

WP ECE SS24

INF-WP Einführung Computer Engineering
"VHDL-Teil"

Prof.: Michael Schäfers

email: **ECE.S24S@Hamburg-UAS.eu**

Raum:

780

(spezifisch für diese Veranstaltung)

 "MS-Teams"

data: https://users.informatik.haw-hamburg.de/~schafers/LOCAL/S24S_ECE

Wer steht gerade da vorn?

Michael Schäfers

Informatikstudium
+
Promotion
an der TU Braunschweig

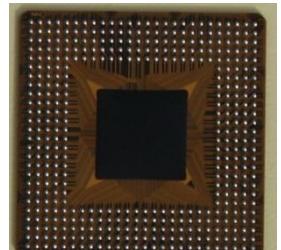
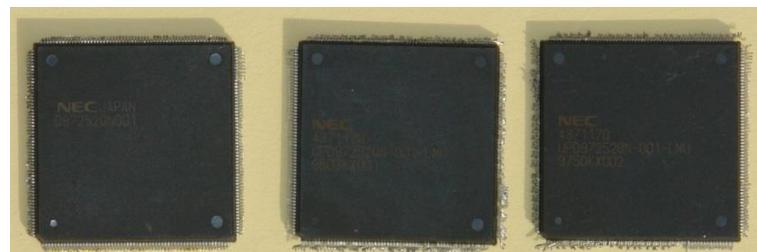
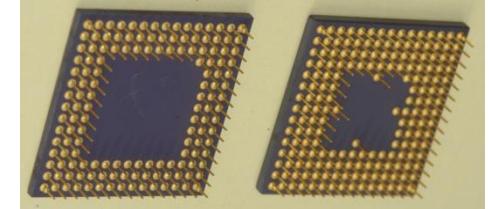
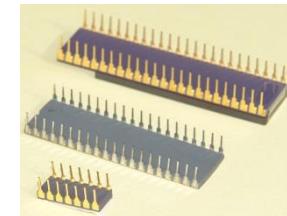
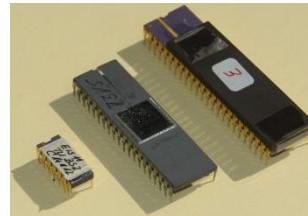
Berufstätigkeit (1995-2002)
Nokia Telecommunications

- Nokia Networks
- NSN (*Nokia Siemens Networks*)
- NSN (*Nokia Solutions and Networks*)

Düsseldorf

R&D → ASIC Entwicklung

HAW Hamburg seit April 2002



<BREAK>

Folien mit **Skipped**-Markierung

- Folien, die rechts unten mit

skipped

markiert sind,
wurden übersprungen.

Sie müssen also zum Vorlesungszeitpunkt nicht beunruhigt sein, wenn eine derartige Folie Dinge enthält, die Sie nicht verstehen.

Sofern das "Thema" nicht später wieder aufgegriffen wird,
ist es nicht Prüfungs-relevant.

Das kennen Sie schon von mir bzw. P1 ;-)

wdn. Aus P1
bekannt.

Folien mit **touched**-Markierung

- Folien, die rechts unten mit

touched

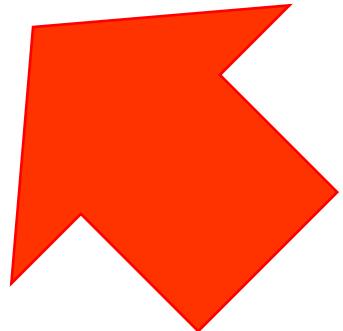
markiert sind,
wurden nur angerissen.

Sie müssen also zum Vorlesungszeitpunkt nicht beunruhigt sein, wenn eine derartige Folie Dinge enthält, die Sie nicht verstehen.

Sofern das "Thema" nicht später wieder aufgegriffen wird,
ist es nicht Prüfungs-relevant.

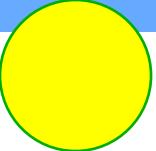
Das kennen Sie schon von mir bzw. P1 ;-)

*wdn. Aus P1
bekannt.*

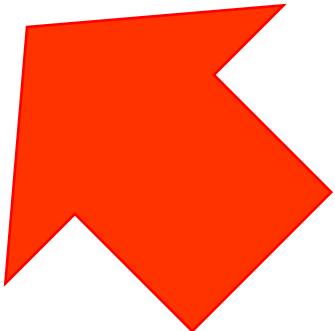


Folien mit grünen Punkt

- Folien mit grünen Punkt sind besonders wichtig



Folien mit gelben Punkt



- Folien mit gelben Punkt sind wichtig

- Die nachfolgenden Folien sind ein Auszug aus der ECE-Veranstaltung des WS20/21
 - Damals waren viele Randbedingungen anders - z.B. andere HW
-
- "Jetzt" in dieser Veranstaltung (SS24) ist vieles überarbeitet & verbessert wurden

VHDL

Vorweg

- VHDL wird nur "vorgestellt"
- VHDL wird **nicht** (vergleichbar zu Java in P1+P2) im Detail erklärt
- Sie werden sich selbstständig in die Sprache einarbeiten müssen
- Informatiker müssen befähigt sein
sich in kürzester Zeit in neue Sprachen einzuarbeiten

Grundsätzliches

- VHDL ist eine HW-Beschreibungssprache
(*HDL: Hardware Description Language*)
- VHDL steht für VHSIC Hardware Description Language
- VHSIC steht für Very High Speed Integrated Circuit
- VHDL ist standardisiert im Standard IEEE1076.
Jedoch inzwischen fünf Versionen:
 - IEEE 1076-1987 Standard VHDL kurz **VHDL'87**
 - IEEE 1076-1993 Standard VHDL kurz **VHDL'93**
 - IEEE 1076-2002 Standard VHDL kurz **VHDL2002**
 - IEEE 1076-2008 Standard VHDL kurz **VHDL2008**
 - IEEE 1076-2019 Standard VHDL kurz **VHDL2019**
- VHDL'98 - der Versuch VHDL objektorientiert zu machen - scheiterte

Was ist VHDL?

- eine weit verbreitete HDL
- eine Programmiersprache
- eine Simulationssprache
- eine Spezifikationssprache
- eine Dokumentationssprache
- eine standardisierte (also nicht proprietäre) Sprache
- brauchbar als Input für Synthese

HDLs und Programmiersprachen

- viele Forderungen werden von "normalen" Programmiersprachen erfüllt
- hohe Grad an Parallelität ist Problem für "normale" Programmiersprachen
- Echtzeit-Sprachen sind ausgerichtet auf das Einhalten realer Zeiten (Echtzeit) und nicht auf exakte Auflösung der zeitlichen Diskretisierung (z.B. Femtosekunden) ohne Echtzeitanforderungen genügen zu müssen
- HDLs sind gewöhnlich an Programmiersprachen angelehnt
 - VHDL an ADA (*und ADA an PASCAL*)*
 - Verilog an C
 - *SystemC an C*
- * *Wer PASCAL oder Modula-2, Oberon, ADA kann, dem kommt sehr viel in VHDL bekannt vor*

gängige HDLs

- 2 HDLs haben sich durchgesetzt und werden noch lange Zeit nebeneinander her existieren
 - VHDL
 - stärker vertreten in Europa
 - stärker auf den höheren Ebenen und "mehr" normale Programmiersprache wurde als HDL entworfen
 - Verilog
 - stärker vertreten in USA und Japan
 - (verstärkt gegenwärtig durch internationale Firmen seine "Verbreitung")
 - stärker auf den niederen Ebenen – sehr guter Gate-Level-Simulator
 - hat sich erst zur HDL entwickelt
- es wird immer wieder versucht auf höherer Ebene ein Sprache zu etablieren
 - SystemC ist ein Beispiel hierfür
- Traum einer Sprache sowohl für HW als auch SW
 - Idee:
 - Nachfolgende Tools generieren daraus automatisch die SW- und die HW-"Teile"

Geschichte von VHDL

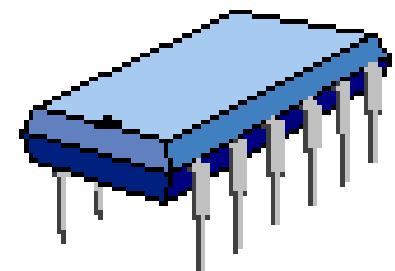
- US DoD forcierte Entwicklung und Einsatz von VHDL, die HDL sollte
 - standardisiert sein und geeignet für
 - Spezifikation,
 - Simulation (also vom Computer lesbar und ausführbar),
 - Dokumentation,
 - und damit insbesondere Austausch, Wiederverwendung und Wartung von Schaltungen gewährleisten
- VHDL basiert bewusst auf ADA
- 1981 formaler Start der VHDL-Entwicklung
- 1985 VHDL Version 7.2
- 1986 IEEE Standard 1076 ("1. VHDL" – VHDL'87)
- 1993 überarbeiteter IEEE Standard 1076 ("2. VHDL" – VHDL'93)
im wesentlichen ein Bug-Fix
- objektorientiertes VHDL immer wieder in der Diskussion
(z.B. als "VHDL'98" – kam aber nicht)
- es folgten VHDL2002, VHDL2008 und schließlich 2019

ein kurzer Ausflug

**Abstraktionsebenen
(für Schaltungen)**

**ein Überblick zur
Hintergrund-Information**

Begriffsklärung



ASIC

- “Application Specific Integrated Circuit”
- keine klare Definition / einheitliche akzeptierte Auffassung von "ASIC"
- (*grundsätzlich*) ein speziell für einen Kunden oder eine Anwendung gefertigter integrierter Schaltkreis

Abstraktionsebene

- **wichtiges** Hilfsmittel für die Entwicklung **komplexer** Systeme
- ist definiert durch eine **reduzierte Sicht** auf diejenigen Elemente, die für diese spezielle Ebene von **Bedeutung** sind. Auf jeder Abstraktionsebene werden daher nur die hier wichtigen Teilespekte betrachtet
- je abstrakter die Ebene, desto “höher” die Ebene
je mehr Details, desto “niedriger” die Ebene
- wichtig bei der Diskussion von Abstraktionsebenen: Für welchen **Zweck** diese Ebenen eingesetzt werden. Hier ist es der Entwurf von ASICs
- in der Praxis “Vermischung der Dimensionen” Verhalten, Struktur und Physik/Geometrie

Funktionale Ebene (Behavioral Level)

- die Funktionale Ebene ist wichtig für die **Spezifikation**
 - es wird nur das **Verhalten** beschrieben und keine Struktur
 - Konzentration auf das “WAS” unter Vernachlässigung des “WIE”
 - Funktion wird beschrieben durch **charakteristische** Variablen und deren Werteverläufen über die Zeit
-
- Zeitmodell: Kausalität
 - beobachtbare Werte: beliebige Werte im frei definierbarem Wertebereich
-
- Arbeitsmittel: Dokumente in natürlicher Sprache, Ein-/Ausgabetabellen, Skizzen, (**VHDL**)
-
- Beispiele: Spezifikation, Pflichtenheft

touched

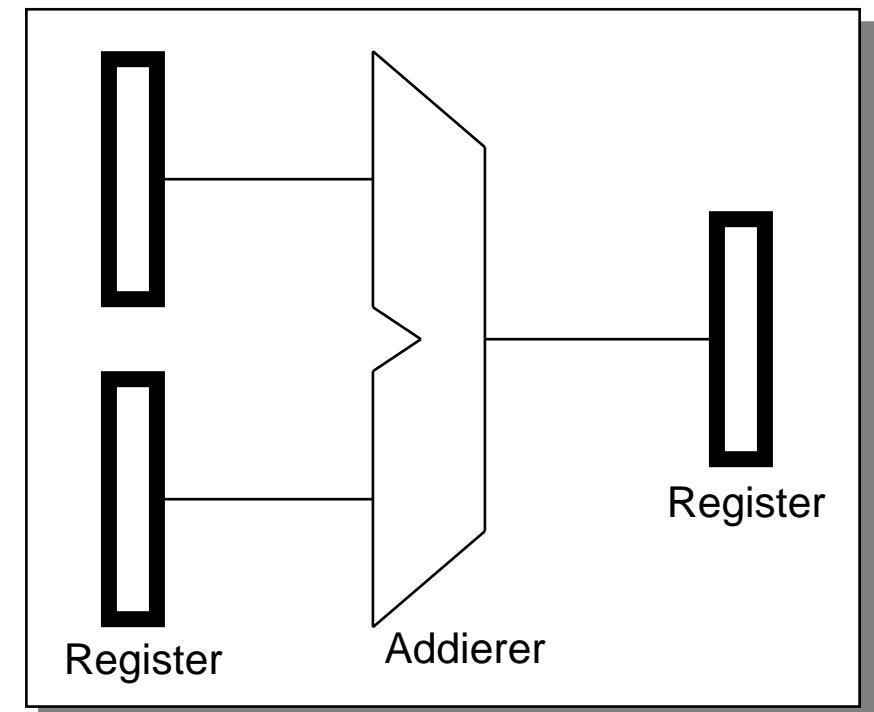
System-Ebene (System Level)

- erster Bezug zur späteren Struktur der Realisierung (Hardwarepartitionierung); Funktionale Einheiten (Blöcke, ASICs) werden bestimmt und beschrieben
- erste Überlegungen zur relativen Lage der Komponenten auf dem Chip (Floorplan)
- Zeitmodell: Kausalität oder diskrete Realzeit
- beobachtbare Werte: Vektoren von Bits mit Interpretation
- Arbeitsmittel: Dokumente, Ein-/Ausgabetabellen, **VHDL**
- Beispiele: Architekturplan, Blockdiagramm

touched

Register-Transfer-Ebene (RTL)

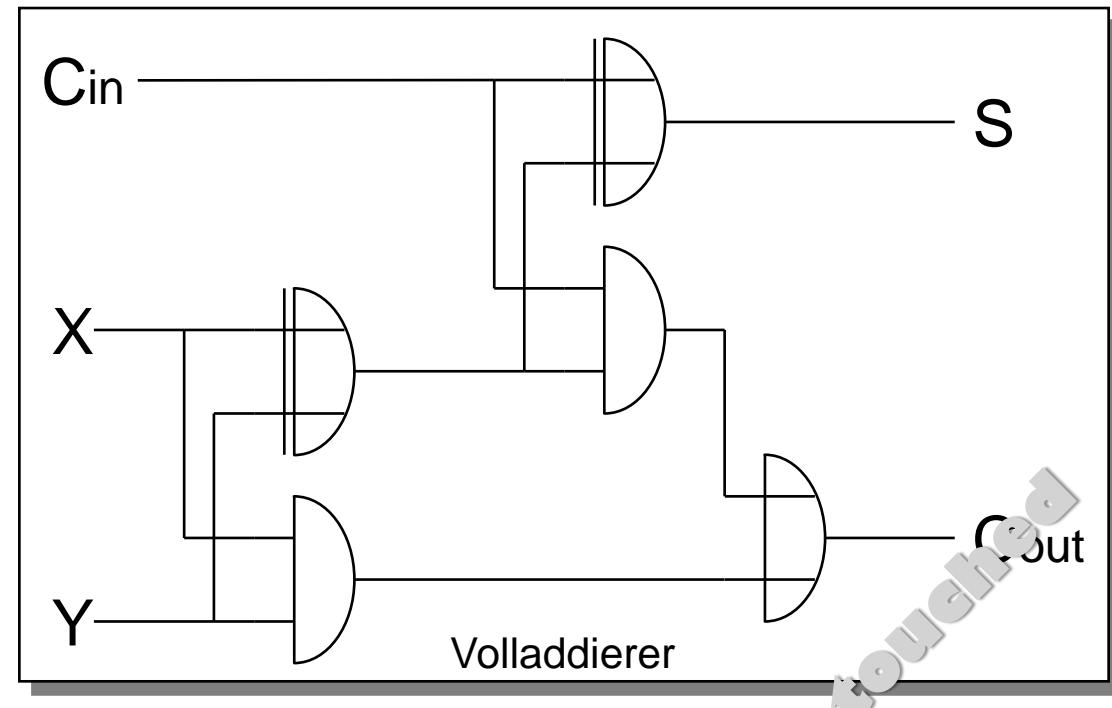
- Ziel ist die **Trennung** der **kombinatorischen** und der **sequentiellen** Logik
 - 1. mögliche Schnittstelle zum Halbleiterhersteller
 - endgültige Hardwarestruktur wird sichtbar
 - **wichtige** Ebene des ASIC-Entwurfs
-
- Zeitmodell: diskrete Realzeit (Takte)
 - beobachtbare Werte: Vektoren von Bits
 - Arbeitsmittel: **VHDL**, **VERILOG**



Logik- oder Gatter-Ebene (Gate Level)

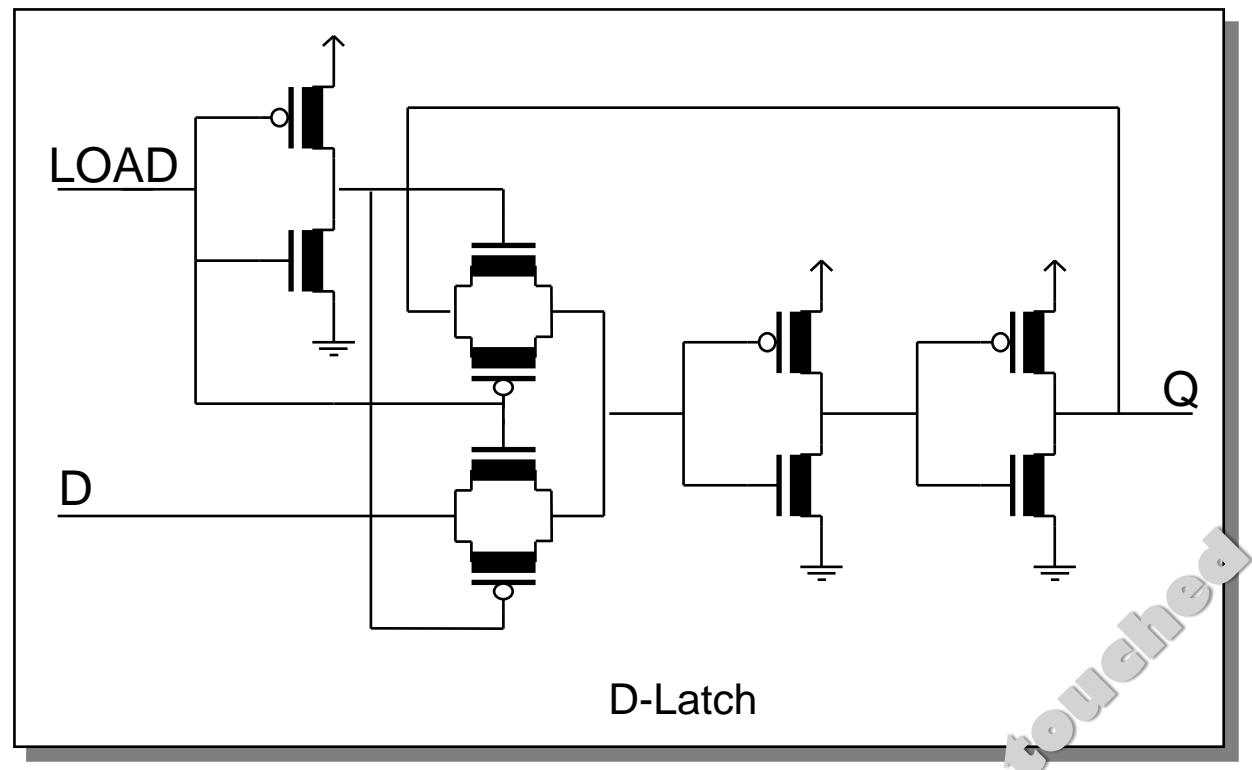
- Schaltungen auf Netzen von bekannten Gattern (AND, OR, INV, NAND, NOR, XOR, D-FF, ...)
- übliche Schnittstelle zum Halbleiterhersteller
- “Arbeitsebene” für viele Flow-Schritte (Timing-Simulation, backannotation, ATPG, Fehlerüberdeckung, P&R, ...)
- früher Schaltplaneingabe (Schematic Entry)

- Zeitmodell: kontinuierliche Realzeit
- beobachtbare Werte:
Bi-Tupel (logischer Wert, Stärke)
- Arbeitsmittel: VERILOG,
VHDL (VITAL)



Schaltkreis-Ebene (Switch Level)

- **Transistoren**, Widerstände, Kondensatoren
- Zeitmodell: kontinuierliche Realzeit
- beobachtbare Werte: kontinuierlicher Wertebereich für Spannungen, Ströme (analog) oder Bi-Tupel (digital)
- Arbeitsmittel:
SPICE (analog)
aber auch Verilog (digital)

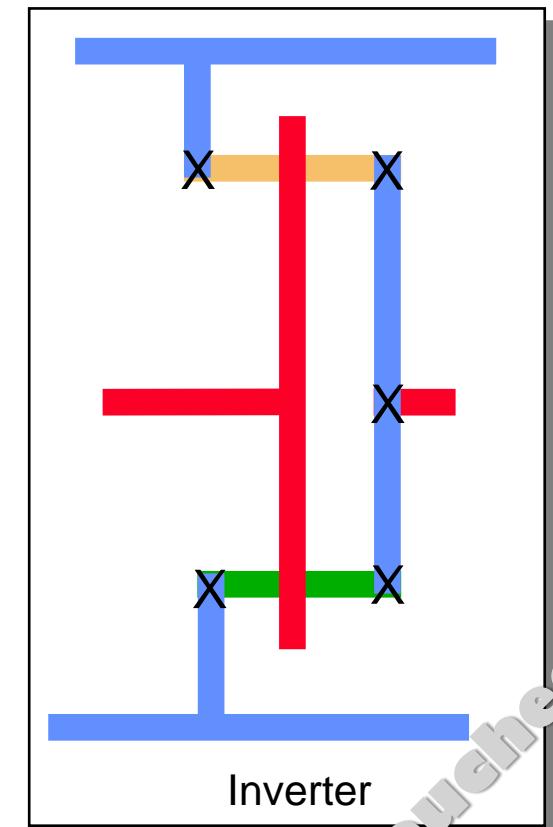


Symbolische Layout-Ebene

- **Stick-Diagramm** (“erweiterte Schaltkreis-Ebene”)
- isolierte Schichten (Layer) in **farbiger** Darstellung
- verschiedene Leitungen eines Layers dürfen sich nicht kreuzen
- zusätzliche Topologieinformation
=> logische Anordnung der Komponenten wird bestimmt
- nicht maßstabsgetreu

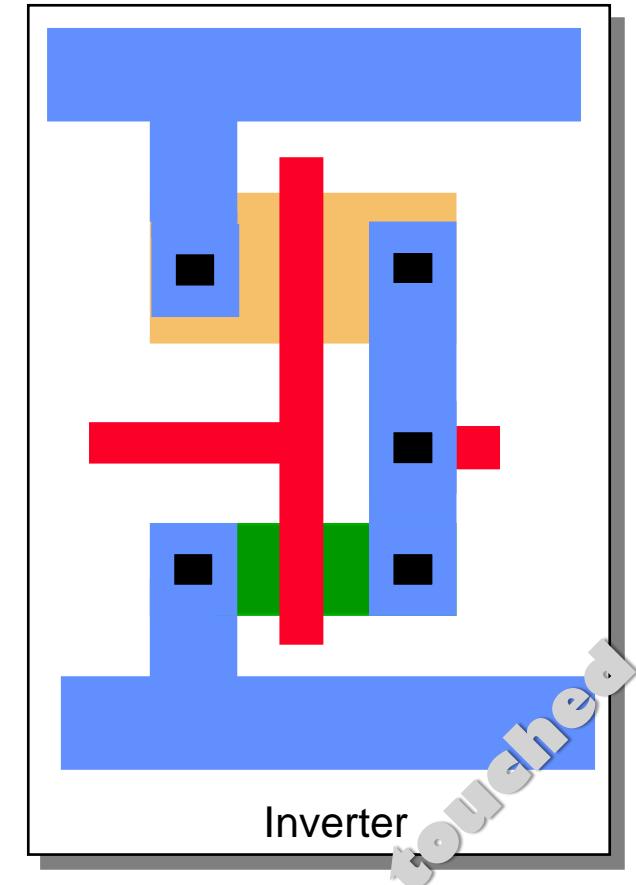
- Zeitmodell: kontinuierliche Realzeit
- beobachtbare Werte: kontinuierlicher Wertebereich
für Spannungen, Ströme

- Arbeitsmittel: SPICE, Editor/Papier



geometrische Layout-Ebene

- **maßstabsgetreue** Vergrößerung der schichtenweisen Strukturen im fertigen Chip
- geometrische Entwurfsregeln (Mindestabstände und Mindestbreiten)
- simuliertes Layout ist die Schnittstelle des Full-Custom-Designers
- geometrische Layout wird in textuelle Beschreibungsform (GDS2, CIF2.0) gewandelt. Nach dem Tape-Out werden hiermit die Masken für die Chipfertigung erstellt
- mittels Extraktor Überführung in Schaltkreisebene
- Zeitmodell: kontinuierliche Realzeit
- beobachtbare Werte: kontinuierlicher Wertebereich für Spannungen, Ströme
- Arbeitsmittel: SPICE

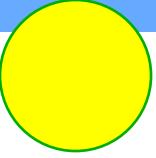


Zusammenfassung

- Einführung von Abstraktionsebenen ermöglicht
 - das **Verständnis** komplexer Systeme
 - den Entwurf **komplexer** Systeme
- hoch ↑ Funktionale Ebene (**Behavioral Level**)
Systeme-Ebene (System Level)
Register-Transfer-Ebene (**RTL**)
Logik- oder Gatter-Ebene (**Gate-Level**)
Schaltkreis-Ebene (Switch-Level)
Symbolische Layout-Ebene
Geometrische Layout-Ebene
- niedrig ↓
- mit dem Fortschreiten der Integrationsdichte werden die implementierten Funktionen komplexer
- mit der steigenden Komplexität nimmt die Bedeutung der höheren Ebenen weiter zu
- ASIC-Entwicklung wird Software-Entwicklung ähnlicher

Anforderungen an HDLs

- formale, korrekte Beschreibung eines HW-Modells
- für Mensch und Maschine leicht lesbare Form
- leicht und schnell erlernbar
- möglichst viele Abstraktionsebenen können, auch gemischt, vorkommen (*abgesehen von der Layout-Ebene – hierfür z.B. EDIF, CIF, GDS-II*)
- Unabhängigkeit von der verwendeten Technologie (CMOS, GaAs, TTL, ...)
- Unterstützung eines modularen Entwurfs und der Wiederverwendbarkeit von Teilschaltungen
- gängige SW-Entwurfsverfahren (top-down, bottom-up, library-based, ...) sollen unterstützt werden
- Standardisierung der HDL soll Austausch der Modelle (und damit Simulation auf verschiedenen Rechnersystemen) ermöglichen
- ausreichende Geschwindigkeit des Simulators auch bei sehr großen und komplexen Modellen



Begriffe

"Normale" Programmiersprache:

- Der Compiler übersetzt den Code in lauffähigen Code - die Applikation.
- Diese SW-Applikation ist das Ziel des SW-Entwicklungs-Prozesses.

HDL:

- Der Compiler übersetzt den HDL-Code in Simulations-Code, der zur Verifikation der (späteren) HW dient
- Die **Synthese** (der "Silicon-Compiler") erzeugt aus dem HDL-Code "etwas" (konkret eine Netzliste), das ein bestimmender Schritt zu Erzeugung der HW ist.
- Die HW-Applikation ist letztlich das Ziel des Entwicklungs-Prozesses.

Unterschiede

Das Programmieren mit einer HDL ähnelt/"folgt" in vielen Punkten dem Programmieren mit einer "normalen" Programmiersprachen. Sehr vieles aus dem Software-Engineering gilt auch in der HW-Entwicklung mit einer HDL.

In einigen Punkten ist es jedoch grundverschieden.

Bei der Programmierung mit einer "normalen" Programmiersprache ist das Ziel:

- performanter Code
- wartbarer Code

Bei der Programmierung mit einer HDL ist das Ziel:

- performante HW
- "kleine" HW
- wartbare HW
(und da diese per Synthese aus VHDL erzeugt wird, wartbarer VHDL-Code)
- Dokumentation der HW-Funktion
- akzeptabel performater Simulations-Code

Unterschiede (Beispiel)

Bei der Programmierung mit einer "normalen" Programmiersprache

- werden "Dinge" nur berechnet, wenn Sie gebraucht werden
- die Dynamik der Programm-Ausführung führt den jeweiligen Code dann nur aus, wenn das "Ergebnis" auch wirklich benötigt wird
- ist Zeit kein Thema
(Ausnahme Echtzeit-Programmierung - hier wichtig "Reaktionszeit der SW")

HW ist statisch, es kann **nicht** spontan in einer Berechnung neue HW in einem IC ergänzt werden.

Bei der "Programmierung" mit einer HDL

- werden "Dinge" auch(bzw. immer) berechnet, obwohl Sie (gerade) gar nicht benötigt werden, weil die HW statisch existent ist und dies Synthese und Wartbarkeit erleichtert
- ist Zeit eine wichtige Größe haben Dinge ("Ereignisse") ein Bezug zur Zeit (z.B. HW liefert nach 2,537ps Ergebnis)

Bemerkung:

Das dynamische Rekonfigurieren von HW wird z.B. durch FPGA unterstützt.

Unterschiede

- die HDL/Sprache selbst und der zugehörige Simulator gehen in der "Praxis" Hand in Hand.
Häufig wird hier nicht sauber unterschieden
- Formal gibt es 3 Schritte
 - analyzing
 - elaboration
 - simulation
- In der Praxis scheint es meist nur zwei Schritte zu geben
 - compilation
 - loading design
 - simulation
- VHDL-Beschreibungen ("Programme") laufen auf einem Simulator.
Deswegen wird gesagt, dass VHDL keine Programmiersprache ist.

Es wird unterschieden zwischen

- Code der HW beschreibt,
also später synthetisiert werden soll (also Synthese-fähig ist)
und entsprechende Eigenschaften haben muss.
Dies ist später häufig RTL-Code.
Solcher Code beschreibt auch eine **Struktur** ("Topologie von HW-Elementen")
- Code der nicht das Ziel hat konkrete HW zu beschreiben
- Das könnte sein
 - Test-Code zur Verifikation (Stichwort "TestFrame")
 - Referenz- oder Spezifikations-Code
- Solcher Code ist typischer Weise **Behavioral Code** und beschreibt (ausschließlich) ein Verhalten
- Behavioral Code muss sich nicht von normalen Software-Code unterscheiden
- Eine mögliche Struktur, die beschrieben wird, erhebt keinen Anspruch auf die spätere HW abgebildet zu werden.

Die HDL: VHDL

Grundsätzliches

- VHDL ist nicht Case Sensitive
- VHDL ist stark typisiert (*strong typing*) sehr ähnlich Java
- Unter VHDL müssen Dinge immer erst deklariert werden bevor sie benutzt werden
- VHDL trennt zwischen "Deklarations-Teil" und "Anweisungs-Teil"
- PASCAL-Kenntnisse sind für VHDL nützlich - werden aber nicht vorausgesetzt
Es macht keinen Sinn extra PASCAL für VHDL zu lernen
Bemerkung richtet sich nur an "die", die PASCAL mal gelernt haben

Typen

skalar/enumeration

- CHARACTER
 - BIT
 - BOOLEAN
 - INTEGER
 - REAL
 - TIME
-
- STD_LOGIC
 - STD_ULOGIC

array

STRING
BIT_VECTOR

access, file

STD_LOGIC_VECTOR
STD_ULOGIC_VECTOR

Datentypen

- CHARACTER für ASCII-Zeichen
- BIT für binäre Werte (0 oder 1) - in der HW denkt man in Bit
werden wir aber nicht/kaum nutzen - sondern std_logic
- BOOLEAN für boolsche Werte (true oder false)
- INTEGER für ganzzahlige numerische Werte ("meist" 32 Bit)
- REAL für Floating Point Werte
- TIME nimmt Zeit auf (später Arbeitsbegriff "V-Zeit")
"Dinge"/Ereignisse werden einer Zeit zugeordnet
- STRING für Zeichenketten (Achtung! VHDL ist nicht Java ;-))
- BIT_VECTOR ist ein ARRAY über BIT
werden wir aber nicht/kaum nutzen - sondern std_logic_vector

Datentypen

- ACCESS ist ein Pointer (Achtung! VHDL ist weder Java noch C ☺)
 - FILE ist ein Datentyp für Dateien
-
- Für Testbenches (TestFrames) bzw. behavioral Code macht FILE vereinzelt Sinn
 - ACCESS ist eher exotisch
 - Beide sind nicht für synthetisierbaren Code geeignet

"Objekte"

- CONSTANT - echte Konstanten
- VARIABLE - kennzeichnet echte primitive Variablen
Wertzuweisungen an Variablen erfolgen sofort
- SIGNAL - eine "andere" "Variablen"-Art
Wertzuweisungen an Signale erfolgen immer verzögert
Modelliert Laufzeiten - HW braucht immer Zeit
"Gibt es so nicht in einer normalen Programmiersprache"
- FILE - hat Datentyp und "Objekt"-Eigenschaften
- Beispiel:

```
variable COUNTER : integer;
```

Assignments

2 Zuweisungen

- $:=$ Variable Assignment

Wert-Zuweisung erfolgt **unmittelbar sofort** an Variable

Variablen bzw. Variable Assignments werden innerhalb von Prozessen genutzt für algorithmische temporäre "Größen"

- $<=$ Signal Assignment

Wert-Zuweisung erfolgt **verzögert** an Signal

Signale repräsentieren HW-Verbindungen und werden genutzt um "Werte" zwischen den Prozessen/Komponenten auszutauschen

- Bemerkung

In VHDL ist die Zeit zweidimensional

Type Qualification/Conversion

Type Qualification

- TYPE ampel IS (gruen, gelb, rot);
- TYPE bunt IS (gruen, rot, blau, gelb, braun, silber, gold);
- v := ampel'(gruen);
- s := string' ("dies ist ein string und kein std_logic_vector");

Type Conversion

- r := REAL(123)
- i := INTEGER(3.1415926);

Operatoren

- AND, OR, NAND, NOR, XOR, NOT,
- SLL, SLA, SRL, SRA, ROL, ROR,
- =, /=, <, <=, >, >=,
- &, +, -, *, /, MOD, REM, **

$$5 \text{ MOD } \underline{3} = 1 \text{ REST } \underline{2}$$

$$5 \text{ MOD } \underline{(-3)} = (-2) \text{ REST } \underline{(-1)}$$

$$(-5) \text{ MOD } \underline{(-3)} = 1 \text{ REST } \underline{(-2)}$$

$$(-5) \text{ MOD } \underline{3} = (-2) \text{ REST } \underline{1}$$

$$\underline{5} \text{ REM } 3 = 1 \text{ REST } \underline{2}$$

$$\underline{5} \text{ REM } \underline{(-3)} = (-1) \text{ REST } \underline{2}$$

$$\underline{(-5)} \text{ REM } \underline{(-3)} = 1 \text{ REST } \underline{(-2)}$$

$$\underline{(-5)} \text{ REM } 3 = (-1) \text{ REST } \underline{(-2)}$$

Operatoren

- $=, /=, <, <=, >, >=$

JAVA-“Gegenstück”

- $=$ entspricht $==$
- $/=$ entspricht $!=$
- $<$ entspricht $<$
- $<=$ entspricht $<=$
- $>$ entspricht $>$
- $>=$ entspricht $>=$

Operatoren

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

...

```
variable X,Y : boolean;
```

...

not X

X and Y

X or Y

X xor Y

X nand Y

X nor Y

...

JAVA-“Gegenstück”

entspricht ! X

entspricht X & Y

entspricht X | Y

entspricht X ^ Y

entspricht ! (X & Y)

entspricht ! (X | Y)

Operatoren

- $+, -, *, /$ wie bekannt für Addition, Subtraktion, Multiplikation und Division
- $\&$ für Konkatenation von Arrays
- $**$ für Power
- MOD, REM - Rest-Berechnung (bei Division)

$$5 \text{ MOD } \underline{3} = 1 \text{ REST } \underline{2}$$

$$5 \text{ MOD } \underline{(-3)} = (-2) \text{ REST } \underline{(-1)}$$

$$(-5) \text{ MOD } \underline{(-3)} = 1 \text{ REST } \underline{(-2)}$$

$$(-5) \text{ MOD } \underline{3} = (-2) \text{ REST } \underline{1}$$

$$\underline{5} \text{ REM } 3 = 1 \text{ REST } \underline{2}$$

$$\underline{5} \text{ REM } \underline{(-3)} = (-1) \text{ REST } \underline{2}$$

$$\underline{(-5)} \text{ REM } \underline{(-3)} = 1 \text{ REST } \underline{(-2)}$$

$$\underline{(-5)} \text{ REM } 3 = (-1) \text{ REST } \underline{(-2)}$$

Operatoren

- SLL Shift Left Logical
 - SLA Shift Left Arithmetical
 - SRL Shift Right Logical
 - SRA Shift Right Arithmetical
 - ROL ROotate Left
 - ROR ROotate Right
-
- anstelle von << und >> usw.
 - wird eher selten benötigt - meist werden ARRAYS zu Fuß umgestellt

```
VARIABLE i : integer;  
VARIABLE r : real;
```

Zahlendarstellungen

i := 1000;	-- 1000	
i := 10e2;	-- 1000	= 10e2 = 10 * 1e2
i := 16#10#;	-- 16	basis=16#wert=0x10#
i := 16#A#;	-- 10	basis=16#wert=0xA#
i := 10#10#;	-- 10	basis=10#wert=10#
i := 8#12#;	-- 10	basis=8#wert=12 _{octal} #
i := 2#1010#;	-- 10	basis=2#wert=1010 _{dual} #
i := 2#0110_0100#;	-- 100	
i := 2#11111111_11111111#;	-- 65535	
i := b"0000_1011";	-- 11	basis=2"wert=1011 _{dual} "
i := x"11";	-- 17	basis=16#wert=0x11#

```
r := i * 1.0;  
r := 3141592.65358979323;  
r := 3_141_592.653_589_793_23;  
r := 43.6e-4;  
r := 10#43.6#e-4;
```

touched

Selektion

```
[if_label:]  
IF condition THEN  
    sequence_of_statements;  
[ELSIF condition THEN  
    sequence_of_statements; ]  
[ELSE  
    sequence_of_statements; ]  
END IF [if_label];
```

IF..THEN..ELSE

```
-- maximum berechnen  
if x < y then  
    max := y;  
else  
    max := x;  
end if;  
  
-- Absolut-Wert berechnen  
if x = integer'low then  
    report "PROBLEM" severity error;  
elsif x < 0 then  
    abs_x := -x;  
else  
    abs_x := x;  
end if;
```

CASE

```
[case_label:]  
CASE expression IS  
    WHEN choices      => sequence_of_statements;  
    [WHEN OTHERS        => sequence_of_statements; ]  
END CASE [case_label];
```

-- CHARACTER nach INTEGER wandeln

```
case char is  
    when '0' =>      int := 0;  
    when '1' =>      int := 1;  
    when '2' =>      int := 2;  
    when '3' =>      int := 3;  
    when '4' =>      int := 4;  
    when '5' =>      int := 5;  
    when '6' =>      int := 6;  
    when '7' =>      int := 7;  
    when '8' =>      int := 8;  
    when '9' =>      int := 9;  
    when others => int := -1; report "PROBLEM" severity note;  
end case;
```

Iteration

LOOP

```
[loop_label:]  
[iteration-scheme] LOOP  
    sequence_of_statements  
END LOOP [loop_label];
```

```
for i in 1 to 10 loop  
    ...  
end loop;
```

```
while condition loop  
    ...  
end loop;
```

```
loop  
    ...  
    if condition then exit; end if;  
    ...  
end loop;
```

```
...  
next [loop_label] when condition  
exit [loop_label] when condition
```

Ausgewählte Statements

Assertion

- gab es auch schon unter Java
- Semantik ist sehr ähnlich, aber Syntax ist anders
- Es gibt eine Severity - eine Gewichtung der Meldung - wie wichtig ist sie?
- Von "unwichtig" nach "wichtig" gibt es:
 - NOTE, WARNING, ERROR, FAILURE, FATAL
- Meldungen können nach ihrer Wichtigkeit gefiltert werden
- Abhängig von der Wichtigkeit einer Meldung kann die Simulation gestoppt werden
- Kann nicht synthetisiert werden
- Macht dennoch in Code für Synthese Sinn

ASSERTION REPORT

```
[assertion_label:]  
ASSERT condition  
[REPORT expression]  
[SEVERITY severity_level];
```

```
[report_label:]  
REPORT expression  
[SEVERITY severity_level];
```

```
-- zusichern das x positiv ist  
assert      (x >= 0)  
report      "x ist negativ"  
severity warning;  
  
-- zusichern das x positiv ist (Alternative)  
if (X<0) then  
    ...  
    report      "x ist negativ"  
    severity warning;  
else  
    ...  
end if;
```

```
-- im STANDARD-Package definiert:  
TYPE SEVERITY_LEVEL IS (NOTE, WARNING, ERROR, FAILURE, FATAL);
```

[null_label:]

NULL;

NULL

```
...
if condition then
  ...
elseif condition then
  ...
elseif condition then
  ...
else
  -- ja, ich habe diesen Fall bedacht und es soll hier nichts passieren
  null;
end if;
...
```

Eine Design-Philosophie ist es alle Fälle auszucodieren. Um explizit anzuzeigen, dass ein Fall, in dem nichts passieren soll, bedacht wurde (und nicht einfach vergessen), ist das NULL-Statement geeignet.

Es gibt Tools (z.B. Coverage-Tools) die überprüfen, ob alle Fälle auskodiert sind!

Concurrency

PROCESS (1)

- VHDL ist "alt"
- Ein im "Java-Sinne" **statischer non-preemptive Thread** heißt in VHDL **PROCESS**
- Ein VHDL-PROCESS ist aktiv / "handelt"
- Ab jetzt nur noch PROCESS für VHDL-PROCESS

PROCESS (2)

- Ein PROCESS muss nicht erst gestartet werden - dies passiert automatisch sofort. HW ist statisch, PROCESSE modellieren HW-Einheiten, sind also sofort existent
- Ein PROCESS kann sich nur in einer ARCHITECTURE befinden

Simulatorsicht:

- Ein PROCESS ist non-preemptive.
- Der PROCESS selbst bestimmt, wenn er die CPU abgibt.
- Es gibt keine Zeitscheiben
- In "üblicher Denke" hätten alle PROCESSE gleiche Priorität.
Es gibt keine "Thread-Prioritäten" für PROCESSE unter VHDL.
- Sofern ein PROCESS "blockiert", bleibt er "blockiert" bis zur "Aufhebung".
- Typischerweise passiert dies (die "Entblockierung") durch ein für den PROCESS "interessantes" Ereignis. Z.B. wenn ein "Eingangs-Signal" seinen Wert ändert.
- In der **Sensitivity-List** eines PROCESS werden die Eingangssignale aufgeführt bei denen ein Ereignis (eine Änderung des Werts) zur "Aufhebung" einer "Blockierung" führt.

PROCESS

```
[process_label:]  
[POSTPONED] PROCESS [ ( sensitivity_list ) [IS] ]  
    process_declarative_part  
BEGIN  
    process_statement_part  
END [POSTPONED] PROCESS [process-label];
```

```
and2:  
process (I1,I2) is  
begin  
    O <= I1 and I2;  
end process and2;
```

```
clk_10GHz:  
process is  
begin  
    loop                                -- loop .. end loop ist unnötig  
        clk <= '0';  
        wait for 50 ps;  
  
        clk <= '1';  
        wait for 50 ps;  
    end loop;                          -- loop .. end loop ist unnötig  
end process clk_10GHz;
```

```
and2_version1:  
process (I1,I2) is  
begin  
    O <= I1 and I2;  
end process and2_version1;
```

```
and2_version2:  
process is  
begin  
    O <= I1 and I2;  
    wait on I1, I2;  
end process and2_version2;
```

Version1 und Version2 haben identisches Verhalten

Wenn der Code HW modelliert - also synthetisiert werden soll,
dann ist Version1 (mit Sensitivity-List, aber ohne wait) unbedingt vorzuziehen

Version2 (mit wait) ist nur für das Verständnis
wait steht typischer Weise in Code, der nicht für Synthese geeignet ist (hat deswegen entsprechende Signalwirkung)

WAIT

Das WAIT-Statement

- veranlasst eine "Blockierung" des zugehörigen PROCESSES
- muss sich innerhalb des jeweiligen PROCESSES befinden
- schließt (syntaktisch) eine Sensitivity-List aus
- sollte für HW-Code gemieden werden (es gäbe "Auflagen")
- ist für behavioral Testbench-Code (z.B. Stimuli-Generator) sehr geeignet

[*wait_label*:]

WAIT [ON *sensitivity_list*] [UNTIL *condition*] [FOR *time_expression*] ;

WAIT

Beispiele:

-- auf Event warten
wait on clk;

(auf Takt-Ereignis warten)

-- auf's Erfüllen der Bedingung warten
wait until clk='0';

(wenn Takt LowPegel hat, weitemachen)

-- Zeit abwarten
wait for 10 ns;

(für 10 ns warten)

-- "terminieren"
wait;

("halten" / warten ohne Ende)

WAIT

```
process (x) is  
begin  
    ...  
end process;
```

```
process is  
begin  
    ...  
    wait on x;  
end process;
```

- *Bemerkung:*
Die Sensitivitäts-Liste in einem Process ist eine andere Form für ein "WAIT ON" am Ende des process_statement_part

Was passiert?

```
...  
process is  
begin  
    loop  
    end loop;  
end process;  
...
```

Was passiert?

```
...
process is
begin
    loop
        wait for 0 ns;
    end loop;
end process;
...
```

Endlosschleifen

Eine "tödliche" Endlosschleife – in der Endlosschleife wird die CPU nicht abgegeben!

```
process is
begin
    loop
        end loop;
end process;
```

Eine "sehr ärgerliche" Endlosschleife – in der Endlosschleife wird die CPU abgegeben!
Wegen `WAIT` hat Simulator Chance zum Eingriff - die (V-)Zeit schreitet aber nicht voran.

```
process is
begin
    loop
        wait for 0 ns;
    end loop;
end process;
```

Synchronisation

Die Synchronisation in VHDL erfolgt durch das WAIT-Statement

- **WAIT ON** *sensitivity_list*
- **WAIT UNTIL** *condition*
- **WAIT FOR** *time_expression*
- *Bemerkung:*
Die Sensitivitäts-Liste in einem Process ist eine andere Form für ein "WAIT ON"
- *Kombinationen sind möglich, sollten aber gemieden werden (Wartbarkeit)*
- *WAIT FOR TIME'HIGH – NOW;*

"Laufzeiten" - "after"

Signalzuweisungen

- Wert-Zuweisungen an Signale erfolgen mit dem " $<=$ " Signal Assignment
- Wert-Zuweisung an Signale erfolgt immer verzögert

Zweck/Sinn:

- Durch die "Verzögerungen" werden die in HW immer existierenden Laufzeiten nachgebildet
- HW reagiert niemals sofort
- Es gibt zunächst 2 wichtige Arten von Verzögerungen bei einem Signal-Assignment
 - Explizit eingeforderte Verzögerungen als Folge von **after**
 - Implizite Verzögerung um "1 Δ -Zyklus" beim "Weglassen" von **after**
- Der Einsatz von **after** "schützt zunächst" Anfänger vor der "Problematik der Δ -Zyklen"

"after"

Beispiel:

- o `<= i after 1 ns;`

Zweck:

- Mit **after** modellieren Sie Laufzeiten bei der Signalzuweisung.

Einsatzmöglichkeiten:

- Grundsätzliche Visualisierung von Verzögerungen, die nicht den späteren entsprechen.
- Modellierung der späteren zu erwartenden Verzögerung in HW.
Konkret bei der Timing-Simulation nach der Backannotation.
- Achtung! Wenn after fehlt,
 - o `<= i;`beträgt die Verzögerung 1 Δ -Zyklus
- Der Einsatz von after "schützt zunächst" Anfänger vor der "Problematik der Δ -Zyklen"

Beispiel "after" (1)

```
...
entity inv is
    generic (
        delayVisualization : time := 1 ns
    );
    port (
        o : out std_logic;
        i : in  std_logic;
    );
end entity inv;

architecture rtl of inv is
begin

    cobilo:
    process ( i )
    begin
        o <= i after delayVisualization;
    end process cobilo;

architecture rtl;
```

skipped

Beispiel "after" (2)

```
...
architecture ... of ... is
...
component inv
    generic (
        delayVisualization : time := 1 ns
    );
    port(
        o : out std_logic;
        i : in  std_logic
    );
end component inv;
...
begin
...
inv_i : inv
    generic map (
        delayVisualization => delayVisualization
    )                                     -- delayVisualization ist generic der zugehörigen entity
    port map (
        o => o_s,
        i => i_s
    )
;
...
end architecture ...;
```

skipped

Warum kein after ?

- Durch künstliches after entsteht u.U. ein künstlicher kritischer Pfad, der i.d.R. nichts mit dem realen kritischen Pfad zu tun hat.
- Dieser kritische Pfad könnte größer sein als die "Periode" - was zu befremdlichen Ergebnissen und Zeitaufwand bei der Fehler-Suche führen könnte.
- Entwurfs-Fehler, die sich in Kombination mit diesem Effekt "hochschaukeln", werden unangenehmer.

Typische Denke:

- 1. Schritt: Schaltung stimmt (zunächst) logisch/algorithmisches (noch kein Timing)
- 2. Schritt: Timing stimmt (nun mit Backannotation)

Natürlich gibt es Ausnahmen bei besonderen Randbedingungen.

Wie bei Echtzeitsystemen gilt:

"Nur" logisch richtig reicht nicht - Zeitanforderungen müssen eingehalten werden.

Concurrency (2)

Arbeitsbegriffe

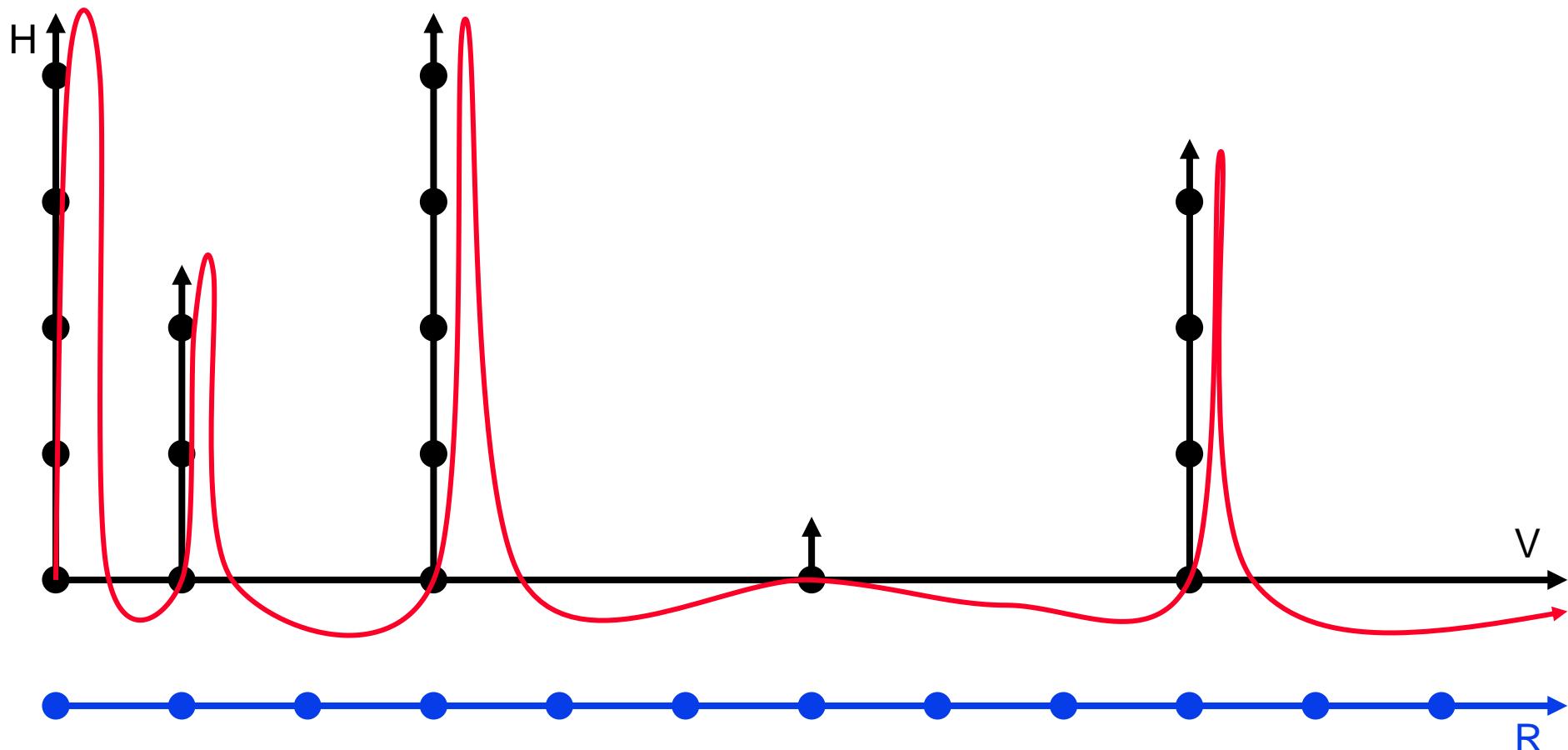
- Die gleich folgenden Begriffe E-, R-, V-, H-Zeit
 - sind Arbeitsbegriffe
 - so also nicht in der Literatur anzufinden
- Wer stehende Begriffe dafür findet, der möge Sie mir bitte mitteilen

VHDL und die Zeit 1

- reale physikalische Zeit
 - die reale Zeit, wie sie von uns erlebt wird
 - gemessen in Sekunden (konkrete Einheit wird je nach Anwendung gewählt)
 - kurz **E-Zeit** (für normale Zeit, so wie wir sie **Erleben**)
- geplante reale physikalische Zeit
 - die reale Zeit, wie sie von uns erlebt werden wird, sofern das Projekt nicht eingestellt oder die Spezifikation modifiziert wird (*andernfalls wird sie nie erlebt ;-)*
 - gemessen in Sekunden (konkrete Einheit wird je nach Anwendung gewählt)
 - kurz **R-Zeit** (für geplant **Reale** Zeit)
- virtuelle physikalische Zeit
 - die direkte Abstraktion der R-Zeit
 - gemessen in Sekunden (konkrete Einheit wird je nach Anwendung gewählt, z.B. ps). Diese Zeit ist diskret - eine eindimensionale Liste von Zeitpunkten.
 - kurz **V-Zeit** (für **Virtuelle** Zeit - die 1.Zeitdimension in VHDL)
- virtuelle "Hilfs-Zeit"
 - wird zur Modellierung der Parallelität benötigt, diese Zeit läuft während eines Zeitpunkts der V-Zeit
 - gemessen in Delta. Dieses Zeit ist diskret (eine eindimensionale Liste von Zeitpunkten)
 - kurz **H-Zeit** (für **Hilfs**-Zeit - die 2.Zeitdimension in VHDL)

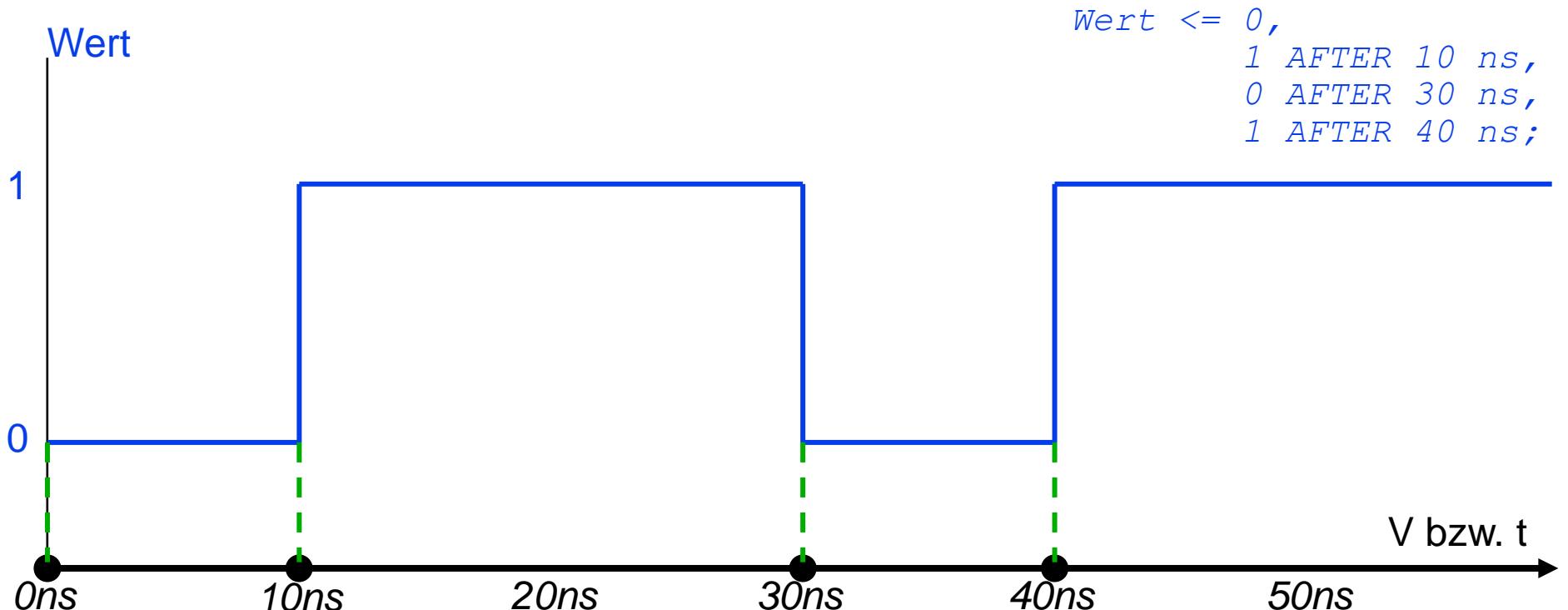
VHDL und die Zeit 2

- zu jedem Zeitpunkt der V-Zeit gibt es genau einen Zeitpunkt der R-Zeit
injektive Abbildung von V nach R aber nicht surjektiv
- zu jedem Zeitpunkt der H-Zeit gibt es genau einen Zeitpunkt der V-Zeit



VHDL und die Zeit 3

- VHDL (ein VHDL-Simulator) ist Ereignis getrieben (*event driven*)
- jedem Ereignis (Transaktion) kann eine Zeitkoordinatenpaar (v, h) zugeordnet werden, z.B. (4510ps, 8 Δ)
- bei der üblichen graphischen Darstellung mit Waves werden die Deltas unterschlagen, es werden die Werte zum jeweils letzten Delta (H-Zeit) eines Zeitpunkts (V-Zeit) dargestellt

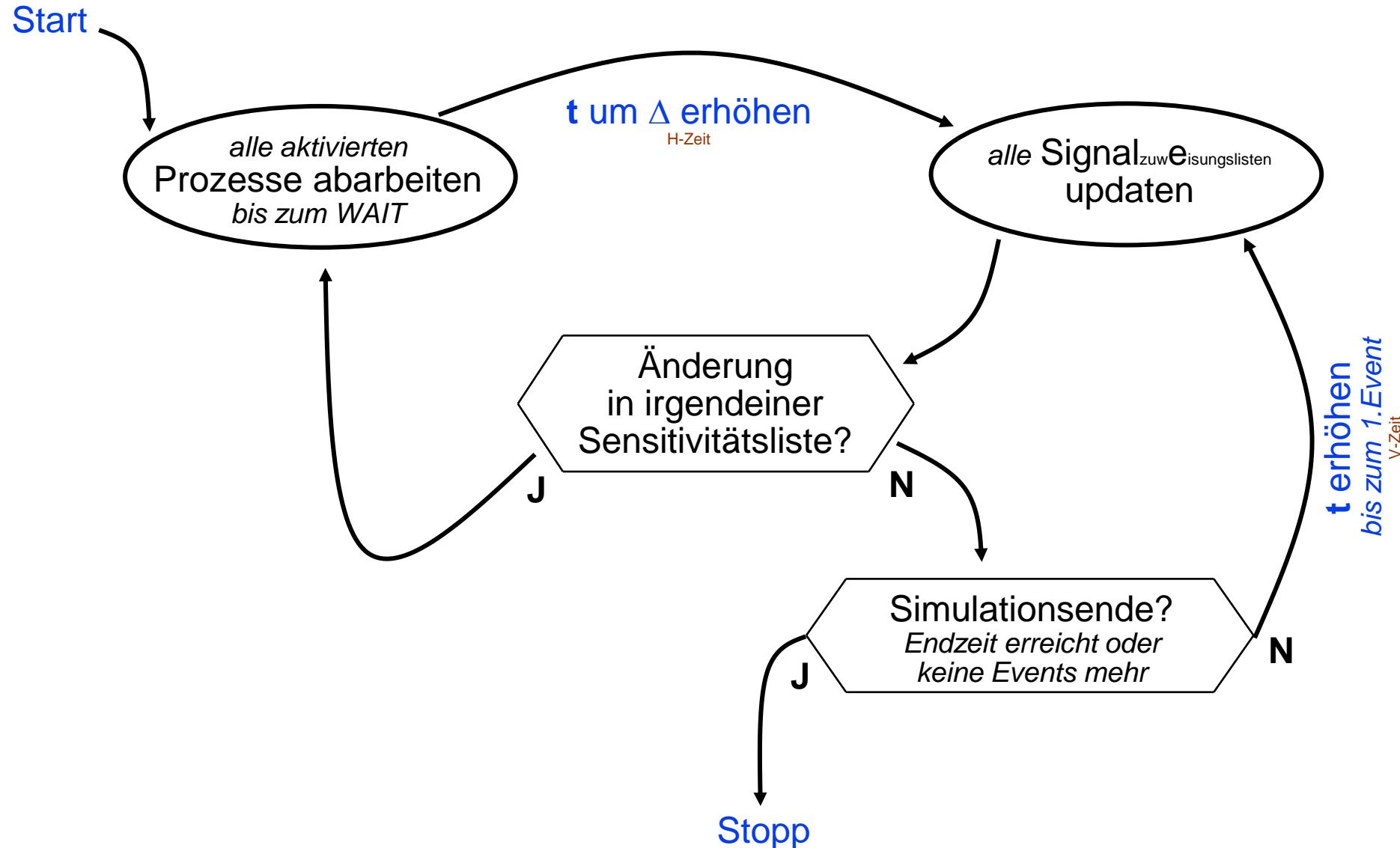


Parallelität und VHDL

- in VHDL passieren "Dinge" entweder
 - sofort in V-&H-Zeit (Variable-Assignment) oder
 - verzögert (Signal-Assignment *und explizites WAIT*)
- zur Realisierung von Parallelität muss auf Signale zurückgegriffen werden
- jede Signalzuweisung löst eine Transaction aus, der ein (V,H)-Zeitkoordinatenpaar zugeordnet wird
Ein Event ist eine Transaction bei der sich was "ändert".
- jede Signalzuweisung "kostet" eine gewisse V/H-Zeit.
 - $x \leq y;$ 0 ns und 1Δ
 - $u \leq v$ AFTER 10 ns; 10 ns
- sofern die Zeit unter VHDL als zweidimensional aufgefasst wird, erfolgt die Berechnung des Wertes einer Signalzuweisung zu einem (V_1, H_1) -Zeitkoordinatenpaar und die Signal-Zuweisung zu einem anderen (V_2, H_2) -Zeitkoordinatenpaar mit:
 $V_1 < V_2 \text{ OR } (H_1 < H_2 \text{ AND } V_1 = V_2)$



VHDL und die Zeit 4 oder das "VHDL-Scheduling"



Prozess-Verarbeitung in VHDL

- alle Prozesse werden parallel ausgeführt
- Prozesse kommunizieren über Signale
- Prozesse werden (*ohne Unterbrechung*) ausgeführt bis sie sich selbst suspendieren (auf WAIT bzw. auf die Sensitivitätsliste* laufen).
VHDL-Prozesse sind non-preemptive!
- Prozesse haben eine Sensitivitätsliste* von Eingangssignalen.
Wenn eine Veränderung (Event) in der Sensitivitätsliste auftritt (Stimulation), wird der zugehörige Prozess (re)aktiviert
- wenn ein Prozess stimuliert wird, reagiert er und wartet danach auf neue Stimuli
- die an ein Signal ausgesendeten Transaktionen werden in einer geordneten Liste gesammelt und heißen **Driver** des Signals. Jeder Transaktion ist ein Bi-Tupel zugeordnet, das den Wert und die Zeit der Transaktion festlegt

* WAITS eingeslossen

Alles klar ? ;-)

- Das VHDL-Simulationsmodell
 - Folie: "VHDL und die Zeit 4 oder das VHDL-Scheduling" ist wichtig für das Verständnis der "Effekte".
- Es erleichtert auf einfache Art die Nachbildung der in der Realität vorhandenen Nebenläufigkeit
- **Das (Simulations-)Ergebnis ist unabhängig vom Scheduling des Simulators!**
Sofern das "System" nicht mit globalen Variablen (Shared Variables) unterlaufen wird.
- Das VHDL-Simulationsmodell macht die korrekte Modellierung der "Nebenläufigkeit" leicht!
- Das VHDL-Simulationsmodell verhindert, dass der Anwender/"Programmierer" bestimmte Fehler macht, die er später vermutlich sehr lange suchen würde.

Routinen

Routinen

- Bereits in PR1 haben "wir" unterschieden zwischen Funktionen und Prozeduren
 - Unter VHDL gibt es diese Keywords
 - Routinen machen primär für behavioral Code Sinn
-
- Zwar lassen sich sogar auch rekursive Routinen mit statischer Rekursionstiefe synthetisieren, aber Routinen sind für synthesefähigen Code exotisch
 - Und Sie wollen auch nicht ewig endlose Diskussionen mit Teamkollegen führen ;-)

```

PROCEDURE name [ (formal_parameter_part) ] IS
    [subprogram_declarative_part]
BEGIN
    [subprogram_statement_part]
END [ PROCEDURE ] [name] ;

```

PROCEDURE

[*procedure_call_label*:]
procedure_name (*actual_parameter_part*) ;

```

procedure swap (
    variable x : inout integer;          -- call by reference
    variable y : inout integer;          -- call by reference
) is
    variable tmp : integer;
begin
    tmp := x;
    x := y;
    y := tmp;
end procedure swap;

...
swap( a, b );

```

```
[ PURE | IMPURE ] FUNCTION name
  [ (formal_parameter_part) ]
RETURN return_type_mark IS
  [ subprogram_declarative_part ]
BEGIN
  subprogram_statement_part
END [FUNCTION] [name] ;
```

FUNCTION (1)

function_name (*actual_parameter_part*) ;

```
function qua (
  constant x : in integer;          -- call by value
) return integer is
  variable resu : integer;
begin
  resu := x*x;
  return resu;
end function qua;
```

...
j := qua(i);

FUNCTION (2)

```
function dig_2_chr (
  constant digit : in integer
) return character is
begin
  case digit is
    when 0      => return '0';
    when 1      => return '1';
    when 2      => return '2';
    when 3      => return '3';
    when 4      => return '4';
    when 5      => return '5';
    when 6      => return '6';
    when 7      => return '7';
    when 8      => return '8';
    when 9      => return '9';
    when others => return 'X';
  end case;
end function dig_2_chr;
```

```
...
c := dig_2_chr( i );
```

FUNCTION (3)

```
-- convert std_logic_vector to string of hex digits
function stv_2_hstr (constant sv : in std_logic_vector) return string is
    variable tmp : std_logic_vector(3 downto 0);
begin
    if (sv'high-sv'low) > 3 then
        -- compute subvector
        tmp := sv(sv'low+3 downto sv'low);
        -- convert subvector
        case tmp is
            when "0000" => return stv_2_hstr(sv(sv'high downto sv'low+4)) & "0";
            ...
            when others => return stv_2_hstr(sv(sv'high downto sv'low+4)) & "?";
        end case;
    else
        -- compute subvector
        tmp := "0000";
        for index in sv'range loop
            tmp(index-sv'LOW) := sv(index);
        end loop;
        -- convert subvector
        case tmp is
            when "0000" => return "0";
            ...
            when others => return "?";
        end case;
    end if;
end function stv_2_hstr;
```

<BREAK>

Ein erstes Mal ENTITY / ARCHITECTURE

ENTITY

- Der "Rahmen" eines "VHDL-Teil-Programms" ist eine ENTITY
- Die ENTITY beschreibt die äußere Sicht - das Interface nach "außen"
- Mit der ENTITY werden die Ein- und Ausgänge festgelegt
- Eine ENTITY hat kein Verhalten

```
ENTITY name IS
  [GENERIC (generic_list) ]
  [PORT (port_list) ]
  [entity_declarative_part]
[BEGIN
  entity_statement_part]
END [ENTITY] [name];
```

ENTITY

```
entity top_level is
end entity top_level;
```

```
entity inverter is
  port (
    o  : out  std_logic;
    i  : in   std_logic
  );
end entity inverter;
```

ARCHITECTURE

- Eine (oder mehrere) ARCHITECTURE(s) ist/sind immer einer ENTITY zugeordnet
- Die ARCHITECTURE beschreibt das Verhalten der ENTITY

```
ARCHITECTURE arc_name OF entity_name IS
    [architecture_declarative_part]
BEGIN
    architecture_statement_part
END [ARCHITECTURE] [arc_name] ;
```

ARCHITECTURE

```
architecture implementation of inverter is
begin

    process ( i ) is
    begin
        o <= not i;
    end process;

end architecture beh;
```

VHDL-Code (Bsp.#1)

```
library work;
use work.all;

library std;
use std.textio.all;

entity firstTest is
end entity firstTest;

architecture beh of firstTest is
begin
    process
        variable wlb : line;
    begin
        loop
            wait for 10 ns;
            write( wlb, string'( "Hallo@" ) );
            write( wlb, now );
            writeline( output, wlb );
        end loop;
    end process;
end architecture beh;
```

läuft nicht mit den "üblichen"/default
Einstellungen von ModelSim

VHDL-Code (Bsp.#2)

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

entity inverter is
  port (
    o : out std_logic;
    i : in  std_logic
  );
end entity inverter;

architecture implementation of inverter is
begin

  process ( i ) is
  begin
    o <= not i;
  end process;

end architecture beh;
```

COMPONENT

- Vergleichbar einem Objekt(Instanz) für dessen Erzeugung die zugehörige Klasse instanziert werden muss,
muss eine ENTITY instanziert werden
- Da HW statisch ist, sind VHDL-Instanzen statisch - eine dynamische Instanziierung hätte kein Gegenstück in HW

Idee

- Soll eine ENTITY e_1 instanziert werden, so muss diese ENTITY e_1 in der jeweiligen anderen ENTITY e_2 zunächst als COMPONENT c_1 deklariert werden, damit sie dann in e_2 verwendet (instanziert) werden kann.
- Gibt es mehr als 1 ARCHITECTURE für e_1 , so kann und muss in einer "Binding Specification" hinter c_1 für jede Instanz von (e_1 bzw. c_1) angegeben werden, welche ARCHITECTURE für die jeweilige Instanz verwendet werden soll
- Tipp: Immer Binding Specification nutzen - die Fehlermeldungen sind "besser"

ENTITY ↔ ARCHITECTURE ↔ COMPONENT

- ENTITY
Definition einer Schnittstelle (black box)
 - ARCHITECTURE
Definition des Verhaltens und/oder der Struktur der ENTITY
(Realisierung der black box)
 - COMPONENT
 - COMPONENT Deklaration
 - COMPONENT Instanziierung
 - bei Instanziierung Zuordnung zu einer konkreten ARCHITECTURE einer ENTITY entspricht physikalischer HW
-
- *ist ENTITY ohne zugehörige ARCHITECTURE oder COMPONENT möglich?
was bedeutet das in HW?*
 - *ist ARCHITECTURE ohne zugehörige COMPONENT möglich?
was bedeutet das in HW?*
 - *ist ARCHITECTURE ohne ENTITY möglich?*
 - *kann eine ENTITY mehrere ARCHITECTUREs haben?*
 - *kann es mehrere COMPONENTs zu einer ENTITY geben?*
 - *müssen alle COMPONENTs einer ENTITY der gleichen ARCHITECTURE zugeordnet sein?*

Example 1

```
ENTITY and2_e IS
  PORT (
    z : OUT std_logic;
    x : IN  std_logic;
    y : IN  std_logic
  );
END ENTITY and2_e;
```

```
ARCHITECTURE and2_a OF and2_e IS
BEGIN
```

and2_p:

```
PROCESS (x, y) IS
```

```
BEGIN
```

```
  z <= x AND y;
```

```
END PROCESS and2_p;
```

```
END ARCHITECTURE and2_a;
```

Die Suffixe e, a, c, i, p dienen nur der Verdeutlichung und sollten **NIEMALS** so im Code verwendet werden

Example 1

```
...
ARCHITECTURE examp_arc OF examp_ent IS
...
SIGNAL a,b,c : std_logic;
...
COMPONENT and2_c IS
  PORT (
    z    : OUT    std_logic;
    x    : IN     std_logic;
    y    : IN     std_logic
  );
END COMPONENT and2_c;
FOR and2_i : and2_c USE ENTITY work.and2_e(and2_a);
...
BEGIN
...
and2_i : and2_c
  PORT MAP (
    z => c,
    x => a,
    y => b
  );
...
END ARCHITECTURE examp_arc;
```

Die Suffixe e, a, c, i, p dienen nur der Verdeutlichung und sollten **NIEMALS** so im Code verwendet werden

Example 2

```
ENTITY and2_e IS
  PORT (
    z_e : OUT std_logic;
    x_e : IN  std_logic;
    y_e : IN  std_logic
  );
END ENTITY and2_e;
```

```
ARCHITECTURE and2_a OF and2_e IS
BEGIN
```

and2_p:

```
PROCESS (x_e, y_e) IS
BEGIN
  z_e <= x_e AND y_e;
END PROCESS and2_p;
```

```
END ARCHITECTURE and2_a;
```

Die Suffixe e, a, c, i, p dienen nur der Verdeutlichung und sollten **NIEMALS** so im Code verwendet werden

Example 2

```
...
ARCHITECTURE examp_arc OF examp_ent IS
  ...
  SIGNAL x,y,z : std_logic;
  ...
COMPONENT and2_c IS
  PORT (
    z_c : OUT std_logic;
    x_c : IN std_logic;
    y_c : IN std_logic
  );
END COMPONENT and2_c;
FOR ALL : and2_c USE
  ENTITY work.and2_e(and2_a)
  PORT MAP (
    z_c => z_e,
    x_c => x_e,
    y_c => y_e
  );
  ...
BEGIN
  ...
  and2_i : and2_c PORT MAP ( z_c => z, x_c => x, y_c => y );
  ...
END ARCHITECTURE examp_arc;
```

Die Suffixe `_e`, `_a`, `_c`, `_i`, `_p`
dienen nur der Verdeutlichung
und sollten **NIEMALS**
so im Code verwendet werden

Example 3

```
ENTITY and2_e IS
  PORT (
    z_e      : OUT    std_logic;
    x_e      : IN     std_logic;
    y_e      : IN     std_logic
  );
END ENTITY and2_e;
```

```
ARCHITECTURE and2_a OF and2_e IS
BEGIN
```

and2_p:

```
PROCESS (x_e, y_e) IS
BEGIN
  z_e <= x_e AND y_e;
END PROCESS and2_p;
```

```
END ARCHITECTURE and2_a;
```

Die Suffixe **e**, **a**, **c**, **i**, **p**
dienen nur der Verdeutlichung
und sollten **NIEMALS**
so im Code verwendet werden

Example 3

```
...
ARCHITECTURE examp_arc OF examp_ent IS
...
COMPONENT and2_c IS
    PORT (
        z_c      : OUT      std_logic;
        x_c      : IN       std_logic;
        y_c      : IN       std_logic
    );
END COMPONENT and2_c;
...
BEGIN
...
SIGNAL x,y,z : std_logic;
...
and2_i : and2_c
    PORT MAP (
        z_c => z,
        x_c => x,
        y_c => y
    );
    ...
END ARCHITECTURE examp_arc;
```

Die Suffixe e, a, c, i, p dienen nur der Verdeutlichung und sollten **NIEMALS** so im Code verwendet werden

Example 3

```
...
CONFIGURATION conf OF examp_ent IS
  FOR examp_arc
    FOR and2_i : and2_c USE
      ENTITY work.and2_e(and2_a)
      PORT MAP (
        z_c => z_e,
        x_c => x_e,
        y_c => y_e
      );
    END FOR;
  END FOR;
END CONFIGURATION conf;
```

Die Suffixe `e`, `a`, `c`, `i`, `p`
dienen nur der Verdeutlichung
und sollten **NIEMALS**
so im Code verwendet werden

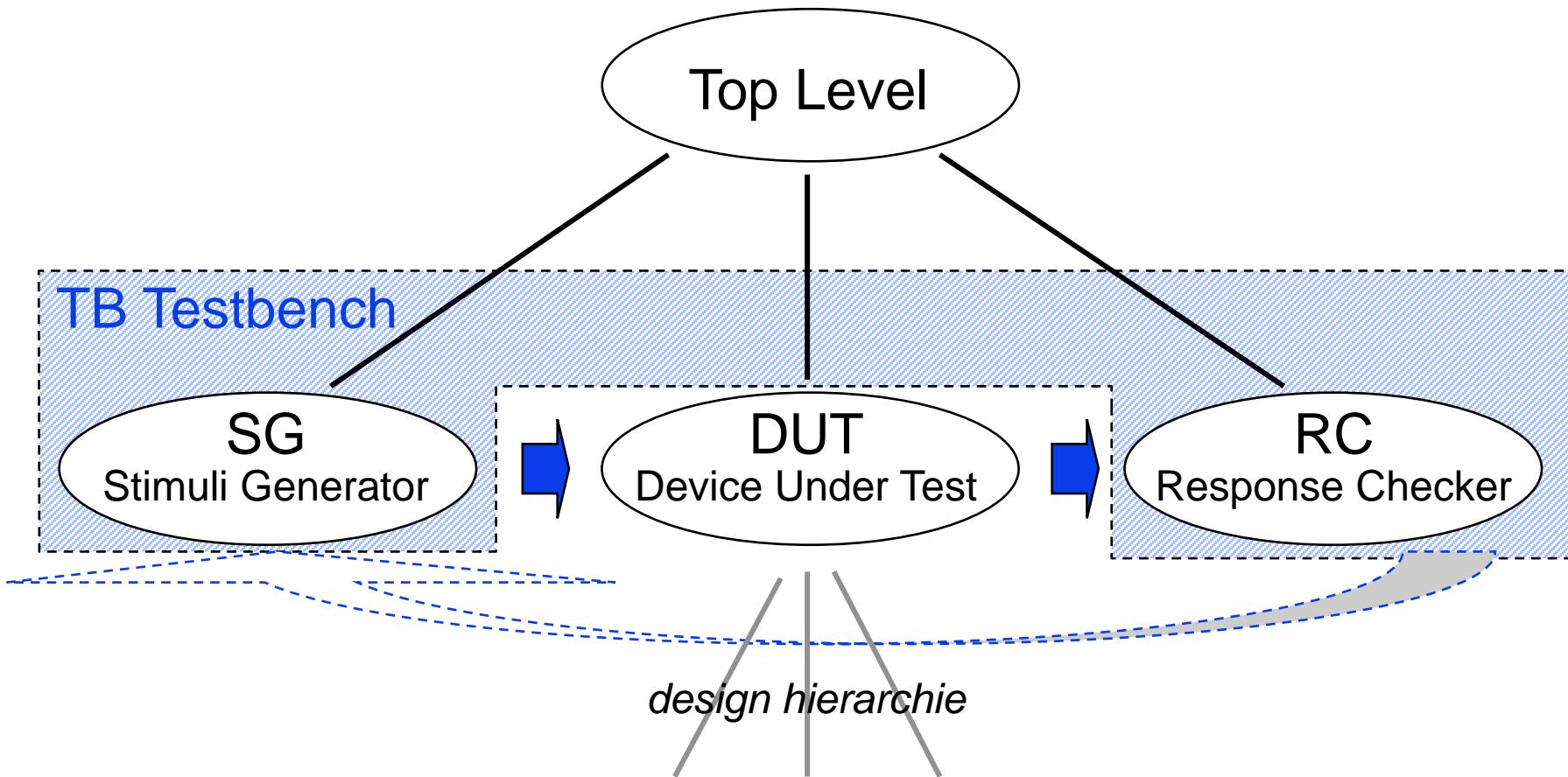
Labor

um Ärger im Labor zu vermeiden sollten

- COMPONENT und ENTITY
- COMPONENT-Ports und ENTITY-Ports

die jeweils gleichen Namen tragen (meist ist dies ohnehin sinnvoll)

Top Level / Testbench



touched

(Sequential) Statements

- *sequential* Signal Assignment Statement
- Variable Assignment Statement
- IF Statement
- CASE Statement
- LOOP Statement
- NEXT Statement
- EXIT Statement
- *sequential* PROCEDURE CALL Statement, RETURN Statement
- *sequential* ASSERTION Statement, REPORT Statement
- WAIT Statement
- NULL Statement

(Concurrent) Statements

- PROCESS Statement
- COMPONENT instantiation Statement
- GENERATE Statement
- concurrent Signal Assignment Statement
- concurrent PROCEDURE CALL Statement
- concurrent ASSERTION Statement
- BLOCK Statement

Parallelität

Parallelität wird in VHDL erzeugt durch die Concurrent Statements:

- Process
- Component Instantiation
- Concurrent Signal Assignment
- Concurrent Procedure Call
- Concurrent Assertion
- *Concurrent Block*

Sie wird "vervielfacht" durch:

- Generate Statement

```
...
L0:
FOR j IN 1 TO 5 GENERATE

    L1:
    IF (j=1) GENERATE
        dff_i_1 : dff_c PORT MAP ( q_c => s(j), d_c => in );
    END GENERATE L1;

    L2_4:
    IF ((1<j) AND (j<5)) GENERATE
        dff_i_2_4 : dff_c PORT MAP ( q_c => s(j), d_c => s(j-1));
    END GENERATE L2_4;

    L5:
    IF (j=5) GENERATE
        dff_i_5 : dff_c PORT MAP ( q_c => out, d_c => s(j-1));
    END GENERATE L5;

END GENERATE L0;
```

kommt später noch mal

touched

"Ende": VHDL als Programmiersprache

- Bis hierhin lag Schwerpunkt der VHDL-Betrachtungen auf den Eigenschaften von VHDL als "Programmiersprache"
- Jetzt in Kombination mit bzw. für HW

Begriffsklärung 1

Schaltnetze

- werden gebildet durch **kombinatorische** Logik
- enthalten **keine** Rückkopplungen (und in der Konsequenz: **keine Zustände**)
- werden in der Praxis auch "kombinatorische Logik" (**combinational logic**) genannt
- sind im mathematische Sinne Funktionen (pure function)
d.h. bei gleicher Eingabe erfolgt gleiche Ausgabe
bzw. bei gleichen Argumenten gibt es gleiches Ergebnis

sequentielle Logik

- repräsentiert einen Zustand (Gedächtnis)
- ein pegelgesteuertes oder einstufiges Flipflop wird in der Praxis meist als **Latch** bezeichnet
d.h. die Zustandsüberführung erfolgt abhängig von einer Bedingung
(Konsequenz: kontinuierliche Zustandsüberführungen während eines Zeitraums)
- ein flankengesteuertes, zweistufiges oder Master-Slave-Flipflop wird in der Praxis meist verkürzt als **Flipflop** bezeichnet
d.h. die Zustandsüberführung erfolgt abhängig von einem Ereignis (Event)

Begriffsklärung 2

Schaltwerke (FSMs)

- werden mit kombinatorischer Logik und Rückkopplungen aufgebaut
- durchlaufen über die Zeit **Zustände** (*state*) bzw. eine Sequenz von Zuständen
- die Zustände werden durch **sequentielle** Logik (*sequential logic*) realisiert
- Schaltwerke werden auch als **FSM** (*finite state machine* bzw. endliche Zustands-Automaten) bezeichnet

FSMs, so wie wir sie wollen,

- nutzen FFs als Zustände.
- Die Zustandsüberführung erfolgt als Konsequenz eines Events (Takt-Flanke)
- Die FSMs sollen sich wie ein Automat verhalten
Eigentlich ist es genau anders herum ;-)
Automaten sind eine Abstraktion einer FSM

Begriffsklärung zur sequentiellen Logik

Latch

- einstufiges Flipflop
- pegelgesteuert (**level sensitive**) i.d.R. vom Takt
- auch: taktzustandsgesteuert aber sogar auch kurz "*Flipflop*" genannt
- korrekt: einstufiges pegelgesteuertes Flipflop
- hier: Kurzform Latch

Flipflop

- zweistufig
- flankengesteuert (**edge sensitive**) i.d.R. vom Takt
- auch: taktflankengesteuert oder aber sogar auch kurz "*Latch*" genannt
- korrekt: zweistufiges flankengesteuertes Flipflop
- hier: Kurzform Flipflop

Achtung

- die Namen Latch und Flipflop sind nicht "genormt". Unterschiedlich Bedeutungen sind möglich (s.o.)

Begriffsklärung zur Synchronität

Synchronisation (*hier, in diesem Zusammenhang*)

- Abstimmung von Aktionen um Gleichzeitigkeit zu erzielen
- Gleichzeitigkeit zu einem speziellen Synchronisationssignal, dem Takt (**clock**)

asynchron

- es liegt keine Gleichzeitigkeit zum Takt vor

synchron

- es liegt Gleichzeitigkeit zum Takt vor

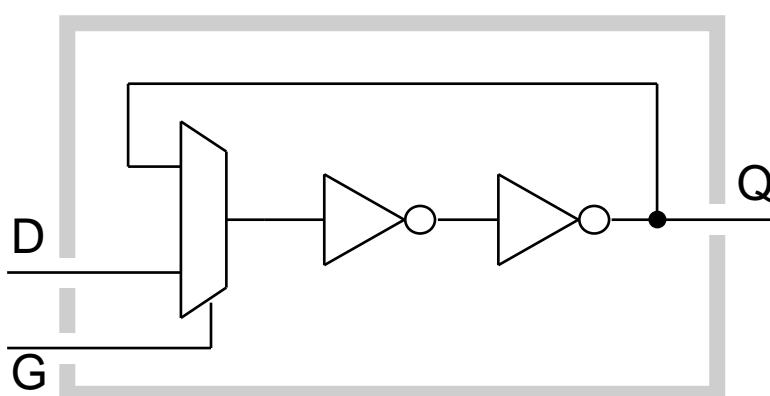
asynchrones Latch

- Latch übernimmt neue Werte unabhängig vom Takt

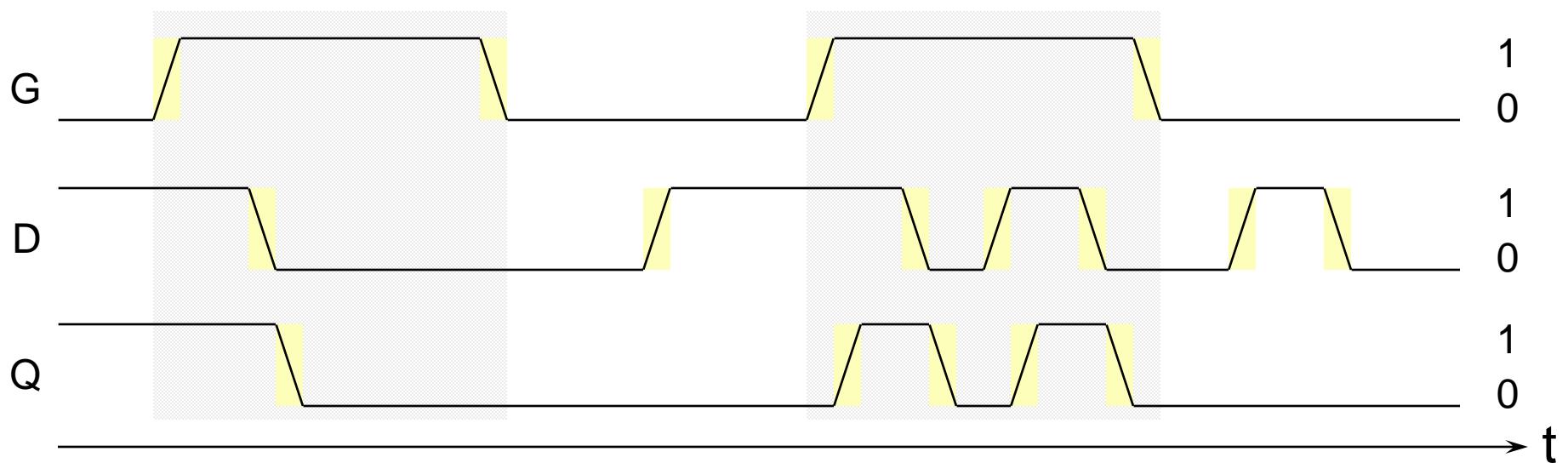
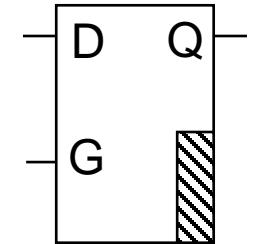
synchrone Latch

- Latch übernimmt neue Werte abhängig vom bzw. synchron zum Takt

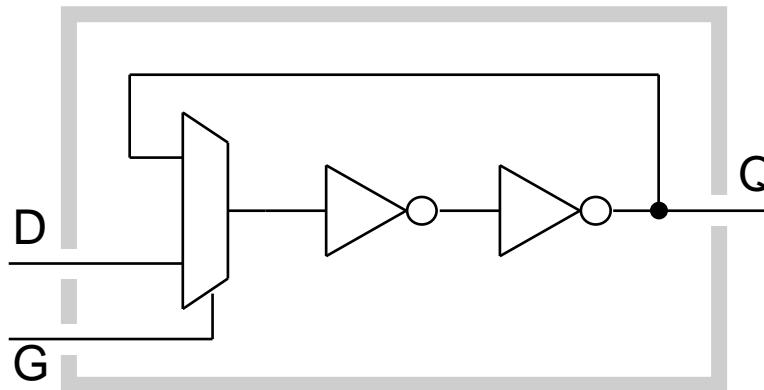
Idee CMOS-Latch



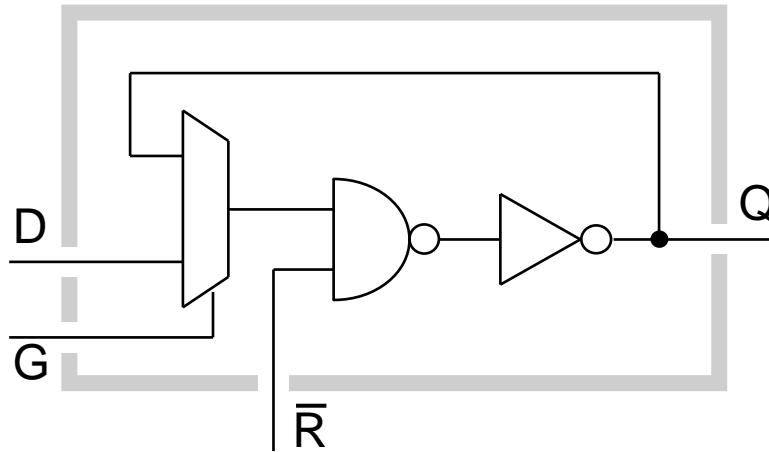
Symbol



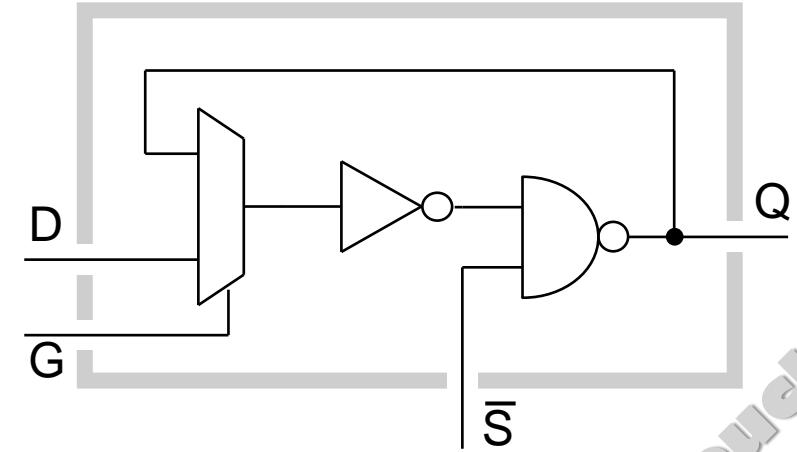
Latch-Varianten I



mit low-active Reset/Clear

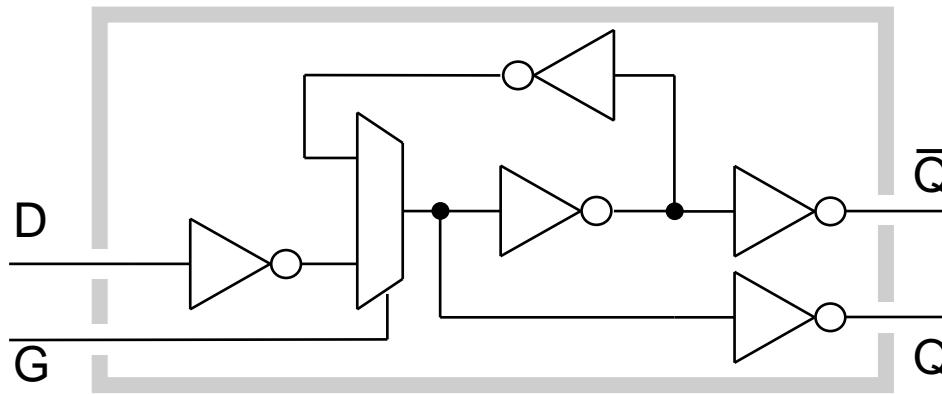


mit low-active (Pre-)Set



touched

Latch-Varianten II

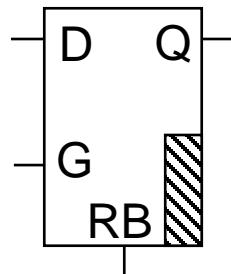


"typisches einfachstes" ASIC-Latch (ohne Reset usw., aber "sicher")

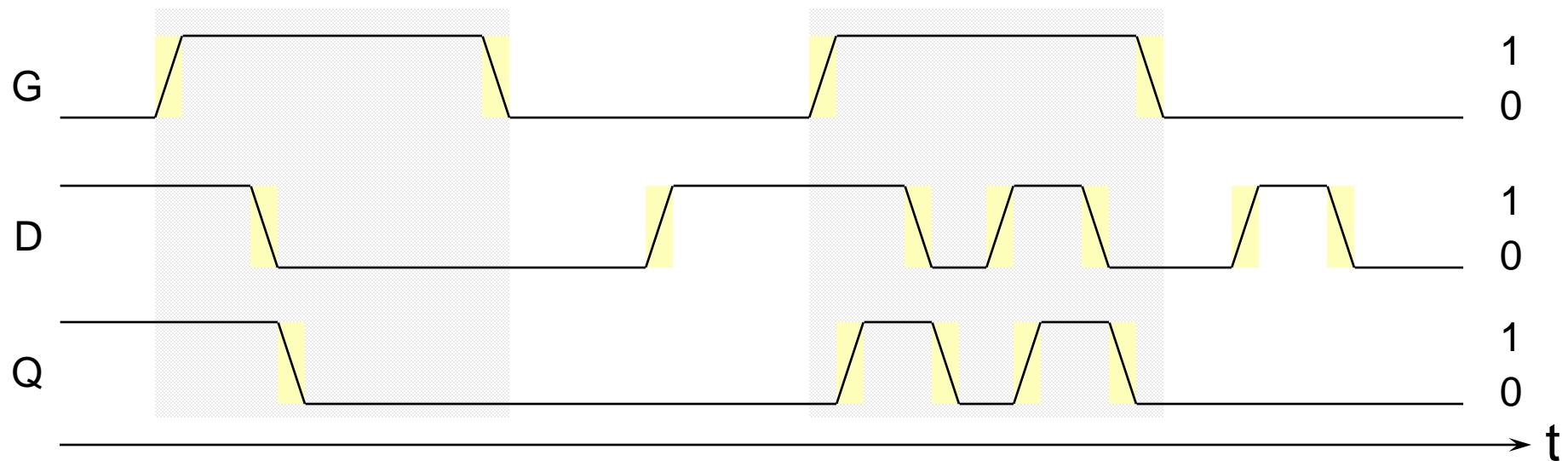
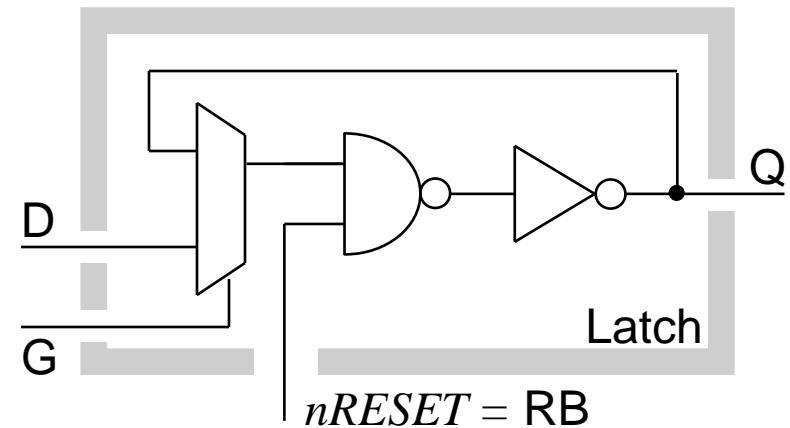
touched

Idee: CMOS-Latch mit low-active Reset

Symbol



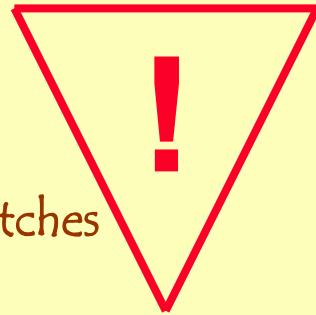
D	G	RB	Q
0	1	1	0
1	1	1	1
x	x	0	0
-- sonst --			Q



Latch (level sensitive)

```
sequlo_latch:  
process (  
    G,          -- Gate  
    nR,         -- Not Reset (amerikanisch RB)  
    D           -- Data  
) is  
begin  
    if  nR='0'   then  
        Q <= '0';           -- reset latch  
    elsif  G='1'   then  
        Q <= D;            -- assign new value  
    end if;  
end process sequlo_latch;
```

Dran denken
Wir wollen keine Latches



Pegel-gesteuerte Latches sollten bis auf seltene Ausnahmen vermieden werden.
Flanken-gesteuerte FFs sind i.d.R. vorzuziehen

Takt

- Der Takt (**Clock** oft als CLK abgekürzt) ist ein wichtiges periodisches Signal
 - Der Takt wechselt zwischen logisch 0 und logisch 1.
Idealisiert passiert
 - der Wechsel (**Flanke**) von 0 nach 1 bzw. 1 nach 0 sofort und
 - die Pegel (0 bzw. 1) "bleiben länger".
- Die Dauer des Zyklus: $0 \rightarrow 1$ -Flanke, 1-Pegel, $1 \rightarrow 0$ -Flanke, 0-Pegel heißt **Takt-Periode**.
Der Kehrwert der Takt-Periode heißt **Takt-Frequenz**.
Der **Duty Cycle** bezeichnet das Verhältnis von der Dauer des 1-Pegels zur Takt-Periode.
- Periode bzw. Frequenz sind konstant. (Was in der Realität leider nur fast stimmt)
 - Zustandsüberführungen synchronisieren sich zum Takt,
bzw. die sequentielle Logik übernimmt die neuen Werte synchron zum Takt.
 - Typisch ist die Zustandsüberführung synchron zur steigenden Flanke (Wechsel: $0 \rightarrow 1$)
(auch positive Flanke genannt).
 - Zustandsüberführungen zu beiden Flanken also sowohl positiv/steigend/ $0 \rightarrow 1$ als auch negativ/fallend/ $1 \rightarrow 0$ ist unüblich (und im Labor unbedingt zu meiden).
 - Wenn alle Zustandsüberführungen in einem Design synchron zur gleichen Taktflanke desselben Takts (üblich ist die steigende Taktflanke) stattfinden spricht man von einem **vollsynchronen Design**.
 - Es gilt die Annahme, dass die Taktflanken "überall" gleichzeitig ankommen.
Was nicht wirklich stimmt - später mehr dazu **Clk-Skew**.

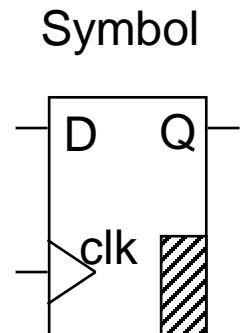
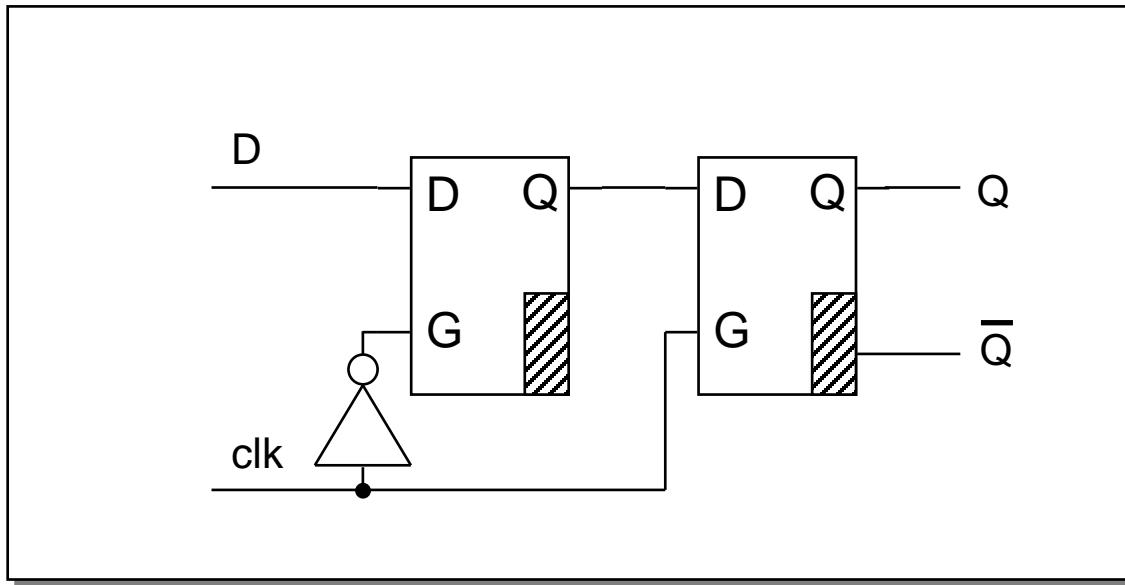
Takt (2)

- FFs "hängen" am Takt. (Der Takt-Eingang eines FF ist am Takt angeschlossen).
- Es ist untersagt den Takt-Eingang eines FFs an etwas anderes als einem Takt anzuschließen.
- Latches sollte nicht verwendet werden
bzw. nur in Ausnahmen die "hier" nicht auftreten
- Der Gate-Eingang eines Latches sollte am Takt angeschlossen werden.
Da Latches in Ausnahme-Situationen eingesetzt werden, gibt es hier häufiger Ausnahmen.

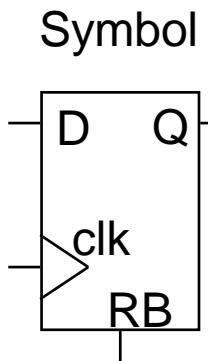
einfaches D-Flipflop

Delay-Flipflop (*D-Flipflop*)

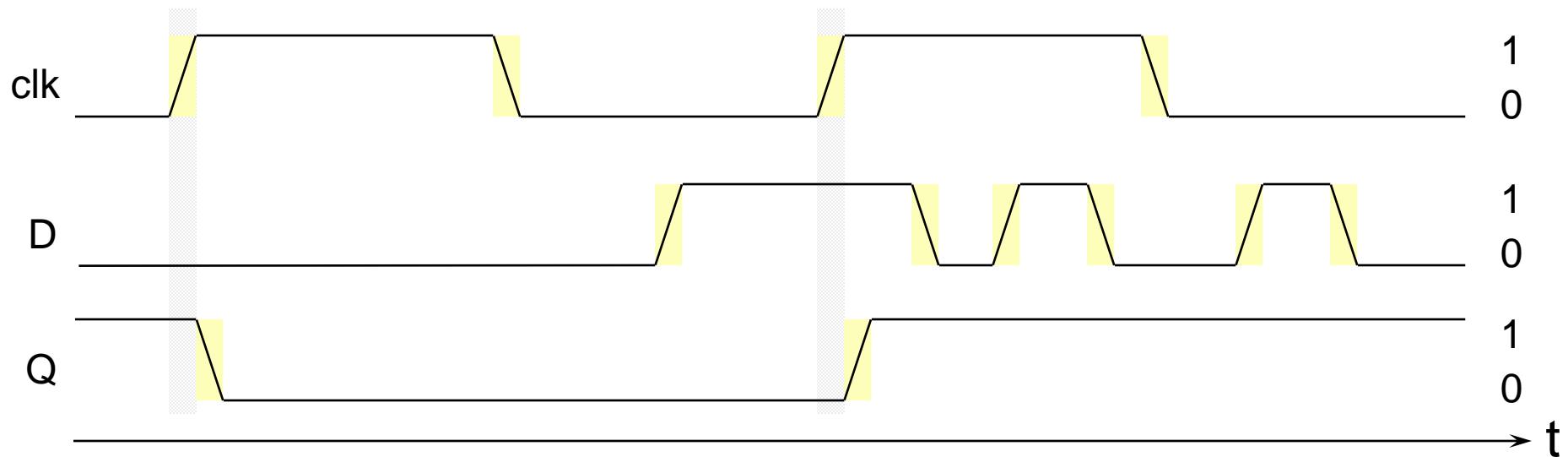
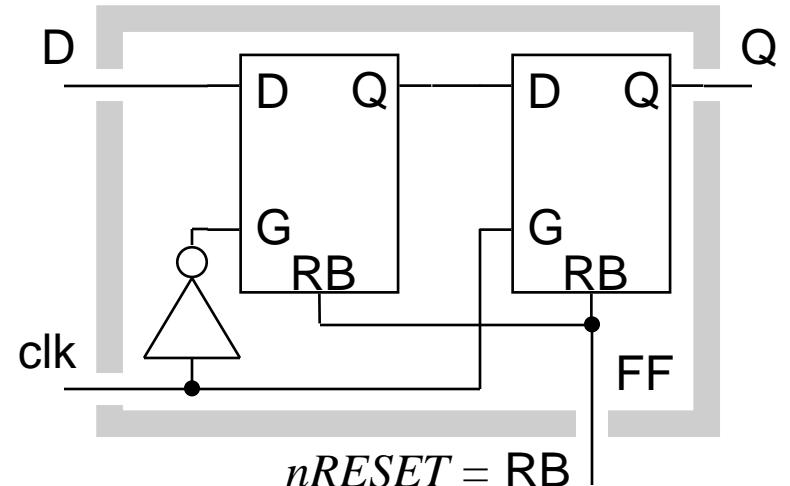
- typische Realisierung als Master-Slave-Flipflop
- Hintereinanderschaltung zweier *pegelgesteuerter* Latches
- synchron *zum Takt (clk)*
- flankengesteuert (edge sensitive)
die Überführung in den neuen Zustand hängt ab von der Takt-Flanke
hier konkret der Wechsel von "0" nach "1" => positiv flankengesteuert



D-FF (*Delay – Flipflop*)



D	clk	RB	Q
0	↑	1	0
1	↑	1	1
x	x	0	0
-- sonst --			Q



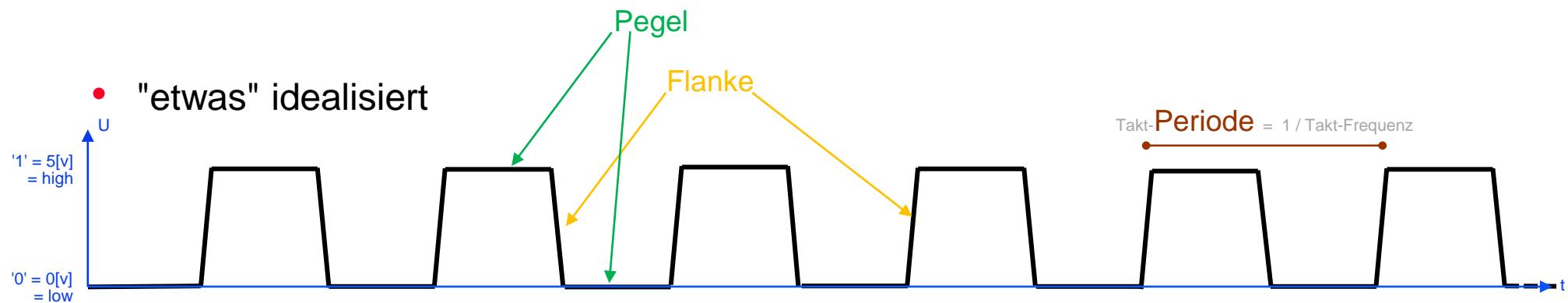
FlipFlop (positive edge sensitive)

```
sequlo_pe_dff:  
process (  
    clk,          -- CLock  
    nR           -- Not Reset  
) is  
begin  
    if  nR='0'  then  
        q <= '0';  
    elsif  clk='1'  and  clk'event  then  
        q <= d;  
    end if;  
end process sequlo_pe_dff;
```

FF bzw. Process ist nicht sensitiv auf **d** !
d steht nicht in der Sensitivity-List !

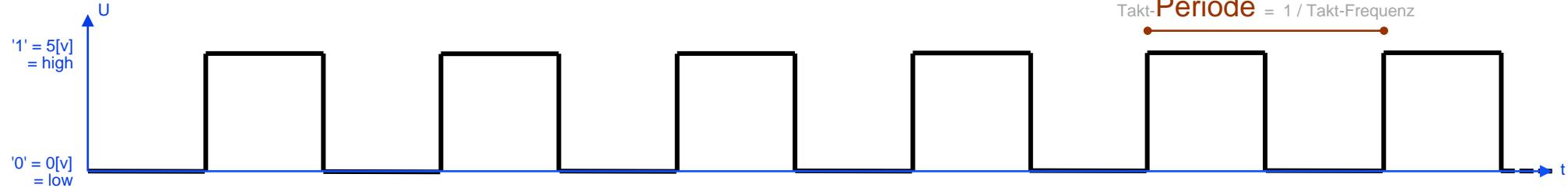
Takt (1)

- "etwas" idealisiert



$$\text{Takt-Periode} = 1 / \text{Takt-Frequenz}$$

- "noch mehr" idealisiert



$$\text{Takt-Periode} = 1 / \text{Takt-Frequenz}$$

- Siehe auch:
<https://de.wikipedia.org/wiki/Taktsignal>

Takt (2)

- Der Takt (**Clock** oft als CLK abgekürzt) ist ein wichtiges periodisches Signal
 - Der Takt wechselt zwischen logisch 0 und logisch 1.
Idealisiert passiert
 - der Wechsel (**Flanke**) von 0 nach 1 bzw. 1 nach 0 sofort und
 - die Pegel (0 bzw. 1) "bleiben länger".
- Die Dauer des Zyklus: $0 \rightarrow 1$ -Flanke, 1-Pegel, $1 \rightarrow 0$ -Flanke, 0-Pegel heißt **Takt-Periode**.
Der Kehrwert der Takt-Periode heißt **Takt-Frequenz**.
Der **Duty Cycle** bezeichnet das Verhältnis von der Dauer des 1-Pegels zur Takt-Periode.
- Periode bzw. Frequenz sind konstant. (Was in der Realität leider nur fast stimmt)
 - Zustandsüberführungen synchronisieren sich zum Takt,
bzw. die sequentielle Logik übernimmt die neuen Werte synchron zum Takt.
 - Typisch ist die Zustandsüberführung synchron zur steigenden Flanke (Wechsel: $0 \rightarrow 1$)
(auch positive Flanke genannt).
 - Zustandsüberführungen zu beiden Flanken also sowohl positiv/steigend/ $0 \rightarrow 1$ als auch negativ/fallend/ $1 \rightarrow 0$ ist unüblich (und im Labor unbedingt zu meiden).
 - Wenn alle Zustandsüberführungen in einem Design synchron zur gleichen Taktflanke desselben Takts (üblich ist die steigende Taktflanke) stattfinden spricht man von einem **vollsynchronen Design**.
 - Es gilt die Annahme, dass die **Taktflanken "überall" gleichzeitig ankommen**.
Was nicht wirklich stimmt - später mehr dazu **Clk-Skew**.

Takt (3)

- FFs "hängen" am Takt. (Der Takt-Eingang eines FF ist am Takt angeschlossen).
- Es ist untersagt den Takt-Eingang eines FFs an etwas anderes als einem Takt anzuschließen.
- Latches sollten nicht verwendet werden
bzw. nur in Ausnahmen die "hier" nicht auftreten
- Der Gate-Eingang eines Latches sollte am Takt angeschlossen werden.
Da Latches in Ausnahme-Situationen eingesetzt werden, gibt es hier häufiger Ausnahmen.

Wdh.: FlipFlop (positive edge sensitive)

```
sequo_pe_dff:  
process (  
    clk,          -- CLock  
    nR           -- Not Reset  
) is  
begin  
    if  nR='0'  then  
        q <= '0';  
    elsif  clk='1'  and  clk'event  then  
        q <= d;  
    end if;  
end process sequo_pe_dff;
```

FF bzw. Process ist nicht sensitiv auf **d** !
d steht nicht in der Sensitivity-List !

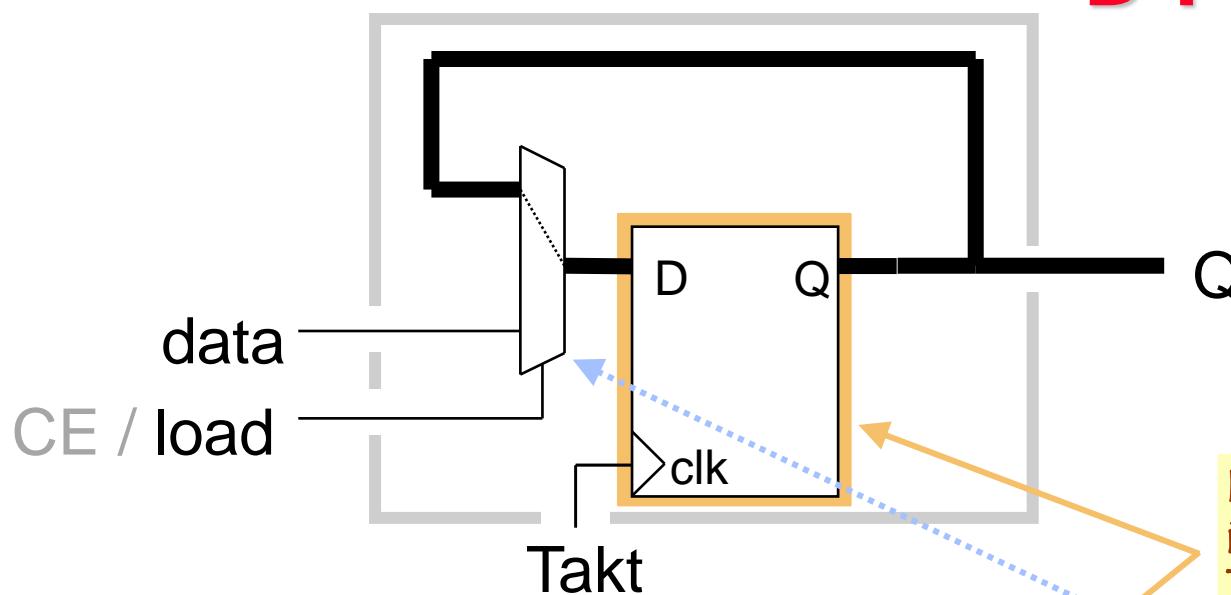
Register (positive edge sensitive)

```
sequo_pe_reg:  
process (  
    clk,          -- CLock  
    nR           -- Not Reset  
) is  
begin  
    if  nR='0'  then  
        q <= (others=>'0');  
    elsif  clk='1'  and  clk'event  then  
        q <= d;  
    end if;  
end process sequo_pe_reg;
```

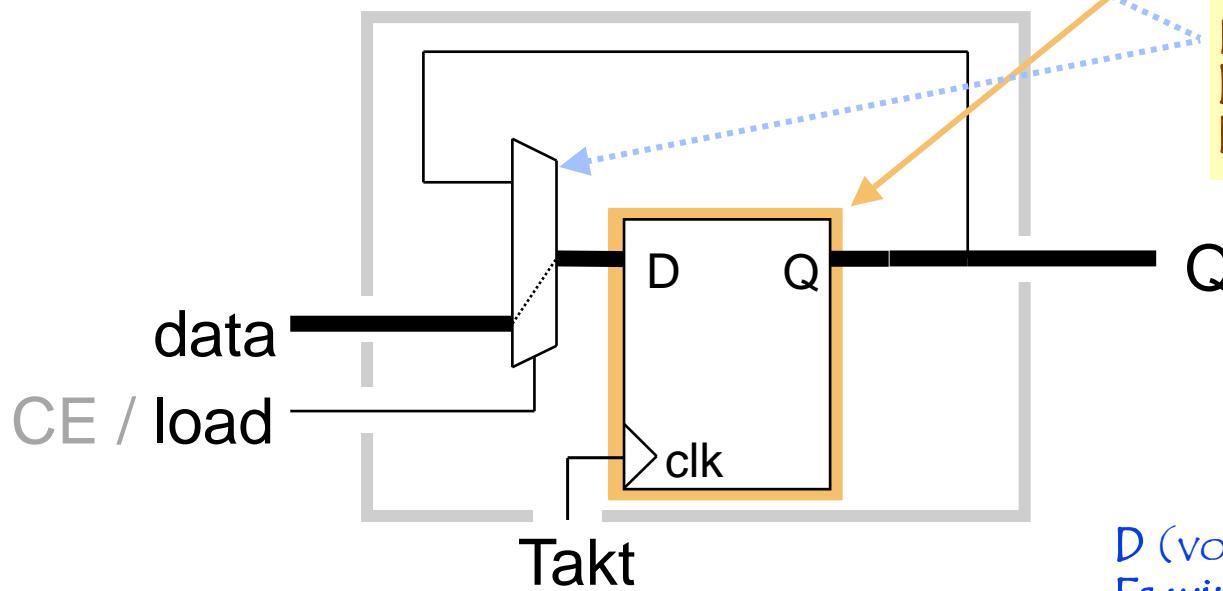
Register bzw. Process ist nicht sensitiv auf **d** !
d steht nicht in der Sensitivity-List !

D-FF erweitert um Load

Speichern



Laden



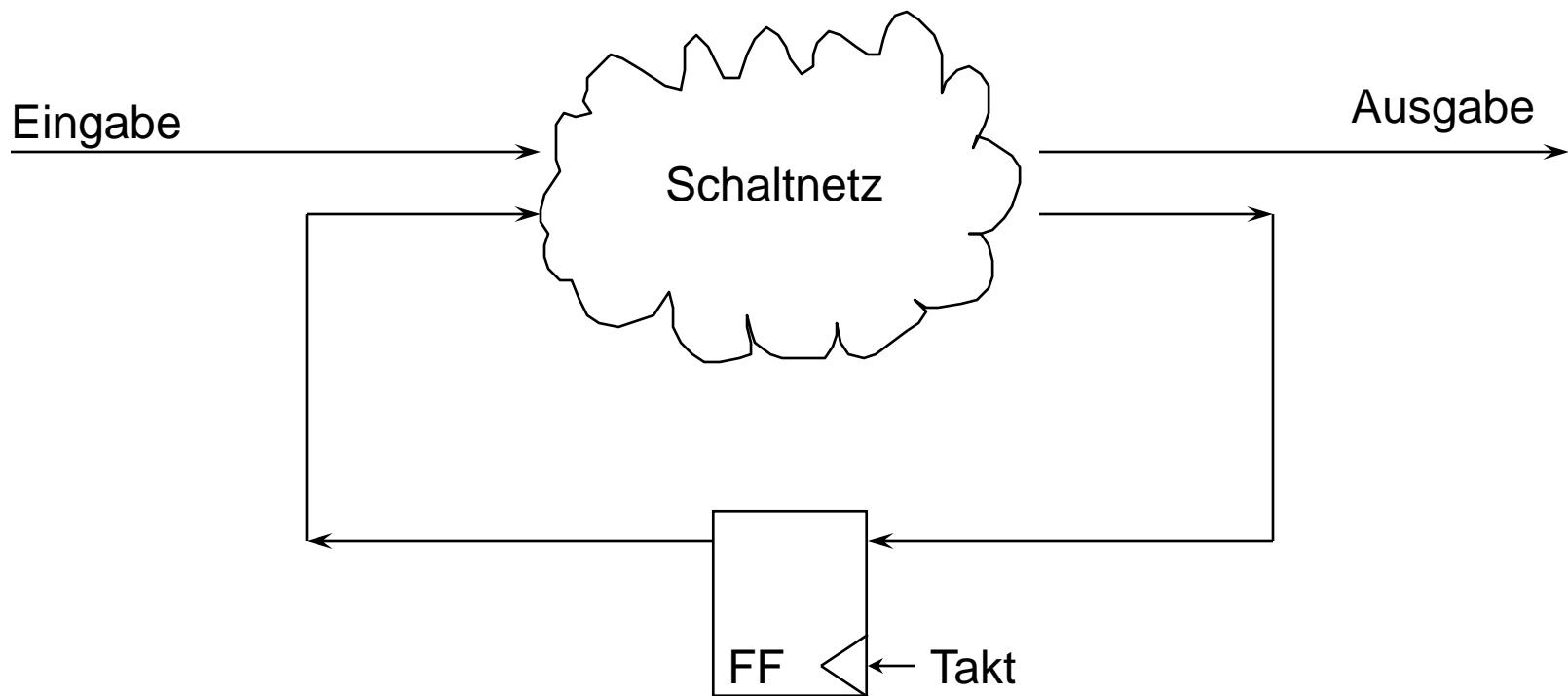
In unserem Coding-Style
ist/wird nur das D-FF
Teil der sequentiellen Logik.

Der Multiplexer ist
bereits Teil der
kombinatorischen Logik

D (von D-FF) wie Delay
Es wird um einen Takt verzögert

Synchrones Schaltwerk / FSM

- Rückkopplung erfolgt über **Abtaktstufe** bzw. sequentielle Logik. Nur diesen Typ von FSM werden/dürfen wir nutzen.



<BREAK>

RESET

- **Reset** ("Rücksetzen") bezeichnet ein Initialisierung-Signal, (das den Startzustand des Automaten herstellt - wichtig für Determinismus).
- Im Zusammenhang mit dem Reset muss unterschieden werden zwischen
 - synchron ↔ asynchron
(zum Takt)
 - low-active ↔ high-active
(bei welchem Pegel wird der Reset/die Initialisierung ausgeführt?)

VHDL-Template

Register/FF mit asynchronem Reset

```
...
selbstErklärenderName:
process ( Takt & Reset ) is
begin
    if Reset then
        Initialisierung(en)
    elsif (gewollte)Taktflanke then
        Zuweisung(en)
    end if;
end process selbstErklärenderName;
...
```

Reset ist asynchron – der Reset wirkt sofort!

Konkreter: asynchroner Reset

```
...
sequolo:
process ( clk, ares ) is
begin
    if ares = '1' then
        reg_cs <= ( others => '0' );
    elsif clk='1' and clk'event then
        reg_cs <= reg_ns;
    end if;
end process sequolo;
...
```

Reset ist asynchron - der Reset wirkt sofort!

Im obigen Beispiel ist Reset high active - bei ares='1' erfolgt "Reset"

"Allgemeines" VHDL-Template

Register/FF mit synchronem (high active) Reset

...

(Signal-)Deklaration der Register-/FF-Signale _cs mit PowerUp-Initialisierung

...

...

selbstErklärenderName:

```
process ( Takt ) is
begin
    if (gewollte) Taktflanke then
        if Reset then
            Initialisierung(en)
        else
            Zuweisung(en)
        end if;
    end if;
end process selbstErklärenderName;
```

...

Reset ist synchron – wie alle anderen Wertänderungen erfolgt Reset synchron zur Taktflanke

Wir implementieren
in ECE immer den
synchronen Reset

!

Konkreter: **synchrone Reset** (high active)

```
...
signal reg_cs : std_logic_vector( ... ) := ( others => '0' );
signal ff_cs  : std_logic := '0';
...
sequlo:
process ( clk ) is
begin
    if clk='1' and clk'event then
        if sres = '1' then
            reg_cs <= ( others => '0' );
        else
            reg_cs <= reg_ns;
        end if;
    end if;
end process sequlo;
...
```

Wir verwenden Xilinx-FPGA
Für Xilinx-Bausteine ist der synchrone Reset die bessere Wahl
Also verwenden wir den synchronen Reset

⇒ GENAU ALLE *_cs-Signale bekommen
die Power-Up-Initialisierung
und den Reset

Reset ist synchron – wie alle anderen Wertänderungen erfolgt Reset synchron zur Taktflanke
Im obigen Beispiel ist Reset high active – bei sres='1' erfolgt "Reset"

Reset

- Unterschied
asynchroner ↔ synchroner Reset
- Wo überhaupt mögliche Probleme?
- Vorteile?
- Nachteile?
- Wie schaut es beim Xilinx-FPGA aus?

touched

Reset

- asynchroner Reset
 - 😊 Reset bei Power-Up leicht realisierbar
 - 😢 kein synchrones Verlassen des Reset
Konsequenz: Die erste Takt-Periode ist zu kurz, Gefahr das Zeit nicht reicht, Ergebnisse nicht fertig, Folgezustand unklar
- synchroner Reset
 - 😢 kein unmittelbarer Reset bei Power-Up
 - 😊 synchrones Verlassen des Reset
- Variante: Reset kommt asynchron und geht synchron
- **Xilinx:**
 - Kein Problem bei Power-Up**
 - Bei Deklaration: `name_cs := initialValue;` initialisiert vor**

touched

Konkreter: **synchrone Reset** (high active)

```
...
signal reg_cs : std_logic_vector( ... ) := ( others => '0' );
signal ff_cs  : std_logic := '0';
...
sequlo:
process ( clk ) is
begin
    if clk='1' and clk'event then
        if sres = '1' then
            reg_cs <= ( others => '0' );
        else
            reg_cs <= reg_ns;
        end if;
    end if;
end process sequlo;
...
```

Wir verwenden Xilinx-FPGA
Für Xilinx-Bausteine ist der synchrone Reset die bessere Wahl
Also verwenden wir den synchronen Reset

⇒ GENAU ALLE *_cs-Signale bekommen
die Power-Up-Initialisierung
und den Reset

Reset ist synchron – wie alle anderen Wertänderungen erfolgt Reset synchron zur Taktflanke
Im obigen Beispiel ist Reset high active – bei sres='1' erfolgt "Reset"

Konkreter: **synchroner Reset** (low active)

```
...
signal reg_cs : std_logic_vector( ... ) := ( others => '0' );
signal ff_cs   : std_logic := '0';
...
sequlo:
process ( clk ) is
begin
    if clk='1' and clk'event then
        if nsres = '0' then
            reg_cs <= ( others => '0' );
        else
            reg_cs <= reg_ns;
        end if;
    end if;
end process sequlo;
...
```

Wir verwenden Xilinx-FPGA
Für Xilinx-Bausteine ist der synchrone Reset die bessere Wahl
Also verwenden wir den synchronen Reset

⇒ GENAU ALLE *_cs-Signale bekommen
die Power-Up-Initialisierung
und den Reset

Reset ist synchron – wie alle anderen Wertänderungen erfolgt Reset synchron zur Taktflanke
Im obigen Beispiel ist Reset low active – bei nsres='0' erfolgt "Reset"

<BREAK>

Einschub: std_logic_1164

```
PACKAGE std_logic_1164 IS
  ...
  TYPE std_ulogic IS (
    'U',   -- Uninitialized           "nicht initialisiert" - Wert darf niemals zugewiesen werden
    'X',   -- Forcing Unknown        "Konflikt"; 0 und 1 treffen aufeinander; Ausgang unklar
    '0',   -- Forcing 0               eine starke/normale 0
    '1',   -- Forcing 1               eine starke/normale 1
    'Z',   -- High Impedance         Hochohmig
    'W',   -- Weak Unknown           Konflikt zwischen schwacher 0 und schwacher 1; Ausgang unklar
    'L',   -- Weak 0                 eine schwache 0 (Pull Down)
    'H',   -- Weak 1                 eine schwache 1 (Pull Up)
    '-'   -- Don't care            NICHT nutzen; good idea gone bad
  );
  TYPE std_ulogic_vector IS ARRAY (NATURAL RANGE <>) OF std_ulogic;

  FUNCTION resolved (s : std_ulogic_vector) RETURN std_ulogic;
  SUBTYPE std_logic IS resolved std_ulogic;
  TYPE std_logic_vector IS ARRAY (NATURAL RANGE <>) OF std_logic;
  ...

```

Einschub: std_logic_1164

```
PACKAGE BODY std_logic_1164 IS
  ...
  TYPE stdlogic_table IS ARRAY(std_ulogic, std_ulogic) OF std_ulogic;
  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' ) -- | - |
);
  ...
  
```

touched

Bemerkung: std_ulogic ↔ std_logic

- std_logic ist "abgeleitet" von std_ulogic
- std_logic hat eine busresolution function und std_ulogic nicht
- bei std_logic kann es mehrere Treiber für ein Signal geben und bei std_ulogic nicht
- die busresolution function von std_logic kostet Simulationszeit
- auf einem üblichen CMOS-Chip darf es keine Tri-State-Busse geben und in der Konsequenz nicht mehrere Treiber, die ein und dasselbe Signal treiben
- std_ulogic verhindert dies auf VHDL-Sprachebene
- Tools (ModelSim) prüfen dies optional auf Toolebene
- Simulationszeit ist bei großen Projekten kritisch
- std_ulogic ist auf VHDL-Sprachebene schneller
- auf Toolebene (ModelSim) werden unnötige busresolution functions aber vermutlich wegoptimiert
- std_logic verträgt sich besser/direkt mit std_logic_arith
- nach der Synthese ist std_logic üblich
- unter unseren spezifischen Randbedingung ist std_logic "angenehmer"

touched

VHDL / Busse

- Busse setzen sich zusammen aus Leitungen (Z.B. Datenbus, Adressbus usw.)
- Unter VHDL heißen die Busse meist VECTOR und sind ARRAYS über den jeweiligen Leitungs-TYPE
 - `bit_vector` ist ein ARRAY über `bit`
 - `std_ulogic_vector` ist ein ARRAY über `std_ulogic`
 - `std_logic_vector` ist ein ARRAY über `std_logic`

Definition (später mehr):

```
...
TYPE std_logic_vector IS ARRAY (NATURAL RANGE <>) OF std_logic;
...
```

Bemerkung / Erneuter Hinweis:

Daumenregel: Im ASIC wird `std_ulogic...` genutzt und auf dem Board `std_logic...` Wir nutzen also eigentlich den falschen TYPE

Busse

Beispiel-Deklarationen:

```
...
signal example : std_logic_vector( 1 to 10 );
signal databus : std_logic_vector( 31 downto 0 );
...
```

- In der Hardware "laufen" Busse vom MSB zum LSB also:

```
signal databus : std_logic_vector( 31 downto 0 );
```

Kurz: Wenn Bus gemeint ist, dann **downto** verwenden.

Leider läuft ein Vector-Literal (sofern seine Ordnung nicht (z.B. durch ein Assignment) spezifiziert ist) von LSB nach MSB - also "**to**".

Wir werden das vermutlich nicht bemerken, weil die Ordnung fast immer durch den Kontext spezifiziert ist.

Bei neueren VHDL-Versionen tritt das Problem verstärkt auf
(in Kombination mit "&"-Operator)

Busse

Beispiele:

```
...
signal  wire      : std_logic;
signal  nibble    : std_logic_vector( 3 downto 0 );
signal  databus   : std_logic_vector( 31 downto 0 );

...
wire <= databus(0);      -- LSB, sonst Bruch mit ungeschriebenen HW-Gesetz
...
wire <= databus(31);    -- MSB, sonst Bruch mit ungeschriebenen HW-Gesetz
...
nibble <= databus( 3 downto 0 );                      -- die unteren 4 Bit
...
nibble <= databus(31) & databus(27) & "00";    -- nur ein Beispiel
...
```

Busse

- Es macht Sinn Dinge, die zusammen gehören zusammenzufassen
- Beispiel: Datenbus anstelle von einzelnen Datenleitungen
- In der Sensitivity-Liste eines PROCESS können auch Busse stehen

example:

```
process ( databus ) is
    ...
begin
    ...
end process example;
```

Der Process ist auf jede Leitung des Datenbusses sensitiv

v sei vom Typ **std_logic_vector(31 DOWNTO 0)**

dann ist

v'LEFT	31	(die linke Grenze)
v'RIGHT	0	(die rechte Grenze)
v'LOW	0	(die untere Grenze)
v'HIGH	31	(die obere Grenze)
v'ASCENDING	FALSE	(da absteigend)
v'RANGE	31 DOWNTO 0	(bzw. v'LEFT,...,v'RIGHT)
v'REVERSE_RANGE	0 TO 31	(bzw. v'RIGHT,...,v'LEFT)
v'LENGTH	32	(da 32 Elemente)

Aggregate

Es gibt verschiedene Arten wie Busse zusammengestellt werden können.

```
variable examp : std_logic;
variable vec1  : std_logic_vector(7 downto 0);
variable vec2  : std_logic_vector(1 TO 4);
variable vec3  : std_logic_vector(31 downto 0);
variable ir    : std_logic_vector(31 downto 0);

...
examp := '0';
...
vec1 := (others => '1');
vec2 := ('1', '0', '1', '0');
vec3 := (0 TO 23 => '0', 24 TO 29 => '1') & "00";
```

<BREAK>

Codierungsphilosophie (1)

- strikte Trennung zwischen kombinatorischer und sequentieller Logik
- in der sequentiellen Logik erfolgen nur Signalzuweisungen und zwar jeweils nur zum Reset und der Taktflanke (bzw. Taktpiegel)
- in der kombinatorischen Logik (Schaltnetz) wird nur auf Variablen gerechnet
 - zunächst werden die Signale auf Variablen umgesetzt
 - dann wird auf Variablen gerechnet
 - dann werden die Ergebnisse über Signale weitergesendet

Tipp: Namen

Kennzeichnung im Namen, ob Variable oder Signal (und was für ein Signal)

- *name_v* **Variable**
- *name_cs* **Clocked signal** kommt unmittelbar aus einem FF
- *name_ns* **New signal / signal to be clocked** geht unmittelbar in ein FF
- *name_s* **Signal (das weder "_cs" noch "_ns" ist)**
- Ports tragen **kein** Suffix

Konsequenz: Z.B. beim synchronen Reset

```
...
signal reg_cs : std_logic_vector( ... ) := ( others => '0' );
...

...
reg:
process ( clk ) is
begin
    if clk='1' and clk'event then
        if sres = '1' then
            reg_cs <= ( others => '0' );
        else
            reg_cs <= reg_ns;
        end if;
    end if;
end process reg;
...
```

Reset ist synchron - wie alle anderen Wertänderungen erfolgt Reset synchron zur Taktflanke
Im obigen Beispiel ist Reset high active - bei sres='1' erfolgt "Reset"

Codierungsphilosophie (2a)

```
exampleIncrementer:  
process ( xi_s ) is  
    variable x_v : integer;  
begin
```

-- Schritt 1: Signale abgreifen und auf Variablen umsetzen

```
x_v := xi_s;
```

-- Schritt 2: auf Variablen Berechnungen durchführen

```
inc_x_v := x_v + 1;
```

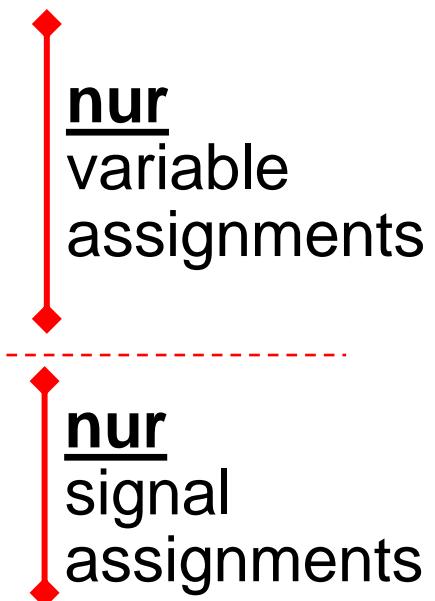
-- Schritt 3: Ergebnisse als Signale versenden

```
inc_s <= inc_x_v;
```

```
end process exampleIncrementer;
```

Alle Berechnungen finden im 2.Schritt statt.

Insbesondere im 3.Schritt gibt es keine Berechnungen, sondern ausschließlich Zuweisungen



Codierungsphilosophie (2b)

```
exampleIncrementer: "Sensitivity-List"
```

```
process ( xi_s )
```

```
    variable x_v : integer;
```

```
begin
```

"rechts"

-- Schritt 1: Signale abgreifen und auf Variablen umsetzen

```
x_v := xi_s;
```

-- Schritt 2: auf Variablen Berechnungen durchführen

```
inc_x_v := x_v + 1;
```

"links"

-- Schritt 3: Ergebnisse als Signale versenden

```
inc_s <= inc_x_v
```

```
end process exampleIncrementer;
```

Signale stehen nur "hier"

Keine Berechnungen

nur
variable
assignments

nur
signal
assignments

Alle Berechnungen finden im 2.Schritt statt.

Insbesondere im 3.Schritt gibt es keine Berechnungen, sondern ausschließlich Zuweisungen

Codierungsphilosophie (2c)

Schritt 1

- Bei Schritt 1 werden alle Signale sichtbar, die gelesen werden – genau diese Signale gehören in die Sensitivity-List.
Stehen sie denn auch alle da?
- Sobald man "sicherer" ist, wird Schritt 1 oft übersprungen, weil dann der Gewinn nur noch gering ist.

Schritt 2

- Da nur auf Variablen gerechnet wird, liest sich dieser Abschnitt wie ein "normales" Programm bzw. "konsequent sequentiell". Keine "Belastung" durch "Effekte der zweidimensionalen Zeit". Dies erhöht Lesbarkeit/Wartbarkeit deutlich
Bis ans Ende von Schritt 2 kann Debugger besser/"normal" unterstützen.

Schritt 3

- Da nur die Ergebnisse mittels Signalen den anderen Prozessen mitgeteilt werden, wird die "Belastung" durch "Effekte der zweidimensionalen Zeit" minimiert.

Kurz:

Die Konsequenz dieser Codierungsphilosophie ist, dass man nie mit Dingen in Kontakt kommt von denen man als Anfänger nichts wissen will.

Codierungsphilosophie (2d)

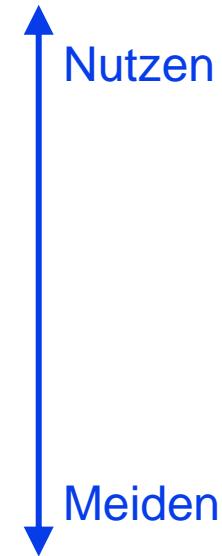
Weitere Konsequenzen (gegenüber "anderen" Codierungsphilosophieren).

- Als Konsequenz der "reduzierten Signal-Menge" wird der Simulator entlastet und die Simulation schneller.
- Viele Simulatoren haben inzwischen "eingebaute Optimierungen" die "Ähnliches machen".
- Die Synthese wird entlastet.
Auch die Synthese muss die Sequenzen bzw. Abhängigkeiten erkennen.
- Der Debugger kann bei der Fehlersuche besser unterstützen.
- Der Designer muss sich "gefühlt" mehr "Gedanken" machen.
Dies täuscht jedoch:
Ohne diese Gedanken entstehen in jeder Codierungsphilosophie Fehler.
Diese Codierungsphilosophie macht die "Gedanken" sichtbar.

Codierungsphilosophie (3)

Kombinatorische Logik

- Bits zusammenstellen (konkatenieren, schieben & "wegwerfen")
- NOT, AND, OR, NAND, NOR
- "=", \oplus XOR
- ADD, SUB, "<", ">"
- MUL
- DIV
- Komplizierteres



Nutzen

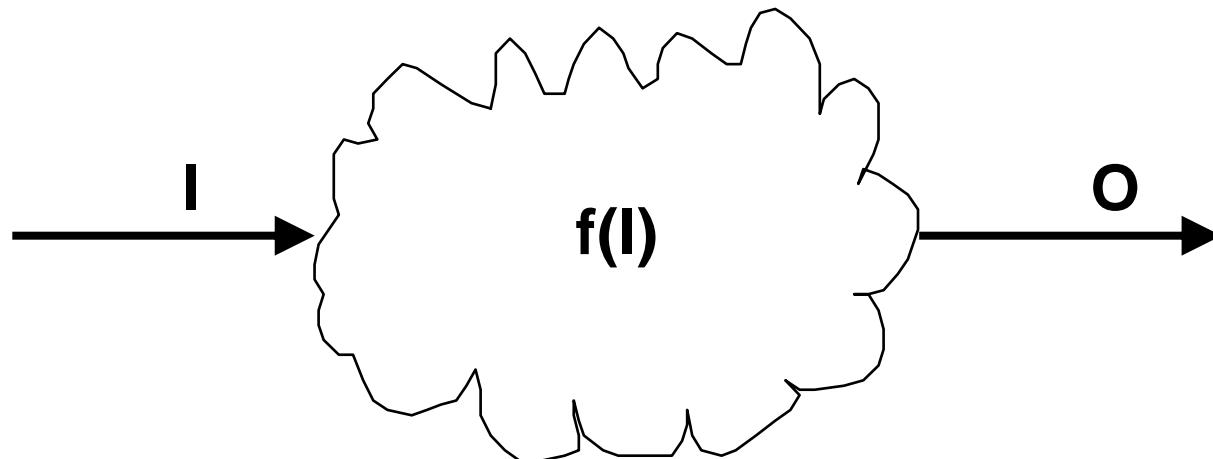
Meiden

Bemerkung

- "=" besteht aus / führt zu Äquivalenz-Gattern
- "<" bzw. ">" besteht aus / führt zu Subtrahierern

Übung

- Welches VHDL Code-Fragment entspricht dem Folgenden?



Lösung (1)

```
...
geeigneter (Process-)Name:
process ( i_s ) is
    variable i_v      : geeigneter Typ ;          -- geeigneter Kommentar
    variable resu_v   : geeigneter Typ ;          -- geeigneter Kommentar
begin
    -- Schritt 1: Signale abgreifen und auf Variablen umsetzen
    i_v := i_s;

    -- Schritt 2: auf Variablen Berechnungen durchführen
    resu_v := f( i_v );                            -- bis hierhin "debug-bar"

    -- Schritt 3: Ergebnisse als Signale versenden
    o_s <= resu_v;
end process geeigneter (Process-)Name;
...
```

Obiges ist nur eine von vielen möglichen Lösungen

Der Process ist Teil einer Architecture, die zu einer Entity gehört
(Im weiteren wird dies nicht mehr extra aufgeführt)

Lösung (2 / alternativ)

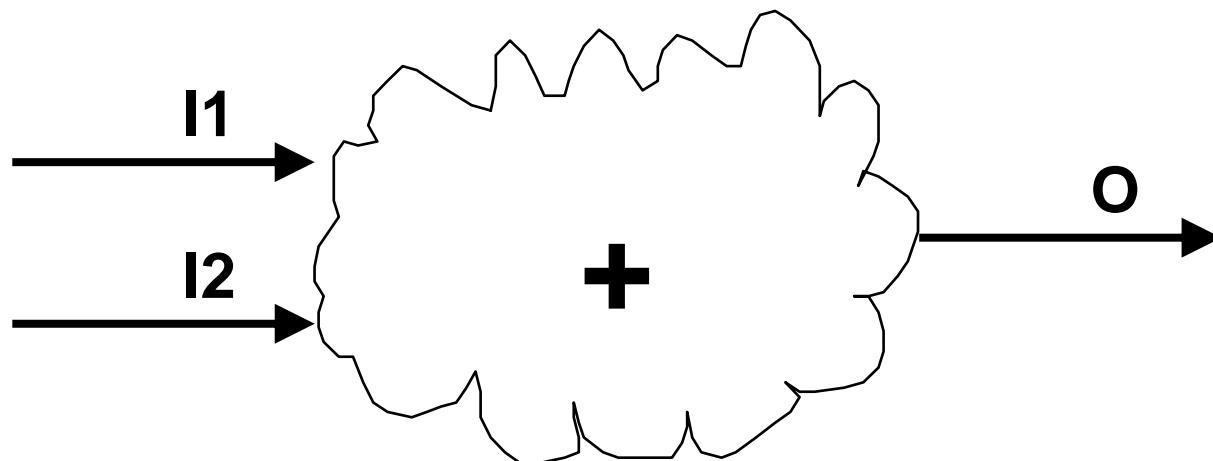
```
...
cl:
process ( i_s ) is
begin
    o_s <= f( i_s );
end process cl;
...
...
```

Obiges ist nur eine von vielen möglichen Lösungen

Der Process ist Teil einer Architecture, die zu einer Entity gehört

Übung

- Welches VHDL Code-Fragment entspricht dem Folgenden?



Lösung (1) ("Warmduscher-Version")

```
...
add:
process ( i1_s, i2_s ) is
    variable i1_v : geeigneter Typ ; -- geeigneter Kommentar
    variable i2_v : geeigneter Typ ; -- geeigneter Kommentar
    variable resu_v : geeigneter Typ ; -- geeigneter Kommentar
begin
    -- Schritt 1: Signale abgreifen und auf Variablen umsetzen
    i1_v := i1_s;
    i2_v := i2_s;

    -- Schritt 2: auf Variablen Berechnungen durchführen
    resu_v := i1_v + i2_v; -- bis hierhin "debug-bar"

    -- Schritt 3: Ergebnisse als Signale versenden
    o_s <= resu_v;
end process add;
...

```

Lösung (2 / alternativ "noch Warmduscher-Version")

```
...
add:
process ( i1_s, i2_s ) is
    variable resu_v : geeigneter Typ ; -- geeigneter Kommentar
begin

    resu_v := i1_s + i2_s;           -- bis hierhin "debug-bar"

    o_s <= resu_v;

end process add;
...
```

Weiterhin klare Trennung zwischen "algorithmischer Berechnung" und "Versenden des Ergebnisses" über Signale

Lösung (3 / alternativ)

```
...
add:
process ( i1_s, i2_s ) is
begin
    o_s <= i1_s + i2_s;
end process add;
...
```

Keine klare Trennung mehr.
"Akzeptabel", wenn alles sinnvoll in eine Zeile passt
und zweifelsfrei keine Fehler enthalten kann.

Diese Art Code zu
schreiben meiden

Lösung (4 / alternativ)

```
...
o_s <= i1_s + i2_s;
...
```

Concurrent Signal Assignment (Zuweisung außerhalb eines Prozesses)

Das Concurrent Signal Assignment befindet sich nicht in einem Prozess.

Es wird "automatisch" der umgebende Prozess erzeugt".

Jedes Concurrent Signal Assignment ist ein eigenständiger Prozess.

Concurrent Signal Assignments meiden (außer es ist bewusst ein eigenständiger Prozess gewollt)

Ein Sequential Signal Assignment (also das "übliche" Signal Assignment)
befindet sich innerhalb eines Prozesses.

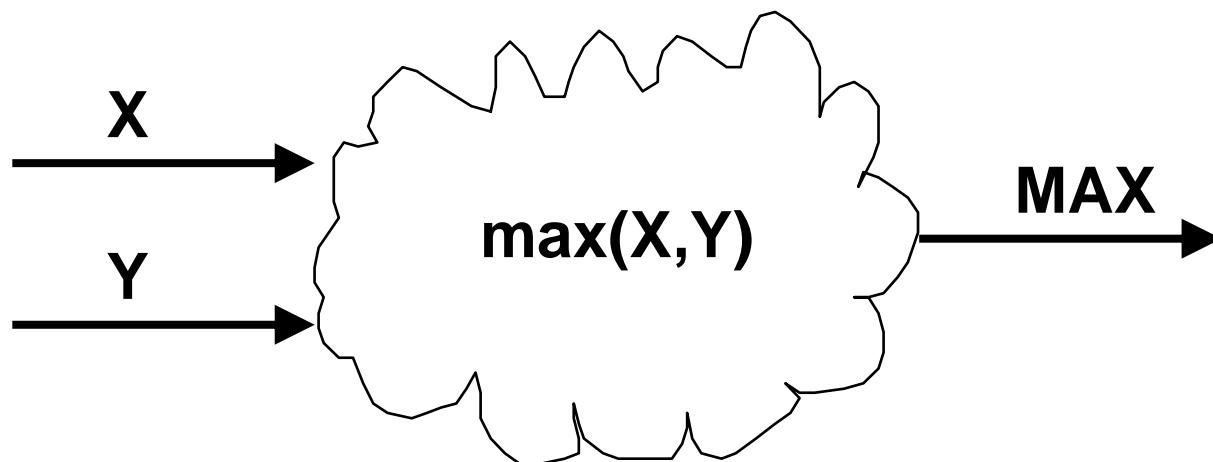
**Diese Art Code zu
schreiben meiden**

"Anfangs"

- Insbesondere am Anfang nur Variante/Lösung 1 oder 2 machen
- Die Anderen erst in Erwägung ziehen, wenn wirklich sicher

Übung

- Welches VHDL Code-Fragment entspricht dem Folgenden?



Lösung ("Warmduscher-Version")

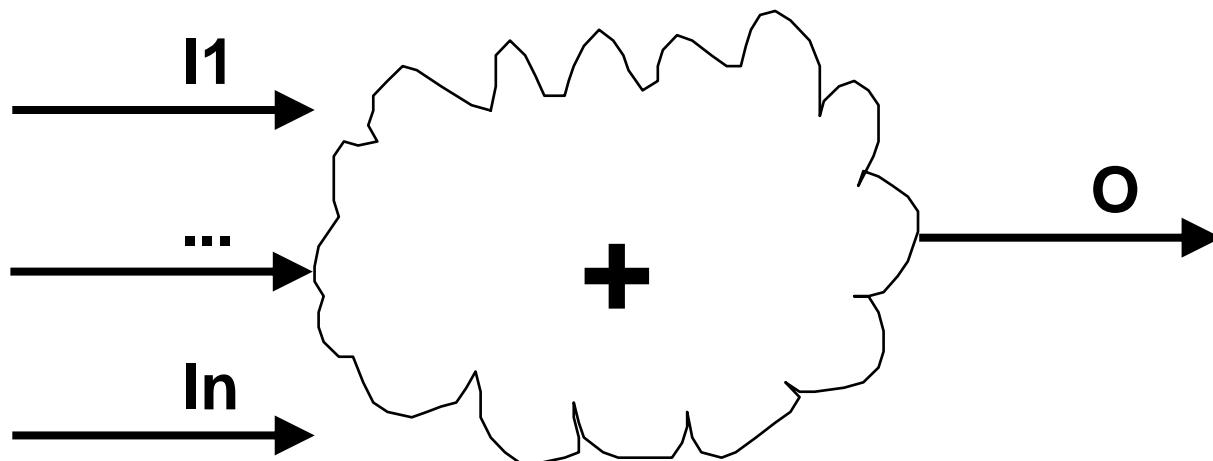
```
...
max:
process ( x_s, y_s ) is
    variable resu_v : geeigneter Typ ;          -- geeigneter Kommentar
    variable x_v      : geeigneter Typ ;          -- geeigneter Kommentar
    variable y_v      : geeigneter Typ ;          -- geeigneter Kommentar
begin
    x_v := x_s;
    y_v := y_s;

    if x_v < y_v  then
        resu_v := y_v;
    else
        resu_v := x_v;
    end if;                                         -- bis hierhin "debug-bar"

    max_s <= resu_v;
end process max;
...
```

Übung

- Welches VHDL Code-Fragment entspricht dem Folgenden?



Lösung ("Warmduscher-Version")

```
...
add:
process ( i1_s, . . . , in_s ) is
    variable i1_v : geeigneter Typ ;           -- geeigneter Kommentar
    ...
    variable in_v : geeigneter Typ ;           -- geeigneter Kommentar
    variable resu_v : geeigneter Typ ;           -- geeigneter Kommentar
begin

    i1_v := i1_s;
    ...
    in_v := in_s;

    resu_v := i1_v + . . . + in_v;           -- bis hierhin "debug-bar"

    o_s <= resu_v;

end process add;
...

```

Lösung (alternativ)

```
...
add:
process ( i1_s, ..., in_s ) is
    variable resu_v : geeigneter Typ ;          -- geeigneter Kommentar
begin
    resu_v := i1_s + ... + in_s;                -- bis hierhin "debug-bar"
    o_s <= resu_v;
end process add;
```

oder -----

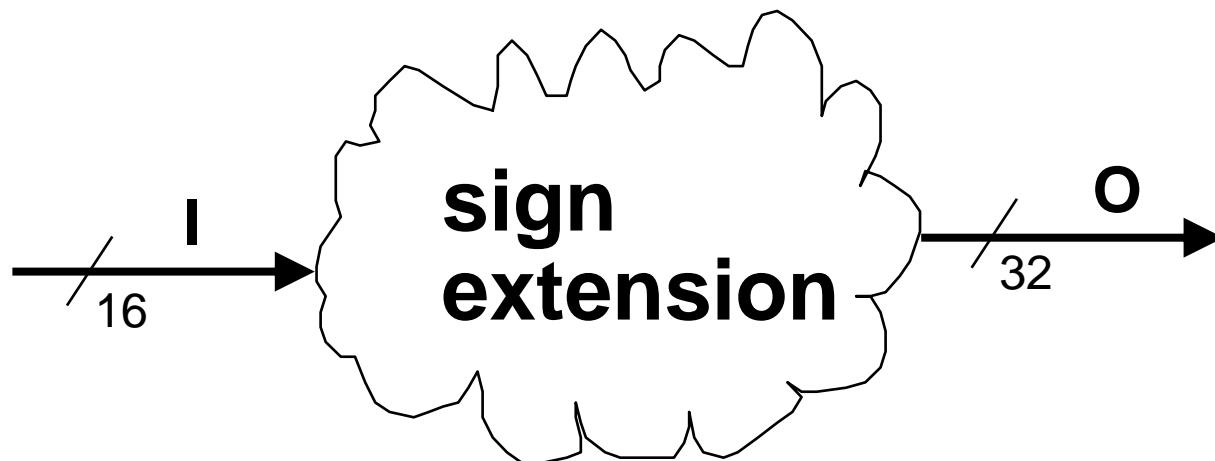
OK bzw. Warmduschercode

```
...
add:
process ( i1_s, ..., in_s ) is
begin
    o_s <= i1_s + ... + in_s;
end process add;
```

Diese Art Code zu schreiben meiden

Übung

- Welches VHDL Code-Fragment entspricht dem Folgenden?



Gegeben sei:

```
signal i_s : std_logic_vector( 15 downto 0 );
signal o_s : std_logic_vector( 31 downto 0 );
```

Lösung ("unüblich")

```
...
se:
process ( i_s ) is
    variable i_v      : std_logic_vector( 15 downto 0 );
    variable msb_v   : std_logic;
    variable resu_v : std_logic_vector( 31 downto 0 );
begin
    i_v := i_s;

    msb_v  :=  i_v( 15 );
    resu_v(31 downto 16) := ( others => msb_v );
    resu_v(15 downto 0)  := i_v;

    o_s <= resu_v;
end process se;
...
```

Lösung (alternativ - ohne "Schritt 1" & "unüblich")

```
...
se:
process ( i_s ) is
    variable resu_v : std_logic_vector( 31 downto 0 );
begin
    resu_v(31 downto 16) := ( others => i_s( 15 ) );
    resu_v(15 downto 0) := i_s;

    o_s <= resu_v;
end process se;
...
```

Lösung

(alternativ - "üblich" & "faul";-)

```
...
se:
process ( i_s ) is
    variable resu_v : std_logic_vector( 31 downto 0 );
begin
    resu_v := ( others => i_s( 15 ) );
    resu_v(15 downto 0) := i_s;

    o_s <= resu_v;
end process se;
...
```

Codierung von FF / Reg mit Load

- Auch in der "Automaten-Denke" entscheidet die Zustandsüberführungs**funktion** ob der Zustand einen neuen Wert annimmt und **nicht** der Zustand selbst.
- Gemäß Coding-Style wird in der kombinatorischen Logik entschieden, ob der Zustand einen neuen Wert annimmt.
- Das Synthesetool wird so besser dabei unterstützt das **jeweils** optimale FF auszuwählen, weil es frei von "Vorgaben" ist.
- Wir müssen und wollen die FFs (spezifischen Gatter) des konkreten Produkts des konkreten Halbleiterherstellers **i.d.R.** gar **nicht** kennen. Andernfalls würden sehr schnell Konflikte mit der Second-Source-Denke erfolgen.
- Wenn wir ein "normales" Programm schreiben, wollen wir auch **nicht** die einzelnen Maschinenbefehle des konkret verwendeten Prozessors kennen und darauf optimieren.

"Allgemeines" VHDL-Template

Register/FF mit synchronem (high active) Reset

...

(Signal-)Deklaration der Register-/FF-Signale _cs mit PowerUp-Initialisierung

...

...

selbstErklärenderName:

```
process ( Takt ) is
begin
    if (gewollte) Taktflanke then
        if Reset then
            Initialisierung(en)
        else
            Zuweisung(en)
        end if;
    end if;
end process selbstErklärenderName;
```

...

Reset ist synchron – wie alle anderen Wertänderungen erfolgt Reset synchron zur Taktflanke

Wir implementieren
in ECE immer den
synchronen Reset

!

Konkreter: **synchrone Reset** (high active)

```
...
signal reg_cs : std_logic_vector( ... ) := ( others => '0' );
signal ff_cs  : std_logic := '0';
...
sequlo:
process ( clk ) is
begin
    if clk='1' and clk'event then
        if sres = '1'  then
            reg_cs  <= ( others => '0' );
        else
            reg_cs  <= reg_ns;
        end if;
    end if;
end process sequlo;
...
```

Wir verwenden Xilinx-FPGA
Für Xilinx-Bausteine ist der synchrone Reset die bessere Wahl
Also verwenden wir den synchronen Reset

⇒ GENAU ALLE *_cs-Signale bekommen
die Power-Up-Initialisierung
und den Reset

Reset ist synchron – wie alle anderen Wertänderungen erfolgt Reset synchron zur Taktflanke
Im obigen Beispiel ist Reset high active – bei sres='1' erfolgt "Reset"

Codierungsphilosophie (1)

- strikte Trennung zwischen kombinatorischer und sequentieller Logik
- in der sequentiellen Logik erfolgen nur Signalzuweisungen und zwar jeweils nur zum Reset und zu der Taktflanke (bzw. Taktpiegel)
- in der kombinatorischen Logik (Schaltnetz) wird nur auf Variablen gerechnet
 - zunächst werden die Signale auf Variablen umgesetzt
 - dann wird auf Variablen gerechnet
 - dann werden die Ergebnisse über Signale weitergesendet

Tipp: Namen

Kennzeichnung im Namen, ob Variable oder Signal (und was für ein Signal)

- *name_v* **Variable**
- *name_cs* **Clocked signal** kommt unmittelbar aus einem FF
- *name_ns* **New signal / signal to be clocked** geht unmittelbar in ein FF
- *name_s* **Signal (das weder "_cs" noch "_ns" ist)**
- Ports tragen **kein** Suffix

Codierungsphilosophie (2a)

```
exampleIncrementer:  
process ( xi_s ) is  
    variable x_v : integer;  
begin
```

-- Schritt 1: Signale abgreifen und auf Variablen umsetzen

```
x_v := xi_s;
```

-- Schritt 2: auf Variablen Berechnungen durchführen

```
inc_x_v := x_v + 1;
```

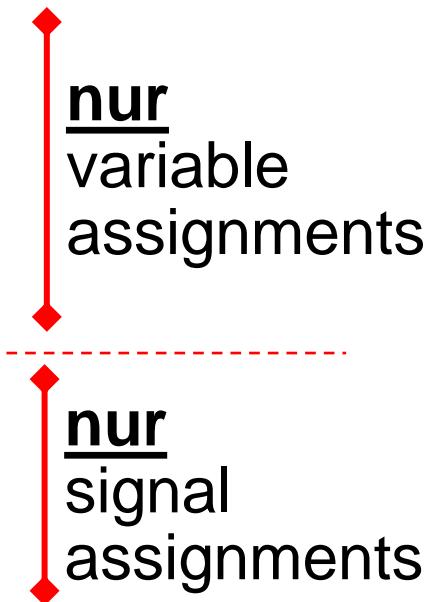
-- Schritt 3: Ergebnisse als Signale versenden

```
inc_s <= inc_x_v;
```

```
end process exampleIncrementer;
```

Alle Berechnungen finden im 2.Schritt statt.

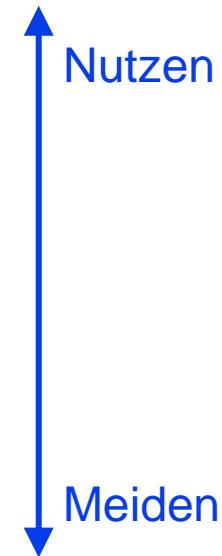
Insbesondere im 3.Schritt gibt es keine Berechnungen, sondern ausschließlich Zuweisungen



Codierungsphilosophie (3)

Kombinatorische Logik

- Bits zusammenstellen (konkatenieren, schieben & "wegwerfen")
- NOT, AND, OR, NAND, NOR
- "=", \oplus XOR
- ADD, SUB, "<", ">"
- MUL
- DIV
- Komplizierteres

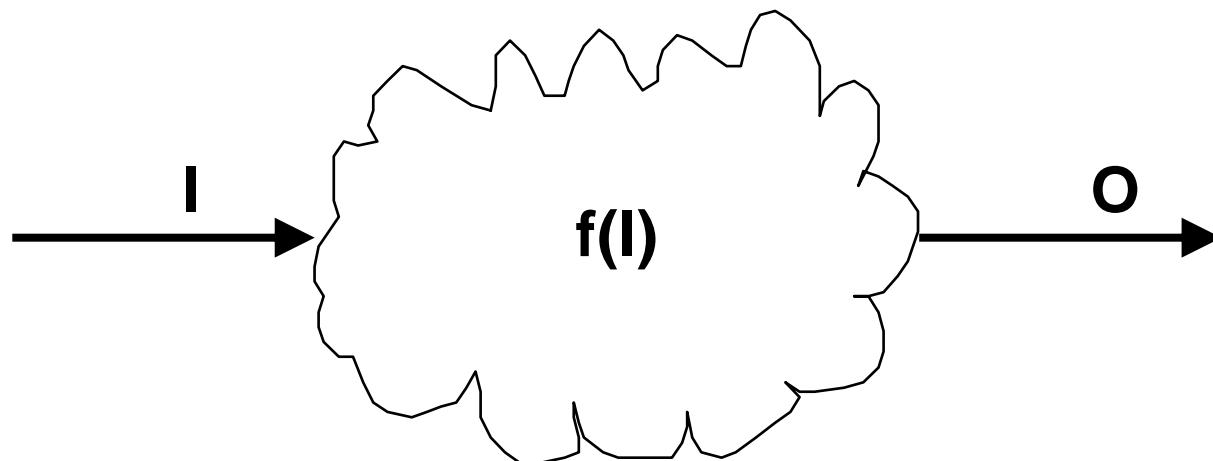


Bemerkung

- "=" besteht aus / führt zu Äquivalenz-Gattern
- "<" bzw. ">" besteht aus / führt zu Subtrahierern

Übung

- Welches VHDL Code-Fragment entspricht dem Folgenden?



Lösung (1)

```
...
geeigneter (Process-)Name:
process ( i_s ) is
    variable i_v      : geeigneter Typ ;          -- geeigneter Kommentar
    variable resu_v   : geeigneter Typ ;          -- geeigneter Kommentar
begin
    -- Schritt 1: Signale abgreifen und auf Variablen umsetzen
    i_v := i_s;

    -- Schritt 2: auf Variablen Berechnungen durchführen
    resu_v := f( i_v );                            -- bis hierhin "debug-bar"

    -- Schritt 3: Ergebnisse als Signale versenden
    o_s <= resu_v;
end process geeigneter (Process-)Name;
...
```

Obiges ist nur eine von vielen möglichen Lösungen

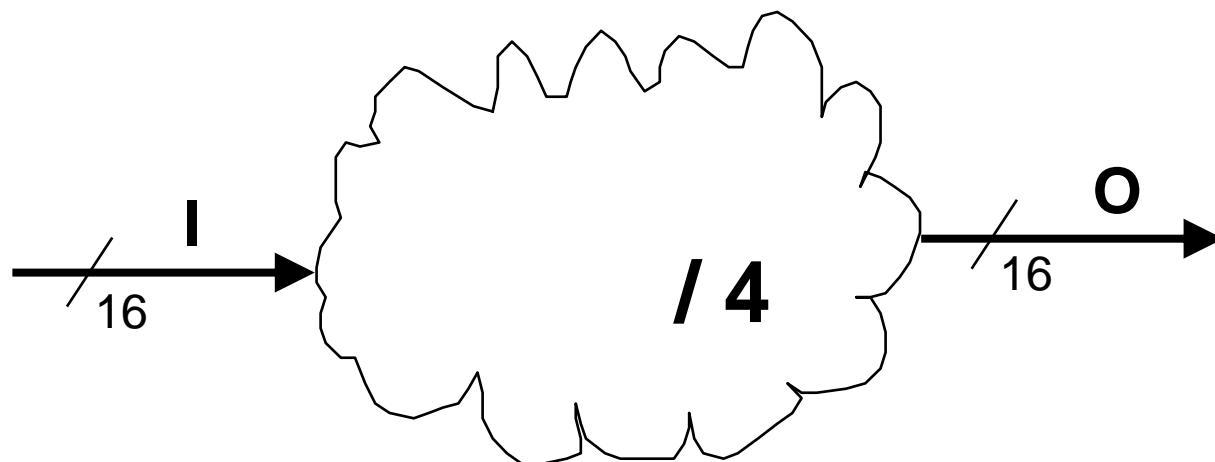
Der Process ist Teil einer Architecture, die zu einer Entity gehört
(Im weiteren wird dies nicht mehr extra aufgeführt)

Alles klar ? ;-)

- Gibt es Fragen zum letzten Mal?

Übung

- Ganzzahlig durch 4 teilen



Gegeben sei:

```
signal i_s : std_logic_vector( 15 downto 0 );
signal o_s : std_logic_vector( 15 downto 0 );
```

Lösung ("unüblich")

```
...
div4:
process ( i_s ) is
    variable i_v      : std_logic_vector( 15 downto 0 );
    variable msb_v    : std_logic;
    variable resu_v   : std_logic_vector( 15 downto 0 );
begin
    i_v := i_s;

    msb_v  :=  i_v( 15 );
    resu_v( 15 downto 14 )  :=  ( others => msb_v );
    resu_v( 13 downto 0 )   :=  i_s( 15 downto 2);

    o_s <= resu_v;
end process div4;
...
```

Lösung (alternativ - ohne "Schritt 1")

```
...
div4:
process ( i_s ) is
    variable msb_v : std_logic;
    variable resu_v : std_logic_vector( 15 downto 0 );
begin
    msb_v := i_s( 15 );
    resu_v( 15 downto 14 ) := ( others => msb_v );
    resu_v( 13 downto 0 ) := i_s( 15 downto 2 );

    o_s <= resu_v;
end process div4;
...
```

Lösung (alternativ)

```
...
div4:
process ( i_s ) is
    variable resu_v : std_logic_vector( 15 downto 0 );
begin
    resu_v( 15 downto 14 )  :=  ( others => i_s( 15 ) );
    resu_v( 13 downto 0 )   :=  i_s( 15 downto 2 );

    o_s <= resu_v;
end process div4;
...
```

Lösung ("üblich" & "faul";-)

```
...
div4:
process ( i_s ) is
    variable resu_v : std_logic_vector( 15 downto 0 );
begin
    resu_v  :=  ( others => i_s( 15 ) );
    resu_v( 13 downto 0 )  :=  i_s( 15 downto 2 );

    o_s <= resu_v;
end process div4;
...
```

Übung

- Erzeuge (im Stimuli Generator) ein periodisches Signal clk mit
 - einer Frequenz vom 1 MHz und
 - einem Duty Cycle von 50%
- Bemerkung: Ein solches Signal heißt Takt

Lösung

```
...
signal clk_s : std_logic;
...
clkgen:
process is
begin
    clk_s <= '0';
    wait for 500 ns;
    clk_s <= '1';
    wait for 500 ns;
end process clkgen;
--  
clk <= clk_s;                                -- Port Wert zuweisen - clk_s intern lesbar
...
...
```

Lösung

```
...
signal clk_s : std_logic;
...
clkgen:
process is
begin
    loop                                -- unnötig(1), aber u.U. besser lesbar
        clk_s <= '0';
        wait for 500 ns;
        clk_s <= '1';
        wait for 500 ns;
    end loop;                            -- unnötig(2), aber u.U. besser lesbar
end process clkgen;
--                                              -- Port Wert zuweisen - clk_s intern lesbar
clk <= clk_s;
...
...
```

Lösung

```
...
signal clk_s : std_logic;
...
clkgen:
process is
begin
    clk_s <= '0';
    wait for sinnvolle Zeit (fullClockCycle);      -- Zeit "hängt" auch vom Reset ab
    loop
        clk_s <= '1';
        wait for 500 ns;
        clk_s <= '0';
        wait for 500 ns;
    end loop;
end process clkgen;
--  
clk <= clk_s;                                -- Port Wert zuweisen - clk_s intern lesbar
...
...
```

Achtung: Beim synchronen Reset - Reset muss eingetaktet werden

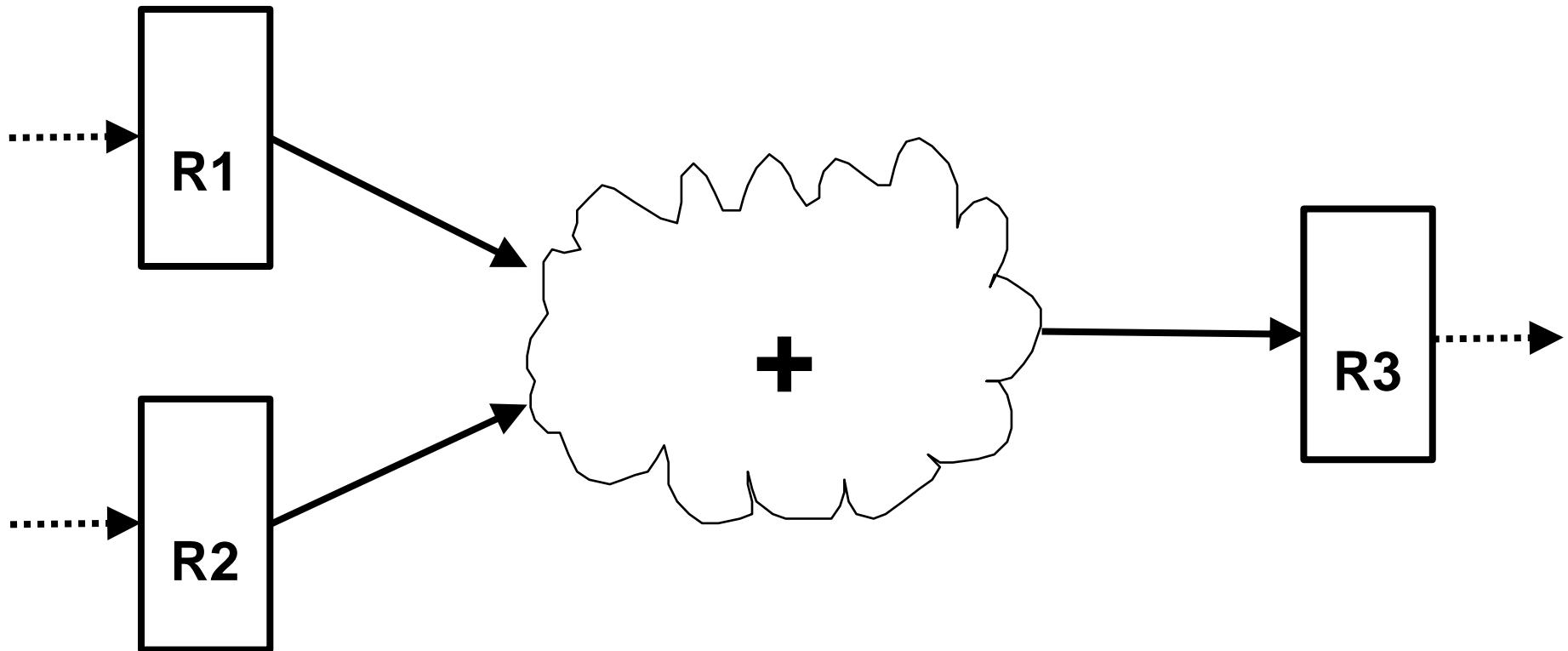
Lösung

```
....  
signal clk_s      : std_logic;  
signal wanted_s : boolean;  
...  
clkgen:  
process is  
begin  
    clk_s <= '0';  
    wait for sinnvolle Zeit;    -- Zeit "hängt" vom Reset ab  
    while wanted_s  loop  
        clk_s <= '1';  
        wait for 500 ns;  
        clk_s <= '0';  
        wait for 500 ns;  
    end loop;  
    wait;  
end process clkgen;  
--  
clk <= clk_s;                      -- Port Wert zuweisen - clk_s intern lesbar  
...  
Im Zusammenhang mit dem Simulatorkommando "run -all" günstiger
```

Übung

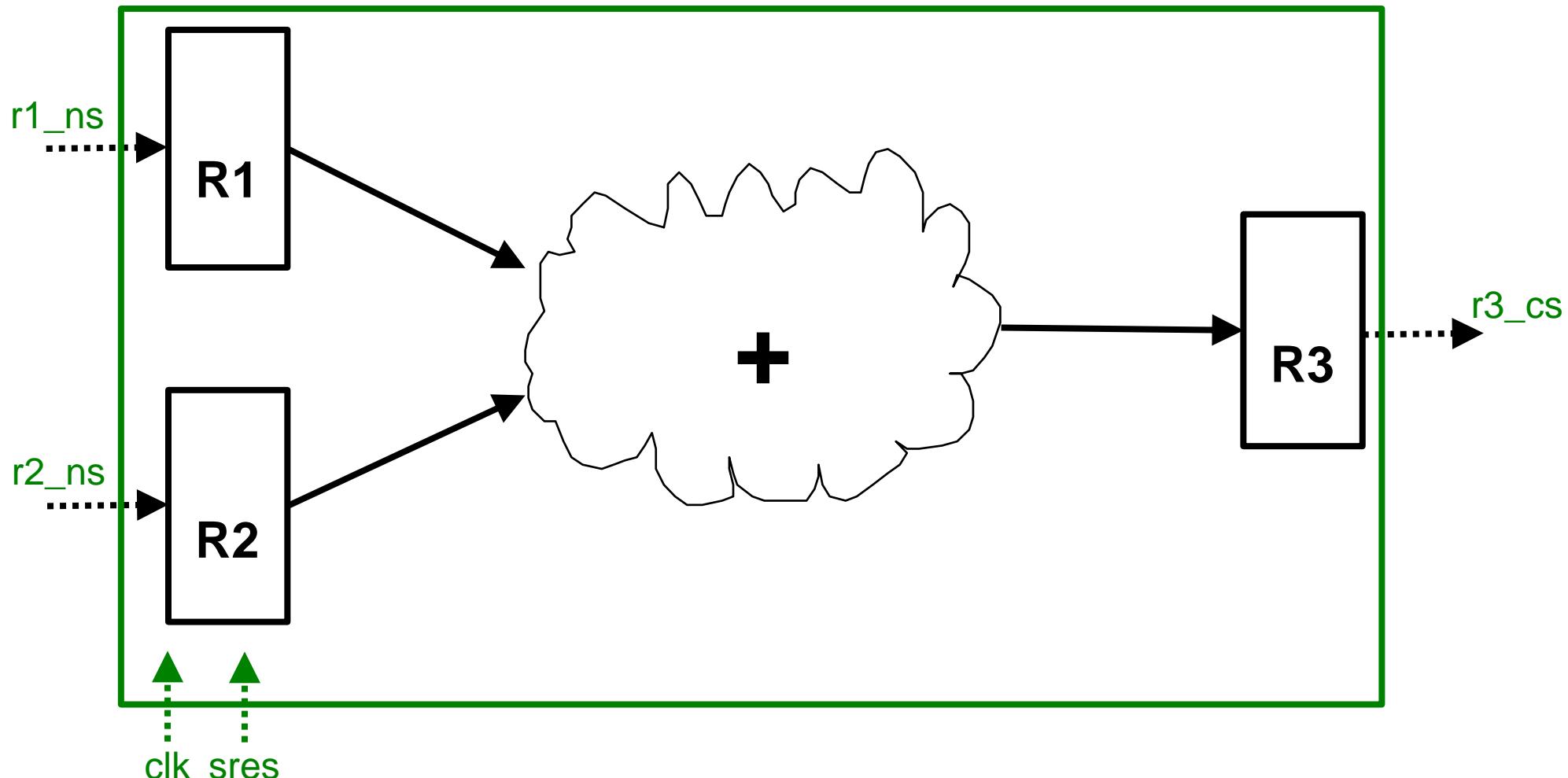
- Welches VHDL Code-Fragment entspricht dem Folgenden?

Die Rechtecke stehen für sequentielle Logik (=Register) und die Wolke für kombinatorische Logik



Übung (2)

- Aus dem zuvor Besprochenen folgt:



Lösung

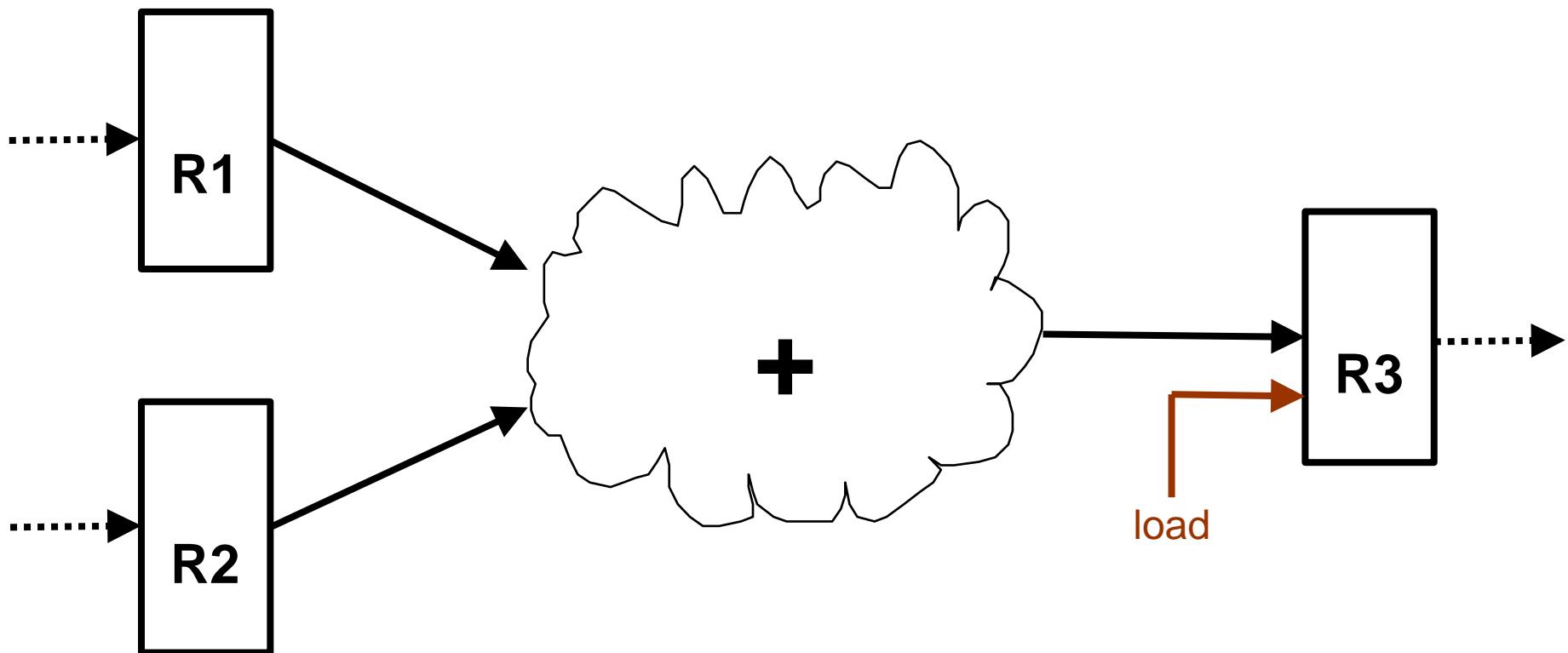
```
...
clk,sres,r1_cs,r2_cs,r3_cs,r1_ns,r2_ns,r3_ns geeignet deklariert und *_cs dabei PowerUp-initialisiert
...
add:
process ( r1_cs, r2_cs ) is
    variable resu_v : geeigneter Typ ;
begin
    resu_v := r1_cs + r2_cs;

    r3_ns <= resu_v;
end process add;

reg:
process ( clk ) is
begin
    if clk = '1' and clk'event then
        if sres = '1' then
            r1_cs <= ( others => '0' );
            r2_cs <= ( others => '0' );
            r3_cs <= ( others => '0' );
        else
            r1_cs <= r1_ns;
            r2_cs <= r2_ns;
            r3_cs <= r3_ns;
        end if;
    end if;
end process reg;
```

Übung

- Welches VHDL Code-Fragment entspricht dem Folgenden?
Register jetzt mit Load (bei Xilinx → CE)

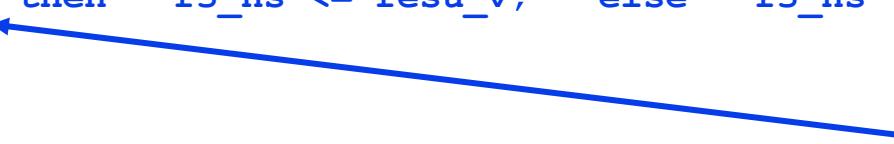


Lösung

```
...
clk,sres,load_s,r1_cs,r2_cs,r3_cs,r1_ns,r2_ns,r3_ns geeignet deklariert und *_cs dabei PowerUp-initialisiert
...
add:
process ( r1_cs, r2_cs, r3_cs, load_s ) is
    variable resu_v : geeigneter Typ ;
begin
    resu_v := r1_cs + r2_cs;

    if load_s = '1' then      r3_ns <= resu_v;    else      r3_ns <= r3_cs;    end if;
end process add;

reg:
process ( clk ) is
begin
    if clk = '1' and clk'event then
        if ( sres = '1' ) then
            r1_cs <= ( others => '0' );
            r2_cs <= ( others => '0' );
            r3_cs <= ( others => '0' );
        else
            r1_cs <= r1_ns;
            r2_cs <= r2_ns;
            r3_cs <= r3_ns;
        end if;
    end if;
end process reg;
```



Der einzige "zulässige" Bruch
mit dem Coding-Sytle
Ausnahme: "Load"

Übung

Beispiel für exponentielle Glättung ($e ::= \text{exponential smoothing}$).
Ein kontinuierlicher Datenstrom soll exponentiell geglättet werden.

- $f(x)_t = 0,5 * x_t + 0,5 * f(x)_{t-1}$
oder alternativ (nur andere Schreibweise)
 $z^*_t = 0,5 * z_t + 0,5 * z^*_{t-1}$

Beispiel:

	x_t	$f(x)_t$	Bemerkung
"init"	0	0	
	16	8	$8 <- 0,5 * 16 + 0,5 * 0$
	32	20	$20 <- 0,5 * 32 + 0,5 * 8$
	64	42	$42 <- 0,5 * 64 + 0,5 * 20$
	46	44	$44 <- 0,5 * 46 + 0,5 * 42$
	24	34	$34 <- 0,5 * 24 + 0,5 * 44$
	88	61	$61 <- 0,5 * 88 + 0,5 * 34$

A vertical green arrow labeled "action" points downwards, indicating the progression of time from top to bottom. The letter "t" is at the bottom of the arrow.

Übung

Beispiel für exponentielle Glättung ($e ::= \text{exponential smoothing}$).
Ein kontinuierlicher Datenstrom soll exponentiell geglättet werden.

- $f(x)_t = 0,5 * x_t + 0,5 * f(x)_{t-1}$
oder alternativ (nur andere Schreibweise)
 $z^*_t = 0,5 * z_t + 0,5 * z^*_{t-1}$
- x kommt mit 32 Bit Breite rein (bzw. Alles soll konsequent 32 Bit groß sein).
Ein mögliches Verlassen des Wertebereichs wird ignoriert
bzw. es wird zugesichert, dass dies niemals passiert.
Ebenso sind Rundungsfehler bei "0,5*" unkritisch
"Integer-Division"s-Genauigkeit reicht.

Zunächst:

- Wie sieht die "Idee" aus ?

Dann:

- Wie sieht der Code aus ?
- Vorschlag: **x** als x , **f** als $f(x)$ und **e** als Ergebnis, das abgeliefert wird

Übung - "Wie sieht Idee aus?"

Beispiel für exponentielle Glättung ($e ::= \underline{\text{exponential smoothing}}$). Ein kontinuierlicher Datenstrom soll exponentiell geglättet werden.

- $f(x)_t = 0,5 * x_t + 0,5 * f(x)_{t-1}$
oder alternativ (nur andere Schreibweise)
 $z^*_t = 0,5 * z_t + 0,5 * z^*_{t-1}$

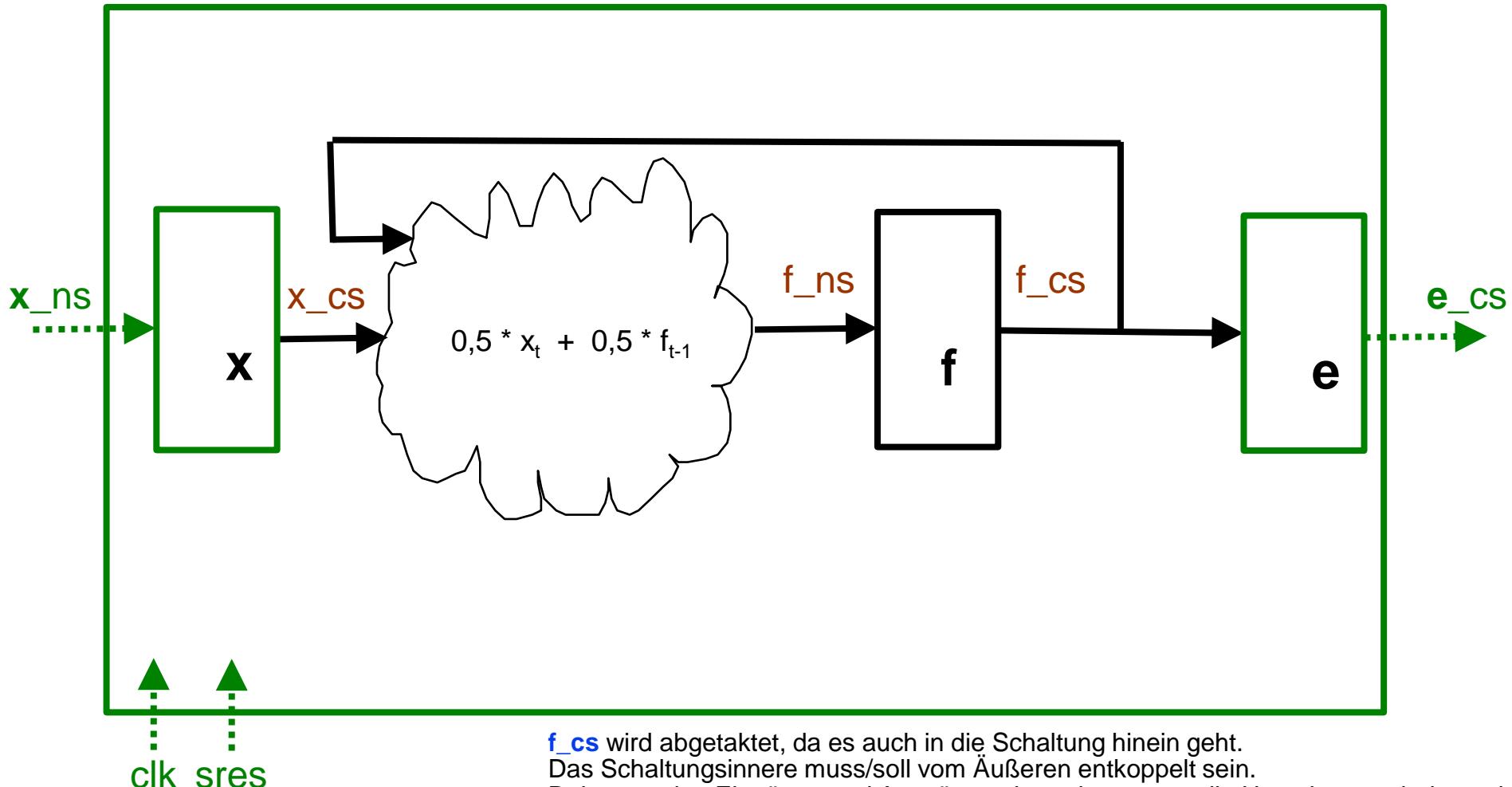
- Welche "Zustände" werden benötigt ?
Was muss sich die FSM zwingend merken ?

Dann (im Falle eines unbekannten Umfelds):

- Schaltung vom unbekannten Umfeld entkoppeln.
- Effekte im Umfeld dürfen nicht auf das Innere der Schaltung durchschlagen.
- Das Umfeld darf nicht Kenntnisse über das Innere der Schaltung haben müssen.
(Kennen wir bereits als Datenkapselung/Information Hiding - jetzt kommt sogar noch die "Physik" hinzu)
- Lösung: Eingänge und Ausgänge abtakten.

Übung/Lösung (Idee)

- Aus dem zuvor Besprochenen folgt:



f_CS wird abgetaktet, da es auch in die Schaltung hinein geht.
Das Schaltungsinnere muss/soll vom Äußeren entkoppelt sein.
Daher werden Eingänge und Ausgänge abgetaktet, wenn die Umgebung unbekannt ist.
Wir denken dran, selbsterklärende Namen sind etwas Feines ;-)
Also später nicht so was wie **f** oder **e** nutzen.

Lösung (1a)

```
...
clk,sres,x_cs,f_cs,e_cs,x_ns,f_ns,e_ns geeignet deklariert und *_cs dabei PowerUp-initialisiert
...
expSmoothi:
process ( x_cs, f_cs ) is
    variable fcur_v : geeigneter Typ ; -- current f(x)
    variable fnew_v : geeigneter Typ ; -- new/next f(x)
    variable opX_v : geeigneter Typ ;
    variable opF_v : geeigneter Typ ;
begin
    fcur_v := f_cs;
    opX_v := x_cs( x_cs'high )
                & x_cs( x_cs'high downto 1);           -- 0,5*x
    opF_v := fcur_v( fcur_v'high )
                & fcur_v( fcur_v'high downto 1); -- 0,5*f(x)
    fnew_v := opX_v + opF_v;

    f_ns <= fnew_v;
    e_ns <= fcur_v;
end process expSmoothi;
...
```

Lösung (1b)

```
...
clk,sres,x_cs,f_cs,e_cs,x_ns,f_ns,e_ns geeignet deklariert und *_cs dabei PowerUp-initialisiert
...
reg:
process ( clk ) is
begin
    if clk = '1' and clk'event then
        if sres = '1' then
            x_cs <= ( others => '0' );
            f_cs <= ( others => '0' );
            e_cs <= ( others => '0' );
        else
            x_cs <= x_ns;
            f_cs <= f_ns;
            e_cs <= e_ns;
        end if;
    end if;
end process reg;
...
```

Lösung (2a - kürzere Alternative)

```
...
clk,sres,x_cs,f_cs,e_cs,x_ns,f_ns geeignet deklariert und *_cs dabei PowerUp-initialisiert
...
expSmoothi:
process ( x_cs, f_cs ) is
    variable f_v : geeigneter Typ ;
    variable opX_v : geeigneter Typ ;
    variable opF_v : geeigneter Typ ;
begin
    opX_v := x_cs( x_cs'high ) + x_cs( x_cs'high downto 1);
    opF_v := f_cs( f_cs'high ) + f_cs( f_cs'high downto 1);
    f_v := opX_v + opF_v;
    f_ns <= f_v;
end process expSmoothi;
...
```

Lösung (2b - kürzere Alternative)

```
...
clk,sres,x_cs,f_cs,e_cs,x_ns,f_ns geeignet deklariert und *_cs dabei PowerUp-initialisiert
...
reg:
process ( clk ) is
begin
    if clk = '1' and clk'event then
        if sres = '1' then
            x_cs <= ( others => '0' );
            f_cs <= ( others => '0' );
            e_cs <= ( others => '0' );
        else
            x_cs <= x_ns;
            f_cs <= f_ns;
            e_cs <= f_cs;
        end if;
    end if;
end process reg;
...
```

<BREAK>

RTL (Register Transfer Level / Registertransferebene)

- ist eine Abstraktionsebene in der Hardware-Modellierung von ASICs & CPLDs.
- ist vermutlich die "wichtigste Arbeitsebene".
- ist die "übliche" Einstiegsebene ("Einstiegsformat") für die RTL-Synthese
- trennt klar die sequentielle Logik (Register / Zustände) von der kombinatorischen Logik (Schaltnetz / Zustandsüberführung&Ausgabefunktion)
- unterstützt "reduzierte Sicht" auf Zustandsüberführungen zu den Taktflanken (die Zeit wird diskretisiert)
- lässt Verantwortung für das Identifizieren und Anordnen der Zustände beim Entwickler.
- Seit ca. 1990 "taugt" RTL-Synthese und die "Kombination" aus HDL-Design und RTL-Synthese verdrängten das manuelle Schematic-Entry (Schaltplan-Eingabe/Schematic-Capture). Die RTL-Ebene wurde die "wichtigste Arbeitsebene". Bei geeignete Modellierung/HDL-Beschreibung erzielt die RTL-Synthese für kombinatorische Logik sehr gute Ergebnisse.
- Im "allgemeinen Fall" liefert die nächst höhere Abstraktionsebene bei der Synthese immer noch unbefriedigende Ergebnisse. Das Problem besteht im Identifizieren von geeigneten Zuständen/sequentieller Logik/Register/FFs.

Der Traum

- Insbesondere in der Informatik ist man bestrebt auf immer höheren Abstraktionsebenen arbeiten zu können um das Lösen immer komplexerer Probleme zu ermöglichen
- An dem Schritt oberhalb von RTL zu arbeiten wird schon lange gearbeitet ("High Level Synthesis")
- Für spezielle Fälle/Anwendungen arbeiten die Tools bereits sehr gut.
- Im "allgemeinen Fall" ist der Mensch noch überlegen

- Der "Traum" / das "Endziel" ist, das Problem in einer Sprache zu beschreiben und der Compiler entscheidet (*sinnvoll*), was in SW und was in HW realisiert wird.

- KI wie z.B: ChatGPT/OpenAI ist ein Schritt in diese Richtung

Operationen / Operatoren und Synthese

Zum Teil **Zur Erinnerung**

Unter VHDL gibt es u.a.:

- **Type Qualification** zur Klärung einer ambiguous expression.
Z.B.: `string'("dies ist ein string und kein std_logic_vector")`
- VHDL-Cast der eine **Type Conversion** ist.
Z.B.: `integer(3.14)`
Der VHDL-Sprachkern bestimmt wofür ein Cast angewendet werden kann.
- Und eine **Type Conversion** mit Hilfe einer Funktion, die aus einer Bibliothek bzw. einem Package kommt.
Z.B.: `to_integer("1100")`
Die jeweilige Bibliothek bestimmt wofür die jeweilige Funktion angewendet werden kann.

Wdh.: Einschub: std_logic_1164

```
PACKAGE std_logic_1164 IS
  ...
  TYPE std_ulogic IS (
    'U',   -- Uninitialized           "nicht initialisiert" - Wert darf niemals zugewiesen werden
    'X',   -- Forcing Unknown        "Konflikt"; 0 und 1 treffen aufeinander; Ausgang unklar
    '0',   -- Forcing 0               eine starke/normale 0
    '1',   -- Forcing 1               eine starke/normale 1
    'Z',   -- High Impedance         Hochohmig
    'W',   -- Weak Unknown           Konflikt zwischen starker 0 und schwacher 1
    'L',   -- Weak 0                 eine starke 0 (Pull Down)
    'H',   -- Weak 1                 eine starke 1 (Pull Up)
    '-'   -- Don't care            NICHT nutzen; good idea gone bad
  );
  TYPE std_ulogic_vector IS ARRAY (NATURAL RANGE <>) OF std_ulogic;

  FUNCTION resolved (s : std_ulogic_vector) RETURN std_ulogic;
  SUBTYPE std_logic IS resolved std_ulogic;
  TYPE std_logic_vector IS ARRAY (NATURAL RANGE <>) OF std_logic;
  ...

```

Wdh.: Einschub: std_logic_1164

```
PACKAGE BODY std_logic_1164 IS
  ...
  TYPE stdlogic_table IS ARRAY(std_ulogic, std_ulogic) OF std_ulogic;
  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' ) -- | - |
  );
  ...
  
```

Bemerkung

- Die Typen **SIGNED** und **UNSIGNED** sind eher zu meiden wenn auch sehr verbreitet
- Grund:
Beim Typ **std_logic_vector** ist unklar, ob das Bit binär ist (z.B. U, X, Z möglich). Das ist falsche Abstraktionsebene für Unterscheidung **SIGNED** / **UNSIGNED**
- Insbesondere für VHDL gilt:

Nur weil man es gegooglet hat, muss es noch lange nicht sinnvoll sein.

ieee.std_logic_arith ↔ ieee.numeric_std

ACHTUNG!

- VHDL unterstützt Operator-Overloading
- VHDL-Signatur schließt Typ des Rückgabewerts mit ein (anders als Java)
- Bibliotheken entscheiden über Bedeutung der Operation und Routinen

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
besser, da genormt - IEEE-Standard
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_[un]signed.all;
funktioniert im Labor, ist aber "nur" in der Praxis "üblich" - kein IEEE-Standard
```

ieee.std_logic_arith ↔ ieee.numeric_std

KLARE ENTSCHEIDUNG fällen

- **entweder nur** ieee.numeric_std

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std[_unsigned].all;
besser, da genormt - IEEE-Standard
```

- **oder nur** ieee.std_logic_arith

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_[un]signed.all;
```

- aber **nie** beides in einer Entity mixen

ieee.numeric_std

ieee.numeric_std

integer können in std_logic_vector umgerechnet werden mit:

```
...
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
...
...
std_logic_vector := std_logic_vector( to_unsigned( Integer_Value, Bit_Width_of_Result ) );
```

ieee.numeric_std

std_logic_vector können in integer umgerechnet werden mit:

```
...
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
...
...
unsigned
integer := to_integer( unsigned( std_logic_vector ) );      -- unsigned
signed
integer := to_integer( signed( std_logic_vector ) );      -- signed
```

ieee.numeric_std

Es kann wie folgt gerechnet werden:

Addition

unsigned

```
std_logic_vector := std_logic_vector( unsigned( std_logic_vector ) + unsigned( std_logic_vector ));
```

signed

```
std_logic_vector := std_logic_vector( signed( std_logic_vector ) + signed( std_logic_vector ));
```

Subtraktion

unsigned

```
std_logic_vector := std_logic_vector( unsigned( std_logic_vector ) - unsigned( std_logic_vector ));
```

signed

```
std_logic_vector := std_logic_vector( signed( std_logic_vector ) - signed( std_logic_vector ));
```

x "kürzer" als y (unsigned)

```
library ieee;
  use ieee.std_logic_1164.all;
  use ieee.numeric_std.all;
...
variable tmp,y,z : std_logic_vector( geeignet downto 0 );
variable x       : std_logic_vector( geeignet downto 0, aber "kürzer" );
...
tmp           := (others=>'0');          -- mit '0' auffüllen
tmp(x'high downto 0) := x;
z             := std_logic_vector( unsigned( tmp ) + unsigned( y ) );
...
```

- Bit-Breiten werden angepasst damit sie übereinstimmen
- automatisches Auffüllen wird u.U. "vom Package" unterstützt

x "kürzer" als y (signed)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
...
variable tmp,y,z : std_logic_vector( geeignet downto 0 );
variable x       : std_logic_vector( geeignet downto 0, aber "kürzer" );
...
tmp           := (others=>x(x'high)) ; -- mit MSB auffüllen
tmp(x'high downto 0) := x;
z             := std_logic_vector( signed( tmp ) + signed( y ) );
...
```

- Bit-Breiten werden angepasst damit sie übereinstimmen
- automatisches Auffüllen wird u.U. "vom Package" unterstützt

Übertrag ? / Wertebereich verlassen ? (unsigned)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
...
variable x,y,r : std_logic_vector( geeignet downto 0 );
variable z      : std_logic_vector( geeignet+1 downto 0 );
...
z := std_logic_vector( unsigned('0' & x) + unsigned('0' & y)); -- mit '0' auffüllen
if z(z'high)='1' then
    es gab Übertrag / der Wertebereich wurde verlassen => reagiere entsprechend
...
r := z( geeignet downto 0 );
...
```

- Bit-Breiten werden angepasst damit sie übereinstimmen

analog für Subtraktion

Übertrag / Wertebereich verlassen (unsigned)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
...
variable x,y,r : std_logic_vector( geeignet downto 0 );
variable z      : std_logic_vector( geeignet+1 downto 0 );
...
z := std_logic_vector( unsigned('0' & x) - unsigned('0' & y)); -- mit '0' auffüllen
if z(z'high)='1' then
    es gab Übertrag / der Wertebereich wurde verlassen => reagiere entsprechend
...
r := z( geeignet downto 0 );
...
```

- Bit-Breiten werden angepasst damit sie übereinstimmen

Überlauf ? / Wertebereich verlassen ? (signed)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
...
variable x,y,r : std_logic_vector( geeignet downto 0 );
variable z      : std_logic_vector( geeignet+1 downto 0 );
...
z := std_logic_vector( signed(x(x'high) & x) + signed(y(y'high) & y)); -- mit MSB auffüllen
if z(z'high) /= z(z'high-1) then
    es gab Überlauf / der Wertebereich wurde verlassen => reagiere entsprechend
...
r := z( geeignet downto 0 );
...
```

- Bit-Breiten werden angepasst damit sie übereinstimmen

analog für Subtraktion Überlauf / Wertebereich verlassen (signed)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
...
variable x,y,r : std_logic_vector( geeignet downto 0 );
variable z      : std_logic_vector( geeignet+1 downto 0 );
...
z := std_logic_vector( signed(x(x'high) & x) - signed(y(y'high) & y)); -- mit MSB auffüllen
if z(z'high) /= z(z'high-1) then
    es gab Überlauf / der Wertebereich wurde verlassen => reagiere entsprechend
...
r := z( geeignet downto 0 );
...
```

- Bit-Breiten werden angepasst damit sie übereinstimmen

Tipp: "Vergleiche" (1)

Beim Annähern an einen "Zielwert":

- Ein Vergleich auf Gleichheit ist in HW günstiger.
- Kostet über den Daumen pro Bit ein Äquivalenzgatter
- Ein Vergleich auf ">", " \geq ", " \leq " oder "<" ist sicherer (Outdoor-Equipment und siehe P1)
- Kostet über den Daumen einen n-Bit Subtrahierer

Tipp: "Vergleiche" (2) - bei "unsigned-Denke"

```
library ieee;
  use ieee.std_logic_1164.all;
  use ieee.std_logic_arith.all;                      -- u.a. für
conv_std_logic_vector
  use ieee.std_logic_unsigned.all;                   -- u.a. für conv_integer

...
constant max : natural := 2000;                     -- 2000[10] = x"7D0"
...

...
if counter_cs >= x"7D0"  then
...
...

if counter_cs >= conv_std_logic_vector( max, 11 )  then
...
...

if conv_integer(counter_cs) >= max  then
```

ieee.std_logic_arith

skipped

ieee.std_logic_arith

integer können in std_logic_vector umgerechnet werden mit:

```
...
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
...
...
std_logic_vector := conv_std_logic_vector( Integer-Value, Bit-Width-Of-Result )
```

skipped

ieee.std_logic_unsigned

- Package `ieee.std_logic_unsigned` nutzt `ieee.std_logic_arith`
- Package `ieee.std_logic_unsigned` gibt Operationen eine **unsigned**-Bedeutung
- (unsigned) `std_logic_vector` kann in `integer` umgerechnet werden mit:

```
...
library ieee;
    use ieee.std_logic_1164.all;
    use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all
...
...
integer := conv_integer( std_logic_vector );
```

mit unsigned-Bedeutung

skipped

ieee.std_logic_signed

- Package **ieee.std_logic_signed** nutzt **ieee.std_logic_arith**
- Package **ieee.std_logic_signed** gibt Operationen eine **signed**-Bedeutung
- (signed) std_logic_vector kann in integer umgerechnet werden mit:

```
...
library ieee;
    use ieee.std_logic_1164.all;
    use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;
...
...
integer := conv_integer( std_logic_vector )
```

mit signed-Bedeutung

skipped

Rechnen mit std_logic (unsigned)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
...
variable x,y,z : std_logic_vector( geeignet downto 0 );
...
z := x + y;
...
```

- **unsigned**-Bedeutung, da **std_logic_unsigned**
- Bit-Breiten müssen übereinstimmen

skipped

Rechnen mit std_logic (signed)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;
...
variable x,y,z : std_logic_vector( geeignet downto 0 );
...
z := x + y;
...
```

- **signed**-Bedeutung, da **std_logic_signed**
- Bit-Breiten müssen übereinstimmen

skipped

x "kürzer" als y (unsigned)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
...
variable tmp,y,z : std_logic_vector( geeignet downto 0 );
variable x         : std_logic_vector( geeignet downto 0, aber "kürzer" );
...
tmp           := (others=>'0');          -- mit '0' auffüllen
tmp(x'high downto 0) := x;
z             := tmp + y;
...
```

- **unsigned**-Bedeutung, da `std_logic_unsigned`
- Bit-Breiten werden angepasst damit sie übereinstimmen
- automatisches Auffüllen wird üblicher weise vom Package unterstützt

skipped

x "kürzer" als y (signed)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;
...
variable tmp,y,z : std_logic_vector( geeignet downto 0 );
variable x         : std_logic_vector( geeignet downto 0, aber "kürzer" );
...
tmp                  := (others=>x(x'high)) ; -- mit MSB auffüllen
tmp(x'high downto 0) := x;
z                   := tmp + y;
...
```

- **signed**-Bedeutung, da `std_logic_signed`
- Bit-Breiten werden angepasst damit sie übereinstimmen
- automatisches Auffüllen wird üblicher weise vom Package unstützt

skipped

Übertrag / Wertebereich verlassen (unsigned)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
...
variable x,y,r : std_logic_vector( geeignet downto 0 );
variable z      : std_logic_vector( geeignet+1 downto 0 );
...
z := ('0' & x) + ('0' & y);    -- mit '0' auffüllen
if z(z'high)='1' then
    es gab Übertrag / der Wertebereich wurde verlassen => reagiere entsprechend
...
r := z( geeignet downto 0 );
...
```

- **unsigned**-Bedeutung, da `std_logic_unsigned`
- Bit-Breiten werden angepasst damit sie übereinstimmen

skipped

analog für Subtraktion

Übertrag / Wertebereich verlassen (unsigned)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
...
variable x,y,r : std_logic_vector( geeignet downto 0 );
variable z      : std_logic_vector( geeignet+1 downto 0 );
...
z := ('0' & x) - ('0' & y);    -- mit '0' auffüllen
if z(z'high)='1' then
    es gab Übertrag / der Wertebereich wurde verlassen => reagiere entsprechend
...
r := z( geeignet downto 0 );
...
```

- **unsigned**-Bedeutung, da `std_logic_unsigned`
- Bit-Breiten werden angepasst damit sie übereinstimmen

skipped

Überlauf / Wertebereich verlassen (signed)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;
...
variable x,y,r : std_logic_vector( geeignet downto 0 );
variable z      : std_logic_vector( geeignet+1 downto 0 );
...
z := (x(x'high) & x) + (y(y'high) & y);    -- mit MSB auffüllen
if z(z'high) /= z(z'high-1) then
    es gab Überlauf / der Wertebereich wurde verlassen => reagiere entsprechend
...
r := z( geeignet downto 0 );
...
```

- **signed**-Bedeutung, da **std_logic_signed**
- Bit-Breiten werden angepasst damit sie übereinstimmen

skipped

analog für Subtraktion Überlauf / Wertebereich verlassen (signed)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;
...
variable x,y,r : std_logic_vector( geeignet downto 0 );
variable z      : std_logic_vector( geeignet+1 downto 0 );
...
z := (x(x'high) & x) - (y(y'high) & y);    -- mit MSB auffüllen
if z(z'high) /= z(z'high-1) then
    es gab Überlauf / der Wertebereich wurde verlassen => reagiere entsprechend
...
r := z( geeignet downto 0 );
...
```

- **signed**-Bedeutung, da **std_logic_signed**
- Bit-Breiten werden angepasst damit sie übereinstimmen

skipped

Kombinationen von Integer und std_logic_vector

Die Packages:

```
ieee.std_logic_arith.all  
ieee.std_logic_signed.all  
ieee.std_logic_unsigned.all  
ieee.numeric_std.all;
```

überladen die Operatoren.

Dort wird definiert was bzgl. `std_logic_vector` geht.

U.a. wird auch das Mixen von `std_logic_vector` und integer unterstützt.

skipped

Kombinationen von Integer und std_logic_vector

Beispiel:

```
...
variable x : std_logic_vector( geeignet downto 0 );
variable z : std_logic_vector( geeignet+1 downto 0 );
variable r : std_logic_vector( geeignet downto 0 );
...
z := (x(x'high) & x) + 7;    -- mit MSB auffüllen
if z(z'high) /= z(z'high-1) then
    es gab Überlauf / der Wertebereich wurde verlassen => reagiere entsprechend
...
r := z( geeignet downto 0 );
...
```

Oder einfach:

```
...
variable cnt : std_logic_vector( geeignet downto 0 );
...
cnt := cnt + 1;                -- round rotating counter
...
```

skipped

```

library IEEE;
use IEEE.std_logic_1164.all;

package std_logic_arith is
  ...
  type UNSIGNED is array (NATURAL range <>) of STD_LOGIC;
  ...
  function "+"(L: UNSIGNED; R: UNSIGNED) return UNSIGNED;
  ...
end Std_logic_arith;

```

IEEE.std_logic_arith

```

library IEEE;
use IEEE.std_logic_1164.all;

package body std_logic_arith is
  ...
  function "+"(L: UNSIGNED; R: UNSIGNED) return UNSIGNED is
    -- pragma label_applies_to plus
    -- synopsys subpgm_id 236
    constant length: INTEGER := max(L'length, R'length);
begin
  return unsigned_plus(
    CONV_UNSIGNED(L, length),
    CONV_UNSIGNED(R, length)
  ); -- pragma label plus
end;
  ...
end std_logic_arith;

```

skipped

IEEE.std_logic_arith / unsigned_plus

```
-- add two unsigned numbers of the same length
-- both arrays must have range (msb downto 0)
function unsigned_plus(A, B: UNSIGNED) return UNSIGNED is
    variable carry: STD_ULOGIC;
    variable BV, sum: UNSIGNED (A'left downto 0);
    -- pragma map_to_operator ADD_UNS_OP
    -- pragma type_function LEFT_UNSIGNED_ARG
    -- pragma return_port_name Z
begin
    if (A(A'left) = 'X' or B(B'left) = 'X') then
        sum := (others => 'X');
        return(sum);
    end if;

    carry := '0';
    BV := B;
    for i in 0 to A'left loop
        sum(i) := A(i) xor BV(i) xor carry;
        carry := (A(i) and BV(i)) or (A(i) and carry) or (carry and BV(i));
    end loop;
    return sum;
end;
```

skipped

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

```
package STD_LOGIC_UNSIGNED is
...
function "+"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
...
end STD_LOGIC_UNSIGNED;
```

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

```
package body STD_LOGIC_UNSIGNED is
...
function "+"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR is
-- pragma label_applies_to plus
constant length: INTEGER := maximum(L'length, R'length);
variable result : STD_LOGIC_VECTOR (length-1 downto 0);
begin
  result := UNSIGNED(L) + UNSIGNED(R); -- pragma label plus
  return std_logic_vector(result);
end;
...
end STD_LOGIC_UNSIGNED;
```

IEEE.std_logic_unsigned

skipped

Achtung (auch bei) bei Multiplikation

(Beispiel-)Problem:

- $3 * 3 = 9 \neq +1 = -1 * -1$

$$11_{\text{unsigned}} = 3$$

$$11_{\text{signed}} = -1$$

$$1001_{\text{unsigned}} = 9$$

$$0001_{\text{signed}} = +1$$

$$11_{\text{binär}} * 11_{\text{binär}} = ???$$

- Es muss auch binär unterschieden werden zwischen signed und unsigned Operanden!
- Entsprechend ist `ieee.std_logic_signed.all` oder `ieee.std_logic_unsigned.all` zu wählen.
- Die Anzahl Bits des Produkts entspricht zunächst der Summe der Bits der Multiplikanden.
- Bei einer Reduktion von Ergebnis/Produkt-Bits ist die Sinnhaftigkeit zu prüfen.

"Mixen" nur! in einem begründeten Ausnahmefall

- Über Use-Statement wird signed oder unsigned als Default ausgewählt.
- I.d.R. führt das **Mixen** von signed und unsigned Operationen zu schwer wartbaren Code und **ist** daher **zu meiden**.
- Wenn es gar nicht anders geht und gemixt werden muss, dann muss der jeweilige Operator qualifiziert werden. Operatoren sind nichts anderes als binäre Funktionen.

- Bsp. für signed:

```
resuSgn := ieee.std_logic_signed."+"( op1Sgn, op2Sgn );
```

Vermutlich Typkonvertierung (mit **SIGNED()**) nötig

- Bsp. für unsigned:

```
resuUns := ieee.std_logic_unsigned."+"( op1Uns, op2Uns );
```

Vermutlich Typkonvertierung (mit **UNSIGNED()**) nötig

- Obiges ist nur in sehr seltenen und begründeten Ausnahmefällen akzeptabel und erfordert definitiv guten Kommentar!

skipped

<BREAK>

Bemerkung / Notiz

- Spontane "Umstellung" um Zeit optimal auszunutzen
- Der Abschnitt FIFO, Flankendetector (&Pulsedetektor) wurde vorgezogen

- FIFO
 - Flankendetektor ("Pulse Shorter")
 - Pulsedetektor
-
- Bemerkung: FIFO ist schon lange bekannt - u.a. aus P1

VHDL-Idee für FIFO 1

```
...
process ( fifo_cs, shf_?s, si_?s ) is
    variable fifo_v : geeigneter Typ ;
    variable so_v    : geeigneter Typ ;
begin
    so_v := fifo_cs( fifo_cs'high );
    if shf_?s='1' then
        fifo_v := fifo_cs( fifo_cs'high-1 downto 0 ) & si_?s;
    else
        fifo_v := fifo_cs
    end if;

    so_?s     <=  so_v;
    fifo_ns  <=  fifo_v;

end process . . .;
...
```

VHDL-Idee für FIFO 2

...

sequlo:

```
process ( clk ) is
begin
    if ( clk'event and clk = '1' ) then
        if ( sres_s = '1' ) then
            fifo_cs <= ( others => '0' );
        else
            fifo_cs <= fifo_ns;
        end if;
    end if;
end process sequlo;
...
```

Flankendetektor / Edge Detector

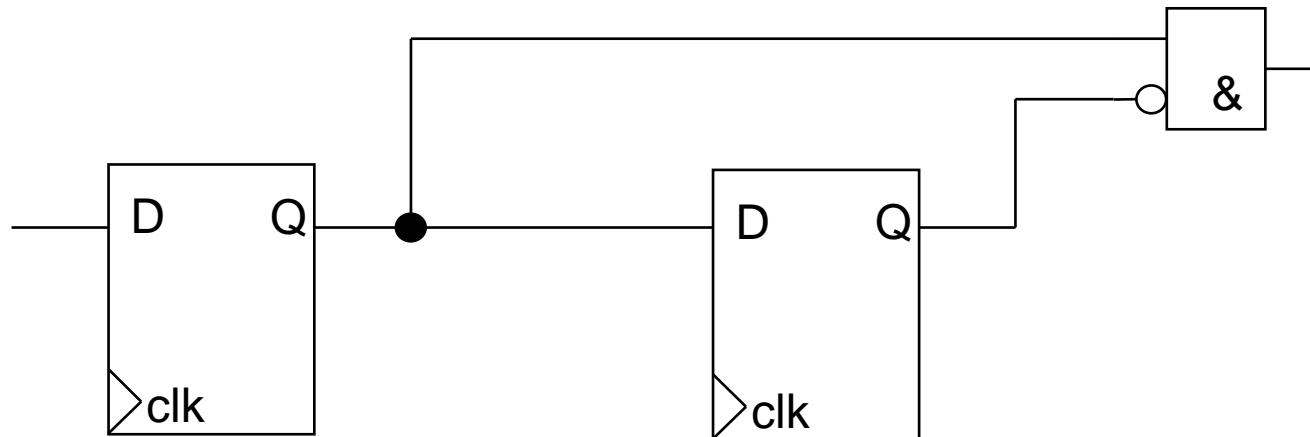
+

Pulse Detector

Flankendetektor / Edge Detector

- Wird auch "**Pulse Shorter**" genannt
- Der Flankendetektor "beobachtet" ein Eingangssignal und detektiert eine Flanke
- Pulse hinter Flanke muss "ausreichend" breit sein

(Rising) Edge Detector



Übung

- Schreiben Sie den VHDL-Code

...

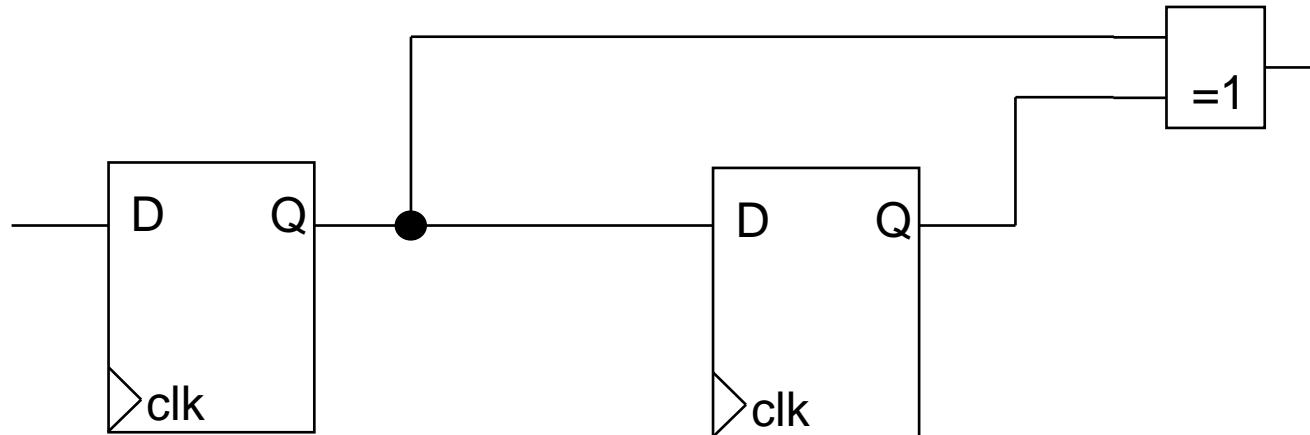
```
osig  <= ff1st_cs and not ff2nd_cs; -- concurrent signal assignment
```

reg:

```
process ( clk ) is
begin
    if clk'event and clk='1' then
        if res='1' then
            ff1st_cs  <= '0';
            ff2nd_cs  <= '0';
        else
            ff1st_cs  <= isig;
            ff2nd_cs  <= ff1st_cs;
        end if;
    end if;
end process reg;
```

...

(Both Edges) Edge Detector



Übung

- Schreiben Sie den VHDL-Code

...

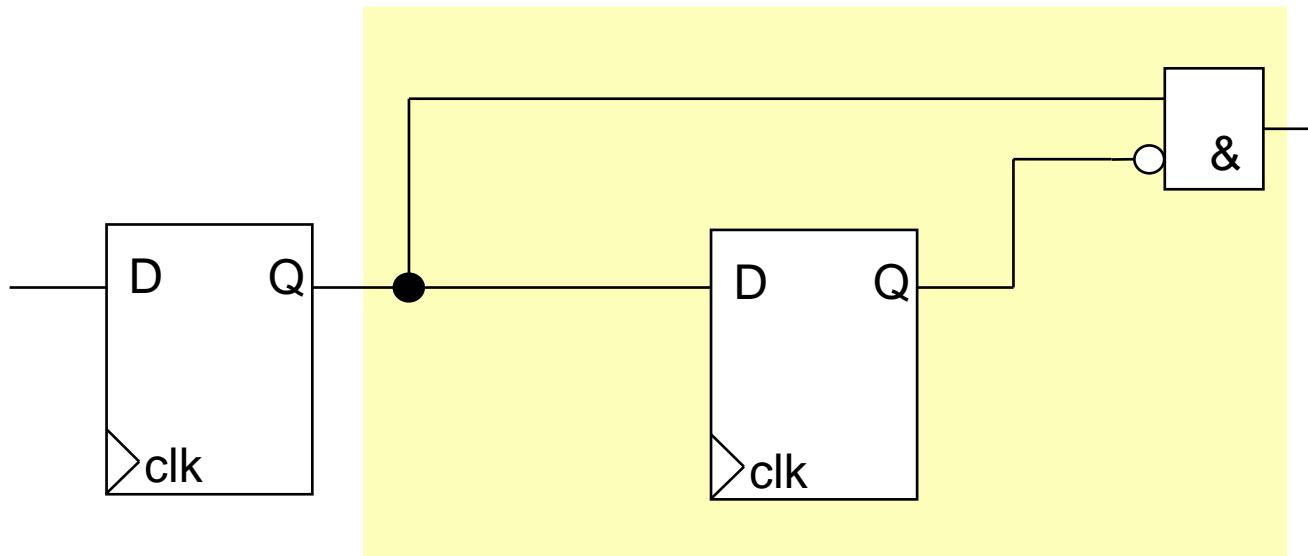
```
osig  <= ff1st_cs xor ff2nd_cs;          -- concurrent signal assignment
```

reg:

```
process ( clk ) is
begin
    if  clk'event and clk='1'  then
        if  res='1'   then
            ff1st_cs  <= '0';
            ff2nd_cs  <= '0';
        else
            ff1st_cs  <= isig;
            ff2nd_cs  <= ff1st_cs;
        end if;
    end if;
end process reg;
```

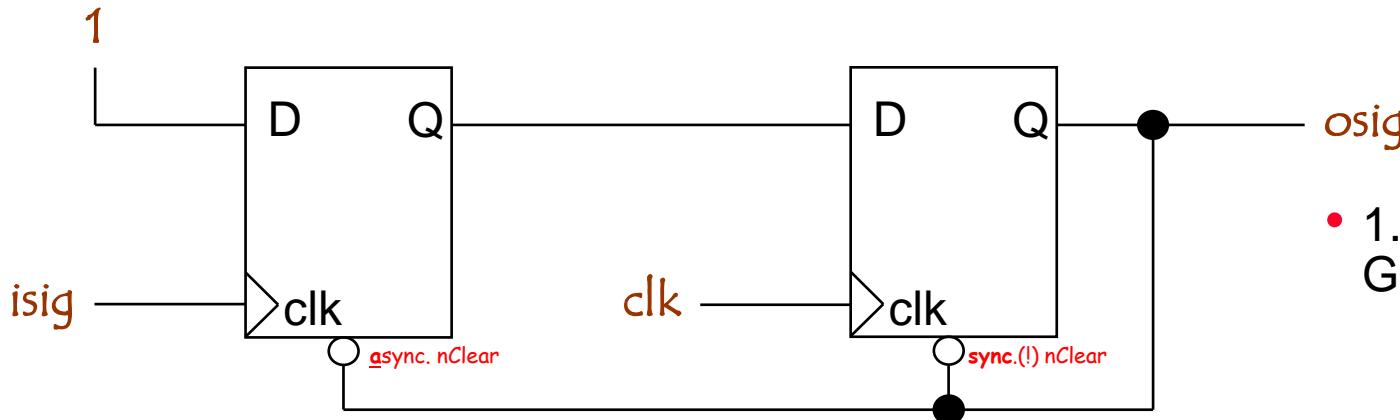
...

Bemerkung **Edge Detector**

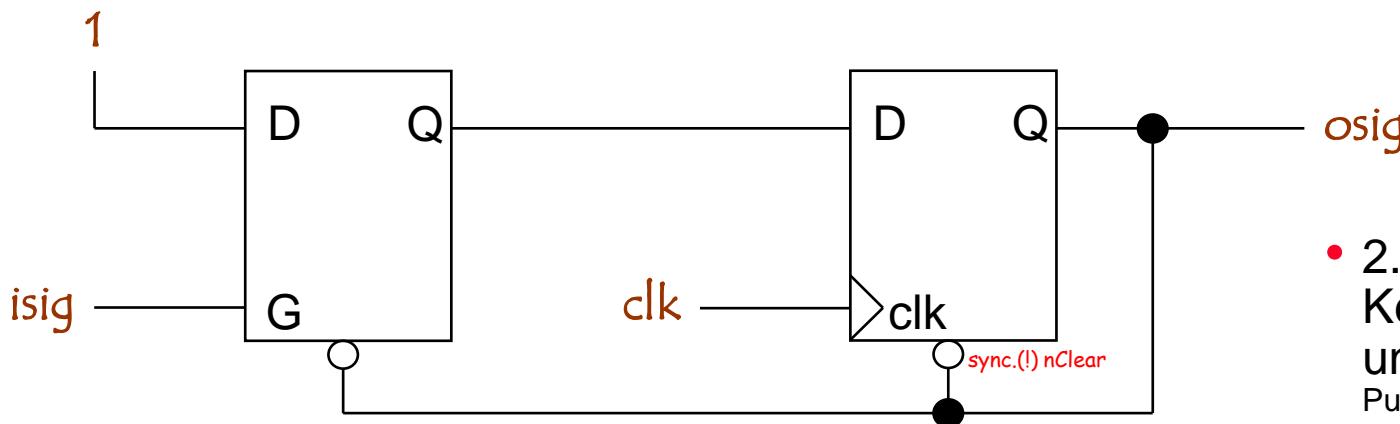


- Der Flanken-Detektor besteht aus zwei FFs
- Ob das "eine FF" (im Bild außerhalb der gelben Kastens) vorne (dem gelben Kasten) oder hinten (hinter dem gelben Kasten) angeordnet ist, ist philosophisch

(High Active)-Pulse Detector



- 1. Variante
Gated Clk



- 2. Variante
Kein Gated Clk
und 1 Latch gespart
Pulse darf nicht zu lang sein

Low Active Reset bzw nClear muss (am "hinteren" FF) synchron sein
sonst (bei dieser Beschaltung) Race Condition

Achtung! Obiges gibt Idee wieder - RESET eines FFs als Konsequenz eines Wertewechsels desselben FFs kann zu "Hold-Time-Violation" führen.

VHDL 1

.Variante

```
...
ff1st:
process ( isig ) is
begin
    if isig'event and isig='1' then
        if ff2nd_cs='1' then
            ff1st_cs  <=  '0';
        else
            ff1st_cs  <=  '1';
        end if;
    end if;
end process ff1st;

ff2nd:
process ( clk ) is
begin
    if clk'event and clk='1' then
        if ff2nd_cs='1' then
            ff2nd_cs  <=  '0';
        else
            ff2nd_cs  <=  ff1st_cs;
        end if;
    end if;
end process ff2nd;

osig  <=  ff2nd_cs;
...
```

skipped

VHDL 2.Variante

```
...
ff1st:
process ( isig ) is
begin
    if isig='1' then
        if ff2nd_cs='1' then
            ff1st_cs  <=  '0';
        else
            ff1st_cs  <=  '1';
        end if;
    end if;
end process ff1st;

ff2nd:
process ( clk ) is
begin
    if clk'event and clk='1' then
        if ff2nd_cs='1' then
            ff2nd_cs  <=  '0';
        else
            ff2nd_cs  <=  ff1st_cs;
        end if;
    end if;
end process ff2nd;

osig  <=  ff2nd_cs;
...
```

skipped

Bemerkung zum Pulse Detector

Achtung!

- Laufzeiten / Timing-Details sind beim Pulse-Detector wichtig und müssen beachtet werden
Ausgang des hinteren FFs führt auf Reset-Eingang. U.U. ist hier Verzögerung nötig um Setup-/Hold-Time-Violation zu vermeiden
- Der Pulse Detector fängt Pulse, also auch Glitches / Spikes / Hazards

Ist das gewollt? - I.d.R. nicht !

Kann garantiert werden, dass keine Glitches / Spikes / Hazards auftreten ?

Der Pulse Detector ist etwas für Experten

skiped

Bemerkung zur Synthese

- Synthese ist sehr aufwendig (NP-Vollständig)
- Synthese nutzt heuristische Verfahren
- Es gibt ein Zeit-Budget
- Als RTL-Designer oft Denke, dass ist Job des Synthese-Tools aus dem VHDL-Code war "Ordentliches" zu machen
- Damit das funktioniert, sollte/darf das Zeit-Budget nicht unnötig belastet werden
- Es ist wichtig, der Synthese nicht eine falsche/schlechte "Startidee" mitzugeben

```

[if_label:]  

IF condition THEN  

    sequence_of_statements;  

ELSIF condition THEN  

    sequence_of_statements;  

ELSE  

    sequence_of_statements;  

END IF [if_label];

```

Wdh.: IF..THEN..ELSE

```

-- maximum berechnen
IF ( x < y ) THEN
    max := y;
ELSE
    max := x;
END IF;

-- Absolut-Wert berechnen
IF ( x = INTEGER'LOW ) THEN
    REPORT "PROBLEM" SERVERYITY ERROR;
ELSIF ( x < 0 ) THEN
    abs_x := -x;
ELSE
    abs_x := x;
END IF;

```

[case_label:]

```
CASE expression IS
    WHEN choices      => sequence_of_statements;
    [WHEN OTHERS        => sequence_of_statements; ]
END CASE [case_label:];
```

Wdh.: CASE

-- CHARACTER nach INTEGER wandeln

```
CASE char IS
    WHEN '0' =>      int := 0;
    WHEN '1' =>      int := 1;
    WHEN '2' =>      int := 2;
    WHEN '3' =>      int := 3;
    WHEN '4' =>      int := 4;
    WHEN '5' =>      int := 5;
    WHEN '6' =>      int := 6;
    WHEN '7' =>      int := 7;
    WHEN '8' =>      int := 8;
    WHEN '9' =>      int := 9;
    WHEN OTHERS => int := -1; REPORT "PROBLEM" SERVERTITY NOTE;
END CASE;
```

Übung (Prioritäten)

- Es soll einer von vier Eingangswerten ausgewählt werden
 - Die Auswahl soll abhängig von 3 Bedingungen erfolgen
 - Wenn die 1.Bedingung erfüllt ist, soll der 1.Wert genommen werden
 - Andernfalls, wenn die 2.Bedingung erfüllt ist, soll der 2.Wert genommen werden
 - Andernfalls, wenn die 3.Bedingung erfüllt ist, soll der 3.Wert genommen werden
 - Andernfalls soll der 4.Wert genommen werden
-
- Wie sieht der VHDL-Code aus?
 - Wie sieht die HW-Idee vor der "Optimierung" aus?

Prioritäten / Mux-Kaskade

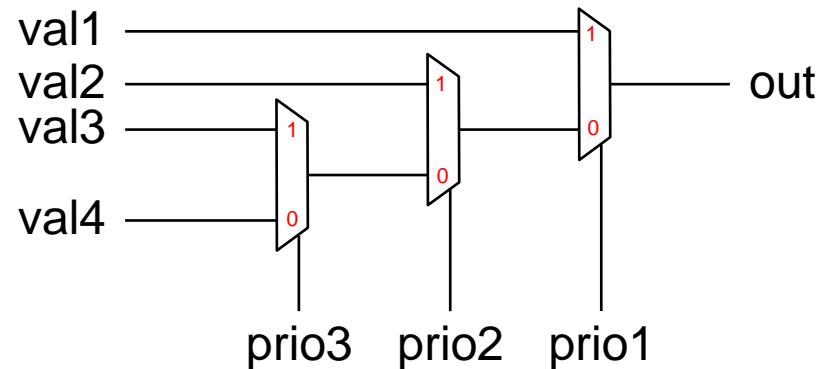
...
Beispiel:

```
process( ... ) is
  ...
begin
  if    prio1 = '1'  then
    out_v := val1_?;
  elsif  prio2 = '1'  then
    out_v := val2_?;
  elsif  prio3 = '1'  then
    out_v := val3_?;
  else
    out_v := val4_?;
end if;
```

```
  out_s <= out_v;
end process Beispiel;
...
```

val1 hat höchste Priorität wird am schnellsten "berechnet" bzw. durchläuft am schnellsten Mux-Kaskade
=> val1 kann später "stabil werden"
val4 (& val3) haben niedrigste Priorität - werden am langsamsten "berechnet"
=> val4(&val3). müssen sich früher stabilisieren

HW-Idee vor der Synthese



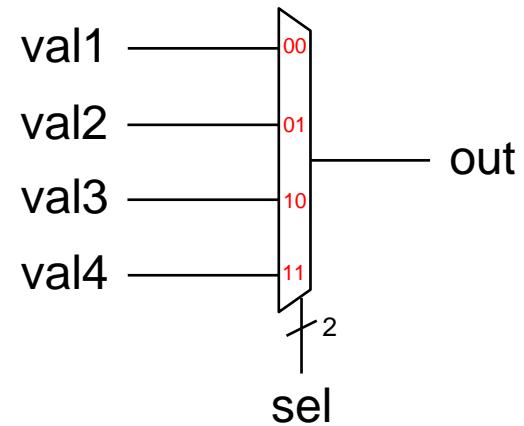
Übung (Keine Prioritäten)

- Es soll einer von vier Eingangswerten ausgewählt werden
 - Es gibt keine Prioritäten
-
- Wie sieht der VHDL-Code aus?
 - Wie sieht die HW-Idee vor der Synthese aus?

Mux (4:1)

```
...
Beispiel:
process( ... ) is
  ...
begin
  case sel is
    when "00" => out_v := val1_?;
    when "01" => out_v := val2_?;
    when "10" => out_v := val3_?;
    when "11" => out_v := val4_?;
  end case;
  out_s <= out_v;
end process Beispiel;
...
```

HW-Idee vor der Synthese



Übung

- Über einen 32 Bit Datenbus wird ein signed Wert gegeben (`in_s`)
- Führen Sie wahlweise eine Sign-Extension für die unteren
 - 8 der eingehende Wert ist im 8 Bit signed Wertebereich
 - 16 der eingehende Wert ist im 16 Bit signed Wertebereich
 - 24 der eingehende Wert ist im 24 Bit signed Wertebereich
 - 32 (bzw. reichen Sie die 32 Bit unverändert weiter)

Bit durch

- Wie sieht der VHDL-Code aus?
- Wie sieht die HW-Idee vor der Synthese aus?

...

Beispiel:

```
process( in_s, sel_s ) is
    ...
begin
    case sel_s is
        when "00" =>
            resu_v           := (others=>in_s(7));
            resu_v( 7 downto 0) := in_s(7 downto 0);
        when "01" =>
            resu_v           := (others=>in_s(15));
            resu_v(15 downto 0) := in_s(15 downto 0);
        when "10" =>
            resu_v           := (others=>in_s(23));
            resu_v(23 downto 0) := in_s(23 downto 0);
        when others =>
            resu_v           := in_s;
            -- including "11"
    end case;

    resu_s <= resu_v;
end process Beispiel;
...
```

Lösung

aufwendiger für Synthese

...

Beispiel:

```
process( ... ) is
    ...
begin
    if ( sel = '1' ) then
        resu_v := op1_? * op2_?;
    else
        resu_v := op3_? * op4_?;
    end if;

    resu_s <= resu_v;
end process Beispiel;
...
```

HW-Bedarf vor der Synthese(-Optimierung): 2 Multiplizierer

Multiplikation dient hier als Metapher für "etwas" aufwendiges (gemessen in HW-Kosten)
Multiplikation ist "bereits so teuer", dass es diesbezüglich lohnt unnötige Kosten zu vermeiden

Synthese-freundlicher

```
...
Beispiel:
process( ... ) is
  ...
begin
  if ( sel = '1' ) then
    tmp1_v := op1_?;
    tmp2_v := op2_?;
  else
    tmp1_v := op3_?;
    tmp2_v := op4_?;
  end if;
  resu_v := tmp1_v * tmp2_v;

  resu_s <= resu_v;
end process Beispiel;
...
```

HW-Bedarf vor der Synthese(-Optimierung): 1 Multiplizierer

aufwendiger für Synthese allgemein

```
...
Beispiel:
process( ... ) is
  ...
begin
  if ( sel = '1' ) then
    resu_v := complexeOperation( op1a_?, ..., opNa_? );
  else
    resu_v := complexeOperation( op1b_?, ..., opNb_? );
  end if;

  resu_s <= resu_v;
end process Beispiel;
...
```

Synthese-freundlicher allgemein

```
...
Beispiel:
process( ... ) is
  ...
begin
  if ( sel = '1' ) then
    tmp1_v := op1a_?;
    tmp2_v := op2a_?;
    ...
    tmpN_v := opNa_?;
  else
    tmp1_v := op1b_?;
    tmp2_v := op2b_?;
    ...
    tmpN_v := opNb_?;
  end if;
  resu_v := complexeOperation( tmp1_v, ..., tmpN_v );

  resu_s <= resu_v;
end process Beispiel;
...
```

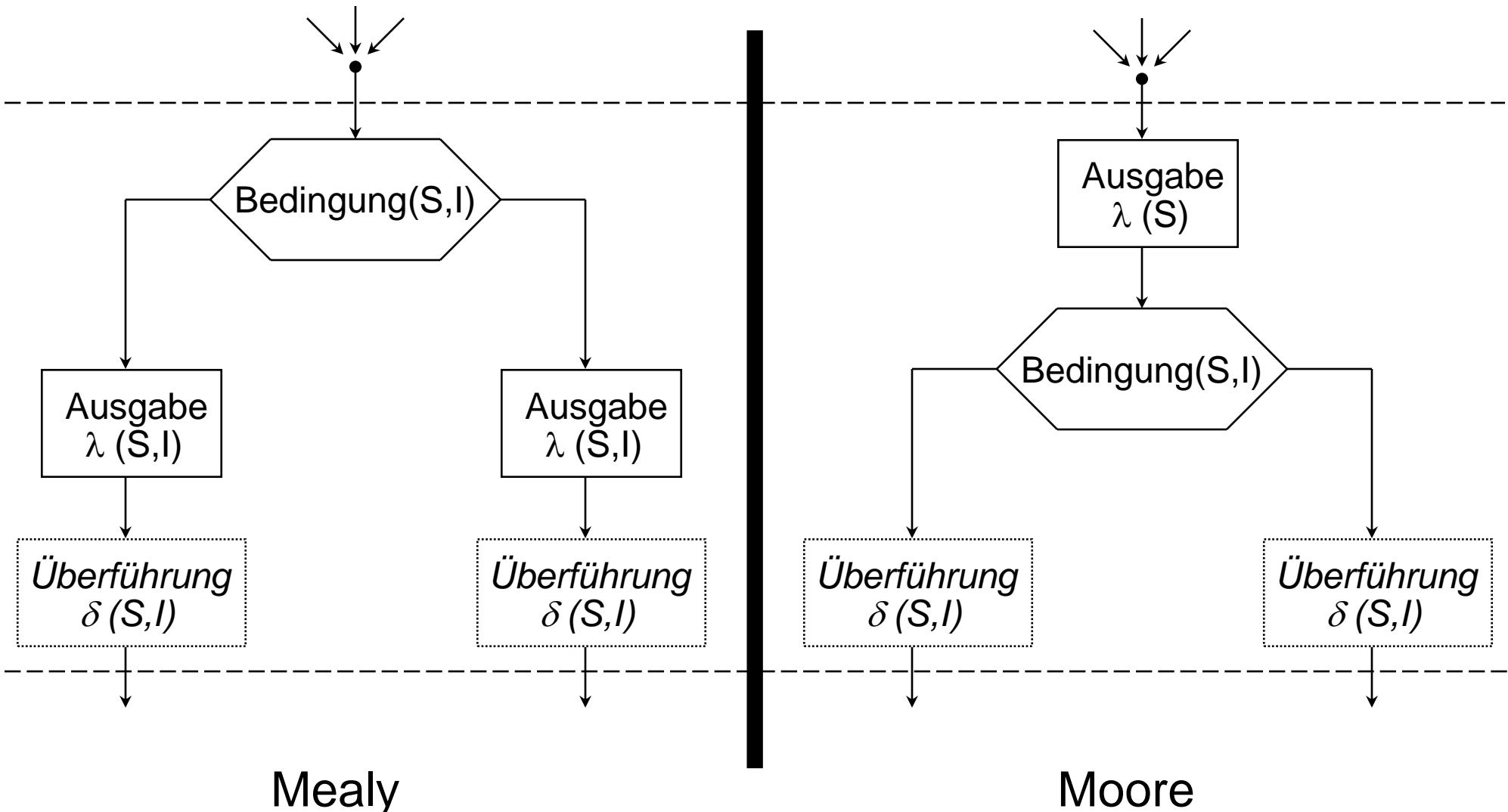
Automaten / Mealy- und Moore-Automaten (1)

- Automaten sind sehr wichtig in der Informatik.
- Hier endliche deterministische Automaten interessant.
- In der HW-Entwicklung spricht man von: FSM (Finite State Machine)
- Grundsätzliche Verhalten eines Automaten ist immer gleich:
 - Automat befindet sich in einem Zustand.
 - Automat wird von außen eine Eingabe (Folge von Zeichen / Signalen) vorgelegt.
 - Eintreffen von Eingabezeichen (Signalen) kann abhängig vom Eingabezeichen und dem gegenwärtigen Zustand Überführung in neuen Zustand / Folgezustand auslösen (Zustandsübergang oder Transition).
 - Automat hat Ausgabe

Mealy- und Moore-Automaten (2)

- Art der Berechnung der Ausgabe bestimmt Automatentyp
- Beim **Moore-Automat** ist die Ausgabe nur vom Zustand abhängig
- Der **Medwedew-Automat** ist Spezialfall des **Moore-Automaten**.
- Ausgänge entsprechen direkt den Zustands-FFs .
Es gibt keine Ausgabefunktion (bzw. Ausgabefunktion ist trivial/Identität)
Wird im Weiteren als Moore-Automat betrachtet.
- Beim **Mealy-Automat** ist die Ausgabe vom Zustand und Eingabe abhängig
- Aufbau / Strukturierung der FSM mit Moore-/Mealy-Automaten hilft Entwurfs-Fehler zu vermeiden.
- Erleichtert Verständnis und fehlerfreien Entwurf.
- Erleichterungen brechen zusammen, wenn Moore & Mealy gemixt werden.

Mealy und Moore Zustände im Flußdiagramm



Moore-Automaten

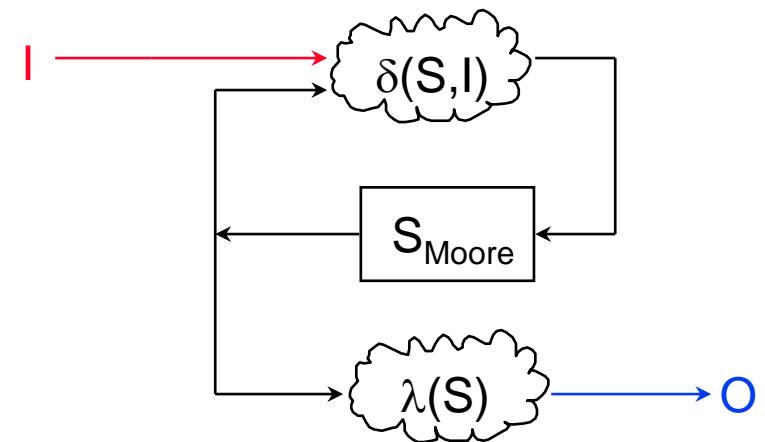
- Schaltwerke, bei denen die Ausgangssignale nur von dem aktuellen Zustand abhängen, realisieren **Moore-Automaten**

formal:

$M_{Moore} = (S, I, O, \delta, \lambda, S_0)$ mit
Zustandsmenge $S = \{S_1, \dots, S_k\}$
Eingabealphabet $I = \{I_1, \dots, I_n\}$
Ausgabealphabet $O = \{O_1, \dots, O_m\}$
Überführungsfunktion $\delta : I \times S \rightarrow S$

→ **Ausgabefunktion $\lambda : S \rightarrow O$**
Startzustand $S_0 \in S$

$$\begin{aligned} O^n &= \lambda(S^n) \\ S^{n+1} &= \delta(I^n, S^n) \end{aligned}$$



- Die (Zustands-)Überführungsfunktion oder auch Übergangsfunktion δ wird im **ÜSN (ÜbergangsSchaltNetz)** implementiert.
- Die Ausgabefunktion oder auch Ausgangsfunktion λ wird im **ASN (AusgangsSchaltNetz)** implementiert.

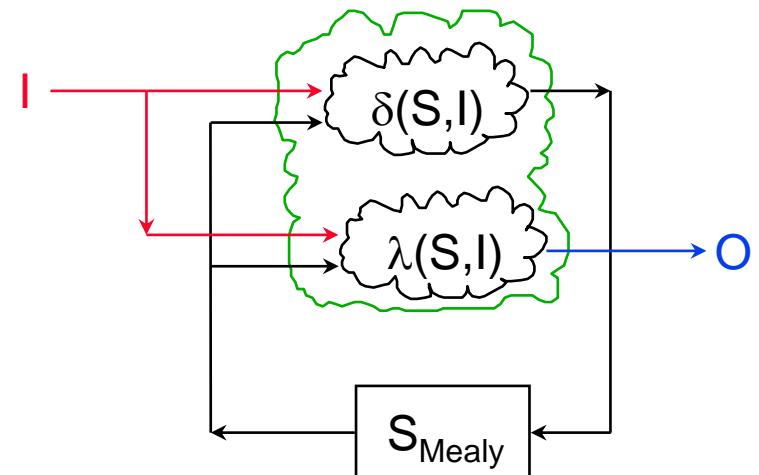
Mealy-Automaten

- Schaltwerke, bei denen die Eingangssignale direkt den Wert der Ausgangssignale beeinflussen, realisieren **Mealy-Automaten**

formal:

$M_{\text{Mealy}} = (S, I, O, \delta, \lambda, S_0)$ mit
Zustandsmenge $S = \{S_1, \dots, S_k\}$
Eingabealphabet $I = \{I_1, \dots, I_n\}$
Ausgabealphabet $O = \{O_1, \dots, O_m\}$
Überführungsfunktion $\delta : I \times S \rightarrow S$
→ **Ausgabefunktion** $\lambda : I \times S \rightarrow O$
Startzustand $S_0 \in S$

$$\begin{aligned} O^n &= \lambda(I^n, S^n) \\ S^{n+1} &= \delta(I^n, S^n) \end{aligned}$$



- Die (Zustands-)Überführungsfunktion oder auch Übergangsfunktion δ wird im **ÜSN** (**ÜbergangsSchaltNetz**) implementiert.
- Die Ausgabefunktion oder auch Ausgangsfunktion λ wird im **ASN** (**AusgangsSchaltNetz**) implementiert.

Mealy und Moore als Pseudo VHDL Code

```
SN: process ( I_s, S_cs ) is
begin
    -- Überführungsfunktion
    S_ns <=  $\delta$ ( S_cs, I_s );
    -- Ausgabefunktion
    O_s <=  $\lambda$ ( S_cs, I_s );
end process SN;
```

```
Zustand: process ( clk ) is
begin
    if clk='1' and clk'event then
        S_cs <= S_ns;
    end if;
end process Zustand;
```

Mealy

```
ÜSN: process ( I_s, S_cs ) is
begin
    -- Überführungsfunktion
    S_ns <=  $\delta$ ( S_cs, I_s );
end process ÜSN;
```

```
ASN: process ( S_cs )
begin
    -- Ausgabefunktion
    O_s <=  $\lambda$ ( S_cs );
end process ASN;
```

```
Zustand: process ( clk ) is
begin
    if clk='1' and clk'event then
        S_cs <= S_ns;
    end if;
end process Zustand;
```

Moore

Warum Moore ?

- Mealy-Automaten können geringfügig/vernachlässigbar schneller sein (um einen Takt)
 - Menschen mögen (oft) Mealy-Automaten
 - Tools mögen Moore-Automaten
 - Mealy- und Moore-Automaten sind "gleichwertig"/"gleichmächtig"
 - Mealy- und Moore-Automaten nicht mixen! Das Ergebnis ist weder Mealy noch Moore.
-
- Moore-Automaten sind vorzuziehen!
Je mehr Anfänger, desto mehr Moore ;-)

Aber warum?

Vergleich: Mealy \leftrightarrow Moore (1)

Mealy

- Änderungen am Eingang können "sofort" auf den Ausgang "durchschlagen".
Ist halt Mealy.

Daraus folgt:

- Glitches/Spikes am Ausgang möglich (wahrscheinlicher)
- Ausgänge später/spät stabil/eingeschwungen
- **Mealy-Automaten "verstärken/verschlimmern" die Glitches/Spikes am Eingang**

Moore

- Änderungen am Eingang können nicht "sofort" auf den Ausgang "durchschlagen".
Ist halt Moore.

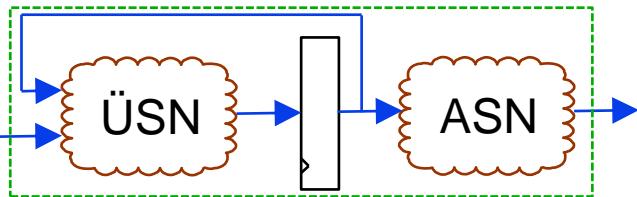
Daraus folgt:

- Glitches/Spikes am Ausgang deutlich unwahrscheinlicher.
 - Treten bei "vernünftigen" Design / ASN praktisch nicht auf.
 - Optimale Voraussetzungen damit keine Glitches/Spikes auftreten.
- Ausgänge früh/früher stabil/eingeschwungen
- **Moore-Automaten "filtern" die Glitches/Spikes am Eingang**

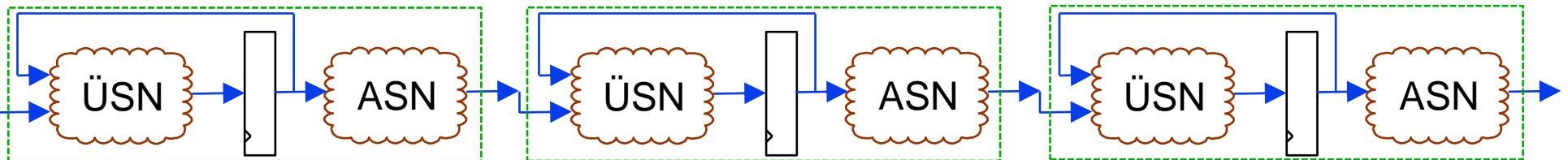
Folie führt in Klausur
oft zu ungünstlichen
bzw. falschen Aussagen

Moore-Automaten

- Ein einzelner Moore-Automat

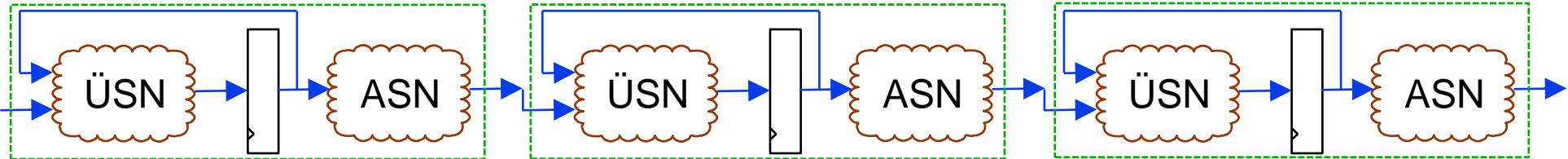


- Hintereinanderschaltung von Moore-Automaten

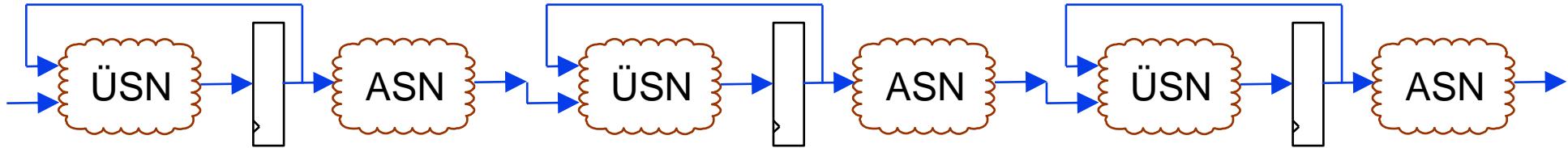


Zusammenschaltung von Moore-Automaten

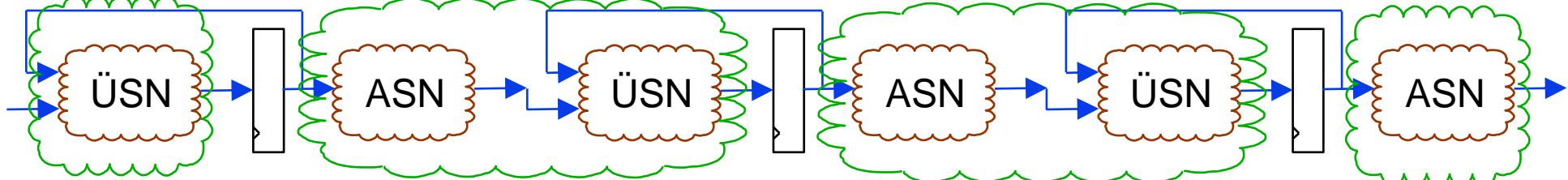
1



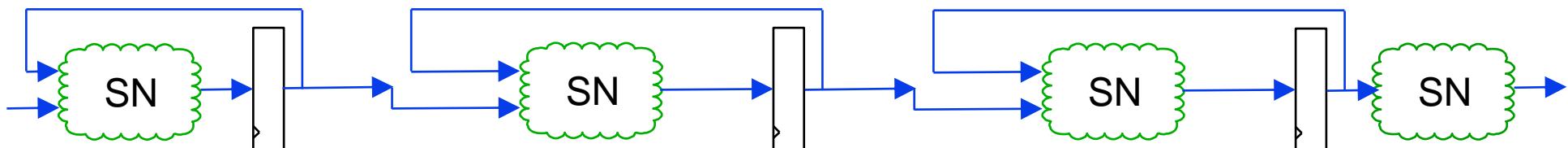
2



3

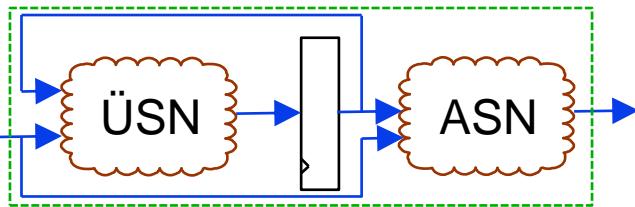


4

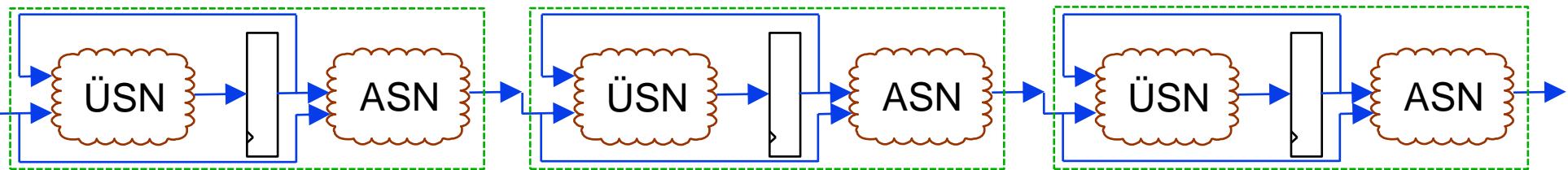


Mealy-Automaten

- Ein einzelner Mealy-Automat

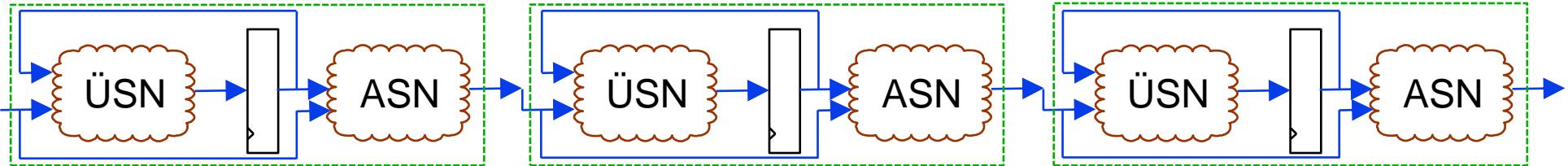


- Hintereinanderschaltung von Mealy-Automaten

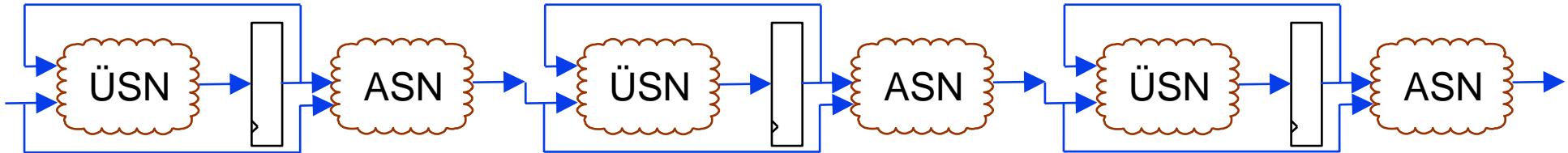


Zusammenschaltung von Mealy-Automaten

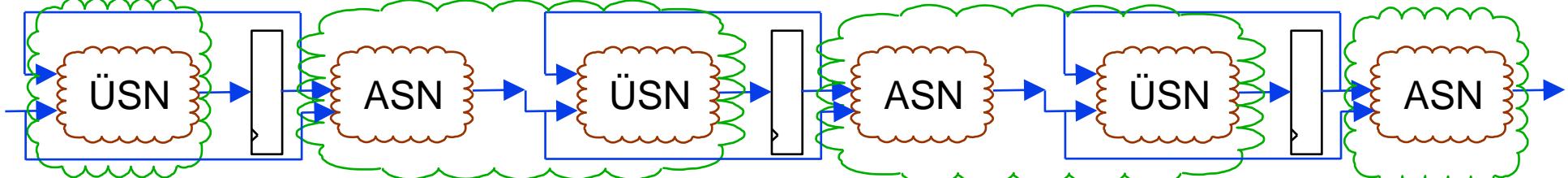
1



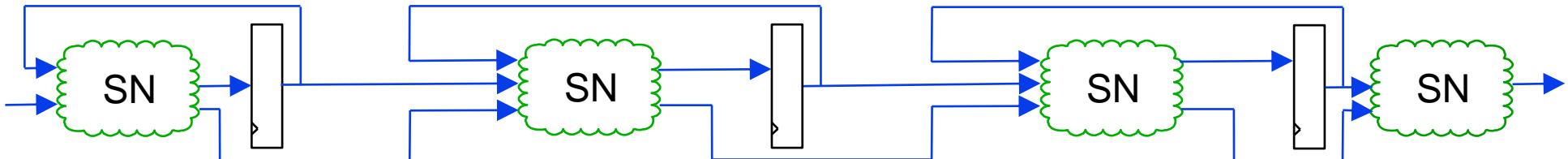
2



3

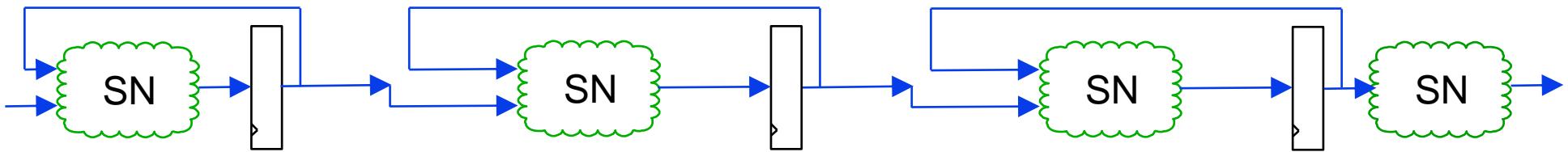


4

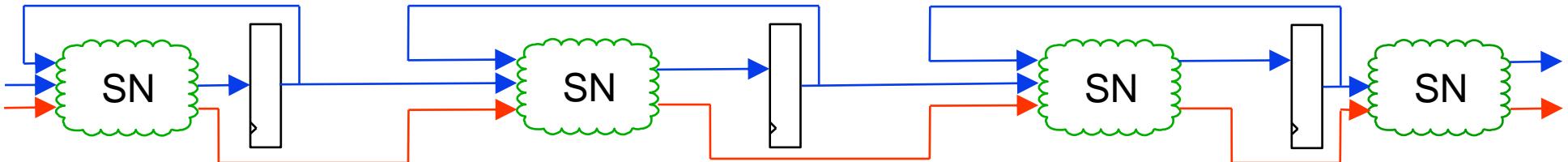


Vergleich: Mealy \leftrightarrow Moore (2)

- Vereinfachte Sicht auf zusammengeschaltete Moore-Automaten nach möglichen Optimierungen (z.B. durch Synthese)



- Vereinfachte Sicht auf zusammengeschaltete Mealy-Automaten nach möglichen Optimierungen (z.B. durch Synthese)



Achtung bei komplexen "Mealy-FSM"

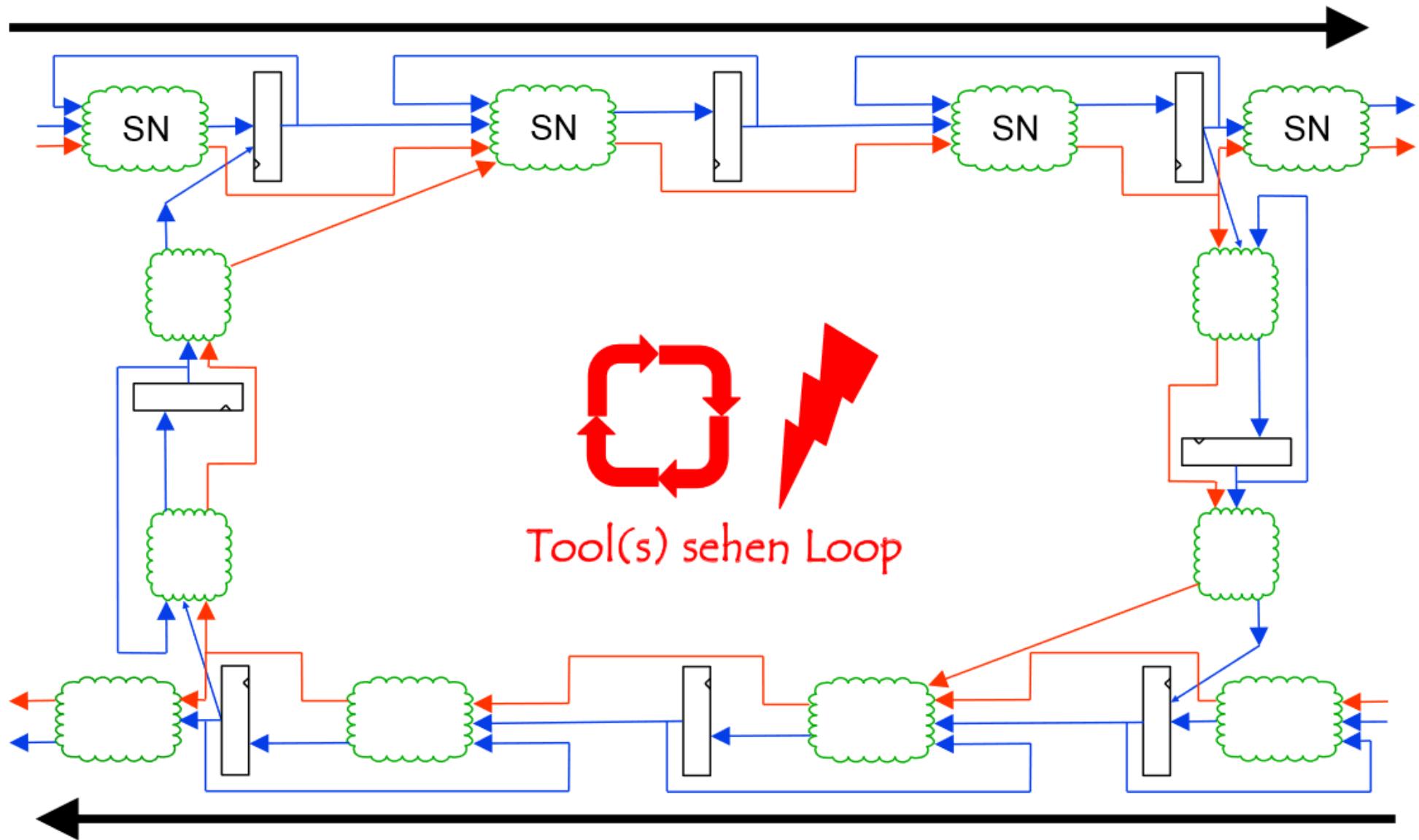
Ausgangssituation:

- In einer komplexen FSM, die intern aus einzelnen Automaten aufgebaut ist, tauschen die Automaten Informationen aus.
- Durch "Rückmeldungen"/Rückkopplungen kann es Kommunikations-Loops geben.
- Diese sind (zunächst) vollsynchron - d.h. jede(r) Rückkopplung/Zyklus läuft über FFs bzw. enthält Abtaktstufen.
- Was, wenn alle am Kommunikations-Loop beteiligten Automaten Mealy-Automaten sind

Beispiel Telekommunikation:

- Kommunikation ist bidirektional (Full Duplex)
2 Datenströme bzw. Datenströme in jede Richtung
- Loops schaltbar um Fehler lokalisieren zu können
 - Sender fordert an, dass ihm Daten zurückgesendet werden
 - Zwischen dem "letzten Loop", der funktioniert und dem "ersten Loop", der nicht funktioniert liegt der Fehler.
- Empfänger kann dasselbe

Gefahr bei Mealy in Kommunikations-Loops oder bidirektionalen Datenströmen



Wie Mealy und Moore koppeln?

Ausgangssituation - ReUse von bereits Existierenden

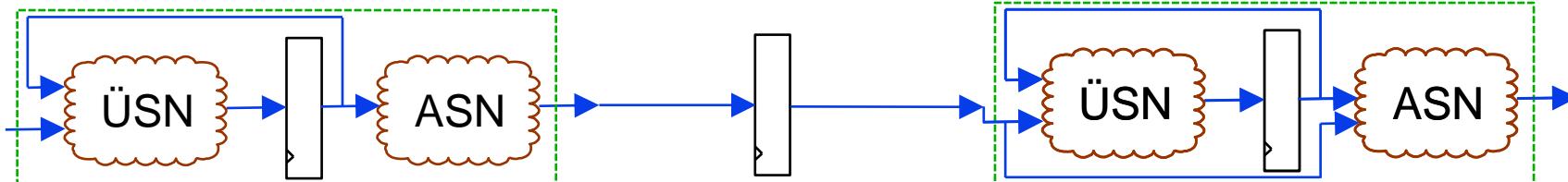
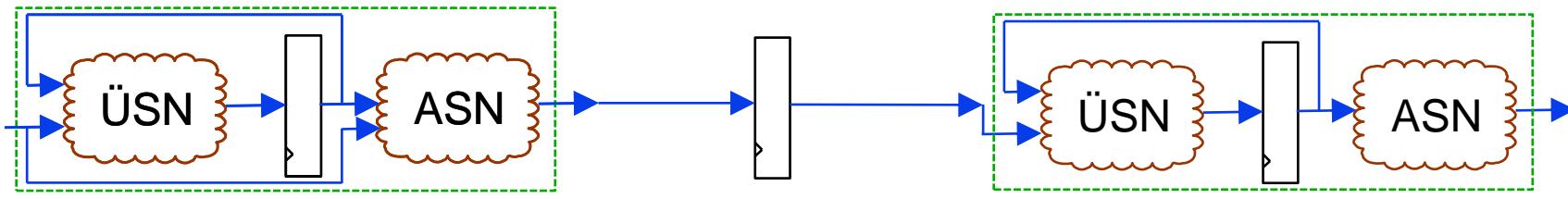
- Der "eine große Block" ist konsequent Mealy
- Der "andere große Block" ist konsequent Moore
- Wie die beiden Blöcke zusammenbauen?

"Trivialer Automat"

- Wenn ÜSN und ASN die identische Abbildung sind, dann "vereinfachen" sich sowohl Mealy- als auch Moore-Automat zu einer Abtaktstufe
- Eine Abtaktstufe/Register/FF "allein" ist also sowohl Mealy- als auch Moore-Automat
- Eine Abtaktstufe allein ist der trivialste Zustands-behaftete Automat

Wie Mealy und Moore koppeln? - Z.B.

Ein Register dazwischen schalten



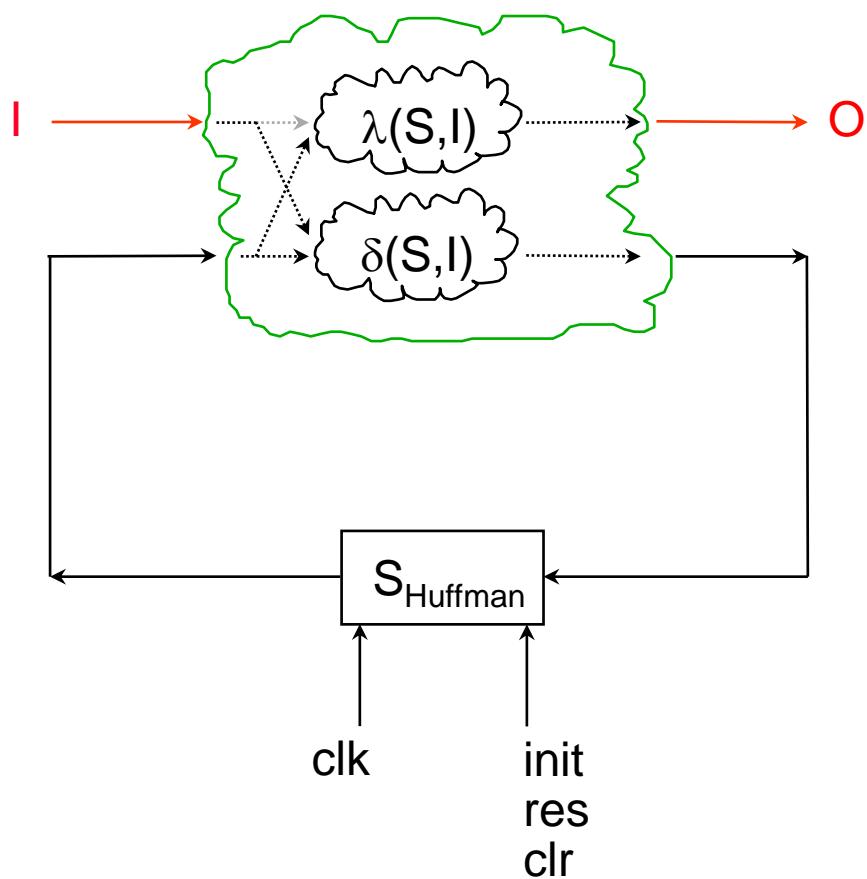
Mealy und Moore in Huffman-Normalform

Denke oft angenehm

resultiert (zunächst) in 2 Prozesse

1x kombinatorisch

1x sequentiell



Mealy und Moore in **Huffman-Normalform** als Pseudo VHDL Code

```
cobilo: process ( I_s, S_cs ) is
begin
    -- Überführungsfunktion
    S_ns <=  $\delta$ ( S_cs, I_s );
    -- Ausgabefunktion
    O_s <=  $\lambda$ ( S_cs, I_s );
end process cobilo;
```

```
sequlo: process ( clk ) is
begin
    if clk='1' and clk'event then
        S_cs <= S_ns;
    end if;
end process sequlo;
```

Mealy

```
cobilo: process ( I_s, S_cs ) is
begin
    -- Überführungsfunktion
    S_ns <=  $\delta$ ( S_cs, I_s );
    -- Ausgabefunktion
    O_s <=  $\lambda$ ( S_cs );
end process cobilo;
```

```
sequlo: process ( clk ) is
begin
    if clk='1' and clk'event then
        S_cs <= S_ns;
    end if;
end process sequlo;
```

Moore

ÜSN

- Wie sieht eigentlich ein Mealy-/Moore-ÜSN aus? (Im allgemeisten Fall)
- Was kann man sich da vorstellen?
- Was passiert überhaupt?
- Was macht " $\delta(\ s_cs, \ I_s \)$ " eigentlich?

- Abhängig vom aktuellem Zustand und aktueller Eingabe wird der nächste Zustand bestimmt
- 1.Idee: (Werte-)Tabelle $\text{Bits}^n \rightarrow \text{Bits}^m$
 - Ein Bit-genau beschriebener (aktueller) Zustand und eine Bit-genau beschriebene Eingabe werden wegen Determininus auf einen jeweils eindeutigen Bit-genau beschriebenen (neuen) Zustand abgebildet
 - Jedoch schnell "sehr viele" Bits
 - Diese Tabelle ist schnell kein praktisch "gehbarer" Weg
- Alternativ-Idee: Auf Bit-Ebene zwischen Kontroll- und Daten-Bits unterscheiden
 - Grund-/Start-/Typische Gedanke:
(Kontroll-)Zustand bestimmt wie (Daten-)Eingabe verarbeitet wird.
Achtung: Es gibt auch Daten-Register sowie "Kontroll"-Eingangs-Signale
 - Zurück zum Grund-Gedanken:
 - Der Zustand bestimmt die Berechnung (also wie die Eingabe verarbeitet wird)
Kontrolle bestimmt/kontrolliert Berechnung der Daten
 - Abhängig vom Zustand (Kontrolle) können sich die Berechnungen unterscheiden
 - Abhängig vom Zustand (Kontrolle) wird die konkrete Berechnung ausgewählt
 - Mehrwege-Fallunterscheidung abhängig vom Zustand (Kontrolle)

ÜSN

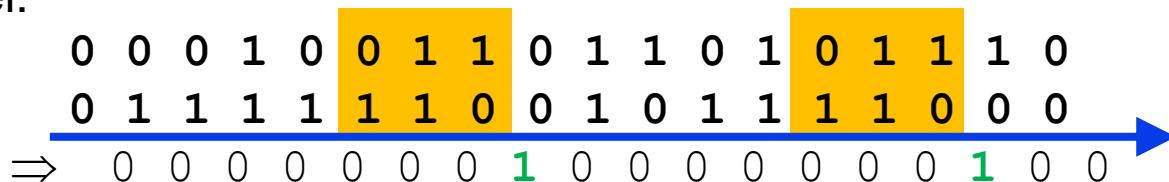
- Mehrwege-Multiplexer entspricht Mehrwege-Fallunterscheidung
- "Select" des Mehrwege-Multiplexers ist abhängig vom Zustand
- Eingänge des Mehrwege-Multiplexers sind die unterschiedlichen Berechnungen
- Über "aktueller Zustand"=Select wird Ergebnis des Mehrwege-Multiplexers ausgewählt
- Ergebnis ist Wert für neuen Zustand

ASN

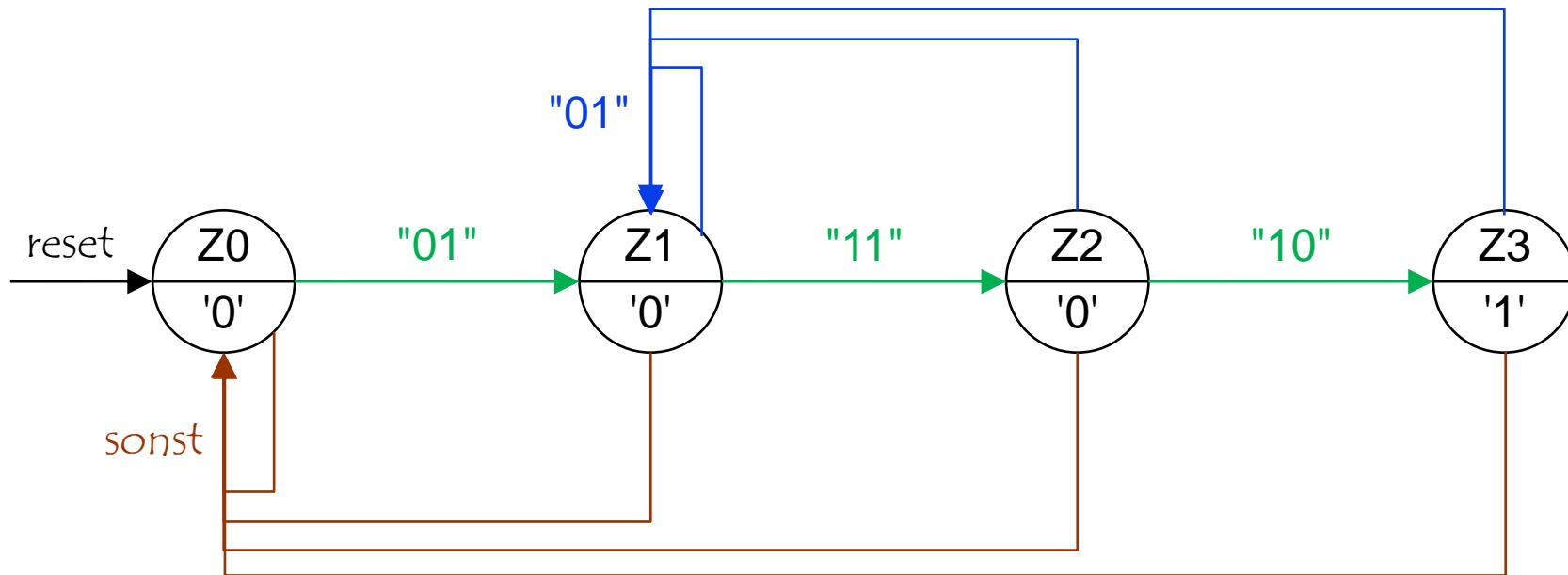
- Analog zu ÜSN - jedoch Unterscheidung zwischen Mealy und Moore,
da
 - Mealy: $O_s \leq \delta(s_{cs}, I_s);$
 - Moore: $O_s \leq \lambda(s_{cs});$

Übung Moore (1 → 3 → 2)

- Sequenz im 2-Bit-Bus erkennen
- "01" → "11" → "10" (oder als Integer 1 → 3 → 2)
- Wenn Sequenz auftrat 1 Takt lang '1' melden, sonst immer '0'
- Moore-Automat
- Beispiel:



Lösung Moore



```
process( i_s, z_cs ) is
```

```
    variable z_v      : geeigneter Typ ;
    variable panic_v : geeigneter Typ ;
```

```
begin
```

```
    panic_v := '0';
```

```
    case z_cs is
```

```
        when "00" =>
```

```
            if ( i_s = "01" ) then
                z_v := "01";
            else
                z_v := "00";
            end if;
```

```
-- ]when
```

```
        when "01" =>
```

```
            if ( i_s = "11" ) then
                z_v := "10";
            elsif ( i_s = "01" ) then
                z_v := "01";
            else
                z_v := "00";
            end if;
```

```
-- ]when
```

Lösung Moore 1

-- default - alles ok

-- es wird auf "01" gewartet

-- es wird auf "11" gewartet

Lösung Moore 2

```
...
when "10" =>                                -- es wird auf "10" gewartet
    if ( i_s = "10" ) then
        z_v := "11";
    elsif ( i_s = "01" ) then
        z_v := "01";
    else
        z_v := "00";
    end if;
-- ]when

when "11" =>                                -- neues Spiel / es wird wieder auf "01" gewartet
    if ( i_s = "01" ) then
        z_v := "01";
    else
        z_v := "00";
    end if;
-- ]when

when others =>
    panic_v := '1';
    z_v := "00";
-- ]when

end case;

panic_s <= panic_v;
z_ns <= z_v;

end process ÜSN;
```

Lösung Moore 3

...

ASN:

```
process( z_cs ) is
    variable found_v : geeigneter Typ ;
begin
    if ( z_cs = "11" ) then
        found_v := '1';
    else
        found_v := '0';
    end if;

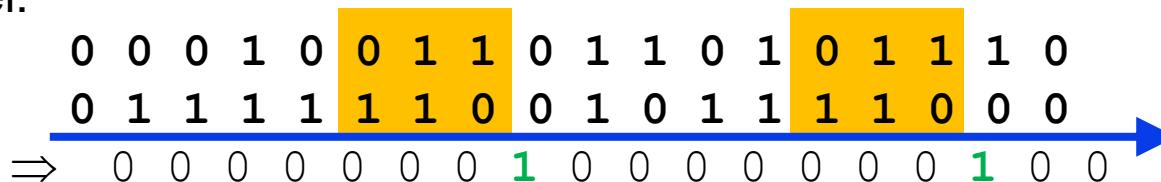
    found_s <= found_v;
end process ASN;
```

sequlo:

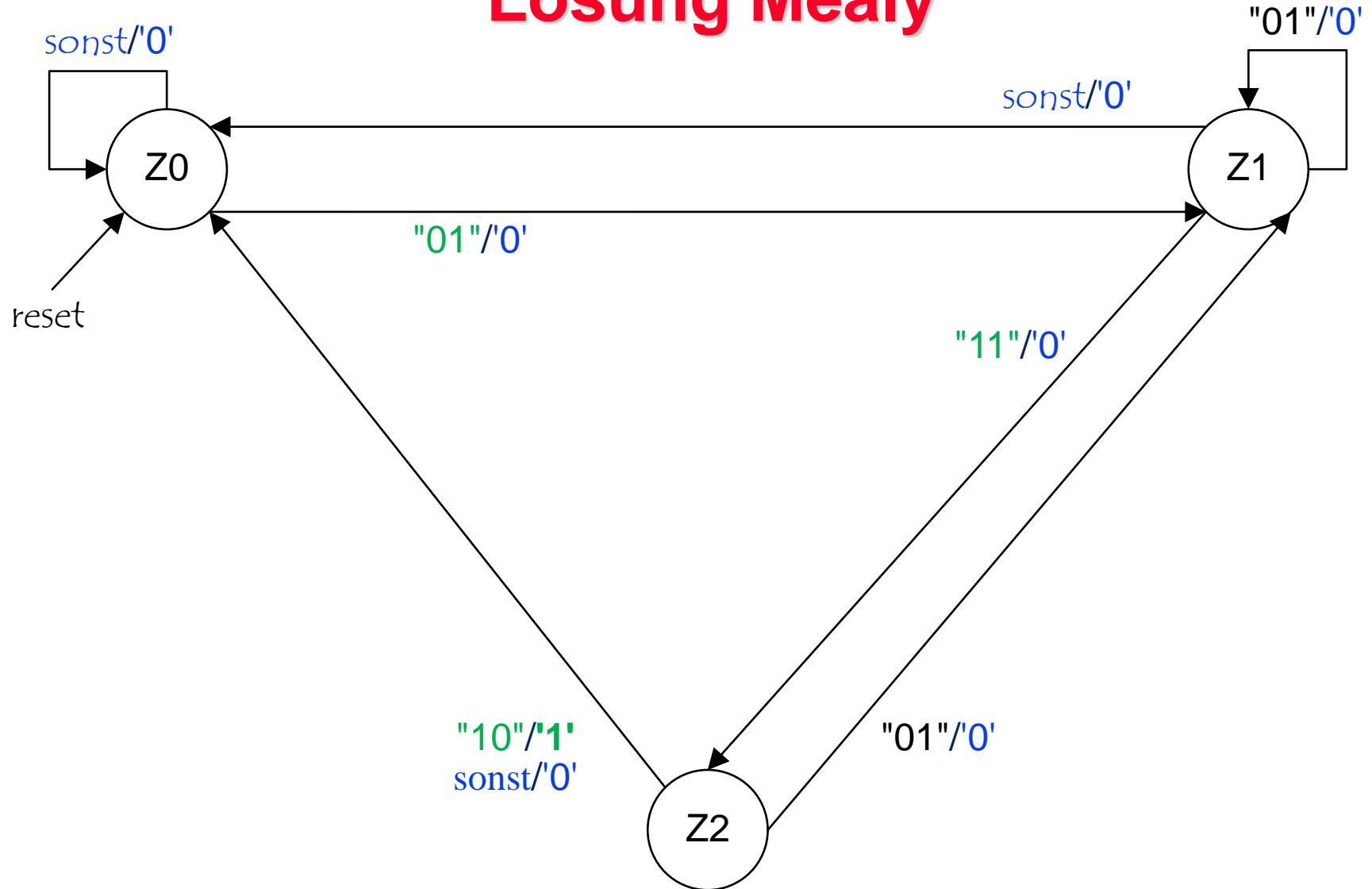
```
process ( clk ) is
begin
    if ( clk'event and clk = '1' ) then
        if ( sres_s = '1' ) then
            z_cs <= ( others => '0' );
        else
            z_cs <= z_ns;
        end if;
    end if;
end process sequlo;
...
```

Übung Mealy (1 → 3 → 2)

- Sequenz im 2-Bit-Bus erkennen
- "01" → "11" → "10" (oder als Integer 1 → 3 → 2)
- Wenn Sequenz auftrat 1 Takt lang '1' melden, sonst immer '0'
- Mealy-Automat
- Beispiel:



Lösung Mealy



...

SN:

```
process( i_s, z_cs, load_s ) is
    variable z_v      : geeigneter Typ ;
    variable found_v : geeigneter Typ ;
    variable panic_v : geeigneter Typ ;
begin
    panic_v := '0';                                -- default
    found_v := '0';                                -- default
    case z_cs is
        when "00" =>
            if ( i_s = "01" ) then
                z_v := "01";
            else
                z_v := "00";
            end if;
        -- ]when

        when "01" =>
            if ( i_s = "11" ) then
                z_v := "10";
            elsif ( i_s = "01" ) then
                z_v := "01";
            else
                z_v := "00";
            end if;
        -- ]when
```

Lösung Mealy 1

-- default
-- default

-- es wird auf "01" gewartet
-- Sequenz bis "01" erkannt

-- noch auf der Suche

-- es wird auf "11" gewartet
-- Sequenz bis "11" erkannt

-- Sequenz bis "01" erkannt
-- Sequenz "verloren"

Lösung Mealy 2

```
...
when "10" =>
    if ( i_s = "10" ) then      -- es wird auf "10" gewartet
        z_v := "00";
        found_v := '1';
    elsif ( i_s = "01" ) then   -- Sequenz erkannt und neue Suche
        z_v := "01";
    else
        z_v := "00";
    end if;
--]when

when others =>
    panic_v := '1';
    z_v := "00";
--]when
end case;

panic_s <= panic_v;
found_s <= found_v;
if ( load_s = '1' ) then    z_ns <= z_v;    else    z_ns <= z_cs;    end if;
end process SN;
...
```

Lösung Mealy 3

```
...
sequlo:
process ( clk ) is
begin
    if ( clk'event and clk = '1' ) then
        if ( sres_s = '1' ) then
            z_cs <= ( others => '0' );
        else
            z_cs <= z_ns;
        end if;
    end if;
end process sequlo;
...
```

Alternative1 / Bemerkung zur Lösung 2 bzw. "Sauber" Mealy

```
...
when "10" =>                                -- es wird auf "10" gewartet

    -- ÜSN
    if ( i_s = "01" ) then                  -- Sequenz bis "01" erkannt
        z_v := "01";
    else
        z_v := "00";
    end if;

    -- ASN
    if ( i_s = "00" ) then                  -- Sequenz erkannt
        found_v := '1';
    end if;

-- ]when
...
```

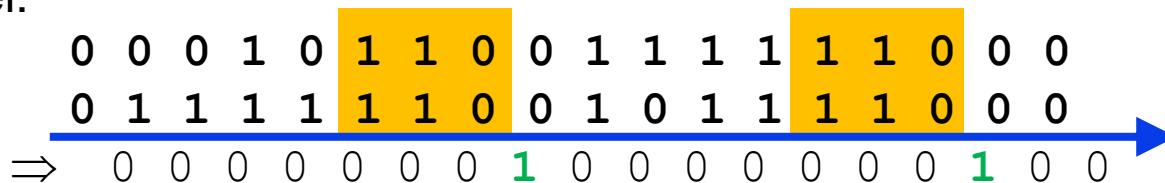
Alternative2 / Bemerkung zur Lösung 2 bzw. "Sauber" Mealy

```
...
-- ÜSN
case z_cs is
    when "00" => ...
    when "01" => ...
    when "10" => ...
    ...
end case;

-- ASN
if ( z_cs = "10" and i_s = "00" ) then          -- Sequenz erkannt
    found_v := '1';
end if;
...
```

Übung (3 → 3 → 0)

- Sequenz im 2-Bit-Bus erkennen
- "11" → "11" → "00" (oder als Integer 3 → 3 → 0)
- Wenn Sequenz auftrat 1 Takt lang '1' melden, sonst immer '0'
- Moore-&Mealy-Automat
- Beispiel:



weiteres BSP.
zum selber
üben

```
process( i_s, z_cs, load_s ) is
    variable z_v      : geeigneter Typ ;
    variable panic_v : geeigneter Typ ;
```

begin

```
    panic_v := '0';
```

```
    case z_cs is
```

```
        when "00" =>
```

```
            if i_s = "11" then
                z_v := "01";
            else
                z_v := "00";
            end if;
```

```
-- ]when
```

```
        when "01" =>
```

```
            if i_s = "11" then
                z_v := "10";
            else
                z_v := "00";
            end if;
```

```
-- ]when
```

```
    ...
```

Lösung Moore 1

Weiteres BSP.
zum selber
üben

Lösung Moore 2

```
...
when "10" =>                                -- es wird auf "00" gewartet
    if i_s = "00" then
        z_v := "11";
    elsif i_s = "11" then
        z_v := "10";
    else
        z_v := "00";
    end if;
--]when

when "11" =>                                -- neues Spiel / es wird wieder auf "11" gewartet
    if i_s = "11" then
        z_v := "01";
    else
        z_v := "00";
    end if;
--]when

when others =>
    panic_v := '1';
    z_v := "00";
--]when

end case;

panic_s <= panic_v;
if load_s = '1' then    z_ns <= z_v;    else    z_ns <= z_cs;    end if;
end process ÜSN;
```

weiteres
zum selber
üben

Lösung Moore 3

...

ASN:

```
process( z_cs ) is
    variable found_v : geeigneter Typ ;
begin
    if ( z_cs = "11" ) then
        found_v := '1';
    else
        found_v := '0';
    end if;

    found_s <= found_v;
end process ASN;
```

sequlo:

```
process ( clk ) is
begin
    if clk'event and clk = '1'  then
        if sres_s = '1'  then
            z_cs <= ( others => '0' );
        else
            z_cs <= z_ns;
        end if;
    end if;
end process sequlo;
...
```

Weiteres BSP
zum selber
üben

...

SN:

```
process( i_s, z_cs, load_s ) is
    variable z_v      : geeigneter Typ ;
    variable found_v : geeigneter Typ ;
    variable panic_v : geeigneter Typ ;
begin
    panic_v := '0';                      -- default
    found_v := '0';                      -- default
    case z_cs is
        when "00" =>
            if i_s = "11" then
                z_v := "01";
            else
                z_v := "00";
            end if;
        -- ]when

        when "01" =>
            if i_s = "11" then
                z_v := "10";
            else
                z_v := "00";
            end if;
        -- ]when
```

Lösung Mealy 1

Weiteres BSP
zum selber
üben

```

...
when "10" =>                                -- es wird auf "00" gewartet
    if i_s = "00" then
        z_v := "00";
        found_v := '1';
    elsif i_s = "11" then
        z_v := "10";
    else
        z_v := "00";
    end if;
-- ]when

when others =>                            -- macht jetzt auch Sinn für bit - "11" nicht auscodiert
    panic_v := '1';
    z_v := "00";
-- ]when

end case;

panic_s <= panic_v;
found_s <= found_v;
if load_s = '1' then      z_ns <= z_v;    else    z_ns <= z_cs;    end if;
end process SN;
...

```

Lösung Mealy 2

weiteres BSP
zum selber
üben

Lösung Mealy 3

```
...
sequlo:
process ( clk ) is
begin
    if  clk'event and clk = '1'  then
        if  sres_s = '1'  then
            z_cs <= ( others => '0' );
        else
            z_cs <= z_ns;
        end if;
    end if;
end process sequlo;
...
```

Weiteres BSP.
zum selber
üben

Alternative1 / Bemerkung zur Lösung 2 bzw. "Sauber" Mealy

```
...
when "10" =>                                -- es wird auf "00" gewartet

    -- ÜSN
    if i_s = "11" then
        z_v := "10";
    else
        z_v := "00";
    end if;

    -- ASN
    if i_s = "00" then
        found_v := '1';
    end if;

-- ]when
...
```

weiteres BSP.
zum selber
üben

Alternative2 / Bemerkung zur Lösung 2 bzw. "Sauber" Mealy

```
...
-- ÜSN
case z_cs is
    when "00" => ...
    when "01" => ...
    when "10" => ...
    ...
end case;

-- ASN
if z_cs = "10" and i_s = "00"  then
    found_v := '1';
end if;
...
```

weiteres BSP.
zum selber
üben

Übung (2 → 5 → 5)

- Sequenz im 3-Bit-Bus erkennen
- "010" → "101" → "101" (oder als Integer 2 → 5 → 5)
- Wenn Sequenz auftrat 1 Takt lang '1' melden, sonst immer '0'
- Moore-&Mealy-Automat
- Beispiel:

0	0	0	1	0	0	1	1	0	1	1	0	1	1	1	0
0	1	1	1	1	1	0	0	0	1	0	1	1	1	0	0
0	1	1	1	1	1	0	1	1	0	1	0	1	1	0	0
⇒	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0

weiteres BSP.
zum selber
üben

```
process( i_s, z_cs, load_s ) is
    variable z_v      : geeigneter Typ ;
    variable panic_v : geeigneter Typ ;
```

begin

```
    panic_v := '0';
```

```
    case z_cs is
```

```
        when "00" =>
```

```
            if i_s = "010" then
                z_v := "01";
            else
                z_v := "00";
            end if;
```

```
-- ]when
```

```
        when "01" =>
```

```
            if i_s = "101" then
                z_v := "10";
            elsif i_s = "010" then
                z_v := "01";
            else
                z_v := "00";
            end if;
```

```
-- ]when
```

Lösung Moore 1

-- default - alles ok

-- es wird auf "010" gewartet

-- es wird auf "101" gewartet

weiteres BSP.
zum selber
üben

Lösung Moore 2

```
...
when "10" =>                                -- es wird auf "101" gewartet
    if i_s = "101"  then
        z_v := "11";
    elsif i_s = "010"  then
        z_v := "01";
    else
        z_v := "00";
    end if;
--]when

when "11" =>                                -- neues Spiel / es wird wieder auf "010" gewartet
    if i_s = "010"  then
        z_v := "01";
    else
        z_v := "00";
    end if;
--]when

when others =>
    panic_v := '1';
    z_v := "00";
--]when
end case;

panic_s <= panic_v;
if load_s = '1'  then  z_ns <= z_v;  else  z_ns <= z_cs;  end if;
end process ÜSN;
```

weiteres
zum selber
üben

Lösung Moore 3

...

ASN:

```
process( z_cs ) is
    variable found_v : geeigneter Typ ;
begin
    if  z_cs = "11"  then
        found_v := '1';
    else
        found_v := '0';
    end if;

    found_s <= found_v;
end process ASN;
```

sequlo:

```
process ( clk ) is
begin
    if  clk'event and clk = '1'  then
        if  sres_s = '1'  then
            z_cs <= ( others => '0' );
        else
            z_cs <= z_ns;
        end if;
    end if;
end process sequlo;
...
```

weiteres BSP
zum selber
üben

Lösung Mealy 1

```
...  
SN:  
process( i_s, z_cs, load_s ) is  
    variable z_v      : geeigneter Typ ;  
    variable found_v : geeigneter Typ ;  
    variable panic_v : geeigneter Typ ;  
begin  
    panic_v := '0';                      -- default  
    found_v := '0';                      -- default  
    case z_cs is  
  
        when "00" =>                  -- es wird auf "010" gewartet  
            if i_s = "010"  then  
                z_v := "01";  
            else  
                z_v := "00";  
            end if;  
        --]when  
  
        when "01" =>                  -- es wird auf "101" gewartet  
            if i_s = "101"  then  
                z_v := "10";  
            elsif i_s = "010"  then  
                z_v := "01";  
            else  
                z_v := "00";  
            end if;  
        --]when  
    ...
```

Weiteres BSP.
zum selber
üben

```

...
when "10" =>                                -- es wird auf "101" gewartet
    if i_s = "101" then
        z_v := "00";
        found_v := '1';
    elsif i_s = "010" then
        z_v := "01";
    else
        z_v := "00";
    end if;
-- ]when

when others =>                            -- macht jetzt auch Sinn für bit - "11" nicht auscodiert
    panic_v := '1';
    z_v := "00";
-- ]when

end case;

panic_s <= panic_v;
found_s <= found_v;
if load_s = '1' then      z_ns <= z_v;    else    z_ns <= z_cs;    end if
end process SN;
...

```

Lösung Mealy 2

Weiteres BSP
zum selber
üben

Lösung Mealy 3

```
...
sequlo:
process ( clk ) is
begin
    if  clk'event and clk = '1'  then
        if  sres_s = '1'  then
            z_cs <= ( others => '0' );
        else if
            z_cs <= z_ns;
        end if;
    end if;
end process sequlo;
...
```

Weiteres BSP.
zum selber
üben

Alternative1 / Bemerkung zur Lösung 2 bzw. "Sauber" Mealy

```
...
when "10" =>                                -- es wird auf "101" gewartet

    -- ÜSN
    if i_s = "010" then
        z_v := "01";
    else
        z_v := "00";
    end if;

    -- ASN
    if i_s = "101" then
        found_v := '1';
    end if;

-- ]when
...
```

weiteres BSP.
zum selber
üben

Alternative2 / Bemerkung zur Lösung 2 bzw. "Sauber" Mealy

```
...
-- ÜSN
case z_cs is
    when "00" => ...
    when "01" => ...
    when "10" => ...
    ...
end case;

-- ASN
if z_cs = "10" and i_s = "101" then
    found_v := '1';
end if;
...
```

weiteres BSP.
zum selber
üben

Diskussion: Idee mit FIFO

- Was wäre wenn "FIFO für Eingangsdaten" und "im FIFO nachschauen"?
- 3x3 FIFO kostet 9 "Zustands-Bits"
- Mealy-Variante hat 3 Zustände bzw. 2 Zustands-Bits - ist also günstiger
- FIFO-Idee ist vermutlich "einfacher" aber schnell deutlich teurer in HW-Kosten
- Projekte insbesondere im "Consumer Electronic"-Bereich sind sehr oft Stückkosten-getrieben

Bemerkung zur Synthese

- Synthese ist sehr aufwendig (NP-Vollständig)
- Synthese nutzt heuristische Verfahren
- Es gibt ein Zeit-Budget
- Als RTL-Designer oft Denke, dass ist Job des Synthese-Tools aus dem VHDL-Code was "Ordentliches" zu machen
- Damit das funktioniert, sollte/darf das Zeit-Budget nicht unnötig belastet werden
- Es ist wichtig, der Synthese nicht eine falsche/schlechte "Startidee" mitzugeben

Übung (Prioritäten)

- Es soll einer von vier Eingangswerten ausgewählt werden
 - Die Auswahl soll abhängig von 3 Bedingungen erfolgen
 - Wenn die 1.Bedingung erfüllt ist, soll der 1.Wert genommen werden
 - Andernfalls, wenn die 2.Bedingung erfüllt ist, soll der 2.Wert genommen werden
 - Andernfalls, wenn die 3.Bedingung erfüllt ist, soll der 3.Wert genommen werden
 - Andernfalls soll der 4.Wert genommen werden
-
- Wie sieht der VHDL-Code aus?
 - Wie sieht die HW-Idee vor der "Optimierung" aus?

Prioritäten / Mux-Kaskade

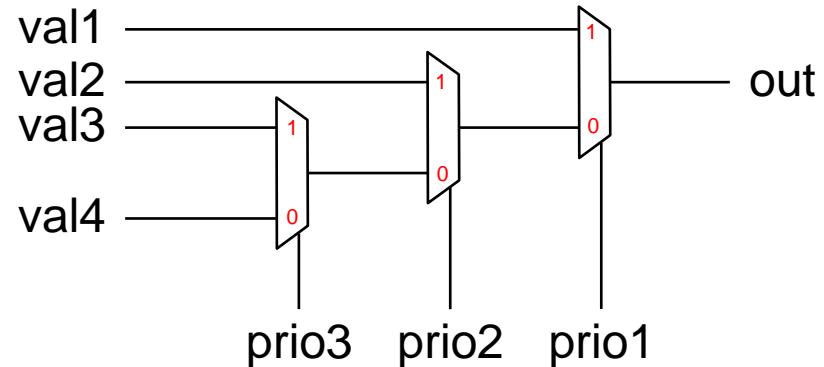
...
Beispiel:

```
process( ... ) is
  ...
begin
  if    prio1 = '1'  then
    out_v := val1_?;
  elsif  prio2 = '1'  then
    out_v := val2_?;
  elsif  prio3 = '1'  then
    out_v := val3_?;
  else
    out_v := val4_?;
end if;
```

```
  out_s <= out_v;
end process Beispiel;
...
```

val1 hat höchste Priorität wird am schnellsten "berechnet" bzw. durchläuft am schnellsten Mux-Kaskade
=> val1 kann später "stabil werden"
val4 (& val3) haben niedrigste Priorität - werden am langsamsten "berechnet"
=> val4(&val3). müssen sich früher stabilisieren

HW-Idee vor der Synthese



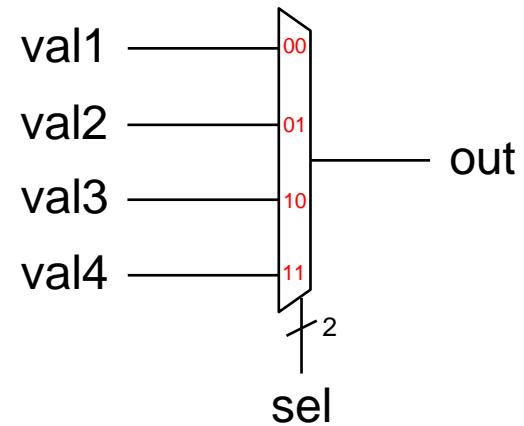
Übung (Keine Prioritäten)

- Es soll einer von vier Eingangswerten ausgewählt werden
 - Es gibt keine Prioritäten
-
- Wie sieht der VHDL-Code aus?
 - Wie sieht die HW-Idee vor der Synthese aus?

Mux (4:1)

```
...
Beispiel:
process( ... ) is
  ...
begin
  case sel is
    when "00" => out_v := val1_?;
    when "01" => out_v := val2_?;
    when "10" => out_v := val3_?;
    when "11" => out_v := val4_?;
  end case;
  out_s <= out_v;
end process Beispiel;
...
```

HW-Idee vor der Synthese



Übung

- Über einen 32 Bit Datenbus wird ein signed Wert gegeben (`in_s`)
- Führen Sie wahlweise eine Sign-Extension für die unteren
 - 8 der eingehende Wert ist im 8 Bit signed Wertebereich
 - 16 der eingehende Wert ist im 16 Bit signed Wertebereich
 - 24 der eingehende Wert ist im 24 Bit signed Wertebereich
 - 32 (bzw. reichen Sie die 32 Bit unverändert weiter)

Bit durch

- Wie sieht der VHDL-Code aus?
- Wie sieht die HW-Idee vor der Synthese aus?

...

Beispiel:

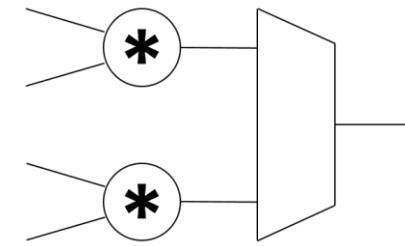
```
process( in_s, sel_s ) is
    ...
begin
    case sel_s is
        when "00" =>
            resu_v           := (others=>in_s(7));
            resu_v( 7 downto 0) := in_s(7 downto 0);
        when "01" =>
            resu_v           := (others=>in_s(15));
            resu_v(15 downto 0) := in_s(15 downto 0);
        when "10" =>
            resu_v           := (others=>in_s(23));
            resu_v(23 downto 0) := in_s(23 downto 0);
        when others =>
            resu_v           := in_s;
            -- including "11"
    end case;

    resu_s <= resu_v;
end process Beispiel;
...
```

Lösung

aufwendiger für Synthese

```
...  
Beispiel:  
process( ... ) is  
    ...  
begin  
    if ( sel = '1' ) then  
        resu_v := op1_? * op2_?;  
    else  
        resu_v := op3_? * op4_?;  
    end if;  
  
    resu_s <= resu_v;  
end process Beispiel;  
...
```

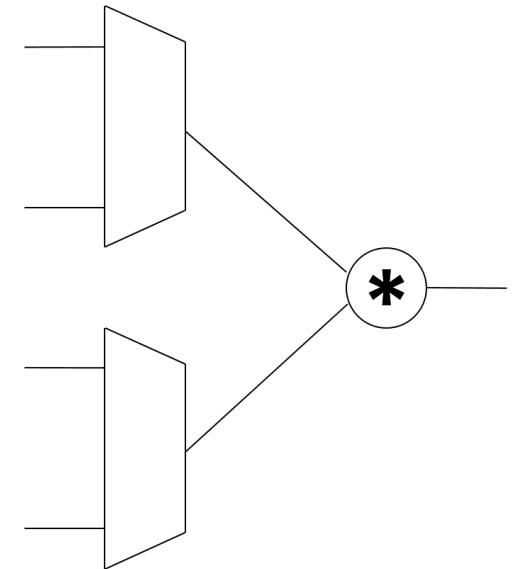


HW-Bedarf vor der Synthese(-Optimierung): 2 Multiplizierer

Multiplikation dient hier als Metapher für "etwas" aufwendiges (gemessen in HW-kosten)
Multiplikation ist "bereits so teuer", dass es diesbezüglich lohnt unnötige Kosten zu vermeiden

Synthese-freundlicher

```
...  
Beispiel:  
process( ... ) is  
  ...  
begin  
  if ( sel = '1' ) then  
    tmp1_v := op1_?;  
    tmp2_v := op2_?;  
  else  
    tmp1_v := op3_?;  
    tmp2_v := op4_?;  
  end if;  
  resu_v := tmp1_v * tmp2_v;  
  
  resu_s <= resu_v;  
end process Beispiel;  
...
```



HW-Bedarf vor der Synthese(-Optimierung): 1 Multiplizierer

aufwendiger für Synthese allgemein

```
...
Beispiel:
process( ... ) is
  ...
begin
  if ( sel = '1' ) then
    resu_v := complexeOperation( op1a_?, ..., opNa_? );
  else
    resu_v := complexeOperation( op1b_?, ..., opNb_? );
  end if;

  resu_s <= resu_v;
end process Beispiel;
...
```

Synthese-freundlicher allgemein

```
...
Beispiel:
process( ... ) is
  ...
begin
  if ( sel = '1' ) then
    tmp1_v := op1a_?;
    tmp2_v := op2a_?;
    ...
    tmpN_v := opNa_?;
  else
    tmp1_v := op1b_?;
    tmp2_v := op2b_?;
    ...
    tmpN_v := opNb_?;
  end if;
  resu_v := complexeOperation( tmp1_v, ..., tmpN_v );

  resu_s <= resu_v;
end process Beispiel;
...
```

Begriffe

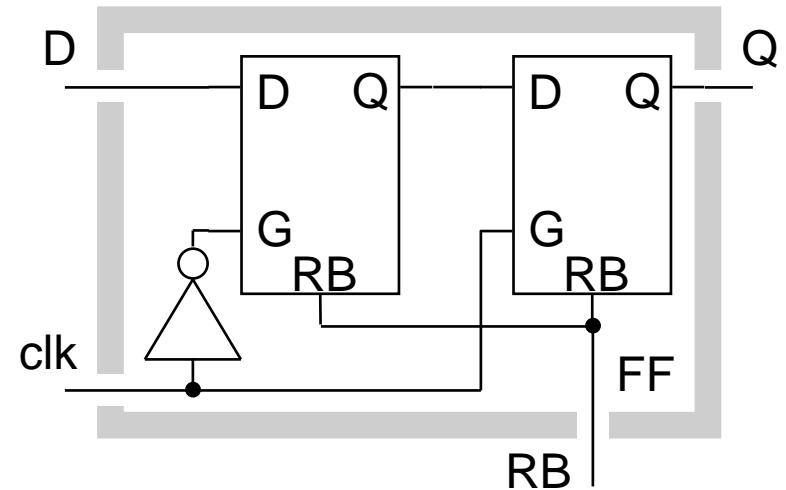
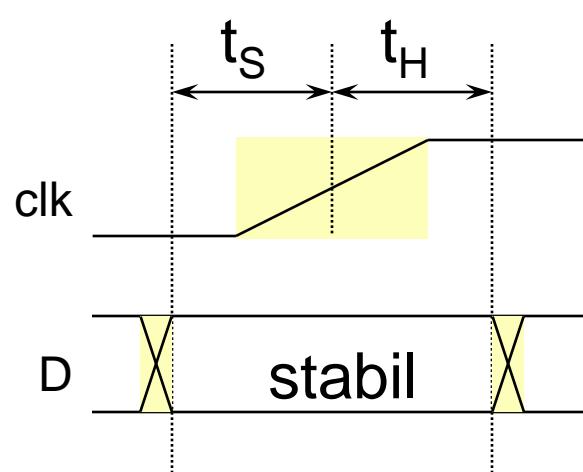
Achtung! (1)

- Designs bestehen (bezüglich sequentieller Logik typischer Weise fast nur) aus FFs
- CMOS (und nicht TTL) ist die beherrschende Technik in ASICs
- Latches sind
 - eher eine "Ausnahme"
 - komplizierter (bezüglich einer fehlerfreien Ansteuerung in einem korrektem Design)
 - behindern Vereinfachungen (z.B. einfache Denkmodelle wie diskrete Zeit)
- (Nicht nur) Anfänger sollten **Latches** innerhalb eines Chips/FPGAs/CPLDs meiden

Achtung! (2)

- Keine sequentielle Logik (Zustände) selber bauen !
Damit gleichbedeutend:
Keine Rückkopplungen in der kombinatorischen Logik.
Algorithmische Rückkopplungen laufen immer über Zustände/Sequentielle Logik.
- Immer auf existierende Zellen zurückgreifen
(bzw. der Synthese die Chance geben dies zu tun)
- Aber warum eigentlich?

Setup- und Hold-Time



- Achtung! Bei der positiven Taktflanke (clk: 0->1) sind für einen kurzen Moment beide Latches offen ($G=1$). Änderungen am Dateneingang sind in diesem Moment kritisch. Flankensteilheit und Treiberstärke der Signale am Daten- und Takteingang beeinflussen das Ergebnis. Diese können von dynamischen Effekten abhängig sein
- Daten müssen eine gewisse Zeit vor der Taktflanke stabil sein (**setup time**) und dürfen sich auch nach der Taktflanke für eine gewisse Zeit nicht ändern (**hold time**)
- In einer Bibliothekszelle liegt die interne Logik extrem dicht beieinander. In einer selbstgebauten Zelle ist dies typischer Weise nicht der Fall.

"Vollsynchrones Design"

- Name bedeutet zunächst:
 - Es gibt FFs/Register und diese "hängen" am Takt.
 - Alle Aktionen werden synchron zum Takt ausgeführt/angestartet.
- Im typischen Fall / Normalfall gilt:
 - Es gibt nur einen Takt
 - Alle FFs/Register hängen an diesem Takt
 - Alle FFs/Register arbeiten "positiv Flanken-gesteuert"
- Der entscheidende Punkt ist:
 - Aus einer "höheren" Abstraktionsschicht passiert die **Zustandsüberführung des "Systems"** zu einem Zeitpunkt.
- Aber wie schaut es in der Realität aus?

<BREAK>

- Die folgenden Folie "Codierungsvarianten" stehen im Zusammenhang mit den "Simple Gates"-Beispielen.
- Achtung in VHDL modelliert man zu 99,9999999999999% nicht so detiliert.
- Unterhalb der RTL-Ebene wird nicht modelliert

Codierungsvarianten

am Beispiel ein Volladdieres

Abkürzungen / Namen

r für Result (Ergebnis)

c für Carry (Übertrag)

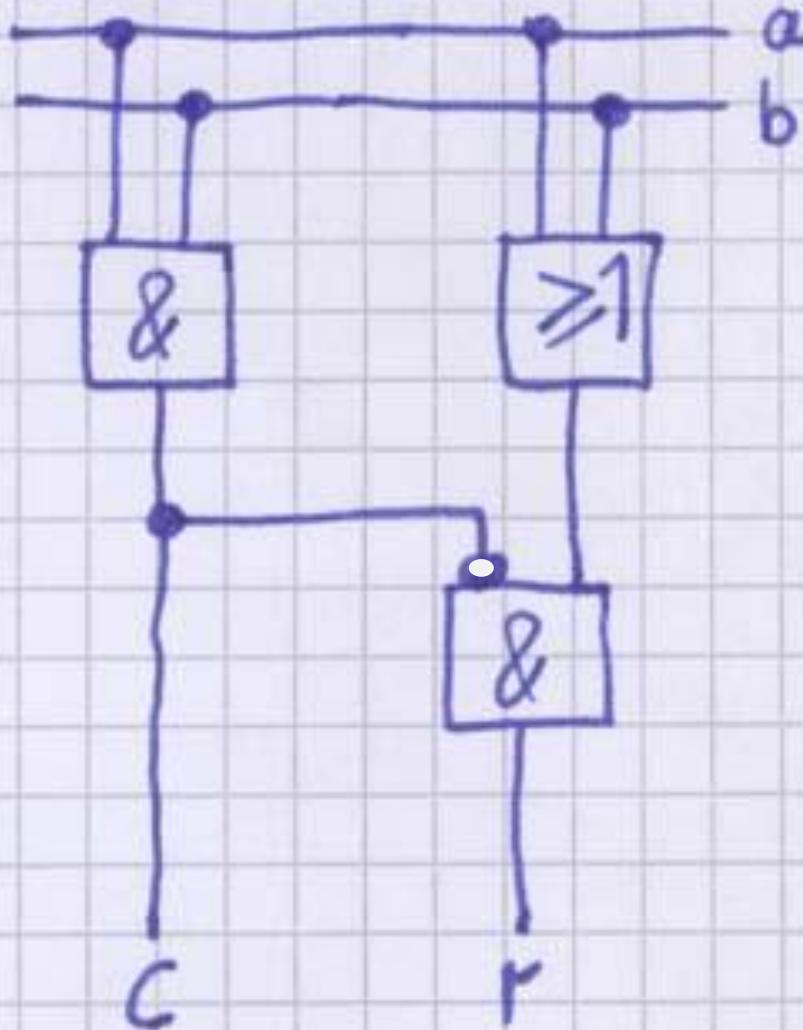
- Teilweise unterscheidet man zwischen
 - Carry (positiver Übertrag) und
 - Borrow (negativer Übertrag)

v für Overflow (Überlauf)

- Teilweise unterscheidet man zwischen
 - Overflow (positiver Überlauf) und
 - Underflow (Unterlauf / negativer Überlauf)
- **HA** / HA für Halbaddierer/Half Adder
- **VA** / FA für Volladdierer/Full Adder

Halbaddierer (HA)

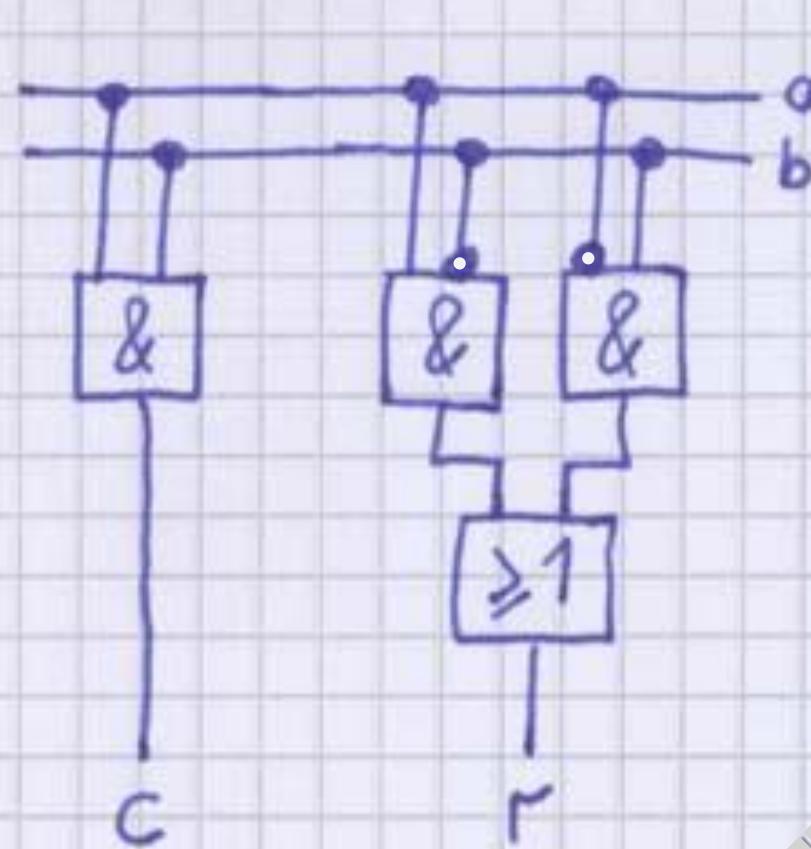
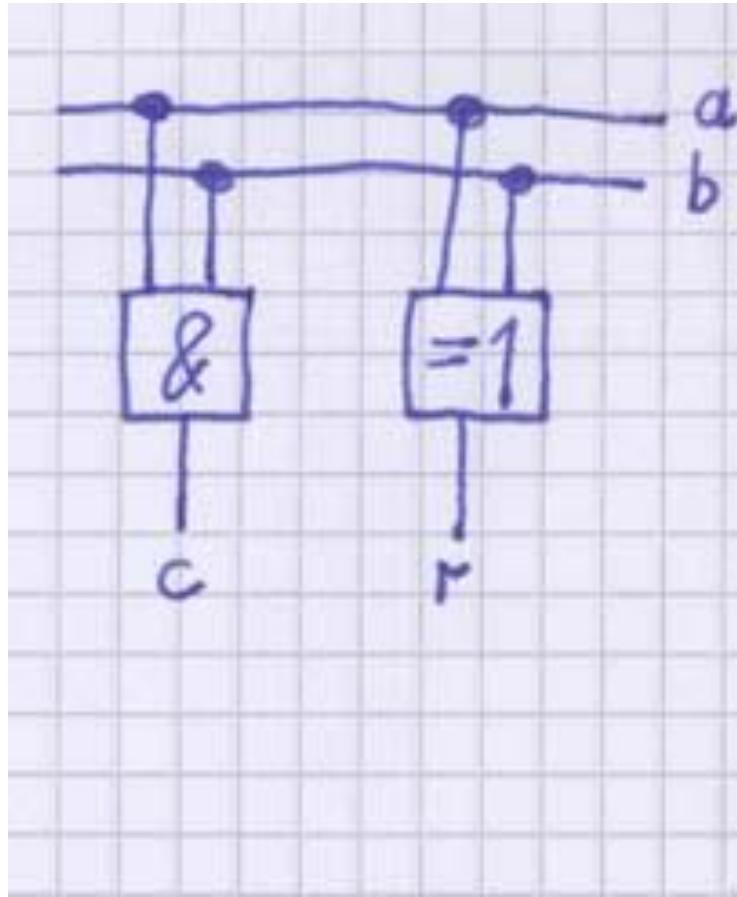
Halbaddierer



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

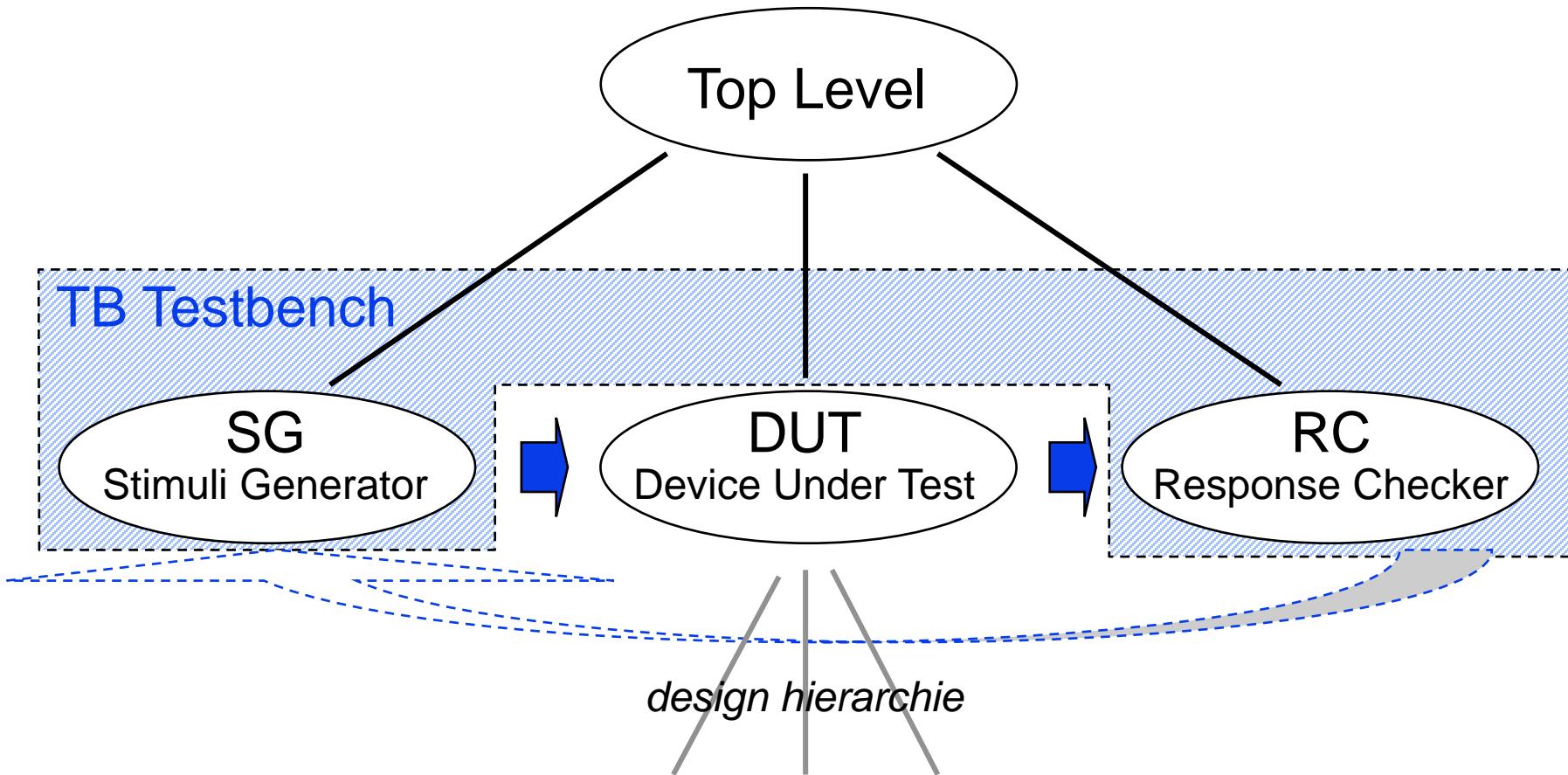


Halbaddierer (Varianten)



touched

Wdh.: Top Level / Testbench



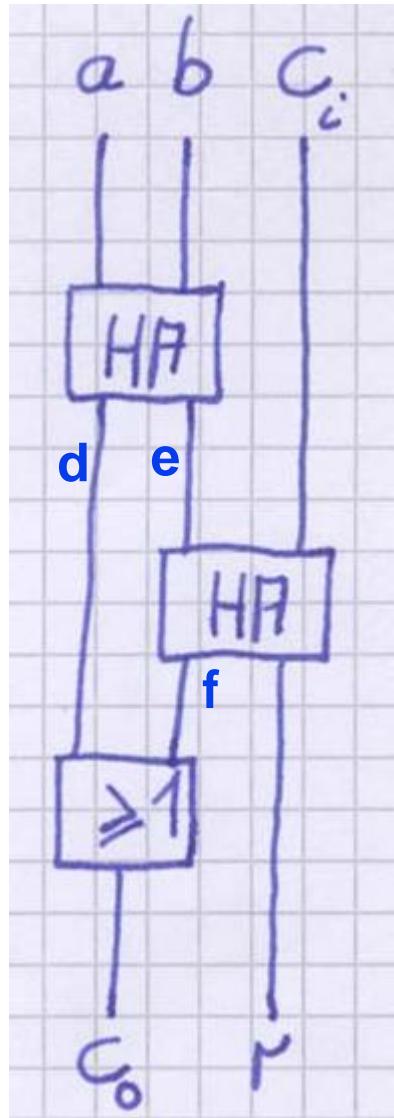
VHDL

Anschauen:

- `.../~/schafers/PUBLIC/EXAMPLEs/VHDL/exampleSimpleGates/and2.vhd`
- `.../~/schafers/PUBLIC/EXAMPLEs/VHDL/exampleSimpleGates/tb4and2.vhd`
- `.../~/schafers/PUBLIC/EXAMPLEs/VHDL/exampleSimpleGates/sg4gate2.vhd`
- `.../~/schafers/PUBLIC/EXAMPLEs/VHDL/exampleSimpleGates/xor2.vhd`
- `.../~/schafers/PUBLIC/EXAMPLEs/VHDL/exampleSimpleGates/tb4xor2.vhd`
- `.../~/schafers/PUBLIC/EXAMPLEs/VHDL/exampleSimpleGates/nand2.vhd`
- `.../~/schafers/PUBLIC/EXAMPLEs/VHDL/exampleSimpleGates/tb4nand2.vhd`
- **`.../~/schafers/PUBLIC/EXAMPLEs/VHDL/exampleSimpleGates/ha.vhd`**
- `.../~/schafers/PUBLIC/EXAMPLEs/VHDL/exampleSimpleGates/tb4ha.vhd`
- `.../~/schafers/PUBLIC/EXAMPLEs/VHDL/exampleSimpleGates/sg4gate2.vhd`
- `.../~/schafers/PUBLIC/EXAMPLEs/VHDL/exampleSimpleGates/rc4ha.vhd`

touched

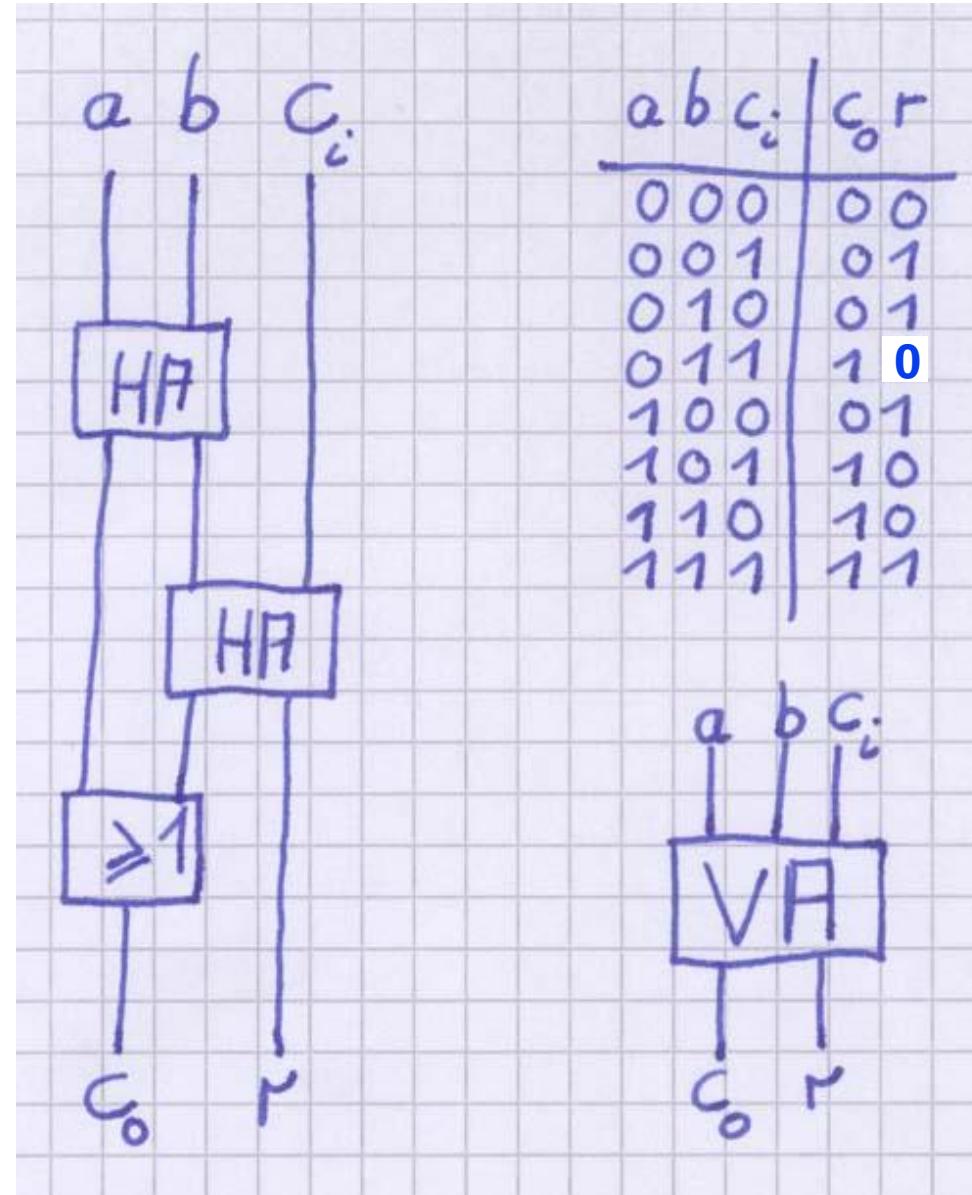
Volladdierer (VA)



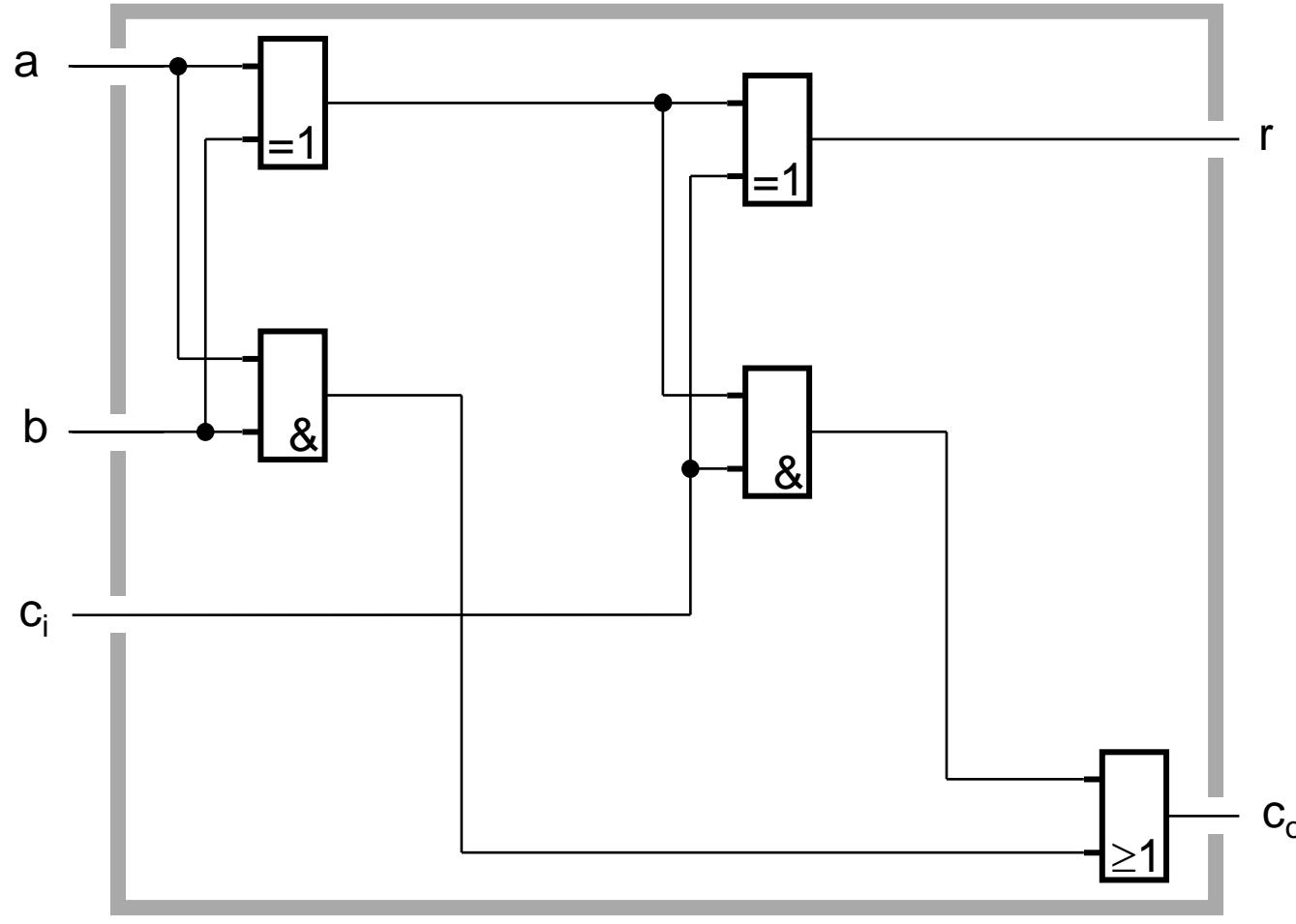
a	b	c_i	d	e	f	r	c_o	r
0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0	1
0	1	0	0	1	0	1	0	1
0	1	1	0	1	1	0	1	0
1	0	0	0	1	0	1	0	1
1	0	1	0	1	1	0	1	0
1	1	0	1	0	0	0	1	0
1	1	1	1	0	0	1	1	1

touched

Volladdierer (VA)

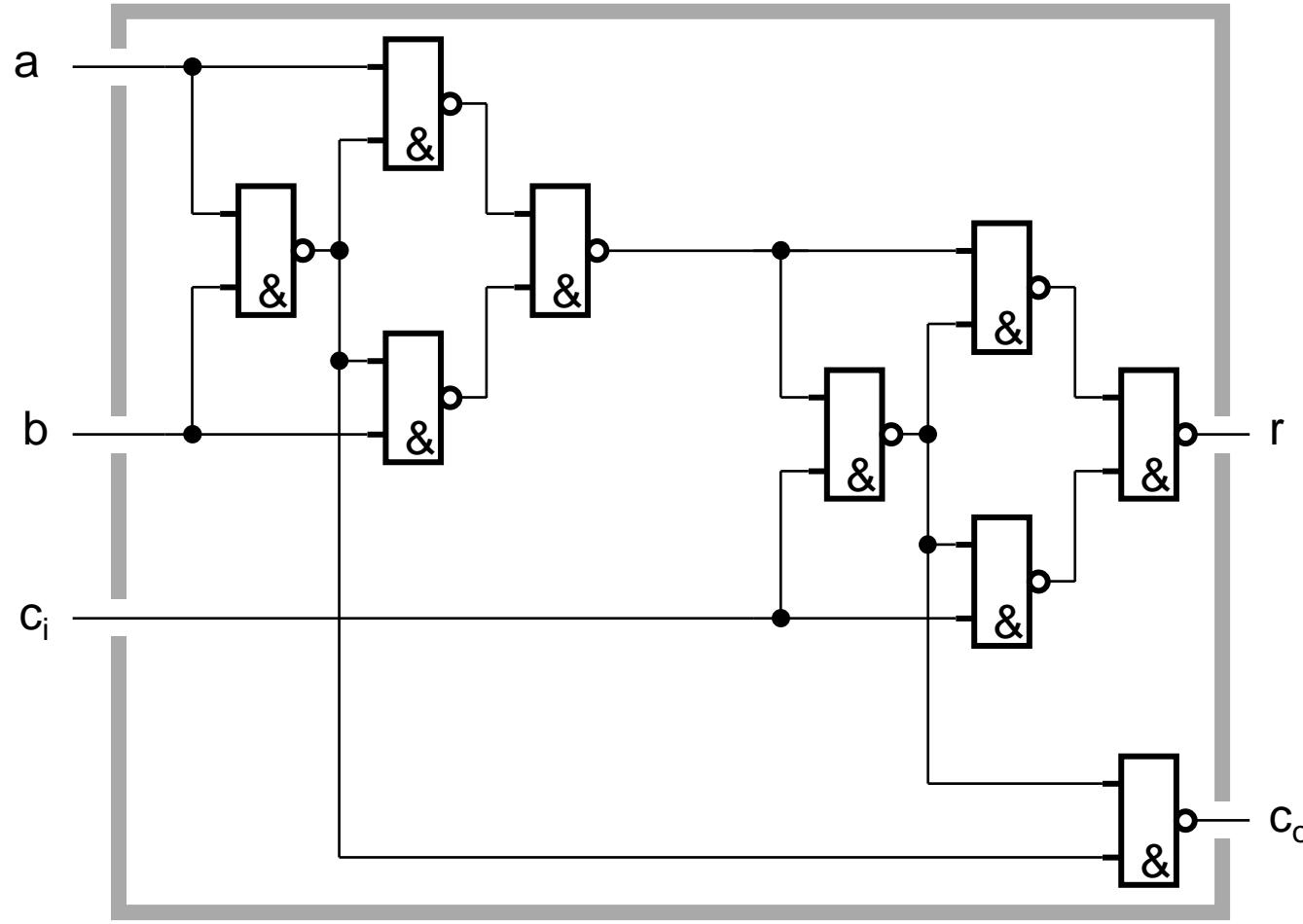


VA – Gatter-Realisierung (1)



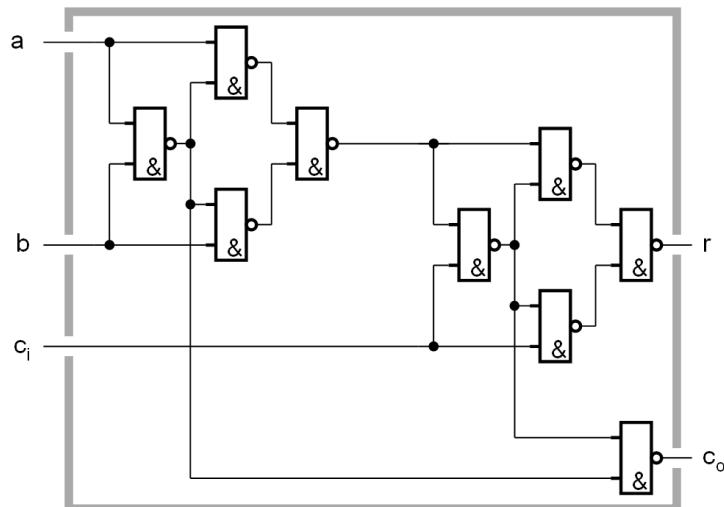
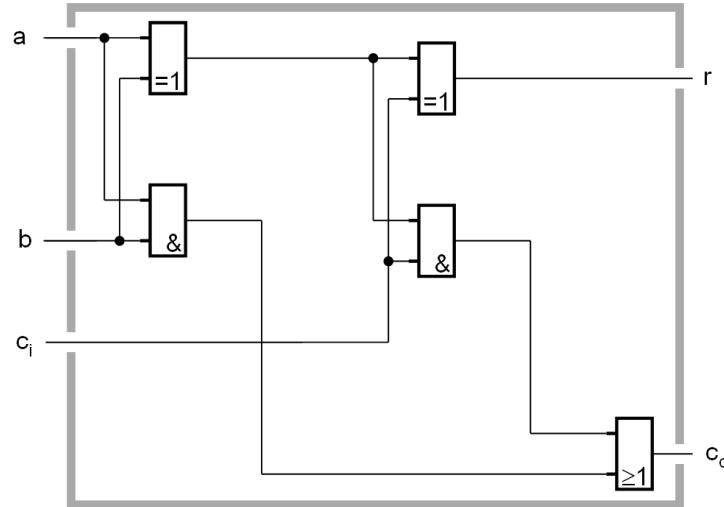
touched

VA – Gatter-Realisierung (2)



touched

Volladdierer (VA)



a b c_i r

HA

HA

≥ 1

c_o

r

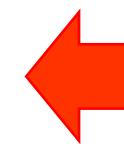
a	b	c_i	c_o	r
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



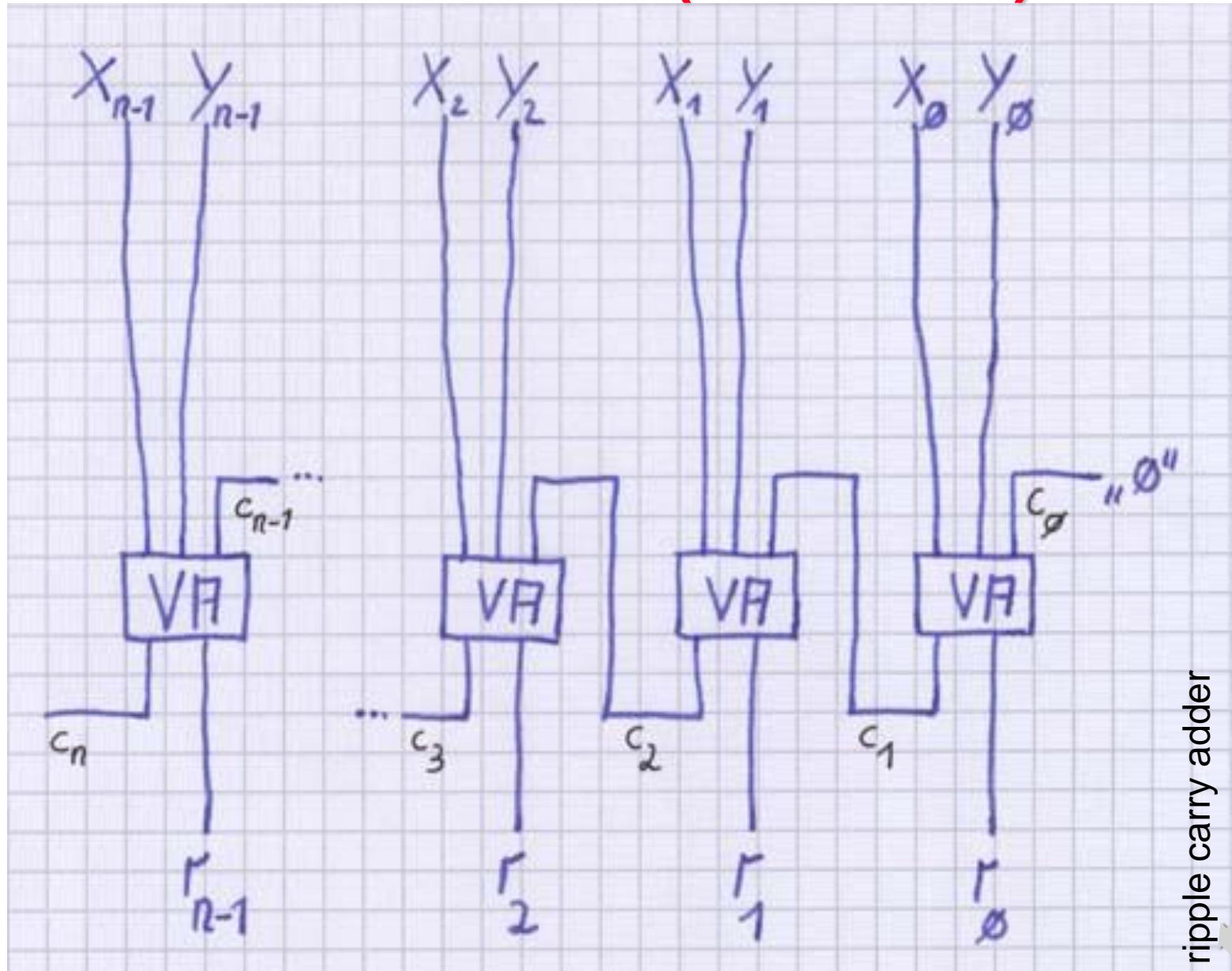
touched

VHDL

Anschauen:

- `...~/schafers/PUBLIC/EXAMPLEs/VHDL/exampleSimpleGates/fa.vhd` 
- `...~/schafers/PUBLIC/EXAMPLEs/VHDL/exampleSimpleGates/tb4fa.vhd`
- `...~/schafers/PUBLIC/EXAMPLEs/VHDL/exampleSimpleGates/sg4gate3.vhd`
- `...~/schafers/PUBLIC/EXAMPLEs/VHDL/exampleSimpleGates/rc4fa.vhd`

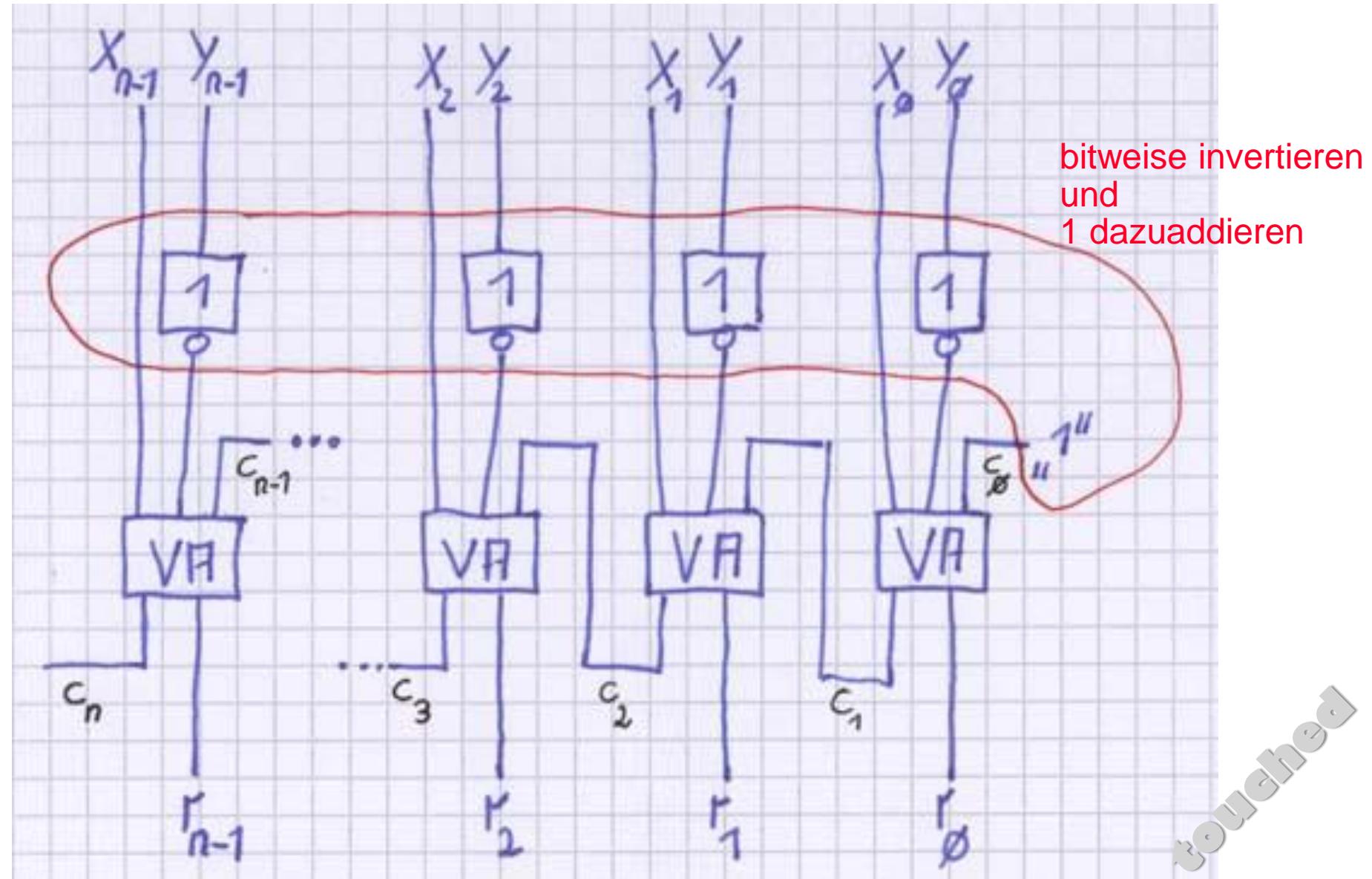
n Bit-Addierer (R = X+Y)



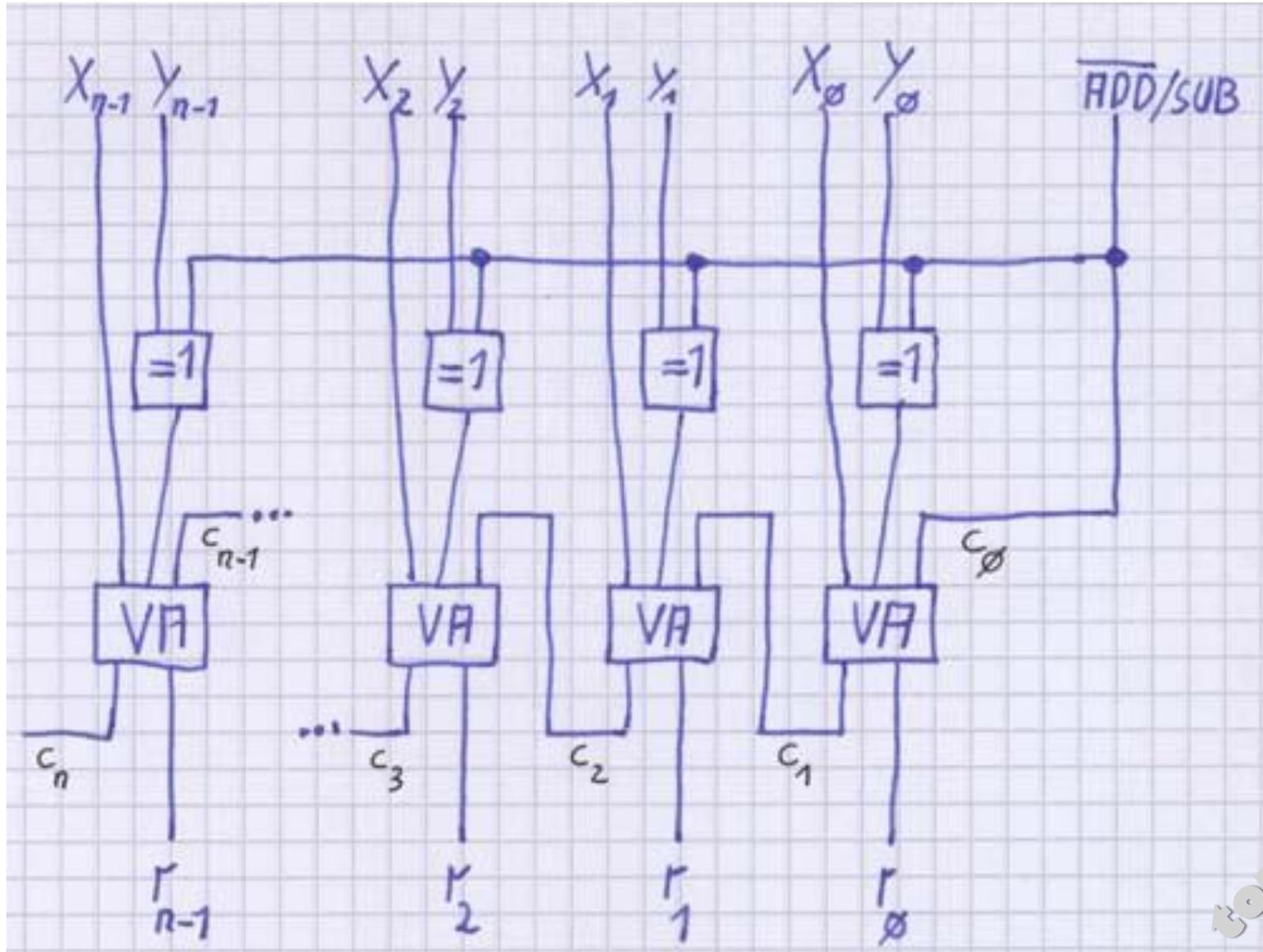
ripple carry adder

touched

n Bit-Subtrahierer ($R = X - Y = X + (-Y)$)



n Bit-Addierer/Subtrahierer



touched

<BREAK>

Was tun wenn andere Frequenz gewünscht? (1)

Grundsätzlich können "andere" Frequenzen mit einer DPLL erzeugt werden

- **DPLL** (Digital Phase Locked Loop) zur Erzeugung der anderen Frequenz.
- DPLL ist ein spezieller Block, der entweder gestellt wird oder von Experten erstellt wird.
- Ein von einer DPLL erstellter Takt zählt nicht als Gated-Clock.
- DPLL ist teuer (in HW-Kosten).
- Jeder zusätzliche Takt ist teuer (in HW-Kosten).
- DPLL ermöglicht "große Freiheitsgrade" bei der Erzeugung von Taktfrequenzen und insbesondere höhere(!) Taktfrequenzen.
- Die entsprechenden Schaltungsteile arbeiten dann mit dem von der DPLL erzeugten Frequenz. Sie bekommen den von der DPLL erzeugten Takt,

Was tun wenn andere Frequenz gewünscht? (2)

Wir wählen aber einfache HW-Kosten-günstige Alternative

Wir nutzen "Ticks" die von einem Tickgenerator erzeugt werden

- **Tickgenerator** zur Erzeugung von Enable-Signalen (=Ticks) mit der gewünschten Frequenz.
- Tickgeneratoren können die Frequenz nur Absenken und (ohne exotische Tricks) nur um ganzzahlige Vielfache – letztlich werden Takte gezählt.
Konsequenz: Die FFs selbst arbeiten weiterhin mit Original-Takt \Rightarrow erhöhter Stromverbrauch gegenüber DPLL-Variante.
- Die entsprechenden Schaltungsteile arbeiten weiterhin mit dem "normalen Takt". Ein enable (=Tick) stößt neue Arbeitsvorgänge an bzw. konkret bestimmt ein Tick wann ein Ergebnis übernommen werden soll.

Was ist ein Tick ?

- Ein Tick ist ein zyklischer Enable-Pulse, der einen Takt lang EINS und sonst NULL ist.
 - Der Begriff kommt aus der Software und ist angelehnt an das Tick einer Uhr.
 - Ein Tick findet oft dort Anwendung, wo die Takt-Frequenz zu hoch ist und "sauber" heruntergeteilt werden kann.
 - Ein Tick wird generiert indem Takte oder andere Ticks gezählt werden.
-
- Ein Tick "geht" typischerweise auf einen Enable-Eingang einer nachfolgenden Schaltung.

Tickgeneratoren hintereinanderschalten

- Tickgeneratoren kosten HW
- Wenn der Takt unterschiedlich heruntergeteilt werden soll ist zu prüfen, ob Tickgeneratoren hintereinander geschaltet werden können um HW-Kosten zu sparen
- Achtung! Ticks müssen in Phase sein
- Um einen Phasenversatz zu vermeiden, werden 2 Ticks erzeugt
 - einen internen rawTick, der im "Mealy-Style" erzeugt wird und
 - einen Tick der abgetaktet ist für die Abnehmer
- Der erste Tick ist nicht abgetaktet und der zweite Tick ist abgetaktet.
- So sind alle Ticks für die "Abnehmer" in derselben "Zeitzone" und die *enableten* Einheiten arbeiten synchron (also ohne Phasenversatz)

(Vorschlag für) Generischer Tick-Generator (1)

```
library work;
use work.all;
use work.support_pkg.all;
...

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity gTickGen is
generic (
    numberOfCycles : positive := 25_000_000); -- 2nd priority defaults for generics
port (
    TICKo : out std_logic; -- TICK (with reduced frequency) Outgoing clocked
    rawTICKo : out std_logic; -- RAW TICK (with reduced frequency) Outgoing unclocked
    ticki : in std_logic; -- TICK/enable Incoming
    clk : in std_logic; -- CLock
    nSReset : in std_logic; -- low active Synchronous RESET
); -- port
constant nob : positive := numberofBits(numberOfCycles); -- Number of Bits required for coding noc unsigned
constant max : positive := numberOfCycles-1; -- Maximum value of counter
constant msbPos : natural := nob-1; -- Most significant bit Position
end entity gTickGen;
```

Bemerkung:

Code (bzw. Umstellung auf numeric_std) konnte leider nicht getestet werden,
da keine VPN-Verbindung zustande kam.

(Vorschlag für) Generischer Tick-Generator (2)

```
architecture rtl.of.gTickGen is
begin
  cobilo:
  process(
    ticki, -- incoming tick respectively VDD / "1"
    cnt_cs -- COUNT of cycles passed
  ) is
    variable cnt_v : std_logic_vector(msbPos downto 0); -- counter for TICK is resp. incoming ticks
    variable tick_v : std_logic; -- (outgoing) tick
  begin
    do computation
    if '1' = ticki then
      incoming_tick -> hence, action
      if unsigned(cnt_cs) >= max then
        tick_v := '1'; -- it's time to send out new tick
        cnt_v := (others => '0'); -- (re)start counting again
      else
        tick_v := '0'; -- NO tick
        cnt_v := std_logic_vector(unsigned(cnt_cs) + 1); -- increment counter
      end if;
    else
      -- NOT enabled -> hence, do nothing
      tick_v := '0'; -- NO tick
      cnt_v := cnt_cs; -- keep value
    end if;
    -- assign computed result(s) to outgoing signal(s)
    tick_ns <= tick_v;
    cnt_ns <= cnt_v;
  end process cobilo;
end architecture rtl.of.gTickGen;
```

Bemerkung:

Code (bzw. Umstellung auf numeric_std) konnte leider nicht getestet werden,
da keine VPN-Verbindung zustande kam.

(Vorschlag für) Generischer Tick-Generator (3)

```
.... sequlo:  
.... process (.clk.) is  
.... begin  
.... if '1' = clk and clk'event then  
.... if '0' = nReset then  
.... cnt_cs <= (others => '0'); ---- nothing counted yet  
.... tick_cs <= '0'; ----- NO tick  
.... else  
.... cnt_cs <= cnt_ns; ----- computed new value for counter  
.... tick_cs <= tick_ns; ----- clocked tick  
.... end if;  
.... end if;  
.... end process sequlo;  
....  
.... -- drive output ports resp. outgoing signals  
.... TICKo <= tick_cs; ----- "tick_cs" comes 1 clock cycle after "ticki"  
.... rawTICKo <= tick_ns; ----- "tick_ns" comes in same clock cycle as "ticki", but is NOT(!) clocked  
....  
end architecture rtl;
```

Bemerkung:

Code (bzw. Umstellung auf numeric_std) konnte leider nicht getestet werden,
da keine VPN-Verbindung zustande kam.

Ein vollständiges und mit ModelSim getestetes Beispiel liegt unter

https://users.informatik.haw-hamburg.de/~schafers/LOCAL/S20W_ECE/CODE/tickGeneratorDemo/

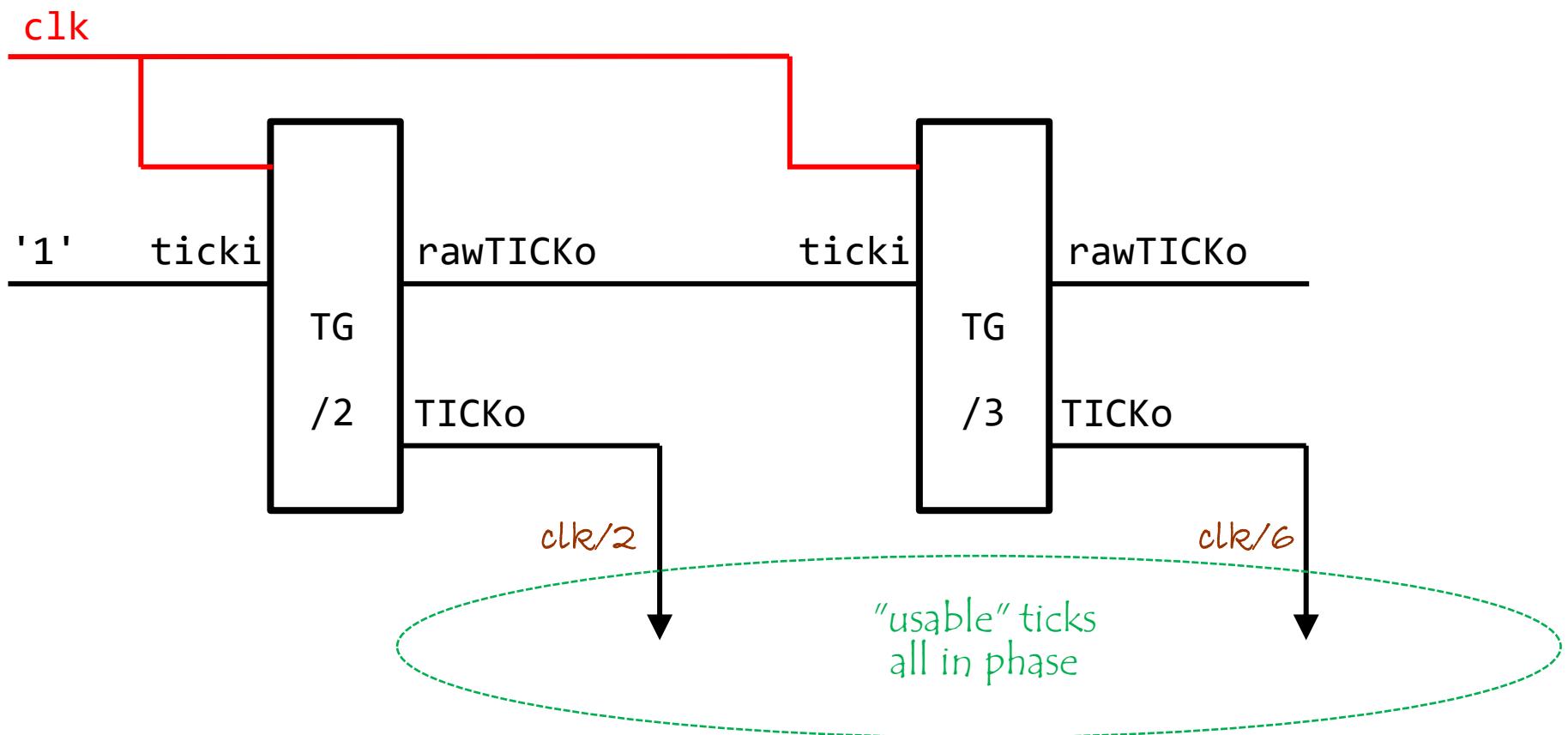
Anwendung eines einzelnen **Tick-Generator**s

```
...
signal VDD  : std_logic      := '1';      -- AUSNAHME! STATISCHES signal
signal GND  : std_logic      := '0';      -- AUSNAHME! STATISCHES signal
...
begin
  ...
  VDD <= '1';
  GND <= '0';

  tg_i : gTickGen
  generic map (
    noc => reasonableValue           -- Number Of (ticki) Cycles
  )
  port map (
    ticko      => theTick_s,          -- TICK (with reduced frequency) Outgoing
    rawTicko   => open,              -- not needed -> hence, unconnected
    ticki      => VDD,                -- always enable
    clk        => clk25MHz,          -- 25MHz clock
    sReset     => sReset            -- Synchronous RESET
  );--]gTickGen
  ...

```

Mehrere Ticks in Phase



(Vorschlag für) "support_pkg" - Package

```
library ...  
  
package support_pkg is  
  
    function numberOfBits( constant value : in positive ) return positive;  
  
end package support_pkg;  
  
  
package body support_pkg is  
  
    function numberOfBits( constant value : in positive ) return positive is  
        variable bitCnt_v : natural;          -- BIT CouNT  
        variable cRest_v  : natural;          -- Current REST  
  
    begin  
        cRest_v := value;  
        bitCnt_v := 0;  
        while cRest_v > 0  loop  
            cRest_v := cRest_v / 2;  
            bitCnt_v := bitCnt_v + 1;  
        end loop;  
        return bitCnt_v;  
    end function numberOfBits;  
  
end package body support_pkg;
```

nur "aufwendig" synthetiserbar bzw. im wesentlichen
nur für statische Berechnungen von Generics geeignet