PORT` oder `GENERIC MAP` kann auch positionell beginnen und namentlich abschließen (Mischform).

Veranschaulichung des Komponenten-Konzeptes



Die Struktur ist beschrieben durch Deklaration und Instanziierung von Komponenten (die ihrerseits in den entsprechenden Entwurfseinheiten implementiert sind).

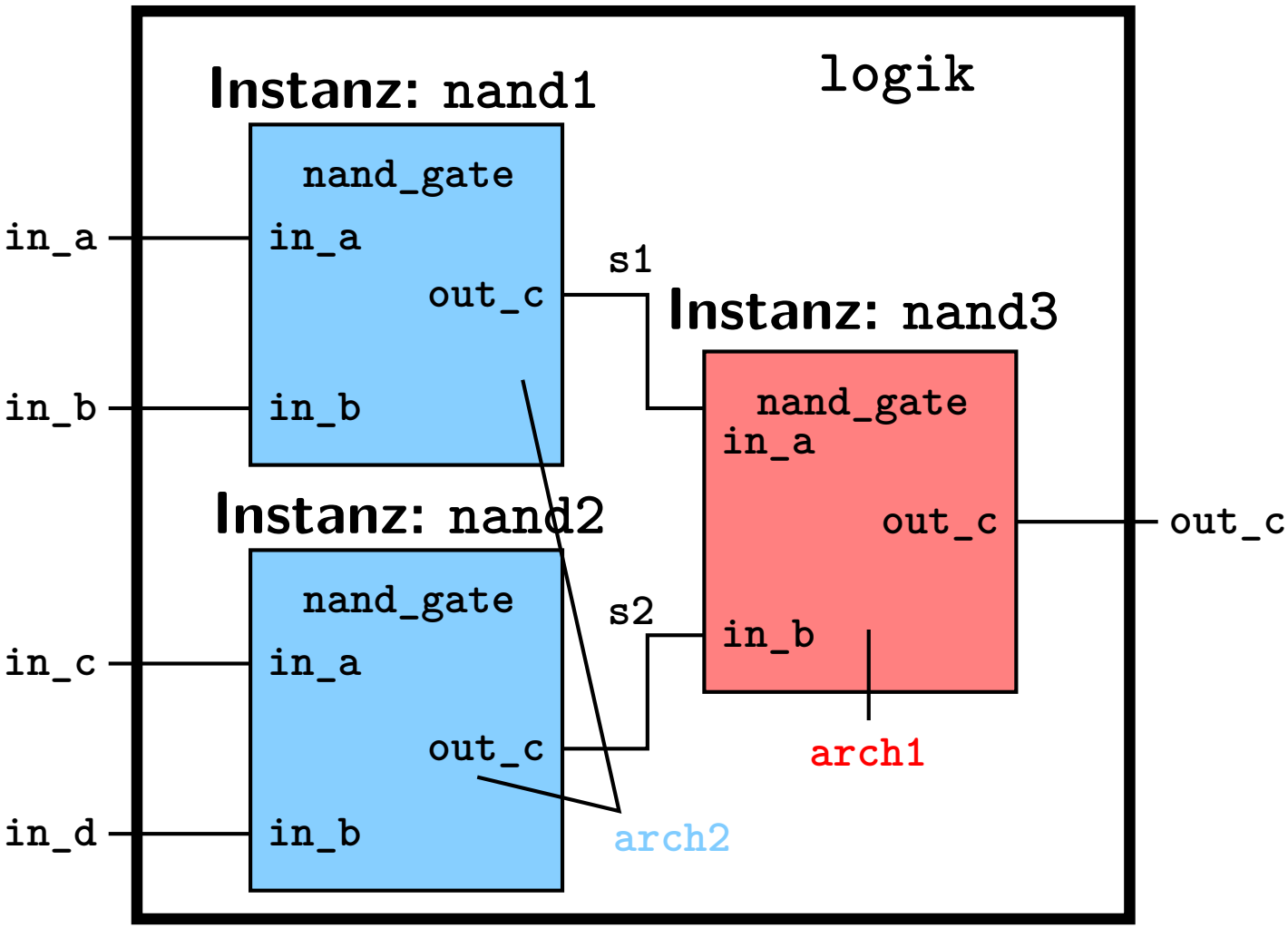
Ein weiteres Beispiel (Basisblockbeschreibung)

```
ENTITY nand_gate IS
  PORT (in_a, in_b : IN bit;
        out_c      : OUT bit);
END ENTITY nand_gate;

ARCHITECTURE arch1 OF nand_gate IS
BEGIN
  out_c <= in_a NAND in_b;
END ARCHITECTURE arch1;

ARCHITECTURE arch2 OF nand_gate IS
BEGIN
  out_c <= '0' WHEN in_a & in_b = "11" ELSE '1';
END ARCHITECTURE arch2;
```

Strukturales Modell mit nand_gate (letzte Seite)

```
ENTITY logik IS
  PORT (in_a, in_b, in_c, in_d : IN bit;
        out_c                  : OUT bit);
END ENTITY logik;
ARCHITECTURE struktur OF logik IS
  COMPONENT nand_gate
    PORT(in_a, in_b : IN bit;
         out_c      : OUT bit);
  END COMPONENT;
  SIGNAL s1, s2 : bit;
  FOR nand3 : nand_gate USE ENTITY work.nand_gate(arch1);
  FOR OTHERS : nand_gate USE ENTITY work.nand_gate(arch2);
BEGIN
  nand1  : nand_gate PORT MAP(in_a => in_a, in_b => in_b,
                                            out_c => s1);
  nand2  : nand_gate PORT MAP(in_a => in_c, in_b => in_d,
                                            out_c => s2);
  nand3  : nand_gate PORT MAP(in_a => s1, in_b => s2,
END ARCHITECTURE struktur;                  out_c => out_c);
```

Graphische Darstellung des letzten Modells



Besonderheiten des letzten Modells

- eine Komponente mit drei Instanzen
  - zwei identische Instanzen mit `arch2`
  - eine unterschiedliche Instanz mit `arch1`

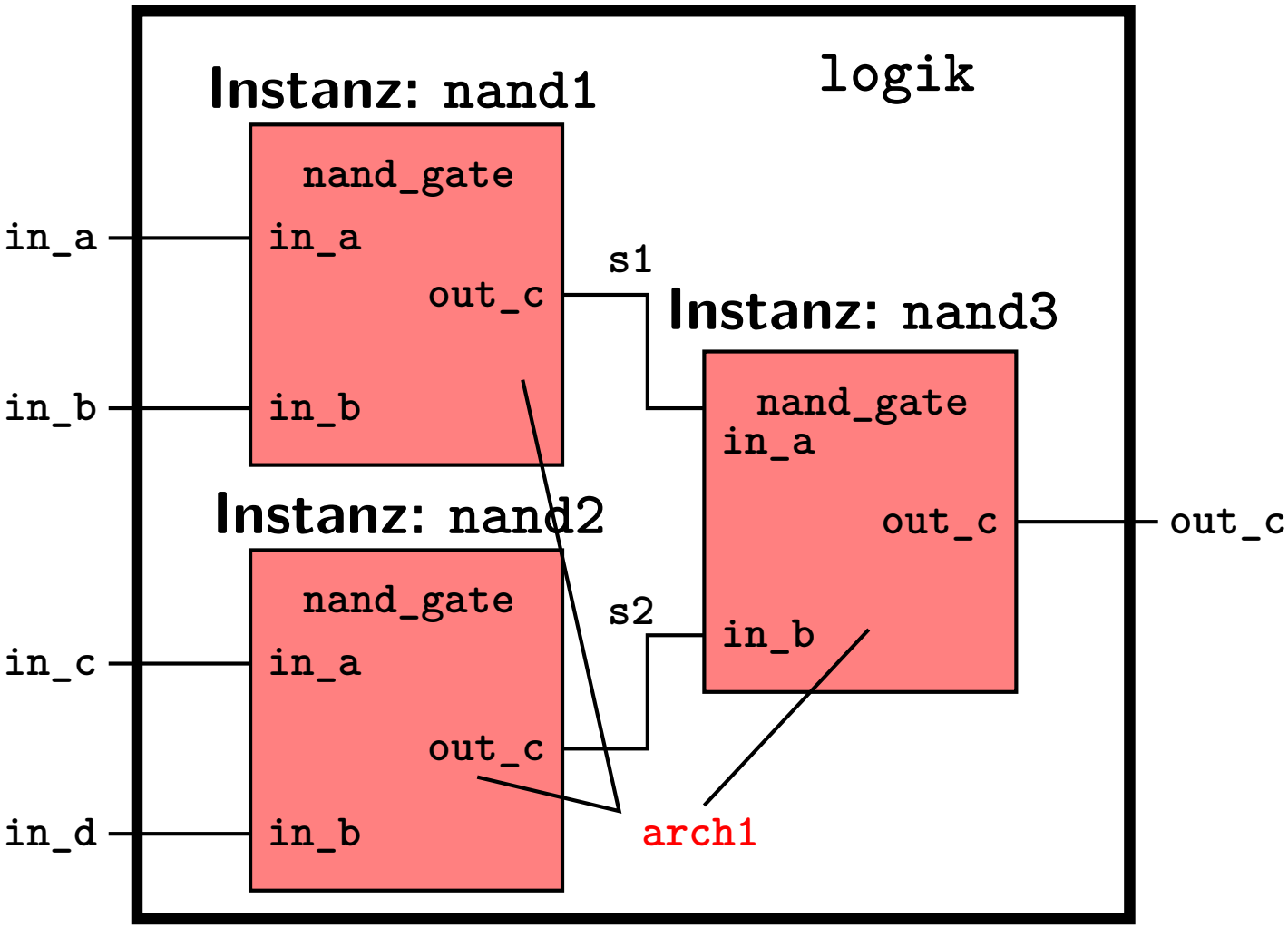
  (ENTITY ist immer gleich!)
- das Schlüsselwort `OTHERS` gibt alle Instanzen an, die noch nicht explizit konfiguriert wurden
- alle drei Instanzen können auch mit `ALL` in einem Schritt konfiguriert werden, z. B.

```
FOR ALL : nand_gate
  USE ENTITY work.nand_gate(arch1);
```

bzw.

```
FOR ALL : nand_gate
  USE ENTITY work.nand_gate(arch2);
```

Beschreibung mit 3 identischen Instanzen




```
FOR ALL : nand_gate USE ENTITY work.nand_gate(arch1);
```

Instantiierung gleicher Komponenten mit GENERATE

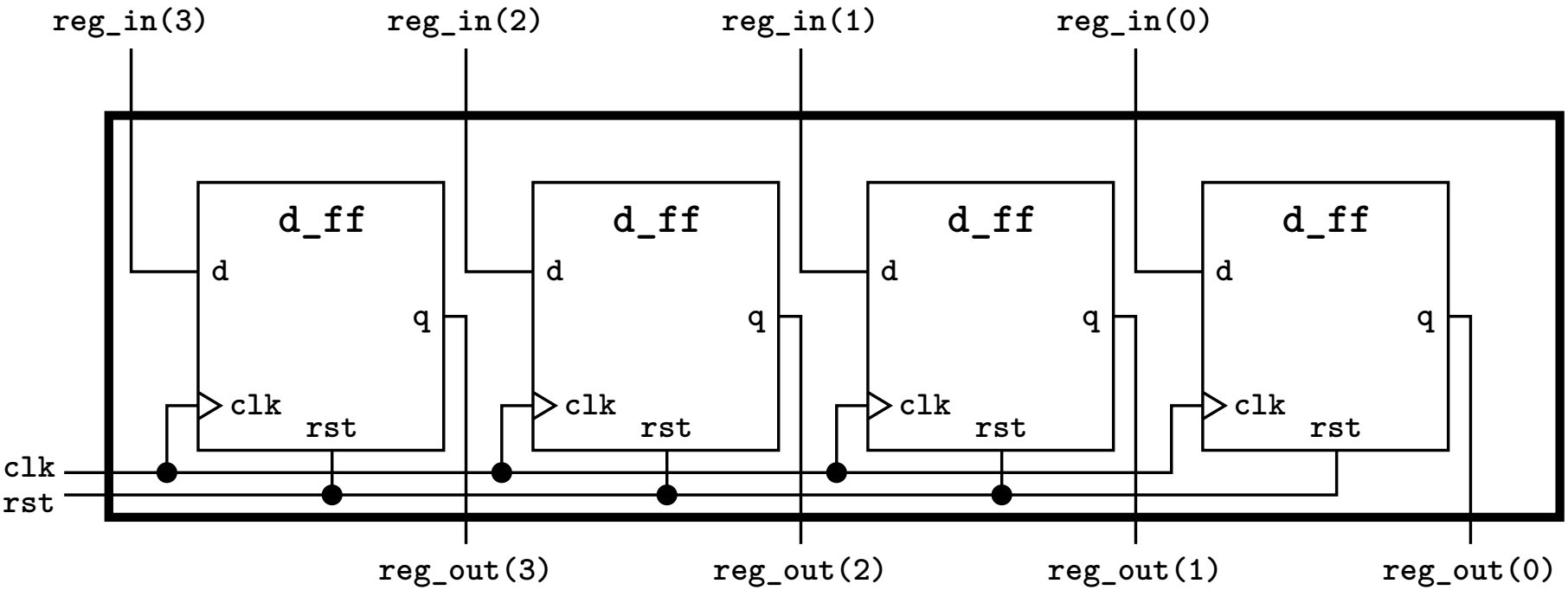
```
ENTITY n_bit_reg IS
  GENERIC (n : positive := 4);
  PORT (clk, rst : IN bit;
        reg_in  : IN bit_vector(n-1 DOWNTO 0);
        reg_out : OUT bit_vector(n-1 DOWNTO 0));
END ENTITY n_bit_reg;
ARCHITECTURE structure OF n_bit_reg IS
  COMPONENT d_ff
    PORT (d, clk, rst : IN bit; q : OUT bit);
  END COMPONENT;
BEGIN
  reg_gen : FOR i IN n-1 DOWNTO 0 GENERATE
    d_ff_instance : d_ff
      PORT MAP (reg_in(i), clk, rst, reg_out(i));
  END GENERATE reg_gen;
END ARCHITECTURE structure;
```

Einsatz von GENERATE (in Kombination mit FOR)

```
generate_name : FOR var_name IN discrete_range GENERATE
  parallel assignments
END GENERATE generate_name;
```

- Die nach der GENERATE-Anweisung stehenden Anweisungen werden entsprechend der discrete_range-Angabe wiederholt
- Wird nicht nur zur Komponenteninstantiierung eingesetzt (z. B. auch mehrfaches Verwenden gleicher nebenläufiger Anweisungen)
- Die Laufvariable der FOR-Schleife stellt einen der wenigen Fälle dar, in dem ein Objekt nicht explizit deklariert wird
- Im Beispiel besonders elegant ist die Nutzung von „GENERIC n“ zur Festlegung der Bitbreite

Graphische Darstellung des resultierenden Modells



Bei der Instantiierung von `n_bit_reg` in einem übergeordneten Modell (z. B. Testumgebung) kann die Bitbreite mit `GENERIC MAP (n => bitbreite);` überschrieben werden. Beim Weglassen von `GENERIC MAP` bzw. bei `GENERIC MAP (n => 4);` entsteht die oben dargestellte Struktur.

Flipflop als Basiselement

Die Komponente d_ff muss natürlich ebenfalls definiert werden, z. B.

```
ENTITY d_ff IS
    PORT (d, clk, rst : IN bit;
          q : OUT bit);
END ENTITY d_ff;
ARCHITECTURE behav OF d_ff IS
BEGIN
  change_state : PROCESS (clk, rst) IS
  BEGIN
    IF rst='1' THEN
      Q <= '0' AFTER 3 ns;
    ELSIF clk'event AND clk = '1' THEN
      Q <= d AFTER 3 ns;
    END IF;
  END PROCESS change_state;
END ARCHITECTURE behav;
```

Zusammenfassung

- Einteilung der Modellierungstechniken
- Strukturale Modellierung

Aufgaben zum Selbststudium

1. Ändern sie das Modell des Auffangregisters (Seite 161) so ab, dass ein Schieberegister entsteht.
2. Erzeugen Sie ein Strukturmodell mit zwei Auffangregistern einstellbarer Bitbreite, die wahlweise (durch eine Eingangssteuerleitung kontrolliert) zwei Eingänge gleichen Bitbreite direkt oder „über Kreuz“ aufnehmen, z. B.

```
cntrl = 0: reg_out1 <= in_1, reg_out2 <= in_2;
cntrl = 1: reg_out1 <= in_2, reg_out2 <= in_1;
```

Aufgabenlösungen zur 5. Vorlesung

Strukturale Modellierung

Der entsprechende VHDL-Quellkode ist auf dem Handout-Server unter

`VHDL/Aufgabenloesungen/VHDL/vhdl_v5_shift_reg.vhd`

und

`VHDL/Aufgabenloesungen/VHDL/vhdl_v5_switch_reg.vhd`

zu finden.

Systementwurf mit VHDL: Vorlesungs-Gesamtübersicht

Systementwurf mit VHDL: Gliederung der 6. Vorlesung

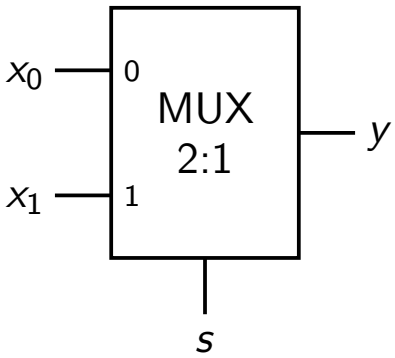
Wesentliche Merkmale

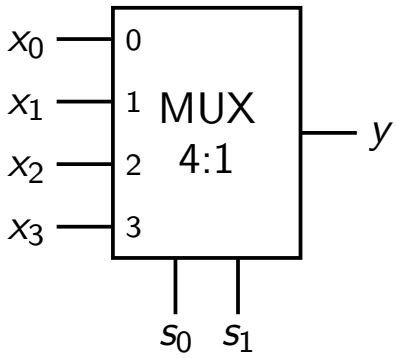
- Das Modell beinhaltet nur nebenläufige Anweisungen, keine Komponenten oder Prozesse
- Der ARCHITECTURE-Teil besteht aus 2 Hauptabschnitten:
  - Definitionen: Deklaration von Signalen, Konstanten, Typen
  - Funktionen: Zuweisung von Werten an Signale (unter Nutzung von Operatoren und bedingten Zuweisungsmechanismen)
- Beschreibung stellt die Funktion der Hardware als eine Menge BOOLEscher und algebraischer Gleichungen dar
- Meistens nicht sehr umfangreich
- Vollsynthetisierbar (vorausgesetzt, alle benutzten Operatoren und Typen sind synthesefähig)

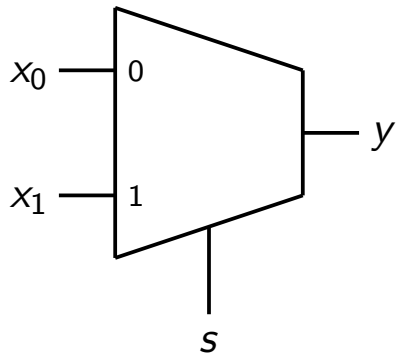
Beispiel eines funktionalen Modells

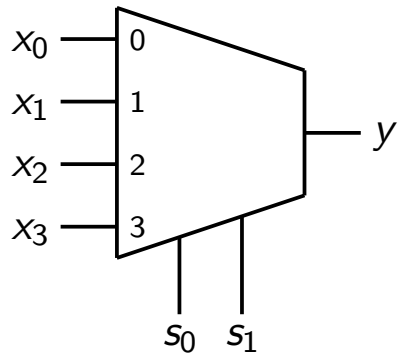
```
ENTITY mux4_to_1 IS
  PORT (in_1, in_2, in_3, in_4, s_1, s_2 : IN bit;
        out_c       : OUT bit);
END ENTITY mux4_to_1;

ARCHITECTURE funkt OF mux4_to_1 IS
SIGNAL sn_1, sn_2 : bit;
BEGIN
  sn_1 <= NOT s_1;
  sn_2 <= NOT s_2;
  out_c <= (in_1 AND sn_2 AND sn_1) OR
           (in_2 AND sn_2 AND  s_1) OR
           (in_3 AND s_2  AND sn_1) OR
           (in_4 AND s_2  AND  s_1);
END ARCHITECTURE funkt;
```

2:1-Multiplexer und 4:1-Multiplexer









s	y
0	x_0
1	x_1

s_1	s_0	y
0	0	x_0
0	1	x_1
1	0	x_2
1	1	x_3

Abhängig von der Belegung der Adressvariablen s_x wird einer der Eingänge auf den Ausgang durchgeschaltet.

Gleiche Schaltung als Strukturmodell

Beschreibung des Basisblocks 2:1-MUX und der ENTITY von 4:1-MUX:

```
ENTITY mux2_to_1 IS
  PORT (in_1, in_2, s : IN bit;
        out_c         : OUT bit);
END ENTITY mux2_to_1;

ARCHITECTURE funkt OF mux2_to_1 IS
BEGIN
  out_c <= (in_1 AND NOT s) OR (in_2 AND s);
END ARCHITECTURE funkt;

ENTITY mux4_to_1 IS
  PORT (in_1, in_2, in_3, in_4, s_1, s_2 : IN bit;
        out_c      : OUT bit);
END ENTITY mux4_to_1;
```

Strukturales Modell eines 4:1-MUX

```
ARCHITECTURE struktur OF mux4_to_1 IS
  COMPONENT mux2_to_1
    PORT(in_1, in_2, s : IN bit;
         out_c         : OUT bit);
  END COMPONENT;
  SIGNAL s1, s2 : bit;
FOR ALL : mux2_to_1 USE ENTITY work.mux2_to_1(funkt);
BEGIN
  mux1  : mux2_to_1 PORT MAP(in_1 => in_1,
            in_2 => in_2, s => s_1, out_c => s1);
  mux2  : mux2_to_1 PORT MAP(in_1 => in_3,
            in_2 => in_4, s => s_1, out_c => s2);
  mux3  : mux2_to_1 PORT MAP(in_1 => s1,
            in_2 => s2, s => s_2, out_c => out_c);
END ARCHITECTURE struktur;
```

Bedingte Signalzuweisungen

Conditional assignment: Das Signal bekommt einen Wert zugewiesen, der von der Erfüllung einer (oder mehrerer) Bedingung(en) *(condition)* abhängt (entspricht einem IF-ELSIF-ELSE-Konstrukt)

```
signal_name <= {value_1 WHEN condition_1 ELSE}
               value_2;
```

Selected assignment: Das Signal bekommt einen Wert zugewiesen, der von der Auswertung eines Ausdrucks *(expression)* abhängt (entspricht einem CASE-Konstrukt)

```
WITH expression SELECT
signal_name <= {value_1 WHEN choice_1,}
               value_n WHEN choice_n;
```

Multiplexer-Beispiel mit *conditional assignment*

```
ENTITY mux4_to_1 IS
  PORT (in_1, in_2, in_3, in_4, s_1, s_2 : IN bit;
        out_c       : OUT bit);
END ENTITY mux4_to_1;

ARCHITECTURE funkt OF mux4_to_1 IS
BEGIN
  out_c <= in_1 WHEN (s_2 = '0' AND s_1 = '0') ELSE
           in_2 WHEN (s_2 = '0' AND s_1 = '1') ELSE
           in_3 WHEN (s_2 = '1' AND s_1 = '0') ELSE
           in_4 WHEN (s_2 = '1' AND s_1 = '1')
                ELSE '0';
END ARCHITECTURE funkt;
```

Multiplexer-Beispiel mit *selected assignment*

```
ENTITY mux4_to_1 IS
  PORT (in_1, in_2, in_3, in_4, s_1, s_2 : IN bit;
        out_c       : OUT bit);
END ENTITY mux4_to_1;

ARCHITECTURE funkt OF mux4_to_1 IS
SIGNAL s : bit_vector(1 DOWNTO 0);
BEGIN
  s <= s_2 & s_1;
  WITH s SELECT
    out_c <= in_1 WHEN "00",
             in_2 WHEN "01",
             in_3 WHEN "10",
             in_4 WHEN "11";
END ARCHITECTURE funkt;
```

Anmerkungen zu bedingter Signalzuweisung

- Werden bei *selected assignment* nicht alle möglichen Werte des Ausdrucks explizit angegeben, so muss der fehlende Wertebereich mit `OTHERS` abgedeckt werden:

```
WITH s SELECT
  out_c <= in_1 WHEN "00",
           in_2 WHEN "01",
           in_3 WHEN OTHERS;
```

- Verändert sich der Wert eines Signals beim Eintreten einer Bedingung nicht, so kann das mit dem Schlüsselwort `UNAFFECTED` explizit gemacht werden:

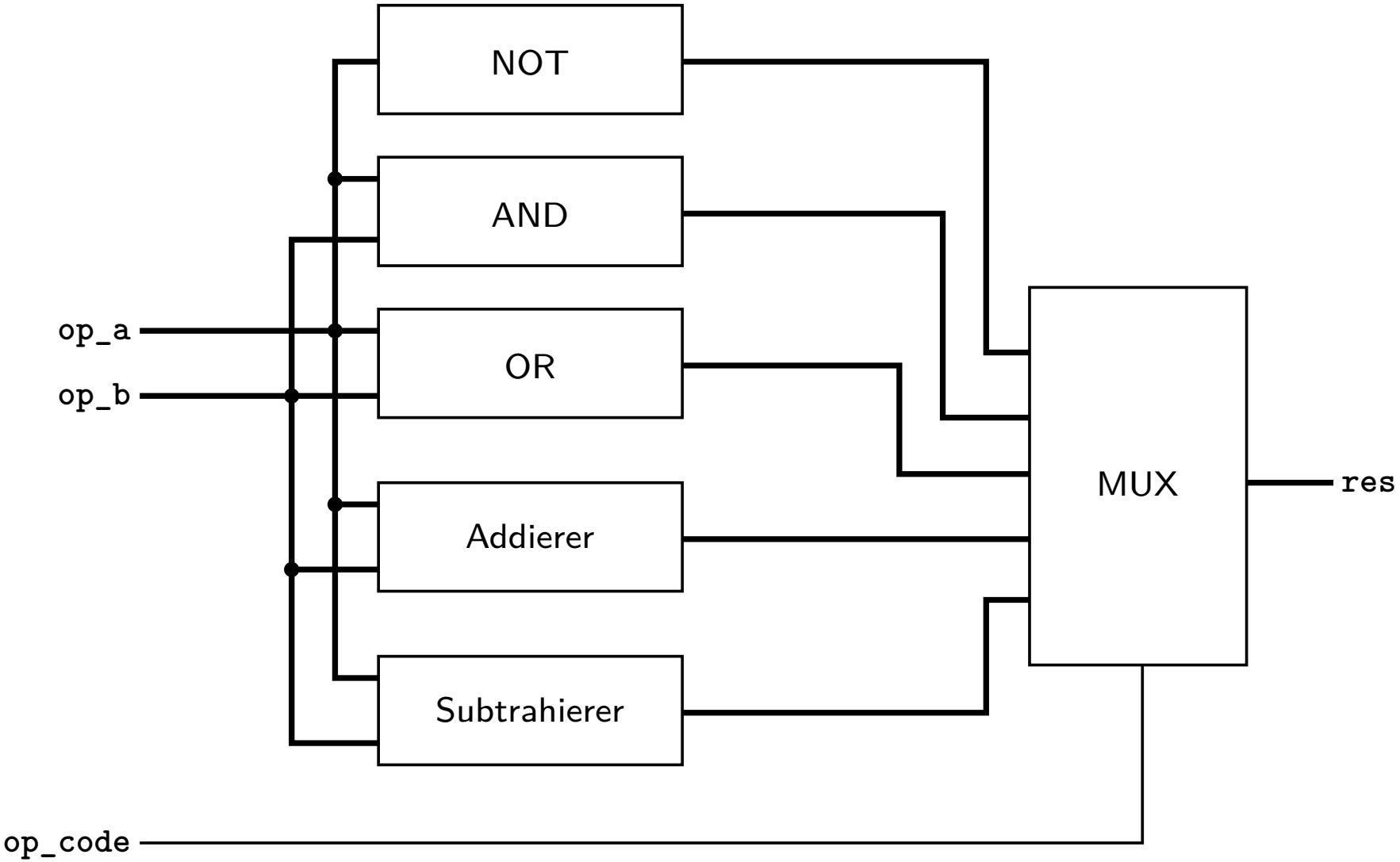
```
  out_c <= in_1 WHEN "00",
           UNAFFECTED WHEN OTHERS;
```

Dann erfolgt auch keine Transaktion auf dem Signal!

Komplexeres Beispiel (eine 8-Bit-ALU)

```
LIBRARY IEEE;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
ENTITY alu IS
  PORT (op_a, op_b : IN std_logic_vector(7 DOWNTO 0);
        op_code    : IN std_logic_vector(2 DOWNTO 0);
        res        : OUT std_logic_vector(7 DOWNTO 0));
END ENTITY alu;
ARCHITECTURE funkt OF alu IS
BEGIN
 WITH op_code SELECT
 res <= (NOT op_a) WHEN "000", (op_a OR op_b) WHEN "010",
  (op_a AND op_b) WHEN "001",
  std_logic_vector(signed(op_a)+signed(op_b)) WHEN "011",
  std_logic_vector(signed(op_a)-signed(op_b)) WHEN "100",
  x"00" WHEN OTHERS;
END ARCHITECTURE funkt;
```

ALU als Blockschaltbild



Wesentliche Merkmale

- Das Modell beschreibt das Verhalten der Schaltung bzw. des Systems mit Hilfe von parallelen Prozessen
- Der ARCHITECTURE-Teil besteht aus mehreren Hauptabschnitten:
  - Definitionen: Deklaration von Signalen, Konstanten, Typen
  - Verhalten: Prozesse
  - Funktionen (optional): Wie bei funktionaler Modellierung
- Beschreibung stellt die Funktion der Hardware mit Hilfe von kommunizierenden nebenläufigen Prozessen dar
- Nicht immer synthetisierbar, oft nur als Referenzmodell

Beispiel eines einfachen Verhaltensmodells

```
ENTITY mux4_to_1 IS
  PORT (in_1, in_2, in_3, in_4 : IN bit;
        s : IN bit_vector(1 DOWNTO 0);
        out_c       : OUT bit);
END ENTITY mux4_to_1;
ARCHITECTURE behaviour OF mux4_to_1 IS
BEGIN
  multiplex : PROCESS(in_1, in_2, in_3, in_4, s)
  BEGIN
    CASE s IS
      WHEN "00" => out_c <= in_1;
      WHEN "01" => out_c <= in_2;
      WHEN "10" => out_c <= in_3;
      WHEN "11" => out_c <= in_4;
    END CASE;
  END PROCESS multiplex;
END ARCHITECTURE behaviour;
```

Prozesse

Sequentielle (nacheinander ablaufende) Anweisungen werden in **Prozessen** zusammengefasst. Alle Prozesse in einem Modell sind nebenläufig, alle Anweisungen innerhalb eines Prozesses sind sequentiell. Prozess aus konzeptueller Sicht:

- bildet einen Hardware-Block nach
- ist ständig aktiv (eine Endlosschleife), wird jedoch durch bestimmte Anweisungen in der Regel angehalten (Warten auf ein Ereignis, z. B. Flanke eines Signals)
- beim Erreichen der letzten Anweisung beginnt die Abarbeitung wieder von vorn
- durch das Konzept der ereignisorientierten Simulation erscheinen die Prozesse auch auf einer Einprozessormaschine als parallel ablaufend

Sequentielle vs. nebenläufige Anweisungen

Nebenläufig: Signalzuweisungen außerhalb von Prozessen (auch bedingte Signalzuweisungen, siehe funktionale Modellierung), Prozesse innerhalb eines Modells

Sequentiell: Signal- und Variablenzuweisungen innerhalb eines Prozesses, WAIT, Verzweigungen, CASE, Schleifen

Häufige Fehlerquelle: Benutzung einer sequentiellen Anweisung außerhalb eines Prozesses (wird vom VHDL-Compiler bemängelt).

Mechanismen der Prozesssynchronisation:

- WAIT-Anweisung
- *Sensitivity list* (Empfindlichkeitsliste)

Wichtig ⚠

In einem einzelnen Prozess darf immer nur ein Synchronisationsmechanismus angewandt werden, d. h. WAIT-Anweisung und Empfindlichkeitsliste gleichzeitig sind nicht zulässig.

Syntax von Prozessen mit WAIT-Anweisungen

```
[process_name :] PROCESS
 [Declarations, Definition, USE statements]
BEGIN
  [sequential statements]
  WAIT ... ;
  {sequential statements
  WAIT ...;}
END PROCESS [process_name];
```

Es muss mindestens eine WAIT-Anweisung vorhanden sein. Syntax von WAIT:

```
WAIT [ON signal_name_1 {, signal_name_n}]
     [UNTIL condition]
     [FOR time_expression];
```

WAIT; ohne Zusatzangaben blockiert einen Prozess für immer!

Semantik von WAIT-Argumenten

ON signal_name_1, ...: Prozess wartet, bis mindestens eines der (nach `ON` aufgeführten) Signale sich verändert

UNITL condition: Prozess wartet, bis die Bedingung erfüllt ist

FOR time_expression: Prozess wartet maximal für die durch `time_expression` angegebene Zeit (wenn kein weiteres Ereignis schon vorher eintrifft)

Wartebedingungen können in einer WAIT-Anweisung kombiniert werden (alle drei oder auch paarweise):

```
WAIT FOR 12 ns;
WAIT ON clk UNTIL clk = '1';
WAIT UNTIL clk'event AND clk = '1';
```

Wichtig ⚠

An einer WAIT-Anweisung wird ein Prozess grundsätzlich angehalten. Erst beim nächsten relevanten Ereignis wird geprüft, ob der Prozess fortfahren kann.

Syntax von Prozessen mit Sensitivity-Liste

```
[process_name :] PROCESS (signal_1 {, signal_n})
 [Declarations, Definition, USE statements]
BEGIN
  [sequential statements, no WAIT!]
END PROCESS [process_name];
```

Der Prozess wird immer dann aktiviert (und einmal komplett durchlaufen), wenn ein Ereignis an mindestens einem der Signale aus der Sensitivity-Liste auftritt.

Kann verhaltensäquivalent durch einen Prozess mit

```
WAIT ON signal_1 {, signal_n};
```

als letzte Anweisung dargestellt werden.

Prozess als syntaktischer Rahmen

Innerhalb von Prozessen dürfen Anweisungen auftreten, die sonst an keiner anderen Stelle des Modells erlaubt sind (sequentielle Anweisungen):

- Variablenzuweisungen
- `WAIT`
- Verzweigungen: `IF-THEN-ELSE`
- Fallunterscheidung: `CASE`
- Schleifen: `LOOP`
- `EXIT` und `NEXT`
- `NULL`

Signalzuweisungen können auch außerhalb von Prozessen erfolgen (können also sowohl sequentiell als auch nebenläufig sein).

Aufgaben zum Selbststudium

1. Erweitern Sie die ALU (Seite 180) um Multiplikation, Verschiebung und Rotation. Achten sie dabei auf die korrekte Typisierung der Operanden (Konvertierung ist notwendig).
2. Beschreiben Sie die ALU als Strukturmodell.

Zusammenfassung

- Funktionale Modellierung
- Verhaltensmodellierung

Aufgabenlösungen zur 6. Vorlesung

Funktionale und strukturale Modellierung

Der entsprechende VHDL-Quellkode ist auf dem Handout-Server unter

`VHDL/Aufgabenloesungen/VHDL/vhdl_v6_alu_mult.vhd`

und

`VHDL/Aufgabenloesungen/VHDL/vhdl_v6_alu_struct.vhd`

zu finden.

Systementwurf mit VHDL: Vorlesungs-Gesamtübersicht

Systementwurf mit VHDL: Gliederung der 7. Vorlesung

Variablenzuweisungen

- Zuweisung des Wertes an eine im Prozess definierte Variable, z. B. `a := "0000"`:

Signalzuweisung	Variablenzuweisung
<=	:=

- Seit VHDL'93 können auch außerhalb von Prozessen sogenannte `SHARED` Variables definiert werden, die in allen Prozessen derselben ARCHITECTURE sichtbar und veränderbar sind. Deren Benutzung birgt jedoch eine Reihe unerwünschter Effekte und sollte daher (insbesondere bei synthesefähigen Modellen) vermieden werden.
- Im Gegensatz zum Signal verändert sich der Wert einer Variablen sofort nach der Abarbeitung der entsprechenden Anweisung (dazu später mehr)

Verzweigungen mit IF-THEN-ELSE

```
IF condition_1 THEN sequential statements
{ELSIF condition_n THEN sequential statements}
[ELSE sequential statements] END IF;
```

Zum Beispiel:

```
ARCHITECTURE behaviour OF mux4_to_1 IS
BEGIN
  multiplex : PROCESS(in_1, in_2, in_3, in_4, s)
  BEGIN
    IF (s= "00") THEN out_c <= in_1;
      ELSIF (s = "01") THEN out_c <= in_2;
      ELSIF (s = "10") THEN out_c <= in_3;
      ELSE out_c <= in_4;
    END IF;
  END PROCESS multiplex;
END ARCHITECTURE behaviour;
```

Auswahl mit CASE

```
CASE expression IS
  {WHEN value_n =>
     sequential statements}
  [WHEN OTHERS =>
     sequential statements]
END CASE;
```

Wenn der Wertebereich von `expression` durch `WHEN`-Angaben nicht komplett ausgeschöpft ist, muss der fehlende Teil mit `OTHERS` abgedeckt werden (bei `std_logic` müssen z. B. auch die Werte `'U'`, `'X'` usw. abgedeckt werden)!

Für ein Beispiel der `CASE`-Benutzung siehe die erste Beschreibung des Multiplexers.

Schleifen mit LOOP

FOR-Schleife:

```
[loop_name:] FOR range LOOP
   sequential statements
END LOOP [loop_name];
```

WHILE-Schleife:

```
[loop_name:] WHILE condition LOOP
   sequential statements
END LOOP [loop_name];
```

Endlosschleife:

```
[loop_name:] LOOP
   sequential statements
END LOOP [loop_name];
```

Bedingung vs. Bereich bei Schleifen

Spezifikation der Bedingung `condition`: Jeder Ausdruck, der einen Wert vom Typ `boolean` liefert (alles, was auch bei einer `IF`-Anweisung stehen könnte). Ein Beispiel:

```
schleife : WHILE in_a="0000" LOOP
  out_b <= out_b - 1;
  out_c <= out_c + 1;
END LOOP schleife;
```

Spezifikation des Bereichs `range`: Eine Laufvariable, die abwärts (`DOWNTO`) oder aufwärts (`TO`) vom Startwert zum Endwert gezählt wird. Ein Beispiel:

```
schleife : FOR i IN 0 TO n-1 LOOP
  out_b <= out_b - 1;
  out_c <= out_c + 1;
END LOOP schleife;
```

Vorzeitiger Schleifenaustritt mit EXIT und NEXT

```
NEXT [loop_name] [WHEN condition];
EXIT [loop_name] [WHEN condition];
```

Beispiele:

```
schleife : FOR i IN 0 TO n-1 LOOP
  out_b <= out_b + 1;
  EXIT schleife WHEN out_b = 15;
END LOOP schleife;

schleife1 : FOR i IN 0 TO n-1 LOOP
  schleife2 : FOR j IN 0 TO m-1 LOOP
    out_b <= out_b + 1;
    NEXT schleife1 WHEN out_b = 15; -- Schleife1 !!!
  END LOOP schleife2;
END LOOP schleife1;
```

Keine Aktion mit NULL

NULL-Anweisung tut das, was der Name sagt: nichts! Einsatz erfolgt meistens als Platzhalter bei IF- und CASE-Anweisungen:

```
ARCHITECTURE behaviour OF mux2_to_1 IS
BEGIN
  multiplex : PROCESS(in_1, in_2, s)
  BEGIN
    CASE s IS
      WHEN "00" => out_c <= in_1;
      WHEN "01" => out_c <= in_2;
      WHEN OTHERS => NULL;
    END CASE;
  END PROCESS multiplex;
END ARCHITECTURE behaviour;
```

Ein komplexeres Beispiel (FSM)

```
ENTITY fsm IS
  PORT (res, clk, x : IN bit;
        y : OUT bit_vector(1 DOWNTO 0));
END ENTITY fsm;

ARCHITECTURE behaviour OF FSM IS
  TYPE state IS (st1, st2, st3, st4);
  SIGNAL st : state;
BEGIN
  output : PROCESS(st)
  BEGIN
    IF ((st = st1) OR (st = st3)) THEN y <= "01";
    ELSIF (st = st2) THEN y <= "11"; ELSE y <= "00";
  END IF;
END PROCESS output;
...
```

Ein komplexeres Beispiel (FSM, Fortsetzung)

```
state_tr : PROCESS(clk, res, x)
  BEGIN
    IF (res = '1') THEN st <= st1;
    ELSIF (clk'event AND clk ='1') THEN
      CASE st IS
        WHEN st1 => IF (x='0') THEN st <= st2;
                                ELSE st <= st4; END IF;
        WHEN st2 => IF (x='0') THEN st <= st3;
                                ELSE st <= st1; END IF;
        WHEN st3 => IF (x='0') THEN st <= st4;
                                ELSE st <= st2; END IF;
        WHEN st4 => IF (x='0') THEN st <= st1;
                                ELSE st <= st3; END IF;
      END CASE;
    END IF;
  END PROCESS state_tr;
END ARCHITECTURE behaviour;
```

Auswirkungen der Nebenläufigkeit

Häufige Fehlerquelle bei Prozessen ist das Vergessen von Nebenläufigkeit:

```
SIGNAL y : std_logic;
p1 : PROCESS                    p2: PROCESS
BEGIN                           BEGIN
  y <= '1' AFTER 40 ns;           y <= '0' AFTER 40 ns;
  WAIT;                           WAIT;
END PROCESS p1;                 END PROCESS p2;
```

- Beide Prozesse werden bei der Initialisierung einmal durchlaufen
- Nach außen müssen sie parallel wirken (Hardware!)

➥ Nach 40 ns entsteht ein Konflikt bei der Zuweisung eines Wertes an `y`, die Auflösungsfunktion ermittelt den Wert `'X'`! Bis zu diesem Zeitpunkt bleibt das Signal `y` uninitialisiert `'U'`, weil `AFTER` die Zuweisung verzögert.

Auswirkungen der Sequentialität

Eine weitere häufige Fehlerquelle bei Prozessen ist die Fehlinterpretation von Sequentialität:

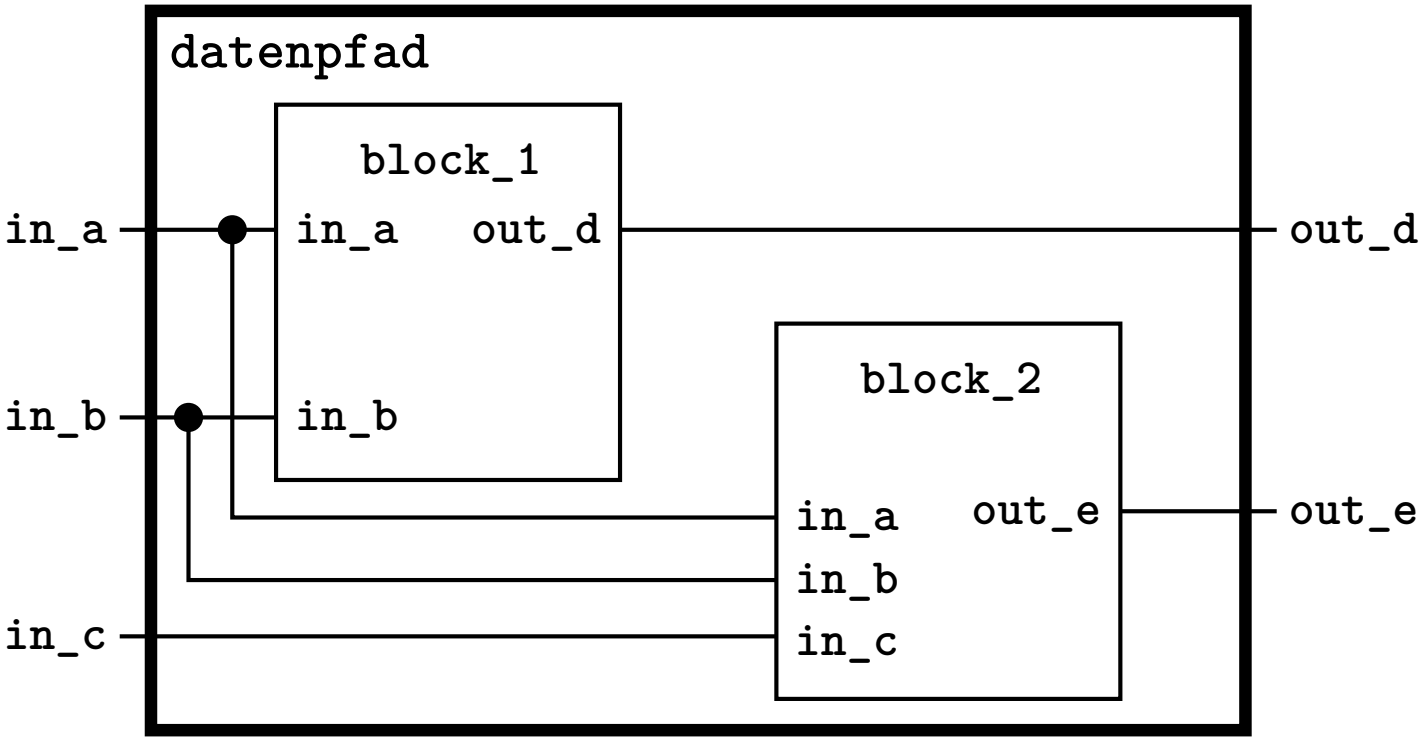
```
SIGNAL y : std_logic;
p1 : PROCESS
BEGIN
  y <= '1' AFTER 5 ns;
  y <= '0' AFTER 15 ns;
  y <= '1' AFTER 20 ns;
  WAIT FOR 100 ns;
END PROCESS p1;
```

- Die Anweisungen innerhalb des Prozesses werden sequentiell ausgeführt, das Signal wird jedoch ereignisorientiert getrieben, d. h. das zuletzt erzeugte Ereignis „überschreibt“ die älteren.

➥ Nach 20 ns wird `y` der Wert `'1'` zugewiesen, bis dahin bleibt das Signal `y` uninitialisiert (`'U'`)!

Ein weiteres Fehlerbeispiel

Skizze des zu modellierenden Systems:



Funktion der Blöcke:

```
out_d <= in_a + in_b;
out_e <= (in_a + in_b)*in_c;
```

Fehlerhafter Modellierungsansatz

```
ENTITY datenpfad IS
  PORT (in_a, in_b, in_c : IN integer;
        out_d            : INOUT integer;
        out_e            : OUT integer);
END ENTITY datenpfad;

ARCHITECTURE behaviour OF datenpfad IS
BEGIN
  rechne : PROCESS (in_a, in_b, in_c)
  BEGIN
    out_d <= in_a + in_b;
    out_e <= out_d*in_c;
  END PROCESS rechne;
END ARCHITECTURE behaviour;
```

Problemanalyse

Kleinere Probleme:

- Verwendung von `integer` an der externen Schnittstelle (besser `bit_vector`, noch besser `std_logic`)
- Deklaration von `out_d` als `INOUT` (bei einer externen Schnittstelle prinzipiell in Ordnung, bei der Verwendung als Block eines komplexeren Modells wird man in der Regel Probleme bei der Synthese bekommen ➠ Tri-State-Buffer sind nur für Außenanschlüsse vorgesehen)

Hauptproblem:

- `out_d` ist ein Signal, beim Durchlaufen des Prozesskörpers wird die Zuweisung eines Wertes an `out_d` vorgemerkt, aber nicht sofort durchgeführt (wie es bei einer Variablen der Fall wäre).

➥ Die Zuweisung an `out_e` „sieht“ den alten Wert von `out_d`, das Verhalten entspricht nicht der Spezifikation!

Eine mögliche Lösung: Variable einführen

```
ENTITY datenpfad IS
  PORT (in_a, in_b, in_c : IN integer;
        out_d, out_e     : OUT integer);
END ENTITY datenpfad;

ARCHITECTURE behaviour OF datenpfad IS
BEGIN
  rechne : PROCESS (in_a, in_b, in_c)
  VARIABLE temp_d : integer;
  BEGIN
    temp_d := in_a + in_b;
    out_d <= temp_d;
    out_e <= temp_d * in_c;
  END PROCESS rechne;
END ARCHITECTURE behaviour;
```

Ein weiterer Lösungsansatz: Zwei Prozesse

```
ENTITY datenpfad IS
  PORT (in_a, in_b, in_c : IN integer;
        out_d            : INOUT integer;
        out_e            : OUT integer);
END ENTITY datenpfad;
ARCHITECTURE behaviour OF datenpfad IS
BEGIN
  rechne_d : PROCESS (in_a, in_b)
  BEGIN
    out_d <= in_a + in_b;
  END PROCESS rechne_d;
  rechne_e : PROCESS (in_c, out_d)
  BEGIN
    out_e <= in_c * out_d;
  END PROCESS rechne_e;
END ARCHITECTURE behaviour;
```

Eine einfache Lösung mit einem Prozess

```
ENTITY datenpfad IS
  PORT (in_a, in_b, in_c : IN integer;
        out_d, out_e     : OUT integer);
END ENTITY datenpfad;

ARCHITECTURE behaviour OF datenpfad IS
BEGIN
  rechne : PROCESS (in_a, in_b, in_c)
  BEGIN
    out_d <= in_a + in_b;
    out_e <= (in_a + in_b) * in_c;
  END PROCESS rechne;
END ARCHITECTURE behaviour;
```

Das Konsistenzproblem entsteht nicht, weil immer die aktuellen Werte von `in_a`, `in_b` und `in_c` gelesen werden.

Vereinigung verschiedener Modellierungstechniken

Die meisten realen Modelle sind keiner eindeutigen Modellierungstechnik zuzuordnen, sondern kombinieren verschiedene Modellierungstechniken in verschiedenen Entwurfseinheiten (und oft sogar innerhalb einer und derselben Entwurfseinheit):

- Die `und_gatter` und `oder_gatter` beim Strukturmodell von `und_oder` waren als funktionale Modelle angegeben (strukturale und funktionale Modellierung in einem Modell)
- Im FSM-Beispiel kann der `output`-Prozess durch folgende funktionale Anweisung ersetzt werden:

```
y <= "01" WHEN (st = st1) OR (st = st3)
       ELSE "11" WHEN (st = st2) ELSE "00";
```

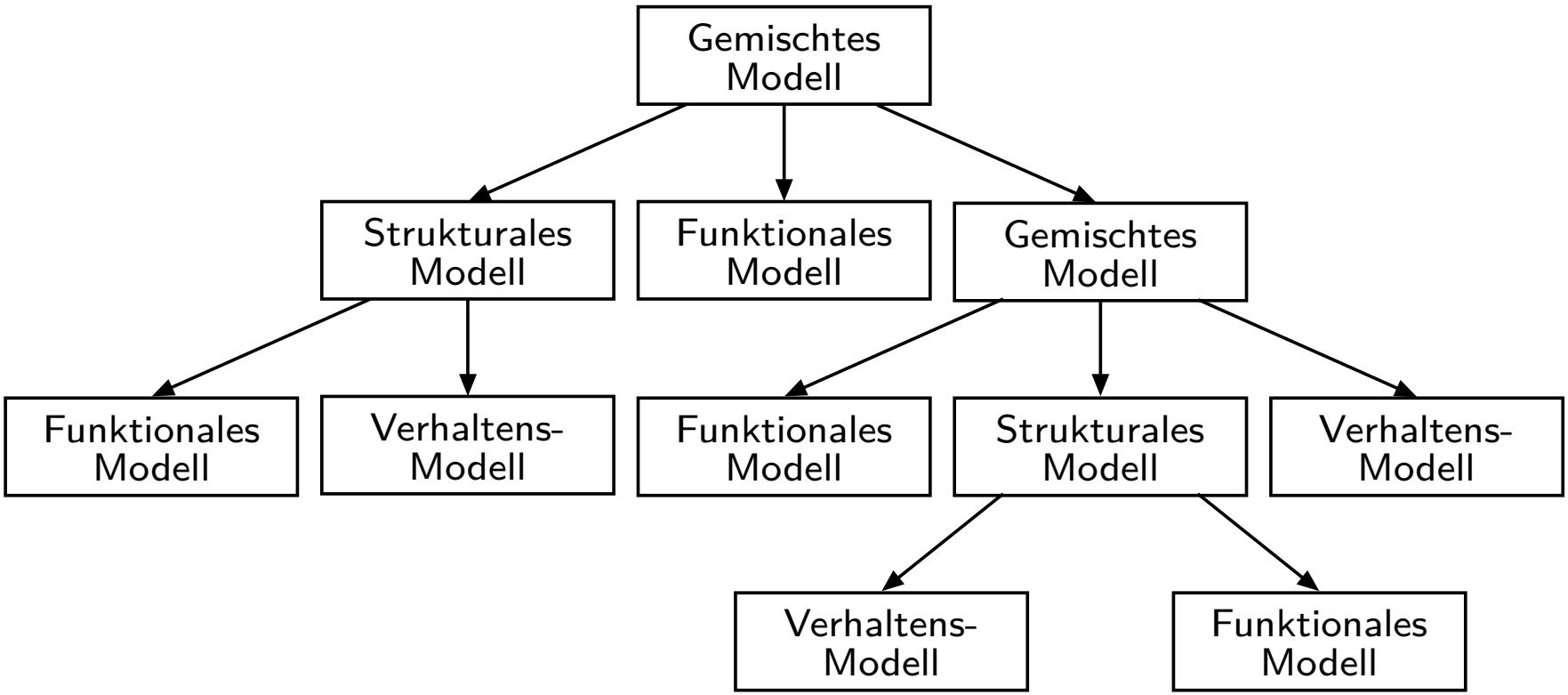
(funktionale Modellierung und Verhaltensmodellierung in einem Modell)

Das Konzept

Die Verwendung von verschiedenen Modellierungstechniken auf verschiedenen Hierarchieebenen in der Struktur des Modells bzw. das Vermischen verschiedener Modellierungstechniken in einer Entwurfseinheit wird als **hierarchische Modellierung** bzw. **gemischte Modellierung** bezeichnet (gängige Praxis beim Entwurf und der Modellierung mit VHDL):

- Passende Modellierungsart für unterschiedliche Hardware-Blöcke innerhalb eines komplexen Systems
- Optimale Nutzung der Vielfalt von Ausdrucksmitteln von VHDL
- Auf der Technologieebene (nach der Bearbeitung durch die Entwurfswerkzeuge) transparent

Beispiel einer globalen Struktur eines hierarchischen Modells



Die unterste Ebene (= „Blätter“) bilden immer funktionale und Verhaltensmodelle, denn eine Struktur impliziert das Vorhandensein weiterer (nicht struktureller) Modelle

Ein einfaches hierarchisches Modell

```
ENTITY und_gatter IS              -- wird spaeter als
  PORT (in_a, in_b : IN bit; -- Komponente verwendet
        out_c      : OUT bit);
END ENTITY und_gatter;
ARCHITECTURE arch OF und_gatter IS
BEGIN
  out_c <= in_a AND in_b;
END ARCHITECTURE arch;

-- Schnitstellenbeschreibung des Hauptmodells
ENTITY logik IS
  PORT (in_a, in_b, in_c : IN bit;
        out_d, out_e     : OUT bit);
END ENTITY logik;
```

Ein einfaches hierarchisches Modell (fortgesetzt)

```
ARCHITECTURE hierarch OF logik IS
  COMPONENT und_gatter
    PORT(in_a, in_b : IN bit;
         out_c      : OUT bit);
  END COMPONENT;
  SIGNAL s : bit;
  FOR und1 : und_gatter USE ENTITY work.und_gatter(arch);
BEGIN
  und1 : und_gatter PORT MAP(in_a => in_a, in_b => in_b,
                                          out_c => s);
  out_d <= s OR in_c;
  p1: PROCESS (in_b, in_c)
  BEGIN
    IF (in_b = '1') THEN out_e <= in_c;
    ELSE  out_e <= NOT in_c; END IF;
  END PROCESS p1;
END ARCHITECTURE hierarch;
```

Aufgaben zum Selbststudium

1. Geben Sie eine weitere Lösungsvariante für das auf der Seite 208 vorgestellte (fehlerhaft modellierte) Problem an.
2. Zeichnen Sie die globale Struktur des auf den Seiten 216–217 angegebenen hierarchischen Modells.

Zusammenfassung

- Abschließende Betrachtungen zur Verhaltensmodellierung
- Hierarchische Modellierung

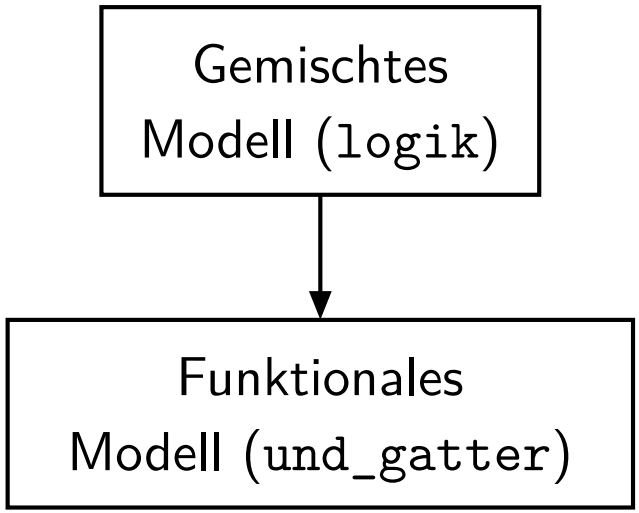
Aufgabenlösungen zur 7. Vorlesung

Verhaltensmodellierung

Hinzufügen von out_d zur Empfindlichkeitsliste:

```
ENTITY datenpfad IS
  PORT (in_a, in_b, in_c : IN integer;
        out_d            : INOUT integer;
        out_e            : OUT integer);
END ENTITY datenpfad;

ARCHITECTURE behaviour OF datenpfad IS
BEGIN
  rechne : PROCESS (in_a, in_b, in_c, out_d)
  BEGIN
    out_d <= in_a + in_b;
    out_e <= out_d*in_c;
  END PROCESS rechne;
END ARCHITECTURE behaviour;
```

Struktur eines hierarchischen Modells



Systementwurf mit VHDL: Vorlesungs-Gesamtübersicht

Systementwurf mit VHDL: Gliederung der 8. Vorlesung

4. Simulation
   - 4.1. Grundbegriffe
   - 4.2. Simulationsablauf
   - 4.3. Besonderheiten

4. Simulation

Simulationsmodell

Simulation ist einer der wichtigsten Aspekte bei der Modellierung von Hardware in VHDL. Das Verständnis der internen Simulationsabläufe ist für eine korrekte Modellierung unabdingbar:

- Zeitbegriff
- Interne Datenstrukturen und deren Verwaltung
- Simulationsablauf
- Objekte und Zuweisungsmodelle
- Häufige Fehlerquellen

Die Ausführung einer VHDL-Beschreibung und Interpretation einzelner Anweisungen sind unter dem Oberbegriff **Simulationsmodell** zusammengefasst. Dieses soll nun im Detail betrachtet werden.

Zeitbegriff

Reale Systemzeit (physikalische Zeit): Zeit im realen System, in der physikalische Prozesse ablaufen (z. B. Veränderungen der Signalpegel, Strom und Spannungsverläufe). Diese Zeit ist kontinuierlich.

Modellzeit: Zeit, die vom VHDL-Simulator nachgebildet wird, in der die in VHDL modellierten Prozesse (≙ Abbilder der physikalischen Prozesse) ablaufen. Diese Zeit ist diskret.

Simulationszeit: Zeit, die ein VHDL-Simulator braucht, um bestimmte Abläufe im Modell zu simulieren (Simulator-Laufzeit). Ist für den Entwerfer in absoluten Größen nicht vorhersagbar. In der Simulationszeit ist die Ordnungsrelation auf der Menge der ausgeführten VHDL-Anweisungen definiert: „später/früher ausgeführt“. Diese Zeit ist kontinuierlich.

Grundüberlegungen

- Nachbildung der echten Hardwareparallelität auf beliebigen (auch Einprozessor-) Maschinen
- Simulationszeit $\neq$ simulierte Zeit (Modellzeit)
- Einheitliche Darstellung der Anweisungen für den Simulator
- Einheitliches Simulationskonzept

➥ Exakt gleiche Simulationsergebnisse für exakt gleiche VHDL-Modelle unabhängig von den internen Implementierungsdetails des Simulators und dem Rechnersystem, auf dem der Simulator läuft

Das Simulationskonzept ist ein Bestandteil des VHDL-Standards.

Einheitliche Darstellung der Anweisungen

- Für den Simulator sehen alle nebenläufigen Anweisungen wie Prozesse aus:
  - alle Prozesse innerhalb eines Verhaltensmodells sind nebenläufig
  - alle strukturalen und hierarchischen Modelle können auf nebenläufige Anweisungen/Anweisungsgruppen zurückgeführt werden
  - alle nebenläufigen Anweisungen/Anweisungsgruppen können mit Prozessen verhaltensäquivalent nachgebildet werden:

```
a <= b AND c;
```

ist verhaltensäquivalent zu

```
assign_1 : PROCESS(b,c)
BEGIN
  a <= b AND c;
END assign_1;
```

➥ Es ist ausreichend, ein schlüssiges Konzept für die Simulation von nebenläufigen Prozessen anzugeben

Konzept der ereignisorientierten Simulation

Ereignis und Transaktion

Ein Ereignis ist eine Werteveränderung eines Signals. Eine Transaktion ist eine Wertezuweisung an ein Signal, die nicht zwangsläufig zu einer Werteveränderung führt.

- Der Zustand des Modells wird nur zu den Modellzeitpunkten evaluiert, an denen Transaktionen eingetragen sind
- Die Modellzeit schreitet nur dann voran, wenn im aktuellen Modellzeitpunkt ein stabiler Zustand eingetreten ist (es treten keine Transaktionen und Ereignisse mehr auf, d. h. es kann kein weiterer Prozess aktiviert werden)

➥ Das Verhalten des Modells wird durch die Zustandsübergänge zwischen den stabilen Zuständen komplett beschrieben, die Modellzeit enthält nur Zeitpunkte, an denen Ereignisse eintreten

Simulationsphasen

Interne Implementierung des Simulators kann vom Konzept der ereignisorientierten Simulation abweichen, nach außen wird jedoch stets die gleiche Sicht erzeugt, die dem ereignisorientierten Konzept entspricht. Die Evaluation des Modellzustandes erfolgt immer nach dem gleichen Schema in zwei Phasen (Δ-Zyklus):

1. Prozessausführungsphase *(process evaluation)*: Alle aktiven Prozesse bis zur END-Anweisung oder der nächsten WAIT-Anweisung abarbeiten. Die Variablenzuweisungen gelten sofort, die Signalzuweisungen werden nur vorgemerkt (sie sind noch unwirksam!)
2. Signalzuweisungsphase *(signal update)*: Alle Signalzuweisungen durchführen. Wenn dadurch weitere Prozesse aktiviert werden können, wird die Modellzeit um Δ erhöht und der ganze Zyklus von vorn durchlaufen (beginnend mit einer neuen Prozessausführungsphase). Sonst wird die Modellzeit auf den Zeitpunkt des nächsten Ereignisses gesetzt.

Das Δ-Konzept

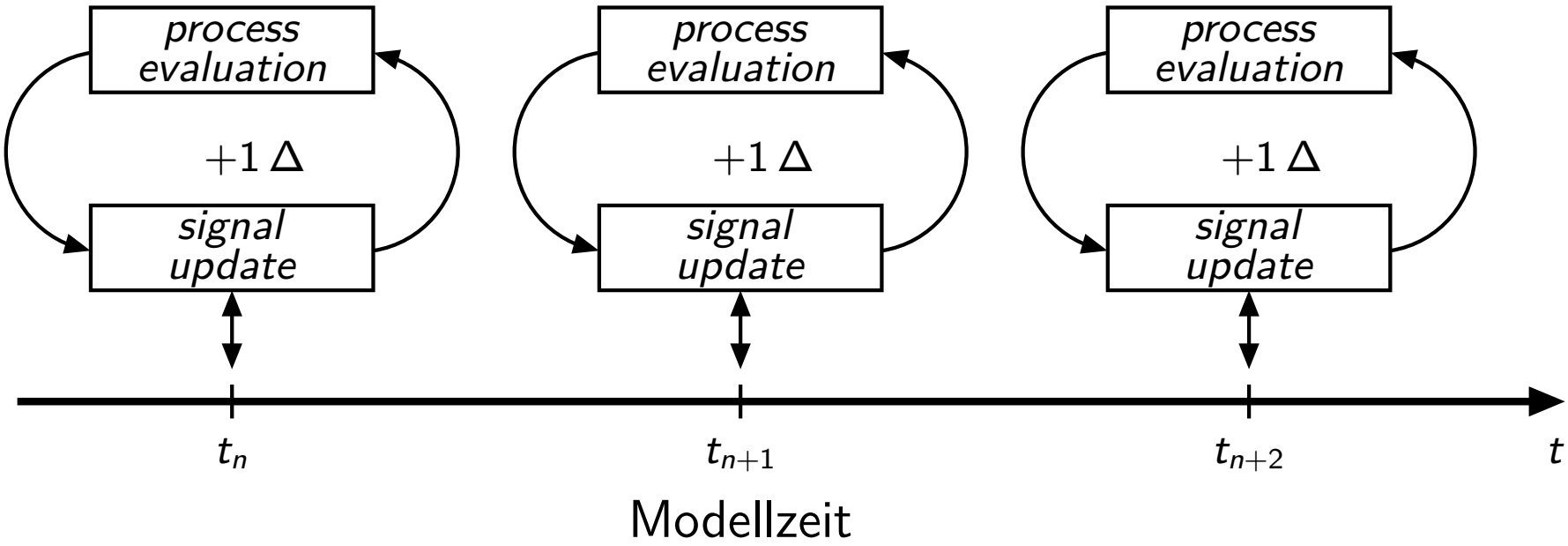
- Δ beschreibt eine unendlich kurze Zeitdauer, d. h. wenn die Modellzeit um Δ erhöht wird, verändert sich die „protokollierte“ Zeitangabe nach außen nicht.
- Pro Modellzeitpunkt werden so viele Δ-Zyklen durchlaufen, bis keine Transaktionen und Ereignisse mehr auftreten (stabiler Zustand)
- Wenn kein neuer Δ-Zyklus mehr gestartet werden kann, wird die Modellzeit auf den nächsten Zeitpunkt gesetzt (für den Transaktionen eingeplant sind) und die Evaluation des Modellzustandes beginnt von vorn mit dem 0-ten Δ-Zyklus für diesen Modellzeitpunkt
- Die meisten VHDL-Simulatoren zeigen (im Einzelschritt-Modus) zusätzlich zu der Modellzeit auch die Nummer des aktuellen Δ-Zyklus an, z. B. „1050 fs+2Δ“

Bedeutung der Δ-Verzögerung

- Im realen System hat jede Signalzuweisung (jedes Ereignis) eine Verzögerungszeit
- Im VHDL-Modell können Signalzuweisungen auch ohne Verzögerungen angegeben werden (in diesem Fall wird die Verzögerungszeit vom Simulator auf 0 Zeiteinheiten gesetzt). Bei Modellierung des Systems auf hohem Abstraktionsniveau sind die exakten Verzögerungen z. B. ohnehin nicht bekannt.

➥ Die Δ-Verzögerung stellt das funktional korrekte Verhalten des Modells her, indem die „fehlenden“ Verzögerungszeiten durch infinitesimal kleine Werte nachgebildet werden. Die Signalzuweisungen können dadurch in zeitlich korrekter Abfolge ausgeführt werden (Aufrechterhaltung des Ursache-Wirkung-Prinzips).

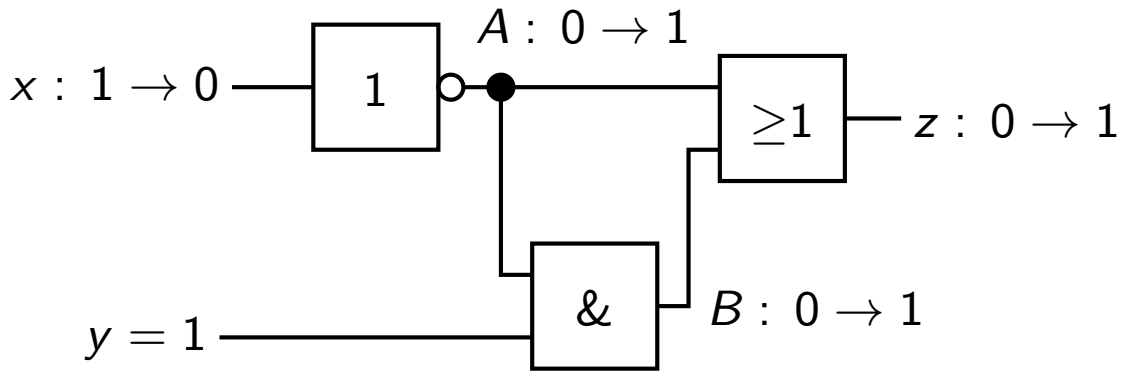
Visualisierung des Δ-Zyklus



- Variablenwerte werden sofort nach der Ausführung der entsprechenden Anweisung durch den Simulator aktualisiert
- Signalwerte werden erst am Ende eines Δ-Zyklus aktualisiert

Die Zeitpunkte t, t_{n+1}, $t_{n+2}, \ldots$ sind in der Regel nicht äquidistant verteilt.

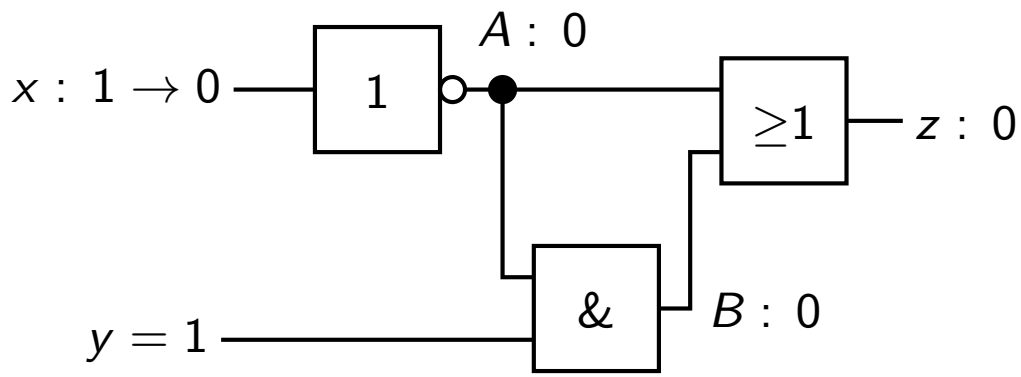
Δ-Zyklus am Beispiel einer kombinatorischen Schaltung



Stand: zum Zeitpunkt t_n ist ein Ereignis am Eingang x eingetreten (Signalwert hat sich von 1 auf 0 verändert). Der Eingang y bleibt unverändert. Ablauf der Simulation:

Modellzeit	Δ-Zyklus	Simulator-Aktivität
t_n	0.	x auswerten (Eingang)
t_n	1.	A auswerten (Inverter)
t_n	2.	B und z auswerten (UND und ODER)
t_n	3.	z auswerten (ODER)
t_{n+1}	0.	...

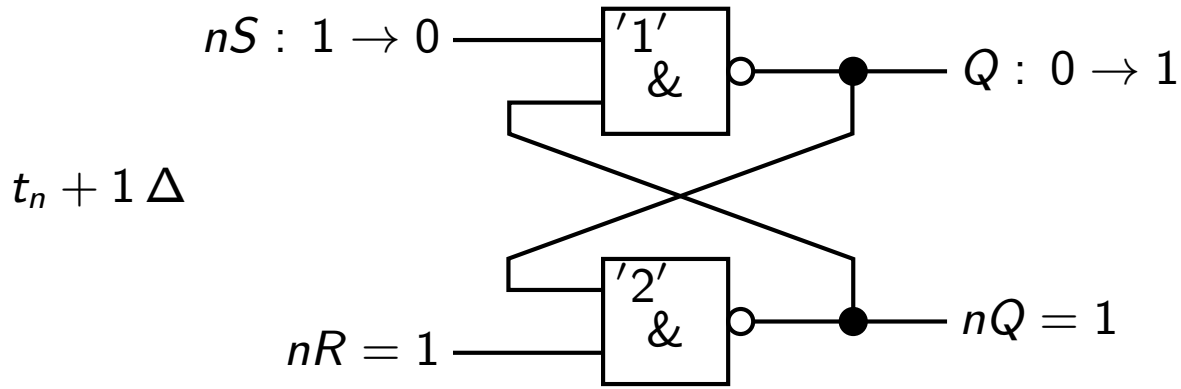
Gleiches Beispiel ohne Δ-Zyklus



- Die Veränderung am Eingang x wird registriert und vorgemerkt, jedoch noch nicht ausgeführt
- A „sieht“ zum Zeitpunkt t_n immer noch den alten Wert von x
  ➠ kein Ereignis registriert
- Es gibt keinen Anlass für eine Veränderung von B und z (keine Ereignisse an den entsprechenden Eingängen)

➥ Das Ereignis am Eingang x wird erst zum Zeitpunkt t_{n+1} der Modellzeit propagiert (Verhalten ist nicht korrekt, da ideale Gatter mit Null-Verzögerung modelliert wurden).

Δ-Zyklus am Beispiel einer sequentiellen Schaltung (1)

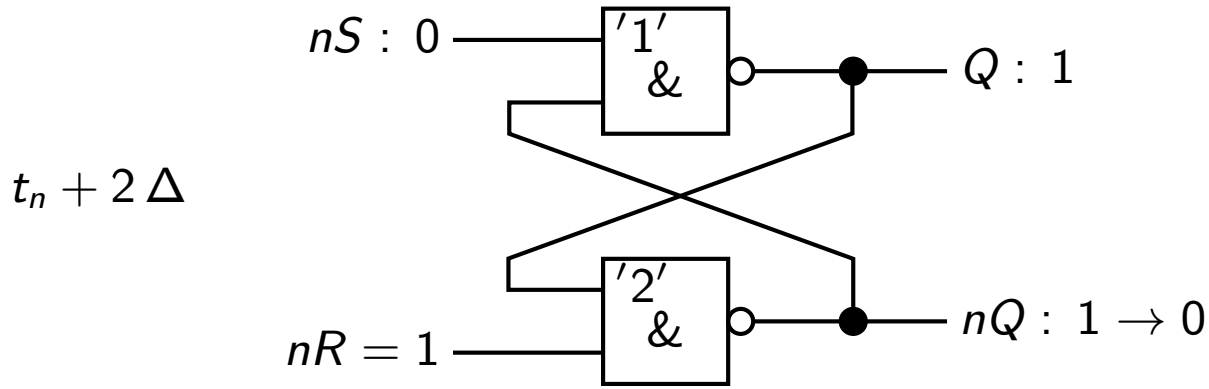



```
Q  <= NOT (nQ AND nS); -- Anweisung '1'
nQ <= NOT (Q AND nR);  -- Anweisung '2'
```

Stand: zum Zeitpunkt t_n ist ein Ereignis am Eingang nS eingetreten (Signalwert hat sich von 1 auf 0 verändert). Der Eingang nR bleibt unverändert 1. Ablauf der Simulation:

Modellzeit	Δ-Zyklus	nS	nR	Q	nQ	Auswertung
t_n	0.	$1 \to 0$	1	0	1	'1'
t_n	1.	0	1	$0 \to 1$	1	–

Δ-Zyklus am Beispiel einer sequentiellen Schaltung (2)

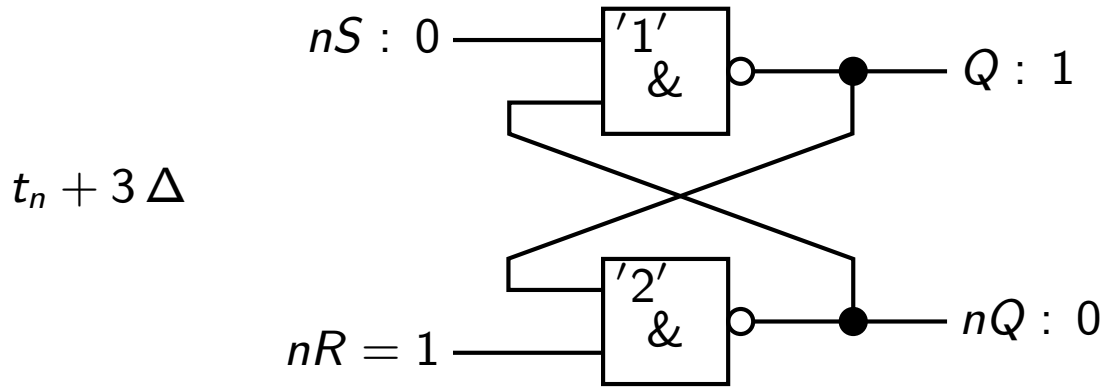



```
Q  <= NOT (nQ AND nS); -- Anweisung '1'
nQ <= NOT (Q AND nR);  -- Anweisung '2'
```

Ablauf mit 2 Δ-Verzögerungen (3 Δ-Zyklen):

Modellzeit	Δ-Zyklus	nS	nR	Q	nQ	Auswertung
t_n	0.	$1 \rightarrow 0$	1	0	1	'1'
t_n	1.	0	1	$0 \rightarrow 1$	1	'2'
t_n	2.	0	1	1	$1 \rightarrow 0$	–

Δ-Zyklus am Beispiel einer sequentiellen Schaltung (3)

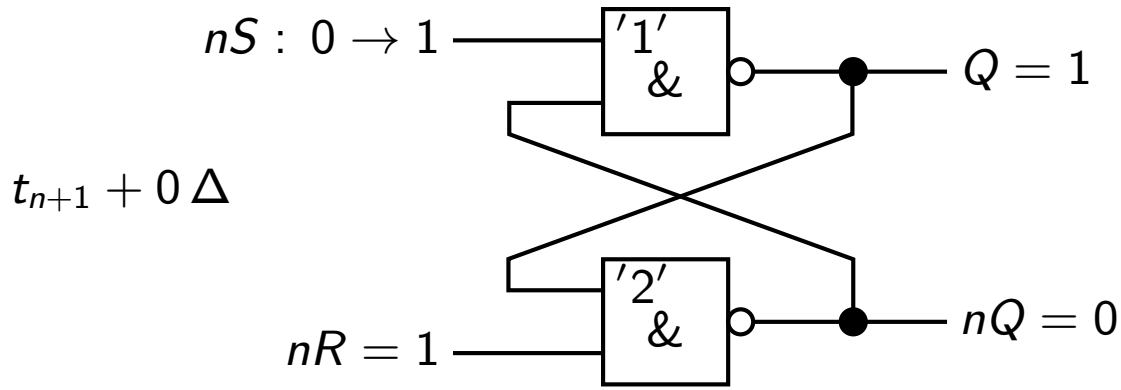



```
Q  <= NOT (nQ AND nS); -- Anweisung '1'
nQ <= NOT (Q AND nR);  -- Anweisung '2'
```

Ablauf mit 3 Δ-Verzögerungen (4 Δ-Zyklen):

Modellzeit	Δ-Zyklus	nS	nR	Q	nQ	Auswertung
t_n	0.	$1 \to 0$	1	0	1	$'1'$
t_n	1.	0	1	$0 \to 1$	1	$'2'$
t_n	2.	0	1	1	$1 \to 0$	$'1'$
t_n	3.	0	1	1	0	– (stabil)

Δ-Zyklus am Beispiel einer sequentiellen Schaltung (4)

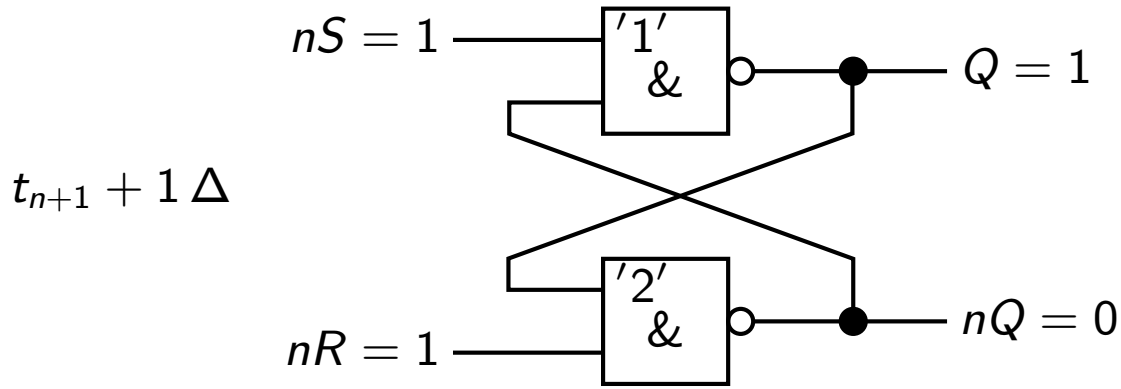



```
Q  <= NOT (nQ AND nS); -- Anweisung '1'
nQ <= NOT (Q AND nR);  -- Anweisung '2'
```

Stand: zum Zeitpunkt t_{n+1} ist ein Ereignis am Eingang nS eingetreten (Signalwert hat sich von 0 auf 1 verändert). Der Eingang nR bleibt unverändert 1. Ablauf der Simulation:

Modellzeit	Δ-Zyklus	nS	nR	Q	nQ	Auswertung
t_n	3.	0	1	1	0	– (stabil)
t_{n+1}	0.	$0 \rightarrow 1$	1	1	0	'1'

Δ-Zyklus am Beispiel einer sequentiellen Schaltung (5)

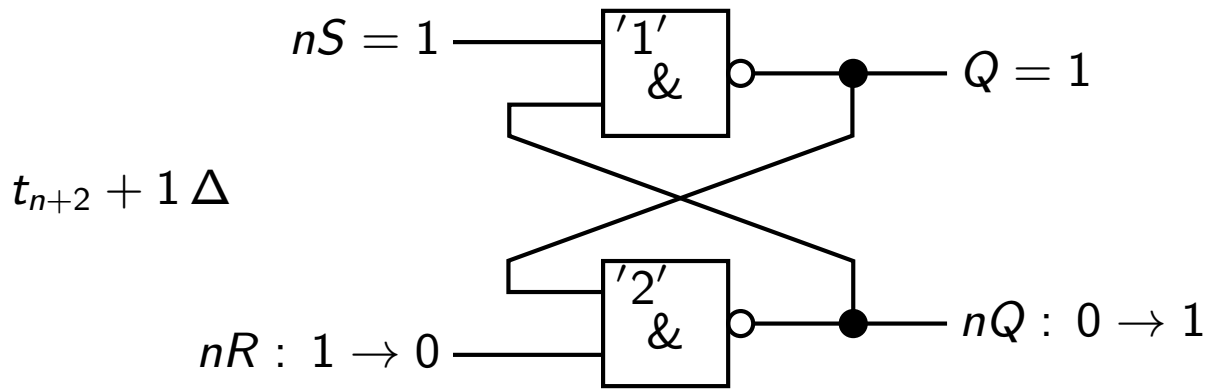



```
Q  <= NOT (nQ AND nS); -- Anweisung '1'
nQ <= NOT (Q AND nR);  -- Anweisung '2'
```

Ablauf mit 1 Δ-Verzögerung (2 Δ-Zyklen):

Modellzeit	Δ-Zyklus	nS	nR	Q	nQ	Auswertung
t_n	3.	0	1	1	0	– (stabil)
t_{n+1}	0.	$0 \rightarrow 1$	1	1	0	'1'
t_{n+1}	1.	1	1	1	0	– (stabil)

Δ-Zyklus am Beispiel einer sequentiellen Schaltung (6)

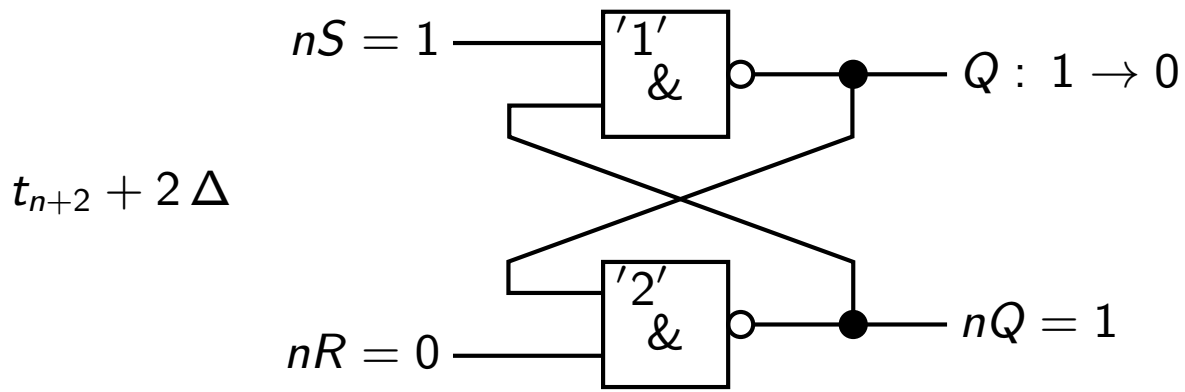



```
Q  <= NOT (nQ AND nS); -- Anweisung '1'
nQ <= NOT (Q AND nR);  -- Anweisung '2'
```

Stand: zum Zeitpunkt t_{n+2} ist ein Ereignis am Eingang nR eingetreten (Signalwert hat sich von 1 auf 0 verändert). Der Eingang nS bleibt unverändert '1'. Ablauf der Simulation:

Modellzeit	Δ-Zyklus	nS	nR	Q	nQ	Auswertung
t_{n+1}	1.	1	1	1	0	– (stabil)
t_{n+2}	0.	1	$1 \to 0$	1	0	'2'
t_{n+2}	1.	1	0	1	$0 \to 1$	–

Δ-Zyklus am Beispiel einer sequentiellen Schaltung (7)

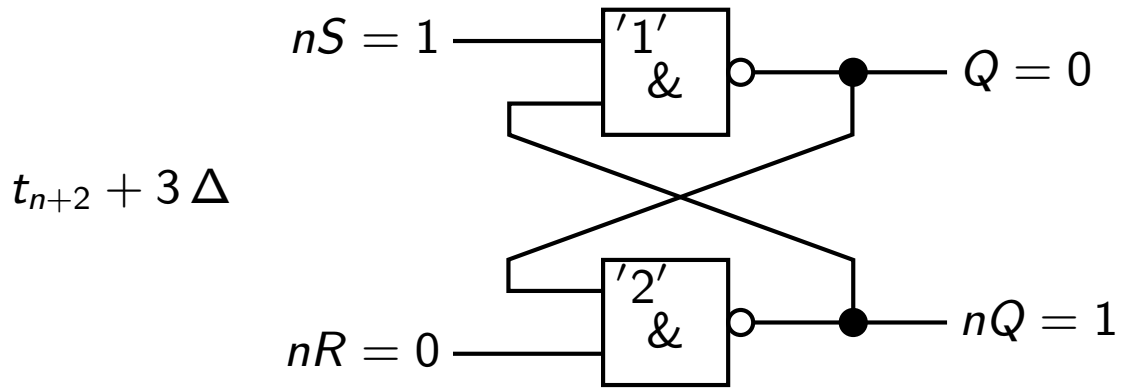



```
Q  <= NOT (nQ AND nS); -- Anweisung '1'
nQ <= NOT (Q AND nR);  -- Anweisung '2'
```

Ablauf mit 2 Δ-Verzögerungen (3 Δ-Zyklen):

Modellzeit	Δ-Zyklus	nS	nR	Q	nQ	Auswertung
t_{n+1}	1.	1	1	1	0	– (stabil)
t_{n+2}	0.	1	$1 \to 0$	1	0	'2'
t_{n+2}	1.	1	0	1	$0 \to 1$	'1'
t_{n+2}	2.	1	0	$1 \to 0$	1	–

Δ-Zyklus am Beispiel einer sequentiellen Schaltung (8)




```
Q  <= NOT (nQ AND nS); -- Anweisung '1'
nQ <= NOT (Q AND nR);  -- Anweisung '2'
```

Ablauf mit 3 Δ-Verzögerungen (4 Δ-Zyklen):

Modellzeit	Δ-Zyklus	nS	nR	Q	nQ	Auswertung
t_{n+1}	1.	1	1	1	0	– (stabil)
t_{n+2}	0.	1	$1 \to 0$	1	0	'2'
t_{n+2}	1.	1	0	1	$0 \to 1$	'1'
t_{n+2}	2.	1	0	$1 \to 0$	1	'2'
t_{n+2}	3.	1	0	0	1	– (stabil)

Das letzte Beispiel bildet ein RS-Flipflop nach

- Grundbaustein sind NAND-Gatter, d. h. nR- und nS-Eingänge sind low-aktiv (entsprechend mit n gekennzeichnet)
- im Modellzeitpunkt t_n erfolgt das Setzen des Flipflops: der Eingang nS wechselt von '1' auf '0'
- im Modellzeitpunkt t_{n+1} speichert das Flipflop den vorher gesetzten Wert: Beide Eingänge sind auf '1'
- im Modellzeitpunkt t_{n+2} erfolgt das Zurücksetzen des Flipflops: der Eingang nR wechselt von '1' auf '0'

➥ durch das Konzept der Δ-Verzögerung wird das korrekte Verhalten des Flipflops auch ohne explizite Angaben der Gatterverzögerungszeiten nachgebildet

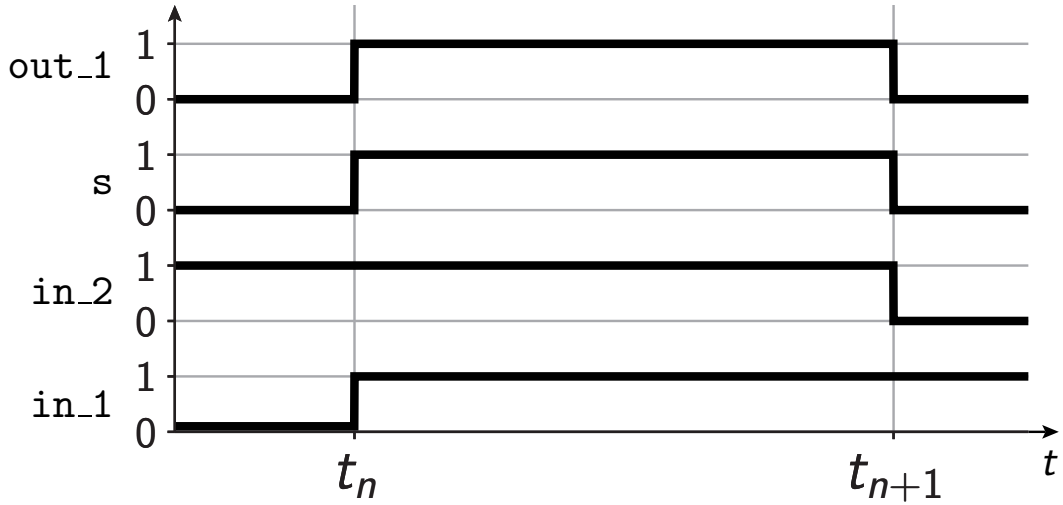
Fehlerquellen

Das Negieren bzw. Fehlinterpretieren des Δ-Konzeptes ist eine häufige Fehlerquelle:

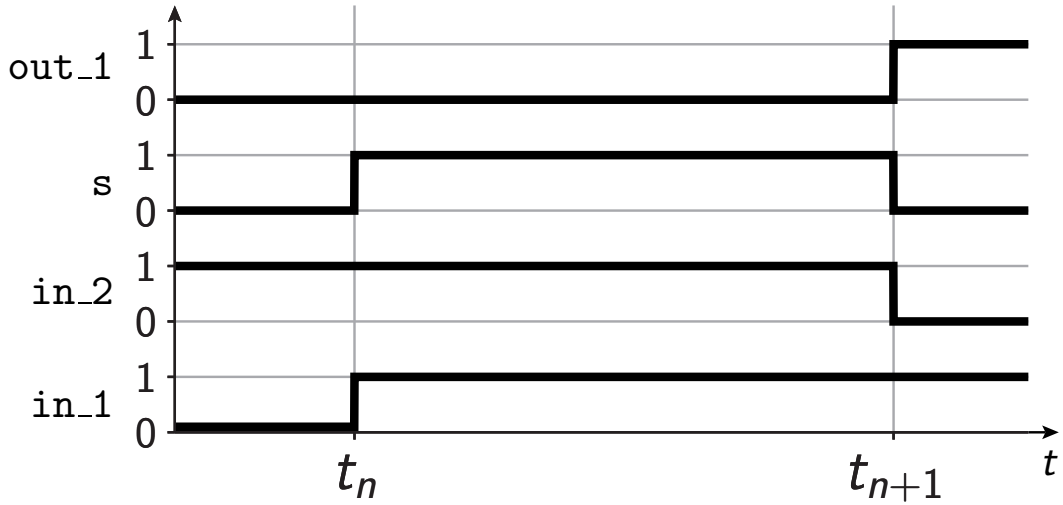
```
ENTITY beispiel IS
PORT (in_1, in_2 : IN bit;
      out_1      : OUT bit);
END ENTITY beispiel;

ARCHITECTURE behaviour OF beispiel IS
SIGNAL s : bit;
BEGIN
  p1: PROCESS (in_1, in_2)
  BEGIN
    s <= in_1 AND in_2; -- Anweisung '1'
    out_1 <= s;         -- Anweisung '2'
  END PROCESS p1;
END ARCHITECTURE behaviour;
```

Erwarteter und tatsächlicher Signalverlauf

Erwarteter Signalverlauf nach oberflächlicher Betrachtung



Tatsächlicher Signalverlauf:



Erläuterung zum letzten Beispiel

- `p1` wird zum Modellzeitpunkt t_n aufgrund eines Ereignisses am `in_1` aktiviert (Prozessausführungsphase des 1. Δ-Zyklus).
- Anweisung $'1'$ wird ausgewertet, **Werteveränderung von `s` wird als Ereignis vermerkt, ist jedoch noch nicht wirksam!**
- Anweisung $'2'$ wird mit dem „alten“ Wert von `s` ausgewertet, d. h. am Ausgang `out_1` tritt kein Ereignis ein.
- Der Δ-Zyklus wird mit der Signalzuweisungsphase beendet, `s` erhält nun einen aktualisierten Wert.
- Keine Ereignisse mehr, die Modellzeit wird auf t_{n+1} gesetzt
- Der Prozess `p1` wird aufgrund des Ereignisses am Eingang `in_2` aktiviert. Erst jetzt wird dem Ausgang `out_1` bei der Auswertung der Anweisung $'2'$ der „neue“ Wert von `s` **aus dem letzten Durchlauf** zugewiesen!

Ursache für das scheinbare „Fehlverhalten“ des Modells

- Anweisungen ′1′ und ′2′ stehen hintereinander innerhalb einer `PROCESS`-Konstruktion

➥ Es sind sequentielle Anweisungen, sie werden in der Reihenfolge ihres Auftretens ausgewertet

- Eine Neuauswertung kann nur bei folgender Aktivierung des Prozesses erfolgen, diese ist nur nach einem Ereignis an mindestens einem der Signale aus der Empfindlichkeitsliste möglich

Lösungsansätze:

- Anweisungen ′1′ und ′2′ nebenläufig modellieren (zwei getrennte Prozesse)
- Signal `s` zur Empfindlichkeitsliste hinzufügen

Signal vs. Variable

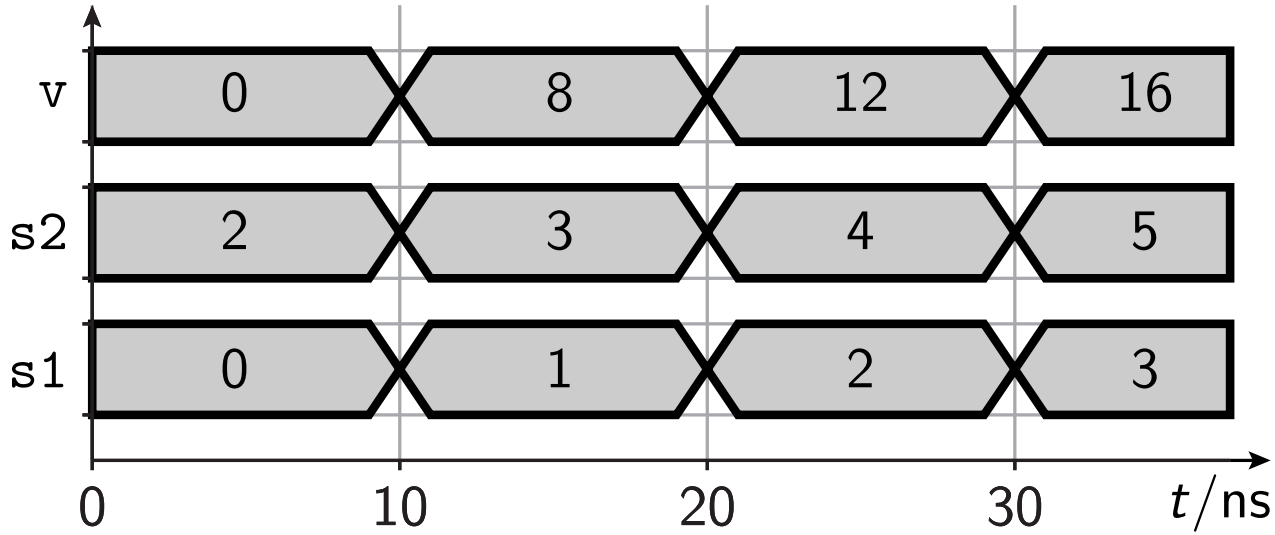
- Variable bekommen einen neuen Wert mit der Auswertung der entsprechenden Anweisung zugewiesen, Signale erst in der Zuweisungsphase des entsprechenden Δ-Zyklus
- Variable hat einen Momentanwert, der nicht zwangsläufig ereignisgebunden sein muss, Signal ist ein zeitabhängiger Informationsträger
- Variable ist ein (prozess-)interner Zwischenspeicher, Signal ist ein Kommunikationsmittel (zwischen den Prozessen)

➥ Das unterschiedliche Verhalten muss bei gleichzeitiger Verwendung von Variablen und Signalen in einem Prozess berücksichtigt werden!

Ein Prozess mit Signalen und Variablen

```
ENTITY sig_var IS
  -- leer
END ENTITY sig_var;

ARCHITECTURE behaviour OF sig_var IS
  SIGNAL s1, s2 : integer := 0;
BEGIN
  s1 <= 1 AFTER 10 ns, 2 AFTER 20 ns, 3 AFTER 30 ns;
  p1 : PROCESS (s1)
    VARIABLE v : integer := 0;
  BEGIN
    s2 <= s1 + 2;
    v := 4 * s2;
  END PROCESS p1;
END ARCHITECTURE behaviour;
```

Zeitlicher Verlauf



Bei der Auswertung der Variablenzuweisung hat s2 noch den Wert aus dem vorhergehenden Prozessdurchlauf, daher erscheint das Ergebnis nach außen „um ein Ereignis verspätet“.

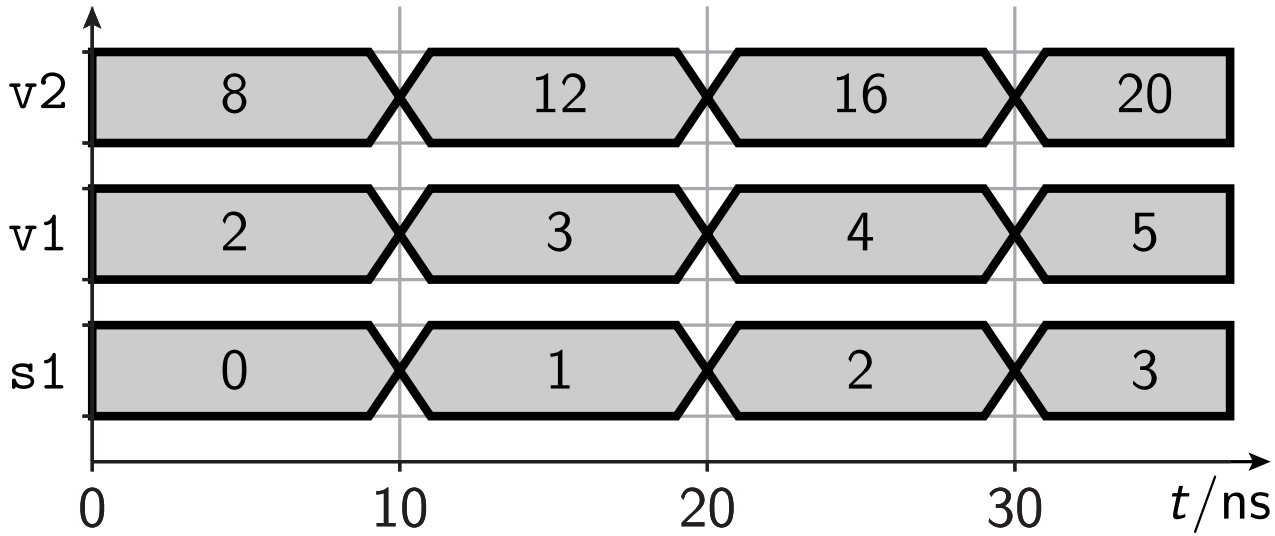
Grund: s2 erhält den aktualisierten Wert erst in der Signalzuweisungsphase des Δ-Zyklus zugewiesen, v wird jedoch bereits in der Prozessausführungsphase mit sofortiger Wirkung aktualisiert.

Lösungsansatz

Ist das gezeigte Verhalten nicht erwünscht, kann z. B. folgende Lösung Abhilfe schaffen (nur Variable):

```
ENTITY sig_var IS
  -- leer
END ENTITY sig_var;
ARCHITECTURE behaviour OF sig_var IS
  SIGNAL s1 : integer := 0;
BEGIN
  s1 <= 1 AFTER 10 ns, 2 AFTER 20 ns, 3 AFTER 30 ns;
  p1 : PROCESS (s1)
    VARIABLE v1, v2 : integer :=0;
  BEGIN
    v1 := s1 + 2;
    v2 := 4 * v1;
  END PROCESS p1;
END ARCHITECTURE behaviour;
```

Zeitlicher Verlauf mit Variablen



Die Variablenwerte werden unmittelbar nach der Auswertung der jeweiligen Anweisung aktualisiert, d. h. in der Prozessausführungsphase. Da der Prozess immer aufgrund eines Ereignisses am Signal `s1` ausgeführt wird (d. h. `s1` hat bereits einen „neuen“ Wert), verhalten sich die Variablenwerte wie erwartet.

Zusammenfassung

- Grundzüge des Simulationsmodells von VHDL
- Auswirkungen des Δ-Konzeptes auf den Simulationsablauf
- Unterschiede zwischen Signalen und Variablen

Systementwurf mit VHDL: Vorlesungs-Gesamtübersicht

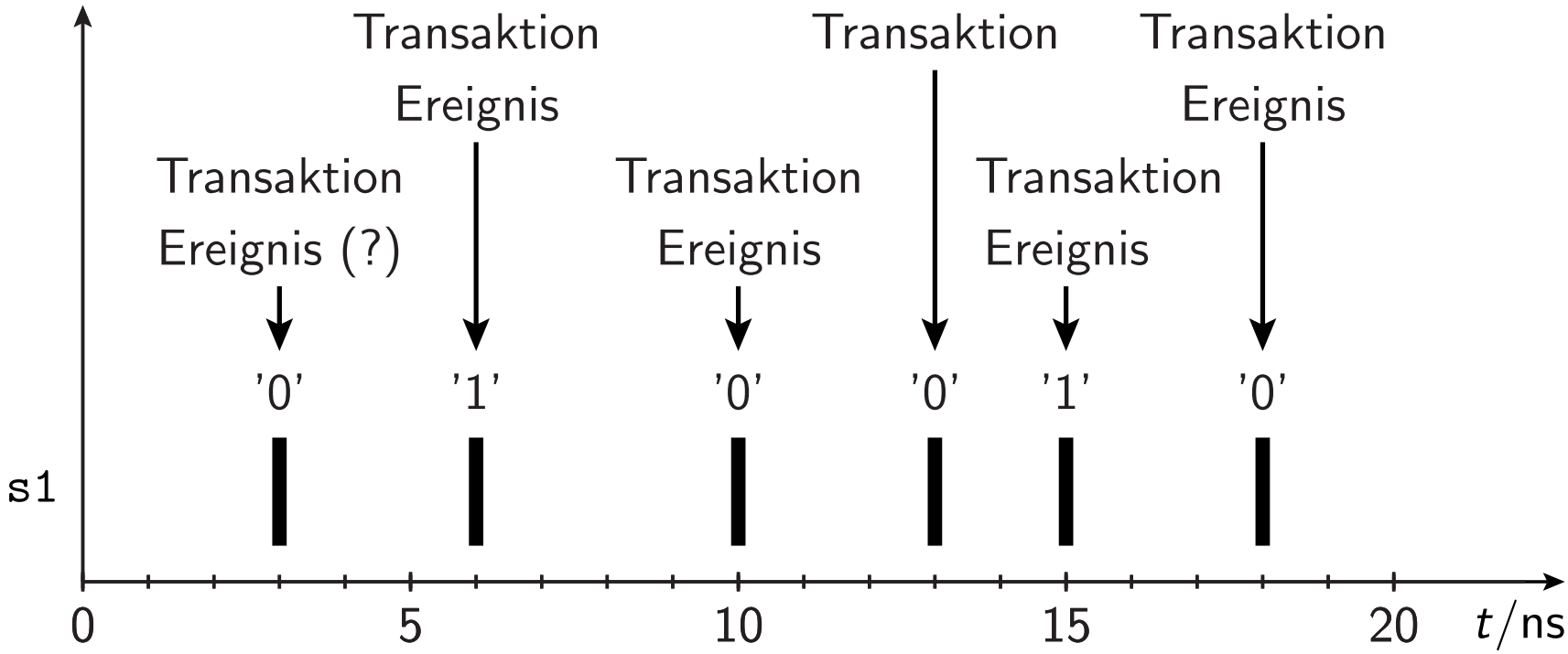
Systementwurf mit VHDL: Gliederung der 9. Vorlesung

Interne Verwaltung von Signalen

- Signalzuweisung sieht die Angabe einer Verzögerungszeit vor
- Wird keine Verzögerungszeit angegeben, so wird implizit der Wert von 0 Zeiteinheiten angenommen
- Alle Wertezuweisungen an ein Signal (sowohl Transaktionen als auch Ereignisse) werden in einer Ereignisliste (mindestens eine pro Signal) verwaltet
- Mit fortschreitender Modellzeit wird die Ereignisliste ständig aktualisiert: Es kommen neue Transaktionen und Ereignisse hinzu, bereits geplante Transaktionen und Ereignisse können unter Umständen gestrichen werden (d. h. sie bleiben wirkungslos)
- Die Verwaltung der Ereignisliste erfolgt nach dem Preemption-Mechanismus, der durch das Verzögerungsmodell bestimmt wird

Beispiel einer Signalzuweisung mit zugehöriger Ereignisliste

```
SIGNAL s1 : bit;
...
s1 <= '0' AFTER  3 ns, '1' AFTER  6 ns,
      '0' AFTER 10 ns, '0' AFTER 13 ns,
      '1' AFTER 15 ns, '0' AFTER 18 ns;
```



Steuerung des Preemption-Mechanismus durch Angabe des Verzögerungsmodells

Inertial: Alle (noch nicht wirksam gewordenen) Ereignisse, die vor und nach der Zeit der neuen Wertezuweisung stattfinden, werden aus der Ereignisliste ersatzlos gestrichen; entspricht dem Verzögerungsverhalten eines Gatters. Wird eingesetzt, wenn kein Verzögerungsmodell angegeben ist (default-Modell).

Transport: Alle Ereignisse, die nach der Zeit der neuen Wertezuweisung oder gleichzeitig damit stattfinden, werden aus der Ereignisliste ersatzlos gestrichen; entspricht dem Verzögerungsverhalten einer Leitung.

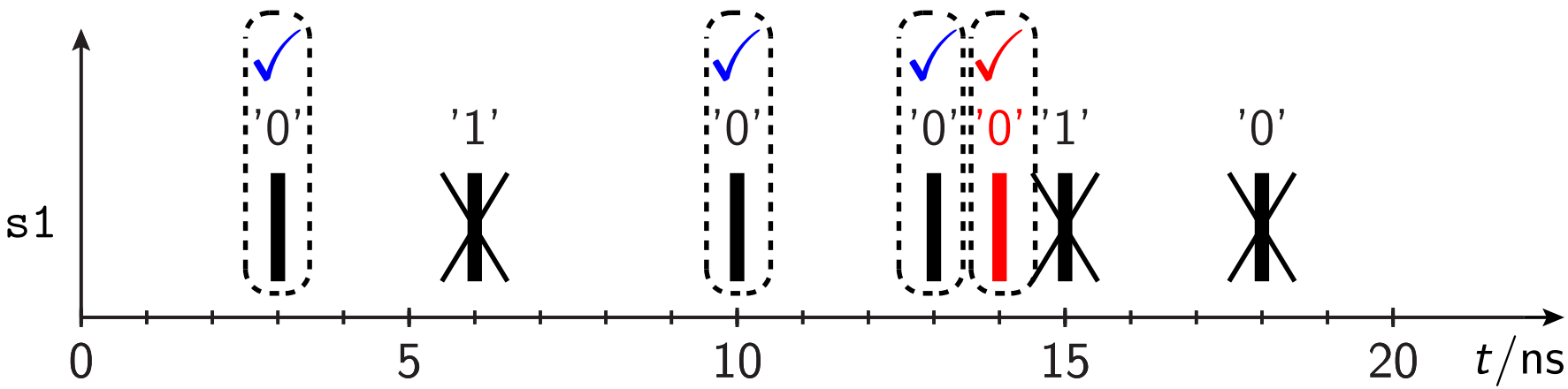
Reject: Wie Inertial, jedoch werden nicht alle Ereignisse vor der Zeit der neuen Zuweisung gestrichen, sondern nur solche, die in einer Signalimpulsdauer resultieren, die kürzer als ein vorgegebener Wert ist (oder diesem Wert gleicht).

Inertial-Verzögerungsmodell

Anpassung der Ereignisliste nach der Ankunft einer neuen Zuweisung:

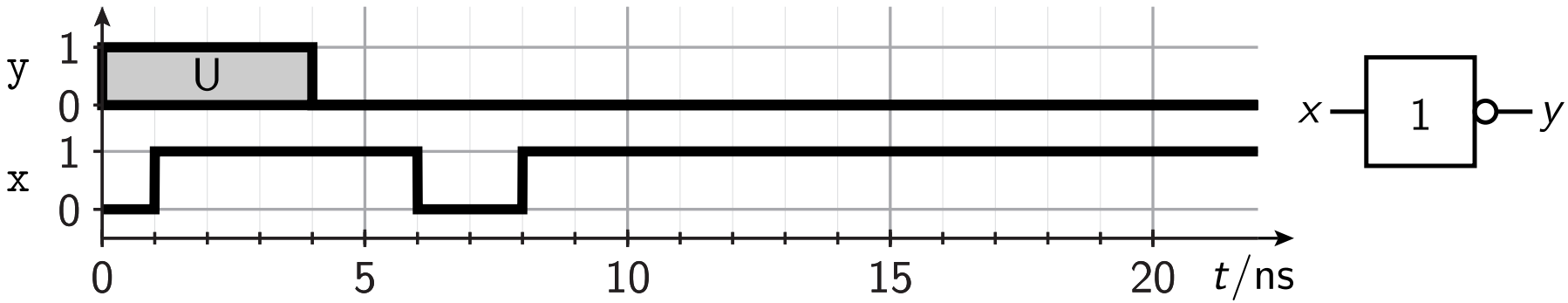
1. Markiere alle Transaktionen, die unmittelbar vor dem neuen Eintrag statt finden, wenn sie den gleichen Signalwert herbeiführen
2. Markiere die aktuelle und die neue Transaktion
3. Entferne alle nichtmarkierten Transaktionen

```
s1 <= '0' AFTER 11 ns; -- bei Modellzeit 3 ns
```



Inertial-Verzögerungsmodell am Beispiel eines NOT-Gatters

```
y <= INERTIAL NOT x AFTER 3 ns; -- x, y sind std_logic
```



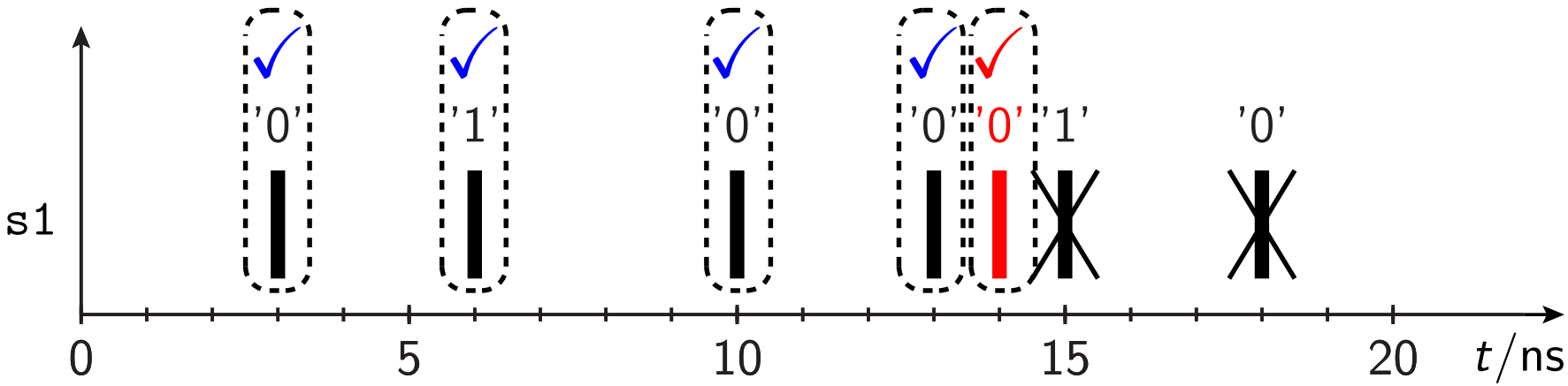
- Zum Zeitpunkt $t = 1\,\text{ns}$ wird die Transaktion für y geplant (x wird zu '1', d. h. y wird zu '0'), durch die Verzögerung von 3 ns wird diese auf die Zeit von 4 ns festgelegt (und auch ausgeführt)
- Zum Zeitpunkt $t = 6\,\text{ns}$ wird eine Transaktion für $t = 9\,\text{ns}$ geplant. Da x aber zum Zeitpunkt $t = 8\,\text{ns}$ den Wert schon wieder verändert, wird diese ersatzlos gestrichen

Transport-Verzögerungsmodell

Anpassung der Ereignisliste nach der Ankunft einer neuen Zuweisung:

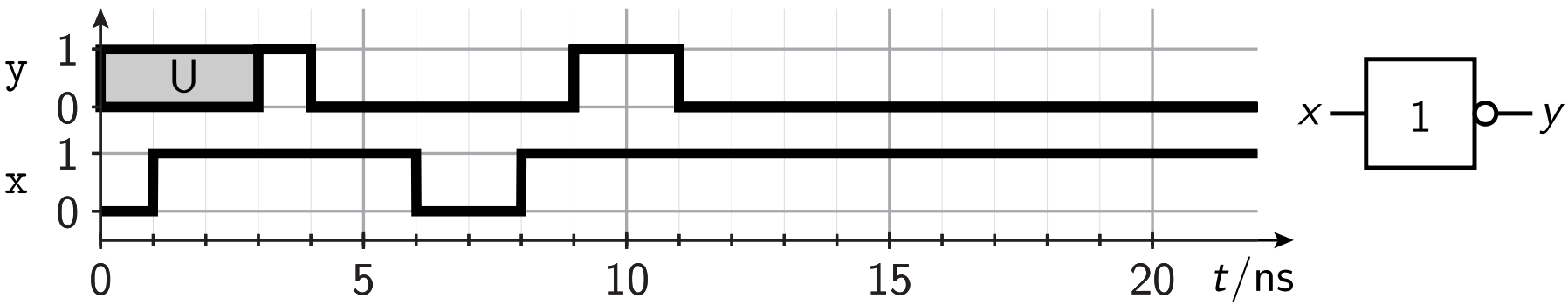
1. Markiere die neue Transaktion und alle Transaktionen, die früher als die neue Transaktion auftreten
2. Entferne alle nichtmarkierten Transaktionen

```
s1 <= TRANSPORT '0' AFTER 11 ns;
  -- bei Modellzeit 3 ns
```



Transport-Verzögerungsmodell am Beispiel eines NOT-Gatters

```
y <= TRANSPORT NOT x AFTER 3 ns; -- x, y sind std_logic
```



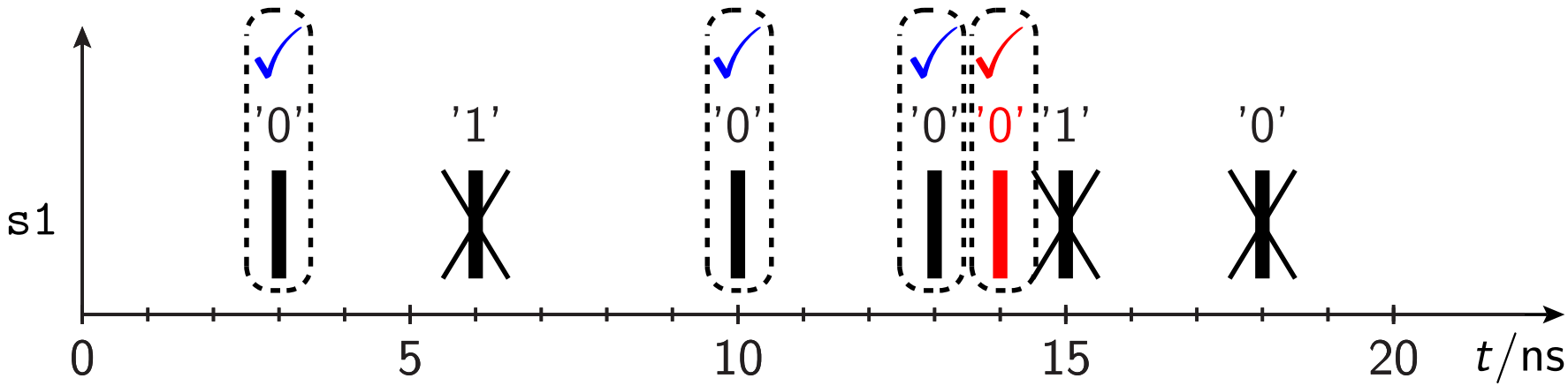
- Zum Zeitpunkt $t = 1\,\text{ns}$ wird die Transaktion für y geplant (x wird zu '1', d. h. y wird zu '0'), durch die Verzögerung von 3 ns wird diese auf die Zeit von 4 ns festgelegt (und auch ausgeführt)
- Zum Zeitpunkt $t = 6\,\text{ns}$ wird eine Transaktion für $t = 9\,\text{ns}$ geplant (und auch ausgeführt)
- Zum Zeitpunkt $t = 8\,\text{ns}$ wird eine Transaktion für $t = 11\,\text{ns}$ geplant (und auch ausgeführt)

Reject-Verzögerungsmodell

Anpassung der Ereignisliste nach der Ankunft einer neuen Zuweisung:

1. Markiere die aktuelle Transaktion und alle Transaktionen, die eine Impulsdauer **länger** als die Zeitvorgabe erzeugen (markiere aufeinander folgende Transaktionen innerhalb der vorgegebener Impulsdauer vor dem neuen Eintrag, wenn sie den gleichen Signalwert herbeiführen)
2. Entferne alle nichtmarkierten Transaktionen

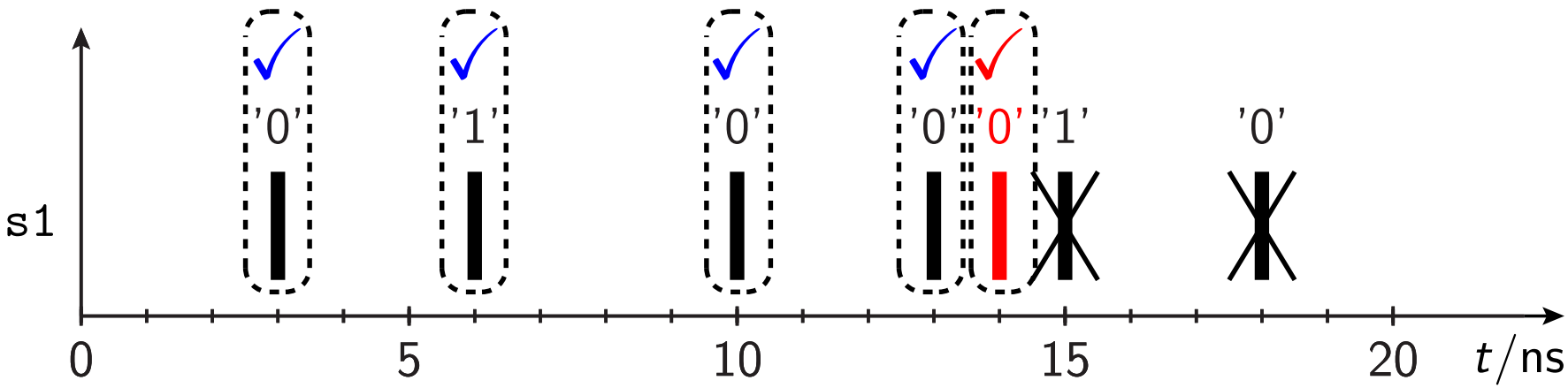
```
s1 <= REJECT 5 ns INERTIAL '0' AFTER 11 ns;
  -- bei Modellzeit 3 ns
```



Reject-Verzögerung mit einer anderen Zeitvorgabe

Letztes Beispiel erzeugte das gleiche Zeitverhalten wie das Inertial-Modell. Bei einer kleineren Zeitvorgabe ist das Verhalten jedoch unterschiedlich (und entspricht dem `TRANSPORT`-Modell):

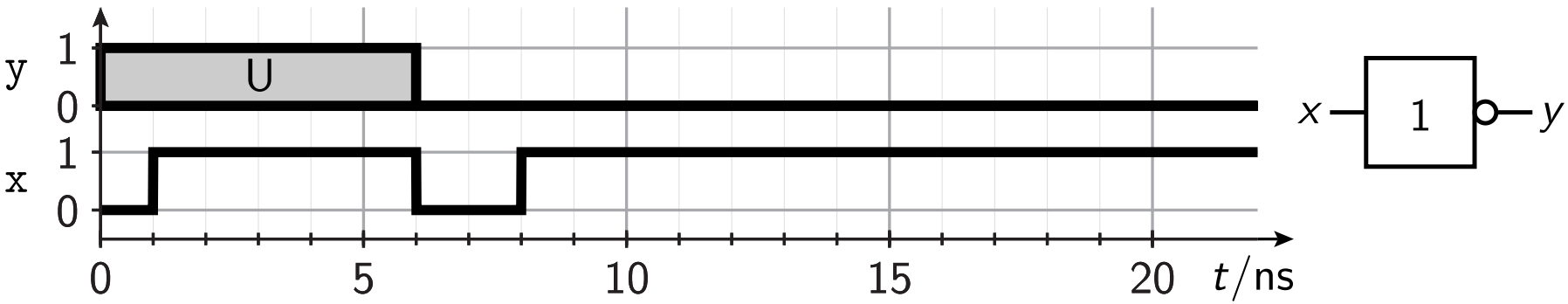
```
s1 <= REJECT 1 ns INERTIAL '0' AFTER 11 ns;
  -- bei Modellzeit 3 ns
```



Da die Transaktionen bei 6 und 10 ns einen Impuls erzeugen, der breiter als 1 ns ist, werden sie nicht unterdrückt!

Reject-Verzögerungsmodell am Beispiel eines NOT-Gatters

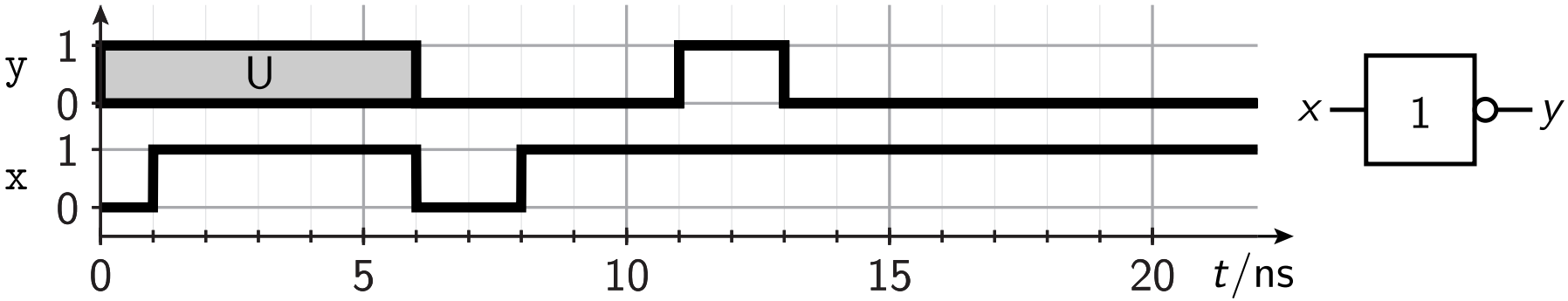
```
y <= REJECT 4 ns INERTIAL NOT x AFTER 5 ns;
```



- Zum Zeitpunkt $t = 1\,\text{ns}$ wird die Transaktion für y geplant (x wird zu '1', d. h. y wird zu '0'), durch die Verzögerung von 5 ns wird diese auf die Zeit von 6 ns festgelegt (und auch ausgeführt)
- Der Impuls am Eingang x zwischen 6 und 8 ns wird unterdrückt, da seine Breite unter 4 ns liegt

Reject-Verzögerungsmodell am Beispiel eines NOT-Gatters (fortgesetzt)

```
y <= REJECT 1 ns INERTIAL NOT x AFTER 5 ns;
```



- Zum Zeitpunkt $t = 1\,\text{ns}$ wird die Transaktion für y geplant (x wird zu '1', d. h. y wird zu '0'), durch die Verzögerung von 5 ns wird diese auf die Zeit von 6 ns festgelegt (und auch ausgeführt)
- Der Impuls am Eingang x zwischen 6 und 8 ns wird nicht unterdrückt, da seine Breite über 1 ns liegt. Durch die Verzögerung von 5 ns erscheint er am Ausgang y zwischen 11 und 13 ns (invertiert)

Mehrdeutigkeiten bei Signalzuweisungen

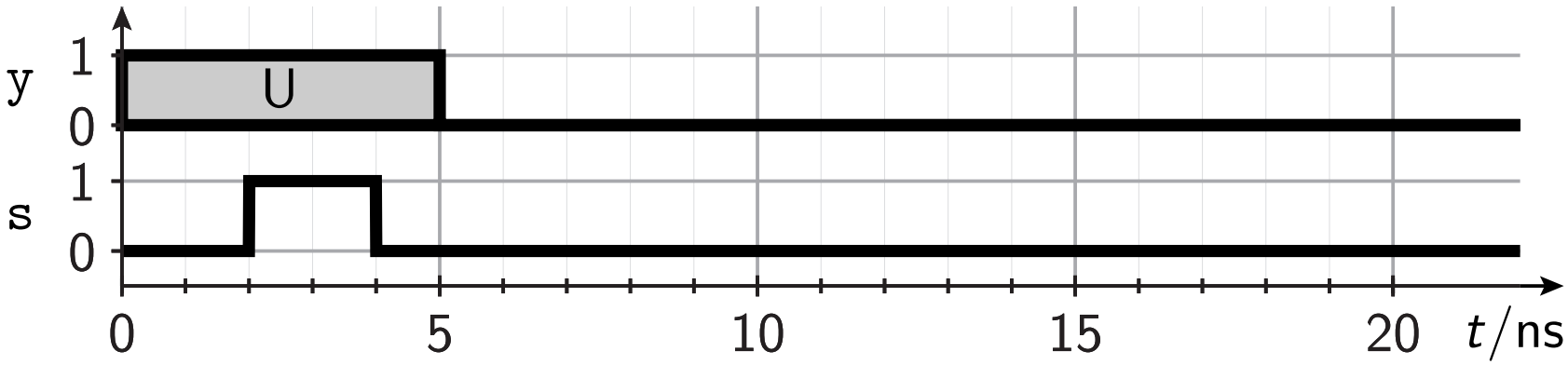
Hier am Beispiel unterschiedlicher HL- und LH-Verzögerungszeiten *(asymmetric transport delay)*:

```
...
 p1 : PROCESS (s)
  BEGIN
    IF s='1' THEN
      y <= TRANSPORT s AFTER 8 ns;
    ELSE
      y <= TRANSPORT s AFTER 5 ns;
    END IF;
  END PROCESS p1;
...
```

Das Verhalten des Eingangssignals:

```
s <= '0', '1' AFTER 2 ns, '0' AFTER 4 ns;
```

Der entsprechende Signalverlauf



- Zum Zeitpunkt 2 ns wird eine Transaktion geplant, die den Signalwert zum Zeitpunkt 10 ns auf '1' setzt
- Zum Zeitpunkt 4 ns wird eine Transaktion geplant, die den Signalwert zum Zeitpunkt 9 ns auf '0' setzt (die Transaktion von 10 ns wird gelöscht)

➥ Das Signal *y* wird mit zum Zeitpunkt 5 ns auf '0' gesetzt und bleibt bei diesem Wert.

Grundsätzliche Vorgehensweise bei mehreren Treibern eines Signals

- Jeder Treiber erhält eine eigene Ereignisliste
- Treten bei der Auswertung der Ereignislisten verschiedener Treiber des gleichen Signals Konflikte auf, so werden diese durch die entsprechende Auflösungsfunktion behandelt (zur Erinnerung: Ein Signal mit mehreren Treibern muss vom aufgelösten Typ sein!)

Zusammengefasst ist die Simulation eine ständige Aktualisierung von Ereignislisten (mit Aufrufen der Auflösungsfunktionen bei Bedarf) unter Berücksichtigung der entsprechenden Verzögerungsmodelle (das Δ-Konzept stellt sicher, dass auch bei fehlenden Verzögerungszeit-Angaben ein korrektes Verhalten des Modells nachgebildet werden kann).

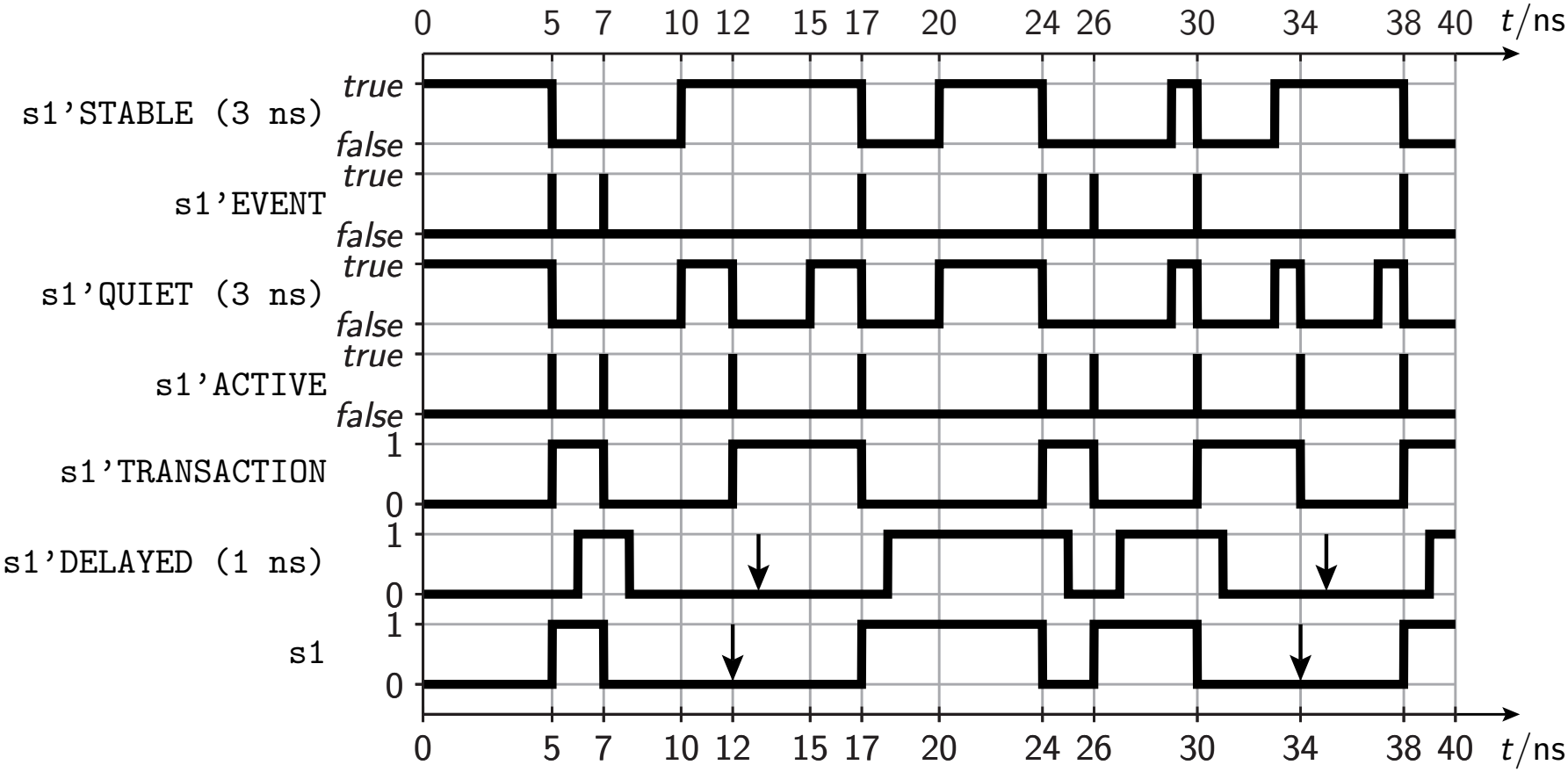
Signalbezogene Attribute

`a'DELAYED(t)`	Signal `a` um die Zeit `t` verzögert
`a'TRANSACTION`	Signal vom Typ `bit`, das in jedem Simulationszyklus wechselt, in dem `a` aktiv ist
`a'STABLE(t)`	`true` wenn `a` Zeit `t` ohne Ereignis war, sonst `false`
`a'QUIET(t)`	`true` wenn `a` Zeit `t` inaktiv war, sonst `false`
`a'EVENT`	`true` wenn `a` im aktuellen Simulationszyklus ein Ereignis hatte, sonst `false`
`a'ACTIVE`	`true` wenn `a` im aktuellen Simulationszyklus aktiv ist, sonst `false`
`a'LAST_EVENT`	Zeitdifferenz zwischen der aktuellen Modellzeit und dem letzten Ereignis auf `a`
`a'LAST_ACTIVE`	Zeitdifferenz zwischen der aktuellen Modellzeit und dem letzten aktiven Zeitpunkt von `a`
`a'LAST_VALUE`	Wert von `a` vor dem letzten Ereignis

Beispiele für signalbezogene Attribute

s1 ist mit '0' initialisiert:

```
s1 <= '1' AFTER  5 ns, '0' AFTER  7 ns, '0' AFTER 12 ns,
      '1' AFTER 17 ns, '0' AFTER 24 ns, '1' AFTER 26 ns,
      '0' AFTER 30 ns, '0' AFTER 34 ns, '1' AFTER 38 ns;
```



Zusammenfassung

- Verzögerungsmodelle und ihre Auswirkungen auf die Aktualisierung der Ereignisliste
- Abschließende Bemerkungen zur Simulation

Aufgaben zum Selbststudium (1)

1. In einem VHDL-Modell ist ein Signalverlauf (Signaltyp ist `bit`) mit

```
x <= '1' AFTER 1 ns,  '0' AFTER 4 ns,
     '1' AFTER 6 ns,  '0' AFTER 10 ns,
     '1' AFTER 15 ns, '1' AFTER 20 ns;
```

   angegeben. Außerdem findet man im gleichen Modell folgende Signalzuweisungen (ebenfalls `bit`-Typen):

```
a <= NOT x AFTER 3 ns;
b <= TRANSPORT x AFTER 3 ns;
c <= REJECT 3 ns INERTIAL NOT x AFTER 4 ns;
```

   Zeichnen Sie das Impulsfolgediagramm, das bei der Simulation dieses Modells entsteht.

Aufgaben zum Selbststudium (2)

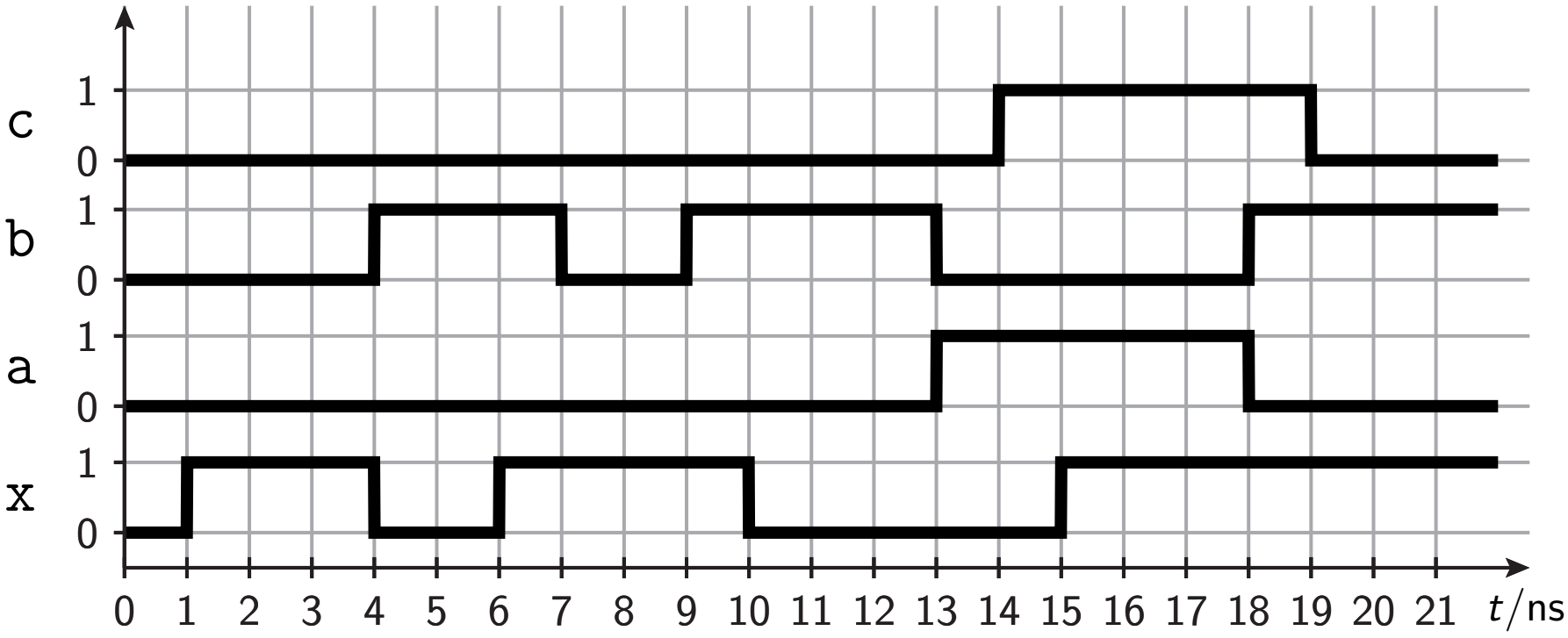
2. Erimtteln Sie die Signalverläufe von `y` im Beispiel auf der Seite 269 für den Eingangssignalverlauf

   ```
   s <= '0', '1' AFTER 2 ns, '0' AFTER 4 ns,
     '1' AFTER 6 ns, '1' AFTER 8 ns '0' AFTER 10 ns;
   ```

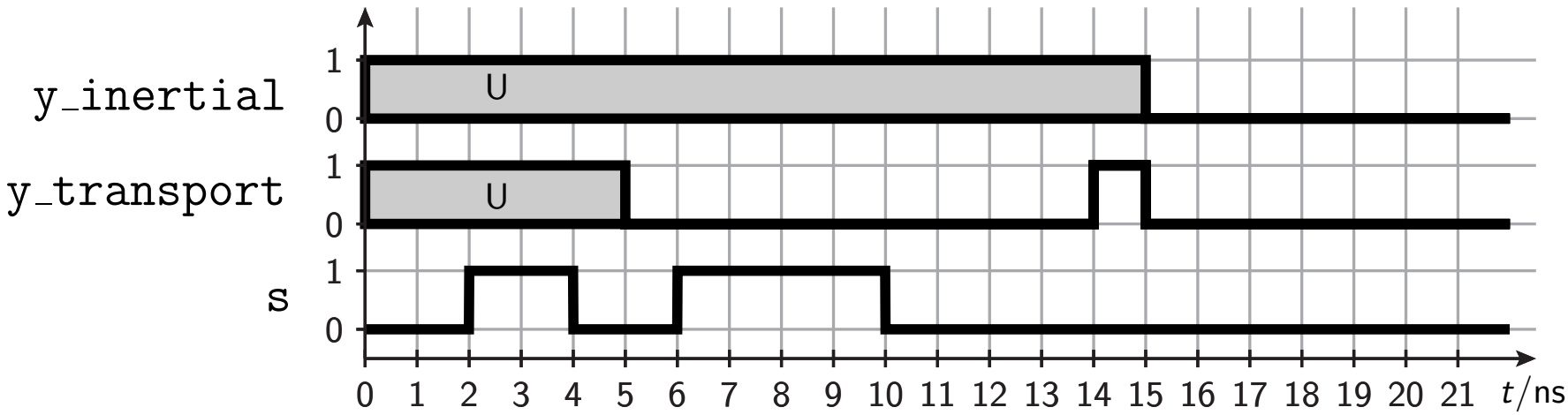
   bei `TRANSPORT` und `INERTIAL` Verzögerungsmodellen.
3. Zeichnen Sie für den Signalverlauf von `s` aus der 1. Aufgabe die Verläufe von `s'DELAYED(1 ns)`, `s'TRANSACTION`, `s'STABLE(2 ns)`, `s'QUIET(3 ns)`, `s'EVENT` und `s'ACTIVE`.

Aufgabenlösungen zur 9. Vorlesung

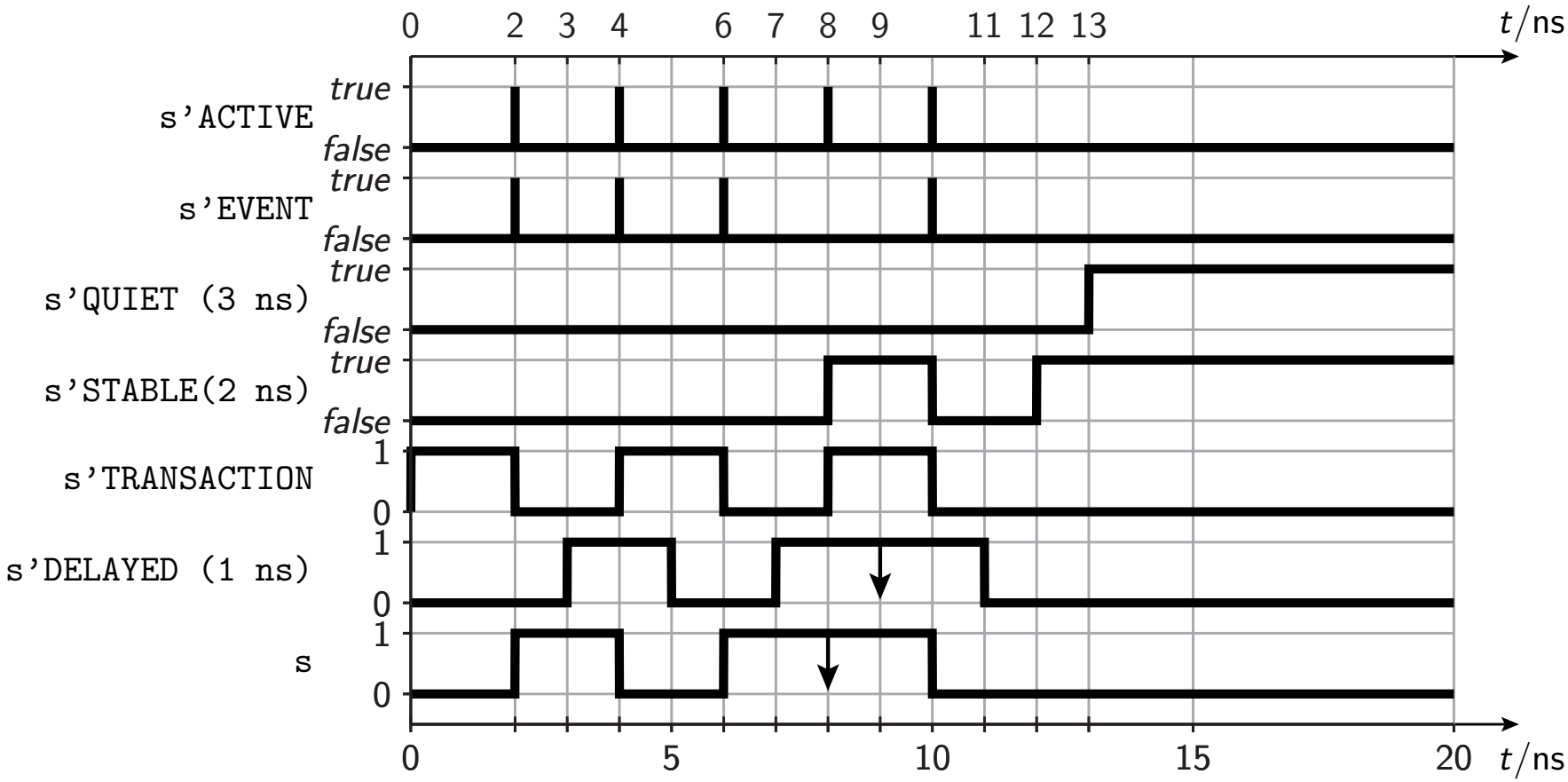
Verzögerungsmodelle (1)



Verzögerungsmodelle (2)



Signalbezogene Attribute



Systementwurf mit VHDL: Vorlesungs-Gesamtübersicht

Systementwurf mit VHDL: Gliederung der 10. Vorlesung

5. Weiterführende Konzepte
   - 5.1. Funktionen und Prozeduren
   - 5.2. Überladung
   - 5.3. Dateien und textuelle Ein-/Ausgabe
   - 5.4. Testumgebungen
   - 5.5. Sonstiges

5. Weiterführende Konzepte

Überblick

Der Sprachumfang reicht weit über die bisher betrachteten Themen hinaus. Einige relevante (jedoch nicht „lebensnotwendige“ Aspekte) sind nachfolgend zusammengefasst. „Weiterführend“ soll hier dementsprechend als „wichtig, aber abdingbar“ und nicht „besonders komplex“ verstanden werden:

- Weitere Strukturierungsmöglichkeiten mit Funktionen und Prozeduren (angelehnt an „gewöhnliche“ Programmiersprachen)
- Mehrfache Verwendung gleicher Operatorenbezeichnungen für Operationen auf verschiedenen Datentypen (Überladung)
- Zugriff auf externe Dateien aus einem VHDL-Modell (insbesondere Ein- und Ausgabe textueller Information)
- Gestaltung und Arten von Testumgebungen
- Sichtbarkeit von Objekten, Zeiger, ...

Arten von Unterprogrammen

In VHDL werden „Unterprogramme“ in zwei Gruppen eingeteilt:

Funktionen sind Unterprogramme mit mindestens einem Argument und genau einem expliziten Rückgabewert (Schlüsselwort `FUNCTION`)

Prozeduren sind Unterprogramme mit beliebig vielen Argumenten und „impliziten Rückgabewerten“ (Schlüsselwort `PROCEDURE`)

	`FUNCTION`	`PROCEDURE`
Argumentenmodi	`IN`	`IN, OUT, INOUT`
Arumentenklassen	Konstanten, Signale	Konstanten, Signale, Variable
Rückgabewerte	exakt einer	beliebig viele (z. B. keine)
Aufruf	in Typkonformen Ausdrücken und Anweisungen	anstelle von sequentiellen und nebenläufigen Anweisungen

Deklaration und Definition von Unterprogrammen

- Deklarationsteil von `ENTITY` und `ARCHITECTURE`
- Deklarationsteil eines Prozesses
- `PACKAGE` (nur Deklaration)
- `PACKAGE BODY` (Deklaration und Definition)

Deklarationsteil einer Funktion:

```
FUNCTION function_name
  [ ( { [CONSTANT bzw. SIGNAL] arg_name_m
    {, arg_name_n} : [IN] arg_type_m [:= def_val_m];}
        [CONSTANT bzw. SIGNAL] arg_name_o
    {, arg_name_p} : [IN] arg_type_o [:= def_val_o])]
  RETURN result_type;
```

Außer Bezeichner `funktion_name` und Rückgabetyp sind alle Angaben optional.

Definitionsteil einer Funktion

```
FUNCTION function_name
-- Wiederholung der Angaben aus der Deklaration
  RETURN result_type IS
-- Zusaetzliche Deklarationsanweisungen
BEGIN
  -- sequentielle Anweisungen
  -- RETURN-Anweisung
  -- keine WAIT-Anweisungen!
END function_name;
```

- Zusätzliche Deklarationsanweisungen können Typen, Untertypen, Konstanten, Variable, Dateien, Aliase und Attribute beinhalten
- Funktionsdefinition darf andere Unterprogrammaufrufe enthalten.

Beispiel einer Funktion

```
FUNCTION implikation (a, b : bit_vector)
 RETURN bit_vector IS
 ALIAS c : bit_vector(0 TO a'LENGTH-1) IS a;
 ALIAS d : bit_vector(0 TO b'LENGTH-1) IS b;
 VARIABLE result : bit_vector(0 TO a'LENGTH-1);
 BEGIN
 ASSERT a'LENGTH = b'LENGTH
 REPORT "Ungleiche Vektorbreite!" SEVERITY failure;
 FOR k IN c'RANGE LOOP
   IF (c(k) = '0' OR d(k) = '1')
     THEN result(k) := '1';
   ELSE result(k) := '0'; END IF;
 END LOOP;
 RETURN result;
END FUNCTION implikation;
```

Funktionsaufruf in einem Modell

PACKAGE mit Deklaraton ist vor ARCHITECTURE einzubinden!

```
ENTITY impl IS
GENERIC (BW : positive := 8);
PORT (in1, in2   : IN  bit_vector(BW-1 DOWNTO 0);
      out1, out2 : OUT bit_vector(BW-1 DOWNTO 0));
END ENTITY impl;

ARCHITECTURE funkt OF impl IS
BEGIN
  out1 <= implikation(in1, in2);
  out2 <= implikation(in1, implikation(in1, in2));
END ARCHITECTURE funkt;
```

Wichtig ⚠

Der Typ der Ports `out1` und `out2` muss mit dem Rückgabetyp der Funktion `implikation()` übereinstimmen.

Deklarationsteil einer Prozedur

```
PROCEDURE procedure_name
 [ ( { [CONSTANT, SIGNAL bzw. VARIABLE] arg_name_m
 {, arg_name_n} : [arg_mode_m] arg_type_m [:= def_val];}
       [CONSTANT, SIGNAL bzw. VARIABLE] arg_name_o
 {, arg_name_p} : [arg_mode_o] arg_type_o [:= def_val])];
```

- Außer Bezeichner `procedure_name` sind alle Angaben optional.
- Wichtiger Unterschied zur Funktion: Keine explizite Spezifikation des Rückgabetyps, da mehrere „Rückgabewerte“ erlaubt sind (Argumente mit Modi `OUT` und `INOUT`, d. h. allgemeine Parameter).

```
PROCEDURE hello_world IS
BEGIN
  ASSERT false REPORT "Hello world!" SEVERITY note;
END hello_world;
```

Definitionsteil einer Prozedur

```
PROCEDURE procedure_name
-- Wiederholung der Angaben aus der Deklaration
  IS
-- Zusaetzliche Deklarationsanweisungen
BEGIN
  -- sequentielle Anweisungen
  -- optional: RETURN-Anweisung
  -- optional: WAIT-Anweisungen
END procedure_name;
```

- Argumente vom Typ `IN` können nur gelesen werden, Argumente vom Typ `OUT` können nur geschrieben werden.
- `RETURN` wird ohne Argument verwendet und bewirkt das sofortige Verlassen der Prozedur.

Beispiel einer Prozedur

```
PROCEDURE d_ff (CONSTANT delay  : IN  time := 1 ns;
                SIGNAL   d, clk : IN  bit;
                SIGNAL   q      : OUT bit) IS
  BEGIN
    IF clk = '1' AND clk'EVENT THEN
       q <= d AFTER delay;
    END IF;
END PROCEDURE d_ff;
```

➥ Verhaltensmodell eines taktflankengesteuerten D-Flipflops

Vorteil: Die Prozedur kann einfach mit den entsprechenden Parametern (Signalen) aufgerufen werden, während bei einer Komponente eine Instanzbildung und Konfiguration notwendig gewesen wären.

Prozeduraufruf in einem Modell

```
ENTITY register_4 IS
  PORT (clk           : IN bit;
        in_d          : IN bit_vector(3 DOWNTO 0);
        out_q         : OUT bit_vector(3 DOWNTO 0));
END ENTITY register_4;

ARCHITECTURE behaviour OF register_4 IS
BEGIN
  ff_3 : d_ff (1.5 ns, in_d(3), clk, out_q(3));
  ff_2 : d_ff (1.5 ns, in_d(2), clk, out_q(2));
  ff_1 : d_ff (1.5 ns, in_d(1), clk, out_q(1));
  ff_0 : d_ff (1.5 ns, in_d(0), clk, out_q(0));
END ARCHITECTURE behaviour;
```

Das Konzept

Überladung *(overloading)*

ist die gleichzeitige Sichtbarkeit mehrerer Bezeichner mit gleichen Namen.

In VHDL können diese Bezeichner

- Unterprogramme (Funktionen und Prozeduren)
- Operatoren
- Werte von Aufzählltypen

sein. Die Überladung ist nur dann möglich, wenn aus dem Aufrufkontext eindeutig ersichtlich ist, welche aus mehreren gleichnamigen Varianten eingesetzt werden soll. Vorteile:

- Vergrößerung von Wertebereichen bei Unterprogrammen und Operatoren
- Erhöhung der Übersichtlichkeit eines Modells

Ein Beispiel zur Überladung von Funktionen

```
FUNCTION maximum (a, b : integer) RETURN integer IS
BEGIN
  IF a >= b THEN RETURN a;
  ELSE           RETURN b;
  END IF;
END maximum;

FUNCTION maximum (a, b : real) RETURN real IS
BEGIN
  IF a >= b THEN RETURN a;
  ELSE           RETURN b;
  END IF;
END maximum;
```

➥ Erzeugt beim Übersetzen keine Fehlermeldung!

Eine weitere Möglichkeit (Überladung von Funktionen)

```
FUNCTION maximum (a, b, c : integer) RETURN integer IS
BEGIN
  IF (a >= b) AND (a >= c)     THEN RETURN a;
  ELSIF (b >= a) AND (b >= c) THEN RETURN b;
  ELSE                                 RETURN c;
  END IF;
END maximum;
```

Die passende Funktion wird anhand von Anzahl der Argumente (letztes Beispiel) bzw. Typen der Argumente (Beispiel von der vorangegangenen Seite) ermittelt und aufgerufen (auch der Rückgabewert kann ein Auswahlkriterium darstellen). Hier wird mit Hilfe der Überladung eine Funktion für Maximumbestimmung unter 2 bzw. 3 Werten vom Typ `integer` bzw. `real` definiert.

Aufruf der überladenen Funktionen

```
ENTITY xyz IS
 -- eine leere ENTITY
END ENTITY xyz;
ARCHITECTURE behavioural OF xyz IS
BEGIN
  ueberladen : PROCESS
    VARIABLE a : real;
    VARIABLE b, c : integer;
  BEGIN
    a := maximum(3.2, 2.1); -- 2. Variante
    b := maximum(0, 1, 3);  -- 3. Variante
    c := maximum(0, 3);     -- 1. Variante
  WAIT;
  END PROCESS ueberladen;
END behavioural;
```

Operatoren

Operatoren sind ein Spezialfall von Funktionen mit einem oder zwei Argumenten, bei denen der Funktionsname eine Zeichenkette ist und beim Aufruf die Argumente dem Namen vor- und nachgestellt werden. Die Überladung funktioniert genauso wie bei den Funktionen:

```
FUNCTION "+" (a, b : bit) RETURN bit IS
BEGIN
  IF (a='0' AND b='0') OR (a='1' AND b='1')
  THEN RETURN '0';
  ELSE RETURN '1';
  END IF;
END "+";

FUNCTION "+" (a : integer; b : real) RETURN real IS
BEGIN
  RETURN real(a)+b;
END "+";
```

Einsatz der überladenen Operatoren

```
ENTITY xyz IS
 -- eine leere ENTITY
END ENTITY xyz;
ARCHITECTURE behavioural OF xyz IS
BEGIN
  ueberladen : PROCESS
    VARIABLE a, b : real;
    VARIABLE c : bit;
BEGIN
  c := '0' + '1'  ;  -- 1. Variante
  a := 2 + 3.3;      -- 2. Variante
  b := "+" (2, 3.3); -- ist auch moeglich
  WAIT;
END PROCESS ueberladen;
END behavioural;
```

Überladung von Werten von Aufzählungstypen

- Gleicher Bezeichner in mehreren Typdefinitionen. Beispiel:

```
TYPE bit IS ('0','1');
TYPE std_ulogic IS ('U','X','0','1','Z','W','L','H','-');
```

- Wenn der Typ aus dem Ausdruck nicht eindeutig erkennbar ist, sollte eine explizite Kennzeichnung in Form von `type_name'(expression)` erfolgen

```
FUNCTION "+" (a : integer; b : real) RETURN integer IS
BEGIN
  RETURN integer(a+integer(b));
END "+";
...
a := integer'(5 + 3.2) + integer'(4 + 3.4);
```

Dateien in VHDL

Typspezifisch: enthalten Objekte eines einzigen Datentyps. Müssen über eine entsprechende Typdefinition deklariert werden:

```
TYPE file_type_name IS FILE OF element_type;
FILE file_name : file_type_name
  [[OPEN file_open_kind_expression] IS string_expression];
```

Textuell: enthalten Objekte verschiedener Datentypen, die als ASCII-Zeichenketten interpretiert werden. Als `file_type_name` wird `text` verwendet (definiert in Package `textio`):

```
USE std.textio.ALL;
FILE file_name : text [[OPEN file_open_kind_expression]
                       IS string_expression];
```

Parameter beim Öffnen von Dateien

- `string_expression`: Physikalischer Name der Datei im Dateisystem
- `file_open_kind_expression`: Zugriffsart

  ```
  TYPE file_open_kind IS (read_mode, write_mode,
                          append_mode);
  ```

  Wie aus der Typdefinition ersichtlich, kann eine Datei zum Lesen (`read_mode`, Standardeinstellung), Schreiben (`write_mode`) oder Anhängen von neuen Daten am Ende (`append_mode`) geöffnet werden.

Beispiele:

```
TYPE integer_file IS FILE OF integer;
FILE idt : integer_file OPEN write_mode IS "int.dat";
FILE text_datei : text IS "text.dat"; -- Lesezugriff
```

Dateizugriff

Wichtig ⚠

Zugriff auf Dateien in Modell erfolgt nur über den logischen Dateinamen `file_name`, der physikalische Name `string_expression` wird nach der Bindung an den entsprechenden logischen Namen nicht mehr im Modell verwendet!

Zugriff auf typspezifische Dateien:

Lesender Zugriff mit `read`:

```
read (file_name, object_name);
read (file_name, object_name, length);
```

`length` liefert bei Objekten vom unbeschränkten Typen (z. B. `bit_vector`) die tatsächlich gelesene Länge

Schreibender Zugriff mit `write`:

```
write (file_name, object_name);
```

Beispiel für Arbeit mit typspezifischen Dateien

```
ENTITY write_byte IS
  PORT (data_8 : IN bit_vector(0 TO 7));
END ENTITY write_byte;

ARCHITECTURE behavior OF write_byte IS
BEGIN
  write_data : PROCESS (data_8)
    TYPE byte_f IS FILE OF bit_vector(0 TO 7);
    FILE output : byte_f OPEN write_mode IS "dt.out";
  BEGIN
    write (output, data_8);
  END PROCESS write_data;
END ARCHITECTURE behavior;
```

Alle Ereignisse auf dem Eingang `data_8` werden in der Datei `dt.out` protokolliert.

Zugriff auf textuelle Dateien

Wichtig ⚠

Vorab `textio`-Package einbinden!

Lesender Zugriff mit `readline` und `read`:

```
readline (file_name, line_object_name);
read (line_object_name, object_name);
```

Schreibender Zugriff mit `writeline` und `write`:

```
writeline (file_name, line_object_name);
write (line_object_name, object_name);
```

Die Funktion `endline(line_objekt_name)` liefert einen `boolean`-Wert zur Überprüfung, ob das Zeilenende erreicht ist (nur textuelle Dateien). Analog kann die Funktion `endfile(file_name)` zum Test des Erreichens vom Dateiende genutzt werden (alle Dateien).

Beispiel für Arbeit mit textuellen Dateien

```
ARCHITECTURE behaviour OF text_test IS
  FILE in_stim : text OPEN read_mode IS "input.dat";
  FILE out_stim : text OPEN write_mode IS "output.dat";
  SIGNAL s, s_out : bit;
BEGIN
  read_data : PROCESS
    VARIABLE l : line;
    VARIABLE t : time;
    VARIABLE data : bit;
  BEGIN
    WHILE (endfile(in_stim) = false) LOOP
      readline (in_stim, l); read (l, data); read (l, t);
      WAIT FOR t; s <= data;
    END LOOP;
    WAIT;
  END PROCESS read_data; ...
```

Beispiel für Arbeit mit textuellen Dateien (fortgesetzt)

```
...
  invert_bit : PROCESS(s)
  BEGIN
    s_out <= NOT s;
  END PROCESS invert_bit;

  write_data : PROCESS (s_out)
    VARIABLE l : line;
  BEGIN
    write (l, s_out);
    write (l, string'(" at time "));
    write (l, now);
    writeline (out_stim, l);
  END PROCESS write_data;
END ARCHITECTURE behaviour;
```

Das Ergebnis nach Simulation

Einlesen der Eingabedaten in Format „`Bitwert Zeit`“, Schreiben der Ausgabedaten im Format „`Bitwert at time Zeit`“:

`input.dat`	`output.dat`
`1 1 ns`	`0 at time 0 ns`
`0 2 ns`	`1 at time 0 ns`
`1 5 ns`	`0 at time 1 ns`
	`1 at time 3 ns`
	`0 at time 8 ns`

Die erste Zeile von `output.dat` stammt vom Initialisierungsdurchlauf. Da der `read_data` Prozess nach dem Einlesen jeder einzelnen Zeile die angegebene Zeitdauer wartet, akkumulieren sich die Zeitangaben: 1+2=3 (ns) sowie 1+2+5=8 (ns).

Sichtbarkeit und Bindung von Dateinamen

Wichtig ⚠

Einem logischen Dateinamen soll immer nur ein physikalischer Dateiname entsprechen!

Häufiger Fehler: Eine Datei in einer Komponente öffnen, die mehrfach instantiiert wird, d. h. physikalischer Name ist gleich (`string_expression`), logische Namen sind unterschiedlich (Zusammensetzung aus Instanzbezeichner und logischem Namen innerhalb der Instanz) ➠ Zugriffsfehler und Inkonsistenzen.
Lösungsalternativen:

1. Die Datei in einem `PACKAGE` deklarieren, das von der Komponente eingebunden wird (alle Instanzen schreiben in eine Datei, Zugriffsfolge wird durch die Simulation bestimmt)
2. Unterschiedliche physikalische Dateinamen (z. B. über Generics) definieren (jede Instanz schreibt in eine eigene Datei)

Sichtbarkeit und Bindung von Dateinamen (fortgesetzt)

Bei dargestellter Vorgehensweise wird die Dateiverwaltung von der Simulationsumgebung übernommen (implizite Dateiverwaltung):

- Ist eine Datei im `PACKAGE`, Deklarationsteil der Architektur oder in einem Prozess definiert, so wird sie beim Simulationsbeginn automatisch geöffnet (gegebenenfalls angelegt) und beim Simulationsende automatisch geschlossen.
- Ist eine Datei im Unterprogramm definiert, so wird sie bei jedem Aufruf dieses Unterprogramms automatisch geöffnet (gegebenenfalls angelegt) und beim Verlassen des Unterprogramms automatisch geschlossen (bei `write_mode` wird der Inhalt jedesmal überschrieben, bei `read_mode` wird immer vom ersten Zeichen an gelesen).

Mehr Flexibilität mit expliziter Dateiverwaltung

```
FILE file_name : file_type_name;
```

Explizites Öffnen mit

```
file_open([file_open_status,] file_name,
          string_expression, file_open_kind);
TYPE file_open_status IS (open_ok, status_error,
                          name_error, mode_error);
```

`open_ok`	Datei erfolgreich geöffnet
`status_error`	Datei ist bereits geöffnet
`name_error`	Bei `read_mode`: die physikalische Datei existiert nicht. Bei `write_mode` und `append_mode`: die physikalische Datei existiert nicht und kann nicht erzeugt werden
`mode_error`	Datei hat Nur-Lesen-Status (`write_mode` und `append_mode`)

Explizites Schließen mit `file_close(file_type);`

Zusammenfassung

- Unterprogramme und Überladung von Bezeichnern
- Arbeiten mit Dateien, Zugriffsmöglichkeiten und Besonderheiten

Aufgabe zum Selbststudium

Schreiben Sie eine weitere `maximum` Funktion, die zwei Bitvektoren beliebiger Breite als Argumente hat und diese beim Vergleich als vorzeichenlose ganze Zahlen interpretiert. Testen Sie Ihre Funktion durch Simulation.

Aufgabenlösungen zur 10. Vorlesung

Maximum-Funktion

Der entsprechende VHDL-Quellkode ist auf dem Handout-Server unter

`VHDL/Aufgabenloesungen/VHDL/vhdl_v10_maximum.vhd`

zu finden.

Systementwurf mit VHDL: Vorlesungs-Gesamtübersicht

1. Einleitung
2. Basiskonzepte
3. Modellierungstechniken
4. Simulation
5. Weiterführende Konzepte
6. Synthese

Systementwurf mit VHDL: Gliederung der 11. Vorlesung

Struktur einer Testumgebung *(test bench)*

Stimulus-Generator: Erzeugung von Eingangssignalen (Eingangs-Testvektoren)

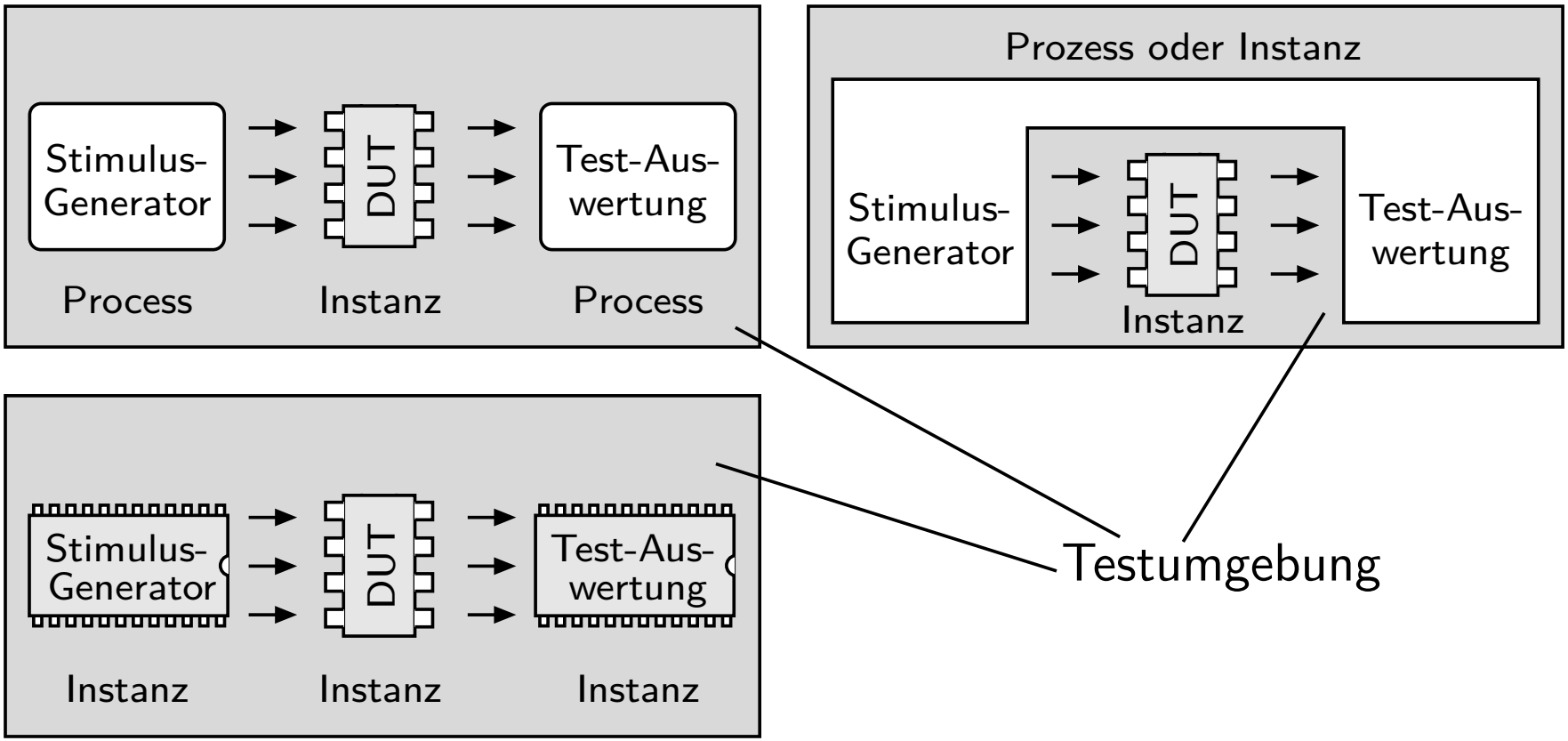
DUT/MUT/UUT *(device under test, model under test, unit under test)*: Das eigentliche Testobjekt

Testauswertung *(response control)*: Vergleich von Sollwerten (Ausgangs-Testvektoren) mit den Ausgangssignalen des Testobjektes

In VHDL können die einzelnen Teile der Testumgebung sowohl in separaten Modellen (Entwurfseinheiten) untergebracht als auch in einem Modell zusammengefasst werden.

Bei einfachen Modellen mit wenigen Testvektoren können die Eingangsstimuli durch direkte Signalzuweisungen innerhalb des Modells erzeugt werden. Bei umfangreicheren Tests empfiehlt sich die Arbeit mit externen Dateien.

Grundsätzliche Strukturierungsmöglichkeiten



Beispiel einer einfachen Testumgebung (DUT)

```
ENTITY dut IS
  PORT (clk : IN std_logic;
        in_a, in_b : IN std_logic;
        out_c : OUT std_logic);
END ENTITY dut;
ARCHITECTURE behaviour OF dut IS
BEGIN
  state_update : PROCESS (clk)
  BEGIN
    IF rising_edge(clk) THEN
      out_c <= in_a AND in_b;
    ELSE
      NULL;
    END IF;
  END PROCESS state_update;
END ARCHITECTURE behaviour;
```

Beispiel einer einfachen Testumgebung (fortgesetzt)

```
ENTITY tb IS
END ENTITY tb;
ARCHITECTURE behavior OF tb IS
 COMPONENT dut PORT (clk, in_a, in_b : IN std_logic;
                     out_c : OUT std_logic);
 END COMPONENT dut;
 SIGNAL clk, s_a, s_b, s_c : std_logic;
 FOR ALL : dut USE ENTITY work.dut(behaviour);
BEGIN
 clk_gen : PROCESS
 BEGIN
  clk <= '0'; WAIT FOR 10 ns; clk <= '1'; WAIT FOR 10 ns;
 END PROCESS clk_gen;
s_a <= '0' AFTER 13 ns, '1' AFTER 24 ns, '0' AFTER 33 ns;
s_b <= '1' AFTER 13 ns, '0' AFTER 24 ns, '1' AFTER 30 ns;
 dut_inst : dut PORT MAP(clk, s_a, s_b, s_c); ...
```

Beispiel einer einfachen Testumgebung (fortgesetzt)

```
...
  check_output : PROCESS
  BEGIN
    WAIT FOR 15 ns;
    ASSERT s_c='1' REPORT "Fehler!" SEVERITY note;
    WAIT FOR 12 ns;
    ASSERT s_c='0' REPORT "Fehler!" SEVERITY note;
    WAIT FOR 12 ns;
    ASSERT s_c='1' REPORT "Fehler!" SEVERITY note;
    WAIT;
  END PROCESS check_output;
END ARCHITECTURE behavior;
```

Hier ohne externen Dateizugriff. DUT ist die Instanz `dut_inst`, Stimulus-Generator besteht aus Prozess `clk_gen` und zwei nebenläufigen Zuweisungen, Testauswertung ist der Prozess `check_output`.

Häufige Fehlerquellen bei der Arbeit mit Testumgebungen

- Missachtung von Verzögerungsmodellen:

```
s_a <= '0' AFTER 13 ns, '1' AFTER 24 ns,
       '0' AFTER 33 ns;
```

  ist **nicht** verhaltensgleich mit

```
s_a <= '0' AFTER 13 ns;
s_a <= '1' AFTER 24 ns;
s_a <= '0' AFTER 33 ns;
```

- Missachtung von Δ-Mechanismus (z. B. Auslesen von Ausgabewerten bevor die zugehörige Eingabebelegung effektiv wirksam geworden ist)
- Synthese von Testumgebungen (es soll ausschließlich DUT synthetisiert werden)

Sichtbarkeit von Objekten

Auf ein Objekt darf innerhalb einer Anweisung nur zugegriffen werden, wenn es an der Stelle des Auftretens dieser Anweisung sichtbar ist. Beispiele:

- Ein im Deklarationsteil einer `ARCHITECTURE` angegebenes Signal ist für alle Prozesse dieser `ARCHITECTURE` sichtbar
- Eine im Unterprogramm deklarierte Variable ist nur in diesem Unterprogramm sichtbar

Besonderheiten treten bei Objekten mit gleichen Namen auf, wenn diese:

- auf verschiedenen hierarchischen Ebenen
- auf einer hierarchischen Ebene

des Modells deklariert sind.

Behandlung von Besonderheiten bei Sichtbarkeit

- Ein Objekt auf einer Hierarchieebene überdeckt (verbirgt) alle Objekte mit gleichem Namen auf höheren Ebenen
- Mehrere Objekte mit gleichem Namen auf einer Hierarchieebene werden nach den Regeln der Überladung behandelt
- Zugriff auf ein nicht sichtbares Objekt oder ein Objekt, das nicht eindeutig zugeordnet werden kann resultiert in einer Fehlermeldung

Der Zugriff auf nicht sichtbare Objekte kann mit Hilfe von *(selected names)* erfolgen:

```
lib_name.pack_name.obj_name_in_pack
lib_name.design_unit_name
record_name.record_element_name
```

Einsatz von *(selected names)*

- Das Format `lib_name.pack_name.obj_name_in_pack` wird benutzt, wenn auf ein Objekt aus einem nicht eingebundenen (mit `USE`-Anweisung) `PACKAGE` zugegriffen werden soll
- Das Format `lib_name.design_unit_name` wird benutzt, wenn auf ein Objekt aus einer anderen Entwurfseinheit innerhalb des gleichen VHDL-Modells zugegriffen werden soll
- Das Format `record_name.record_element_name` wird benutzt, wenn auf ein einzelnes Element aus einem Objekt eines `RECORD`-Typen zugegriffen werden soll

↪ Eine Technik, die gelegentlich nützlich ist, aber nicht zur Standardentwurfspraxis gehören soll

Zeiger

VHDL unterstützt Zeiger als einen speziellen Datentyp:

```
TYPE pointer_type IS ACCESS type_name;
```

Das Schlüsselwort `ACCESS` definiert `pointer_type` als Zeiger-Typ auf den Typ `type_name`. Deklaration von Zeigern:

```
VARIABLE pointer_name : pointer_type
         := NEW type_name;                  -- 1. Variante
VARIABLE pointer_name : pointer_type
         := NEW type_name'(def_value); -- 2. Variante
VARIABLE pointer_name : pointer_type
         := NULL;                           -- 3. Variante
```

1. Variante erzeugt einen Zeiger, reserviert den notwendigen Speicherplatz und weist dem Objekt (das vom Zeiger referenziert wird) den am weitesten links stehenden Wert aus der Deklaration von `type_name` zu (bei der 2. Variante wird `def_value` zugewiesen). 3. Variante legt einen Zeiger an, ohne den Speicherplatz für das Objekt zu reservieren.

Arbeiten mit Zeigern

- Speicherplatz allozieren mit `NEW` (siehe Deklaration)
- Speicherplatz freigeben mit `deallocate(pointer_name)`
- Zeiger dereferenzieren mit `ALL`, z. B. liefert `pointer1.ALL` nicht den Zeigerwert, sondern den Wert des Objektes, das vom Zeiger referenziert wird
- Zeiger können nur als Variable, nicht als Signale deklariert werden, d. h. sie dürfen nur innerhalb von Prozessen und Unterprogrammen auftreten
- Zeiger (genau genommen speicherplatz-bezogene Operationen auf Zeigern) können nicht synthetisiert werden

➥ Einsatz nur für reine Simulationsmodelle und Testumgebungen

Ein Modell mit Zeigern

```
ARCHITECTURE behaviour OF pointer_demo IS
BEGIN
  point1 : PROCESS
    TYPE p_int IS ACCESS integer;
    VARIABLE p1 : p_int := NEW integer'(5);
    VARIABLE p2 : p_int := NEW integer;
    VARIABLE p3 : p_int := NULL;
    VARIABLE i1, i2, i3 : integer;
  BEGIN
    IF p1.ALL = 4 THEN p2.ALL := 3;
    ELSE p2.ALL := 4; END IF;
    p3 := NEW integer;
    p3.ALL := p2.ALL;
    i1 := p1.ALL; i2 := p2.ALL; i3 := p3.ALL;
    deallocate(p1); deallocate(p2); deallocate(p3);
  END PROCESS point1;
END ARCHITECTURE behaviour;
```

Rekursive Datenstrukturen

Bei der Definition der rekursiven Datenstrukturen mit Zeigern ist es erlaubt, einen Typ zunächst nur mit Namen anzugeben (vollständige Deklaration erfolgt dann mit der entsprechenden Datenstruktur):

```
TYPE listenelement;  -- Unvollstaendige Deklaration
TYPE pointer IS ACCESS listenelement;
TYPE listenelement IS RECORD -- Vollstaendige
  nachfolger   : pointer;    -- Deklaration
  listeninhalt : integer;
END RECORD;
```

Zur Erinnerung: Eine unvollständige Typendeklaration ist auch in einem `PACKAGE` erlaubt, wenn die vollständige Deklaration im zugehörigen `PACKAGE BODY` erfolgt.

6. Synthese

Synthese und Sprachumfang

Synthese

ist der Prozess der Überführung einer (V)HDL-Beschreibung in eine schaltungstechnisch direkt realisierbare Form.

Die meisten EDA-Werkzeuge bieten ein integriertes Front-End zur VHDL-Synthese.

Nicht alle Sprachkonstrukte und Beschreibungsformen können synthetisiert werden:

- `AFTER`–Zeitangaben, Initialwerte, `ASSERT` u. ä. werden bei der Synthese ignoriert (evtl. mit Warnhinweis)
- Zeiger, Dateizugriffe, Funktion `NOW` u. ä. können nicht synthetisiert werden (Fehlermeldung)
- ...

Da es für jede Aufgabenstellung mehrere Lösungsvarianten gibt, gilt bei der VHDL-Synthese: *What you write is what you get!*

Verbindung synthesefähiger und nicht synthesefähiger Anweisungen in einem Modell

VHDL-Simulationsmodelle enthalten oft Anweisungen, die nicht synthetisiert werden können. Ein synthesefähiges Modell muss von solchen Anweisungen frei sein. Methoden zur Verbindung von synthetisierbaren und nicht-synthetisierbaren Code-Abschnitten in einem Entwurf:

- Erzeugung verschiedener Architekturen (z. B. eine für die Simulation und eine für die Synthese), Zuordnung zur `ENTITY` mittels `CONFIGURATION` oder (bei Implementierung von Architekturen in separaten Dateien) durch Weglassen des Simulationsmodells bei der Synthese
- Einfügen von „Pragmas“ (Anweisungen für das Synthese-Frontend), die nicht-synthesefähige Abschnitte ausblenden, z. B. `-- rtl_synthesis on` und `-- rtl_synthesis off` (IEEE RTL-Synthese-Standard)

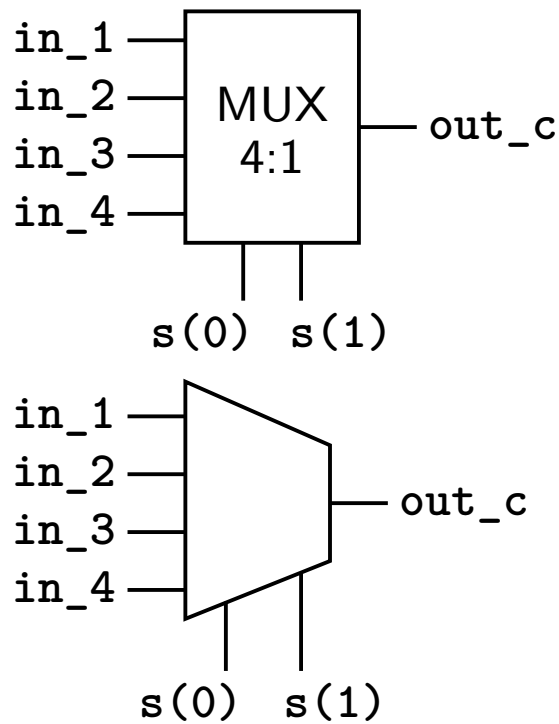
Synthesewerkzeuge und Standards

- IEEE 1076.6-1999, *IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis*
  - spezifiziert einen Minimalumfang an synthetisierbaren Sprachmitteln, die von allen Werkzeugen unterstützt werden sollten
  - orientiert sich an VHDL'87 (!)
- IEEE 1076.6-2004, eine Revision des Synthese-Standards
  - spezifiziert einen Maximalumfang an synthetisierbaren Sprachmitteln, die unterstützt werden können
  - orientiert sich an VHDL'93 (da mit VHDL-2000 und VHDL-2002 keine synthesefähigen Erweiterungen eingeführt wurden, gilt der Standard effektiv für VHDL-2002)

➥ Konkrete Synthese-Möglichkeiten sind werkzeugabhängig

Einführendes Beispiel

```
ENTITY mux4_to_1 IS
  PORT (in_1, in_2, in_3, in_4 : IN bit;
        s : IN bit_vector(1 DOWNTO 0);
        out_c       : OUT bit);
END ENTITY mux4_to_1;

ARCHITECTURE funkt OF mux4_to_1 IS
BEGIN
  WITH s SELECT
  out_c <= in_1 WHEN "00",
           in_2 WHEN "01",
           in_3 WHEN "10",
           in_4 WHEN "11";
END ARCHITECTURE funkt;
```



Einfache vs. komplexe Schaltnetze

Zur Synthese einfacher Schaltnetze eignen sich funktionale Beschreibungen:

- Schnittstelle (Außenanschlüsse) werden in der `ENTITY` beschrieben
- `ARCHITECTURE` beinhaltet die Schaltnetzbeschreibung:
  - Schaltalgebraische Gleichungen mit den Operatoren `NOT`, `AND`, `NAND`, `OR`, `NOR`, `XOR`, `XNOR`
  - Multiplexer-Logik mit *conditional* und *selected assignment*

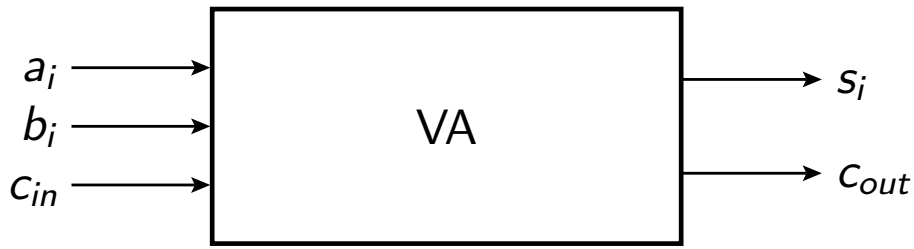
Bei komplexeren Schaltnetzen kann zur besseren Übersichtlichkeit auf strukturale Beschreibungen zurückgegriffen werden, z. B.

- Volladdierer (VA) als ein Basisblock (funktionale Beschreibung)
- Carry-Save-Addierer (CSA) als strukturales Modell, bestehend aus mehreren VA
- Multiplizierer als strukturales Modell, bestehend aus mehreren CSA und Zusatzlogik

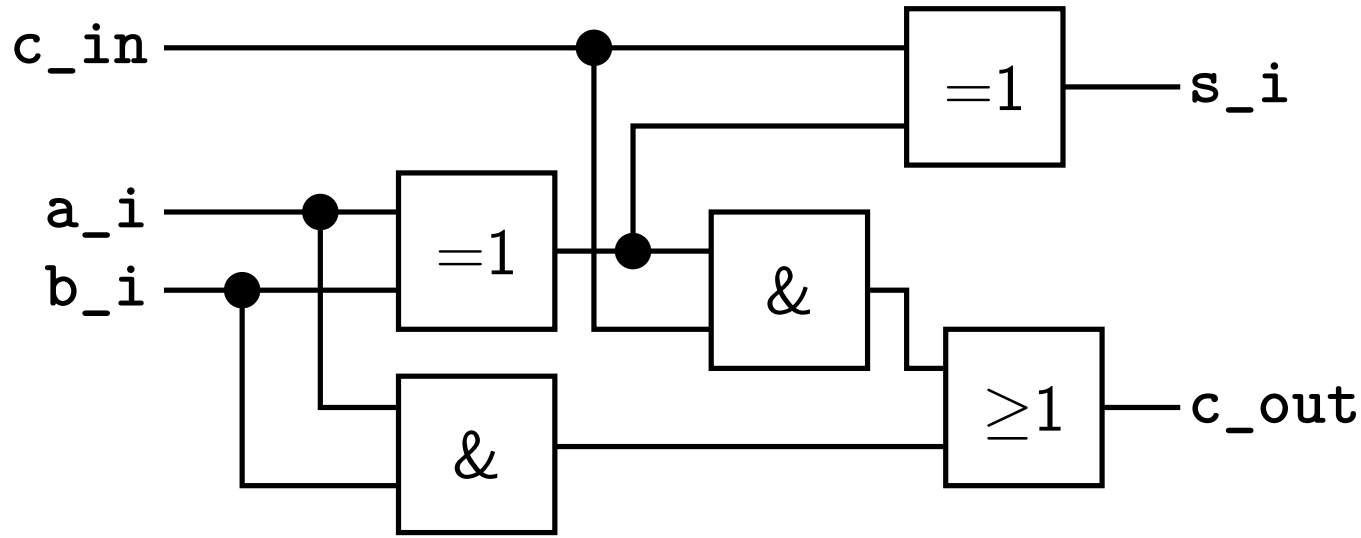
Volladdierer als funktionales VHDL-Modell

```
ENTITY vadd IS
  PORT (a_i, b_i, c_in : IN bit;
        s_i, c_out     : OUT bit);
END ENTITY vadd;
ARCHITECTURE funkt OF vadd IS
BEGIN
  s_i <= a_i XOR b_i XOR c_in;
  c_out <= (a_i AND b_i) OR (c_in AND (a_i XOR b_i));
END ARCHITECTURE funkt;
```

a_i	b_i	c_{in}	s_i	c_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



Volladdierer nach der Synthese



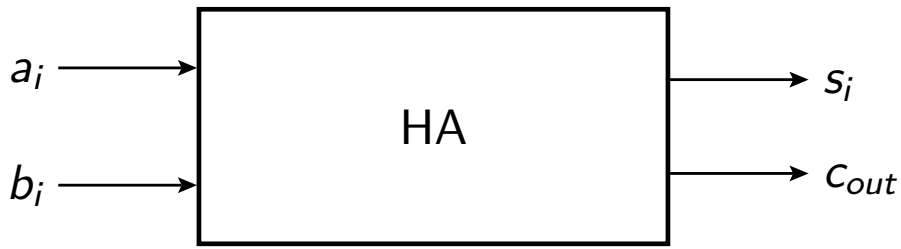

```
s_i <= a_i XOR b_i XOR c_in;
c_out <= (a_i AND b_i) OR (c_in AND (a_i XOR b_i));
```

Die synthetisierte Schaltung spiegelt die parallele Struktur des Volladdierers wieder: Beide Signalzuweisungen sind nebenläufig und werden bei der Synthese auch so interpretiert (siehe 6. Vorlesung)!

Halbaddierer als funktionales VHDL-Modell

```
ENTITY hadd IS
  PORT (a_i, b_i   : IN bit;
        s_i, c_out : OUT bit);
END ENTITY hadd;

ARCHITECTURE funkt OF hadd IS
BEGIN
  s_i <= a_i XOR b_i;
  c_out <= a_i AND b_i;
END ARCHITECTURE funkt;
```

a_i	b_i	s_i	c_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



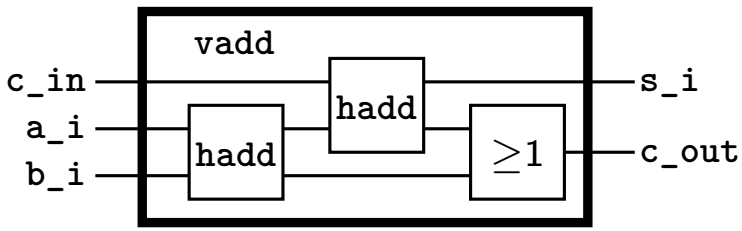
Volladdierer als synthetisierbares Struktur-Modell

```
ENTITY vadd IS
  PORT (a_i, b_i, c_in : IN bit;
        s_i, c_out     : OUT bit);
END ENTITY vadd;

ARCHITECTURE structure OF vadd IS
  COMPONENT hadd
    PORT(a_i, b_i : IN bit;
         s_i, c_out     : OUT bit);
  END COMPONENT;
  SIGNAL s_1, c_1, c_2 : bit;
BEGIN
  ha_1 : hadd PORT MAP(a_i, b_i, s_1, c_1);
  ha_2 : hadd PORT MAP(c_in, s_1, s_i, c_2);
  c_out <= c_1 OR c_2;
END ARCHITECTURE structure;
```

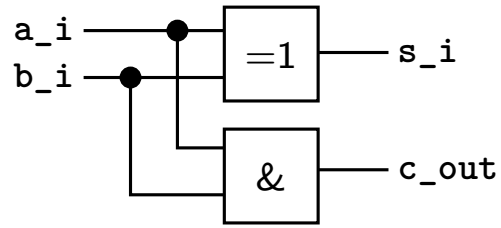
Volladdierer nach der Synthese

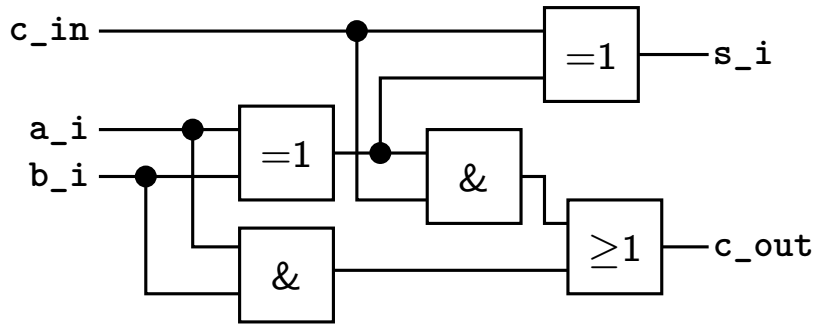
Modellstruktur

```
ENTITY hadd IS
...
END ENTITY hadd;
ARCHITECTURE funkt OF hadd IS
...
END ARCHITECTURE funkt;
```



Syntheseergebnis





Anmerkung: High-Level-Synthese berücksichtigt noch nicht die Zieltechnologie (z. B. Verfügbarkeit von XOR-, AND- und OR-Gattern). Die Umsetzung der synthetisierten Logik in plattformspezifische Primitive erfolgt erst später bei der Technologie-Abbildung *(technology mapping)*.

Synthese BOOLEscher Operatoren

- Auswertung erfolgt linksassoziativ
- `NOT` hat die höchste Priorität
- `AND`, `OR` und `XOR` sind assoziativ und können daher bei mehrfacher Verwendung in einer Zuweisung ohne Klammern erscheinen
- `NAND`, `NOR` und `XNOR` sind **nicht** assoziativ und müssen daher bei mehrfacher Verwendung in einer Zuweisung mit Klammerung versehen werden (sonst Fehlermeldung)

```
y <= NOT (a AND b AND c); -- NAND mit 3 Eingaengen
y <= (a NAND b) NAND c;   -- kein NAND mit 3 Eingaengen
y <= a NAND b NAND c;     -- Fehler!
y <= a XOR b XOR c;       -- XOR mit 3 Eingaengen
y <= a XNOR b XNOR c;     -- evtl. Fehler
                          -- (Tool-abhaengig)
```

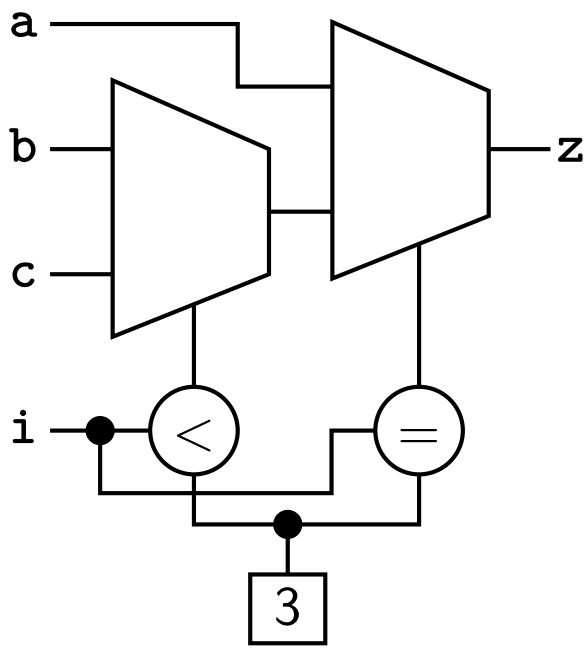
Komplexere Schaltnetze werden mit Prozessen modelliert

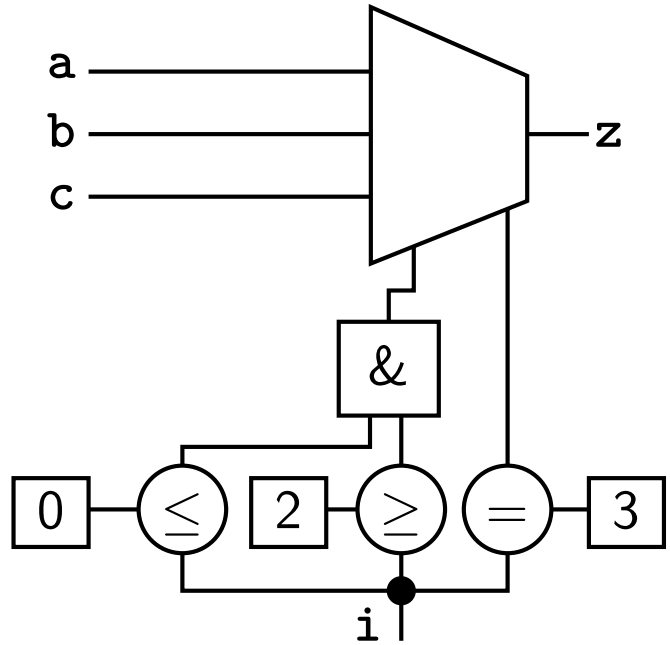
```
ENTITY netz_1 IS
  PORT (i       : IN  integer RANGE 0 TO 9;
        a, b, c : IN  std_logic_vector(7 DOWNTO 0);
        z       : OUT std_logic_vector(7 DOWNTO 0));
END ENTITY netz_1;

ARCHITECTURE behaviour OF netz_1 IS
BEGIN
  p1 : PROCESS (i,a,b,c)
  BEGIN
    IF (i=3) THEN    z <= a;
    ELSIF (i<3) THEN z <= b;
    ELSE             z <= c;
    END IF;
  END PROCESS p1;
END ARCHITECTURE behaviour;
```

Andere funktional äquivalente Beschreibung von netz_1

```
ENTITY netz_2 IS
  PORT (i       : IN  integer RANGE 0 TO 9;
        a, b, c : IN  std_logic_vector(7 DOWNTO 0);
        z       : OUT std_logic_vector(7 DOWNTO 0));
END ENTITY netz_2;
ARCHITECTURE behaviour OF netz_2 IS
BEGIN
  p1 : PROCESS (i,a,b,c)
  BEGIN
    CASE i IS
      WHEN 3      => z <= a;
      WHEN 0 TO 2 => z <= b;
      WHEN OTHERS => z <= c;
    END CASE;
  END PROCESS p1;
END ARCHITECTURE behaviour;
```

Syntheseergebnisse für `netz_1` und `netz_2`





Der Unterschied besteht darin, dass bei `IF`-Struktur der Hauptzweig der `IF`-Bedingung zuerst ausgewertet wird, während bei `CASE` alle Zweige „gleichberechtigt“ sind. Anmerkung: `integer` als Datentyp für E/A-Ports sollte vermieden werden (hier nur aus Platzgründen als Ausnahme). Stattdessen: IEEE-Packages `std_logic_{unsigned, signed, arith}` (nur eines davon in einem Modell!)

IF- und CASE-Strukturen bei der Synthese

- IF-THEN-ELSE-Konstrukt beinhaltet eine implizite Priorisierung, da die Auswertung der Bedingungen in der Reihenfolge der Abarbeitung erfolgt: IF ➠ ELSIF ➠ ELSE. Die Signallaufzeiten sind unterschiedlich:
  - ☺ Kann zu Optimierungszwecken verwendet werden, z. B. kürzester Pfad zuerst
  - ☹ Resultiert bei vielen verschachtelten Konstrukten in langen (= langsamen) kombinatorischen Pfaden, verschlechtert das Zeitverhalten
- CASE-Konstrukt entspricht grundsätzlich einer Auswahl aus vollkommen gleichberechtigten Varianten (Multiplexer, Decoder), d. h. die entsprechenden Signallaufpfade in der synthetisierten Schaltung sind gleich lang

Zusammenfassung

- Erzeugen von Testumgebungen
- Sichtbarkeit von Objekten
- Zeiger
- Synthesegerechte Beschreibung von Schaltnetzen

Aufgaben zum Selbststudium

1. Entwickeln Sie eine Testumgebung zum Testen eines 1-aus-16 Dekoders, wobei die Stimuli aus einer Eingabedatei eingelesen werden und die Testergebnisse in eine Ausgabedatei geschrieben werden.
2. Erzeugen Sie eine einfach verkettete Liste, die als Elemente `integer`-Werte enthält. Diese sollen aus einer Datei eingelesen werden (die Dateigröße ist unbekannt und kann variieren).
3. Erzeugen Sie eine synthesegerechte Beschreibung eines Matrixmultiplizierers (siehe „Digitaltechnik 1“) mit einstellbarer Operandenbreite. Simulieren und synthetisieren Sie diese. Vergleichen sie das Ergebnis mit der Synthese einer Multiplikation aus einem einfachen $*$-Operator.

Aufgabenlösungen zur 11. Vorlesung

Testumgebungen, Zeiger und Schaltnetzsynthese

Der entsprechende VHDL-Quellkode ist auf dem Handout-Server unter

`VHDL/Aufgabenloesungen/VHDL/vhdl_v11_dec_test.vhd`,
`VHDL/Aufgabenloesungen/VHDL/vhdl_v11_liste.vhd`

und

`VHDL/Aufgabenloesungen/VHDL/vhdl_v11_matrix_mult.vhd`

zu finden.

Die Datei `input1_aus_16.dat` enthält Teststimuli für den Dekoder und muss in das entsprechende Verzeichnis kopiert werden (dieses muss bei der Deklaration angegeben werden).

Die Datei `int_input.dat` enthält `integer`-Werte, aus denen die Liste bestehen soll und muss in das entsprechende Verzeichnis kopiert werden (dieses muss bei der Deklaration angegeben werden).

Systementwurf mit VHDL: Vorlesungs-Gesamtübersicht

1. Einleitung
2. Basiskonzepte
3. Modellierungstechniken
4. Simulation
5. Weiterführende Konzepte
6. Synthese

Systementwurf mit VHDL: Gliederung der 12. Vorlesung

Prozesse bei Simulation und Synthese

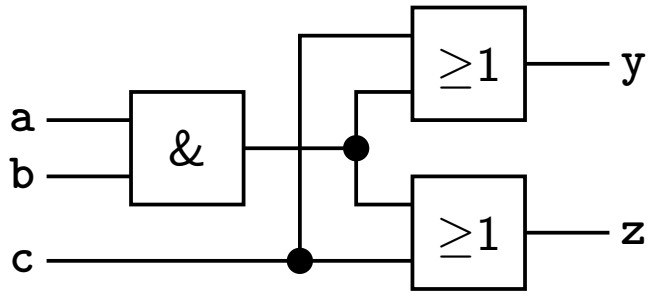
- Prozesse werden als nebenläufige Anweisungen behandelt (zwei unabhängige Prozesse in einer Architektur erzeugen zwei „parallele“ Schaltstrukturen)
- Bei der Verwendung einer Empfindlichkeitsliste zur Modellierung kombinatorischer Logik sollten
  - alle Signale auf der rechten Seite einer Zuweisung
  - alle anderen Signale, deren Wert abgefragt (gelesen) wird, z. B. in einer `IF`-Bedingung

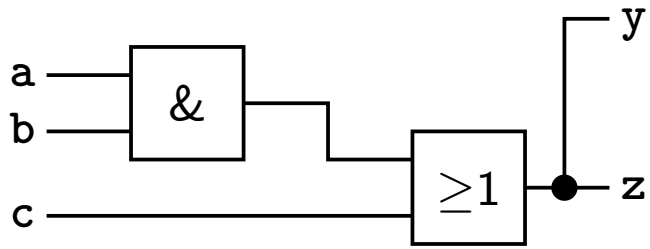
  in die Liste aufgenommen werden, damit die Simulationsergebnisse mit dem Verhalten der synthetisierten Schaltung übereinstimmen (in VHDL-2008 kann die Empfindlichkeitsliste mit `ALL`-Anweisung beschrieben werden, die das Einhalten von diesen Regeln erzwingt)

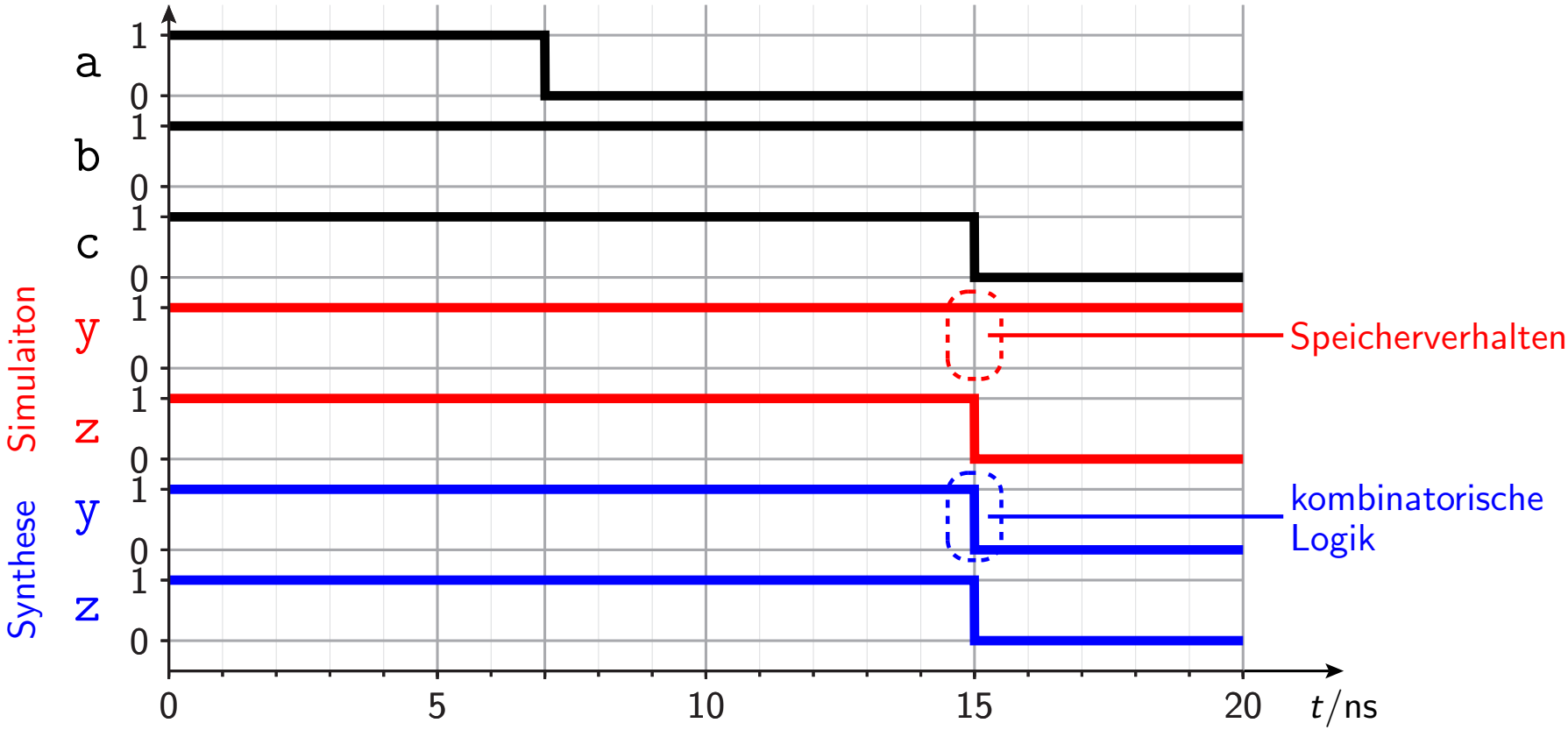
Beispiel einer unvollständigen Empfindlichkeitsliste

```
ENTITY sensitiv IS
  PORT (a, b, c : IN  bit;
        y, z    : OUT bit);
END ENTITY sensitiv;
ARCHITECTURE behaviour OF sensitiv IS
BEGIN
  p1 : PROCESS (a,b) -- unvollstaendig
  BEGIN
    y <= (a AND b) OR c;
  END PROCESS p1;
  p2 : PROCESS (a,b,c) -- vollstaendig
  BEGIN
    z <= (a AND b) OR c;
  END PROCESS p2;
END ARCHITECTURE behaviour;
```

Das `sensitiv`-Modell bei der Simulation und Synthese



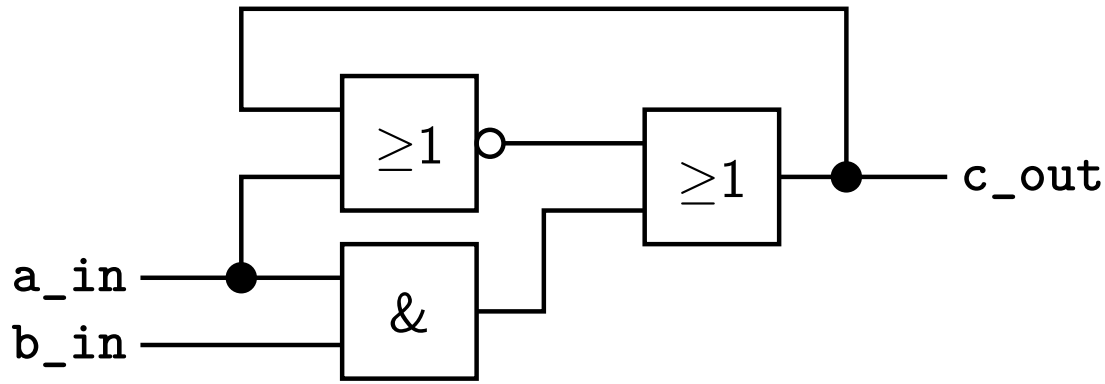




Häufige Fehlerquelle: Kombinatorische Schleifen

```
ENTITY comb_loop IS
  PORT (a_in, b_in : IN  bit;
        c_out      : OUT bit);
END ENTITY comb_loop;

ARCHITECTURE behaviour OF comb_loop IS
BEGIN
  p1 : PROCESS (a_in, b_in)
  VARIABLE temp : bit;
  BEGIN
    IF (a_in = '1') THEN temp := b_in;
    ELSE temp := NOT temp;
    END IF;
    c_out <= temp;
  END PROCESS p1;
END ARCHITECTURE behaviour;
```

Kombinatorische Schleife als Syntheseergebnis



Das Problem: Gilt im Initialzustand `a_in=0`, so wird der `ELSE`-Zweig aktiv, d. h. die Variable `temp` wird gelesen, ohne dass sie vorher einen Wert zugewiesen bekam. Der Simulator entnimmt den Wert aus der Ereignisliste (z. B. '0' als Standardwert). Das Synthesewerkzeug entgegen erzeugt eine Verbindung zwischen dem Ausgang `c_out` („alter Wert“ von `temp`) und dem benötigten Eingang für `NOT` (als NOR-Gatter implementiert).

- ➥ Kombinatorische Schleife, Speicher- und eventuell Schwingverhalten!

Umgang mit *don't cares*

Bei der Beschreibung der Ausgangswerte wie üblich, das Ergebnis ist abhängig vom verwendeten Synthese-Werkzeug

Bei der Beschreibung der Eingangswerte wird der entsprechende Wert nicht als „`0 oder 1`“, sondern als ein Logikwert „–“ interpretiert und kann für die Synthese nicht verwendet werden! Ein Beispiel (der Typ vom Port `Eingang` ist `std_logic_vector`):

```
CASE Eingang IS
  WHEN "000" => Ausgang <= "010";
  WHEN "001" => Ausgang <= "001";
  WHEN "10-" => Ausgang <= "000";
  WHEN OTHERS => Ausgang <= "011";
END CASE;
```

➥ Es gilt `Ausgang = "011"` sowohl bei `Eingang = "100"` als auch bei `Eingang = "101"`!

don't cares in VHDL 2008

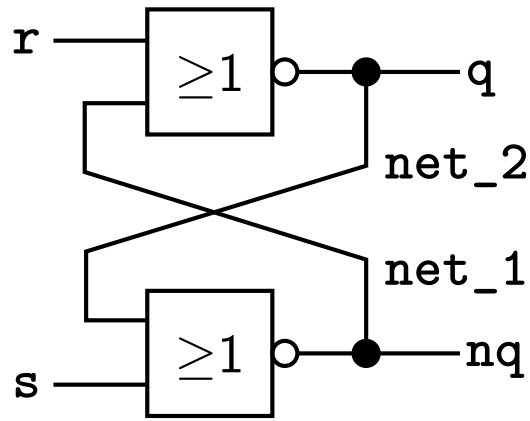
Mit dem *matched case* Konstrukt „case?" ist es möglich, *don't cares* auch als Eingangsgrößen zu verwenden. Im folgenden Kodeabschnitt

```
CASE? Eingang IS
  WHEN "1--" => Ausgang <= "010";
  WHEN "01-" => Ausgang <= "001";
  WHEN "001" => Ausgang <= "000";
  WHEN OTHERS => Ausgang <= "011";
END CASE?;
```

steht die erste `WHEN`-Zeile tatsächlich für `"100"`, `"101"`, `"110"`, `"H00"` usw. Dann ist allerdings die Zuweisung eines *don't care* an den `Ausgang` im gleichen Zuge nicht erlaubt.

Ein einfaches Speicherelement (asynchrones RS-Flipflop)

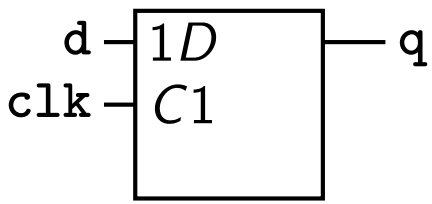
```
ENTITY rs_ff IS
  PORT (r, s  : IN bit;
        q, nq : OUT bit);
END ENTITY rs_ff;
ARCHITECTURE behavior OF rs_ff IS
  SIGNAL net_1, net_2 : bit;
BEGIN
  net_1 <= s NOR net_2;
  net_2 <= r NOR net_1;
  q     <= net_2;
  nq    <= net_1;
END ARCHITECTURE behavior;
```



- Erzeugt kombinatorische Schleife. Aufgrund von Fehlerempfindlichkeit und Problemen bei der Timing-Analyse soll man bevorzugt synchrone Speicherelemente einsetzen!

Eintaktzustandsgesteuertes Speicherelement (1TZS D-FF, D-Latch)

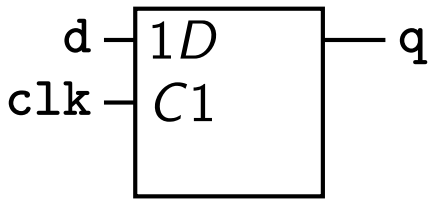
```
ENTITY d_latch IS
  PORT (d, clk : IN bit;
        q      : BUFFER bit);
END ENTITY d_latch;

ARCHITECTURE functional1 OF d_latch IS
BEGIN
  q <= d WHEN clk = '1' ELSE q;
END ARCHITECTURE functional1;
ARCHITECTURE functional2 OF d_latch IS
BEGIN
  WITH clk SELECT
    q <= d WHEN '1',
         q WHEN OTHERS;
END ARCHITECTURE functional2;
```



Beschreibung ohne BUFFER

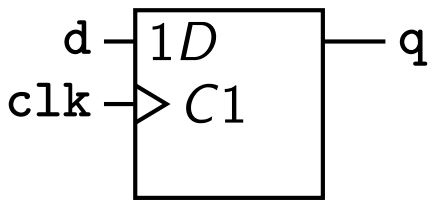
Alternativ (um BUFFER-Port zu vermeiden) kann auf eine Verhaltensbeschreibung zurückgegriffen werden:

```
ENTITY d_latch IS
  PORT (d, clk : IN bit;
        q      : OUT bit);
END ENTITY d_latch;

ARCHITECTURE behaviour OF d_latch IS
BEGIN
  d_latch_proc : PROCESS (clk, d) IS
  BEGIN
    IF clk = '1' THEN
      q <= d;
    END IF;
  END PROCESS d_latch_proc;
END ARCHITECTURE behaviour;
```



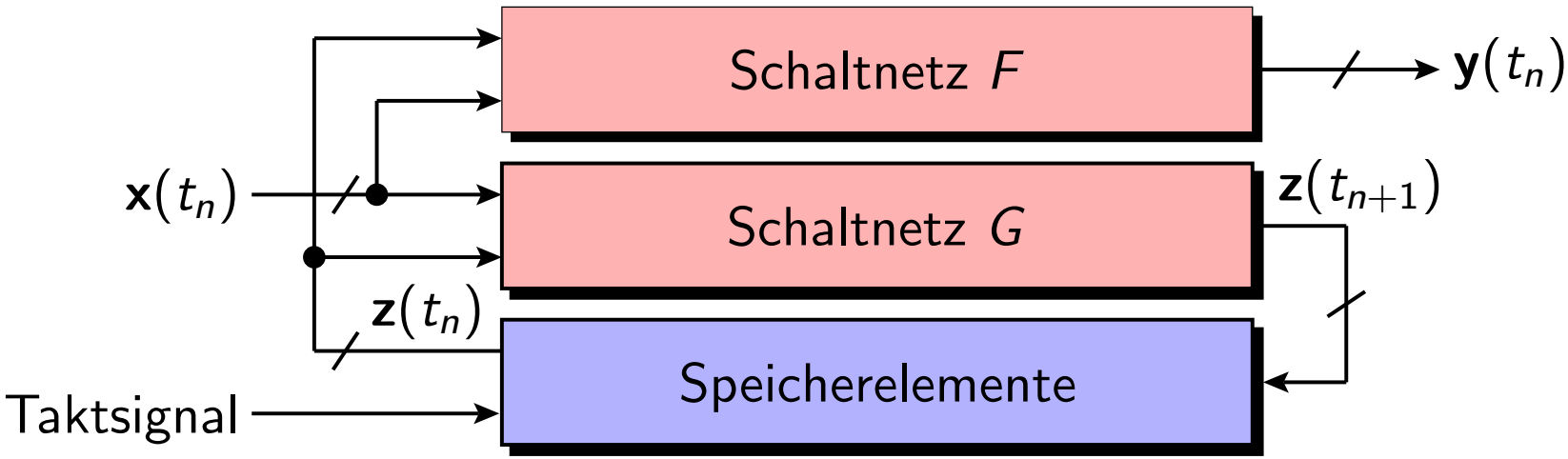
Eintaktflankengesteuertes Speicherelement (1TFS D-FF)

```
ENTITY d_ff IS
  PORT (d, clk : IN bit;
        q      : OUT bit);
END ENTITY d_ff;

ARCHITECTURE behaviour OF d_ff IS
BEGIN
  d_ff_proc : PROCESS (clk) IS
  BEGIN
    IF clk'event AND clk = '1' THEN
      q <= d;
    END IF;
  END PROCESS d_ff_proc;
END ARCHITECTURE behaviour;
```



Bei allen vorgestellten FF-Modellen sollte ein globales RESET-Signal hinzugefügt werden (hier aus Platzgründen weggelassen).

Allgemeine Struktur eines Schaltwerks



Was muss modelliert werden?

- Übergangsfunktion g (Schaltnetz G)
- Ausgabefunktion f (Schaltnetz F)
- Speicherelemente

Modellierungsansätze

Drei-Prozess-Beschreibung: Übergangsfunktion, Ausgabefunktion und Speicherelemente mit je einem eigenen Prozess

- ☺ Strukturiert, gut lesbar
- ☹ langsamere Simulation

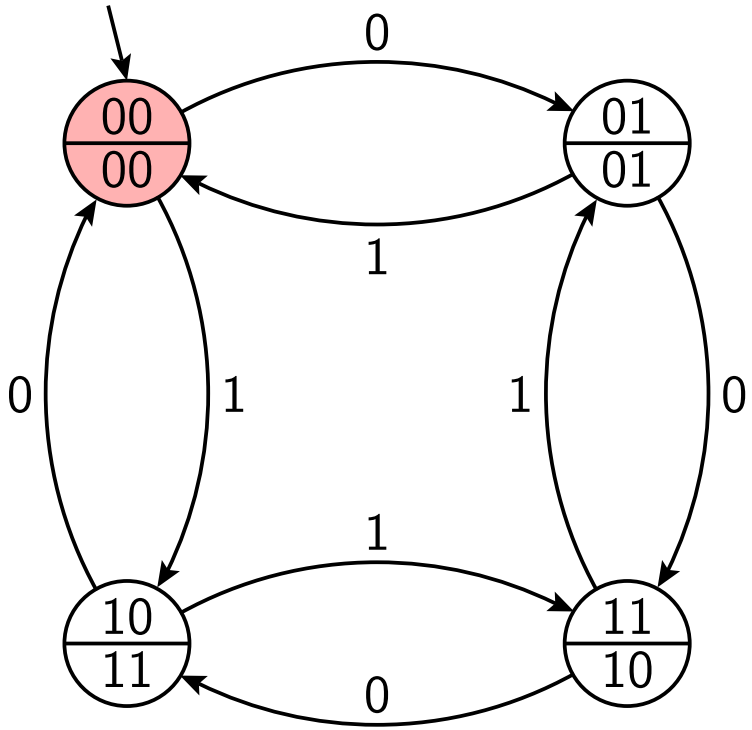
Ein-Prozess-Beschreibung: Übergangsfunktion, Ausgabefunktion und Speicherelemente zusammen in einem Prozess

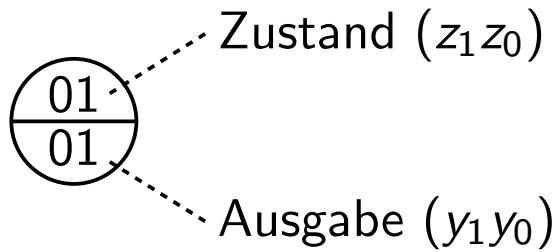
- ☺ schnelle Simulation
- ☹ unübersichtlich, nicht konform mit RTL-Modell

Zwei-Prozess-Beschreibung: Übergangsfunktion und Ausgabefunktion zusammen in einem Prozess, Speicherelemente in einem anderen Prozess

- ☺☹ Kompromiss zwischen Ein- und Drei-Prozessbeschreibung

Ein Beispielschaltwerk






```
ENTITY fsm_1 IS
  PORT (x, clk, res : IN bit;
        y           : OUT bit_vector(1 DOWNTO 0));
END ENTITY fsm_1;
```

Drei-Prozess-Beschreibung (1. Prozess, Speicherelemente)

```
ARCHITECTURE three_proc OF fsm_1 IS
  SIGNAL state, next_state : bit_vector(1 DOWNTO 0);
BEGIN
  store : PROCESS(clk, res) -- Speicherelemente
  BEGIN
    IF (res = '1') THEN state <= "00";
    ELSIF (clk'event AND clk = '1') THEN
      state <= next_state;
    END IF;
  END PROCESS store;
...
```

Drei-Prozess-Beschreibung (2. Prozess, Übergangsfunktion)

```
transition : PROCESS(x, state) -- Uebergangsfunktion
BEGIN
  CASE state IS
    WHEN "00" => IF  (x='0') THEN next_state <= "01";
                 ELSE next_state <= "10"; END IF;
    WHEN "01" => IF  (x='0') THEN next_state <= "11";
                 ELSE next_state <= "00"; END IF;
    WHEN "10" => IF  (x='0') THEN next_state <= "00";
                 ELSE next_state <= "11"; END IF;
    WHEN "11" => IF  (x='0') THEN next_state <= "10";
                 ELSE next_state <= "01"; END IF;
    WHEN OTHERS => next_state <= "00";
  END CASE;
END PROCESS transition;
```

Drei-Prozess-Beschreibung (3. Prozess, Ausgabefunktion)

```
  output : PROCESS(state) -- Ausgabefunktion
  BEGIN
    CASE state IS
      WHEN "00" => y <= "00";
      WHEN "01" => y <= "01";
      WHEN "10" => y <= "11";
      WHEN "11" => y <= "10";
      WHEN OTHERS => y <= "00";
    END CASE;
  END PROCESS output;
END ARCHITECTURE three_proc;
```

Oder alternativ mit zwei nebenläufigen Zuweisungen:

```
y(0) <= state(0) XOR state(1);
y(1) <= state(1);
```

Default-Zuweisungen

Eine sinnvolle Erweiterung (insbesondere bei partiell spezifizierten Zustandsräumen) kann die Hinzunahme einer Default-Zuweisung von Zustands- und Ausgabewerten vor der CASE-Anweisung sein:

```
transition : PROCESS(x, state) -- Uebergangsfunktion
BEGIN
  next_state <= "00"; -- <- Default-Wert
  CASE state IS
    WHEN "00" => ...
    WHEN OTHERS => NULL; ...
output : PROCESS(state) -- Ausgabefunktion
BEGIN
  y <= "00"; -- <- Default-Wert
  CASE state IS
    WHEN "00" => y <= "00"; ...
    WHEN OTHERS => NULL; ...
```

Zwei-Prozess-Beschreibung (2. Prozess)

Speicherelemente ähnlich wie bei Drei-Prozess-Beschreibung.

```
ARCHITECTURE two_proc OF fsm_1 IS
...
 trans_out : PROCESS(x, state) -- Ausgabe und
  BEGIN                        -- Uebergangsfunktion
  CASE state IS
   WHEN "00" => y <= "00"; IF (x='0') THEN next_state <= "01";
    ELSE next_state <= "10"; END IF;
   WHEN "01" => y <= "01"; IF (x='0') THEN next_state <= "11";
    ELSE next_state <= "00"; END IF;
   WHEN "10" => y <= "11"; IF (x='0') THEN next_state <= "00";
    ELSE next_state <= "11"; END IF;
   WHEN "11" => y <= "10"; IF (x='0') THEN next_state <= "10";
    ELSE next_state <= "01";  END IF;
   WHEN OTHERS => next_state <= "00";
  END CASE;
 END PROCESS trans_out;
END ARCHITECTURE two_proc;
```

Ein-Prozess-Beschreibung (Ausgabe einen Takt später!)

```
ARCHITECTURE one_proc OF fsm_1 IS ...
  fsm : PROCESS(clk, res) -- Ausgabe und Uebergangsfunktion
  BEGIN
    IF (res = '1') THEN state <= "00"; y <= "00";
    ELSIF (clk'event AND clk = '1') THEN
     CASE state IS
       WHEN "00" => y <= "00"; IF (x='0') THEN state <= "01";
         ELSE state <= "10"; END IF;
       WHEN "01" => y <= "01"; IF (x='0') THEN state <= "11";
         ELSE state <= "00"; END IF;
       WHEN "10" => y <= "11"; IF (x='0') THEN state <= "00";
         ELSE state <= "11"; END IF;
       WHEN "11" => y <= "10"; IF (x='0') THEN state <= "10";
         ELSE state <= "01";  END IF;
       WHEN OTHERS => state <= "00";
     END CASE;
    END IF;
  END PROCESS fsm;
END ARCHITECTURE one_proc;
```

Ein-Prozess-Beschreibung ohne Ausgabeverzögerung

Bei der Ein-Prozess-Beschreibung sind alle Signale in den „getakteten“ Prozess eingebettet: Der Ausgang wird automatisch durch zusätzliche FFs synchronisiert, die Ausgabe wird um einen Taktzyklus verzögert. Vermeiden durch Anpassung der Ausgangsfunktion:

```
CASE state IS
  WHEN "00" => IF  (x='0') THEN state <= "01"; y <= "01";
    ELSE state <= "10"; y <= "11"; END IF;
  WHEN "01" => IF  (x='0') THEN state <= "11"; y <= "10";
    ELSE state <= "00"; y <= "00"; END IF;
  WHEN "10" => IF  (x='0') THEN state <= "00"; y <= "00";
    ELSE state <= "11"; y <= "10"; END IF;
  WHEN "11" => IF  (x='0') THEN state <= "10"; y <= "11";
    ELSE state <= "01"; y <= "01"; END IF;
  WHEN OTHERS => state <= "00";
END CASE;
```

Anmerkungen

- Grundsätzlich sind alle synchronen Automatentypen (Mealy, Moore, Medwedew) mit allen drei Varianten modellierbar
- Drei-Prozess-Variante sollte aus Übersichtlichkeitsgründen vorgezogen werden
- Bei Modellierung mit `bit` und `bit_vector` Datentypen ist die vollständige Beschreibung des Zustandsraumes (falls gegeben) möglich. Modellierung mit `std_logic` ist jedoch universell einsetzbar und portierbar.
- Im Beispiel wurde ein Moore-Schaltwerk beschrieben, bei Mealy-Schaltwerken muss der 3. Prozess um die Eingangssignale in der Empfindlichkeitsliste erweitert werden (bei Medwedjew-Schaltwerken kann der 3. Prozess durch eine nebenläufige Anweisung „`output <= state`“ ersetzt werden)

Anmerkung zu partiell spezifizierten Zustandsräumen

Die Verwendung des `WHEN OTHERS`-Konstruktes bei der Übergangsfunktion ist keine Garantie für einen sicheren Entwurf, da dieses werkzeugabhängig behandelt wird.

Wichtig ⚠

Speziell bei Altera Quartus II: Wird keiner der mit `OTHERS` spezifizierten Zustände durch die normale FSM-Operation erreicht, so wird auch keine entsprechende Zustandsübergangslogik synthetisiert, d. h. das Schaltwerk ist nicht sicher!

Sollen die partiell spezifizierten Zustandsräume sicher behandelt werden, so müssen die entsprechenden Zustände entweder explizit kodiert (Übergänge durch den Entwerfer frei wählbar) oder eine Syntheseoption „`Safe state machine`“ auf „`on`“ gesetzt werden (alle Übergänge führen in den Initialzustand).

➥ `WHEN OTHERS` trotzdem niemals weglassen!

Ein- und/oder Ausgabe-Synchronisation

- Manchmal automatisch durch Vorgänger- oder Nachfolgerstufen gegeben
- Kann bei Bedarf in einem separaten Prozess zusätzlich implementiert werden:

```
SIGNAL x_sync : bit;
SIGNAL y_sync : bit_vector(1 DOWNTO 0);
synchronize : PROCESS(clk, res)
BEGIN
  IF (res = '1') THEN
    x_sync <= '0'; y <= "00";
  ELSIF (clk'event AND clk = '1') THEN
    x_sync <= x; y <= y_sync;
  END IF;
END PROCESS synchronize;
```

Zustandskodierung

- Manchmal fest vorgegeben
- Kann (und soll) ansonsten dem Synthese-Werkzeug überlassen werden

  ```
  TYPE state_type IS (S0, S1, S2, S3);
  SIGNAL state, next_state : state_type;
  ```

  Entsprechend müssen dann in allen Zuweisungen an `state` oder `next_state` die expliziten Werte durch `S0,...,S3` ersetzt werden.

➥ Die meisten Synthese-Werkzeuge bieten die Wahl der Zustandskodierung für alle Schaltwerke in einem Entwurf als Option bei den Synthese-Einstellungen

Zustandskodierung wurde im Detail in der Vorlesung „Digitaltechnik 2“ (Abschnitt „Schaltwerke“) besprochen.

Andere Varianten der expliziten Zustandskodierung

```
-- 1. Variante
TYPE state_type IS (S0, S1, S2, S3);
ATTRIBUTE ENUM_ENCODING : string;
ATTRIBUTE ENUM_ENCODING OF state_type :
                    TYPE IS "01 11 00 10"; -- "Random"
SIGNAL state, next_state : state_type;
-- 2. Variante
SUBTYPE state_type IS bit_vector(3 DOWNTO 0);
CONSTANT S0 : state_type := "0001"; -- One-hot
CONSTANT S1 : state_type := "0010";
CONSTANT S2 : state_type := "0100";
CONSTANT S3 : state_type := "1000";
SIGNAL state, next_state : state_type;
```

Das Attribut `ENUM_ENCODING` ist standardisiert, wird aber nicht von allen Werkzeugen unterstützt.

Anmerkungen zur Altera Quartus II Software

- FSM mit expliziter (durch den Anwender vorgegebenen) Zustandskodierung werden korrekt synthetisiert, jedoch nicht als solche erkannt und auch nicht in „FSM View“ angezeigt. Das ist kein Fehler, sondern eine Besonderheit der Software.
- Zustandskodierung kann (wenn nicht im VHDL-Code explizit vorgegeben) durch die entsprechende Syntheseoption „`State Machine Processing`“ eingestellt werden. „`Auto`“ bedeutet dabei, dass das Synthese-Frontend selbst die Kodierung aussucht.
- Die Kombination von Einstellungen „`Safe state machine=on`“ und „`State Machine Processing=one-hot`“ bzw. „`State Machine Processing=auto`“ kann zum schnellen Anstieg der Komplexität führen.

Verwendung der mehrwertigen Logik nach IEEE-1164

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
ENTITY fsm_1 IS
  PORT (x, clk, res : IN std_logic;
        y           : OUT std_logic_vector(1 DOWNTO 0));
END ENTITY fsm_1;
ARCHITECTURE three_proc OF fsm_1 IS
TYPE state_type IS (S0, S1, S2, S3);
SIGNAL state, next_state : state_type;
BEGIN
  store : PROCESS(clk, res) -- Speicherelemente
  BEGIN
    IF (res = '1') THEN state <= S0;
    ELSIF rising_edge(clk) THEN
      state <= next_state; END IF;
  END PROCESS store;
...
```

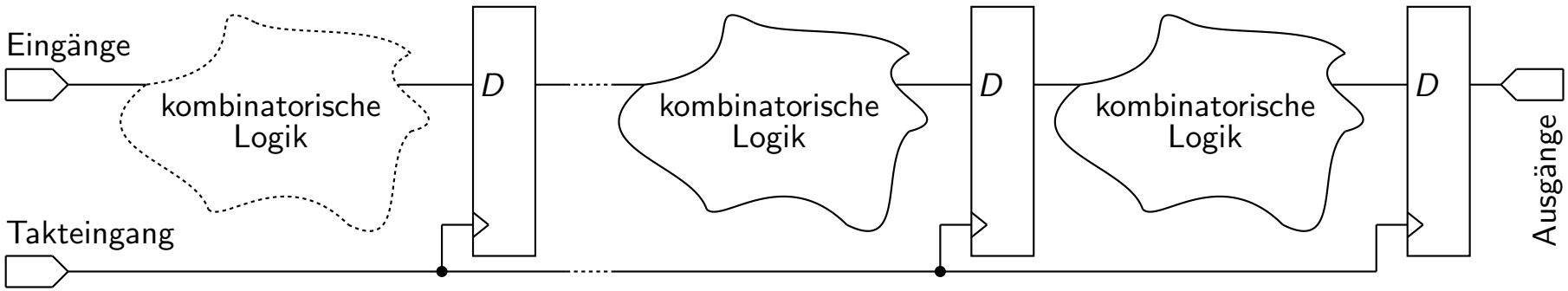
Übergangsfunktion mit IEEE-1164

```
transition : PROCESS(x, state) -- Uebergangsfunktion
BEGIN
  next_state <= S0;
  CASE state IS
    WHEN S0 => IF  (x='0') THEN next_state <= S1;
                 ELSE next_state <= S3; END IF;
    WHEN S1 => IF  (x='0') THEN next_state <= S2;
                 ELSE next_state <= S0; END IF;
    WHEN S2 => IF  (x='0') THEN next_state <= S3;
                 ELSE next_state <= S1; END IF;
    WHEN S3 => IF  (x='0') THEN next_state <= S0;
                 ELSE next_state <= S2; END IF;
    WHEN OTHERS => next_state <= S0;
  END CASE;
END PROCESS transition;
...
```

Ausgabefunktion mit IEEE-1164

```
...
  output : PROCESS(state) -- Ausgabefunktion
  BEGIN
    y <= "00";
    CASE state IS
      WHEN S0 => y <= "00";
      WHEN S1 => y <= "01";
      WHEN S2 => y <= "11";
      WHEN S3 => y <= "10";
      WHEN OTHERS => y <= "00";
    END CASE;
  END PROCESS output;
END ARCHITECTURE three_proc;
```

Register-Transfer-Level (RTL) Modell



- Trennung von Speicherelementen (Registern) und kombinatorischer Logik durch Modellierung in getrennten Prozessen
- Datenpfad einer Schaltung als Pipeline
- Vermeidung von taktzustandsgesteuerten Speicherelementen (Latches)
- Kritischer Pfad ist der längste kombinatorische Pfad (zwischen zwei Register-Stufen)

Beschreibung von Speicherelementen (getaktete Prozesse)

```
IF clk'event AND clk='1' ... -- positive Flanke
IF clk'event AND clk='0' ... -- negative Flanke
WAIT UNTIL clk'event AND clk='1' ... -- positive Flanke
WAIT UNTIL clk'event AND clk='0' ... -- negative Flanke
IF rising_edge(clk) ...  -- positive Flanke
IF falling_edge(clk) ... -- negative Flanke
```

- Die ersten vier Varianten sollten bei `std_ulogic` nicht eingesetzt werden, wenn ausschließlich die LH- oder HL-Flanken erkannt werden sollen
- Seit VHDL-2008 sind die Funktionen `rising_edge` und `falling_edge` auch für `bit`- und `boolean`-Objekte definiert (vorher nur `std_ulogic`)
- In einer Anweisung nur ein Typ der Flankensteuerung (nur positive oder nur negative Flanke)

Allgemeine Kodierungsrichtlinien für synthesefähige Modelle

- kombinatorische und sequentielle Logik in getrennten Prozessen modellieren (RTL-Modell)
- keine Initialisierungswerte verwenden (Verhaltensunterschiede zwischen Simulation und realer Schaltung)
- in die Empfindlichkeitsliste von getakteten Prozessen nur das Taktsignal und Reset aufnehmen
- in die Empfindlichkeitsliste von kombinatorischen Prozessen alle Signale, deren Wert abgefragt (gelesen) wird, aufnehmen
- alle Signale und Variablen in kombinatorischen Prozessen müssen vor dem ersten Lesen einen definierten Wert besitzen (Vermeiden von Latches)

Allgemeine Kodierungsrichtlinien für synthesefähige Modelle (fortgesetzt)

- auch bei `std_logic`: Nur einen Treiber pro Signal verwenden (außer bei tristate-fähigen Busverbindungen)
- keine Signale, auf die schreibend zugegriffen wird, in die Empfindlichkeitsliste des entsprechenden kombinatorischen Prozesses einfügen (Vermeiden von kombinatorischen Schleifen)
- Variablen nur zur Speicherung von Zwischenergebnissen innerhalb von Prozessen verwenden (keine `SHARED VARIABLES`)
- alle Variablen müssen vor dem ersten Lesen einen definierten Wert besitzen (Vermeiden von Latches)
- komplexe Modelle sorgfältig strukturieren und kommentieren!

Zusammenfassung

- Synthese von Speicherelementen
- Synthese von Schaltwerken
- Besonderheiten verschiedener Beschreibungsformen
- Umgang mit Quartus II Software bei der Schaltwerksynthese
- RTL-Modell und Richtlinien zur Erzeugung synthesefähiger Modelle

Aufgaben zum Selbststudium

1. Beschreiben und synthetisieren Sie das Schaltwerk zur Symbolerkennung („Digitaltechnik 2“, 4. Vorlesung) und das Schaltwerk zur Flankenerkennung („Digitaltechnik 2“, 5. Vorlesung).
2. Setzen Sie die Lösung der Aufgabe zur 5. Vorlesung „Digitaltechnik 2“ (Mealy-Schaltwerk) in VHDL um.

Aufgabenlösungen zur 12. Vorlesung

Schaltwerksynthese

Der entsprechende VHDL-Quellkode ist auf dem Handout-Server unter

`VHDL/Aufgabenloesungen/VHDL/vhdl_v12_symbol_fsm.vhd`,
`VHDL/Aufgabenloesungen/VHDL/vhdl_v12_flanke_fsm.vhd`,

und

`VHDL/Aufgabenloesungen/VHDL/vhdl_v12_mealy_fsm.vhd`

zu finden.