Modellieren digitaler Schaltungen mit VHDL

Markus Pfaff, Alexander Schuster

SS 1999

Inhaltsverzeichnis

1 Motivation **5**

1.1 Begriffserklärung 5

1.2 VHDL - Beschreibung digitaler Hardware 6

2 Die sequentiellen Sprachelemente von VHDL **8**

2.1 *entity / architecture* 8

2.2 Prozesse 9

2.3 Variablen und Konstante 10

2.4 Typen 11

2.4.1 Vordefinierte Typen 11

2.4.2 Typdefinition 12

2.4.3 Felder (Arrays) 12

2.4.4 Zusammengesetzte Typen (Records) 13

2.5 Operatoren 14

2.5.1 Logische Operatoren 14

2.5.2 Vergleichsoperatoren 15

2.5.3 Arithmetische Operatoren 15

2.6 Ablaufstrukturen 15

2.6.1 IF-Statement 15

2.6.2 CASE-Statement 16

2.6.3 LOOP-Statement 16

2.7 Assertion 17

2.8 Unterprogramme 18

2.8.1 Funktionen 18

2.8.2 Prozedur 19

3 Simulation **19**

3.1 Simulationsvorbereitung 19

3.2 Simulationsstart 20

3.3 Hardware 20

3.4 Simulationszeit 20

3.5 Kommunikation zwischen Prozessen 22

4 Der Simulation Cycle, signals, processesund Attribute **24**

4.1 Der *Simulation Cycle* 25

4.2 Der *Simulation Cycle*mit *driver*(*signals*) 25
4.3 Hardware-Simulation 29
4.3.1 Simulation mit `wait for` 30
4.3.2 Simulation mit `wait on` 31
4.4 Verzögernde *signals* 31
4.5 Zeitfortschritt in der Simulation 33
4.6 Revision des *Simulation Cycle* 34
4.7 Alles über `wait` 35
4.8 Attribute 36
4.9 Der neue *Simulation Cycle* 36
4.10 Was kann VHDL bisher? 39

5 Verzögerung und ihre Modellierung 39
5.1 Waveform und Stimuli 40
5.2 Physikalische Verzögerung 42
5.3 Modellierung der Verzögerung: *transport delay* 43
5.4 Auswirkungen des *transport delay*auf den driver 44

6 Trägheit und ihre Modellierung 45
6.1 Physikalische Trägheit 46
6.2 Modellierung der Trägheit: *inertial delay* 48
6.3 Auswirkungen des *inertial delay*auf den *driver* 48
6.4 Zur Beachtung beim *inertial delay* 50

7 Idealisierung 51
7.1 Delta Delay 51
7.2 Simulation Cycle 54
7.3 Postponed Prozesse 55

8 Strukturbeschreibung 56
8.1 Nachteile des bisherigen Entwurfs 56
8.2 Strukturierungsmechanismen in VHDL 56
8.3 Entity 57
8.4 Architecture 58
8.5 Instantiation 59
8.6 Instantiation mittels Generate 61
8.7 Parametrisierung 61

8.8 Instantiation mit Components 62

9 Spracherweiterungen **63**

9.1 Packages 63
9.2 Libraries 64
9.3 Package: IEEE.std_logic_1164 65

10 Concurrent Statements **65**

10.1 Sensitivity List 65
10.2 Concurrent Signal Assignment 66
10.3 Conditional Signal Assignment 66
10.4 Selected Signal Assignment 66
10.5 Concurrent Assertion Statement 67
10.6 Concurrent Procedure Call 67

11 Busse **67**

11.1 Eingänge mit mehreren Treibern 67
11.2 Resolution Funktion 68
11.3 Busse 70
11.4 std_logic und std_ulogic 71

12 Weiteres **72**

12.1 Schritte zur Simulation 72
12.2 Overloading 72
12.3 Package std.textio 73
12.4 Allgemeinere Literale 73

Literatur **75**

1 Motivation

Die Entwicklung von elektronischen Schaltungen ist eng mit der Datenverarbeitung verbunden. Wie der Mensch in seiner gesamten Entwicklung nach Hilfsmitteln gesucht hat um sich das Leben zu erleichtern, versuchten schon viele Wissenschafter in ihrer Zeit, eine automatische Rechenmaschine zu konstruieren. Bis Konrad Zuse 1935 beschränkten sich diese Versuche u.a. von John Napier (1600), Schikkard (1623), Leibniz (1673) P.M. Hahn (1775) Babbage (1820) und Hollerith (um 1890; er gründete die spätere IBM) auf mechanische Rechenhilfen [KG75]. Erst Zuse versuchte es mit Relais und später mit Elektronenröhren [Zus86].

1945 formulierte John v. Neumann die Grundlegende Architektur unserer heutigen Computer, während im selben Jahr der erste frei programmierbare Computer ENIAC in Amerika entwickelt wurde [KG75]. Der Vormarsch der Computer war und ist ohne digitalen Schaltungen undenkbar, aber genauso ist die Entwicklung in der Technik der digitalen Schaltung von der Revolution der Computer geprägt. Zu Beginn die Komplexität der Schaltungen noch überschaubarer und man hatte nicht die Rechenleistung, um neue Schaltungen vor deren Entwicklung in Rechenanlagen zu simulieren. Diese Simulation ist es aber, was VHDL so wertvoll in der Entwicklung neuer digitaler Schaltungen macht.

1.1 Begriffserklärung

Hardware ganz allgemein ist das, was uns die deutsche Übersetzung dieses englischen Wortes auch vermittelt: „harte Ware“. Man kann sich unter Hardware also etwas Dingliches vorstellen, etwas Angreifbares, physisch vorhandenes. Dies könnten z.B. sein:

- Rechner (Computer), EDV-Anlagen, andere Automaten
- Mikroprozessoren, Ein- und Ausgabe Schnittstellen
- Register, Multiplexer
- FlipFlops, Inverter, Gatter
- Transistoren, Widerstände, Kondensatoren
- Siliziumstrukturen, Halbleiterschichten
- Elektronen, Ionen, Photonen

Eigenschaften

Zwei Eigenschaften von (digitaler) Hardware sind von besonderem Interesse: Durch die Verbindung von einzelnen Elementen einer Schaltung ergibt sich eine *Struktur*. Diese stellt eine Ordnung der Elemente dar.

Alle Elemente eines Hardwaresystems haben ein ganz spezifisches *Verhalten*. Durch die Struktur des Gesamtsystems und der dadurch entstehenden Ordnung, tragen die einzelnen Elemente der digitalen Hardware mit ihrem Verhalten und durch die strukturell vorgegebene Verknüpfung ihres Verhaltens zu einem *Gesamtverhalten* der digitalen Hardware bei.

Das Verhalten dieses Gesamtsystems ergibt sich aus dem Einzelverhalten der Elemente und der Struktur der Kombination dieser einzelnen Elemente. Jedes Element trägt entscheidend zum Verhalten des Gesamtsystems bei, aber auch die Struktur, also die Art der Kombination, beeinflußt das Verhalten des Gesamtsystems. Das erhöht stark die Komplexität eines Systems.

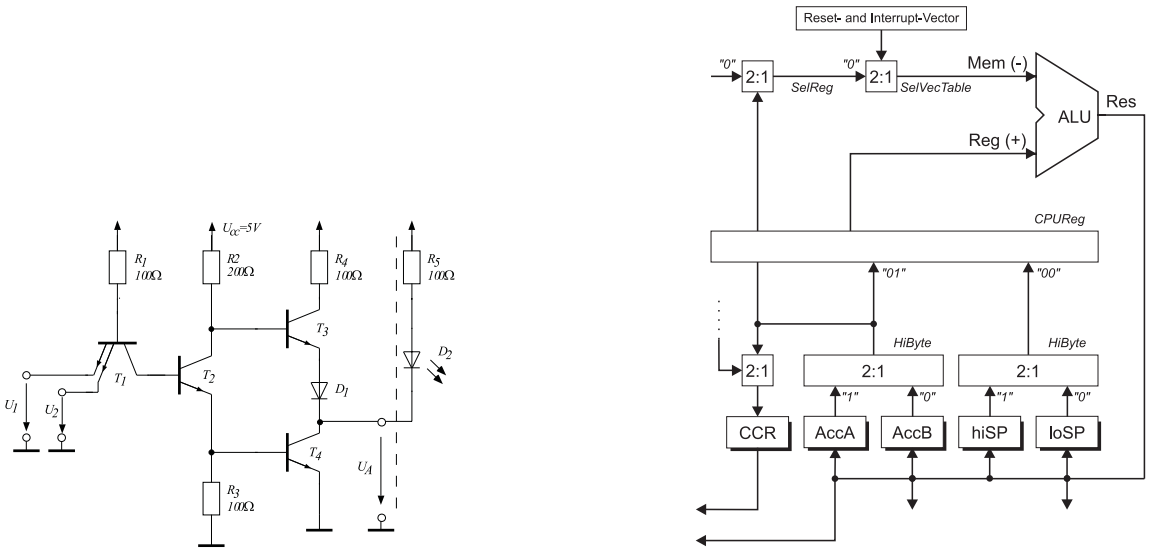


Abbildung 1: Strukturdarstellung

1.2 VHDL - Beschreibung digitaler Hardware

Die Abkürzung *VHDL* steht für VHSIC Hardware Description Language wobei *VHSIC* Very High Speed Integrated Circuits bedeutet. VHDL wurde 1983 vom amerikanischen Department of Defense im VHSIC Programm initiiert, ist seit 1987 als IEEE Standard 1067 genormt und wird alle 5 Jahre überarbeitet [Ash96]. Inzwischen ist VHDL die *quasi* Standard-Hardwarebeschreibungssprache.

Um sich Gedanken über das Verhalten und die Struktur eines Systems zu machen und dieses anderen mitzuteilen, ist eine Beschreibung davon notwendig. Dabei gibt es, je nach Anwendung und System, mehrere Möglichkeiten und Kategorien. So können zumindest folgende Kategorien von Verhaltensbeschreibungen unterschieden werden.

- Mathematische Beschreibung (z.B. Differentialgleichungssystem)
- Funktionale Beschreibung (z.B. LiSP)
- Algorithmische, Prozedurale Beschreibung (C/C++, Java, VHDL)

VHDL dient zur algorithmischen Beschreibung von Struktur und Verhalten eines digitalen Gesamtsystems und zu dessen Simulation. Enthalten sind Sprachelemente, die eine solche Beschreibung im hohen Maße unterstützen und anderen Programmiersprachen sehr ähnlich sind.

Vorteile der Beschreibung digitaler Hardware

Während die algorithmische Beschreibung von Abläufen (Software) von einem Computer Schritt für Schritt ausgeführt wird, dient VHDL einen anderen Zweck. Durch die algorithmische *Beschreibung* kann der Entwickler sich mehr der Funktion und dem Verhalten der Hardware und der Struktur komplexerer Teile widmen. Die Prüfung der elektronischen Bedingungen und der Aufbau der elektronischen Struktur der Elementarbauteile übernimmt dann ein „VHDL Compiler“. Es ergeben sich durch diese Beschreibung der Hardware Vorteile:

- *Eindeutige Kommunikationsgrundlage* Die Beschreibung der Hardware in VHDL kann als gemeinsame Sprache zwischen verschiedenen Teilen eines Unternehmens fungieren. So kann der

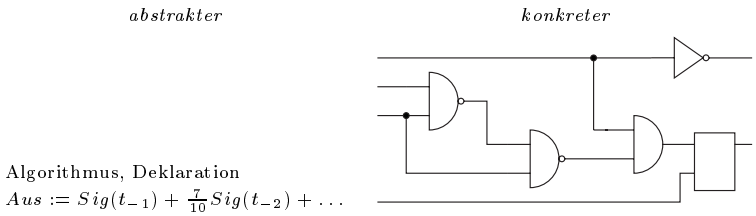


Tabelle 1: Beispiel Hallgerät

Entwickler der Fertigungsabteilung genau mitteilen, was für eine digitale Schaltung gebaut wird und gleichzeitig der Marketing Abteilung klar machen, wie deren Funktionsweise aussieht und wo deren Vorteile liegen. Außerdem kann diese Beschreibung helfen, eindeutig die Urheberschaft der digitalen Schaltung zu klären. Eine solcherart standardisierte Beschreibung ist wie eine gemeinsame Sprache, mit der Ideen und Ansichten anderen Menschen mitgeteilt werden können.

- *Simulation* Die VHDLBeschreibung einer Schaltung stellt zugleich ein ausführbares Modell dar. Dadurch lassen sich die Funktionsweisen des IC schon testen, noch bevor er gebaut ist. Dies übernimmt ein Computer, der das Verhalten der Schaltung aufgrund der Beschreibung simuliert. Dies ist besonders wichtig bei der Entwicklung.
  - Die Fehlersuche wird vereinfacht, da ein Fehler schnell behoben werden kann, ohne die Schaltung wieder neu herstellen zu müssen.
  - Die Kosten werden reduziert, da kein Material als unbrauchbarer Abfall übrigbleibt.
  - Die Entwicklungszeit wird verkürzt und damit das Unternehmen am Markt schlagkräftiger.
  - Es kann die korrekte Funktionsweise nachgewiesen werden, wodurch in der Anwendung klar ist, wie sich das System verhält. Eine Uhr zeigt dann die korrekte Zeit an und nicht etwas anderes.
- *Hardwarerealisierung und Synthese* Durch die so erfolgte eindeutige Beschreibung einer Schaltung kann diese automatisiert erzeugt werden. Die Umsetzung in Basiselemente (Transistoren, Widerstände, ...) erfolgt vom Computer, der gleichzeitig aufgrund der Beschreibung in der Lage ist, hinsichtlich verschiedener Parameter zu optimieren. (Reaktionszeit, Fläche, Leistungsaufnahme, ...) Damit ergibt sich aus der VHDL Beschreibung ein fertiges Produkt.

Ein wesentlicher Vorteil algorithmischer Beschreibung ist darüberhinaus die Verwendung von *Abstraktion* und *Hierarchie* um die Komplexität des beschriebenen Systems zu bewältigen. VHDL unterstützt die Unterteilung des Entwurfs in Komponenten und die hierarchische Zusammensetzung dieser Komponenten zu einem Gesamtsystem. Die Komplexität dieser Teile kann vom einfachen Gatter (z.B. NAND) bis hin zu komplexen Funktionseinheiten (z.B. Prozessorkern) reichen. Weiters erlaubt VHDL eine unterschiedliche Beschreibung jeder Design-Einheit. So kann durch eine algorithmische Beschreibung mit den Mitteln einer höheren Programmiersprache das *Verhalten* definiert oder die *Struktur* als Verbindung von Elementen im Sinne einer Hierarchie beschrieben werden. Ein Beispiel dazu zeigt Tabelle 1.

So gleicht der Entwicklungsprozess von Hardware mit Hilfe von VHDL auch dem bereits aus der Softwareentwicklung bekannten Modell: Ein iterativer Prozess erlaubt den Entwurf und die schrittweise Verfeinerung der Beschreibung eines Systems von einem sehr abstrakten Modell bis zur Realisierung der Hardware. Aus einer (informalen) Spezifikation wird ein Verhaltensmodell mittels VHDL

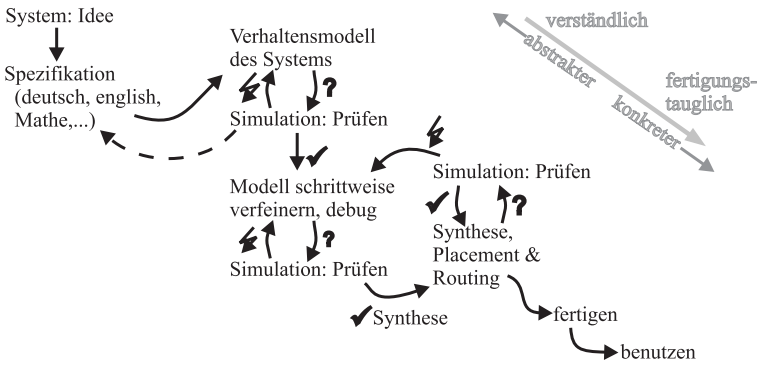


Abbildung 2: Designflow

erstellt, das durch Simulation auf seine Tauglichkeit geprüft wird. Schrittweise wird dieses Modell verfeinert, bis es synthetisiert werden kann. Die Funktionsfähigkeit kann zu jeder Zeit durch Simulation überprüft werden. Sind die Syntheseergebnisse befriedigend, kann das System gefertigt werden (s. Abb. 2).

2 Die sequentiellen Sprachelemente von VHDL

VHDL enthält auch die grundlegenden Sprachelemente jeder anderen Programmiersprache in einer eigenen Syntax. Darüber hinaus nimmt VHDL speziell Rücksicht auf die Dynamik einer digitalen Schaltung und stellt zusätzliche Sprachkonstrukte zur Verfügung. VHDL ist nicht objektorientiert im strengen Sinn, da es keine Klassen mit Vererbung gibt. Es gibt aber ein Konzept zur Trennung von der Definition der Schnittstelle (Ein- und Ausgaben) eines Schaltungselementes und Implementation des Elementes. IEEE entwickelt gerade einen neuen Standard, in dem VHDL um objektorientierte Konzepte erweitert wird. Es gibt auch Konzepte um die Dynamik einer Schaltung modellieren zu können.

2.1 entity / architecture

Das *entity / architecture* Konzept stellt die Trennung zwischen der Definition der Schnittstelle und der Implementation der Schaltung dar. Durch diese Trennung ist es leicht möglich, eine Schaltung zu modularisieren. Die Design-Einheiten arbeiten zusammen, kennen aber vom jeweils anderen nur die Schnittstelle. Wie die konkrete Realisierung des anderen aussieht, ist für jedes Modul zunächst auch nicht von Bedeutung. Damit kann die Implementierung von Modulen geändert werden, ohne daß andere Module dadurch in Mitleidenschaft gezogen werden. `entity` Beschreibt die Schnittstelle oder die externe Sicht einer Design-Einheit, einer Entität. Die Ein- und Ausgabe erfolgt über so genannte `ports`. Sie stellen für die Entität die „Türe“ zur Außenwelt dar. Über diese läuft jede Kommunikation zu den restlichen Teilen des Systems. Die Erklärung von `generic` erfolgt später.

```
entity entity_name is
  [generic ( generic_list );]
  [port ( port_list );]
  [declarations;]
end entity entity_name;
```

Eine Entität mit Eingabe und Ausgabe könnte so aussehen:

```
entity Subtractor is
```

```
    port (      a : in  integer;
                b : in  integer;
           result : out integer);
end entity Subtractor;
```

Dieses Beispiel zeigt eine Deklaration mit Ein- und Ausgabeports. Die Definition durch die Angabe von `port` gefolgt von der Liste der Ein- und Ausgabe - Parameter. `in` definiert einen Eingabeport, `out` einen Ausgabeport und `inout` steht für einen Port, der sowohl Ein- als auch Ausgabe darstellt.

Eine `architecture` einer Entität ist eine konkrete Implementierung der Verhaltensbeschreibung. Es ist klar, daß eine Entität mehrere Architekturen haben kann. Diese können dabei verschiedene Entwicklungsstadien beschreiben, in der verschiedene Lösungsmöglichkeiten probiert werden. Oder je nach Anforderung kommt z.B. eine in der Reaktionszeit oder in der Fläche optimierte Lösung zur Anwendung. Die Architektur darf nicht den selben Namen wie die Entität haben. Daher bietet sich zur Unterscheidung an, dass z.B. `Bhv` (für behaviour = Verhalten) oder `Struct` (für structure = Struktur) an den Namen der Entität angehängt wird. Die Auswahl einer bestimmten `architecture` erfolgt dann später über eine `configuration`.

```
architecture architecture_name of entity_name is
  [architecture_declarations]
begin
  concurrent_statements
end [ architecture ] [ architecture_name ];
```

Die Architektur beschreibt das Verhalten und die Struktur der Entität. In der Deklaration der Entität gibt es keine Codefragmente zu diesem Zweck. Diese stehen nur in der Architektur.

2.2 Prozesse

Ein `process` formuliert dynamisches Verhalten und reagiert auf definierte Ereignisse. Ein solcher Prozess ist einem Programm einer Programmiersprache sehr ähnlich: die Anweisungen werden sequentiell abgearbeitet, es gibt Kontrollanweisungen zur Steuerung, lokale Variablen und Datentypen können verwendet werden und Unterprogrammtechnik kann zum Einsatz kommen (*Prozeduren*, *Funktionen*).

Da mit VHDL Hardware simuliert werden soll, und Hardwareelemente *immer gleichzeitig* aktiv sind, gibt es spezielle Konstrukte, die festlegen, wann der Prozess zu aktivieren ist. Üblicherweise wird das entweder durch eine *sensitivity list* oder `wait`-Statements realisiert. Die Arbeitsweise eines Prozesses ist geprägt davon, daß er Anweisungen ausführt und dann auf Ereignisse wartet. Tritt es ein, führt er die Anweisungen weiter aus und wartet dann wieder. Es kann innerhalb einer Architektur mehrere Prozesse geben.

```
[process_label:] process [ ( sensitivity_list ) ] [ is ]
  process_declarations
begin
  sequential_statements
end process [ process_label ] ;
```

Ist der Prozess am Ende der Anweisungsfolge angelangt, so beginnt er wieder mit der ersten Anweisung. Ein Prozess läuft oder wartet ewig und kann nicht gestoppt werden. Keine Regel ohne Ausnahme, für die Entwicklungsphase und die Fehlersuche gibt es Hilfsmittel, die einen Prozess mit einer Fehlermeldung beenden lassen. Dies dient der Fehlersuche und hat nur während der Simulation ihren Sinn. Solche Anweisungen können aber nicht auf konkrete Hardware abgebildet werden und müssen vor der Erzeugung der Hardware entfernt werden. Ein Prozess könnte zum Beispiel so aussehen:

```
Subtracting: process is
  -- local declarations
  variable AwakenPeople           : integer;
  variable People, SleepingPeople : integer;
```

```
begin
  -- sequential action
  People := PeopleCount;    -- externaly defined
  SleepingPeople := SleepingPeopleCount;
  AwakenPeople := People-SleepingPeople;
  wait;
end process Subtracting;
```

2.3 Variablen und Konstante

Wie jede andere Programmiersprache kennt auch VHDL Variablen und Konstanten. *Identifier* dienen dazu Objekte zu benennen. Mit Ausnahme einiger reservierter Wörter kann der Benutzer beliebige Namen verwenden, dabei gilt:

- Zeichensatz `'a'..'z', 'A'..'Z', '0'..'9', '_'`
- das erste Zeichen *muss* ein Buchstabe sein
- der Unterstrich darf weder am Beginn noch am Ende stehen
- Groß- und Kleinschreibung werden nicht unterschieden

VHDL '93 erlaubt als Erweiterung auch Bezeichner aus allen ASCII Zeichen bestehen können, diese müssen aber von zwei *Backslashes* eingeschlossen sein. Bei solchen Bezeichnern wird zwischen Groß- und Kleinschreibung unterschieden.

Variablen

Die Deklaration einer Variable wird über das Schlüsselwort `variable` eingeleitet, gefolgt vom Namen der Variable(n), einem Doppelpunkt und dem Typ und danach optional die Definition des Anfangswertes durch `:=`. Werden Variable bei der Deklaration nicht explizit initialisiert, so werden zu Beginn der Simulation skalare Datentypen mit dem niedrigsten darstellbaren Wert und Aufzählungstypen mit dem ersten Wert der Aufzählungsliste initialisiert. Die Zeile wird wie bei jeder VHDL Anweisung mit ; abgeschlossen.

```
variable identifier_list : type [range][:= initial_value];
```

Beispiele für Variablendefinition und Zuweisung:

```
variable Points    : integer;
variable MyPoints : integer := 0;
  ...
MyPoints := MyPoints + 1;  -- assignment
```

Damit mehrere Prozesse auf eine Variable gemeinsam zugreifen können, muss eine solche Variable mit dem Schlüsselwort `shared` deklariert werden. Gibt es in einer Architektur mehrere Prozesse, so muß jede architektur-globale Variable mit diesem Schlüsselwort versehen werden, da potentiell alle Prozesse auf diese globalen Variablen zugreifen können.

Konstanten

Die Deklaration einer Konstante wird über das Schlüsselwort `constant` eingeleitet, gefolgt vom Namen der Konstante, von einem Doppelpunkt und dem Typ und danach zwingend den Wert.

```
constant identifier : type [range] := value;
```

Konstanten sind nur initialisierbar, sie können nur gelesen aber nicht beschrieben werden.

```
constant HighYield : integer := 10;
```

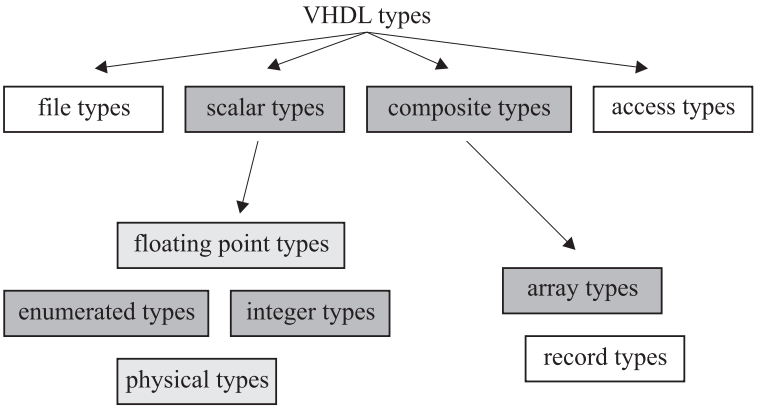


Abbildung 3: Typ-Klassifikation

2.4 Typen

VHDL ist eine stark typisierte Sprache, d.h. Konstanten, Variable und Signale haben einen festgelegten Typ. Bei der Codeanalyse wird die Konsistenz der Datentypen bei Operationen und Zuweisungen überprüft, gegebenenfalls müssen *Konvertierungsfunktionen* benutzt werden.

VHDL kennt einige vordefinierte Standardtypen. Darüber hinaus gibt es die Möglichkeit, Erweiterungen (definierte Typen, Funktionen, Prozeduren, Entitäten, Architekturen, ...) aus sogenannten Paketen (`package`) zu laden. Diese definieren zusätzliche Datentypen. Neben kommerziellen Paketen gibt es standardisierte Pakete der IEEE. Eine Übersicht über die Typklassifikation gibt Abb.3.

2.4.1 Vordefinierte Typen

Folgende skalare Typen sind vordefiniert:

Integer Der Definitionsraum von `integer` reicht von $-2^{31}-1$ bis $+2^{31}-1$. Die default-Darstellung ist dezimal, soll eine andere Zahlenbasis verwendet werden muss sie explizit angegeben werden. (binär: `2#...#`, oktal `8#...#` oder hexadezimal `16#...#`). Zu `integer` sind meist noch Untertypen definiert, wie `positive` (1..n) und `natural` (0..n).

Real Als `real` definierte Zahlen sind zwischen $-1.0E+38$ und $+1.0E+38$ zulässig.

Character Die `character` Literale entsprechen dem standard ASCII Zeichensatz, die darstellbaren Zeichen werden dabei in einfache Hochkommas eingeschlossen: `'0' 'A' '!'`.

Boolean Literale vom Typ `boolean` sind `true` und `false`.

String Ein `string` wird von doppelten Hochkommas umschlossen.

Bit Der Typ `bit` kennt als Literale die zwei logischen Werte `'0'` und `'1'`.

Time Ein Zeitausdruck vom Typ `time` wird in VHDL durch ein numerisches Literal und eine zugehörige Maßeinheit dargestellt. Als Maßeinheiten sind definiert: `fs, ps, ns, us, ms, sec, min, hr`. Beispiel: `12 ns, 100 sec, 5 us`

Aufzählungstypen

2.4.2 Typdefinition

Typen werden mit dem Schlüsselwort `type` oder `subtype` deklariert. `type` erzeugt einen neuen Typ, während `subtype` den Wertebereich eines bestehenden Typs einschränkt. Das hat Auswirkungen auf die Typkompatibilität. Typen sind grundsätzlich nicht kompatibel und müssen ineinander konvertiert werden. Bei vielen Datentypen versucht das der VHDL Simulator aufgrund des Code-Kontextes. Wo es nicht möglich ist, entsteht eine Fehlermeldung. Ein Subtyp ist hingegen zum Basistyp kompatibel, es entfällt die Konvertierung. Eine Variable des Basistyp wird allerdings beim Abbild auf den Subtyp überprüft, ob deren Wert in die Einschränkung paßt oder nicht.

```
type type_name is type_definition;
subtype subtype_name is type type_constraint;
```

Eine Enumeration ist eine Aufzählung von gültigen Bezeichnern. Diese Bezeichner können Zeichen-Literale sein (z.B.:'c') oder beliebige Namen (z.B.: low), wobei Schlüsselwörter als Namen nicht erlaubt sind. Wird derselbe Name in verschiedenen Enumerationen verwendet, so kann es zu Konflikten kommen. Der VHDL Simulator versucht aufgrund des CodeKontextes zu entscheiden, von welchem Enumerationstyp der Name stammt. Ist dies nicht möglich, gibt er einen Fehler aus. Durch explizite Angabe des Kontextes kann das verhindert werden.

```
type enum_name is (literal {, literal});
```

```
type aSignedWordTwoBytesLong is range -32768 to 32767;
type aProbability is range 0.0 to 1.0;
type aSeason is (spring, summer, autumn, winter);          -- enumeration
type ChaosOfElements is ('a', '4', a, help, strange, k6);

subtype aPeopleCounter is integer range 1 to 20;
subtype aLimitedProbability is aProbability range 0.0 to 0.5;
```

2.4.3 Felder (Arrays)

Feldtypen werden mit dem Schlüsselwort `array` erzeugt. Bei der Definition eines neuen Feldtyps gibt es zwei verschiedene Arten. Die eine, bei dem im Typ die Dimension (Ausdehnung) des Arrays angegeben ist, heißt *constrained array* Typ. Beim anderen Typ wird in der Definition keine Bereichsangabe gemacht. Dieser offene Feldtyp wird *unconstrained array* Typ genannt. Die Dimensionsangabe wird durch `range<>` ersetzt. Mehrdimensionale Felder können erzeugt werden, indem der Typ der Feldelemente wieder ein Feld ist. Der Elementzugriff erfolgt indem zwei von einander durch Beistrich getrennte Indizes verwendet werden.

```
type type_name is array (index_range) of element_type;
type type_name is array (enum_type) of element_type;
type type_name is array (index_type range <>) of element_type;
type type_name is array (index_type range index_range) of element_type

index_range ::= idx_bound ( to | downto ) idx_bound
```

```
type string is array (positive range <>) of character;
type MathVector is array (integer range <>) of real;
type bit_vector is array (integer range <>) of bit;
type instruction is (ADD, SUB, LDA, LDB, STA, STB, OUTA);

type AnAddress is range (0 to 1023);
type AWord is array (31 downto 0) of bit;
type AMemory is array (AnAddress) of AWord;
type AMatrix is array (1 to 2, 1 to 2) of real;
type AInstr_Flag is array (instruction) of bit;
```

Der Zugriff auf Elemente des Arrays kann direkt erfolgen, oder es können mehrere Elemente bei der Zuweisung zusammengefaßt werden (*Aggregate-Assignment*, *aggregate* = sammeln) . Dies erfolgt

durch den *gets*-Operator (=>).

```
aggregate ::= ( element_association { , element association } )
element_association ::= [ choices => ] expression
choices ::= choice { | choice }
choice ::= simple expression
         | discrete_range
         | element_simple_name
         | others
```

```
type ARegister is array (1 to 4) of integer;
variable XReg : ARegister := (0, 0, 0, 0);      -- initialize array
  ...
XReg := (1=>1453, 3=>789, 2=>354, 4=>0);        -- aggregate assignment
XReg := (4=>1111, others=>0);                   -- XReg = (0, 0, 0, 1111)
XReg := (others=>0);                            -- XReg = (0, 0, 0, 0)
XReg := (1|3=>0, others=>1);                    -- XReg = (0, 1, 0, 1)
```

Zuweisung von Feldern

Bei der Zuweisung von Feldern an ein anderes Feld, müssen die Elemente der Felder vom gleichen Typ sein und beide Felder müssen dieselbe Länge haben. Die Zuweisung erfolgt dann Element für Element, beginnend mit dem niedrigsten Index. Dabei spielt die Richtung des Indexbereiches (aufsteigend oder absteigend) keine Rolle. Bei beiden Feldern wird mit dem Element mit dem niedrigsten Index begonnen.

```
variable AppleBus, SpecialBus : bit_vector(31 downto 0);
variable SiemensBus : bit_vector(1 to 4);
variable KebaBus : bit_vector(3 downto 0);

SpecialBus := AppleBus;              -- assign whole array
SpecialBus(3) := AppleBus(10);       -- assign just one element
SiemensBus := KebaBus;               -- SiemensBus(1) := KebaBus(3) !!
                                     -- SiemensBus(2) := KebaBus(2) !!
                                     -- SiemensBus(3) := KebaBus(1) !!
                                     -- SiemensBus(4) := KebaBus(0) !!
```

Um aus einem längeren Feld einen kleineren Teil herauszunehmen kann eine spezielle Indizierung verwendet werden. Dabei wird der Bereich angegeben, der interessiert (*slice*) . Dies kann dazu benützt werden um ein Teil eines langen Feldes einem kürzeren Feld zuzuweisen.

```
variable AppleBus : bit_vector(31 downto 0);
variable CannonBus : bit_vector(15 downto 0);

CannonBus := AppleBus (30 downto 15);
```

Der *concatenation*Array-Operator bildet aus zwei eindimensionalen Feldern ein neues eindimensionales Feld des gleichen Typs. Das erzeugte Feld besteht aus den Elementen des linken Operands gefolgt von denen des Rechten.

```
variable ABus, SBus, XBus : bit_vector(3 downto 0);
variable A, B, C, D : bit;
variable BroadBus : bit_vector(7 downto 0);

XBus := A & B & C & D;                  -- XBus(3) := A
                                        -- XBus(2) := B
                                        -- XBus(1) := C
                                        -- XBus(0) := D
BroadBus := ABus & SBus;
XBus := ABus(3 downto 2) & SBus(1 downto 0);
```

2.4.4 Zusammengesetzte Typen (Records)

Record Typen werden mit dem Schlüsselwort `record` erzeugt. In ihnen lassen sich, anders als bei Feldern, Objekte verschiedenen Datentyps zusammenfassen. Dies funktioniert so, wie es auch in anderen Programmiersprachen üblich ist. Alle Elemente werden mit Namen und *subtype* deklariert, besitzen zwei Elemente denselben *subtype* können sie gemeinsam deklariert werden.

A	B	A and B	A nand B	A or B	A nor B	A xor B	A xnor B
0	0	0	1	0	1	0	1
0	1	0	1	1	0	1	0
1	0	0	1	1	0	1	0
1	1	1	0	1	0	0	1

Tabelle 2: Wahrheitstabelle für logische Operatoren

	Operator	Bedeutung
sll	shift left logical	leerer Platz wird mit '0' gefüllt
srl	shift right logical	leerer Platz wird mit '0' gefüllt
sla	shift left arithmetical	letztes Element vor leerem Platz wird kopiert
sra	shift right arithmetical	letztes Element vor leerem Platz wird kopiert
rol	rotate left	Elemente werden rotiert
ror	rotate right	Elemente werden rotiert

Tabelle 3: Schiebeoperatoren

```
type record_type_name is record
  element_name : element type;
  element_name : element type;
    . . .
end record record_type_name;
```

Zuweisungen zu einem Record können entweder durch einen Aggregat-Ausdruck oder durch die Zuweisung von Objekten an die einzelnen Elemente geschehen.

```
type RegName is (AX, BX, CX, DX);
type Operation is record
       Mnemonic       : String (1 to 10);
       OpCode         : Bit_Vector(3 downto 0);
       Op1, Op2, Res : RegName;
end record;

variable Instr1, Instr2, Instr3: Operation;

Instr1 := ("ADD_AX,_BX", "0001", AX, BX, AX);      -- aggregate assignment
Instr2 := ("ADD_AX,_BX", "0010", others => BX);

Instr3.Mnemonic := "MUL_AX,_BX";                   -- direct assignment
Instr3.Op1 := AX;
```

2.5 Operatoren

2.5.1 Logische Operatoren

Logische Operatoren werden auf Wahrheitswerte (true, false) in Variablen vom Typ `boolean`, auf Binärwerte in Variablen vom Typ `bit` oder auf Arrays dieser Typen angewendet. VHDL kennt dazu folgende Operatoren bei denen der Typ des Ergebnisses dem der Argumente entspricht. Werden 2 bit Werte verglichen ist das Ergebnis wieder ein bit Wert. Eine Auflistung der logischen Operatoren und eine zugehöroge Wahrheitstabelle zeigt Tab. 2. `true` entspricht binär '1' und `false` entspricht binär '0'. Beide Operanden müssen vom selben Typ sein.

Weiters gibt es noch bitweise Schiebeoperatoren, die als erstes Argument ein eindimensionales Feld von `bit`- oder `boolean`-Wert erwarten und als zweites Argument einen `integer`-Wert. Sie sind in der Spezifikation von VHDL-87 nicht enthalten, sondern erst in späteren Standards. Eine Übersicht über die verfügbaren Schiebeoperatoren gibt Tab. 3.

Folgende Beispiele verdeutlichen die Wirkung von Schiebeoperatoren:

```
variable Zm5 : BIT_VECTOR(3 downto 0) := ('1','0','1','1');
```

Operator	integer	real	Bedeutung
+,-	•	•	Addition, Subtraktion
,/	•	•	Multiplikation, Division
mod	•		positiver Divisionsrest
rem	•		Divisionsrest mit Vorzeichen von Divisor

Tabelle 4: Arithmetische Operatoren

```
Zm5 sll  1          -- ('0', '1', '1', '0')
Zm5 sll  3          -- ('1', '0', '0', '0')
Zm5 sll -3          -- Zm5 srl 3
Zm5 srl  1          -- ('0', '1', '0', '1')
Zm5 srl  3          -- ('0', '0', '0', '1')
Zm5 srl -3          -- Zm5 sll 3
Zm5 sla  1          -- ('0', '1', '1', '1')
Zm5 sla  3          -- ('1', '1', '1', '1')
Zm5 sla -3          -- Zm5 sra 3
Zm5 sra  1          -- ('1', '1', '0', '1')
Zm5 sra  3          -- ('1', '1', '1', '1')
Zm5 sra -3          -- Zm5 sla 3
Zm5 rol  1          -- ('0', '1', '1', '1')
Zm5 rol  3          -- ('1', '1', '0', '1')
Zm5 rol -3          -- Zm5 ror 3
Zm5 ror  1          -- ('1', '1', '0', '1')
Zm5 ror  3          -- ('0', '1', '1', '1')
Zm5 ror -3          -- Zm5 rol 3
```

2.5.2 Vergleichsoperatoren

Das Ergrbnis von Vergleichsoperatoren ist immer vom Typ `boolean`. Als Vergleichsoperatoren stehen zur Verfügung:

```
>       -- greater than
<       -- less than
>=      -- greater than or equal
<=      -- less than or equal
=       -- equal
/=      -- not equal
```

2.5.3 Arithmetische Operatoren

Arithmetische Operatoren sind nur auf numerische Datentypen definiert, Tab. 4 gibt einen Überblick über die arithmetischen Operatoren und ihre Bedeutung.

2.6 Ablaufstrukturen

2.6.1 IF-Statement

Das *If-Statement* ist ähnlich wie in anderen Programmiersprachen durch das Schlüsselwort `if` definiert. Bei Überschneidungen in den Bedingungen ist durch die Priorität der Bedingungen gewährleistet, dass nur ein Zweig von Anweisungen ausgeführt wird. Zuerst wird die Bedingung 1 geprüft und nur, wenn diese unwahr (false) ist, wird die nächste Bedingung geprüft. `elsif`-Zweige können beliebig oft definiert werden, ein `else`-Zweig kann, muss aber nicht vorkommen.

```
if condition then
   sequential_statements
{ elsif condition then
   sequential_statements }
[ else
   sequential_statements ]
end if;
```

2.6.2 CASE-Statement

Im Unterschied zum *If Statement* existieren beim *Case-Statement* keine Prioritäten der Bedingungen. Daher darf es zu keinen Überschneidungen kommen. Es müssen aber alle Fälle abgedeckt werden. Das Statement wird mit dem Schlüsselwort `case` eingeleitet, jeder Bedingung wird ein `when` vorangestellt.

```
case expression is
   when choice => sequential_statements | null
   { when choice => sequential_statements | null }
   [ when others => sequential_statements | null ]
end case;
```

`when others` wird ausgeführt, wenn keine andere Bedingung erfüllt ist. Dieser Zweig muss nicht vorhanden sein, es sei denn, alle `when`-Zweige decken nicht alle Fälle ab. Der `when others`-Zweig muss der letzte Zweig im Statement sein. Steht in einem Zweig `null` so wird in diesem Fall keine Aktion ausgeführt.

2.6.3 LOOP-Statement

Als Schleifenkonstrukt wird in VHDL das *Loop-Statement* verwendet. Dabei kommen drei verschiedene Arten des Statements zum Einsatz.

while loop

Die `while loop` Schleife prüft die Bedingung für die Ausführung der Anweisungen innerhalb der Schleife am Anfang ab. Solange diese Bedingung den Wahrheitswert `true` ergibt, wird die Schleife durchlaufen.

```
loop_label: while condition loop
   {  statement |
      exit [loop_label] [when condition]; |
      next [loop_label] [when condition]; }
end loop loop_label;
```

Eine `exit`-Anweisung innerhalb der Schleife führt dazu, dass diese Schleife sofort verlassen wird. Eine `next`-Anweisung veranlasst, daß die Schleife sofort, ohne der Ausführung der nachfolgenden Anweisungen, mit der nächsten Iteration beginnt.

for loop

Die `for loop` Schleife zählt eine Variable von einem Startwert bis zu einem Endwert. Solange der Endwert nicht überschritten ist, wird die Schleife durchlaufen. Der Wertebereich der Variable kann direkt oder durch einen Enumerationstyp angegeben werden. Der Schleifenparameter muss nicht deklariert werden, er wird implizit durch das Schleifenkonstrukt aufgrund des Typs der angegebenen Grenzen definiert und ist nur innerhalb der Schleife sichtbar. Der Schleifenparameter ist nur lesbar, also wie eine Konstante zu behandeln. Diese Schleife kann ebenfalls durch eine `exit`-Anweisung terminiert werden.

```
loop_label: for loop_parameter in range loop
   {  statement |
      exit [loop_label] [when condition]; |
      next [loop_label] [when condition]; }
end loop loop_label;
```

loop

Die `loop` Schleife prüft weder am Anfang noch am Ende eine Bedingung. Die Schleife wird endlos durchlaufen, wenn nicht an irgendeiner Stelle innerhalb der Schleife eine `exit` Anweisung steht.

```
loop_label: loop
   {  statement |
      exit [loop_label] [when condition]; |
      next [loop_label] [when condition]; }
end loop loop_label;
```

Beispiele

Hier folgen einige Beispiele zu Schleifenkonstrukten in VHDL:

```
subtype Range_Type is POSITIVE range 1 to 8; begin

WhileLoop: while i <= 8 loop
   Output_X(i) := Input_X(i+8);
   i := i + 1;
end loop WhileLoop;

ForLoop: for count_value in 1 to 8 loop
   Output_X(count_value) := Input_X(count_value + 8);
end loop ForLoop;

ForLoop2: for count_value in Range_Type loop
   Output_X(count_value) := Input_X(count_value +8);
end loop ForLoop2;

Loop_X: loop
   a_v := 0;
   Loop_Y: loop
      Next_1: next Loop_X when condition_1; -- next iteration of Loop_X
      Output_1(a_v) := Input_1(a_v);
      a_v := a_v + 1;
      Exit_1: exit when condition_2;        -- exit Loop_Y
   end loop Loop_Y;
   B(i) := Output_1(i);
   Exit_2: exit Loop_X when condition_3;    -- exit Loop_X
end loop Loop_X;
```

2.7 Assertion

Im Falle eines Fehlers ist es wünschenswert, den VHDL Simulator zu beenden und eine entsprechende Fehlermeldung auszugeben. Der Befehl `assert` stellt diese Funktion zur Verfügung. `assert` prüft eine Bedingung, ist diese Bedingung nicht erfüllt, so wird die angegebene Fehlermeldung ausgegeben. Es trat dann eine Verletzung der Bedingung auf (*assertion*). Dem Befehl kann mit `report` die Fehlermeldung angegeben werden und mit `severity` wird die Schwere dieses Fehlers angegeben.

```
assert condition
   [ report string ]
   [ severity severity_level ] ;

 report string
   [ severity severity_level ] ;
```

Wird kein *severity level* angegeben, nimmt der Simulator implizit den Level `ERROR` an. Als *severity level* sind folgende Werte definiert:

- `NOTE` Informationen werden an der Konsole des Simulators angezeigt.
- `WARNING` Eine Warnung wird an den Simulator weitergegeben.
- `ERROR` Eine Fortsetzung der Simulation ist nicht möglich.
- `FAILURE` Ein schwerer Fehler ist aufgetreten und die Simulation muss beendet werden.

Der `report` Befehl gibt, ähnlich wie der `assert` Befehl eine Meldung aus, allerdings unabhängig von einer Bedingung.

```
assert (ReadWrite = Read)          -- no writing on ROM
   report "Alf: Tried to write ROM!"
   severity warning;

if ReadWrite = Write then
   if MemoryType = RAM then
      -- write to memory
   else
      -- ROM is not writeable
      report "Alf: Tried to write ROM!"
         severity warning;
   end if
end if
```

2.8 Unterprogramme

Unterprogramme (Funktionen und Prozeduren) stellen eine wichtiges Werkzeug zur Abstraktion dar. Sie dienen zur Aufteilung der Gesamtaufgabe in kleinere Teillösungen. Jede Funktion und Prozedur hat eine Parameterliste (eventuell eine Leere). Die Parameter dieser Liste können durch die Schlüsselwörter `in`, `out` und `inout` zu Eingabe- oder Ausgabeparametern gemacht werden. Wird nichts angegeben, so wird `out` angenommen. `constant` bezeichnet einen konstanten Wert. Lokale Deklarationen werden bei jedem Aufruf neu evaluiert, d.h. lokale Variablen behalten ihren Wert nicht zwischen zwei Aufrufen.

Im Unterschied zu Prozessen sind Unterprogramme nur dann aktiv, wenn sie aufgerufen werden, nach Abarbeitung des Codes werden sie wieder inaktiv. Dagegen sind stets alle Prozesse gleichzeitig aktiv.

2.8.1 Funktionen

Eine *Funktion* ist ein Unterprogramm das entweder einen Algorithmus zur Berechnung von Werten oder eine Verhaltensbeschreibung darstellt. Funktionen sind Ausdrücke, die einen Wert eines bestimmten Typs zurückgeben. Dieser Wert kann skalar oder komplex sein und wird mit der `return`-Anweisung festgelegt. Eine Funktion wird mit dem Schlüsselwort `function` definiert. Der Funktionsaufruf stellt einen Ausdruck (expression) dar.

```
function function_name (parameters) return type is
   declarations
begin
   sequential statements
end function function_name;
```

Hier folgt ein Beispiel für die Deklaration und den Aufruf einer Funktion:

```
function Larger (Value1, Value2 : integer) return integer is
   variable Result : integer;
begin
   if Val1 > Val2 then
      Result := Val1;
```

```
    else
       Result := Val2;
    end if
    return Result;
end function Larger;

...

TheBetterChoice := Larger(YourAccount, MyAccount);
```

2.8.2 Prozedur

Eine *Prozedur* ist einer Funktion ähnlich, nur gibt sie keine Werte zurück, sie wird mit dem Schlüsselwort `procedure` definiert. Der Prozeduraufruf stellt eine Anweisung (statement) dar.

```
procedure procedure_name ( formal_parameter_list ) is
   procedure_declarations
begin
   sequential statements
end procedure procedure_name;
```

Hier folgt ein Beispiel für die Deklaration und den Aufruf einer Prozedur:

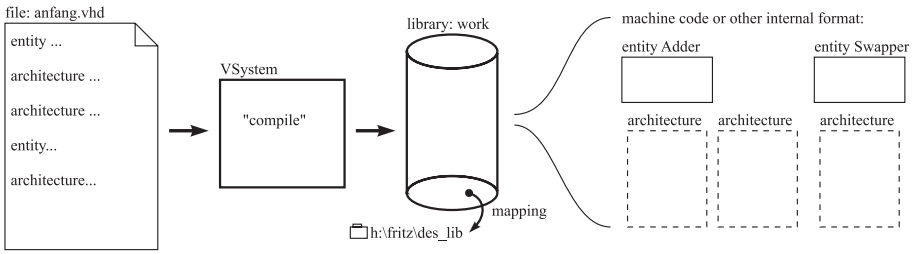
```
procedure Swap (variable A, B : inout integer;
                constant PermitSwap : boolean) is
   variable Temporal : integer;
begin
   if PermitSwap then
      Temporal := A;
      A := B;
      B := Temporal;
      return;
   end if;
end procedure Swap;

...

If YourAccount = Larger(YourAccount, MyAccount) then
   Swap (YourAccount, MyAccount, true);
end if;
```

3 Simulation

Ein Ziel von VHDL ist die Simulation einer Schaltung. Diese Simulation der bisher gezeigten Sprachelemente soll nun vorgestellt werden.

3.1 Simulationsvorbereitung

Vor der Simulation muß der Code analysiert und aufbereitet (compiliert) werden. Der Ablauf dabei ist folgender:



1. Die Datei mit dem VHDLCode wird geladen.
2. Der Code wird analysiert (compiliert).
3. In einer Bibliothek wird der analysierte Code abgelegt. Diese Bibliothek kann physisch im Dateisystem an eine bestimmte Stelle (Verzeichnis) gesetzt werden.
4. Aus anderen Bibliotheken werden die verwendeten Entitäten und Architekturen geladen und die Verknüpfungen kontrolliert.

3.2 Simulationsstart

Beim Start der Simulation wird folgender Ablauf durchgeführt:

1. Aktivierung der Deklarationen und Allokierung des Speicher. (*elaboration*)
2. Das Modell wird in den Anfangszustand gesetzt. (*initialization*)
3. Die Simulation beginnt (*simulation*)

3.3 Hardware

Charakteristisch für Hardware ist, daß immer alle Teile gleichzeitig aktiv sind. Es ist nicht möglich einzelne Teile auszuschalten und nacheinander ablaufen zu lassen. VHDL dient dazu, Hardware zu modellieren und benötigt deshalb ein Konzept, um diese Parallelität auf einem sequentiell arbeitenden Rechner zu simulieren. Diese hochgradige Parallelität wird durch die Prozesse realisiert. Bei der Simulation einer Schaltung stellen sich daher folgende Fragen:

1. Wie können Prozesse miteinander kommunizieren, und wie funktioniert der Datenaustausch zwischen Prozessen? Da die einzelnen Komponenten von Hardware miteinander verknüpft sind, muss es eine Möglichkeit geben, diese Verknüpfungen in VHDL zu modellieren.
2. Wie können parallele Prozesse auf einer sequentiellen Maschine simuliert werden? Single-Processor Maschinen können ja zu jedem Zeitpunkt nur einen Befehl ausführen, es bedarf also eines Mechanismus zur Modellierung der Parallelität.
3. Wie kann die Geschwindigkeit der simulierten Hardware auf einer Maschine untersucht werden, die um ein vielfaches langsamer ist? Auch digitale Hardware arbeitet nicht diskret sondern kontinuierlich.

Um diese Fragen beantworten zu können, ist ein (kurzer) Einblick in die Grundlagen der Simulationstechnik erforderlich.

3.4 Simulationszeit

Da der Ablauf der Simulation im Allgemeinen langsamer ist als die simulierte Schaltung selber ablaufen würde, bedient man sich einer virtuellen Simulationszeit. Simulieren heißt, dass ein Rechner alle Schritte so ausführt, als ob die simulierte Hardware das selber tun würde. Genauso wir in die echte Zeit engebettet sind, wird für ein Modell im Simulator eine eigene Zeit zugrunde gelegt.

Die Realzeit und die Simulationszeit laufen nicht im Gleichtakt ab. So kann die Simulation so rechenintesiv sein, daß in der Simulation z.B. erst 1 Sekunde vergangen ist, aber in der Realität bereits

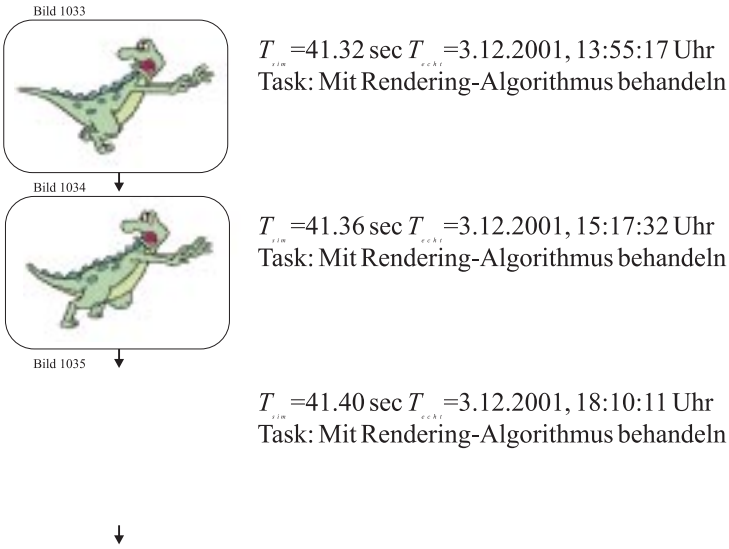


Abbildung 4: Simulationszeit

2 Stunden zur Berechnung benötigt wurden. Die Realzeit umgibt uns, die Simulationszeit umgibt das Modell im Simulator. Sie ist Ausdruck der Zeit, die eine solche existierende Schaltung benötigt würde. Um das nochmals zu verdeutlichen eine kleine Analogie: Die Herstellung eines Kinofilms benötigt mehrere Monate oder Jahre (Abb. 4). Im Kino selber dauert der Film dann nur wenige Stunden.

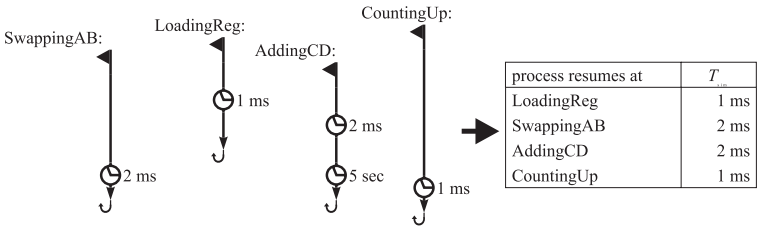
Alle Prozesse eines Modells werden parallel ausgeführt, jeder einzelne Prozess dabei sequentiell abgearbeitet. Alle werden mit der selben Simulationszeit ausgeführt. Üblicherweise ist innerhalb eines Prozesses eine `wait` Anweisung, An der der Prozess unterbrochen wird und wartet.

```
Boring: process is
   -- declare something
begin
   -- do something
   wait;
end process Boring;
```

In den meisten Fällen wird man aber zur Modellierung Prozesse verwenden, die nicht bis zum Ende der Simulation warten, sondern nach einem bestimmten Ereignis die Arbeit wieder aufnehmen. Dies könnte z.B. nach Ablauf einer gewissen Zeitspanne sein.

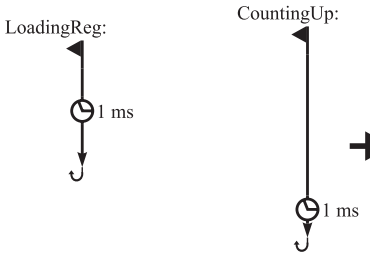
```
HaveMoreFun: process is
   -- declare something
begin
   -- do something
   wait for 14 ms;
end process HaveMoreFun;
```

Alle Prozesse werden – in beliebiger Reihenfolge – vom Simulator ausgeführt, bis dieser auf ein `wait`-Statement trifft. Die Abarbeitung des Prozesses wird ausgesetzt (*suspended*) und der Simulationszeitpunkt, zu dem die Abarbeitung wieder fortgesetzt werden soll, wird in einer Liste, der *Process Resume List* , eingetragen.



process resumes at	T_{sim}
LoadingReg	1 ms
SwappingAB	2 ms
AddingCD	2 ms
CountingUp	1 ms

Während die Abarbeitung von Anweisungen tatsächlich Zeit benötigt, ändert sich die Simulationszeit nicht. Wurden alle Prozesse bearbeitet und in der *Process Resume List* eingetragen, sucht der Simulator den Prozess aus der Liste, der am frühesten wieder aktiv wird. Die Simulationszeit wird nun fortgeschrieben.

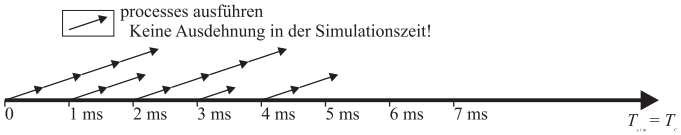


process resumes at	T_{sim}
LoadingReg	2 ms
SwappingAB	2 ms
AddingCD	2 ms
CountingUp	2 ms

Die Simulationszeit T_{sim} (in VHDL: T_c) wird nun auf 1 ms gesetzt, da zu diesem Zeitpunkt die Abarbeitung von Prozessen wieder aufgenommen werden muss. In diesem Fall werden zwei Prozesse erneut ausgeführt, die beiden anderen werden erst bei einer Simulationszeit $T_c = 2ms$ wieder aktiv. Nach Abarbeitung der beiden Prozesse, wird wieder berechnet, wann sie zum nächsten mal aktiv werde sollen, und dieser Wert wird wieder in die *Process Resume List* eingetragen.

In der neuen Liste sieht man, daß die Angaben der Wartezeit in absolute Zeiten der Ereignisse umgesetzt werden. Sequentielle Anweisungen in einem Prozeß verbrauchen keine Simulationszeit. Die Simulationszeit ändert sich nur durch zeitliche Ereignisse. Sequentielle Anweisungen benötigen daher nur Rechenzeit (Realzeit) geschehen aus Sicht der Simulation aber gleichzeitig. Es besteht nur ein kausaler Zusammenhang, kein temporaler.

Nach diesem Konzept wird die Simulation fortgeführt bis die *Process Resume List* leer ist, d.h. keine Prozesse mehr ausgeführt werden sollen, oder die maximale Simulationszeit T_{max} erreicht ist.



3.5 Kommunikation zwischen Prozessen

Wenn Prozesse das dynamische Verhalten der Bauteile darstellen, so ist es immens wichtig, dass diese Bauteile auch miteinander Daten austauschen können. Die Frage ist nur, wie dies geschehen kann. Dazu gibt es jetzt zwei Möglichkeiten, wobei wir uns an dieser Stelle auf eine beschränken, auf gemeinsame Variablen (*shared variables*).

Gemeinsame Variablen sind nur innerhalb einer Architektur möglich, da die Ein- und Ausgabe Ports automatisch Signale sind. Die Idee der gemeinsamen Variablen ist gleich der Idee globaler Variablen in anderen Programmiersprachen. In VHDL ist es aber nötig, solche Variablen noch extra mit `shared` zu Kennzeichnen. Auf diese gemeinsamen Variablen kann jeder Prozeß zugreifen. Dabei

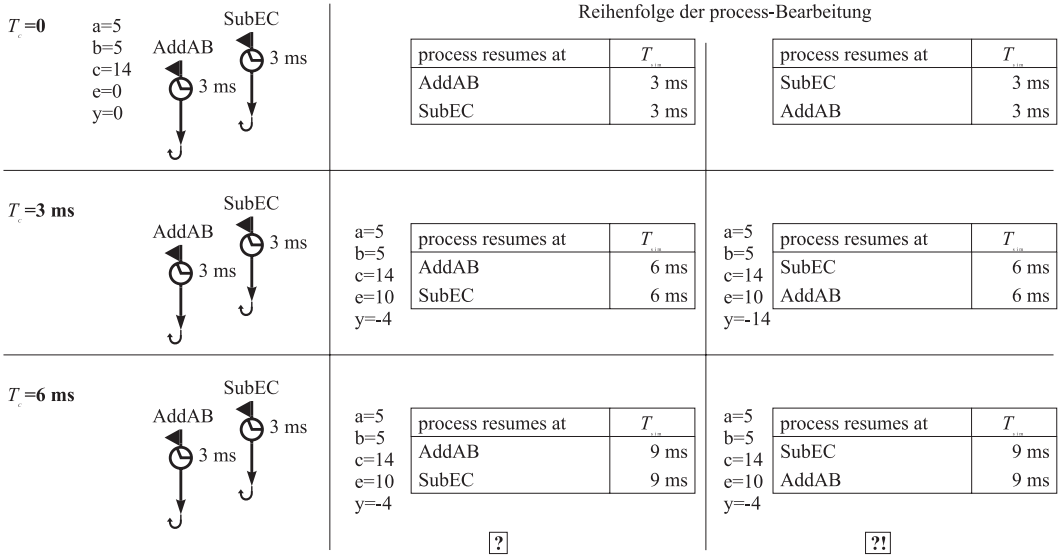
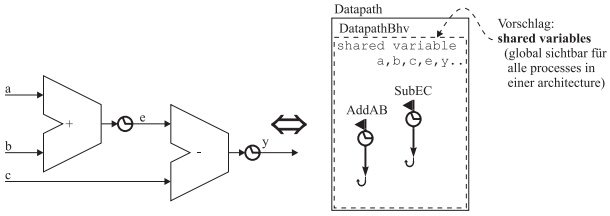


Abbildung 5: Schreibtischtest 1

kann es natürlich zu Problemen kommen.

```
entity Datapath is
end entity Datapath

architecture DatapathBhv of Datapath is
   shared variable a,b : integer := 5;
   shared variable c   : integer := 14;
   shared variable y   : integer := 0;
   shared variable e   : integer := 0;
begin
   AddAB: process is
   begin
      wait for 3 ms;
      e := a + b;
   end process AddAB;

   SubEC: process is
   begin
      wait for 3 ms;
      y := e - c;
   end process SubEC;
end architecture DatapathBhv;
```

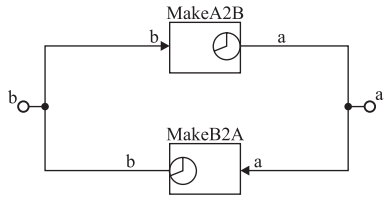


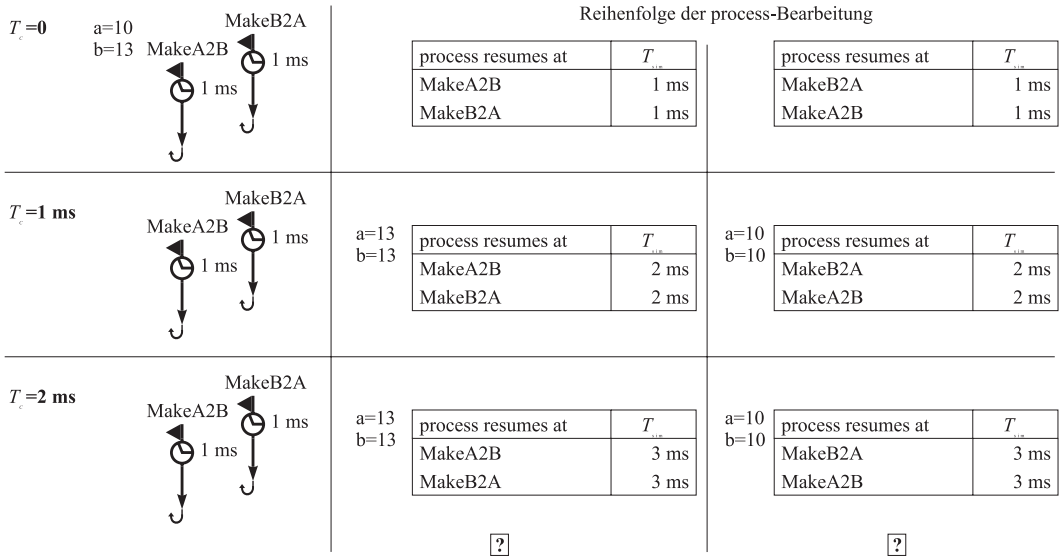
In diesem Beispiel werden zwei Prozesse definiert, die das Verhalten einer Harwarekomponente modellieren sollen.

Wenn man nun die ersten Simulationsschritte näher betrachtet (Abb. 5) stellt man fest, dass es offenbar nicht gleichgültig ist, welcher Prozess zuerst ausgeführt wird. Zur Simulationszeit $T_c = 3ms$ sieht man, dass in einem Fall das Ergebnis $y = -4$ und im anderen Fall $y = -14$ ist. Das widerspricht aber dem Sinn der Simulation, die Hardwarekomponente würde nicht deterministisch arbeiten, was aber in Wirklichkeit nicht passiert.

Noch deutlicher wird das bei der Hardwarekomponente Swapper.

```
entity Swapper is
end entity Swapper;
architecture SwapperBhv of Swapper is
   shared variable a : integer := 10;
   shared variable b : integer := 13;
begin
   MakeA2B: process is
   begin
      wait for 1 ms;
      a := b;
   end process MakeA2B;
   MakeB2A: process is
   begin
      wait for 1 ms;
      b := a;
   end process MakeB2A;
end architecture SwapperBhv;
```





Wieder zeigt sich, wenn man die ersten Simulationszyklen betrachtet ein unterschiedliches Verhalten, je nachdem welcher Prozess zuerst ausgeführt wird. Schon ab der Simulationszeit $T_c = 1ms$ sind die Werte beider Variablen gleich, abhängig davon welcher Prozess zuerst bearbeitet wurde.

Offensichtlich löst der Mechanismus der *shared variables* nicht das Problem der Kommunikation zwischen Prozessen, da das Ergebnis von der Reihenfolge der Prozessbearbeitung abhängt. Als Lösung würde sich eine andere Art von „Datenspeicher“ eignen, der nicht sofort während der Prozessbearbeitung neue Were annimmt, sondern diese Änderung der Werte nur plant und erst dann tatsächlich durchführt, wenn alle Prozesse bearbeitet wurden. Dieser Mechanismus zur Speicherung von Werten wird *signals* genannt.

4 Der Simulation Cycle, signals, processesund Attribute

Nach dem vorigen Kapitel wird klar, dass uns die Simulation von parallel arbeitender Hardware mit einer sequentiell arbeitenden Maschine vor ein nicht triviales Problem stellt.

4.1 Der Simulation Cycle

Wir schreiben einen einfachen Algorithmus zur Simulation nun explizit auf. Bisher sind wir implizit von der Existenz dieses Algorithmus ausgegangen, besonders im vorigen Kapitel. Nun ist es an der Zeit unsere implizite Annäherung zu konkretisieren.

Dieser Algorithmus beschreibt zunächst nur den eigentlichen Ablauf einer Simulation, ohne auf die Probleme der Parallelität einzugehen.

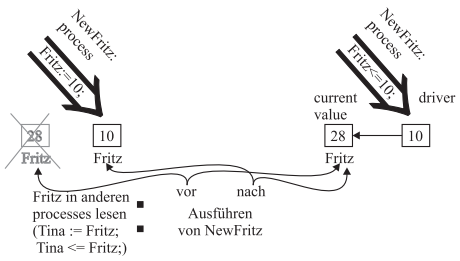
Initialisierung	passiert genau einmal, ähnlich der Codeausführung im Modulrumpf
$T_C := 0$	setzte $T_{current}$ auf 0 (aktuelle Zeit = 0)
alle default-Werte zuweisen	wie die Initialisierung von *variables* in anderen Programmiersprachen
alle *processes* bis *suspension* (wait...) ausführen	jeder *process* wird (in zufälliger Reihenfolge ausgewählt) ausgeführt bis er auf eine `wait`-Anweisung stößt
T_N ermitteln *(when will earliest* process *resume?)*	T_N = der kleinste Zeitpunkt, zudem ein *process* wieder aktiv wird (z.B. sei $T_C = 10ms$ und ein *process* wurde durch die Anweisung `wait for 25ms` *suspended*, so muss dieser *process* zum Zeitpunkt $T_C = 35ms$ wieder aktiv werden). T_N ist das Minimum der Aktivierungszeiten aller *processes*.
Simulation	Sie ist eine Schleife mit der Abbruchbedingung: aktuelle Simulationszeit = maximale Simulationszeit
Neue Simulationszeit setzen: $T_C := T_N$ (falls $T_C = T_{max(Simulator)}$ dann STOP!)	der Zeitpunkt T_N ist der kleinste, zudem ein *process* aktiviert werden muß
alle für T_C *resuming processes* in beliebiger Folge sequentiell ausführen	nur die *processes* ausführen, die tatsächlich aktiviert werden müssen (die bis zum jetzigen Zeitpunkt T_C durch `wait for` angehalten wurden)
T_N ermitteln *(when will earliest* process *resume?)*	wie bei der Initialisierung
Wiederhole Simulation	

4.2 Der Simulation Cyclemit driver(signals)

Die Kausalität der Wertzuweisungen der *variables* wird mit dem Simulation-Cycle aus Kapitel 4.1 nicht realisiert, d.h dass der Verlauf der Werte der *variables* nicht abgebildet wird. Ändert ein *process* den Wert einer *variable*, dann greifen die folgenden *processes* auf den nun neuen Wert zu, auch wenn diese *processes* zur gleichen Zeit T_C aktiviert wurden. Die Kausalität der Wertzuweisungen ist somit nicht erfüllt. Dies ist nur dann der Fall, wenn alle *processes* für ein bestimmtes T_C mit den gleichen, aktuell gültigen *variable*-Werten rechnen.

Ein neues **Kommunikationskonzept** mit folgenden Abläufen soll die Simulation der Parallelität realisieren und Kausalität garantieren:

Alle *processes* sequentiell berechnen, wobei die Ergebnisse nicht sofort zugewiesen werden, sondern lediglich **vorgemerkt** werden (in einem Berechnungsschritt greifen alle *processes* noch auf die alten Werte zu). Erst wenn alle parallelen *processes* sequentiell abgearbeitet wurden, werden alle **vorgemerkten** Werte tatsächlich zugewiesen. Dieses Vorgehen garantiert die Kausalität zum einen und macht zum anderen die Simulation auf sequentiellen Computern erst möglich.



Diese Art der Zuweisung wird durch *signals* realisiert. *signals*sind bildlich gesehen *variables*mit zeitlichem Gedächtnis. Das Merken selbst erfolgt im *driver* (auch *Plan*). Ein *driver*ist eine Liste, in der alle geplanten Änderungen von *signals*vorgemerkt werden (das Gedächtnis der *signals*). *signals*werden mit dem Schlüsselwort `signal` definiert. Eine Zuweisung an Signals wird mit `<=` durchgeführt.

```
signal identifier_list : type [range][:= initial_value];
```

Der Algorithmus *Simulation Cycle*mit *driver*zu diesem Vorgehen sieht folgendermaßen aus:

Initialisierung	
$T_C := 0$	
alle default-Werte für *variable*zuweisen und alle default-Werte für *signals*im *driver*eintragen	
Pläne für alle *signals*auswerten (Pläne nicht wegwerfen!)	der geplante Wert einer *variable*(im *driver*gespeichert) wird nun tatsächlich der *variable*zugewiesen. Alle künftigen Lese-Zugriffe verwenden dann bereits diesen neuen Wert. Der *driver*wird dabei **nicht** gelöscht!
alle *processes*bis *suspension* (wait...) ausführen	dabei Entstehen neue Werte für *variables*und neue *driver*für *signals*
T_N ermitteln *(when will earliest* process*resume?)*	
Simulation	
$T_C := T_N$ (falls $T_C = T_{max(Simulator)}$ dann STOP!)	der Zeitpunkt T_N ist der kleinste, zudem ein *process*aktiviert werden muß
*driver*für alle *signals*auswerten (Pläne nicht wegwerfen!)	wie bei der Initialisierung
alle für T_C *resuming processes*in beliebiger Folge sequentiell ausführen	es entstehen neue Werte für *variables*und neue *driver*für *signals*
T_N ermitteln *(when will earliest* process*resume?)*	
Wiederhole Simulation	

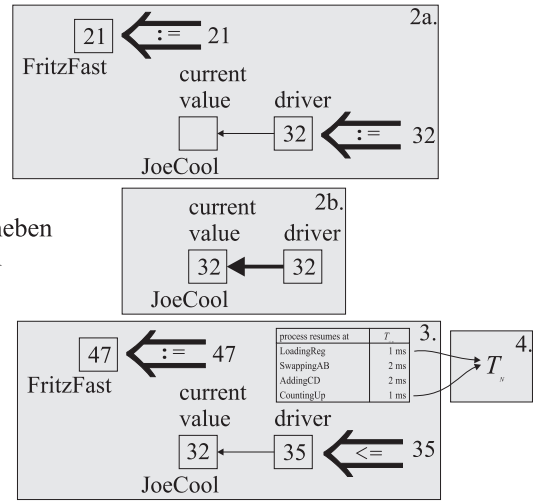
Beispiel

Hierbei wird veranschaulicht was ein *driver*eigentlich ist und welchen Zweck er hat. Die Abarbeitung des Beispieles erfolgt Schritt für Schritt nach dem *Simulation Cycle*mit *driver*. Insbesondere soll dabei veranschaulicht werden, dass bei einer Werte-Zuweisung auf ein *signal*nicht sofort dessen Wert verändert wird, sondern zunächst diese Veränderung nur im *driver*gespeichert wird, was die Kausalität

der *signal*-Werte für alle *processes*(parallel aktiv!) garantiert.

```
variable FritzFast: integer := 21;
signal JoeCool: integer := 32;
```

1. $T_c := 0$;
2a. alle default-Werte
```
variable FritzFast
    : integer := 21;
signal JoeCool
    : integer := 32;
```
für variables: direkt zuweisen,
für signals: als Plan (driver) aufheben
2b. Pläne für alle signals auswerten
(Plan nicht wegwerfen!)
3. Alle processes in beliebiger Reihenfolge ausführen:
Es entstehen neue Werte für variables und driver-Inhalte für signals.
4. T_N ermitteln (when will earliest process resume?)



Schritt 1 und 2a dem *signal* `JoeCool` wird 32 nicht direkt zugewiesen, sondern der Wert 32 wird in den *driver*geschrieben.

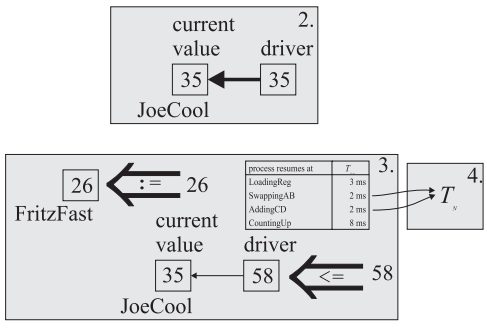
Schritt 2b erst jetzt erfolgt die Zuweisung (`JoeCool` hat nun den Wert 32)

neue Zuweisung `FritzFast := 47; JoeCool <= 35;`

Schritt 3 und 4 `JoeCool` hat noch den Wert 32! (35 im *driver*)

cycle

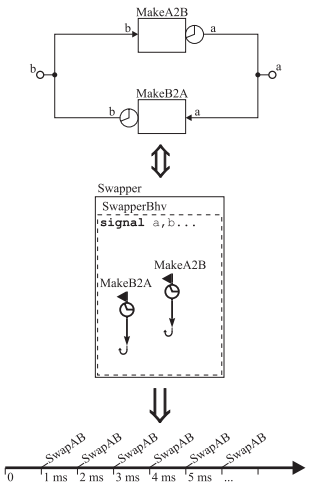
1. $T_c := T_N$
($T_c = T_{max(Simulation)}$? ⇒ STOP!)
2. Driver für alle signals auswerten
(Plan nicht wegwerfen!)
3. Alle für T_c resuming processes in beliebiger Reihenfolge ausführen:
Es entstehen neue Werte für variables und driver-Inhalte für signals.
4. T_N ermitteln (when will earliest process resume?)



Schritt 2 `JoeCool` hat nun den Wert 35 (der *driver*ebenfalls!)

Beispiel Swapper

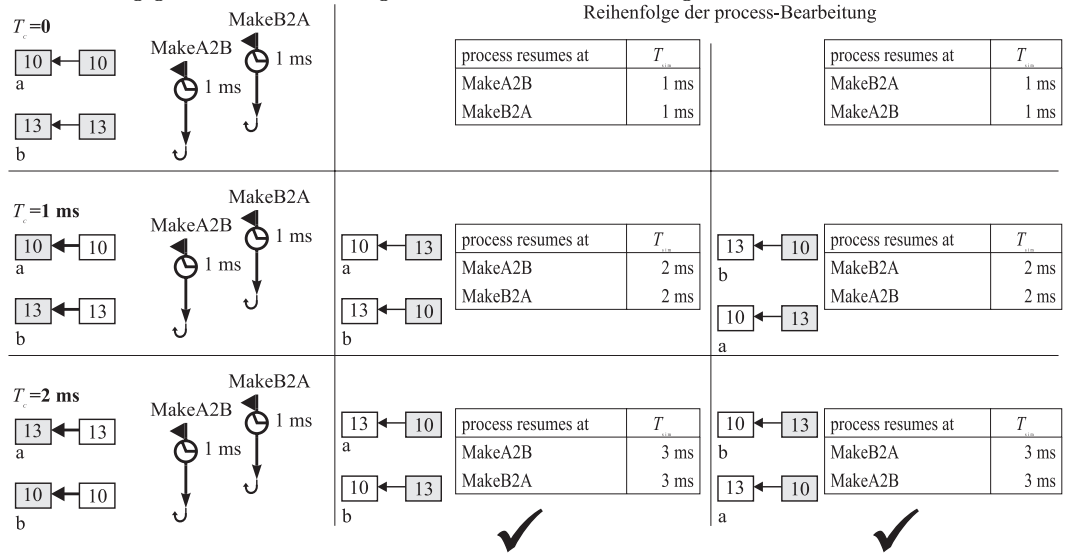
```
entity Swapper is end entity Swapper;

architecture SwapperBhv of Swapper is
  signal a : integer := 10;
  signal b : integer := 13;
begin
  MakeA2B: process is
  begin
    wait for 1 ms;
    a <= b;
  end process MakeA2B;
  MakeB2A: process is
  begin
    wait for 1 ms;
    b <= a;
  end process MakeB2A;
end architecture SwapperBhv;
```



Initialisierung	
$T_C := 0$	
Zuweisungen und Pläne auswerten	`Signal a : integer := 10;` im *driver* von `a` steht 10 und `a` hat den Wert 10, `Signal b : integer := 13;` im *driver* von `b` steht 13 und `b` hat den Wert 13
alle *processes* bis *suspension* (wait...) ausführen	(*process* MakeB2A zufällig ausgewählt) ***process*-Eintritt:** `MakeB2A` **Anweisung:** `wait for 1ms` $\Rightarrow$ Eintrag in process-*resume-list:* "`MakeB2A` - 1ms" (`MakeB2A` wird zur Zeit $T_C = 1ms$: wieder aktiv). **Anweisung:** `b` $\Leftarrow$ `a;` Im *driver* von `b` wird der Wert von `a` (10) eingetragen. ***process*-Eintritt:** `MakeA2B` **Anweisung:** `wait for 1ms` $\Rightarrow$ Eintrag in process-*resume-list:* "`MakeA2B` - 1ms" (`MakeA2B` wird zur Zeit $T_C = 1ms$: wieder aktiv). **Anweisung:** `a` $\Leftarrow$ `b;` Im *driver* von `a` wird der Wert von `b` (13) eingetragen.
T_N ermitteln *(when will earliest* process*resume?)*	aus der *process-resume-list:* den kleinsten Zeiteintrag (1ms) suchen.

Simulation	
$T_C := T_N$	1 ms
*driver*für alle *signals*auswerten	`a` erhält den Wert seines *drivers*(13) `b` erhält den Wert seines *drivers*(10) (alle *driver*bleiben erhalten!)
alle für T_C *resuming processes*in beliebiger Folge sequentiell ausführen	`MakeA2B` und `MakeB2A` haben den Eintrag 1ms in der *process-resume-list* ⇒ Abarbeiten von `MakeA2B` und `MakeB2A` in beliebiger Reihenfolge (*process*`MakeB2A` zufällig ausgewählt) ***process*-Eintritt:** `MakeB2A` **Anweisung:** `wait for 1ms` ⇒ Eintrag in *process-resume-list:* "`MakeB2A` - 2ms" (`MakeB2A` wird zur Zeit $T_C = 2ms$: wieder aktiv). **Anweisung:** `b` ⇐ `a`; Im *driver*von `b` wird der Wert von `a` (10) eingetragen. ***process*-Eintritt:** `MakeA2B` **Anweisung:** `wait for 1ms` ⇒ Eintrag in *process-resume-list:* "`MakeA2B` - 2ms" (`MakeA2B` wird zur Zeit $T_C = 2ms$: wieder aktiv). **Anweisung:** `a` ⇐ `b`; Im *driver*von `a` wird der Wert von `b` (13) eingetragen.
T_N ermitteln *(when will earliest* process*resume?)*	Aus der *process-resume-list:* den kleinsten Zeiteintrag suchen.
Wiederhole Simulation	

Das gewünschte Ergebnis liegt vor: Die Werte der beiden *signals*wurden vertauscht und das Ergebnis ist unabhängig von der Reihenfolge der *process*-Abarbeitung. Hier eine anschauliche Darstellung:



Wie aus diesem Beispiel ersichtlich liefert der *Simulation Cycle*mit *driver*korrekte Endergebnisse im Sinne einer Hardware-Simulation. Unberücksichtigt blieb aber bisher das zeitliche Verhalten von Hardware, d.h. die Frage: Wann liegen welche Ergebnisse an?

4.3 Hardware-Simulation

Zunächst ein paar physikalische Tatsachen, die bei der Simulation zu beachten sind:

1. Hardware ist immer aktiv:
   Liegt Spannung an Hardware an, so reagiert sie sofort auf alle Änderungen. Der Vergleich mit einem Rohrnetz unter Wasserdruck soll dies veranschaulichen: Öffnet man einen Hahn, so strömt unmittelbar Wasser heraus und nicht erst dann, wenn man hinsieht. Das "hinsehen" entspricht

einem Prozeduraufruf (`IF` *LookWaterFlow* `THEN` ...), bei dem man erst zum Zeitpunkt des Aufrufes feststellen kann, ob sich ein Zustand geändert hat.

Aus diesem Grund werden auch in der Softwareentwicklung alle zeitkritischen Reaktionen auf äußere Einflüsse über *Interrupt*-Routinen realisiert. Sie sprechen auf äußere Einflüsse (*Interrupts*) sofort an (der momentane Programmablauf wird unterbrochen). Auch Software kann somit in gewissen Situationen "immer aktiv" erscheinen.

Alle *event*-basierten Softwaresysteme arbeiten nach dem gleichen Prinzip: Tritt ein bestimmtes Ereignis (z.B. `Click`) auf so wird die entsprechende Methode (z.B `edtTextClick`) des aktuell fokussierten Objektes (`edtText`) aufgerufen. Eine wichtige Anforderung an Hardwaresimulatoren ist eine genaue Abbildung dieses Verhaltens.

2. Hardware arbeitet parallel
   Befinden sich in einem Wasserrohrnetz zwei Wasserhähne, so reagieren beide sofort auf äußere Einflüsse (aufdrehen). Hardware, vorausgesetzt mit der nötigen Spannung und dem nötigen Strom versorgt reagiert ebenso sofort und parallel.

   Ein *event*-basiertes Softwaresystem hingegen arbeitet immer nur ein Ereignis nach dem anderen ab. Dabei kann es zu folgender Situation kommen:
   Ein `Ereignis2` tritt eine kurze Zeitdauer t nach `Ereignis1` auf. `Ereignis1` wird demnach zuerst behandelt, wobei eine `Methode1` aufgerufen und abgearbeitet wird. Dauert diese Abarbeitung nun länger als t (Eintritt von `Ereignis2`), so kann `Ereignis2` nicht unmittelbar behandelt werden, sondern die zu `Ereignis2` korrespondierende `Methode2` wird erst nach Beendigung der aktiven `Methode1` behandelt.

   Diese Tatsache erhält dann relevante Bedeutung, wenn `Methode2` Werte verändert, die auch `Methode1` verwendet, da offensichtlich `Methode1` immer nur die Werte verwendet, welche vor bzw. bei ihrem Aufruf aktuell waren. Im Wasserrohr-Beispiel könnte demnach ein Wasserhahn noch lange nach dem Abdrehen der Zuleitung laufen!

   Vereinfacht gesagt kann aus dem selben Grund die Simulation mit *Interrupt*-Routinen nicht mehr korrekte Ergebnisse liefern, wenn der verwendete Computer weniger Prozessoren hat als *processes*parallel simuliert werden sollen. Im allgemeinen hat unser Simulationscomputer weniger Prozessoren, als zu simulierende *processes*.

   Das Konzept des *Simulation Cycle*mit *driver*simuliert hingegen beliebig viele *processes*mit nur einem Prozessor!

3. Hardware verbraucht Zeit
   Im 1. Punkt (Hardware ist immer aktiv) hatten wir den Anschein erweckt, dass eine Reaktion auf Änderungen sofort (ohne Zeitverzögerung) eintritt. Diese Reaktion tritt in Wirklichkeit erst nach einer gewissen Zeitverzögerung auf

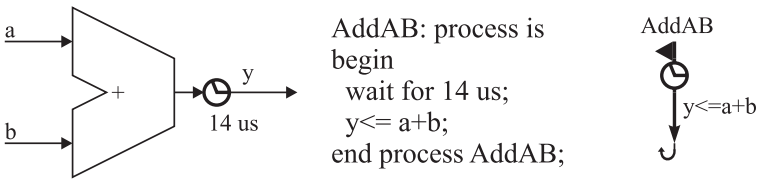
   Ändert man die Eingänge (Spannung) eines Gatters (z.B. `AND, NAND, ...`), so ändern sich aufgrund physikalischer Gesetzte die Ausgänge nicht zum gleichen Zeitpunkt, sondern etwas verzögert. Die Zeitdauer dieser Verzögerung hängt von der Bauart der betrachteten Bausteine ab und wird vom Hersteller ermittelt und angegeben.

Die Punkte 1) und 2) werden bereits vom *driver*-Konzept realisiert. Die Simulation der unter Punkt 3) beschriebenen **Verzögerung** gelingt mit den bisherigen Konzepten nicht, was eine Revision des *Simulation Cycle*zur Folge hat.

4.3.1 Simulation mit `wait for`

```
wait for time;
```

Am Beispiel eines Addierers wird ein erster Entwurf für die Simulation der zeitliche Verzögerung mit Hilfe von `wait for` demonstriert:



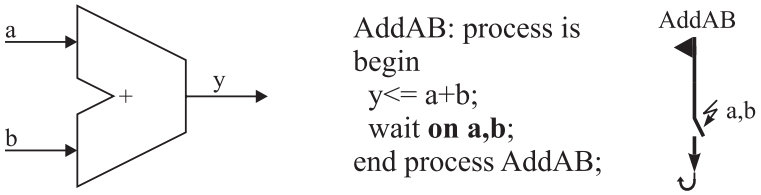
Dem *signal* `y` wird der Wert von `(a+b)` erst nach $14\mu s$ zugewiesen. Diese Zeitdauer entspricht der Verzögerung der realen Hardware für diese Berechnung. Der zeitliche Faktor kann also mit `wait for` simuliert werden. Wunderbar, alles in Ordnung – noch nicht ganz, denn ändert sich `a` oder `b`, dann sollte das neue Ergebnis $14\mu s$ nach dieser Änderung dem *signal* `y` zugewiesen werden. Tatsächlich aber wartet der *process* `AddAB` einfach $14\mu s$ nach Aktivierung, ändert dann gegebenenfalls das *signal* `y` und wartet wieder $14\mu s$ usw. völlig unabhängig von einer möglichen Änderung von `a` oder `b`.

4.3.2 Simulation mit `wait on`

Da der *process*immer dann ausgeführt werden soll, wenn sich `a` bzw. `b` verändert, versuchen wir einen Entwurf mit `wait on a,b;`.

```
wait on aSignal {, aSignal};
```

Der aktive *process*hält so lange an, bis sich entweder das *signal* `a` oder das *signal* `b` seinen Wert ändert, oder beide sich ändern.



Bei diesem Entwurf erfolgt eine neue Berechnung von `y` immer dann, wenn sich `a` oder `b` verändert haben. Die zeitliche Verzögerung von $14\mu s$ (für die Berechnung selbst) muß natürlich noch eingebaut werden. Dies könnte wie im Entwurf 1 geschehen (durch `wait for` $14\mu s$ vor der Anweisung "$y <= a + b$;"). Warum muß dieser Versuch scheitern?

Nehmen wir an, dass sich `a` bzw. `b` bereits innerhalb der $14\mu s$ (wait for) erneut verändert, so würde das soeben entworfene Modell unbeeindruckt davon die angefangenen $14\mu s$ abwarten, und dann den Wert `a+b` dem *signal* `y` zuweisen. Wenn diese Änderung von `a` bzw. `b` nach $10\mu s$ (gerechnet ab dem Beginn von `wait for` $14\mu s$) stattgefunden hat, so würde das neue Ergebnis bei diesem Modell nach $4\mu s$ dem *signal* `y` zugewiesen werden, was der realen Verzögerung der Hardware widerspricht, da die Berechnung $14\mu s$ (physikalische Gesetzmäßigkeit) dauert! Ein weiterer Entwurf (diesmal der letzte) soll das gewünschte Verhalten bringen.

4.4 Verzögernde signals

Wie die beiden letzten Entwurfs-Versuche mit bekanntem Wissen zeigten, ist eine zeitliche-Simulation mit den bisher vorgestellten Simulationsmechanismen nicht realisierbar.

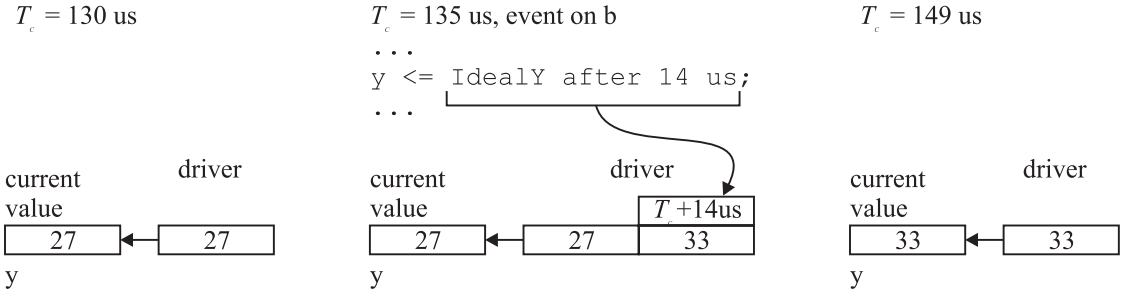
Die zeitliche Verzögerung von Hardware bedeutet in der uns bekannten Simulationsumgebung

nichts anderes als dass sich *signals*nicht sofort nach Zuweisung, sondern erst nach Ablauf einer bestimmten Zeitdauer ändern dürfen.

Die Zeitdauer die Hardware benötigt um eine Berechnung durchzuführen, d.h. vom Beginn einer Änderung eines Eingangswertes bis zum Zeitpunkt zu dem der neue Wert am Ausgang anliegt heißt *delay* und wird mit t_{delay} bezeichnet.
Die Simulation von t_{delay} in VHDL erfolgt mit **verzögernden-Signals** . Da im *driver*bereits für jedes *signal*dessen zukünftige Änderung gespeichert wird, liegt die Idee nahe auch dort (im *driver*) die Verzögerungszeit einzubauen: Für jeden *driver*-Eintrag wird zusätzlich zum Wert auch der Zeitpunkt der zukünftigen Wertänderung mitgespeichert.

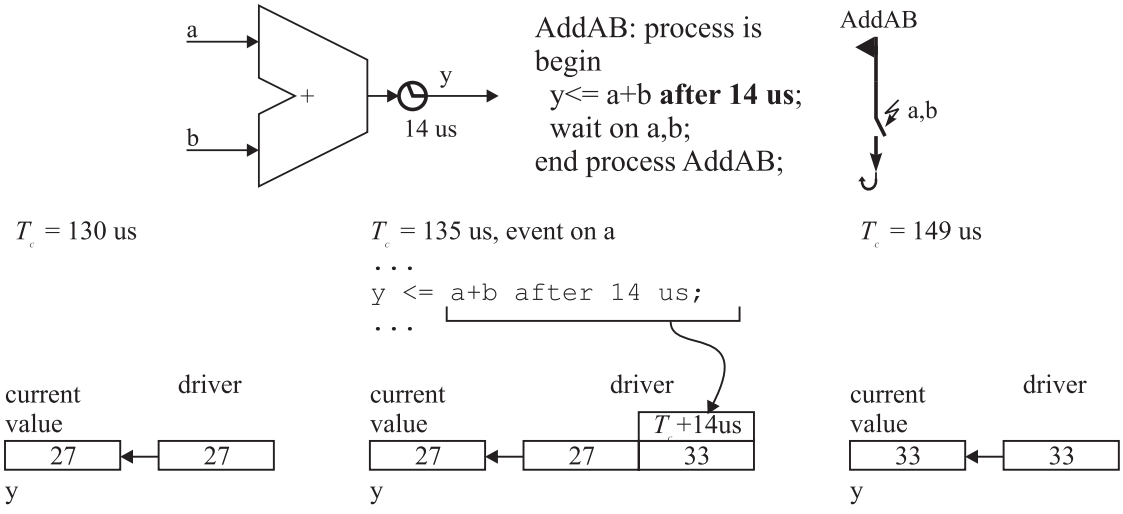
```
aSignal <= aValue after aTime;
```

Der Wert von T_C (aktuelle Simulationszeit) wird im Simulator gespeichert und ist bekannt. T_C ist immer die Zeitbasis für jede Wertänderung (z.B. bedeutet `y <= 10 after 14s`: `y` soll den Wert 10 erhalten zum Zeitpunkt $T_C(jetzt) + 14s$)



Beispiel Addierer

Ein einfacher Addierer, bei dem sich der Ausgang (`y`) aus der Summe der beiden Eingängen (`a, b`) berechnet, soll mit gegebener **Verzögerung** simuliert werden.

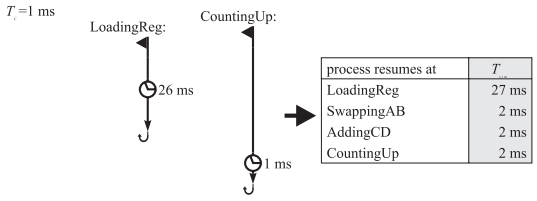


4.5 Zeitfortschritt in der Simulation

Beim *Simulation Cycle*mit *driver*ist im Schritt 4 T_N (*when will earliest* process*resume?*) zu ermitteln, dies und die Einführung der *verzögernden-signals* stehen im engen Zusammenhang. Im Folgenden wird erklärt welche Schritte in der Simulation ablaufen, wenn ein process angehalten (*suspended*) wurde (Ende von Schritt 3: alle für T_C *resuming processes*bis *suspension* in beliebiger Reihenfolge ausführen).

Immer wenn ein *process*angehalten wird, erfolgt ein Eintrag in der *process resume list* mit folgender Unterscheidung:

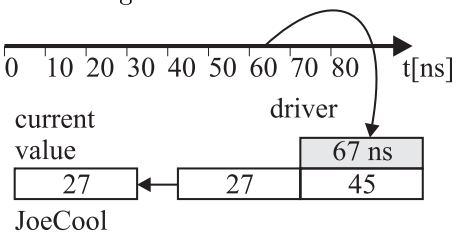
1. *process*durch `wait for...` angehalten
   Sei $T_C = 0$ und wird der *process* `SwappingAB` durch die Anweisung `wait for 2ms` *suspended*, so erfolgt ein Eintrag (`SwappingAB` – 2ms) in der *process-resume list*:



2. *process*durch `wait on...` angehalten
   Wurde ein *process*durch die Anweisung "`wait on...`" *suspended*, so wird er von einer *signal*-Änderung wieder wieder aktiviert (diese Änderung kann durchaus von einem anderen *process*verursacht werden).

   Dabei ist Folgendes zu berücksichtigen:

   - Der Zeitpunkt hängt vom nächsten *event* auf einem *signal*ab
   - *signal*wird durch Zeitsteuerung im *driver*aktiviert



   - aktive *signals*

   Ändert ein *signal* `s` seinen Wert, so erfolgt seine Eintragung in die *signal-event list*. Für alle *processes*, die durch `wait on s` *resumed* wurden wird ein Eintrag in der *process-resume list*mit dem Zeitpunkt, zu dem das *signal*`s` seinen Wert ändert getätigt. Dadurch wird der *suspended process*zum richtigen Zeitpunkt wieder aktiv, nämlich genau dann, wenn das *signal*`s` seinen Wert ändert.
   Beispiel:

```
AddAB: process is
begin
```

```
    y <= a + b after 14 sec
    wait on a,b;
end process AddAB;
...
a <= 10 after 14 sec
...
```

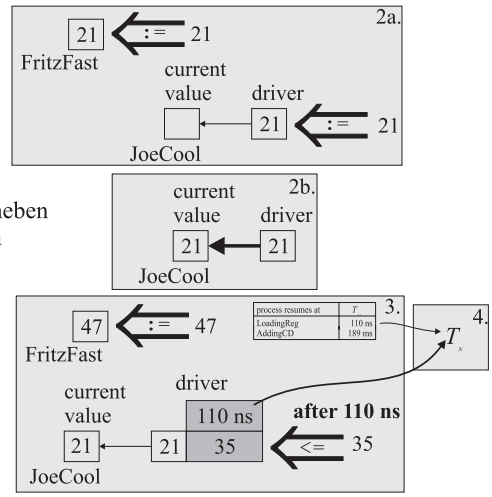
⇒: Eintrag im *driver* von `a` mit dem Zeitwert $T_C + 14s$ (obere Zeile) und dem Wert 10 (untere Zeile)
und Eintrag von `a` in die *signal-event list* ⇒ Eintrag von `AddAB` in die *process-resume list* mit dem Zeiteintrag $T_C + 14s$.

Ein detailiertes Beispiel mit Schreibtischtest wird im Kapitel 4.9, Seite 36 gezeigt. Zunächst wird der revidierte *Simulation Cycle* und der neu *Simulation Cycle* behandelt (dabei erfolgt ein Exkurs zu *wait* und zu den *Attributes*.

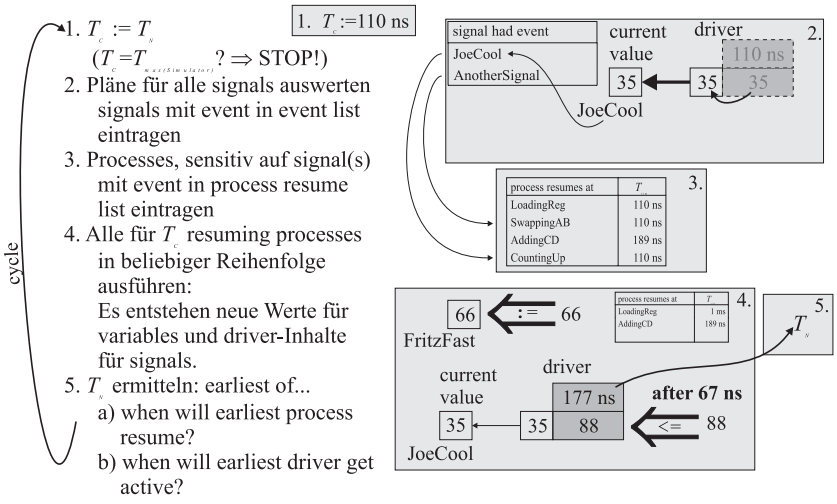
4.6 Revision des Simulation Cycle

Durch die Einführung der *verzögernden signals* muß der *Simulation Cycle* überarbeitet werden. Eintragungen in die *signal-event list* und in die *process-resume list* werden nun explizit aufgeführt, im wesentlichen aber läuft der Algorithmus wie gewohnt ab:

- Initialization:
  1. $T_c := 0$;
  2a. alle default-Werte
  ```
  variable FritzFast
       : integer := 21;
  signal JoeCool
       : integer := 21;
  ```
  für variables: direkt zuweisen,
  für signals: als Plan (driver) aufheben
  2b. Pläne für alle signals auswerten (Plan nicht wegwerfen!)
  3. Alle processes in beliebiger Reihenfolge ausführen: Es entstehen neue Werte für variables und driver-Inhalte für signals.
  4. T_n ermitteln (when will earliest process resume?)



- Simulation:



4.7 Alles über `wait`

Eine wichtige Gruppe von VHDL-Anweisungen sind die `wait` Anweisungen, von denen einige bereits in den vergangenen Beispielen vorgestellt wurden.

```
wait;
wait on signal_list;
wait until condition;
wait for time;
```

Weitere wichtige `wait`-Konstrukte:

- `wait`
- `wait for ...`
- `wait on ...`
- `wait on Fred, Alex for 10 sec;`
  (warte auf *event* von `Fred` oder `Alex`, aber maximal für 10 Sekunden)
- `wait on Fred, Alex until Fred > 30;`
  (warte auf *event* von `Fred` oder `Alex`, und zusätzlich `Fred > 30` wahr ist)
- `wait until Alex = 30;`
  (entspricht `wait on Alex until Alex = 30`)
- `wait on Fred, Alex until Alex = 30 for 10 sec;`
  (warte auf *event* von `Fred` oder `Alex`, und zugleich `Alex` = 30 wahr ist, für maximal 10 Sekunden)
- `wait until Alex > 30 for 10 sec;`
- ...

4.8 Attribute

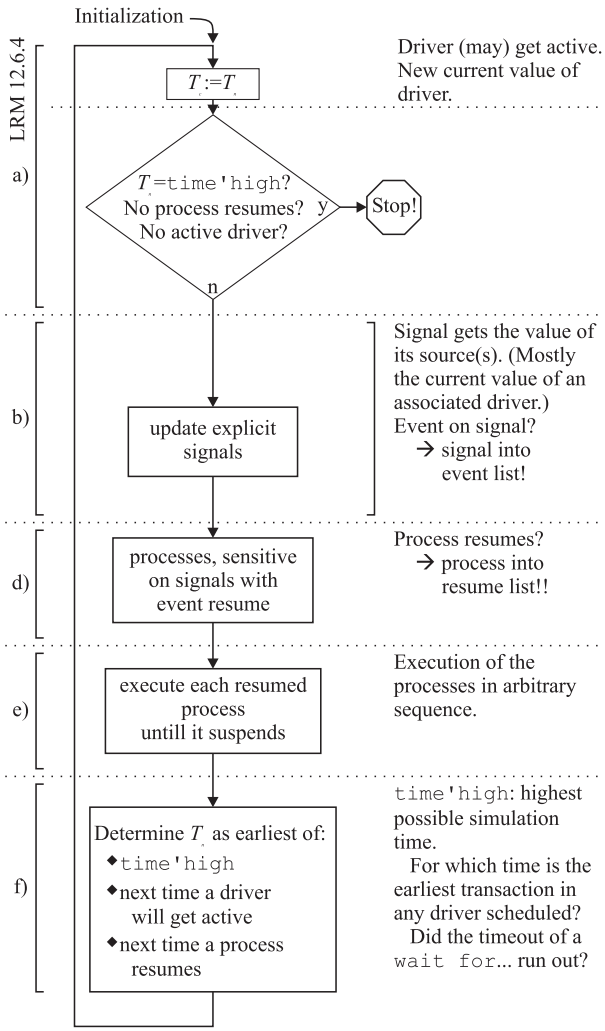
Attribute bieten bisher unerwähnte Möglichkeiten der VHDL-Programmierung. Sie ermöglichen eine detaillierte Bezugnahme auf den, dem *tick* (') vorangestellten Bezeichner und realisieren so eine Vielfalt von einfachen, sehr effizienten Zugriffen.

```
signal BinNr : bit_vector (31 downto 0);
signal Flag : bit;

BinNr'left =   31            BinNr'right =  0 -- left, right bound
BinNr'low  =    0            BinNr'high  = 31 -- lower, upper bound
BinNr'range =  31 downto 0                    -- range of array
BinNr'lenght = 32                             -- length of array

                  -- boolean functions:
Flag'event        -- event occurred in current simulation cycle?
Flag'stable(T)    -- no change in the last T time units?
Flag'active       -- transaction occurred in current simulation cycle
Flag'delayed(T)   -- signal Flag delayed for T time units

Flag'last_event   -- time, since the last event occurred
Flag'last_active  -- time, since the last transaction occurred
Flag'last_value   -- value before the last event
```

```
type bit_vector is array (integer range <>) of bit;
variable BinNr : bit_vector(31 downto 0);
variable CalcResult : bit_vector(13 downto 0);

function NrOfOnes (ToCountOn : bit_vector)
    return integer is variable TheNrOfOnes
integer := 0;
begin
    for Index in ToCountOn'range loop
        if ToCountOn(Index) = '1' then
            TheNrOfOnes := TheNrOfOnes + 1;
        end if;
    end loop;
    return TheNrOfOnes;
end function NrOfOnes;

WhatUWant := NrOfOnes(BinNr);
WhatUWant2 := NrOfOnes(CalcResult);
```

4.9 Der neue Simulation Cycle

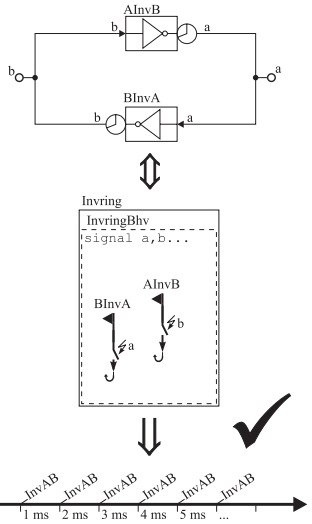
Die Einführung der *Attribute*, der *signal*-event list und der *process-resume list*führt zu einer neun Formulierung des *Simulation Cycle*:

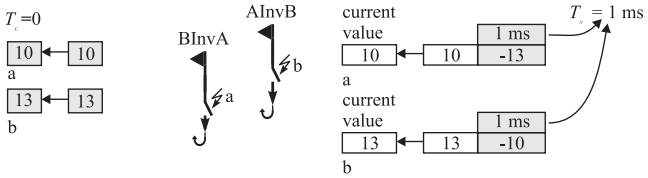


Beispiel Inverterring

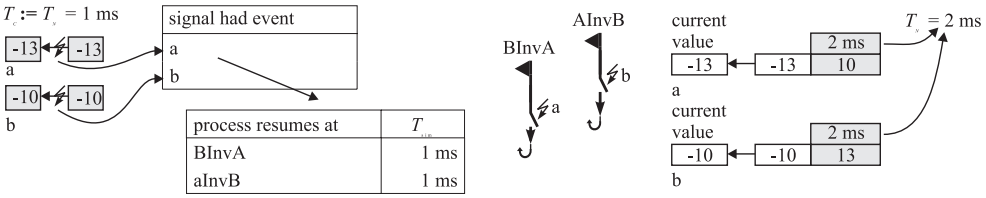
Am Beispiel Inverterring wird ein Schreibtischtest der Simulation mit Hilfe des **neuen** Algorithmus demonstriert.

```
entity Invring is
end entity Invring;

architecture InvringBhv of Invring is
  signal a : integer := 10;
  signal b : integer := 13;
begin

  AInvB: process is
  begin
    a <= -b after 1ms;
    wait on b;
  end process AInvB;

  bInvA: process is
  begin
    b <= -a after 1ms;
    wait on a;
  end process BInvA;

end architecture InvringBhv;
```





Init: $T_C = 0$
$a = 10(1ms)$
$b = 13(1ms)$
$T_N = 1ms$



a: $T_C = T_N = 1ms$
$T_C \neq time'high \Rightarrow$ weiter zu Schritt b

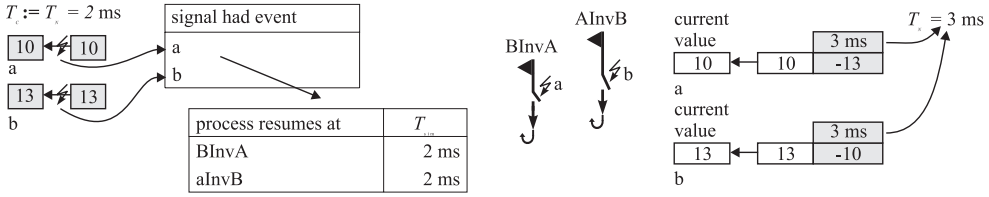
b: $a = -13 \Rightarrow$ Eintrag signal `a` in signal-event list
$b = -10 \Rightarrow$ Eintrag signal `b` in signal-event list

d: `BInvA` (1ms) in *process-resume list*
`AInvB` (1ms) in *process-resume list*

e: `AInvB` (beliebig gewählt) ausführen (bis `wait on b`)
`BInvA` ausführen (bis `wait on a`)
dabei erfolgen die *driver*-Einträge: `a` (10, 2ms) und `b` (13, 2ms)

f: $T_N = 2ms$

gehe zu a



a: $T_C = T_N = 2ms$
$T_C \neq time'high \Rightarrow$ weiter zu Schritt b

b: $a = 10 \Rightarrow$ Eintrag signal `a` in signal-event list
$b = 13 \Rightarrow$ Eintrag signal `b` in signal-event list

d: `BInvA` (2ms) in *process resume list*
`AInvB` (2ms) in *process resume list*

e: `AInvB` (beliebig gewählt) ausführen (bis `wait on b`)
`BInvA` ausführen (bis `wait on a`)
dabei erfolgen die *driver*-Einträge: `a` (-13, 3ms) und `b` (-10, 3ms)

f: $T_N = 3ms$

gehe zu a

a: ...

4.10 Was kann VHDL bisher?

Von dem *Simulation Cycle*mit *driver*, über die Einführung der *signals*bis hin zum neuen *Simulation Cycle*haben wir ein ganz bestimmtes Ziel verfolgt – **Hardware-Simulation** mit folgenden Eigenschaften:

- Simulation von paralleler, permanent aktiver Hardware
- Simulation von Hardware-Geschwindigkeit und Verzögerung
- Simulationsausführung auf sequentiellen Rechenmaschinen

Mit Hilfe eines Schreibtischtest des letzten Beispieles (Invertering) wurde gezeigt, dass diese Forderungen mit dem neuen *Simulation Cycle*erfüllt werden.

Mit Ausnahme von einigen zeitlichen Modellierungen ist das eingangs formulierte Ziel aus Kapitel 2: *korrekte Simulation von Hardware auf sequentiellen Rechenmaschinen* erreicht.

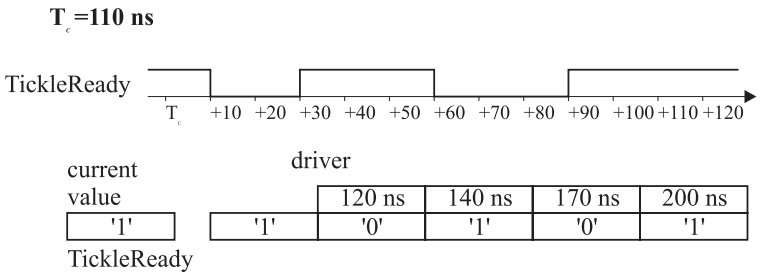
5 Verzögerung und ihre Modellierung

In diesem Kapitel werden die Modellierungsmöglichkeiten der Zeit genauer betrachtet. Insbesondere werden die Verzögerung (*transport delay*) und die Trägheit (*inertial delay*) der Hardware selbst und deren Modellierung in VHDL behandelt.

5.1 Waveform und Stimuli

Über die bisher beschriebenen Möglichkeit einem *signal*Werte zuzuweisen hinaus gibt es eine Erweiterung, um einem *signal*gleich mehrere Werte zuweisen zu können. Statt den Treiber eines *signal*mehrmals hintereinander zu laden, kann dem *signal*eine *waveform* zugewiesen werden. Beide Blöcke im folgenden Statement sind äquivalent.

```
TicklReady   <= '0' after 10 ns;   -- series of single assignments
TicklReady   <= '1' after 30 ns;
TicklReady   <= '0' after 60 ns;
TicklReady   <= '1' after 90 ns;

TickleReady <= '0' after 10 ns,   -- assignment of waveform
               '1' after 30 ns,
               '0' after 60 ns,
               '1' after 90 ns;
```



Stimuli sind *waveforms* mit der Funktion einer Nachbildung der Umgebung einer Schaltung. Anschaulich bilden *Stimuli* eine Schnittstelle nach Außen (z.B. wird ein Clock-Signal, welches nicht in der betrachteten Schaltung selbst erzeugt wird, durch *Stimuli* modelliert).

Beispiel D-FlipFlop

Bei der Modellierung eines D-FF sind folgende *signals* zu berücksichtigen:

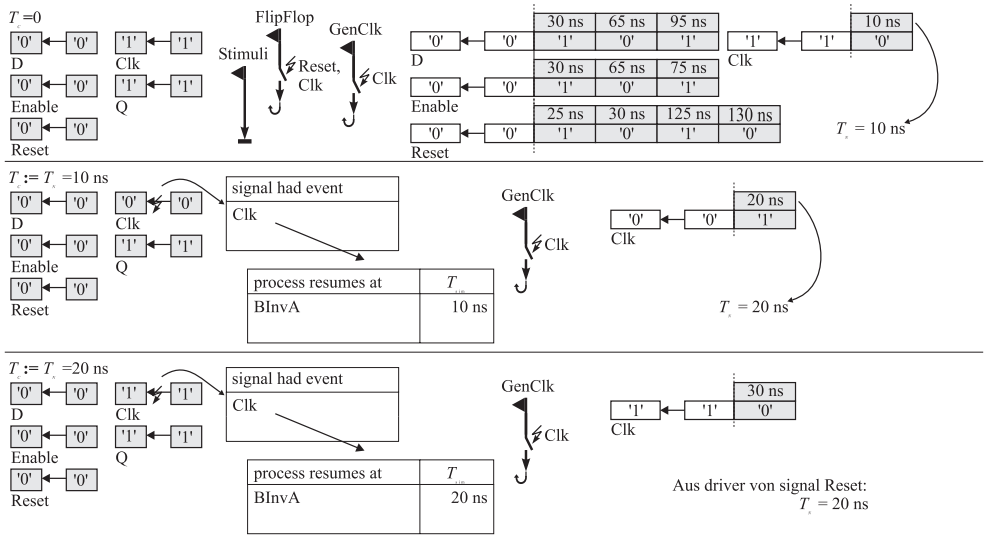
- `D` (Eingang)
- `Enable` (Eingang)
- `Reset` (Eingang)
- `Clk` (Eingang)
- `Q` (Ausgang)

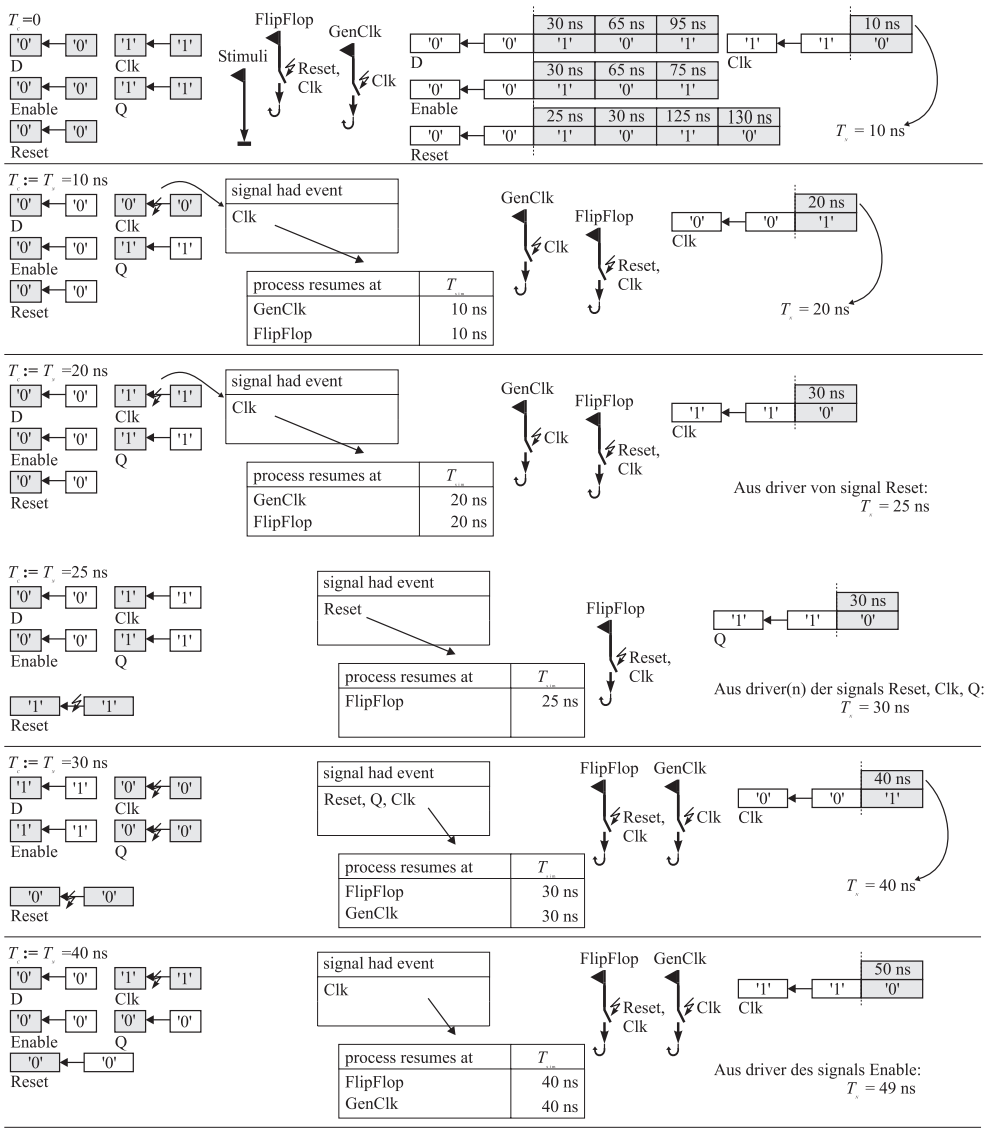
Als *Stimuli* werden hierbei die *signals* `D`, `Enable` und `Reset` modelliert, da sie asynchron von außen in die Schaltung eingehen (Nachbildung einer beliebigen Eingabe). Die *signals* `Q` und `Clk` werden im `FlipFlop`-*process*selbst modelliert, da `Clk` ein periodisches *signal*ist und `Q` in diesem *process*berechnet wird. Insbesondere wird das Testen von Sonderfällen durch dieses Verfahren sehr einfach und übersichtlich.

```
entity TestingFF is
end entity TestingFF;

architecture CompleteTest of TestingFF is
  signal D, Enable, Reset : bit := '0';
  signal Q, Clk           : bit := '1';
begin

FlipFlop: process is
begin
```

```
  if Reset = '1' then                        -- check for reset signal
    Q <= '0' after 5 ns;                     -- reset output
  elsif (Clk'event and Clk = '1') then       -- detect rising clock edge
    if Enable = '1' then                     -- check if enabled
      Q <= D after 6 ns;                     -- calculate output
    end if
  end if
  wait on Clk, Reset;                        -- sensitive for clock and reset
end process FlipFlop;

GenClk: process is
begin
  Clk <= not(Clk) after 10 ns;               -- invert clock signal
  wait on Clk;                               -- wait until inversion
end process GenClk;

Stimuli: process is
begin
  D      <= '1' after 30 ns,
            '0' after 65 ns,
            '1' after 95 ns;
  Enable <= '1' after 49 ns,
            '0' after 95 ns,
            '1' after 159 ns;
  Reset  <= '1' after 25 ns,
            '0' after 30 ns,
            '1' after 125 ns,
            '0' after 130 ns;
  wait;
end process Stimuli;

end architecture CompleteTest;
```

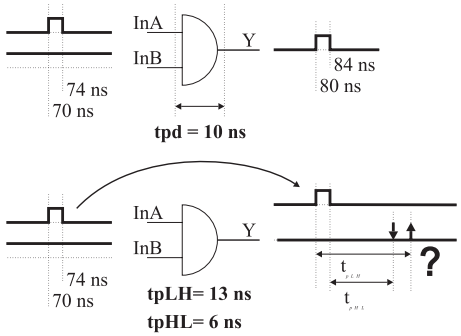
Einen, in gewohnter Weise dargestellten Schreibtischtest der Modellierung des `D-FlipFlop`:





5.2 Physikalische Verzögerung

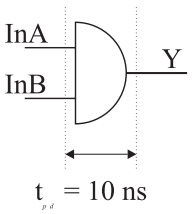
Bei der Simulation von physischer Hardware ist folgendes Verhalten zu berücksichtigen: Gilt $t_{pd} = t_{pHL} = t_{pLH}$ so wird ein Rechteckimpuls am Eingang eines Gatters um T_{pd} verzögert am Ausgang sichtbar. Gilt jedoch $t_{pLH} \geq$ Impulsbreite, so zeigt sich bei einer Änderung des *signals*von *Low* nach *High* und wieder zurück, dass am Ausgang keine *signal*-Änderung sichtbar wird, da der Ausgang erst nach t_{pLH} den Wert *High* annehmen würde, und zu dieser Zeit das Eingangssignal bereits wieder den Wert *Low* hat.



5.3 Modellierung der Verzögerung: transport delay

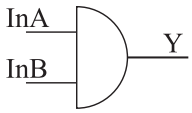
Im Kapitel 5.2 wurde der Zusammenhang von t_{pLH} und t_{pHL} im Bezug auf die Modellierung der zeitlichen Komponente verdeutlicht. Der Fall $t_{pd} = t_{pHL} = t_{pLH}$ kann in VHDL sehr einfach modelliert werden:

```
entity TransportDealay is
end entity TransportDelay;

architecture AND_Gate of TransportDelay is
    signal InA: bit := '0';
    signal InB: bit := '1';
    signal y:   bit;
begin
    Gate: process is
    begin
        y <= transport InA and InB after 10 ns;
        wait on InA, InB;
    end process Gate;

    Stimulate: process is
    begin
        InA <= transport '1' after 70 ns,
                         '0' after 74 ns;
        wait;
    end process Stimulate;
end architecture And_Gate;
```



Das *transport delay*stellt sicher, dass der gesamte Impuls verzögert durch das Gatter geleitet wird. Im Fall $t_{pHL} \neq t_{pLH}$ sieht die Realisierung in VHDL etwas anders aus:

```
entity TransportDealay is
end entity TransportDelay;

architecture AND_Gate of TransportDelay is
    signal InA: bit := '0';
    signal InB: bit := '1';
    signal y:   bit;
begin
    Gate: process is
        constant Tphl: time := 6ms;
        constant Tplh: time := 13ms;
        variable Tp:   time;
    begin
        if y = '0' then
            Tp := Tplh;
        else
            Tp := Tphl;
        end if;

        y <= transport InA and InB after Tp;
        wait on InA, InB;
    end process Gate;

    Stimulate: process is
    begin
        InA <= transport '1' after 70 ns,
                         '0' after 74 ns;
        wait;
    end process Stimulate;
end architecture And_Gate;
```



tpLH= 13 ns

tpHL= 6 ns

5.4 Auswirkungen des transport delayauf den driver

Eine verzögerte Zuweisung auf ein *signal*wird im *driver*als Plan gespeichert. Der Begriff *Plan* resultiert aus der Tatsache dass die eingetragenen Änderungen tatsächlich nur künftig geplante Änderungen sind und keine sofortigen, da die berechneten *driver*-Änderungen immer nur vom aktuellen Zustand des gesamten Systems ausgehen.

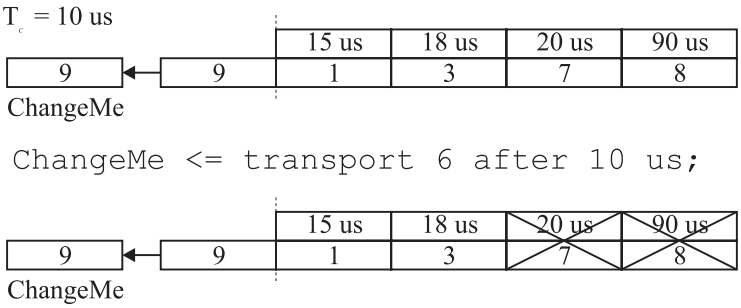
Was passiert aber, wenn der Plan für ein *signal*für einen bestimmten Zeitraum bereits berechnet wurde und eine (unerwartete) Änderung für dieses *signal*innerhalb dieses Zeitraumes eintritt? Die Kausalität kann durch einfaches Eintragen im *driver*aller Änderungen nicht garantiert werden, da die Einträge, resultierend aus einer aktuellen Berechnung, die künftigen Änderungen nicht berücksichtigen können.

Alle künftigen Änderungen hängen aber von jenen Änderungen ab, die zeitlich vor ihnen passieren, was am Beispiel des folgenden VHDL-Code Ausschnittes verdeutlicht werden soll:

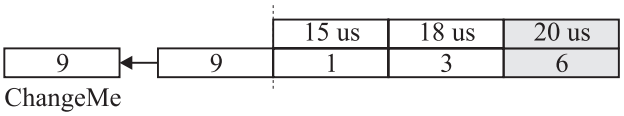
```
if signalA = '0' then
    signalB := '1' after 10 ms;
else
    signalB := '0' after 10 ms;
end if;
```

Der Wert von `signalB` hängt vom Wert `signalA` zum Zeitpunkt T_C (aktuelle Zeit) ab. Sei zum aktuellen Zeitpunkt `signalA = '0'`, so erfolgt ein Eintrag im *driver*von `signalB` mit dem Zeitwert: T_C+10ms und dem Wert '1'. Wird zu einem Zeitpunkt **vor** T_C+10ms (z.B. von einem andern *process*) die Zuweisung: `signalA := '1'` ausgeführt, so muß der nun nicht mehr gültige **Plan** (*driver*) von `signalB` korrigiert werden!

a Alle alten Zeiten (Wertpaare oder *transactions*), die später oder gleichzeitig als die früherste neu einzuordnende *transaction* aktiv würden werden gelöscht.

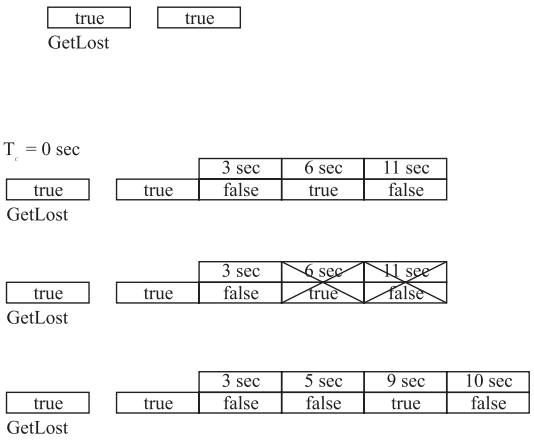


b Alle neuen *transactions* an den *driver*anhängen (eintragen)



Beispiel driver-Korrektur

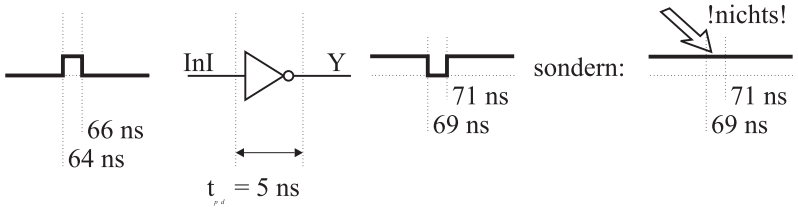
```
entity KeepTrack is
end entity KeepTrack;

architecture Demo of KeepTrack is
    signal GetLost: boolean := true;
begin
    SeveralTimes: process is
    begin
        GetLost <= transport
             false after  3 sec;
             true  after  6 sec;
             false after 11 sec;
            ...
        GetLost <= transport
             false after  5 sec;
             true  after  9 sec;
             false after 10 sec;
        wait;
    end process SeveralTimes;
end architecture Demo;
```



Die Berechnung der *driver*Inhalte erfolgt nach dem vorgestellten Verfahren. Bei der ersten Zuweisung wird der Plan von `GetLost` mit 3 Werten gefüllt. Bei der zweiten Zuweisung werden zuerst die letzten beiden Wertpaare im Plan gelöscht (6 sec und 9 sec), da die früheste neue *transaction* bei 5 sec stattfindet. Dann werden die 3 neuen *transactions* am Plan angehängt.

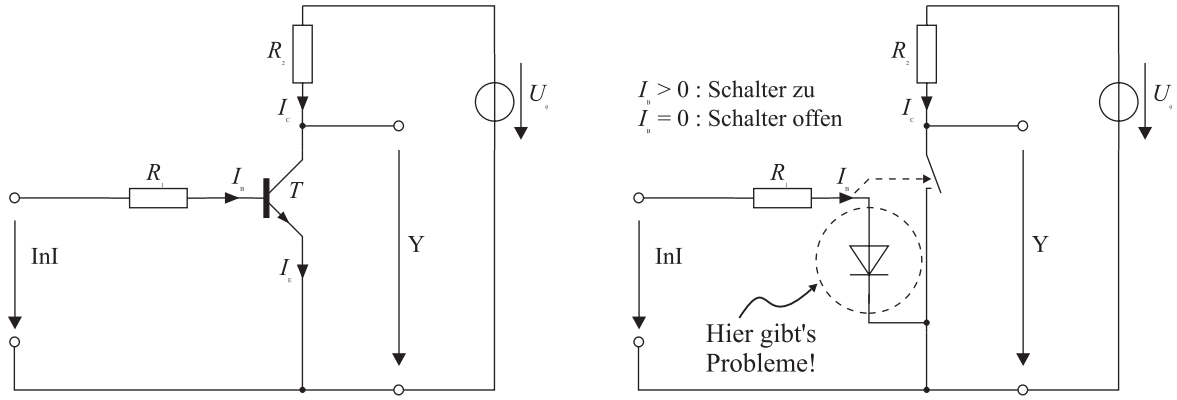
6 Trägheit und ihre Modellierung

Im Kapitel 5 wurde die physikalische Verzögerung und ihre Modellierung behandelt, darüberhinaus ist bei der Simulation von Elektronik die Trägheit zu beachten. Sie wird mit *inertial delay*bezeichnet. Ist die Änderung des Eingang-*signal*von einem Inverter zeitlich kürzer als seine Verzögerungszeit, so ändert der Ausgang seinen Wert nicht!

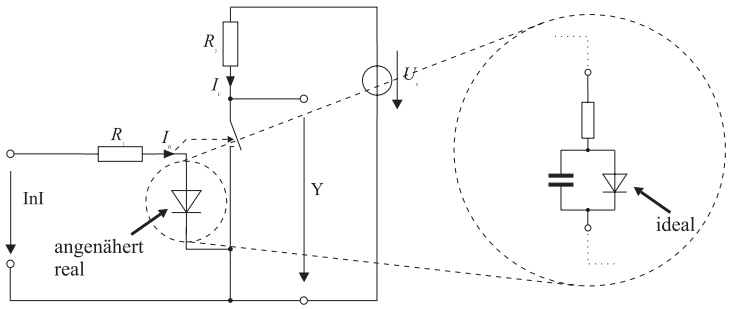


6.1 Physikalische Trägheit

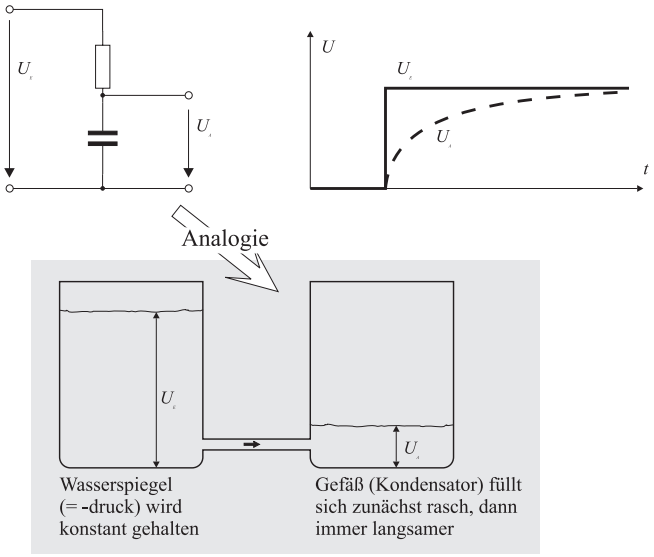
Betrachten wir zunächst das Schaltbild eines Invertes. Die vereinfachte Darstellung im Bild rechts zeigt eine Diode, die nachfolgend genauer untersucht wird.



Bei der Darstellung im rechten Bild handelt es sich um eine reale Diode. Diese reale Diode kann in eine Parallelschaltung einer Kapazität und einer idealen Diode zur weiteren Betrachtung umgewandelt werden.



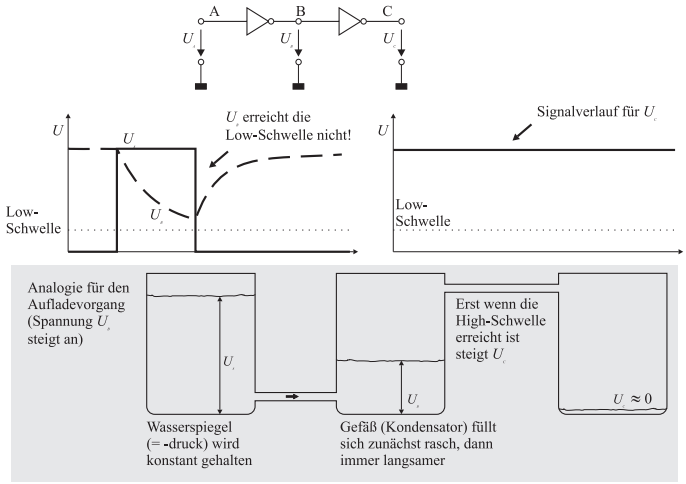
Die Spannungsänderung einer Kapazität ist nicht sprunghaft sondern degressiv (beim Aufladevorgang). Bis zum Erreichen einer gewissen Spannung verstreicht eine gewisse Zeit. Zum besseren Verständnis die Analogie des Aufladevorgangs einer Kapazität zu Wassergefäßen:



Der linke Waserbehälter hat zu Beginn einen hohen Wasserstand (großer Druck am Gefäßboden). Durch die Rohrverbindung zum rechten Behälter (zu Beginn niedriger Wasserstand) strömt aufgrund des Druckunterschiedes Wasser vom linken in den rechten Behälter. Je größer der Druckunterschied ist, umso größer ist der Durchfluß (Volumen pro Zeiteinheit) im Verbindungsrohr. Dabei gleichen sich die beiden Wasserstände nach und nach aneinander an, bis sie zuletzt gleich sind. Dabei ist der Wasserstand des linken Gefäßes (Zufluß) konstant, wie auch die anliegende Spannung konstant ist. Je geringer der Unterschied der Wasserstände wird, desto geringer wird auch der Druckunterschied und der Durchfluß. Der rechte Behälter füllt sich zuerst schnell, dann immer langsamer, gleichartig dem Spannungsaufbau in einer Kapazität.

Erst wenn die Spannung am Gatter-Eingang einen gewisser Schwellwert erreicht, kann sich am Gatter-Ausgang eine Änderung zeigen. Zwei verschiedene Schwellwerte sind zu berücksichtigen (je nach möglicher Änderung):
High-Schwelle und **Low-Schwelle**. Erst dann, wenn die **High-Schwelle** erreicht wird beginnt der Spannungsanstieg (Abfall) am Gatter-Ausgang (analoges gilt für die **Low-Schwelle**).

Schaltet man zwei Inverter in Serie, so kann am Ausgang nur dann eine Änderung eintreten, wenn die **High-Schwelle** für den zweiten Inverter erreicht wird. Die folgende Abbildung zeigt wieder einen anschaulichen Vergleich mit drei Wasserbehälter.



6.2 Modellierung der Trägheit: inertial delay

Wie in Kapitel 6.1 erklärt wurde, muß ein Pegel über eine Mindestzeit am Gattereingang anliegen, bevor er am Gatterausgang Wirkung zeigen kann. Diese Eigenschaft wird als `Trägheit` bezeichnet. Bei der Modellierung mit VHDL wird hierfür der Begriff *inertial delay*verwendet.

Beispiel zu Verzögerung und Trägheit

Dieses zeigt, wie in VHDL *inertial delay*und *transport delay*angegeben werden.

```
AndIt: process is
    constant Tpropagation : time := 10 ns;
    constant Tinertial    : time :=  8 ns;
begin
    y <= reject Tinertial inertial InA and InB
         after Tpropagateion;
    wait on InA, InB;
end process AndIt;
```

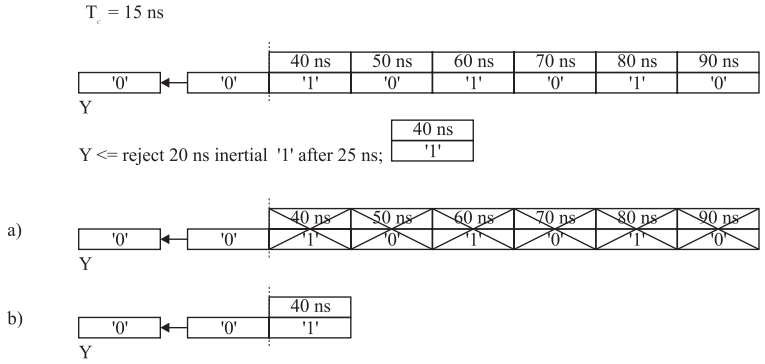
6.3 Auswirkungen des inertial delayauf den driver

Wie beim *transport delay*gelten auch für das *inertial delay*neue Regeln zur *driver*-Aktualisierung:

- Vorab wie beim *transport delay*:
  - (a) Alle alten *transactions*, die später (oder gleichzeitig) als die früheste neu einzuordnende *transaction* aktiv würden löschen
  - (b) Neue *transaction* an *driver*anhängen
- Zusätzlich beim *inertial delay*:
  - (c) Alle neuen *transactions* markieren (die im aktuellen *Simulation Cycle*hinzugekommenen)
  - (d) Alte *transactions* markieren, die um mehr als $T_{inertial}$ früher aktiv würden, als die früheste neue *transaction* (*rejection limit*)
  - (e1 - en) Für alle markierten *transaction*: Alte *transactions* markieren, die vor einer markierten *transaction* mit gleichem Wert-Eintrag stehen (nicht relevant)

– (f,g) Alle unmarkierten *transactions* (dies sind nur alte *transactions*!) löschen, dabei den aktuellen Wert des *driver*nicht löschen

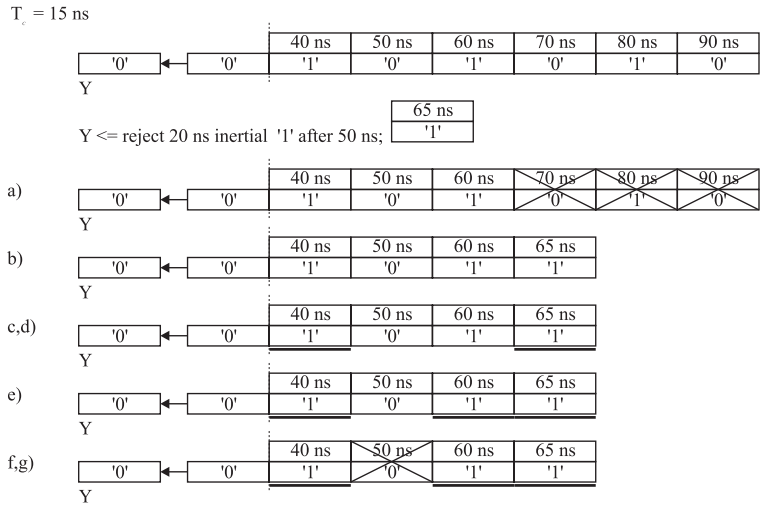
Beispiel zur Auswirkung des inertial delayauf den driver



T_C = 15 ns `y <= reject 20 ns inertial '1' after 25 ns;` ⇒ Zeiteintrag mit 40 ns und Wert = '1'

a Streichen aller Einträge größer oder gleich 40 ns, der gesamte Plan wird verworfen.

b Anhängen des neuen Eintrags



T_C = 15 ns `y <= reject 20 ns inertial '1' after 50 ns;` ⇒ Zeiteintrag mit 65 ns und Wert = '1'

a Streichen aller Einträge größer 65 ns

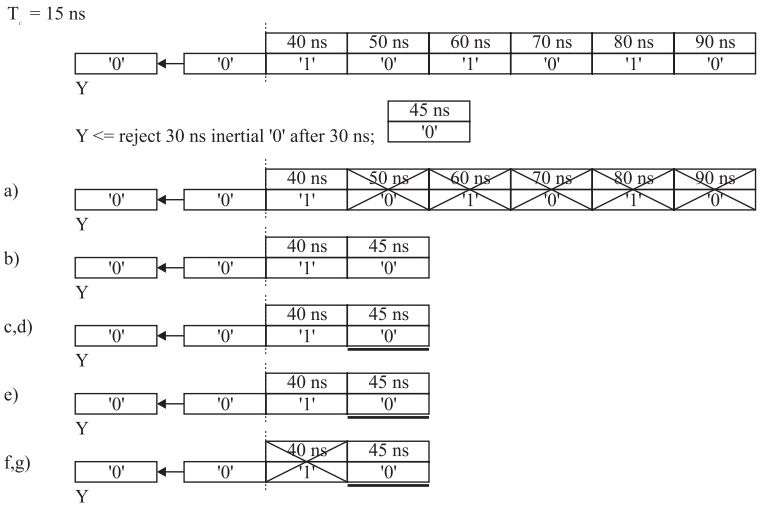
b Anhängen des neuen Eintrags

c Markieren aller neuen *transactions*

d $T_{inertial}$ = 20 ns, 65 ns - 20 ns = 45 ns: alle *trabsactions* vor 45 ns markieren

e Nur die *transaction* bei 60 ns steht vor einer markierten *transaction* mit demselben Wert

f,g Die unmarkierte *transaction* wird gelöscht



$T_C = 15$ ns `y <= reject 30 ns inertial '0' after 30 ns;` ⇒ Zeiteintrag mit 45 ns und Wert = '0'

a Streichen aller Einträge größer 45 ns

b Anhängen des neuen Eintrags

c Markieren aller neuen *transactions*

d $T_{inertial} = 30$ ns, 45 ns - 30 ns = 15 ns: alle *trabsactions* vor 15 ns markieren

e Keine unmarkierte *transaction* steht vor einer Markierten mit gleichem Wert.

f,g Die unmarkierte *transaction* wird gelöscht

6.4 Zur Beachtung beim inertial delay

Bei der Verwendung von *inertial delay*ist immer zu bedenken:

- Wie die *driver*-Korrektur beim *transport delay*, gilt auch für das *inertial delay*: **Vorsicht** bei mehreren sequentiellen (in einem *process* oder *subprogram*) *signal assignments*.
- Die Beobachtung von „unerklärlichen Fehlern"kann durch das Löschen von *transactions* im *driver*hervorgerufen werden (gilt auch für *transport delay*).
- *signal- assignments* verändern nicht den Wert eines *signal*, sondern lediglich seinen assoziierten *driver*. Der Wert eines *signal*zieht frühestens im nächsten *Simulationszyklus* nach (gilt auch für *transport delay*).

Häufig gilt: $T_{propagation} = T_{inertial}$

Wird bei einer *signal*-Zuweisung nichts anderes angegeben, so wird $T_{propagation} = T_{inertial}$ angenommen. Aus dieser Standard-Annahme resultiert die gleiche Bedeutung für folgende VHDL-Anweisungen:

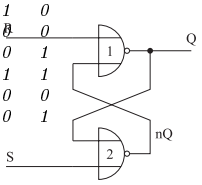
```
-- eine VHDL Anweisung:
y <= reject 10 ns inertial InA after 10 ns;

-- oder kuerzer geschrieben:
y <= inertial InA after 10 ns;

-- und noch kuerzer:
y <= InA after 10 ns;
```

7 Idealisierung

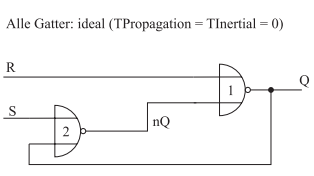
7.1 Delta Delay

Beim Design von Hardware sind Verzögerungszeiten und Trägheit primär oft nicht interessant. Als Idealisierung wird daher oft $T_{propagation} = T_{inertial} = 0$ angenommen. Trotzdem sollen auch in diesem idealisierten Modell Einschwingvorgänge berücksichtigt werden. Am Beispiel eines (idealen) RS-FlipFlops soll demonstriert werden, wie der Simulator reagiert.

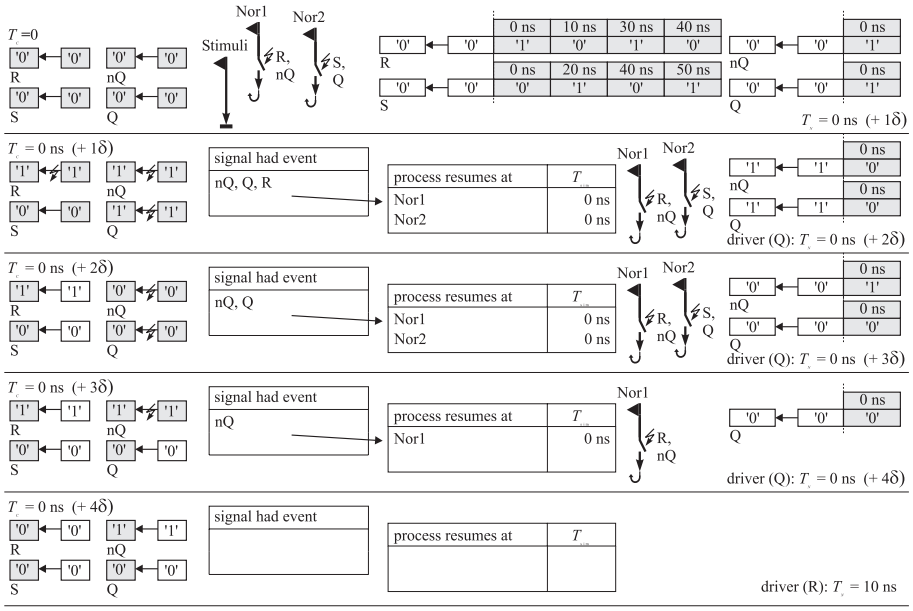
```
entity ShowRSFF is
end entity ShowRSFF;

architecture DemoIdeal of ShowRSFF is
  signal R, S, Q, nQ : bit;
begin
  Nor1: process is
  begin
    Q <= R nor nQ;
    wait on R, nQ;
  end process Nor1;

  Nor2: process is
  begin
    nQ <= S nor Q;
    wait on S, Q;
  end process Nor2;

  Stimuli: process is
  begin
    R <= '1' after  0 ns,      -- time    R  S
         '0' after 10 ns,      ------------------
         '1' after 30 ns,      --  0 ns   1  0
         '0' after 40 ns;      -- 10 ns   0  0
    S <= '0' after  0 ns,      -- 20 ns   0  1
         '1' after 20 ns,      -- 30 ns   1  1
         '0' after 40 ns,      -- 40 ns   0  0
         '1' after 50 ns;      -- 50 ns   0  1
    wait;
  end process Stimuli;
end architecture DemoIdeal
```



=



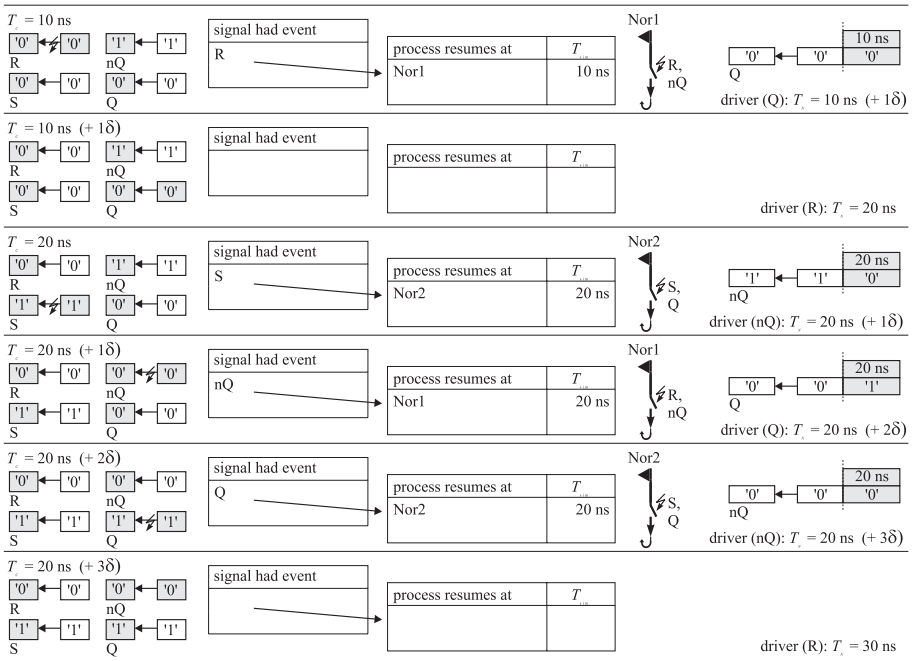
Betrachten wir wieder Anhand der Schreibtischsimulation, was passiert:



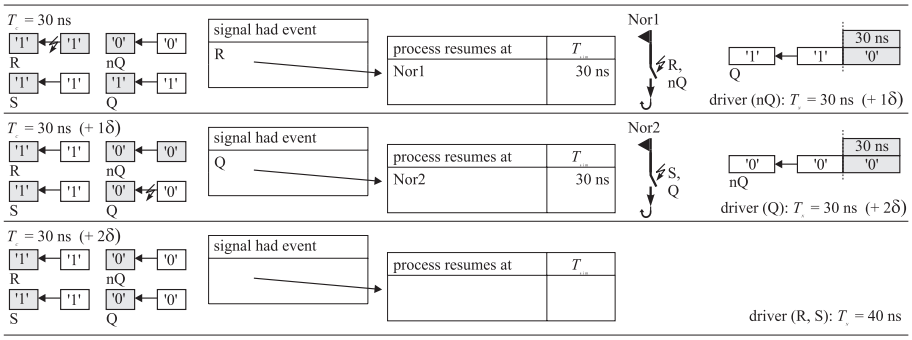
Zum Zeitpunkt $T_C = 0$ werden die Treiber von `R` und `S` ausgewertet, und beide *signals*bekommen neue Werte. Die *processes* `Nor1` und `Nor2` werden aktiv, und `Q` und `nQ` bekommen ebenfalls zum Zeitupnkt $T_C = 0$ neue Einträge im Plan, weil im idealisierten Modell keine Verzögerung definiert wurde!

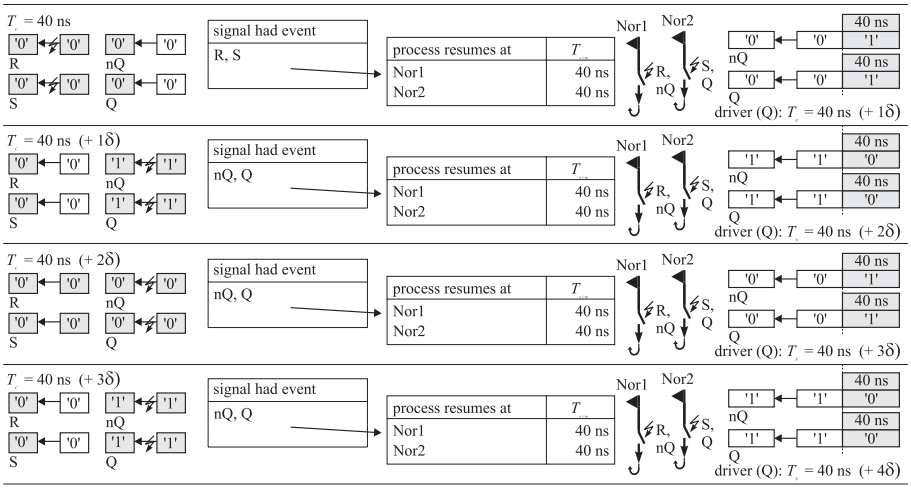
`R`, `nQ` und `Q` bekommen zum Zeitpunkt $T_C = 0$ neue Werte, und die *processes* `Nor1` und `Nor2` werden wieder aktiv, ohne dass die Simulationszeit fortgeschritten ist. Wieder bekommen `Q` und `nQ` neue Einträge im Plan zum aktuellen Simulationszeitpunkt, noch einmal werden beide *processes* `Nor1` und `Nor2` aktiv. Nun wird lediglich `nQ` verändert, und der *process* `Nor1` wird noch einmal aktiv. Erst danach ändert sich nichts mehr an den Plänen der *signals*, und keine Prozesse werden mehr aktiv. Nun kann die Simulationszeit fortgeschrieben werden.

Obwohl T_C noch immer auf 0 ns steht, sind während dieser Phase der Simulation 4 sogenannte *delta cycle* vergangen.



Bei $T_C = 10$ ist nur 1 *delta cycle* notwendig, bis die *process resume list* leer ist und die Simulationszeit auf 20 ns fortgeschrieben werden kann. Auf ähnliche Weise wird die Simulation bis zu $T_C = 40$ fortgeführt.

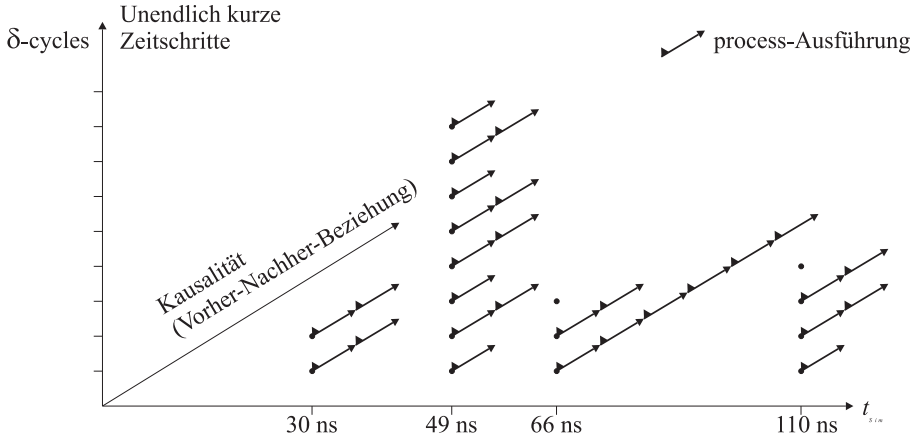




Das Hin- und Herpendeln von Q und nQ setzt sich in gleicher Weise "ewig" fort... (T_s bleibt allerdings bei 40 ns!)

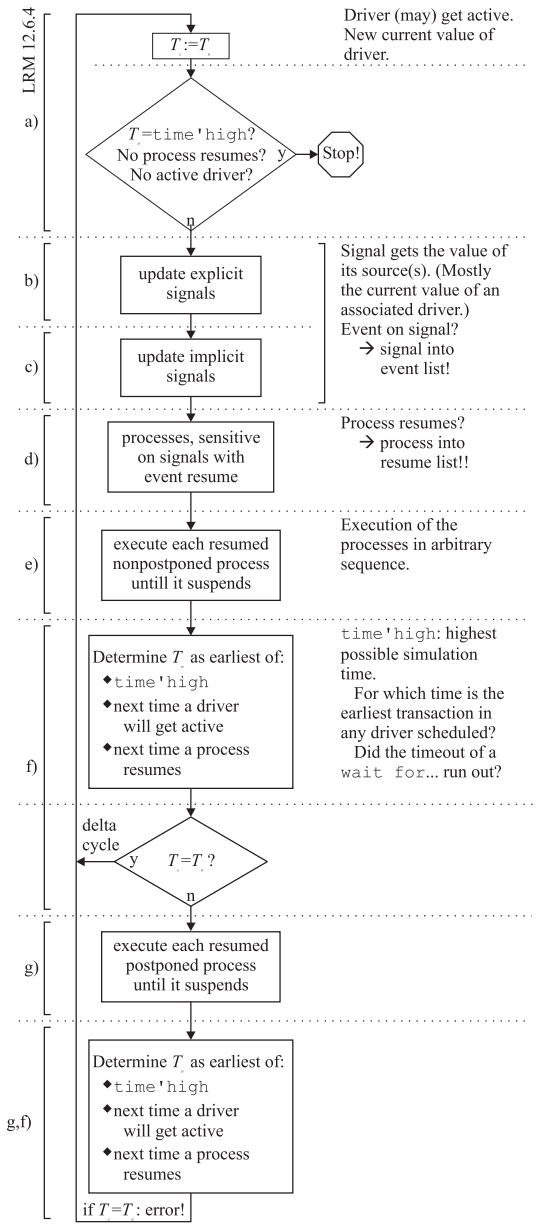
Bei $T_C = 40$ kann man feststellen, dass die Simulation hier in einem Zyklus festsitzt. Bei jedem *delta cycle* haben beide *signals* `Q` und `nQ` events, beide *processes* `Nor1` und `Nor2` werden wieder aufgerufen. `Q` und `nQ` schwingen zwischen den beiden Zuständen '0' und '1', die Simulationszeit kann nicht fortgeschrieben werden.

Um diese Simulation durchführen zu können, benutzt VHDL ein zweidimensionales Zeitmodell. Einerseits wird die Simulationszeit in den definierten Zeiteinheiten fortgeschrieben, wobei die kleinste in VHDL definierte Zeiteinheit 1 fs ist. Die *delta cycles* sind nicht ein Teil der Simulationszeit sondern ein „unendlich kurzer Zeitraum “ während dessen sich *signals* noch ändern können. *Delta cycles* können also nicht in ms umgerechnet werden, sie sind eine eigene zeitliche Dimension in VHDL.



7.2 Simulation Cycle

Bezieht man das Konzept der *delta cycles* in den *simulation cycle* mit ein, so ergibt sich der Ablauf der Simulation folgendermaßen:



7.3 Postponed Prozesse

Haben sich alle *signals*zum Simulationszeitpunkt T_C eingeschwungen, d.h. sind alle *delta cycles* abgearbeitet worden, so wird vor dem nächsten Simulationszyklus die Simulationszeit fortgeschrieben (T_C wird erhöht). VHDL bietet das Konzept der *postponed processes* um genau zu diesem Zeitpunkt einen *process*genau einmal zu aktivieren. Ein solcher *process*läuft also noch bevor die Simulationszeit fortgeschrieben wird, alle *signals*sind bereits stabil.

Postponed processes werden mit dem Schlüsselwort `postponed` deklariert. Sie eignen sich vor allem zur Laufzeitanalyse oder zur Erzeugung von Stimuli.

```
TestFinalValues: postponed process is
begin
```

```
    assert (A < B)
      report "Joe: A and B don't relate right finally !"
      severity warning;
    wait on A, B;
  end postponed process TestFinalValues;
```

8 Strukturbeschreibung

Bisher beschränkt sich die Abstraktion im Hardwareentwurf auf die zeitliche Modellierung oder die Verwendung von abstrakten Datentypen. So wurden alle notwendigen Prozesse innerhalb einer `architecture`, also auf der selben Ebene implementiert. Darunter fällt auch die mehrfache Implementierung von Prozessen mit gleichem Verhalten.

8.1 Nachteile des bisherigen Entwurfs

- Codeverdopplung: Kommen in einer Schaltung mehrere Komponenten vor, die die gleiche Funktionalität aufweisen (z.B. oder Nor1, Nor2) mussten diese mehrmals beschrieben werden.
- Unzureichende Hierarchische Struktur: Unübersichtlicher Schaltungsentwurf steigerte die Komplexität bei großen Schaltungen. Zur Strukturierung wurden lediglich herkömmliche Hilfsmittel (Funktionen und Prozeduren) innerhalb der Prozesse verwendet. Die horizontale und vertikale Strukturierung der parallel ablaufenden Prozesse selbst fehlte ganz.
- Keine Abstraktion: Jede Teilschaltung muß immer wieder bis ins unterste Detail neu beschrieben werden.
- Durch die eingeschränkte hierarchische Modellierung war auch nur ein eingeschränktes *information hiding* möglich. Dieses galt nur für Variablen innerhalb von Prozessen, Funktionen und Prozeduren.

8.2 Strukturierungsmechanismen in VHDL

VHDL bietet die Möglichkeit, Module zu erstellen, die durch eine Schnittstelle (*ports*) und die dazugehörige Beschreibung der inneren Funktionalität (*architecture*) definiert werden. Von dieser Definition ausgehend können beliebig viele Instanzen erzeugt werden. Werden in einer Schaltung beispielsweise mehrere FlipFlops benötigt, wird für die FlipFlops eine Moduldefinition erstellt (entity/architecture). Davon werden soviele Instanzen wie notwendig in der Schaltung instantiiert.

- Strukturierung: Ein großes Problem wird in kleinere Teilprobleme zerlegt, dadurch wird die Übersicht über die gesamte Schaltung verbessert. Das entspricht dem Prinzip der Top-Down-Entwicklung im Software-Engineering.
- Schachtelung: Ein Modul kann wiederum mehrere Instanzen verschiedener anderer Module beinhalten. Das Entity/Architecturemodell ist hier vergleichbar mit objektorientierten Programmiersprachen (C++ etc.).
- Information Hiding: Die interne Funktionsweise der Module bleibt unsichtbar und ist für den Benutzer dieser Module nicht relevant. Die Kommunikation zwischen den einzelnen Modulen erfolgt nur über externe Schnittstellen.

```
entity BusControl is
  port (Enable       : in bit;
        Clk, Reset   : in bit;
        AdrBus       : in bit_vector(7 downto 0);
        Valid, Busy  : out bit;
        Error        : out bit;
        DataBus      : inout bit_vector(15 downto 0));
end entity BusControl
```

Je nach Perspektive des Betrachters ergeben sich 2 unterschiedliche Sichtweisen eines Moduls.

- Externe Sichtweise (Blackbox): Bei der Verwendung eines fertigen Moduls ist für den Benutzer nur seine Schnittstelle relevant, also die Ein- und Ausgangsports. Die genaue Art der Implementierung ist in den meisten Fällen nebensächlich. Eine *entity* ist die Modulbeschreibung und entspricht der externen Sichtweise.
- Interne Sichtweise: Stellt die Perspektive eines Modulentwicklers dar. Dieser beschäftigt sich mit dem genauen Aufbau und der Funktionalität des Moduls. Die *architecture* ist die Implementierung einer *entity* und entspricht der internen Sichtweise.

8.3 Entity

```
entity Bezeichner is
   [ generic ( genericport_liste ); ]
   [ port ( port_liste ); ]
   { typ_deklaration }
end [ entity ] [ Bezeichner ] ;

genericport_liste :=
 Bezeichner1 {, ...} : [ mode ] subtyp [ := expression ] {; ...}

port_liste :=
 Bezeichner1 {, ...} : [ mode ] subtyp [ := expression ] {; ...}

mode := in | out | inout
```

Eine `entity` besteht also aus einem Bezeichner, einer Liste von Generic- und Portbeschreibungen, sowie Typdeklarationen. `ports` sind globale *signals*, deren Datenfluss durch die Angabe des `mode` definiert ist. Die *entity*-Deklaration ist ein syntaktisch erforderlicher Bestandteil jedes VHDL-Entwurfes, auch wenn in speziellen Fällen keine Schnittstelle nach außen benötigt wird (z.B. Testbench). Die genaue Funktionsweise der *entity*-Deklaration wird anhand einiger Beispiele erläutert.

```
entity Testbench is
end Testbench ;
```

Diese (einfachste) Variante der *entity*-Beschreibung wird verwendet um eine Umgebung zu schaffen, innerhalb der fertig erstellte Module getestet werden. Für diesen Testrahmen werden keine Ein- oder Ausgangsports benötigt, da sämtliche Testsignale intern erzeugt werden. Die Testbench ist also eine Toplevel-*entity*, ein Simulationsmodell für die zu testenden *entities*.

```
entity FlipFlop is
   port (R : in bit;
         S : in bit;
         Q : out bit;
         nQ: out bit);
end entity FlipFlop;
```

Dieses Beispiel beschreibt die Entity `FlipFlop` mit jeweils 2 Ein- und Ausgangsports vom Typ bit.

```
entity 4BitAdder is
   port (a1, a2, a3, a4 : in bit := 0 ;
         y1, y2, c      : out bit);
end entity 4BitAdder;
```

Bei diesem 4-Bit-Addierer werden die Input-Ports mit einem Default-Wert (0) belegt, der gültig ist, solange keine Beschaltung des jeweiligen Eingangs erfolgt.

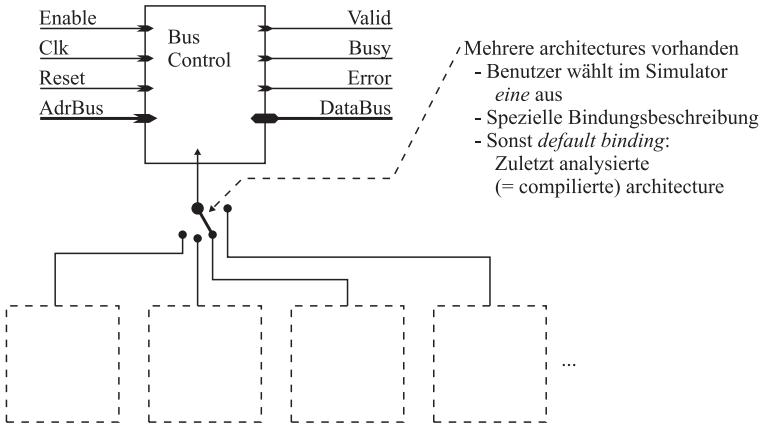
```
entity BCD_to_7Segment is
   port ( bcd_in         : in std_ulogic_vector (0 to 3);
          SSegment_out   : out std_ulogic_vector (0 to 6));

   subtype Ssegment is bitVector (0 to 6);
   type BCD_array is array (0 to 9) of Ssegment;
   constant BCD : BCD_array := (B"1111110", B"1100000", ...);

end entity BCD_to_7Segment;
```

Um nach außen hin zu verdeutlichen, wie die Kodierung der 7 Segmente innerhalb der *entity* erfolgt, wurden in diesem Beispiel Typ- und Konstantendefinitionen in den *entity*-Header gezogen. In der *entity*- Deklaration können so auch Umgebungs oder Initialisierungsparameter des Moduls zugänglich gemacht werden.

8.4 Architecture

```
architecture Bezeichner of Entity_Bezeichner is
   { block_declarative_item }
begin
   { concurrent_statement }
end [ architecture ] [ Bezeichner ] ;

block_declarative_item :=
   subprogram_declaration
   | subprogram_body
   | type_declaration
   | subtype_declaration
   | constant_declaration
   | signal_declaration
   | alias_declaration
   | component_declaration
   | configuration_specification
```

Nachdem das Interface des Moduls in der *entity*- Deklaration beschrieben wurde, kann in einem oder mehreren `architecture`-Teilen die Implementierung vorgenommen werden. Jeder *architecture*-Teil beschreibt dabei eine unterschiedliche Sicht der *entity*. Sind mehrere *architectures* vorhanden, bestimmt entweder der Benutzer die *architecture*, die verwendet werden soll, oder durch eine Bindungsbeschreibung wurde das bereits festgelegt. Ansonsten verwendet der Simulator die zuletzt analysierte *architecture*.



Die *Block-Declarative-Items* in der *architecture* sind Deklarationen, die zur Implementierung benötigt werden. Sie können Typ-, Konstant-, Signal- oder andere Deklarationen beinhalten. Die

Concurrent-Statements in der *architecture* beschreiben die Funktionsweise des Moduls. Eine Form eines Concurrent-Statements ist ein Prozess, der aus sequentiellen Befehlen bestehen kann. Eine weiter Möglichkeit des Concurrent-Statements ist eine Zusammenfassung verschiedener Teilschaltungen (Components).

```
architecture Behavioral of FlipFlop is
begin
    FlipFlop_Process1 : process is
    begin
        Q := R nor nQ;
        wait for 20 ns;
    end process FlipFlop_Process1;

    FlipFlop_Process2 : process is
    begin
        nQ := S nor Q;
        wait for 20 ns;
    end process FlipFlop_Process2;
end architecture FlipFlop;
```

In der `architecture Behavioral` der Entity FlipFlop sind 2 Prozesse, die das Verhalten der Nor-Gatter aus denen das FlipFlop aufgebaut werden soll beschreiben.

Die Trennung von *entity* und *architecture* bringt mehrere Vorteile mit sich. Nach dem Architekturentwurf steht die Definition der *entity* fest und bleibt (meist) stabil. Diese Definition ist eine unabhängige Compilation-Einheit und kann bereits vor dem Entwurf des Verhaltens (statisch) getestet werden. Das Verhalten wird dann in einer *architecture* festgelegt, während des Entwicklungsverlaufs kann diese Beschreibung verfeinert und unabhängig von der zugehörigen *entity* compiliert werden.

8.5 Instantiation

Entsprechend des eingangs beschrieben Schablonenprinzipes kann man, ausgehend von der Moduldefinition (Entity/Architecture) beliebig viele Instanzen erzeugen und in eine Schaltung integrieren. Eine Instanz kann man sich als Kopie eines Moduls vorstellen. So können komplexe Gesamtsysteme aus einzelnen Bausteinen zusammengesetzt werden, da in der Architecture einer Entity wieder andere Entities instantiiert und verwendet werden können. Die *ports* dieser Entities werden an globale *signals* der Architecture oder an *ports* anderer Entities angeschlossen.

Um eine Entity zu instantiieren muss bekannt sein, aus welcher Library die Entity stammt und welche Architecture verwendet werden soll. Diese Form der Instantiation entspricht der direkten Instantiation einer Entity. Direkt deshalb, weil eine konkret spezifizierte Entity und Architecture zur Anwendung kommt.

```
Instanz_Bezeichner : entity Entity_Bezeichner
   [ ( Architecture_Bezeichner) ]
   [ port map ( Port_Verbindungsliste ); ]
```

```
library work;
library WilmaLib;

entity PlugAndPlay is
  port (Data       : inout bit_vector (7 downto 0);
        Init       : in bit;
        Clk, Reset : in bit;
        Write      : in bit;
        Configured : out bit;
        A1, A2, A3 : out bit);
end entity PlugAndPlay;

architecture Struct of PlugAndPlay is
  signal A, B, C, D, E, F, B, H, I, J, Karl, Fritz : bit;
  signal BA, BB, BC : bit_vector (7 downto 0);
begin
```

```
  Timing: entity Work.Timer(Behavioral)
    port map (Trigger => A, TimeOut => B);

  Decoding: entity Work.RegisterDecode(RTL)
    port map (DataBus       => Data,
              InitRegister  => Init,
              Reset         => Reset,
              Clk           => Clk,
              Delay         => A,
              NotARegister  => C,
              WrittenA      => D,
              Error         => E,
              ...
  Configuring: antity Work.ConfigurationRegister(RTL)
    port map (Valid => Configured,
              ...

end architecture Struct;
```

In der `port map` Liste erfolgt die Zuordnung aller Ports einer instantiierten Entity zu Signals oder Ports der umgebenden Architecture. Das Instantiation Statement ist, wie des Process Statement, ein Concurrent Statement für die parsallele Umgebung.

Im folgenden Beispiel beschreiben wir ein Schieberegister mit 4 Verzögerungsschritten, durch D-FlipFlops realisiert werden. Wir setzen dabei voraus, daß in einer bestehenden Library, hier FlipFlops genannt, ein DFlipFlop als Entity/Architecture vorhanden ist.

```
library FlipFlops;

entity SchiebWeg4 is
   port  ( clk_in   : in bit,
           data_in  : in bit,
           data_out : out bit);
end entity SchiebWeg4;

architecture behav of SchiebWeg4 is
   signal D1, D2, D3;
begin
   FF1 : entity FlipFlops.DFlipFlop(behav)
       port map (clk => clk_in,
                 d => data_in,
                 q => D1);
   FF2 : entity FlipFlops.DFlipFlop(behav)
       port map (clk => clk_in,
                 d => D1,
                 q => D2);
   FF3 : entity FlipFlops.DFlipFlop(behav)
       port map (clk => clk_in,
                 d => D2,
                 q => D3);
   FF4 : entity FlipFlops.DflipFlop(behav)
       port map (clk => clk_in,
                 d => D3,
                 q => data_out);
end architecture SchiebWeg4;
```

Die Instanzen aller 4 FlipFlops werden hier nach dem gleichen Schema erstellt:

```
FFn : entity FlipFlops.DFlipFlop(behav)
      port map (clk => ..., d => ..., q => ...);
```

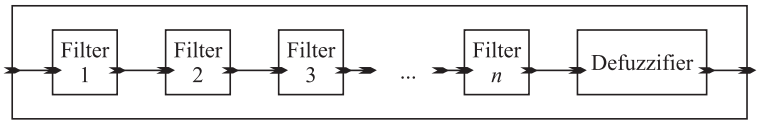
Dabei ist `FFn` der Name der Instanz, FlipFlops die Bezeichnung der Library, in der sich das DFlipFlop befindet und behav eine Architecture zur Entity DFlipFlop. In der Port-Verbindungsliste werden die Ein- und Ausgangsports der Instanzen mit den Signals der Architecture SchiebWeg4 verbunden. Es wäre auch eine verkürzte Schreibweise der Verbindungsliste möglich, jedoch muß die Reihenfolge der Signals beachtet werden.

```
FF4 : entity FlipFlops.DflipFlop(behav)
      port map (clk_in, D3, data_out);
```

Aus dieser Schreibweise ergibt sich jedoch der Nachteil, daß die Verbindungen zu den Ports ohne genauer Kenntnis der Entity-Deklaration nicht sofort ersichtlich sind.

8.6 Instantiation mittels Generate

Die bekannte Art der Instantiation kann sehr aufwendig werden, wenn sehr viele Entities hintereinander geschaltet werden sollen, wie im nächsten Beispiel:



Das Generate-Statement wird in einer Architecture verwendet, um eine oder mehrere Instanzen eines Moduls (Entity/Architecture) zu generieren. Es kann an eine Bedingung geknüpft sein, oder innerhalb einer Schleife zum Generieren mehrerer Instanzen eingesetzt werden.

```
entity MuddFilter is
  port (Unfiltered : in integer;
        Filtered   : out integer);
end entity MuddFilter;

architecture Struct of MuddFilter is
  constant FilterDepth : integer := 10;
  constant DefuzzyfyIt : boolean := true;

  type BunchOfStageResults is array (integer range <>) of integer;
  signal FilteredThrough : BunchOfStageResults (1 to FilterDepth);

begin

  -- first filter stage
  FirstStage: entity Work.Filter(Somehow)
    port map (Noisy => Unfiltered, Calm => FilteredThrough(1));

  -- the next stages
  OtherStages: for Stage in FilteredThrough'left to
                            FilteredThrough'right-1 generate
  begin
    OneOtherStage: entity Work.Filter(SomehowElse)
      port map (Noisy => FilteredThrough(Stage),
                Calm => FilteredThrough(Stage+1));
  end generate OtherStages;

  -- defuzzyfyer
  MayDefuzzyfy: if DefuzzyzyIt generate
  begin
    Defuzzyfying: entity Work.Defuzzyfyer(Anyhow)
      port map (Fuzzy => FilteredThrough(FilteredThrough'length),
                Clean => Filtered);
  end generate MayDefuzzyfy;

  MayNotDefuzzyfy: if not (DefuzzyfyIt) generate
    ...
end architecture Struct;
```

In diesem Beispiel wird eine Filterkette erstellt. In dem Array `FilteredThrough` werden alle Ergebnisse der einzelnen Filterstufen zwischengespeichert. Zuerst wird die erste Filterstufe instantiiert und mit dem Port `Unfiltered` und dem ersten Element von `FilteredThrough` verbunden. Anschließend werden alle weiteren Filterstufen generiert und deren Ein- ind Ausgänge mit den Entsprechenden Elementen von `FilteredThrough` verbunden. Wenn die Konstante `DefuzzyfyIt` auf True gesetzt ist, wird noch ein Defuzzyfyer instantiiert.

8.7 Parametrisierung

Eine einmal entwickelte Entity soll möglichst universell einsetzbar sein. Wie im vorigen Beispiel ist es sinnvoll, die Anzahl der Filterstufen durch eine Konstante anzugeben, da bei einer Änderung der Anzahl von Filterstufen nur noch die Konstante geändert werden muss. Allerdings hat diese

Implementation noch einen großen Nachteil: Ändert sich die Anzahl der Filterstufen, so muss die Architecture neu kompiliert werden, und für jede Filterkettenlänge muss eine eigene Architecture erstellt werden.

VHDL ermöglicht durch Angabe eines `generic` die Entity bei ihrer instantiation zu parametrisieren. Das vorige Beispiel kann so abgeändert werden, dass die Filterkette universeller einsetzbar wird.

```
entity ParamMuddFilter is
  generic (FilterDepth : integer := 10;
           DefuzzyfyIt : boolean := true);
  port (Unfiltered : in integer;
        Filtered   : out integer);
end entity ParamMuddFilter;

architecture Struct of ParamMuddFilter is

  type BunchOfStageResults is array (integer range <>) of integer;
  signal FilteredThrough : BunchOfStageResults (1 to FilterDepth);

begin
  ...
end architecture Struct;
```

Um diese parametrisierbare Entity zu verwenden, müssen bei der Instantiation die generics ebenfalls zugewiesen werden (sofern sie nicht mit default-Werten versehen sind).

```
entity Equalizer is
  port (RawSound       : in integer;
        Volume         : in integer;
        EqualizedSound : out integer);
end entity Equalizer;

architecture Struct of Equalize is
  constant FilterStages : integer := 5;
  constant Defuzzyfication : boolean := false;
  signal ControlledRawSound : integer;
begin

  ControlVolume: process is
  begin
    ControlledRawSound <= RawSound * Volume;
    wait on RawSound, Volume;
  end process ControlVolume;

  Filtering: entity Work.ParamMudFilter(Struct)
    generic map (FilterDepth => FilterStages,
                 DefuzzyfyIt => Defuzzyfication);
    port map (Unfiltered => ControlledRawSound,
              Filtered => EqualizedSound);

end architecture Struct;
```

8.8 Instantiation mit Components

Eine Alternative zur Entity ist die `component`. Eine *component* bietet nach außen hin die selbe Funktionalität wie eine Entity. Hinter der Schnittstelle einer *component* verbirgt sich ein konkretes Entity/Architecture-Modul. Deshalb kann eine *component* genauso instanziert werden wie eine Entity. Das Beispiel SchiebWeg4 ist hier mittels *components* realisiert.

```
library FlipFlops;

entity SchiebWeg4 is
   port  ( clk_in   : in bit,
           data_in  : in bit,
           data_out : out bit);
end entity SchiebWeg4;

architecture behav of SchiebWeg4 is
   signal D1, D2, D3;
```

```
    component DFF is
        port ( clk  : in bit,
               d    : in bit,
               q    : out bit);

    for FF1,FF2,FF3,FF4 : DFF
        use entity FlipFlops.DflipFlop(behav);
    end for;
begin
    FF1 : component DFF
        port map (clk_in, data_in, D1);
    FF2 : component DFF
        port map (clk_in, D1, D2);
    FF3 : component DFF
        port map (clk_in, D2, D3);
    FF4 : component DFF
        port map (clk_in, D3, data_out);
end architecture SchiebWeg4;
```

Im block-declarative-item-Teil der Architecture befindet sich die component-declaration und im Anschluß daran die configuration-specification. Die configuration-specification legt fest, welche konkrete Entity/Architecture hinter der Component steckt. Fehlt die configuration-specification ganz oder teilweise, sucht der Default-Binding-Mechanismus des VHDL-Compilers eine passende Entity und Architecture. Gleichnamige Ports und Generics werden dabei miteinander verbunden, überzählige Ports/Generics der entity werden offen gelassen. Der Vorteil der Instantiation mittels *components* ist die leichte Austauschbarkeit der Konfigurationen (Entity/Architecture). Eine leichte Austauschbarkeit ist deshalb erwünscht, weil in den unterschiedlichen Phasen des Schaltungsentwurfes (Simulation, Synthese) verschiedene Konfigurationen benötigt werden. Die direkte Entity-Instantiation ist erst mit dem VHDL-93-Standard möglich, davor (VHDL-87) konnten Entitys nur in Form von *components* eingebettet werden.

9 Spracherweiterungen

9.1 Packages

Eine *package* ist eine Sammlung von oft benötigten Deklarationen. Der Inhalt einer *package* ist meist auf einen speziellen Anwendungsbereich zugeschnitten. In eine *package* können Typ- und Datendeklarationen, Funktionen, Prozeduren und Components integriert sein. Eine *package* besteht aus einer Declaration und einem optionalen *package-body*. Die *packages* werden zusammen mit Entity/Architecture-Modulen in Librarys abgelegt.

Die package-Deklaration ist die äußere Schnittstelle der *package*. Sie repräsentiert die externe Sicht.

```
package Package_Bezeichner is
  { package_declarative_item }
end [ package ] [ Package_Bezeichner ];
```

Das package-declarative-item ist dabei eine Sammlung von Declarations, die Types, Subtypes, Constants, Signals, Components, Funktions- und Procedure-Declarations beinhalten kann.

```
package Basic_Pkg is
    constant word_size : positive := 16;
    subtype  word is bit_vector (word_size   1 downto 0);
    subtype  doubleword is bit_vector (2*word_size   1 downto 0);
    procedure add (a, b     : in  word;
                   result   : out word;
                   overflow : out boolean);
end package Basic_Pkg;
```

In der *package*-Deklaration befinden sich von den Procedures und Functions nur die Declarations. Die Implementierung der Procedures und Functions ist aufgrund des Information-Hiding-Prinzips

von der externen Sicht auf eine *package* getrennt. Sie befindet sich daher im *package-body*.

```
package body Basic_Pkg is
  { package_body_declarative_item }
end [ package body ] [ Package_Bezeichner ];
```

```
Package body Basic_Pkg is
   procedure add (a, b      : in  word;
                  result    : out word;
                 overflow : out boolean);
begin
   procedure add (a, b      : in  word;
                     result    : out word;
                     overflow : out boolean) is
     variable sum : bit_vector (word_size downto 0);
   begin
         sum := a + b;
         result := sum;
         overflow := sum(word_size) = 1;
   end procedure add;

end package body Basic_Pkg;
```

Die Procedures und Functions müssen im Package-Body genauso wie in der zugehörigen Package-Declaration definiert werden.

Die in eine Library abgelegten Packages können in verschiedenen Varianten zur Implementierung verwendet werden. Wir gehen nachfolgend davon aus, daß die Package Basic_Pkg in die Library Work integriert wurde.

```
library work;
use work.Basic_Pkg.all;

entity Data_Transfer is
  port ( data_word : out word;
         address   : in  doubleword );
end entity Data_Transfer;
```

Hier werden alle in der Package Basic_Pkg der Library work enthaltenen Deklarationen eingebunden. Sie können genauso direkt, wie im aktuellen File deklarierte Types etc., verwendet werden.

```
library work;
use work.Basic_Pkg;

entity Data_Transfer is
  port ( data_word : out Basic_Pkg.word;
         address   : in  Basic_Pkg.doubleword );
end entity Data_Transfer;
```

In diesem Fall muß bei der Anwendung der Typen aus dem Package Basic_Pkg der Package-Bezeichner mit angegeben werden. Damit können Konflikte mit anderen Bezeichnern vermieden werden.

```
library work;
use work.Basic_Pkg.word, work.Basic_Pkg.doubleword;

entity Data_Transfer is
  port ( data_word : out word;
         address   : in  doubleword );
end entity Data_Transfer;
```

Die benötigten Typen werden selektiv aus der Package Basic_Pkg importiert. Sie können wie beim Beispiel 1 direkt verwendet werden.

9.2 Libraries

Bereits erstellte und getestete Entity/Architecture-Module und Packages können in einer *library* zusammengefaßt und der weiteren Verwendung in neuen Schaltungen zugeführt werden. Eine Standard-

Library ist die IEEE-Standard-Library, die vom *Institute of Electrical and Electronic Engineers* erstellt wurde. Am Markt sind unzählige zusätzliche VHDL-Librarys erhältlich. Selbst erstellte Module und Packages werden meist in einer *library work* abgelegt.

Die Library `std` mit der Package standard, in der alle wichtigen standard-Deklarationen enthalten sind, wird implizit in jeden VHDL-Entwurf eingebunden. Grundlegende Typen, Funktionen und Operatoren, wie z.B. integer, boolean, +, AND usw. müssen deshalb nicht explizit importiert werden.

Standard-Library IEEE Diese Library wurde vom *Institute of Electrical and Electronic Engineers* standardisiert. Sie enthält wichtige Spracherweiterungen. Eine davon ist die standardisierte Package std_logic_1164. Sie enthält eine erweiterte Form des Standardtyps bit.

9.3 Package: IEEE.std_logic_1164

Der Typ `bit` reicht zur Beschreibung von digitalen Schaltungen und deren Zuständen oft nicht aus. Beispielsweise kann ein Tristateausgang damit nicht nachgebildet werden (zusätzlicher hochohmiger Zustand). Das Package std_logic_1164 führt deshalb einen neuen Datentypen `std_ulogic` ein.

	Beschreibung
U	nicht initialisiert
X	Zustand unbekannt
0	logisch null
1	logisch eins
Z	hochohmig
W	schwach unbekannt
L	schwach null
H	schwach eins
-	dont care nicht beachten

Auch die logischen Grundoperatoren auf diesen neuen Datentyp werden im Package definiert (durch Overloading). Weitere Informationen dazu sind in nachfolgenden Kapiteln zu finden.

10 Concurrent Statements

VHDL-Modelle lassen sich durch die Verwendung nebenläufiger Anweisungen vereinfachen. Außerdem existiert zu den meisten sequentiellen Anweisungen ein nebenläufiges Pendant. Der Begriff Nebenläufigkeit bezeichnet hier sowohl die parallele Arbeit mehrerer Einheiten an einem Problem, als auch die Arbeit unabhängiger Einheiten an unterschiedlichen Aufgaben. Die Parallelität ist ein Spezialfall der Nebenläufigkeit. Im Gegensatz zur Verwendung von Prozessen, die selbst nebenläufige Anweisungen darstellen, erlauben spezielle nebenläufige Anweisungen die erhebliche Vereinfachung eines Modells. Solange ein Prozeß durch eine dieser nebenläufigen Anweisungen ersetzbar ist, kann auf die gesamte Prozessumgebung verzichtet werden.

10.1 Sensitivity List

Die Aktivierung eines Prozesses wurde bisher durch wait-statements realisiert:

```
DoIt : process
begin
   -- action
   ...
   -- end of action
   wait on A, B, C, D,...
end process;
```

Alternativ dazu kann im Prozesskopf die Sensibilisierung auf ein Signal in vereinfachter Form angegeben werden:

```
DoIt: process (A, B, C, D,...)
begin
  -- action
  ...
  -- end of action
end process;
```

10.2 Concurrent Signal Assignment

Vereinfachungen von *concurrent signal assignments* werden im folgenden Beispiel demonstriert.

```
process (a, b)
begin
  y <= a NAND b after 5ns;
end process;
```

Das obenstehende Beispiel zeigt einen Prozess, der eine sequentielle Signalzuweisung enthält. In Abhängigkeit von den Eingangssignalen a und b wird der Prozess aktiviert und die zugehörige NAND-Funktion berechnet. Diesen Prozess kann man durch eine nebenläufige Signalzuweisung ersetzen. Umgekehrt erzeugt der VHDL-Analysator als Vorbereitung für die Simulation aus jeder nebenläufigen Anweisung intern wieder einen Prozess. Dabei werden alle Signale der rechten Seite als sensitive Signale in eine wait- on-Anweisung aufgenommen.

```
y <= a NAND b after 5ns;
```

Diese Vereinfachung mindert nicht nur den Programmieraufwand, es erhöht sich auch die Lesbarkeit eines Modells durch die Verwendung von nebenläufigen Anweisungen beträchtlich.

10.3 Conditional Signal Assignment

Im Beispiel zum Concurrent-Signal-Assignment wurde eine Signalzuweisung vereinfacht. Es lassen sich aber auch andere Kontrollstrukturen direkt in eine Architecture einfügen. Dazu dienen die abhängigen (conditional signal assignment) und die ausgewählten (selected signal assignment) Signalzuweisungen.

```
process (s, a, b)
begin
  if s = 0 then y <= a after 5ns;
  else y <= b after 5ns;
  end if;
end process;
```

Das Beispiel zeigt zunächst einen Prozeß mit einer if-Anweisung (*conditional signal assignment*). Diese kann man durch eine abhängige Signalzuweisung wie folgt ersetzen:

```
y <= a after 5ns when s = 0
     else b after 5ns;
```

10.4 Selected Signal Assignment

Analog zum conditional signal assignment wird hier das *selected signal assignment* behandelt.

```
Multiplexer: process
begin
  case selector is
    when "00" => Out1 <= In0;
    when "01" => Out1 <= In1;
    when "10" => Out1 <= In2;
    when "11" => Out1 <= In3;
  end case;
  wait on selector;
end process;
```

Dieser Prozess (selected signal assignment) kann ersetzt werden durch:

```
with selektor select Out1 <= In0 when 00 ,
                             In1 when 01 ,
                             In2 when 10 ,
                             In3 when 11 ;
```

10.5 Concurrent Assertion Statement

Die nächste nebenläufige Anweisung, die hier behandelt werden soll, ist die Assertion-Anweisung (*concurrent assertion statement*). Sie stellt Informationen über den Simulationsverlauf zur Verfügung.

```
TestFF: process
begin
   assert (NOT R = 1 AND S = 1)
       report Error: S and R both 1;
       severity error;
wait on R, S;
end process;
```

Dieser Prozess kann durch die folgende Vereinfachung in einer Architecture ersetzt werden:

```
assert (NOT R = 1 AND S = 1)
   report Error: S and R both 1;
   severity error;
```

10.6 Concurrent Procedure Call

Auch ein Prozeduraufruf in einem Prozess kann durch eine nebenläufige Anweisung ersetzt werden (*concurrent procedure call*):

```
Calling: process begin
   CheckTiming (D, Clk, Reset, TSetup, THold);
   wait on D, Clk, Reset;
end process;
```

Nebenläufiges Pendant:

```
Calling: CheckTiming (D, Clk, Reset, TSetup, THold);
```

Funktionsaufrufe können nicht durch ein Concurrent-Statement vereinfacht werden.

11 Busse

11.1 Eingänge mit mehreren Treibern

Was passiert aber auf ein Signal mehrere Treiber wirken? Das heißt, daß dem Signal in konkurrenten Anweisungen (parallel ablaufenden Prozessen, konkurrenten Signalzuweisungen, konkurrenten Prozeduraufrufen usw.) Werte zugewiesen werden.

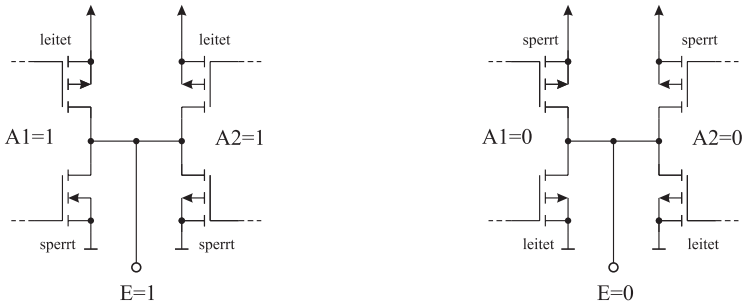
```
entity signaltest is
   port ( clk : in bit;
       a : out bit);
end;

architecture stupid of signaltest is begin
  Stupid1: a<='1';
  Stupid2: a<='0';
end stupid;
```

Versucht man dieses Beispiel zu compilieren, erzeugt der Compiler (in diesem Fall Modelsim) folgende Fehlermeldung:

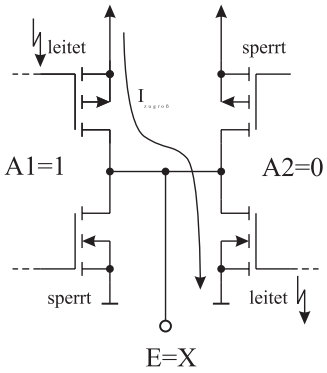
```
# - Loading package standard
# - Compile entity signaltest
# - Compiling architecture stupid of signaltest
# ERROR: mytsig.vhd(16): Nonresolved signal a already has a source (on line 9).
```

Standardmäßig werden Signale mit mehreren Treibern nicht aufgelöst. Der Grund dafür ist, daß sich abhängig von der verwendeten Schaltungstechnik, verschiedene logische Verhaltensformen ergeben, wenn mehrere Treiber den Wert eines Signals bestimmen.

Betrachtet man als Schaltungstechnik gebräuchliche Ausgangsschaltungen mit Gegentakt- Endstufe, ergeben sich zunächst bei gleichem Zustand der Ausgangstreiber A1 und A2 keine Probleme. Der Eingang E nimmt den Zustand der Ausgänge an.



Haben jedoch die Ausgangstreiber unterschiedliche Zustände, wird der schwächere der leitenden Transistoren wegen zu große Ströme seine Funktion für immer verweigern, und somit ist der Zustand des Eingangs unbestimmt.



Bei anderen Schaltungstechniken können sich andere logische Verhaltensformen ergeben, z.B bei Open-Collector Ausgänge mit Pullup-Widerstand.

11.2 Resolution Funktion

Durch die verschiedenen Schaltungstechniken entsteht die Notwendigkeit eines Mechanismus, mit dem die Regeln zur Auflösung von Signalen mit mehreren Treibern beschreibbar sind. Dieser Mechanismus ist in VHDL durch *resolution functions* gegeben.

```
signal_name: resolution_funktion_name type_name;
subtype subtype_name is resolution_funktion_name type_name;
```

Ein aufzulösendes Signal kann über eine Resolution-Funktion deklariert werden, oder man er-

zeugt einen Datentyp (subtype) der mit einer Resolution Funktion verbunden ist. Wird dem Signal signal_name, oder einem Signal vom Typ subtype_name, ein Wert zugewiesen, wird implizit die mit dem Signal oder dem Typ verbundene Resolution Funktion aufgerufen und der Wert des Signals berechnet.

Resolution Funktions werden wie normale Funktionen deklariert. Als Argument der Funktion wird ein Array variabler Länge mit Elementen vom Typ typ_name übergeben, die Funktion liefert einen Wert vom Typ typ_name zurück.

In den beiden nachfolgenden Beispielen wird das logische Verhalten der untersuchten Schaltungstechniken durch die Datentypen `WIREDAND_logic` und `BIT3_logic` beschrieben.

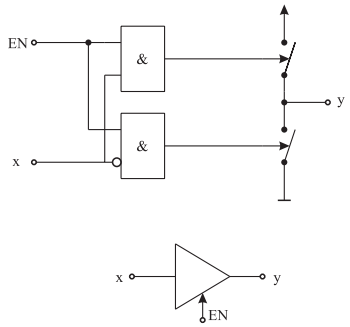
```
Package WIREDAND is
   function resolve_WIREDAND (insignal: bit_vector)
            return bit;
   subtype WIREDAND_logic is resolve_WIREDAND bit;
end package WIREDAND;
package body WIREDAND is
   function resolve_WIREDAND (insignal: bit_vector) return bit is
       variable result: bit := '1';
   begin
       for i in insignal'range loop
           result:=result and insignal(i);
       end loop;
       return result;
   end function resolve_WIREDAND;
end package body WIREDAND;

use work.WIREDAND.all;

entity signaltest is
   port ( clk : in bit;
          a   : out WIREDAND_logic);
end;

architecture wiredandtest of signaltest is
begin
  Test1:     a<='1';
  Test2:     a<='0';
end wiredandtest;
```

```
package BIT3 is
   type BIT3_ulogic is ('X','0','1');
   type BIT3_ulogic_vector is array (natural range <>) of BIT3_ulogic;
   function resolve_BIT3 (insignal: BIT3_ulogic_vector)
            return BIT3_ulogic;
   subtype BIT3_logic is resolve_BIT3 BIT3_ulogic;
end package BIT3;

package body BIT3 is
   function resolve_BIT3 (insignal: BIT3_ulogic_vector)
       return BIT3_ulogic is
       variable result: BIT3_ulogic ;
   begin
       result:=insignal(insignal'low);
       for i in insignal'range loop
           if result /= insignal(i) then result := 'X';
           end if;
       end loop;
       return result;
   end function resolve_BIT3;
end package body BIT3;

use work.BIT3.all;
entity signaltest is
   port ( clk : in bit;
          a   : out BIT3_logic);
end;

architecture bit3test of signaltest is
begin
  Test1:     a<='0';
  Test2:     a<='1';
```

```
end bit3test;
```

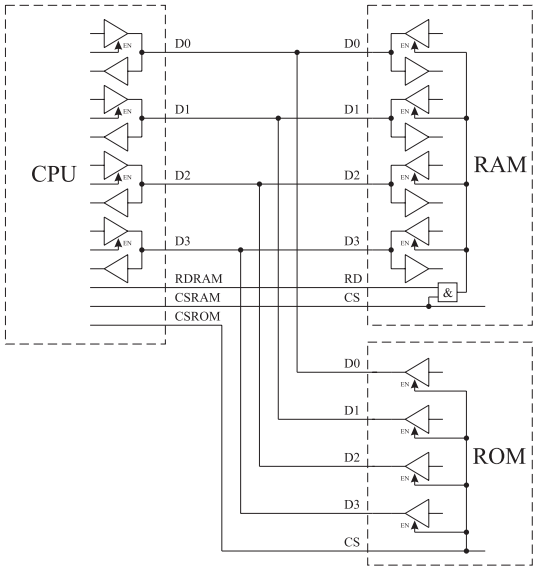
11.3 Busse

Ein wichtiger Anwendungsfall bei dem die Parallelschaltung von Ausgängen (konkurrente Signalzuweisungen) zu einer Schaltungsvereinfachung führt ist, wenn wahlweise einer von mehreren Ausgängen den Zustand eines Signals bestimmt. Man spricht dann von einem Bus-System. Dafür wird ein weiterer logischer Zustand benötigt, der hochohmige Z-Zustand.



Bei dieser Schaltungstechnik mittels *tristate*-Ausgänge, gibt es zwar dieselben Probleme, wenn mehrere Ausgänge unterschiedliche 0,1 Zustände haben, wie bei der mit BIT3_logic beschriebene Schaltungstechnik. Befinden sich aber alle bis auf einen Ausgang (treibender Ausgang) im Z-Zustand, bestimmt dieser Ausgang den Wert des Eingangs. Im folgendem Beispiel wird das logische Verhalten dieser Schaltungstechnik durch den Datentyp BIT4_logic beschrieben.

```
package BIT4 is
   type BIT4_ulogic is ('X','0','1','Z');
   type BIT4_ulogic_vector is array (natural range <>) of BIT4_ulogic;
   function resolve_BIT4 (insignal: BIT4_ulogic_vector)
            return BIT4_ulogic;
   subtype BIT4_logic is resolve_BIT4 BIT4_ulogic;
end package BIT4;

package body BIT4 is
   type BIT4_tabelle is array (BIT4_ulogic, BIT4_ulogic) of BIT4_ulogic;
   constant tabelle: BIT4_tabelle := -- 'X','0','1','Z'
                                     (('X','X','X','X'),   --   'X'
                                      ('X','0','X','0'),   --   '0'
                                      ('X','X','1','1'),   --   '1'
                                      ('X','0','1','Z'));  --   'Z'
   function resolve_BIT4 (insignal: BIT4_ulogic_vector) return BIT4_ulogic is
       variable result: BIT4_ulogic:='Z';
   begin
       for i in insignal'range loop
           result := tabelle(result,insignal(i));
       end loop;
       return result;
   end function resolve_BIT4;
end package body BIT4;
```



Dieses Beispiel zeigt anhand eines 4-Bit Datenbus die Funktionsweise eines Bus-Systems. Zur Vereinfachung wurden die Adressleitungen weggelassen. Die wichtigste Eigenschaft eines Bus- Systems ist, daß nur eine Komponente (CPU, RAM, ROM) auf den Bus treiben darf, ansonsten könnte ein undefinierter Bus-Zustand entstehen.

Soll die CPU aus dem RAM lesen, müssen sich die Ausgangstreiber D0-D3 der CPU und des ROMs im hochohmigen Zustand "Z" befinden. Nur das RAM darf auf den Datenbus treiben.(CPU D0-D3="Z", RDRAM="1", CSRAM="1", CSROM="0" ⇒ ROM D0-D3=Z)

Soll die CPU aus dem ROM lesen, müssen sich die Ausgangstreiber D0-D3 der CPU und des RAMs im hochohmigen Zustand "Z" befinden. Nur das ROM darf auf den Datenbus treiben.(CPU D0-D3="Z", CSRAM="0" ⇒ RAM D0-D3="Z", CSROM="1")

Soll die CPU ins RAM schreiben, müssen sich die Ausgangstreiber D0-D3 des RAMs und des ROMs im hochohmigen Zustand "Z" befinden. Nur die CPU darf auf den Datenbus treiben.(RDRAM="0", CSRAM="1" ⇒ ROM D0-D3="Z", CSROM=0-¿ROM D0-D3="Z")

11.4 std_logic und std_ulogic

In dem Package IEEE.std_logic_1164 wird durch die Datentypen `std_logic` und `std_ulogic` ein Logiksystem mit neun Signalwerten, bzw. Treiberstärken definiert, das für die Simulation und Synthese von Hardware besser geeignet ist als der Standardtyp Bit.

`std_ulogic` ist ein nicht-aufgelöster Datentyp (ulogic=unresolved logic). Der aufgelöste Datentyp `std_logic` ist mit einer Auflösungsfunktion verbunden, und kann deshalb für Signale mit mehreren Treibern verwendet werden. Die Auflösungstabelle für `std_logic` ist folgendermaßen definiert:

	U	X	0	1	Z	W	L	H	-
U	U	U	U	U	U	U	U	U	U
X	U	X	X	X	X	X	X	X	X
0	U	X	0	X	0	0	0	0	X
1	U	0	X	1	1	1	1	1	X
Z	U	X	0	1	Z	W	L	H	X
W	U	X	0	1	W	W	W	W	X
L	U	X	0	1	L	W	L	W	X
H	U	X	0	1	H	W	W	W	X
-	U	X	0	X	X	X	X	X	X

In Beispiel BIT4 wurde bereits gezeigt, wie Auflösungstabellen in Auflösungsfunktionen verwendet werden. In der Auflösungstabelle ist zu sehen, daß die schwach logischen Zustände (W, L, H) von den treibenden Zuständen (X, 0, 1) dominiert werden. So könnte beispielsweise die vorher beschriebene Schaltungstechnik Wired-AND auch mittels `std_logic` beschrieben werden:

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

entity signaltest is
   port (  clk : in bit;
           a   : out std_logic);
end;

architecture wiredandtest of signaltest is
begin
  Test1: a<='H';     -- schwach logisch 1 wegen Pullup-Widerstand
  Test2: a<='0';     -- dominiert schwach logisch 1

end wiredandtest;
```

12 Weiteres

Einer der Hauptgründe für das Beschreiben eines Modells in VHDL ist, das Verhalten des Modells simulieren zu können. Dafür werden die folgenden drei Phasen durchlaufen. Die Analyse und Elaboration sind außerdem notwendig, um ein Model für die Synthese vorzubereiten.

12.1 Schritte zur Simulation

In der Analyse-Phase wird die Modellbeschreibung auf syntaktische und semantische Korrektheit überprüft. Wobei nicht das ganze Modell analysiert werden muß, sondern es auch möglich ist, einzelne Entities und Architectures zu analysieren. Wurde eine Entity oder Architecture fehlerfrei analysiert, wird diese in einer Library gespeichert.

In der Elaborations-Phase werden die Modelteile im Speicher des Simulationsrechners erzeugt und miteinander verknüpft. Dafür wird das Modell beginnend mit der Hauptkomponente (das ist die Komponente, in die alle anderen Komponenten direkt oder indirekt eingebunden sind) ausgewertet (elaboriert). Es werden alle Signale (einschließlich der Port-Signale) erzeugt und miteinander verknüpft, die in der Komponente direkt enthaltenen Prozesse (Parallelzuweisungen) angelegt und mit den Signalen verknüpft, und alle (direkt) eingebundenen Komponenten elaboriert.

Nach der Elaboration kann die Simulation des Modells durchgeführt werden. In VHDL werden nur diejenigen Zeitpunkte simuliert, bei denen Veränderungen im Modell stattfinden können. Die simulierte Zeit schreitet dementsprechend während der Simulation nicht kontinuierlich sondern in Sprüngen fort. Dieses Simulationsprinzip wird diskrete Event-Simulation genannt (DEVS).

12.2 Overloading

Overloading bezeichnet, daß mehrere Unterprogramme (procedure, function, operator) mit gleichem Namen definiert sind, aber unterschiedliche Parametertypen, beziehungsweise unterschiedlich viele Parameter haben. Bei Verwendung dieser Unterprogramme wird dann, entsprechend Anzahl und Typ der Argumente, die entsprechende Funktion/Procedure ausgewählt.

```
function add (x1: integer, x2: integer) return integer is ...
function add (x1: real, x2: real) return real is ...
function add (x1: integer, x2:integer, x3: integer) return integer is ...
function add (x1: real, x2:real, x2: real) return real is ...
```

Mittels Overloading lassen sich auch bestehende Operatoren (not,and,or..) aus dem Standard-Package erweitern. So werden beispielsweise für die in IEEE.std_logic_1164 bereitgestellten Typen `std_logic` und `std_ulogic` Standardoperatoren neu definiert. Durch das Overloading können diese dann vom Benutzer wie gewohnt verwendet werden. Das folgende Beispiel zeigt, wie in std_logic_1164 der not-Operator für std_ulogic-Typen neu definiert wird:

```
TYPE stdlogic_1d IS ARRAY(std_ulogic) OF std_ulogic;
CONSTANT not_table: stdlogic_1d :=
    -- ---------------------------------------------------
    --  |  U    X    0    1    Z    W    L    H    -   |
    -- ---------------------------------------------------
        ('U', 'X', '1', '0', 'X', 'X', '1', '0', 'X');

FUNCTION "not" ( l : std_ulogic_vector ) RETURN std_ulogic_vector IS
        ALIAS lv : std_ulogic_vector ( 1 TO l'LENGTH ) IS l;
        VARIABLE result : std_ulogic_vector ( 1 TO l'LENGTH ) := (OTHERS => 'X');
    BEGIN
        FOR i IN result'RANGE LOOP
            result(i) := not_table( lv(i) );
        END LOOP;
        RETURN result;
    END;
```

Weiters dürfen verschiedene Aufzählungstypen gleiche Literale beinhalten, jedoch muß bei Verwendung durch den Kontext eindeutig bestimmt sein, welchem Aufzählungstyp das verwendete Literal zugeordnet ist.

```
type auslastung is (schwach, mittel, stark, unbestimmt);
type fehler is (warnung, kritisch, unbestimmt);
variable busstatus: auslastung;

busstatus:=unbestimmt;   -- eindeutig -> busstatus ist vom Typ auslastung
```

12.3 Package std.textio

Durch das Package `std.textio` ist eine formatierte ASCII-Ausgabe in Dateien, und Eingabe aus ASCII-Dateien möglich. So können beispielsweise Testvektoren aus einer Datei eingelesen, und in der Simulation verwendet werden, oder Simulationsergebnisse in einer Datei gespeichert werden, die gegebenenfalls für weitere Applikationen oder Simulationen als Eingabe dienen. In `std.textio` werden unter anderem die Datentypen `text` und `line` deklariert, sowie die Funktion `readline` für das Lesen von Textzeilen aus Textdateien, und `writeline` zum Schreiben in Textdateien:

```
type line is access string;
type text is file of string;

procedure readline (file f: text; l: out line);
procedure writeline (file f: text; L: inout line);
```

Eine Reihe von read- und write-Prozeduren erlauben das Bearbeiten der Textzeilen. Mittels Overloading werden read/write-Prozeduren für die Datentypen bit, bit_vector, boolean, character, integer, real, string und time bereitgestellt. Beispielsweise stehen folgende Prozeduren zum Lesen und Schreiben von integer-Typen zur Verfügung:

```
procedure read (L: inout line; value: out integer;);
procedure read (L: inout line; value: out integer; good: out boolean);

procedure write (L: inout line; value : in integer; justified : in side :=
                 right; field : in width := 0);
```

12.4 Allgemeinere Literale

Die Zahlendarstellung ist in VHDL standardmäßig dezimal. Für die Darstellung anderer Zahlenbasen, muß die Basis explizit angegeben werden. (based literals)

```
integer#based_integer[.based_integer]#[E[+]integer|-integer]
based_integer<-(digit|letter){[_]...}
```

```
variable test: integer;
test:=10;              -- gleiche Zuweisungen in unterschiedlicher
test:=2#1010#;         -- Basisdarstellung
test:=16#A#;
```

Ebenso sind für Bit strings Darstellungen mit unterschiedlicher Basis möglich.

```
(B|O|X) "␣[␣(digit|letter){[_]...}]␣"
```

```
variable test: bit_vector(0..3);
test:= 1010 ;          -- gleiche Zuweisungen in unterschiedlicher
test:=X A ;            -- Basisdarstellung
```

Literatur

[Ash96] P. J. Ashenden. *The Designer's Guide to VHDL.* Morgan Kaufmann Publishers, San Francisco, 1996.

[KG75] W. Waiter K. Ganzitom. *Die Geschichtliche Entwicklung der Datenverarbeitung.* IBM, Sindelfingen, 5. auflage edition, 1975.

[Zus86] K. Zuse. *Der Computer. Mein Lebenswerk.* Berlin, Heidelberg, 2. auflage edition, 1986.