Grundlagen der Digitaltechnik Foliensatz 2: Simulation

G. Kemnitz

11. Mai 2022

Contents

Simulation

Die Simulation einer digitalen Schaltung bestimmt die Zeitverläufe der internen Signale und Ausgabesignale für vorgegebene Eingabesignalverläufe.

Simulationsbeschreibungen dienen auch zur Synthese, d.h. zur automatischen Berechnung einer Schaltung mit der Funktion des Simulationsmodells (siehe später Foliensatz F3).

Simulationssprache wird VHDL sein.

Der Simulator für die Übungsaufgaben ist »ghdl«, freie Software unter Windows und Linux.

1 Einführung VHDL

1.1 Hallo Welt

Die Hardware-Beschreibungssprache VHDL

Das Akronym VHDL:

V VHSIC (**V**ery **H**igh **S**peed **I**ntegrated **C**ircuits)

H **H**ardware

D **D**escription

L **L**anguage

- In Europa verbreitetste Hardwarebeschreibungssprache,
- erweiterte imperative Sprache (ADA).
- Andere Hardwarebeschreibungssprachen: Verilog, System-C

- Bestandteile eines VHDL-Projekts:
  - Entwurfseinheiten,
  - Packages,
  - Bibliotheken.

Beschreibungsstruktur einer Entwurfseinheit

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  -- Schnittstellenbeschreibung
4  entity Schaltungsname is
5  [generic (<Liste_der_Konfigurationsparameter>);]
6  [port (<Liste_der_Anschlusssignale>);]
7  end entity;
8  -- Beschreibung der Schaltung
9  architecture Beschreibung of Schaltungsname is
10  <Vereinbarungen>
11 begin
12  <Anweisungen>
13 end architecture;
```

Zeile 1, 2: Library- und Use-Anweisung; Einbinden von im Package definierten Bibliotheksobjekten.

Zeile 4-7: Schnittstellenbeschreibung; Vereinbarung der Konfigurationsparameter und Anschlusssignale.

Zeile 9-13: Beschreibung der Entwurfseinheit.

Zeile 10: Vereinbarungen von Datentypen, Signalen, Unterprogrammen etc.

Zeile 12: Anweisungen zur Beschreibung der internen Struktur und/oder Funktion.

Beschreibungsmittel für die interne Struktur und Funktion:

- Einbindung von Teilschaltungen (⇒ Strukturbeschreibung).
- Prozesse: Rahmen zur Verhaltensbeschreibung mit imperativen Anweisungen (⇒ Funktionsbeschreibung).
- Nebenläufige Signalzuweisung (Kurzschreibweisen für Prozesse mit nur einer Signalzuweisung).

Im Weiteren verwendete Schreibweisen/Farben

key	Schlüsselwort
[...]	optionales Element, darf Null mal oder einmal enthalten sein.
{...}	optionales Element, darf beliebig oft enthalten sein
...|...	Alternative, eines der aufgezählten Elemente muss enthalten sein
<Symbol>	Metasymbol (Nichtterminalsymbol): Symbol das nach weiteren Regeln zu ersetzen ist.
Name, ieee	eigener Bezeichner, standardisierter Bezeichner
1, "ab", ...	Wertangaben (Zahlen, Zeichenketten, ...)
--Text	Kommentar

Bezeichner sind explizit zu definieren,

- keine Mehrfachdefinition,
- keine Schlüsselworte.

In VHDL

- beginnen Bezeichner mit einem Buchstaben,
- bestehen nur aus den Buchstaben 'A' bis 'Z', 'a' bis 'z', den Ziffern '0' bis '9' und dem Unterstrich '_',
- dürfen nicht mit einem Unterstrich enden und keine zwei Unterstriche hintereinander enthalten.
- Keine Unterscheidung zwischen Groß- und Kleinschreibung.

Simulierbare Entwurfseinheit (Testrahmen)

- keine Anschlusssignale,
- imperative Teilbeschreibungen in Prozesse gekapselt.

```
1  entity hallo_welt[1] is
2  end entity;
3  architecture a of hallo_welt is
4  begin
5   process
6   begin
7    -- Meldung ausgeben und Prozess beenden
8    report "Hallo Welt";
9    wait;
10  end process;
11 end architecture;
```

- Simulationsausgabe:

  hallo_welt.vhd:8:3:@0ms:(report note): Hallo Welt

[1] Beispielprogramm hallo_welt.vhd.

Beschreibungsteile, die keine Entwurfseinheiten sind, werden vorzugsweise in Packages beschrieben:

- Typenvereinbarungen, Konstanten,
- Unterprogramme, ...

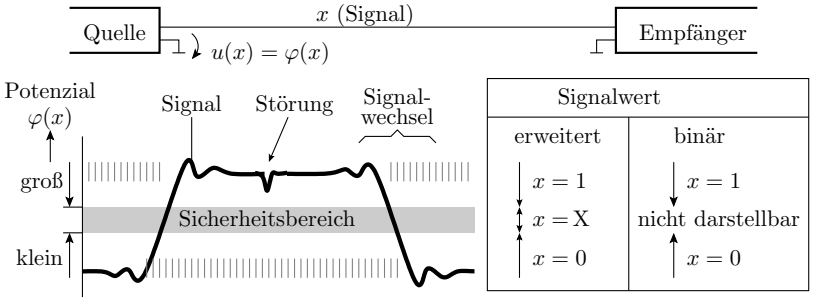
```
-- Definitionsteil, exportierte Vereinbarungen
package hallo_welt_pack[2] is
 constant c: string:="Hallo Zeichenkette";
 function SchreibeText(s: string) return string;
end package;
-- Beschreibung der Unterprogramme
package body hallo_welt_pack is
 function SchreibeText(s: string) return string is
 begin
  return s;
 end function;
end package body;
```

Benutzung von Packages

```
use work.hallo_welt_pack.all;
entity hallo_welt1[3] is
end entity;
architecture a of hallo_welt1 is
begin
 process
 begin
  report "Hallo Welt";
  wait for 1 ns;
  report c;
  wait for 1 ns;
  report SchreibeText("Hallo Funktionsaufruf");
  wait;
 end process;
end architecture;
```

1.2 Signale, Datentypen

Binäre Signale



Ein Signal ist ein zeitlicher Werteverlauf einer physikalischen Größe. Binäre Signale unterscheiden zur Informationsdarstellung zwei Werte: klein und groß, 0 und 1 oder falsch und wahr. Die physikalischen Trägergrößen Strom und Spannung können sich nicht sprunghaft ändern. In den Zeitfenstern von Werteänderungen sind binäre Signale ungültig.

[2] Beispielprogramm hallo_welt_pack.vhd.
[3] Beispielprogramm hallo_welt1.vhd.

VHDL-Typen für binäre Daten

In VHDL sind die Typen für binäre Daten Aufzählungstypen:

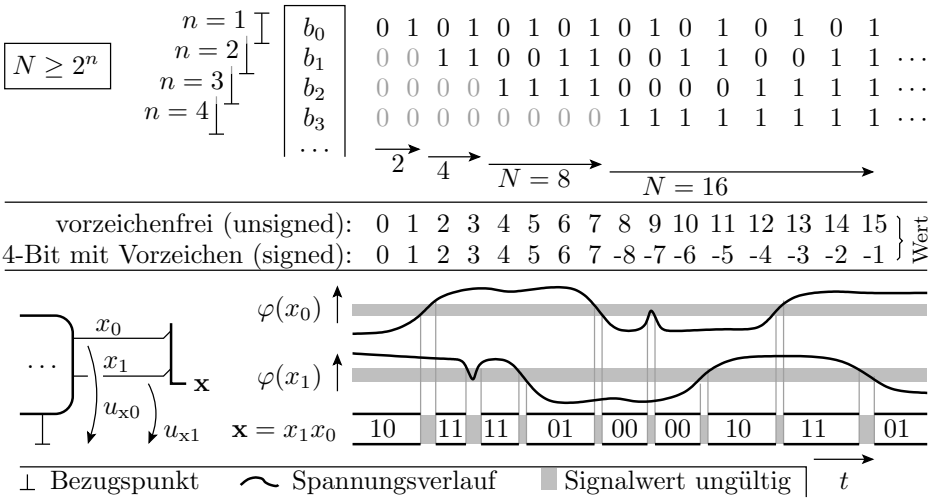
```
-- Bittypen aus std.standard
type bit is ('0', '1');
type boolean is (false, true);
-- Bittyp aus ieee.std_logic_1164
type std_logic is ('U', 'X', '0', '1', ...);
```

- '0', '1' – druckbare Zeichen;
- `false`, `true`– symbolische Konstanten).

Vereinbarung einer Variablen:

```
variable b1, b2: boolean := true;
...
b1 := b1 and b2;
```

Bitvektoren



$N > 2$-wertige Informationen werden als Bitvektoren dargestellt. Ein Bitvektor ist nur gültig, wenn alle Bits gültig sind.

Bitvektortypen und Zahlentypen

Bitvektoren sind eindimensionale Bitfelder:

```
type bit_vector is array (natural range <>) of bit;
type std_logic_vector is array(natural range<>)
     of std_logic;
type unsigned is array(natural range<>)
     of std_logic;
type signed is array(natural range<>) of std_logic;
```

- `unsigned` und `signed` sind gemeinsam mit den arithmetischen Operatoren im Package `ieee.numeric_std` definiert.
- Bitvektorkonstanten sind Zeichenketten, z.B.:

```
"0101" -- Wert auch fuer bit_vector
"0X1U" -- Wert nicht fuer bit_vector
```

Zahlentypen

Indexbereiche sind Zahlen- oder Aufzählungstypen.

- Vordefinierte Zahlentypen:

```
type integer is range -2**31 to 2**31-1;
subtype natural is integer range 0 to integer'high;
subtype positive is integer range 1 to integer'high;
```

- 'high und 'low sind Attribute zur Abfrage der größten und kleinsten Werte eines Zahlentyps.
- Der Ausdruck (natural'range <>) legt einen bei der Instanzbildung zu begrenzenden Indexbereich fest:

Wertebereichsbeschränkung bei der Instanzbildung:

```
signal slv: std_logic_vector(3 downto 0);
variable idx: natural range 0 to 3;
...
slv(idx+1) <= slv(idx);
```

Ohne Textverarbeitung geht es nicht

```
type character is (      -- 128 Zeichen des ASCII-Zeichensatzes
nul, soh, stx, etx, eot, enq, ack, bel, bs, tab, lf,  vt,  ff,  cr,  so,  si,
dle, dc1, dc2, dc3  dc4 nak, syn, etb, can, em, sub, esc, fsp, gsp, rsp, usp,
' ', '!', '"', '#', '$', '%', '&', ''', '(', ')', '*', '+', ',', '-', '.', '/',
'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', ':', ';', '<', '=', '>', '?',
'@', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', '[', '\', ']', '^', '_',
'`', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', '{', '|', '}', '~', del,
  ...);               -- 128 weitere Zeichen (ISO 8859 Latin-1)
```

```
type string is array(positive range <>) of character;
```

Wegen (positive'range <>) beginnt der Indexbereich von Zeichenketten mit eins, statt wie bei Bitvektoren mit null. Zeichen, die der Latin-1 Zeichensatz nicht enthält, sind selbst in Kommentaren unzulässig.

Die Zeit hat einen eigenen Typ

```
type time is range Minimalwert to Maximalwert
 units fs;
  ps = 1000 fs;   ns = 1000 ps;   us = 1000 ns;
  ms = 1000 us;  sec = 1000 ms;  min =   60 sec;
  hr =   60 min;
end units;
```

Die davon abgeleitete Verzögerungszeit ist ein Untertyp, der auf positive Werte beschränkt ist:

```
subtype delay_length is time range 0 fs to time'high;
```

```
signal x, y: std_logic_vector(3 downto 0);
constant td: delay_length := 2 ns;
...
y <= x after td;
```

1.3 Imperative Modelle

Testrahmen für imperative Beschreibungen

Die Funktion der Teilschaltungen wird imperativ beschrieben, als normales Programm für einen normalen Rechner und ist wie ein normales Programm testbar. Beschreibungsaufbau:

- Vorspann:

```
1 -- benoetigte Libraries und Packages
2 library ieee;
3 use ieee.std_logic_1164.all; -- WB: 'U', 'X', '0'
4 use ieee.numeric_std.all;    -- BV für Zahlen
```

- Entity ohne Anschlusssignale:

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

- Architecture mit dem Prozess mit dem eigentlichen Programm:

```
1  architecture a of test is
2  -- Vereinbarungen insbesondere von Signalen
3   signal s: std_logic;
4  begin
5   process
6    -- Vereinbarung insbesondere von Variablen
7    variable v: integer;
8   begin
9    -- zu testende imperative Anweisungsfolge
10   s <= '1'; v:=0;
11   report("s=" & std_logic'image(s) &
12                 "_v=" & integer'image(v));
13   wait; -- beendet Simulation
14  end process;
15 end architecture;
```

- Was geben die Anweisungen in Zeile 11 und 12 aus?

Beschreibungsmittel speziell für den Test

Kleine imperative Beschreibungen werden im einfachsten Fall mit Konstanten als Eingabe und Ausgaben auf der Konsole getestet. Minimalmenge an Beschreibungsmitteln:

- Feldkonstanten für die Testeingaben und Wiederholschleifen für die Testabarbeitung (siehe Hausübung).
- report(<*Ausgabetext*>): Ausgabe einer Textzeile.
- <*Text*> & <*Text*>: Verketten von Texten.
- <*Datentyp*>'**image**(...): Konvertierung in eine Textdarstellung, nur für skalare Typen (Zahlen-, Aufzählungs- und physikalische Typen).

Für zusammengesetzte Typen (Felder und Strukturen) und für abgeänderte Wertedarstellungen wird hier meist die str()-Funktion überladen.

Definition und Nutzung einer str()-Funktion

```
 -- Textkonvertierung der 3 Variablenwerte
 function str(a: std_logic, b: std_logic, c: std_logic) return string is
 begin
  return "␣a=" & std_logic'image(a) &
         "␣b=" & std_logic'image(b) &
         "␣c=" & std_logic'image(c);
 end function;
begin -- hier folgen die zu testenden Anweisungen
 a:='1'; b:='X'; c := a and b;  -- Ausgabetext
 report(str(a, b, c));   -- a='1' b='X' c='X'
 a := '1'; b:='0'; c := a and b;
 report(str(a, b, c));   -- a='1' b='0' c='0'
 wait;
end process;
end architecture;
```

Durchführung der Simulation

- Im Editor Programm eingeben und vervollständigen[4]:

```
library ieee;
use ieee.std_logic_1164;

entity test_anw is end entity;

architecture a of test_anw is
 begin
 ... -- Anweisungen der Folie zuvor
 end process;
end architecture:
```

- In ein Arbeitsverzeichnis als test_anw.vhd speichern.
- Terminal öffnen und in das Arbeitsverzeichnis wechseln.

- Package und Testrahmen analysieren (analyze):

```
...$ ghdl -a test_anw.vhd
```

- Zu einem ausführbaren Programm verbinden (make):

```
...$ ghdl -m test_anw
```

- Simulation ausführen (run) und Programmausgaben:

```
...$ ghdl -r test_anw
```

- Terminalausgabe:

```
a='1' b='X' c='X'
a='1' b='0' c='0'
```

Auf den weiteren Folien stehen oft nur die Vereinbarungen und Anweisungen der Beispielprogramme. Komplettierung und Test jeweils wie hier beschrieben.

[4] Oder von http://techwww.in.tu-clausthal.de

Definition von Datenobjekten und Anfangswerten

```
constant <bez>{,<bez>}: Typ [:=<aw>];
variable <bez>{,<bez>}: Typ [:=<aw>];
signal   <bez>{,<bez>}: Typ [:=<aw>];
```

(bez – Bezeichner; aw – Anfangswert). Ohne Angabe ist der Anfangswert der erste Wert der Typdefinition:

bit	std_logic	boolean	integer	natural	positive
'0'	'U'	false	integer'low	0	1

```
signal i: integer; signal b: std_logic;
signal p: positive;
...
report("i=" & integer'image(i) &  -- Was wird
       "␣b=" & std_logic'image(b)& -- ausge-
       "␣p=" & positive'image(p)); -- geben?
```

- Der Indexbereich von Bitvektoren kann absteigend oder aufsteigend vereinbart werden.
- Elementezuordnung bei Wertezuweisungen von links nach rechts:

```
signal s: std_logic_vector(0 to 3) := "1100";
signal f: std_logic_vector(3 downto 0)
                                   := "1100";
```

Vektorelement	s(0)	s(1)	s(2)	s(3)	f(3)	f(2)	f(1)	f(0)
Anfangswert	1	1	0	0	1	1	0	0

- Ohne Anfangswertangabe Elementeinitialisierung mit Standardwerten.

```
signal slv: std_logic_vector(3 downto 0);
```

- Welchen Anfangswert hat die Variable slv?

Anweisungsfolge zum Ausprobieren

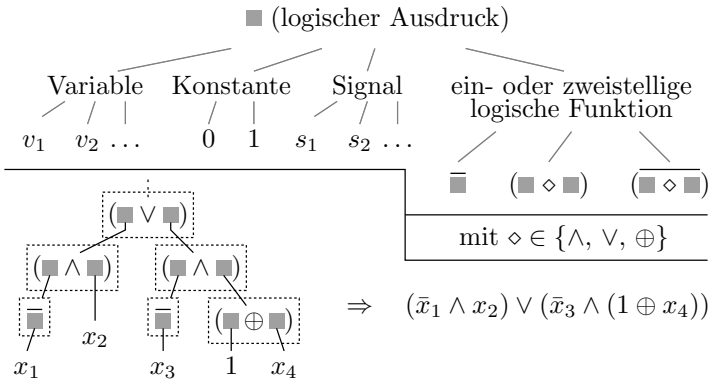
```
1 process
2  variable slv: std_logic_vector(3 downto 0):="0X10";
3 begin
4  for idx in slv'range loop
5   report("slv(" & integer'image(idx) & ")="
                  & std_logic'image(slv(idx)));
6  end loop;
7  wait;
8 end process;
```

---

1. Welche Anweisungen sind vor Zeile 1 zu ergänzen?
2. Was wird in Zeile 5 ausgegeben?
3. Wozu dient die Wait-Anweisung in Zeile 7?
4. Welche Anweisungen fehlen nach Zeile 8?

Logische Operatoren und Ausdrücke

Ein (logischer) Ausdruck ist eine Textbeschreibung für einen baumartigen Berechnungsfluss:



$x_2\,x_1$	Inverter $\bar{x}_1$	UND $x_1 \wedge x_2$	NAND $\overline{x_1 \wedge x_2}$	ODER $x_1 \vee x_2$	NOR $\overline{x_1 \vee x_2}$	XOR $x_1 \oplus x_2$	XNOR $\overline{x_1 \oplus x_2}$
0 0	1	0	1	0	1	0	1
0 1	0	0	1	1	0	1	0
1 0		0	1	1	0	1	0
1 1		1	0	1	0	0	1
VHDL-Schlüsselwort	not	and	nand	or	nor	xor	xnor

Logische Operatoren in VHDL: **not**, **and**, **nand**, **or**, **nor**, **xor** und **xnor**.

- »not« hat Vorrang.
- Alle 2-stelligen VHDL-Operatoren sind gleichberechtigt.
- Ausführungsreihenfolge mit Klammern festlegen.

Welchen Typ und welchen Wert haben folgende Ausdrücke:

1. (`true` **and** `false`) **or** `true`?
2. `true` **and** (`false` **or** `true`)?

Verarbeitung ungültiger Bitwerte

```
type std_logic is ('U', 'X', '0', '1', ...)
```

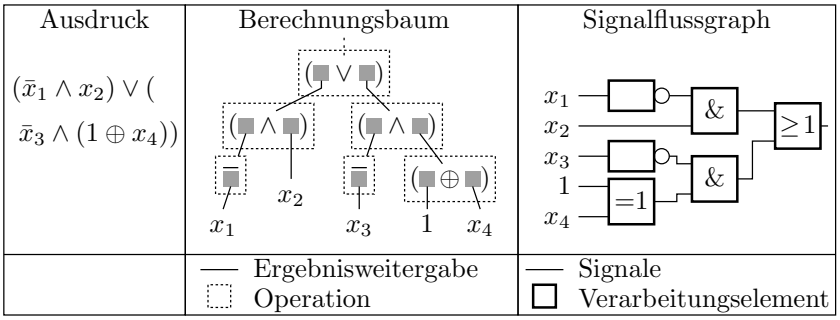
- ungültig (X): Wert kann »0« oder »1« sein
- Simulationsregeln:

$$\overline{\mathrm{X}} = \mathrm{X} \qquad \begin{array}{lll} \mathrm{X} \wedge 0 = 0 & \mathrm{X} \vee 0 = \mathrm{X} & \mathrm{X} \oplus 0 = \mathrm{X} \\ \mathrm{X} \wedge 1 = \mathrm{X} & \mathrm{X} \vee 1 = 1 & \mathrm{X} \oplus 1 = \mathrm{X} \\ \mathrm{X} \wedge \mathrm{X} = \mathrm{X} & \mathrm{X} \vee \mathrm{X} = \mathrm{X} & \mathrm{X} \oplus \mathrm{X} = \mathrm{X} \end{array}$$

Welcher Wert wird y zugewiesen?

```
variable x: std_logic_vector(3 downto 0);
variable y: std_logic;
...
x:="1001"; y:=(x(3) and x(2)) or (x(1) and x(0));
x:="10X1"; y:=(x(3) and x(2)) or (x(1) and x(0));
x:="X011"; y:=(x(3) and x(2)) or (x(1) and x(0));
```

Ausdruck, Berechnungsfluss und Signalfluss



Ein Ausdruck kann nicht nur einen Berechnungs- sondern auch einen Signalfluss beschreiben. Aus den Operatoren werden Gatter und aus den Eingaben, Zwischenergebnissen und dem Ergebnis Signale. Eine Minimierung der Operationen im Ausdruck minimiert die Gatteranzahl der korrespondierenden Schaltung.

Signalzuweisung

Einem Signal werden Wert-Zeit-Tupel zugewiesen:

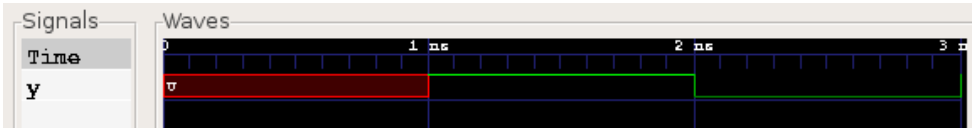
```
<Signalname> <= <Ausdruck> [after <td>]
  {, <Ausdruck> [after <td>]};
```

Ausdruck – Ausdruck zur Werteberechnung; *td* – Ausdruck für die Berechnung der Verzögerungszeit. Beispiel:

```
signal y: std_logic;
...
y <= '1' after 1 ns, '0' after 2 ns, '1' after 3 ns;
```

Berechnete Signalverläufe können graphisch dargestellt werden:

```
ghdl -r test_sig[5] --wave=test_sig.ghw
gtkwave test_sig.ghw [test_sig.sav]
```



Warteanweisungen und Weckbedingungen

Signalverläufe werden in der Regel nicht in einem Schritt zugewiesen, sondern berechnet. Der Simulator hat eine Systemvariable **now** für die simulierte Zeit, die beim Simulationsstart null ist und mit Warteanweisungen erhöht werden kann. Eine Warteanweisung legt den Prozess schlafen, bis die Weckbedingung erfüllt ist. Weckbedingungen:

- Änderung eines Signals aus der Weckliste

  ```
  wait on <Weckliste>;
  ```

- Verstreichen einer Verzögerungszeit

  ```
  wait for <Verzoegerungszeit>;
  ```

- Eintreten einer beliebig beschreibbaren Bedingung:

  ```
  wait until <Bedingung>;
  ```

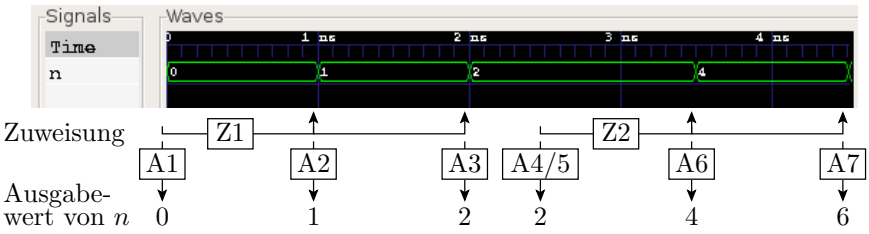
Zuweisungen von Signaländerungen erfolgen zur Simulationszeit **now** + t_d (**now** – aktuelle Simulationszeit; t_d – Verzögerungszeit).

[5] Ausführbare Simulationsdatei mit der Signalzuweisung oben.

Programm mit Warteanweisungen

```
signal n: natural;
...
begin
process[6]
 function str(n: integer) return string is
  begin
   return "␣t=" & time'image(now) &
          "␣n=" & integer'image(n);
  end function;
 begin
 Z1: n <= n+1 after 1 ns, n+2 after 2 ns;
                  report("A1:" & str(n));
  wait on n;      report("A2:" & str(n));
  wait on n;      report("A3:" & str(n));
  wait for 0.5 ns; report("A4:" & str(n));
 Z2: n <= n+2 after 1 ns, n+4 after 2 ns;
```

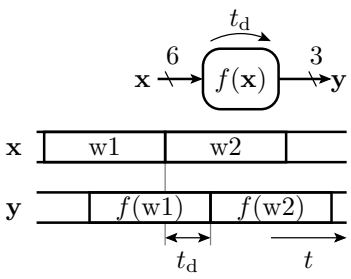
```
Z1: n <= n+1 after 1 ns, n+2 after 2 ns;
                  report("A1:" & str(n));
 wait on n;       report("A2:" & str(n));
 wait on n;       report("A3:" & str(n));
 wait for 0.5 ns; report("A4:" & str(n));
Z2: n <= n+2 after 1 ns, n+4 after 2 ns;
                  report("A5:" & str(n));
 wait on n;       report("A6:" & str(n));
 wait on n;       report("A7:" & str(n));
```



1.4 Ereignisgesteuerte Simul.

Ereignisgesteuerte Simulation

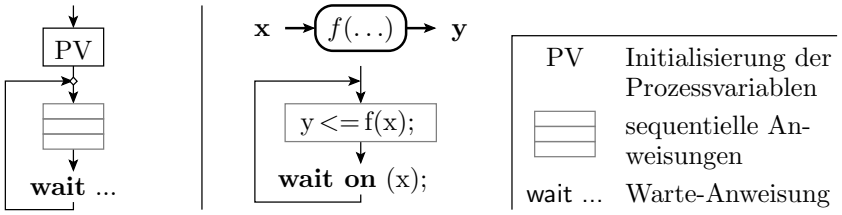
```
signal x: std_logic_vector(5 downto 0);
signal y: std_logic_vector(2 downto 0);
constant td: delay_length := 1 ns;
...
process
begin
 y <= f(x) after td;
 wait on x;
end process;
```



- Jede Teilschaltung überwacht ständig ihre Eingabesignale.
- Bei einer Änderung werden die Werte der Ausgabesignale neu berechnet und verzögert zugewiesen.
- Ausgabeänderungen lösen Neuberechnungen in den nachfolgenden Teilschaltungen aus.

[6] Aus dem Beispielprogramm test_wait.vhd.

Prozessmodell einer Teilschaltung

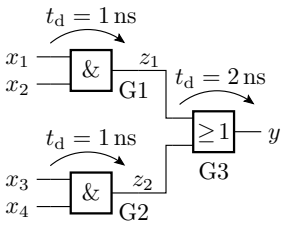


Endlosschleife:

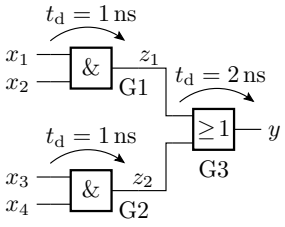
- Berechnung der Ausgabewerte aus den Eingabewerten.
- Verzögerte Zuweisung an das Ausgabesignal.
- Warten auf eine Eingabeänderung.

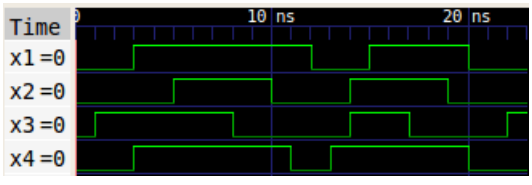
3-Gatter-Schaltung

- ein Prozess je Gatter + Eingabeprozess



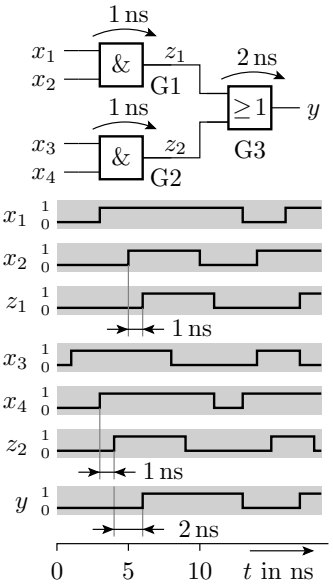

```
entity SimModell is
end entity;
architecture Sim of SimModell is
 signal x1, x2, x3, x4, z1, z2, y: std_logic:='0';
begin

-- Simulation G1
G1: process
begin
 z1<=x1 and x2 after 1 ns;
 wait on x1, x2;
end process;

-- Simulation G2
G2: process
begin
 z2<=x3 and x4 after 1 ns;
 wait on x3, x4;
end process;

-- Simulation G3
G3: process
 begin
  y<=z1 or z2 after 2 ns;
  wait on z1, z2;
 end process;

-- Eingabeprozess
Eingabe: process
 begin
  wait for 1 ns;
  x3 <= '1';
  wait for 2 ns;
  x1 <= '1';
  x4 <= '1';
   ...
```





t_{sim}	Prozess	Signalzuweisung	schwebende Änderungen	Signale $x_4\,x_3\,x_2\,x_1\,z_2\,z_1\,y$
0 ns	G1	$z_1 \Leftarrow 0 \wedge 0$ nach 1 ns	————	0 0 0 0 0 0 0
0 ns	G2	$z_2 \Leftarrow 0 \wedge 0$ nach 1 ns	————	0 0 0 0 0 0 0
0 ns	G3	$y \Leftarrow 0 \vee 0$ nach 2 ns	————	0 0 0 0 0 0 0
0 ns	Eingabe		@1 ns: E	0 0 0 0 0 0 0
1 ns	E	$x_3 \Leftarrow 1$	@1 ns: $x_3 \rightarrow 1$ + @3 ns: E	0 1 0 0 0 0 0
1 ns	G2	$z_2 \Leftarrow 1 \wedge 0$ nach 1 ns	@3 ns: E	0 1 0 0 0 0 0
3 ns	E	$x_1 \Leftarrow 1$, $x_4 \Leftarrow 1$	@3 ns: $x_1 \rightarrow 1$, @3 ns: $x_4 \rightarrow 1$, @...:E	0 1 0 1 0 0 0 1 1 0 1 0 0 0
3 ns	G1	$z_1 \Leftarrow 1 \wedge 0$ nach 1 ns	@3 ns: $x_4 \rightarrow 1$,..	1 1 0 1 0 0 0
3 ns	G2	$z_2 \Leftarrow 1 \wedge 1$ nach 1 ns	@4 ns: $z_2 \rightarrow 1$,..	1 1 0 1 0 0 0
4 ns	G3	$y \Leftarrow 0 \vee 1$ nach 2 ns	@6 ns: $y \rightarrow 1$,..	1 1 0 1 1 0 0
5 ns	...	...	...	1 1 1 1 1 0 0
6 ns	...	...	...	1 1 1 1 1 0 1

Initialisierung — keine schwebende Änderung E Eingabeprozess



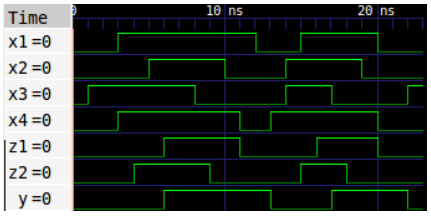
Beobachterprozess für die Signaländerungen

```
Beob: process
 variable tmp: string(1 to 3);
 variable s: string(1 to 15) := "x=.... z=.. y=.";
begin
 tmp := std_logic'image(x1); s( 6) := tmp(2);
 tmp := std_logic'image(x2); s( 5) := tmp(2);
 tmp := std_logic'image(x3); s( 4) := tmp(2);
 tmp := std_logic'image(x4); s( 3) := tmp(2);
 tmp := std_logic'image(z1); s(11) := tmp(2);
 tmp := std_logic'image(z2); s(10) := tmp(2);
 tmp := std_logic'image( y); s(15) := tmp(2);
 report(s);
 -- warte auf eine Signalaenderung
 wait on x1, x2, x3, x4, z1, z2, y;
end process;
```

- Was für einen Text gibt die Report-Anweisung aus?

Simulationsergebnis

```
@0ms: x=0000 z=00 y=0
@1ns: x=0100 z=00 y=0
@3ns: x=1101 z=00 y=0
@4ns: x=1101 z=10 y=0
@5ns: x=1111 z=10 y=0
@6ns: x=1111 z=11 y=1
@8ns: x=1011 z=11 y=1
@9ns: x=1011 z=01 y=1
@10ns: x=1001 z=01 y=1
@11ns: x=1001 z=00 y=1
@11ns: x=0001 z=00 y=1
@12ns: x=0000 z=00 y=1
@13ns: x=0000 z=00 y=0
@13ns: x=1000 z=00 y=0
@14ns: x=1110 z=00 y=0
@15ns: x=1110 z=10 y=0
```




```
@15ns: x=1111 z=10 y=0
@16ns: x=1111 z=11 y=0
@17ns: x=1111 z=11 y=1
@17ns: x=1011 z=11 y=1
@18ns: x=1011 z=10 y=1
@19ns: x=1001 z=01 y=1
@20ns: x=1001 z=00 y=1
@20ns: x=0000 z=00 y=1
@22ns: x=0000 z=00 y=0
@22ns: x=0100 z=00 y=0
```

(Beispielprogramm sim_3gb.vhd)

Prozesse mit Weckliste

Prozesse zur Beschreibung der Funktion von Schaltungen haben praktisch immer nur eine einzige Warteanweisung auf Signaländerungen, und zwar am Ende der Anweisungsfolge:

```
G1: process
begin
 z1 <= x1 and x2 after 1 ns;
 wait on x1, x2;
end process;
```

Übersichtlicher ist ein Prozess mit Weckliste:

```
G1_PWList: process (x1, x2)
begin
 z1 <= x1 and x2 after 1 ns;
end process;
```

»Prozess mit Weckliste« vermeidet den Fehler »Warteanweisung vergessen«, bei dem sich die Simulation aufhängt. Warum?

Nebenläufige Signalzuweisungen

Prozesse für Verarbeitungsfunktionen werden bei jeder Änderung eines Eingabesignals geweckt (alle Eingabesignale in der Weckliste). Enthält ein solcher Prozess (z.B. die Beschreibung der Funktion eines Gatters) nur eine Signalzuweisung, darf der Prozessrahmen weggelassen werden. Die Beschreibung:

```
G1_PWList: process (x1, x2)
begin
 z1<=x1 and x2 after 1 ns;
end process;
```

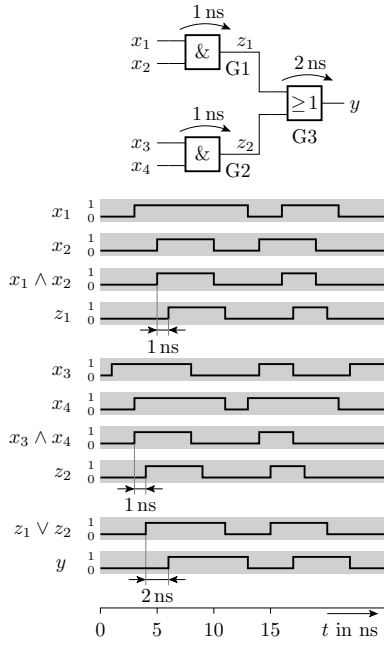
ist identisch mit der nebenläufigen Signalzuweisung:

```
z1 <= x1 and x2 after 1 ns;
```

Diese verkürzte Schreibweise beugt dem Fehler »vergessenes Eingabesignal in der Weckliste« vor.

Beispielschaltung mit neben- läufigen Signalzuweisungen

```
signal x1, x2, x3, x4, z1, z2,
       y: std_logic := '0';
...
G1: z1<=x1 and x2 after 1ns;
G2: z2<=x3 and x4 after 1ns;
G3: y<=z1 or z2 after 2 ns;

Eingabe: process
begin
  wait for 1 ns; x3<='1';
  wait for 2 ns; x1<='1';
                 x4<='1';
  ...
  wait;
end process;
```

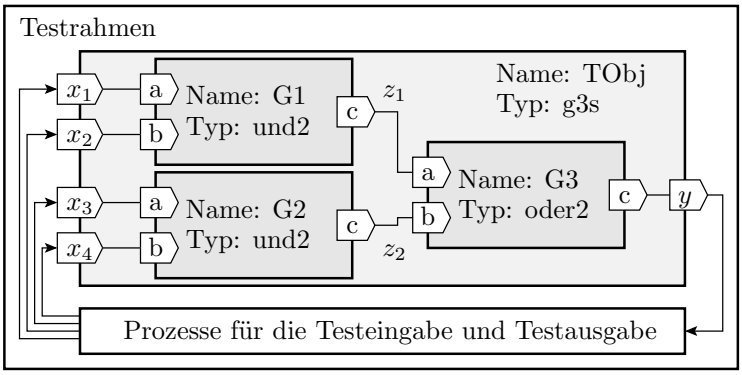


Zusammenfassung

- Die zeitlichen Abläufe werden mit Signalzuweisungen und Warteanweisungen nachgebildet.
- Das Funktionsmodell einer kombinatorischen Schaltung ist ein Prozess, der die Ausgabewerte aus den aktuellen Eingabewerten berechnet, verzögert zuweist, sich bis zur nächsten Eingabeänderung schlafen legt und nach dem Aufwachen dieselbe Berechnungsfolge wiederholt.
- Bei einer genauen Simulation des Zeitverhaltens wird jede Teilschaltung durch einen eigenen Prozess simuliert. Zusätzlich benötigt die Simulation einen Prozess zur Bereitstellung der Eingabesignale.

2 Strukturbeschreibung

Strukturbeschreibung



Digitale Systeme werden hierarchisch aus Teilschaltungen zusammengesetzt. Jede Teilschaltung hat eine Schnittstelle, die die Anschlusssignale definiert. Eine Strukturbeschreibung hat auch eine Schnittstelle, definiert interne Signale und beschreibt für jede Teilschaltung, mit welchen Signalen deren Anschlüsse verbunden sind. Das oberste Hierarchie-Element ist der Testrahmen.

Schnittstellenvereinbarung

```
entity <Entwurfseinheit> is
  [generic (<Liste der Konfigurationsparameter>);]
  [port (<Anschlussliste>)];
end entity;
```

Die Anschlussliste ist eine kommaseparierte Liste von Anschlussvereinbarungen der Form:

```
<Signalname> {,<Signalname>}:<Richtung> <Datentyp>
    [:=<Standardwert>]
```

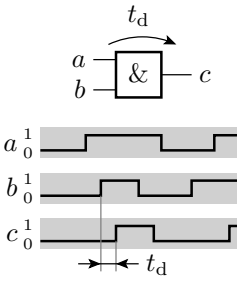
Mögliche Signalflussrichtungen sind

```
in     Eingang
out    Ausgang
inout  Ein- und Ausgang
buffer Ausgang mit rücklesbarem Wert
```

Der Standardwert ist der Anfangswert zum Simulationsbeginn[7].

[7] Ohne Angabe der Standardwert des Datentyps.

Beschreibung eines UND-Gatters




```
-- Schnittstellenvereinbarung
entity und2 is
 generic (td: delay_length := 0 ns);
 port (a, b: in std_logic;
       c: out std_logic);
end entity;

-- Beschreibung
architecture mit_td of und2 is
begin
  c <= a and b after td;
end architecture;
```

Über generic können der Entwurfseinheit bei der Instanziierung Parameter zugeordnet werden, im Beispiel die Verzögerungszeit. Wird in den von unserem Entwurfssystem generierten Post-Place-and-Route-Simulationsmodellen genutzt.

Teilschaltungsinstanzen

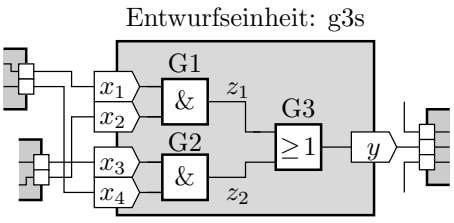
Syntax für die Einbindung einer Entwurfseinheit als Teilschaltung:

```
[<Teilschaltungsname>:] entity [<Bibliothek>.]
     <Entwurfseinheit>[(<Beschreibung>)]
     [generic map(<Zuordnungsliste Parameter>)]
     port map(<Zuordnungsliste Anschluesse>);
```

Zuordnungslisten können in VHDL namens- oder positionsbasiert beschrieben werden:

- namensbasiert: kommaseparierte Liste aus Tupeln *<Objekt zum zuordnen> => <zugeordnetes Objekt>*
- positionsbasiert: kommaseparierte Liste der zugeordneten Objekte.

Strukturbeschreibung der 3-Gatterschaltung




```
-- Schnittstellenvereinbarung
entity g3s is
 port (x1, x2, x3, x4: in std_logic;
       y: out std_logic);
end entity;

architecture Struktur of g3s is
-- Vereinbarung der internen Signale
 signal z1, z2: std_logic;
begin
-- Instanziierung und Verbindung der Gatter
...
```

```
 G1: entity work.und2(mit_td)
  generic map (1 ns)              --positionsb. Zuordn.
  port map (x1, x2, z1);          --positionsb. Zuordn.

 G2: entity work.und2(mit_td)
  generic map (td => 1 ns)         --namensb. Zuordnung
  port map (a=>x3, b=>x4, c=>z2);--namensb. Zuordnung

 G3: entity work.oder2(mit_td)
  generic map (td => 2 ns)         --namensb. Zuordnung
  port map (a=>z1, b=>z2, c=>y); --namensb. Zuordnung
end architecture;
```

(Aus dem Beispiel g3s.vhd)

Instanzbildung mit Platzhaltern

ISE[8] empfiehlt eine ältere Art der Instanziierung:

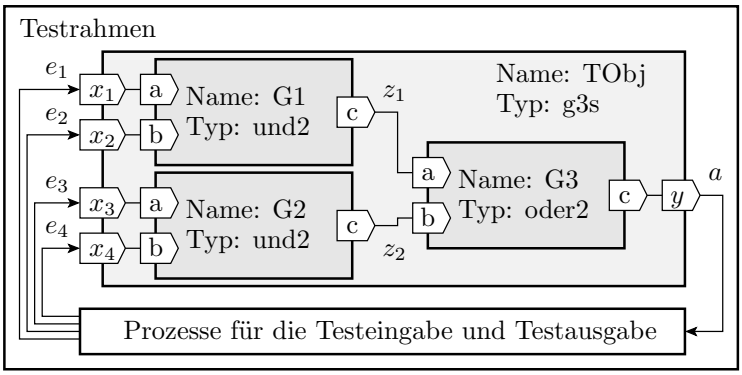
```
[<Teilschaltungsname>:] component <Entwurfseinheit>
    [generic map(<Zuordnungsliste Parameter>)]
    port map(<Zuordnungsliste Anschluesse>);
```

Das erfordert eine zusätzliche Deklaration der Schnittstelle als Komponente (Package oder Deklarationsteil der Entwurfseinheit):

```
component <Entwurfseinheit> is
  [generic (<Liste der Konfigurationsparameter>);]
  [port (<Anschlussliste>)];
end [component][<Entwurfseinheit>];
```

Eine Komponentenbeschreibung entsteht durch Kopieren der Schnittstellenbeschreibung und Ersatz von **entity** durch **component**. Bläht Beschreibung auf. Kein erkennbarer Nutzen.

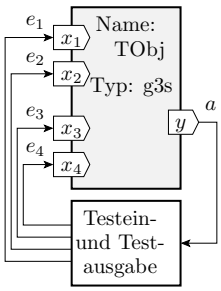
Testrahmen



Das oberste Hierarchieelement einer Simulation ist der Testrahmen (Testbench). Er besteht im Wesentlichen aus einer Instanz des Testobjekts, Vereinbarungen der an das Testobjekt angeschlossenen Signale sowie Prozessen zur Bereitstellung der Testeingaben und Auswertung der Testausgaben.

[8] In der Übung genutztes Entwurfssystem.

```
entity test_g3s is
end entity;

architecture a of test_g3s is
 signal e1,e2,e3,e4,a: std_logic:='0';
begin
 TObj: entity work.g3s(Struktur)
  port map (x1=>e1, x2=>e2, x3=>e3,
            x4=>e4, y=>a);

Eingabe: process
  begin
   wait for 1 ns;  e3<='1';
   wait for 2 ns;  e1<='1'; e4<='1';
   ...
  wait;
 end process;
```



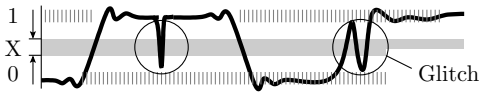
Zur Simulation wird die Hierarchie aufgelöst und dasselbe 4-Prozess-Modell, das ab Folie 13 beschrieben wurde, erzeugt.

Zusammenfassung

- Zur hierarchischen Strukturierung werden die Prozesse zu Entwurfseinheiten zusammengefasst, mit Schnittstellen versehen und in übergeordnete Entwurfseinheiten als Instanzen eingebunden.
- Die oberste Ebene in der Beschreibungshierarchie eines Simulationsmodells ist der Testrahmen, eine Entwurfseinheit ohne Anschlüsse.
- Bei der Schaltungsanalyse vor der Simulation wird die Hierarchie wieder in einzelne über Signale kommunizierende Prozesse – das eigentliche Simulationsmodell – aufgelöst.
- Beispielprogrammdateien:
  - und2.vhd: Verhaltensbeschreibung UND-Gatter
  - oder2.vhd: Verhaltensbeschreibung ODER-Gatter
  - g3s.vhd: Strukturbeschreibung der 3-Gatter-Schaltung
  - test_g3s.vhd: Testrahmen für die 3-Gatter-Schaltung.

3 Laufzeittoleranz

Schwächen des bisherigen Modells



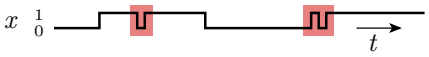
- Signalführende Größen ändern sich stetig und
- sind zwischen den Werteänderungen kurzzeitig ungültig.
- Verzögerungszeiten unterliegen Fertigungsstreuungen,
- Temperaturabhängigkeit, Alterung, Glitches, ...

Definition 1. Laufzeittoleranz: Eine digitale Schaltung ist laufzeittolerant, wenn ihre Funktion nicht von Zeitparametern abhängt, solange diese innerhalb ihrer zulässigen Toleranzbereiche liegen.

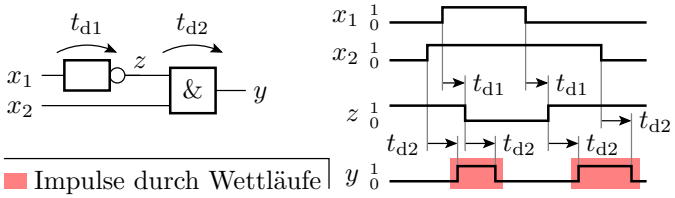
3.1 Glitches

Glitches und Wettläufe

Glitches sind kurze Pulse, die bei logischen Verknüpfungen von Signalen mit unterschiedlicher Verzögerung entstehen:

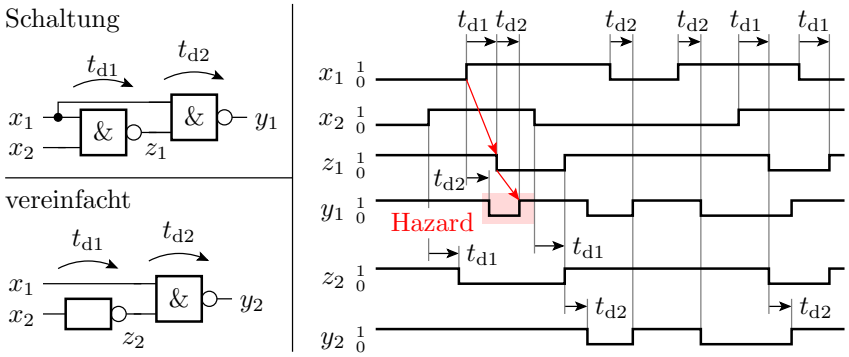


Eine Ursache sind Wettläufe. Das sind fast zeitgleiche Änderungen an mehreren Eingängen einer Teilschaltung. Auch wenn vor und nach der Änderung derselbe Wert ausgegeben wird, können im Änderungsmoment am Ausgang kurze Pulse auftreten.



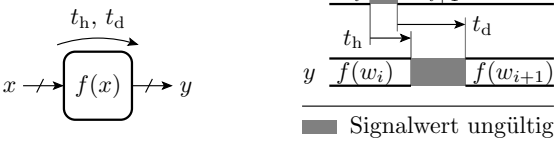
Hazards

Hazards sind Glitches, die durch eine Signaländerung an nur einem Eingang entstehen. Ursache sind rekonvergente[9] Signalflüsse mit unterschiedlichen Verzögerungszeiten.



3.2 Simulation von Zeittoleranzen

Simulation toleranzbehafteter Verzögerungszeiten




```
y <= <ungueltig> after th, f(x) after td;
```

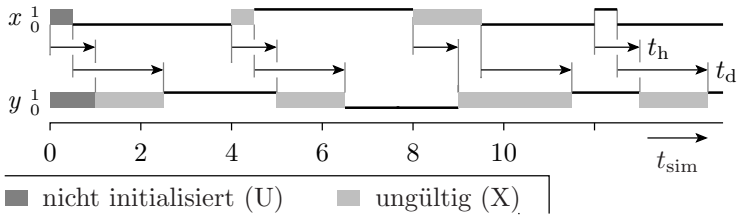
t_h	Haltezeit
t_d	Verzögerungszeit
$f(x)$	logische Verarbeitungsfunktion

Weiter zu verarbeitende Signalwerte sind innerhalb ihrer Gültigkeitsfenster abzutasten.

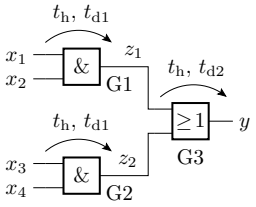
[9] Verzweigende Signalflüsse, die wieder zusammen treffen.

Simulation eines Inverters

```
signal x, y: std_logic;
...
x <= '0' after 0.5 ns, 'X' after 4 ns,
     '1' after 4.5 ns, 'X' after 8 ns,
     '0' after 9.5 ns, '1' after 12 ns,
     '0' after 12.5 ns;
y <= 'X' after 1 ns, not x after 2 ns;
```

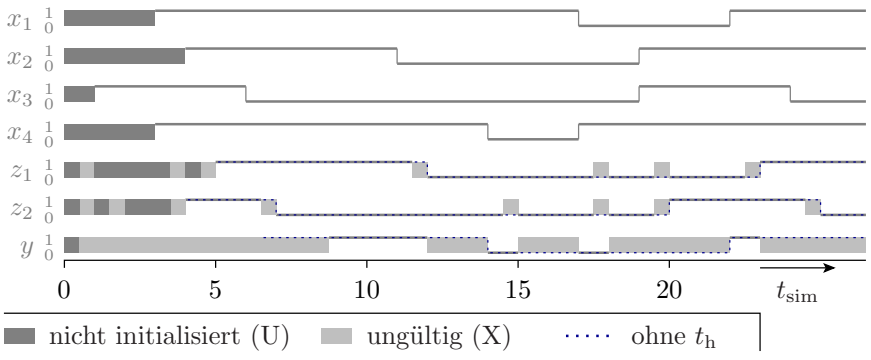


Simulation einer Gatterschaltung




```
signal x1, x2, x3, x4,
    z1, z2, y: std_logic;
constant th: delay_length:=500 ps;
constant td1: delay_length:=1 ns;
constant td2: delay_length:=2 ns;
...
G1:z1<= 'X' after th, x1 and x2 after td1;
G2:z2<= 'X' after th, x3 and x4 after td1;
G3: y<= 'X' after th, z1 or  z2 after td2;
Eingabe: process begin
 wait for 1 ns; x3<='1';
 wait for 2 ns; x1<='1'; x4<='1';
 wait for 1 ns; x2<='1';
 wait for 3 ns; x3<='0'; ...; wait;
end process;
```

```
G1:z1<= 'X' after th, x1 and x2 after td1;
G2:z2<= 'X' after th, x3 and x4 after td1;
G3: y<= 'X' after th, z1 or  z2 after td2;
```



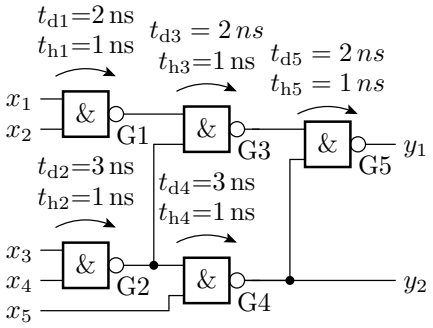
- Ausgaben meiste Zeit ungültig / möglicherweise falsch.
- Schaltungen, die ungültige Signalwerte verarbeiten, sind unzuverlässig.

3.3 Laufzeitanalyse

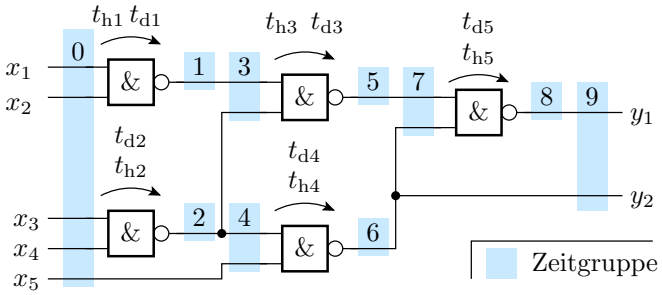
Laufzeitanalyse

Die Laufzeitanalyse berechnet die Halte- und Verzögerungszeit einer übergeordneten kombinatorischen Schaltung aus den Halte- und Verzögerungszeiten der Teilschaltungen durch Addition entlang aller Pfade und macht die Simulation von Zeittoleranzen oft überflüssig.

Pfade	$\sum t_{h.i}$	$\sum t_{d.i}$
G1-G3-G5	3 ns	6 ns
G2-G3-G5	3 ns	7 ns
G2-G4-G5	3 ns	**8 ns**
G2-G4	2 ns	6 ns
G4-G5	2 ns	5 ns
G4	**1 ns**	3 ns



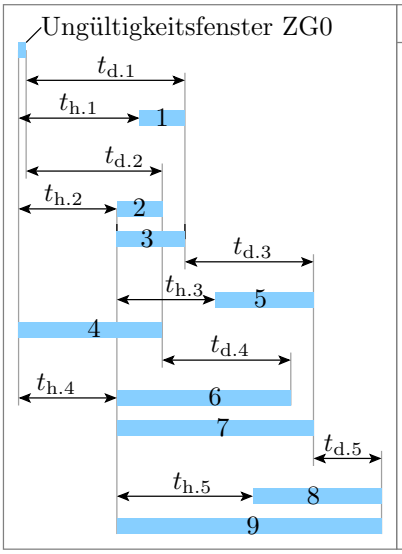
Die automatische Laufzeitanalyse im Rechner arbeitet mit Zeitgruppen. Eine Zeitgruppe ist eine Zusammenfassung von Signalen, z.B. aller Eingabesignale.



Halte- und Verzögerungszeiten zwischen Zeitgruppen:

Zeitgruppe	t_h	t_d
von 0 bis 1	t_{h1}	t_{d1}
von 0 bis 2	t_{h2}	t_{d2}
von 0 bis 3	$\min(t_{h1}, t_{h2})$	$\max(t_{d1}, t_{d2})$

Zeitgruppenweise Berechnung der Halte- und Verzögerungszeiten durch fortgesetzte Additionen, Minimum- und Maximumbildung.



Zeitgruppenhaltezeit	Zeitgruppenverzög.
$t_{H.1} = t_{h.1}$	$t_{D.1} = t_{d.1}$
$t_{H.2} = t_{h.2}$	$t_{D.2} = t_{d.2}$
$t_{H.3} = \min(t_{H.1}, t_{H.2})$	$t_{D.3} = \max(t_{D.1}, t_{D.2})$
$t_{H.5} = t_{H.3} + t_{h.3}$	$t_{D.5} = t_{D.3} + t_{d.3}$
$t_{H.4} = \min(t_{H.2}, 0)$	$t_{D.4} = \max(t_{D.2}, 0)$
$t_{H.6} = t_{H.4} + t_{h.4}$	$t_{D.6} = t_{D.4} + t_{d.4}$
$t_{H.7} = \min(t_{H.5}, t_{H.6})$	$t_{D.7} = \max(t_{D.5}, t_{D.6})$
$t_{H.8} = t_{H.7} + t_{h.5}$	$t_{D.8} = t_{D.7} + t_{D.5}$
$t_{H.9} = \min(t_{H.6}, t_{H.8})$	$t_{D.9} = \max(t_{D.6}, t_{D.8})$

4 Speicher

Speicher

Die bisher behandelten Verarbeitungsfunktionen bilden aus Eingaben mit einer Verzögerung im Bereich von t_h bis t_d nach einer Funktion $f(x)$ Ausgaben:

```
y <= <ungueltig> after th, f(x) after td;
```

Rechner und andere digitale Schaltungen benötigen zusätzlich Speicher für die Abtastung und Aufbewahrung von Daten:

- Latches und Register für einzelne Bits oder Bitvektoren,
- adressierbare Blockspeicher zum wahlfreien Lesen und Schreiben großer Datenmengen,
- anders organisierte Blockspeicher (FIFO, Stapelspeicher, ...),
- ...

Latches und Register

Latches übernehmen zustands- und Register taktflankengesteuert:

```
-- Latch ohne Prozessrahmen und Zeitverhalten
if E='1' then
 y <= x;
end if;
-- Register ohne Prozessrahmen und Zeitverhalten
if rising_edge(T) then
 y <= x;
end if;
```

(x, y – Ein- und Ausgabesignal, Bitvektortyp; E, T – Übernahme- und Taktsignal, Bittyp; rising_edge() – true, wenn steigende Flanke).

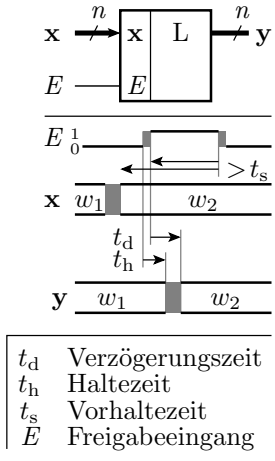
Übernahme- und Taktsignale müssen zeitgenau und glitch-frei sein.

4.1 Latches

Latchmodell mit Prozessrahmen

Das Ausgabesignal y ist sowohl nach jeder Änderung des Freigabe- (Enable-) Signals E als auch nach jeder Eingabeänderung neu zu berechnen. Weckliste mit beiden Signalen.

```
signal x, y: <Bit- oder Bitvektortyp>
signal E: std_logic;
...
process(x, E)
begin
 if E='1' then
  y <= x;
 end if;
end process;
```

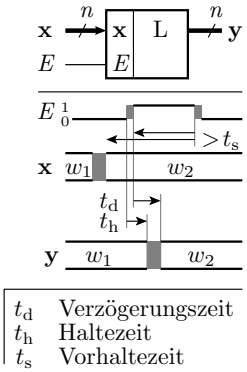


t_d	Verzögerungszeit
t_h	Haltezeit
t_s	Vorhaltezeit
E	Freigabeeingang

Zusätzlich Übernahmeverzögerung.

Speicherung gültiger Werte, nur wenn ...

Ergänzung des Zeitverhaltens



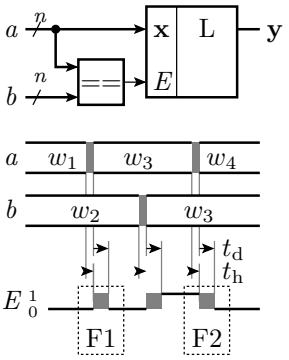
- Übernahme um t_h und t_d verzögert.
- Speicherung gültiger Werte nur, wenn bei Deaktivierung von E alle Eingabesignale für mindestens eine Vorhaltezeit t_s stabil anliegen.

t_d	Verzögerungszeit
t_h	Haltezeit
t_s	Vorhaltezeit

```
process(x, E)
begin
 if rising_edge(E) or (E='1' and x'event) then
  y <= <ungueltig> after th,
       x after td;
 elsif falling_edge(E) and (E'last_event < ts
       or x'last_event < ts) then
  y <= <ungueltig>;
 end if;
end process;
```

Die Tücken einer Schaltung mit Latches

```
signal a, b, y: std_logic_vector(n-1 downto 0);
...
process(a, b)
begin
 if a = b then
  y <= a;
 end if;
end process;
```



Laut Zielfunktion wird der Wert von a gespeichert, wenn er mit dem von b übereinstimmt. Das dargestellte Simulationsmodell ohne Zeitverhalten verhält sich auch so, aber mit Verzögerungszeiten andere Funktion (nicht laufzeittolerant).

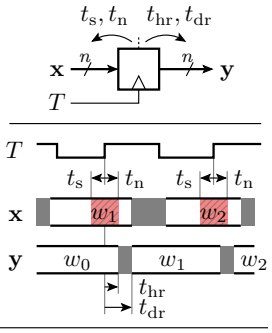
Fehlersituation F1: Bei einer Änderung von a ist dessen Wert ungültig. Dabei ist nicht für jede Änderung ausschließbar dass er kurzzeitig gleich dem von b ist und das Latch den Wert w_3 oder den ungültigen Wert davor übernimmt.

Fehlersituation F2: Das Freigabesignal E wird erst deaktiviert, wenn a wieder ungleich b ist. Da der Vergleich das Signal von E gegenüber der Änderung von b verzögert, wird der Wert w_4 oder der ungültige Wert davor übernommen.

Nicht laufzeitrobust. Abweichungen zwischen Simulationsergebnis und tatsächlichem Verhalten schwer ausschließbar. Experten beherrschen auch solche Entwürfe. Synthese versagt oft (siehe später F3).

4.2 Register

Register als Abtastelement



Verarbeitungsergebnisse müssen innerhalb ihrer Gültigkeitsfenster abgetastet werden. Robustester Umgang mit Zeittoleranzen: taktflankengesteuerte Registerabtastung.

t_s	Vorhaltezeit	t_{dr}	Verzögerungszeit

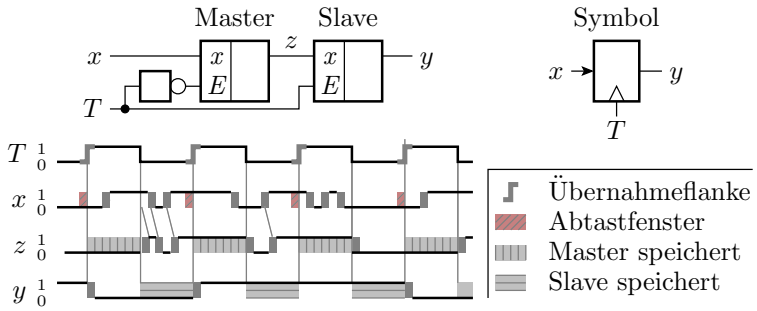
```
signal x,y: std_logic_vector
       (n-1 downto 0);
signal T: std_logic;
...
process(T)
begin
 -- Beschreibung ohne Zeitverhalten
 if rising_edge(T) then
  y <= x;
 end if;
end process;
```

Simulationsmodell mit Zeitverhalten

```
process(T)
begin
 if rising_edge(T) then
  y <= <ungueltig> after
           thr, x after tdr;
  -- Kontrolle Abtastfenster
  wait for tn;
  if x'last_event < ts+tn then
   y <= <ungueltig>;
  end if;
 end if;
end process;
```

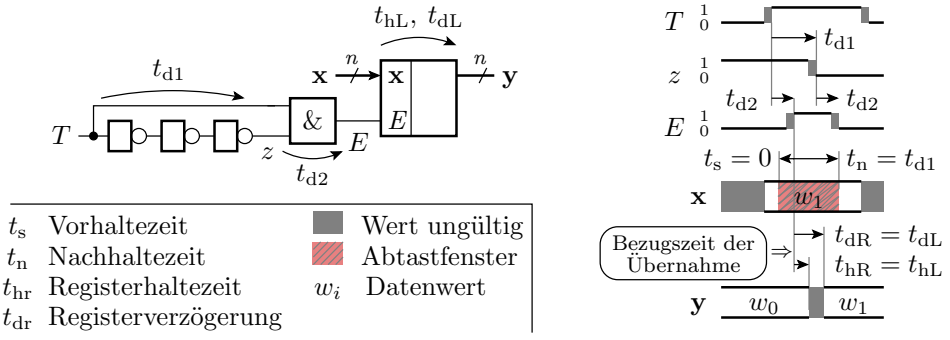
- Halte- und Verzögerungszeit beziehen sich auf die Taktflanke.
- Signalwechsel und ungültige Werte im Abtastfenster invalidieren den gespeicherten Wert.

Master-Slave-Prinzip



Vor der aktiven Taktflanke übernimmt der Master und der Slave speichert. Nach der aktiven Taktflanke gibt der Master die gespeicherten Daten an den Slave weiter. Die Eingabedaten brauchen eine Vorhaltezeit ($t_s > 0$), aber keine Nachhaltezeit ($t_n = 0$).

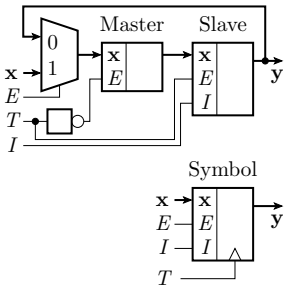
Gepulstes Latch als Register



Erzeugung eines kurzen Freigabeimpulses für ein Latch an der aktiven Taktflanke. Halbe Latch-Anzahl gegenüber Master-Slave-Lösung. Empfindlicher gegenüber Laufzeitungenauigkeiten, d.h. schwerer zu entwerfen. Eingabedaten brauchen keine Vorhaltezeit ($t_s = 0$), dafür eine Nachhaltezeit ($t_n > 0$).

Register mit Freigabe- und Initialisierungseingang

```
signal x,y: std_logic_vector (n-1 downto 0);
signal E, I, T: std_logic;
...
process(I, T):
begin
 if I='1' then
  y <= <Anfangswert>;
 elsif E='1' and rising_edge(T) then
  y <= x;
 end if;
end process;
```

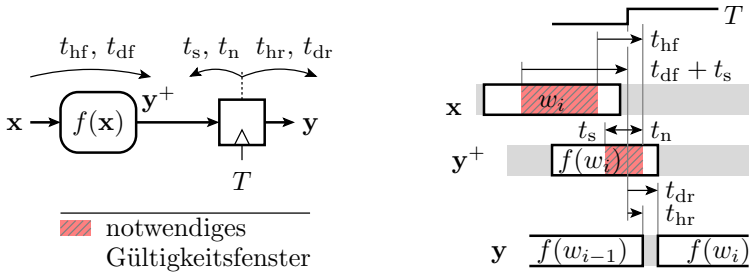


- Initialisierung: Zustandsgesteuertes (Rück-) Setzen des Slaves[10].
- Freigabe: Taktflankengesteuerte Übernahme des Ist- oder des Eingabewerts.

4.3 Verarbeitung und Abtastung

Abtastung nach der Verarbeitung

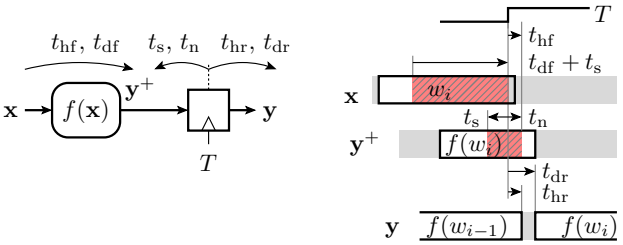
Bei der Abtastung von Verarbeitungsergebnissen ist der Abtastwert auch nur bei jeder aktiven Taktflanke neu zu berechnen. Halte- und Verzögerungszeit sind die des Registers. Das notwendige Gültigkeitsfenster der Eingabe verschiebt sich um die Halte- und Verzögerungszeit der Verarbeitungsfunktion nach vorn.



(t_{hf}, t_{df} – Halte- und Verzögerungszeit Verarbeitung; t_s, t_n, t_{hr}, t_{dr} – Vorhalte-, Nachhalte-, Halte- und Verzögerungszeit Register.)

VHDL-Modell für $t_{hf} \leq t_n$

```
process(T)
begin
 if rising_edge(T) then
  y <= <ungueltig> after
            thr, f(x) after tdr;
  -- Kontrolle Abtastfenster
  wait for tn-thf; -- muss positiv sein
  if x'last_event < ts + tdf + tn - thf then
   y <= <ungueltig>;
  end if;
 end if;
end process;
```

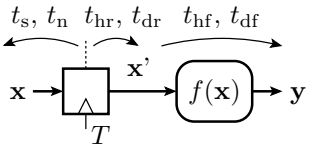


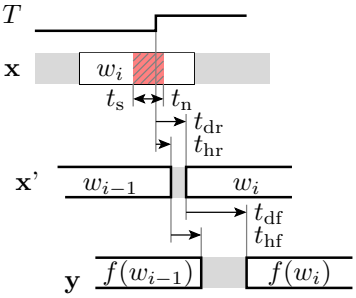
Für $t_{hf} > t_n$ ist die Kontrolle der Gültigkeit komplizierter.

[10] Nur laufzeitrobust, wenn Änderungen von I am Takt ausgerichtet sind.

Verarbeitung abgetasteter Signale

Neuberechnung auch nur bei jeder aktiven Taktflanke. Vor- und Nachhaltezeit sind die des Registers. Die Halte- und Verzögerungszeit verlängern sich um die der Verarbeitungsfunktion. Alle Zeitfenster über Laufzeitanalyse (ohne Simulation) bestimmbar.

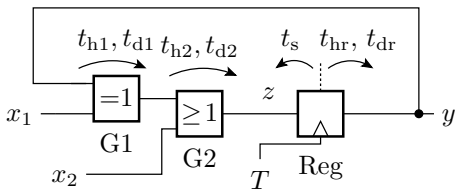




(t_{hf}, t_{df} – Halte- und Verzögerungszeit der Verarbeitungsfunktion; t_s, t_n, t_{hr}, t_{dr} – Vorhalte-, Nachhalte-, Halte- und Verzögerungszeit des Registers.)

```
process(T)
begin
 if rising_edge(T) then
  y <= <ungueltig> after
    thr + thf, f(x) after tdr+tdf;
  -- Kontrolle Abtastfenster
  wait for tn;
  if x'last_event < ts + tn then
   y <= <ungueltig>;
  end if;
 end if;
end process;
```

Aufgabe: Bestimmung der Gültigkeitsfenster



$t_{h1} = 1\,\text{ns}$
$t_{d1} = 4\,\text{ns}$
$t_{h2} = 1\,\text{ns}$
$t_{d2} = 3\,\text{ns}$
$t_s = 2\,\text{ns}$
$t_n = 0\,\text{ns}$
$t_{hr} = 1\,\text{ns}$
$t_{dr} = 2\,\text{ns}$

1. In welchem Zeitfenster vor der aktiven Taktflanke müssen die Signale x_1, x_2, z und y gültig sein?
2. In welchem Zeitfenster nach der aktiven Taktflanke ist y gültig?
3. Wie groß muss der Zeitabstand zwischen zwei aktiven Taktflanken mindestens sein?

Zur Kontrolle

1. Notwendige Gültigkeit vor der aktiven Taktflanke:

Signal	von	bis
z	$t_s = 2\,\text{ns}$	0
x_2	$t_s + t_{d2} = 5\,\text{ns}$	$t_{h2} = 1\,\text{ns}$
x_1 und y	$t_s + t_{d2} + t_{d1} = 9\,\text{ns}$	$t_{h2} + t_{h1} = 2\,\text{ns}$

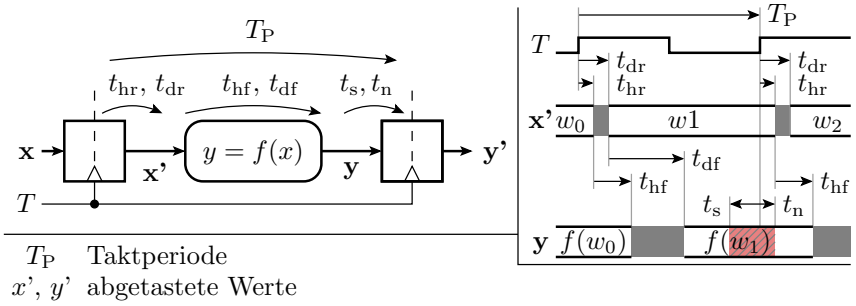
2. Das Signal y ist bis $t_{hr} = 1\,\text{ns}$ und ab $t_{dr} = 2\,\text{ns}$ nach der aktiven Taktflanke gültig.
3. Mindestzeitabstand zwischen zwei aktiven Taktflanken:

$$t_{d1} + t_{d2} + t_s + t_{dr} = 11\,\text{ns}$$

4.4 Register-Transfer-Funktionen

Register-Transfer-Funktion

Die Register einer Schaltung haben in der Regel denselben Takt. Verarbeitungsfunktionen sind von Registern eingerahmt. Die Verarbeitung beginnt mit der aktiven Taktflanke und endet mit der nächsten aktiven Taktflanke.



Die Werte von y sind im weißen Zeitfenster gültig und müssen zur korrekten Übernahme im gestreiften Abtastfenster gültig sein.

Die Summe aus den Verzögerungszeiten des Registers und der Verarbeitungsfunktion darf nicht größer als die Taktperiode sein:

$$T_P \geq T_{P.min} = t_{dr} + t_{df} + t_s$$

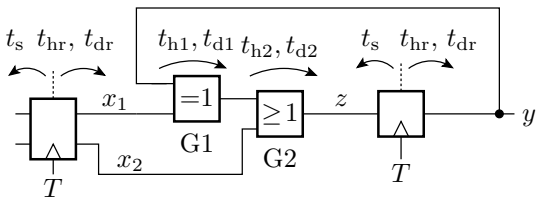
Die Nachhaltezeit des Registers darf nicht größer als die Summe der Haltezeiten sein:

$$t_n \leq t_{n.max} = t_{hr} + t_{hf}$$

Beide Kontrollen hängen nicht von den zu verarbeitenden Daten und Signalverläufen ab und können über eine Laufzeitanalyse auch ohne Simulation mit Zeitverhalten durchgeführt werden.

RT-Funktionen werden in der Regel ohne Berücksichtigung von Halte-, Verzögerungs- und Vorhaltezeiten simuliert.

Aufgabe: Bestimmung der maximalen Taktfrequenz



$t_{h1} = 1\,\text{ns}$
$t_{d1} = 4\,\text{ns}$
$t_{h2} = 1\,\text{ns}$
$t_{d2} = 3\,\text{ns}$
$t_s = 2\,\text{ns}$
$t_n = 0\,\text{ns}$
$t_{hr} = 1\,\text{ns}$
$t_{dr} = 2\,\text{ns}$

1. Wie groß muss der Zeitabstand T_p zwischen zwei aktiven Taktflanken mindestens sein?
2. Wie groß darf die Taktfrequenz $f_T = T_P^{-1}$ maximal sein?

Zur Kontrolle

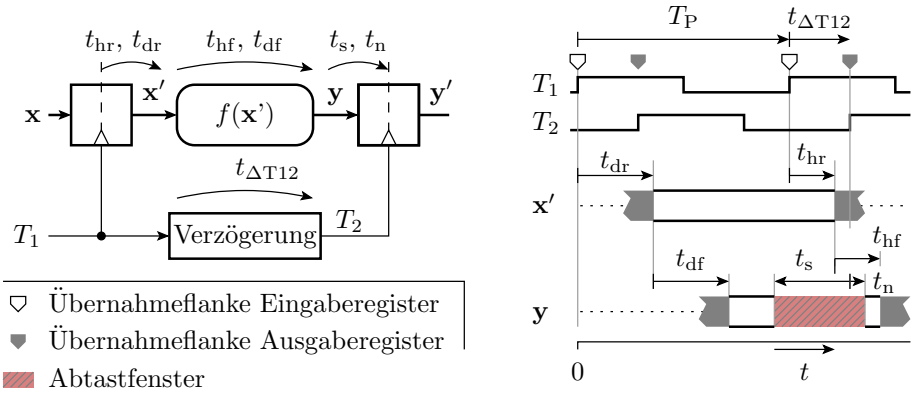
1. Minimaler Zeitabstand zwischen zwei aktiven Taktflanken:

$$T_p \geq t_{d1} + t_{d2} + t_s + t_{dr} = 11\,\text{ns}$$

2. Maximale Taktfrequenz:

$$f_T = T_P^{-1} \leq \frac{1}{11\,\text{ns}} \approx 91\,\text{MHz}$$

Taktversatz



Die Taktflanken zwischen Eingangs- und Ausgangsregister können um einen Zeit $t_{\Delta T12}$ versetzt sein, absichtlich zur Erhöhung der max. Taktfrequenz oder unbeabsichtigt durch unterschiedliche Verzögerungen von der gemeinsamen Taktquelle.

Die Bedingung für eine korrekte Datenübernahme mit Taktversatz lauten:

- $t_{dr} + t_{df} + t_s \leq T_P + t_{\Delta T12}$
- $t_{hr} + t_{hf} \geq t_n + t_{\Delta T12}$

Aus diesen Bedingungen ergibt sich für den maximal zulässigen Taktversatz:

$$t_{\Delta T12} \leq t_{hr} + t_{hf} - t_n$$

Mit dem maximal zulässigen Taktversatz beträgt die maximale Taktfrequenz:

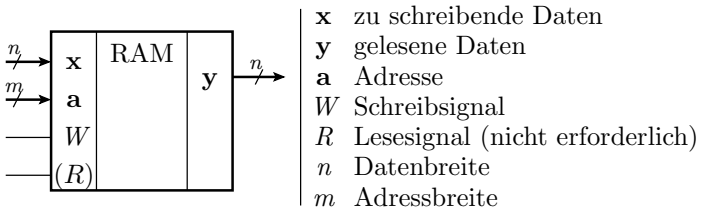
$$f_T \leq \frac{1}{t_{dr} + t_{df} + t_s + t_n - (t_{hr} + t_{hf})}$$

Die maximal mögliche Anzahl von Berechnungsschritten pro Zeit hängt nicht von der Verzögerung, sondern von der Differenz aus Verzögerungs- und Haltezeit ab. Man kann sogar Rechenwerke bauen, die gleichzeitig mehr als eine Berechnung durchführen, eine sog. Wellen-Pipeline.

4.5 Adressierbare Speicher

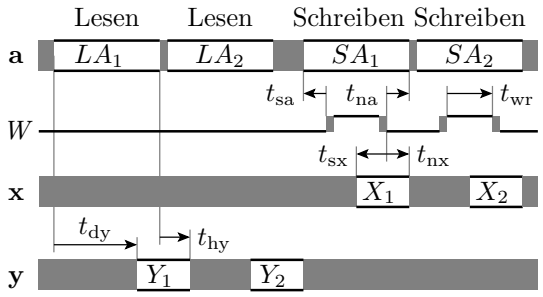
Adressierbare Speicher

Speicherung größerer Datenmengen.



- Speichern: Daten und Adresse anlegen, Schreiben aktivieren.
- Lesen: Adresse anlegen (Lesen aktivieren) und Daten übernehmen.

Beim Schreiben sind ähnlich wie bei Latches Vor- und Nachhaltezeiten zu berücksichtigen. Beim Lesen hat der Speicher wie eine Verarbeitungsfunktion eine Halte- und eine Verzögerungszeit.

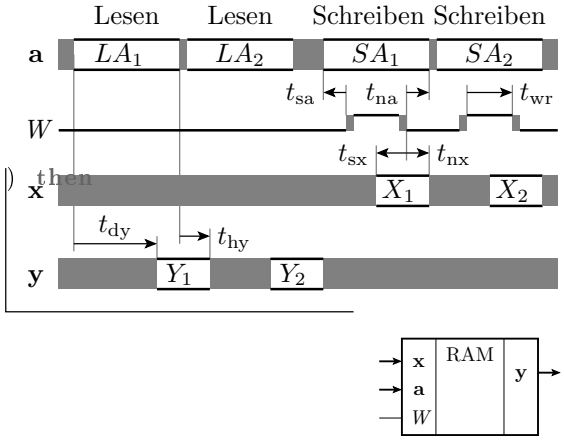


LA_i	Leseadresse
SA_i	Schreibadresse
Y_i	gelesener Wert
X_i	zu schreibender Wert

a	Adresssignal
x	Eingangsdaten
W	Schreibsignal
y	Ausgabedaten
t_{hy}	Lesehaltezeit
t_{dy}	Leseverzögerungszeit
t_{sa}	Adressvorhaltezeit
t_{na}	Adressnachhaltezeit
t_{sx}	Datenvorhaltezeit
t_{nx}	Datennachhaltezeit
t_{wr}	Schreibpulsbreite

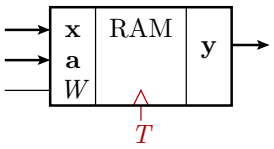
- Beim Schreiben muss die Adresse, während der Schreibpuls aktiv ist, stabil anliegen.
- Die Daten müssen wie bei einem Latch im Zeitfenster um die Deaktivierung des Schreibpulses gültig sein.
- Vor- und Nachhaltezeiten sowie Verzögerungen sind schaltungsbedingt um Zehnerpotenzen länger als bei Gattern und Registern (siehe später Foliensatz F5).

```
process(a, W, x)
begin
 -- Lesezugriff
 y <= <ungueltig> after
    thy, mem(a) after tdy;
 -- Schreibzugriff
 if not(W='0') and
  (a'event or (W'event and a'last_event<tsa)) then
  mem <= <ungueltig>;
 elsif W='1' then
  mem(a) <= <ungueltig> thw, x after tdw;
 elsif W'event and W='0' then
  wait for tnx; <*1>
  if x'last_event < tsx + tnx then
   mem(a) <= <ungueltig>;
  end if;
 end if;
end process;
```



*1: Für $t_{na} > 0$ müsste noch für t_{na} gewartet und die Nachhaltebedingung kontrolliert werden.

Synchrone Speicherschnittstellen

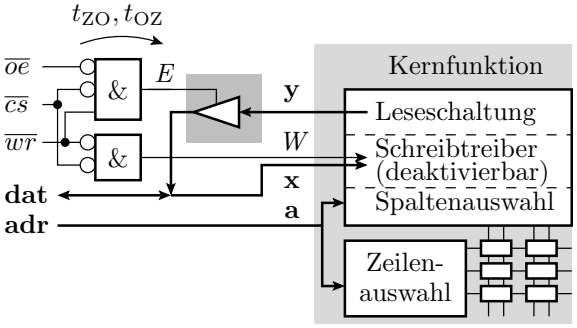


x	zu schreibende Daten
y	gelesene Daten
a	Adresse
W	Schreibsignal
T	Takt

Erweiterung des Blockspeichers um Register zur Ein- und Ausgabe-Synchronisation (siehe später RAM-Controller).

```
process(T)    -- Modell ohne Vorhalte-, Nachhalte-
begin         -- Halte- und Verzoegerungszeiten
 if rising_edge(T) then
  if W='1' then mem(a) <= x;
  else                  y <= mem(a);
  end if;
 end if;
end process;
```

Modell eines asynchronen RAM-Schaltkreises

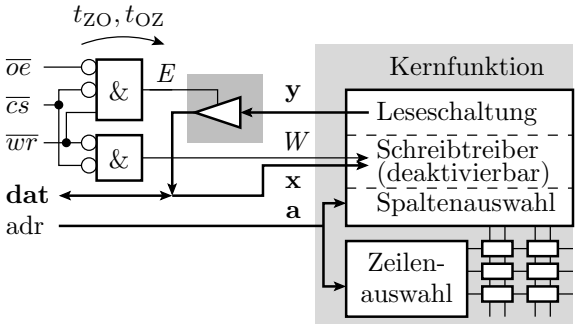


Schaltkreisanschlüsse:
- $\overline{oe}$ Ausgabefreigabesignal
- $\overline{cs}$ Schaltkreisauswahlsignal
- $\overline{wr}$ Schreibsignal
- **adr** Adresssignal
- **dat** bidirektionales Datensignal

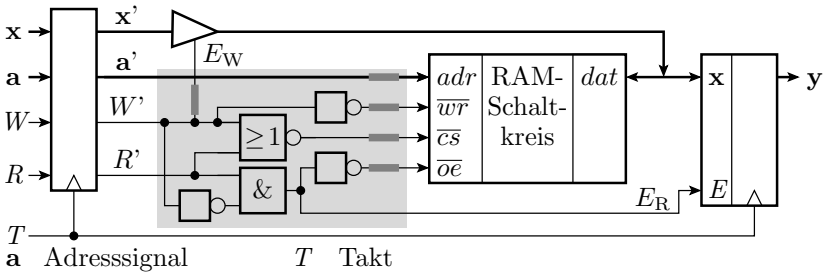
Verzögerung des Ausgabetreibers:
- t_{OZ} Deaktivierungsverzögerung
- t_{ZO} Aktivierungsverzögerung (im Beispiel unten 2×4 ns)

RAM-Schaltkreise haben zusätzlich eine Schnittstellenschaltung. Asynchrone RAMs haben typisch einen bidirektionalen EA-Datenbus und die low-aktiven Steuersignale $\overline{cs}$ (**C**hip **S**elect), $\overline{wr}$ (**Wr**ite) und $\overline{oe}$ (**O**utput **E**nable). Das Simulationsmodell besteht aus dem Modell für die Kernschaltung und einem Modell für die Schnittstelle.

```
signal adr: unsigned(17 downto 0);
signal dat, y: std_logic_vector(15 downto 0);
signal n_oe, n_cs, n_wr, W, E: std_logic;
...
-- Ausgabetreiber
process(oe, y)
begin
 if  E='1' then
  dat <=y after ...;
 else
  dat <= (others=>'Z') after ...;
 end if; end process;

-- Gatter mit Zeitverhalten
 W <= not n_cs and not n_wr after ...;
 E <='X', not n_oe and not n_cs and n_wr after ...;

-- Instanziierung der Kernfunktion mit Zeitparametern
 ram(W=>W, a=>adr, x=>dat, y=>y, thy=>2 ns, tdy=>10 ns,
  tsa=>0 ns, tna=>0 ns, twr=>8 ns, tsx=>6 ns);
```



RAM-Controller



- **a** Adresssignal
- **x** Eingabedaten
- **y** Ausgabedaten
- W Schreibsignal
- R Lesesignal
- T Takt
- E_W Freigabesignal für Schreibdaten
- E_R Übernahmesignal für gelesene Daten
- individuell abzugleichende Verzögerung
- laufzeitkritische Teilschaltung

Für die Nutzung eines SRAMs ist eine Schnittstellenschaltung mit dem grau unterlegten laufzeitkritischen Teil zu entwickeln[11].

[11] Die Laufzeiten müssen auf wenige Nanosekunden genau stimmen. In ISE Erzeugung mit dem MIG (**M**emory **I**nterface **G**enerator), der für die zeitkritischen Schaltungsteile die Verdrahtung mit generiert.