Praktikum
Grundlagen von Hardwaresystemen
Wintersemester 2016

Versuch 2: VHDL-Einstieg

27. Oktober 2016

Fachbereich 12: Informatik und Mathematik
Institut für Informatik
Professur für Eingebettete Systeme
Prof. Dr. Uwe Brinkschulte
Andreas Lund

Johann Wolfgang Goethe-Universität
Frankfurt am Main

Inhaltsverzeichnis

1 Einleitung

Dieser Versuch bietet eine Einführung in die Vorgehensweise bei der Entwicklung eines digitalen Systems in VHDL und behandelt die verschiedenen Möglichkeiten wie eine passende Architektur zu einer (Teil-)Komponente erstellt werden kann. Es werden sowohl die Eigenschaften der algorithmischen, Datenfluss- und Strukturbeschreibung behandelt als auch das generelle Methodik mit der ein komplexes System in VDHL entsteht. Mit dem Wissen aus dem Kapitel Grundlagen und eingen inhaltlichen Anleihen aus der HWR Vorlesung sollen dann folgende Aufgaben gelöst werden:

- Datenflussbeschreibung: Minimierung einer logischen Funktion auf einen möglichst kurzen boolschen Ausdruck und anschließende Implementierung.
- Strukturbeschreibung: (Grafische) Komposition eines Schaltelements aus mehreren einfacheren Elementen.
- Algorithmische Beschreibung: Erstellen der Architektur eines komplexeren Schaltelements mit den Methoden einer höheren Programmiersprache. In der Literatur wird die algorithmische Beschreibung auch häufig als Verhaltensbeschreibung bezeichnet.

2 Grundlagen

Beim Entwurf komplexer Systeme gibt es grundsätzlich zwei verschiedene Vorgehensweisen: das Top-down Prinzip und das Bottom-up Prinzip. Beim Top-down Verfahren beginnt man mit dem übergeordneten, abstrahiertem Gesamtkonzept und unterteilt es in mehrere weniger komplexe Komponenten, für die entweder bereits eine Lösung existiert, oder die ebenfalls nach dem Top-down Verfahren weiter behandelt werden. Beim Bottom-up Verfahren werden bekannten einfachen Strukturen so lange miteinander kombiniert und komplexere Strukturen gebildet bis die gewünschte Funktionalität des Gesamtsystem erreicht ist.
Wie aus dem erstem Versuch bekannt, werden die zu einem System gehörenden Schnittstellen für die Ein- und Ausgabe in VHDL durch die Entity definiert. Das Verhalten eines Systems wird in seiner Architektur festgelegt. Dabei gibt es keine Einschränkungen was die Komplexität des Systems angeht. Komplexe Systeme lassen sich am schnellsten durch eine algorithmische Beschreibung umsetzten, da in VHDL Konstrukte von Hochsprachen verwendet werden können. So lassen sich möglichst schnell und kompakt sowohl parallele als auch sequenzielle Abläufe beschreiben.
Wie den Studenten bereits aus Versuch 1 Aufgabe 6 bekannt sein dürfte entspricht bei der algorithmischen Beschreibung das Verhalten gerade in zeitlicher Hinsicht nicht immer genau dem was man vom Algorithmus erwartet. Ein großer Unterschied zu anderen Programmiersprachen besteht darin das Signalzuweisungen eben nicht unmittelbar gültig sind und spätere Operatoren im Prozess mit anderen Signalbelegungen arbeiten als VHDL Anfänger annehmen.
Theoretisch können auch sehr große und komplex Systeme wie zum Beispiel ein Mikrocontroller komplett in einer einzelnen Architektur algorithmisch beschrieben werden. Da solch ein monolithischer Ansatz schnell unübersichtlich wird und eine Aufteilung auf mehrere Entwickler nicht erlaubt, wird wie beim Top-down Prinzip beschrieben das Gesamtsystem in mehrere kleinere Unterkomponenten aufgeteilt, diese separat entwickelt und dann miteinander vernetzt.
Das Vernetzten mehrere kleinere Komponenten zu einer Größeren geschieht mit Hilfe der Strukturbeschreibung. Vereinfacht beschrieben müssen dafür nur die verwendeten Komponenten und die Vernetzung der entsprechenden Entitys aufgelistet werden.
Ob die eingebundenen Unterkomponenten dabei selbst durch eine Strukturbeschreibung zusammengesetzt wurden oder direkt durch einen Algorithmus oder als Datenflussbeschreibung implementiert wurden ist nicht von Bedeutung, da eine Entity nicht festlegt wie ihre Architektur aufgebaut ist und auch verschiedene Architekturen zu einer Enitity erlaubt sind.
Zusammenfassend lässt sich also sagen, dass es drei verschiedene Methoden gibt eine Architektur zu erstellen:

- Bei der algorithmischen Beschreibung wird das Verhalten direkt mit den Befehlen einer Hochsprache formuliert.
- Bei der Strukturbeschreibung werden bereits vorhandene Komponenten (welcher Architektur auch immer) miteinander vernetzt, um eine eine neue komplexere Architektur zu bilden.
- Bei der Datenflussbeschreibung werden einfache logische Ausdrücke direkt auf die Eingangssignale einer Architektur angewendet um die Ausgangswerte zu erzeugen.

3 Wiederholung: VHDL Grundlagen

3.1 Signale

Signale sind ein weiteres zentrales Element von VHDL. Ähnlich wie Variablen in anderen Programmiersprachen sind Signale in VHDL die Träger von Information. Nach einer Signalzuweisung nimmt das Signal einen bestimmten Wert an. Natürlich kann das Signal auf den augenblicklichen Wert abgefragt werden oder mit anderen Signalen und geeigneten Operatoren verknüpft werden.

Vergleichbar mit einem Stück Draht verbindet ein Signal Ausgang und Eingang zweier Teilschaltungen (z. B. Gatter, Flip-Flop etc.). Doch nicht jedes Signal ist nach der Synthese auch tatsächlich als Verbindungsleitung vorhanden. Während der Synthese können Signale als Folge einer Optimierung verschwinden oder neue hinzukommen.

Jedes Signal ist von einem eindeutig zu definierenden Typ und besitzt einen eindeutigen Namen. Zur Einführung soll hier zunächst nur der Signaldatentyp `std_logic` bzw. `std_logic_vector` benutzt werden, welcher den Anwendern aus dem std_logic_1164 Paket der Bibliothek IEEE zur Verfügung steht. Das Paket std_logic_1164 stellt 9-wertige Logik dar, kann aber auch 3-, 4- und 5-wertige Logik realisieren. Das heißt, dass der Wertevorrat des Signaldatentyps `std_logic` aus neuen logischen Werten besteht. Für das Praktikum sind folgende vier Werte relevant: 0, 1, U, X, wobei 1 für die logische Eins, 0 für die logische Null, U für nicht initialisiert und X für unbekannt (z.B. durch einen Kurzschluss) steht.

Die Belegungstabelle eines Inverters mit 4-wertiger Logik:

U	X	0	1
U	X	1	0

Die Belegungstabelle eines UND-Gatters mit 4-wertiger Logik:

	U	X	0	1
U	U	U	0	U
X	U	X	0	X
0	0	0	0	0
1	U	X	0	1

Die Belegungstabelle eines ODER-Gatters mit 4-wertiger Logik:

	U	X	0	1
U	U	U	U	1
X	U	X	X	1
0	U	X	0	1
1	1	1	1	1

Die IEEE-Library ist sehr weit verbreitet. Abhängig von der verwendeten Entwurfsumgebung können auch andere Bibliotheken mit anderen Bezeichnern verwendet werden, die man von einem Hersteller erhalten oder die man selbst geschrieben hat.

Im Quellcode können diese Datentypen verwendet werden, wenn *vor* der `entity`-Deklaration die IEEE-Bibliothek mit Hilfe einer `library`-Anweisung deklariert wird. Zusätzlich muss durch die `use`-Anweisung angegeben werden, dass alle Komponenten des `std_logic_vector_1164` Pakets verwendet werden sollen.

```
Library IEEE;
use IEEE.std_logic_1164.all;
```

Signale können an folgenden Stellen im VHDL-Code deklariert werden:

- als Port der `entity`-Deklaration für entity-globale Signale.
- innerhalb einer `architecture` als Architektur-globale Signale. Im Allgemeinen werden alle Signale, die keine Ports sind, so deklariert.

Syntax eines Signals
`signal` <Signalname> : <Datentyp>;

Für den Modus der Ein-/Ausgänge einer Portdeklaration gilt:

1. `in:` Eingang, nur auf rechter Seite von Variablen-/Signalzuweisungen, also in Ausdrücken, zulässig.
2. `out:` Ausgang, nur auf linker Seite von Signalzuweisungen zulässig.
3. `inout:` bidirektionale Leitung, kann im Code lesend und schreibend benutzt werden.

3.2 Busse

Ein `std_logic_vector` stellt einen aus mehreren `std_logic` Signalen bestehenden Bus dar. Dieser Bus kann entweder aufsteigend, z.B. als `std_logic_vector(0 to 7)` oder aber abfallend als `std_logic_vector(7 downto 0)` bezeichnet werden.
Beispiele:

- *testbus_1 : in std_logic_vector (7 downto 0);*
  Damit ist sowohl die Signalrichtung als auch die Busbreite, Typ und Format definiert.
- *testbus_2 <= "00000001";*
  Abhängig davon, wie der Bus in der Entity deklariert wurde, bedeutet diese Zuweisung eine 1 oder eine 128.
- Es sind auch Zuweisungen auf einen Teil des Bus möglich:
  testbus_3 (0 to 3) <= "0101";
  Oder auf eine einzelne Signalleitung:
  testbus_3 (5) <= '1';

4 Wiederholung: VHDL Code-Beispiele

Verhalten: Die Grundstruktur der algorithmischen Beschreibung ist der ***Prozess***.

Syntax eines Prozesses:

<Prozessname> : `process` (<Sensitivitätsliste>)
  <Deklarationsteil>
begin
  <sequentielle-Anweisungen>
end process <Prozessname>;

Listing 1: Verhaltensbeschreibung eines NAND-Gatters

```
architecture behavior of NAND_gate is
begin
  P1:process(a,b)
  begin
    if (a = '1') and (b = '1') then
      c <= '0';
    else
      c <= '1';
    end if;
  end process P1;
end behavior;
```

Ein VHDL-Prozess ist einem sequenziellen Task einer Programmiersprache vergleichbar, mit den bekannten Konzepten:

- sequentielle Abarbeitung der Anweisungen
- Kontrollanweisungen zur Steuerung des Ablaufs
- Verwendung lokaler Variablen
- Unterprogrammtechniken (Prozeduren und Funktionen)

In der Sensitivitätsliste tauchen all die Signale auf, deren Änderung die Abarbeitung des Prozesses auslösen.

Datenfluss: Bei dieser Beschreibung wird der Datenfluss über einfache logische Funktionen modelliert.

Listing 2: Datenflussbeschreibung eines NAND-Gatters

```
architecture dataflow of NAND_gate is
begin
  c <= not ( a and b );
end dataflow;
```

Struktur: Strukturbeschreibungen sind Netzlisten aus Bibliothekselementen. Diese Elemente werden instanziiert und über Signale miteinander verbunden. Im Beispiel werden wir NAND-Gatter aus AND und NOT aufbauen.

Listing 3: Strukturbeschreibung eines NAND-Gatters

```
Library IEEE;
use IEEE.std_logic_1164.all;

entity And_gate is
  port (
    in0, in1 : in  STD_LOGIC;
    out0     : out  STD_LOGIC
  );
end And_gate;

architecture dataflow of And_gate is
begin
  out0 <= in0 and in1;
end dataflow;

Library IEEE;
use IEEE.std_logic_1164.all;

entity Inverter is
  port (
    in0  : in  STD_LOGIC;
    out0 : out  STD_LOGIC
  );
end Inverter;

architecture dataflow of Inverter is
begin
  out0 <= not in0;
end dataflow;

-- entity der Schaltung

Library IEEE;
use IEEE.std_logic_1164.all;

entity NAND_gate is
  port (
  a, b : in  STD_LOGIC;
  c    : out STD_LOGIC
  );
end NAND_gate;
```

```
-- Strukturelle Beschreibung der Schaltung

architecture structure of NAND_gate is

component And_gate
  port (
    in0, in1 : in  STD_LOGIC;
    out0     : out  STD_LOGIC
  );
end component;

component Inverter
  port (
    in0  : in  STD_LOGIC;
    out0 : out  STD_LOGIC
  );
end component;

signal And_out0, Inverter1_out0 : STD_LOGIC;

begin
  And_2 : And_gate port map (a, b, And_out0);
  Inverter1 : Inverter port map (And_out0, Inverter1_out0);

  -- Signal mapping
  c <= Inverter1_out0;
end structure;
```

Alternativ lassen sich Strukturbeschreibung auch automatisch generieren indem mehrere Komponenten in einem Editor mit grafischer Oberfläche miteinander verknüpft werden. Dies wird ausführlich im nächsten Kapitel erklärt und soll in Aufgabe 3 als Leitfaden dienen. Trotzdem empfehlt es sich das Kapitel vorzubereiten.

5 Grafische Vernetzung von Komponenten

Beim modernen Schaltungsdesign beschreibt man den Entwurf hauptsächlich algorithmisch, und extrahiert aus dieser Beschreibung mittels einer automatischen Synthese eine strukturelle Beschreibung mit Gattern, Flipflops und Registern, die dann weiterverarbeitet werden kann. Es gibt aber durchaus noch Gründe für eine strukturelle Beschreibung in VHDL:

- zur Verbindung von großen Bausteinen, die algorithmisch beschrieben sind. Heutzutage werden solche Bausteine auch als sog. IP-Cores (intellectual property cores) gehandelt, d.h. Chiphersteller kaufen bestimmte Komponenten für ihren Entwurf als VHDL-Beschreibung von Drittanbietern.
- bei *regulären Strukturen* wie z.B. Speichern ist eine strukturelle Beschreibung durchaus sinnvoll. Um solche großen Strukturen zu beschreiben, bietet das ISE Webpack das *Schemata*-Konzept, mit dem sich strukturelle Beschreibungen, bestehend aus beliebig vielen Einzelkomponenten, einfach per Drag & Drop erzeugen lassen.
- es kann sein, dass algorithmische Konstrukte vom Synthesewerkzeug nicht oder nur ungenügend umgesetzt werden, so dass man hier auf eine bessere selbstdefinierte strukturelle Beschreibung zurückgreifen muss.

Für die Erzeugung komplexerer und aufwendiger struktureller Beschreibungen, bieten VHDL und die Entwicklungsumgebung ISE Webpack zwei Konzepte an. Ziel dieser ist es, eine möglichst einfache Erzeugung der Netzliste (Instanziierung und Vernetzung der Komponenten/Entities), also die eigentliche Beschreibung der strukturellen Zielarchitektur, dem Entwickler bereitzustellen. Diese sind:

- `generate`-Anweisung
- `Schemata` (Schaltsymbole)

Die `generate`-Anweisung ist ein Konzept von VHDL selbst und wird in diesem Praktikum aufgrund des zeittechnischen Aufwands und der Komplexität nicht betrachtet. Dennoch ist das Prinzip und die Funktionsweise die, dass durch die Verwendung von Schleifen eine parametrisierte iterative Instanziierung der Komponenten durchgeführt werden kann.

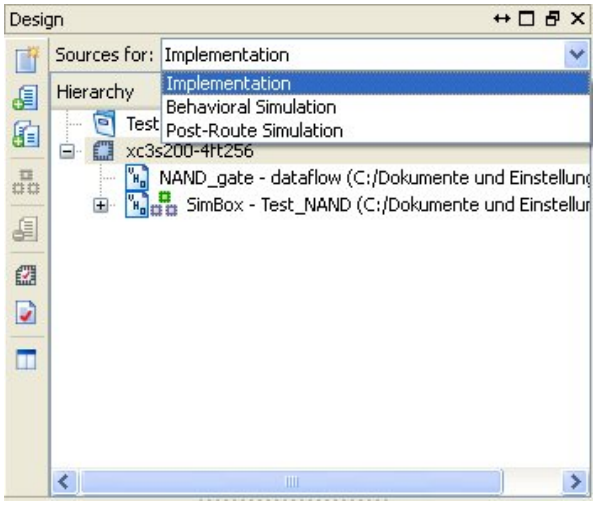
Das zweite Konzept ist Teil der Entwicklunsgumbegung. Im ISE Webpack ist es möglich eine visualisierte Darstellung der VHDL Entwürfe in Form von `Schemata` zu erzeugen. Schemata sind, auf diesen Kontext bezogen, technische Schaltsymbole die den jeweiligen Entities zugeordnet sind. Unter Verwendung dieser Symbole kann der Entwickler den Entwurf dann visuell per Drag & Drop aus den jeweiligen Schaltsymbolen erstellen. Der zugehörige VHDL-Quellcode wird automatisch über die verknüpften Komponenten/Entites generiert, welcher später auch eingesehen und verwendet werden kann.

5.1 Verwendung von Schemata im ISE Webpack

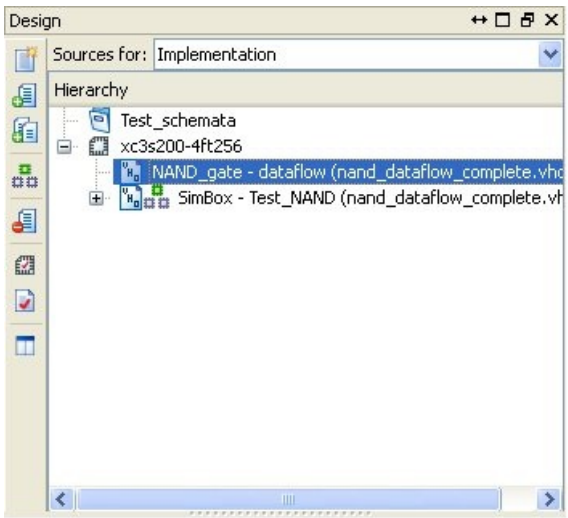
Um Ihnen das Prinzip einfach zu verdeutlichen, wird hier ein Beispiel ausgeführt, welches die strukturelle Beschreibung eines XOR-Gatters basierend, auf vier NAND-Gattern verwendet.

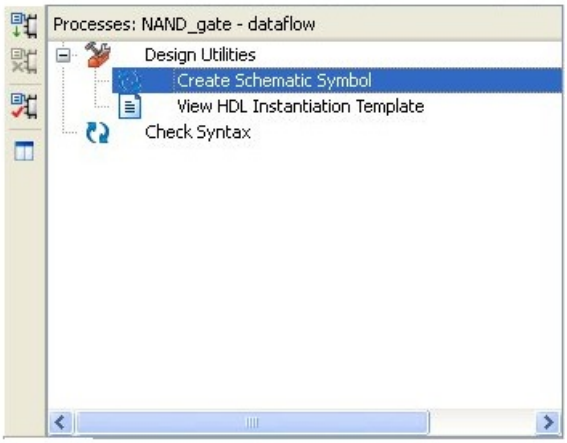
Bevor Sie Entities schematisch im ISE Webpack visuell per Drag & Drop instanziieren und verschalten können, müssen zunächst für die benötigten Entities die zugehörigen Schaltsymbole (Schematic Symbols) erzeugt werden. Dazu gehen Sie wie folgt vor:

- Im Drop Down Menü `Sources for:` des Design-Fensters den Vorgang `Implementation` auswählen.



- Auswahl der entsprechenden Entity im Design-Fenster. Dabei ist es wichtig, dass sich die Datei der Entity im Verzeichnis des aktuellen Projekts befindet. Sollte die Entity nicht bereits Teil des Projekts sein, so fügen Sie diese dem Projekt zuvor hinzu (Rechtsklick auf FPGA-Design (xc3s200) → `Add Copy of Source`). Die Erzeugung des Schaltsymbols starten Sie im Prozess-Fenster unter dem Punkt `Design Utilities` → `Create Schematic Symbol`.

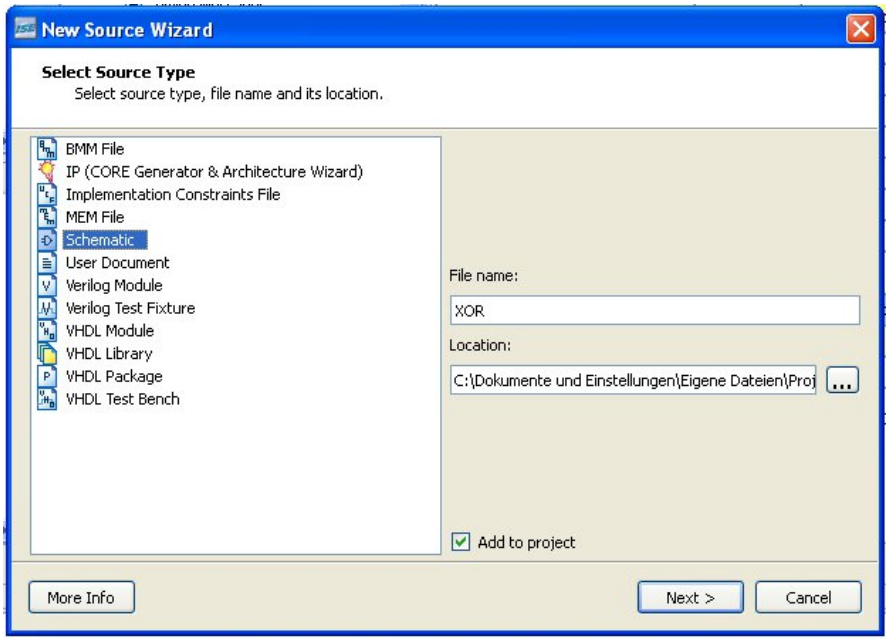




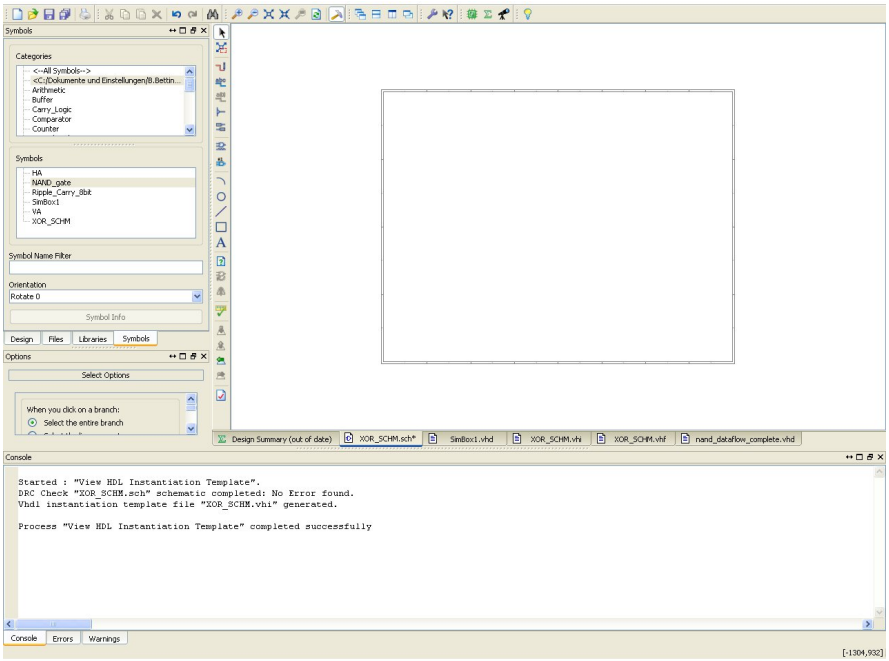
Wiederholen Sie diesen Vorgang gegebenenfalls für alle Entitäten, die für die strukturelle Beschreibung der Zielentity notwendig sind.

Nachdem alle Schaltsymbole erstellt wurden, kann die eigentliche strukturelle Beschreibung begonnen werden. Erzeugen Sie dazu eine neue VHDL Schematic Datei, mit welcher die Beschreibung einer Entity schematisch erfolgen kann. Dazu gehen Sie wie folgt vor:

- Anlegen einer neuen VHDL Schematics Datei mit Rechtsklick auf das FPGA-Design (xc3s200). Aus dem Kontextmenü wählen Sie `New Source`. Der **create new File Wizard** wird geöffnet.
- Als Dateiformat wählen Sie auf der linken Seite `Schematic`. Im Namensfeld geben Sie den Namen der Entity ein. Mit Next gelangen Sie zur Zusammenfassung. Klicken Sie dort auf Finish um die Datei zu erzeugen.

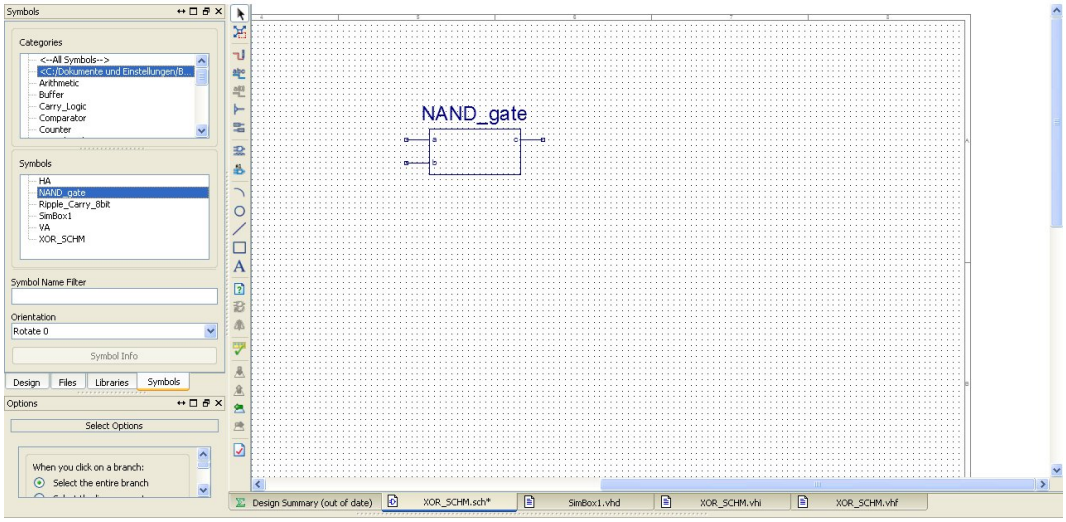


- Anschließend wird die schematische Ansicht der Entity geöffnet. Durch einen Doppelklick auf die Schemata-Datei im Design-Fenster gelangen Sie ebenfalls zu dieser Ansicht. Um wieder zur normalen Ansicht (Design-Fenster) zu wechseln, wählen Sie den Reiter `Design` unterhalb des Symbol-Fensters aus.

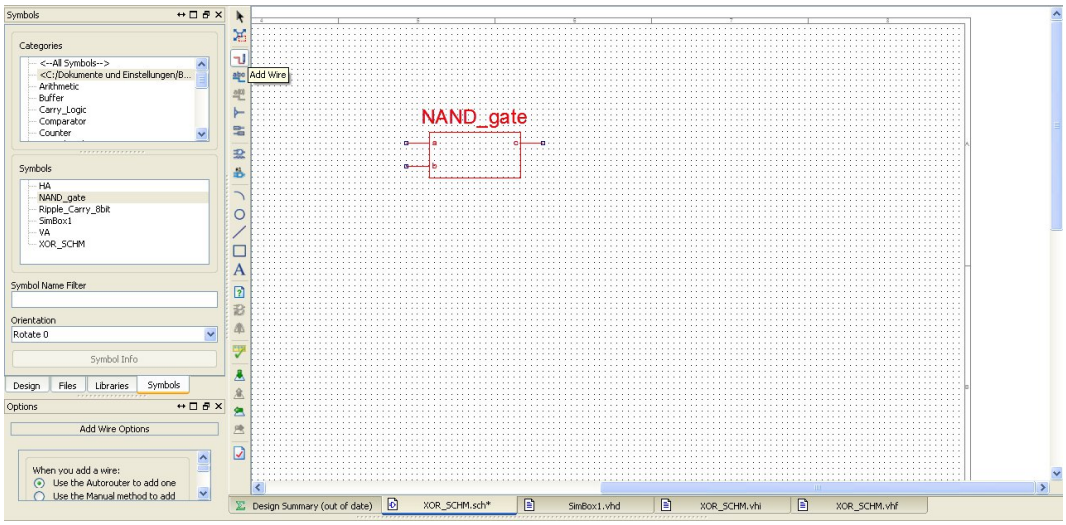


- Nun können Sie die strukturelle Bechreibung schematisch mittels Drag & Drop vornehmen. Um eine Entity zu instanziieren, wählen Sie das zugehörige Schaltsymbol aus dem Symbol-Fenster auf der linken Seite aus. Damit Sie nicht lange suchen müssen, wählen Sie aus dem darüberliegenden Kategorie-Fenster ihr aktuelles Projekt aus (zweite Position). Somit werden lediglich ihre Entitäten angezeigt.

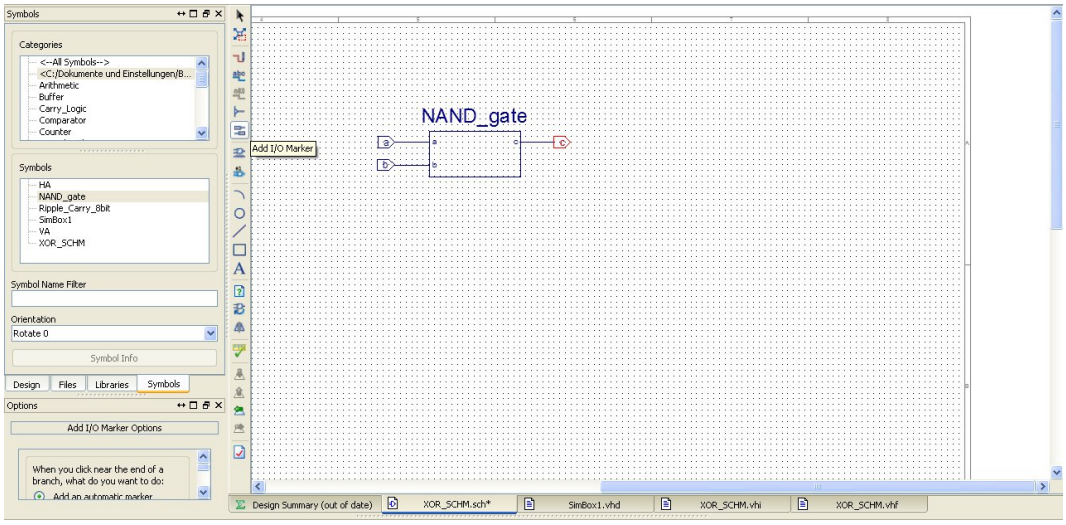
- Nachdem Sie die gewünschte Entity ausgewählt haben, klicken Sie einfach im Schemata-Plan auf der rechten Seite auf eine freie Fläche. Das Schaltbild wird dann dort platziert. Mit der Platzierung wird die Entity instanziiert. Wiederholen Sie diesen Vorgang gegebenenfalls für alle weiteren Entitäten, die benötigt werden.



- Um die erzeugten Entity-Instanzen miteinander zu vernetzen, wählen Sie aus der Werkzeug-Toolbar in der Mitte den Punkt `Add Wire` (dritte Position von oben) aus. Um nun zwei IO-Ports miteinander zu verbinden, klicken Sie auf einen der beiden Ports und ziehen Sie eine Linie (Verdrahtung) zum anderen.

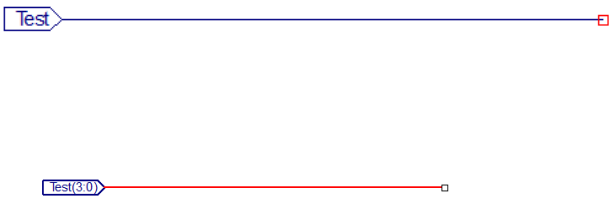


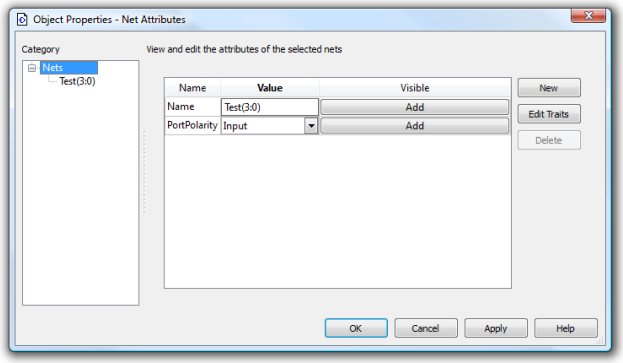
- Um ein IO-Port als Ein- bzw. Ausgangssignal der aus dem Schemata resultierenden Entity zu deklarieren, wählen Sie aus der Werkzeug-Toolbar in der Mitte den Punkt `Add I/O Marker` (siebte Position von oben) aus und klicken Sie auf den Pin, der als I/O-Port deklariert werden soll. Das Verhalten bzw. die Polarität des Ports ist wählbar, indem Sie auf dessen Symbol doppelklicken und im sich öffnenden Eigenschafts-Fenster unter `Nets` die `PortPolarity` auf `In- bzw. Output` stellen.



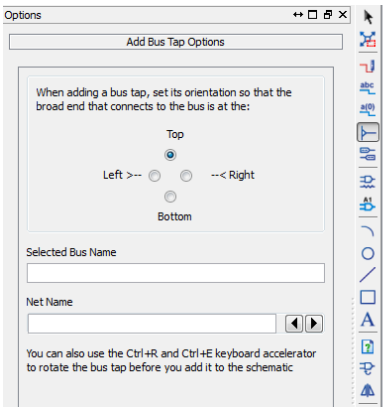
- Um einen Bus zu erstellenm klicken Sie auf `Add Wire` und erstellen Sie einen einfachen Draht. Diese Leitung hat zunächst nur eine Breite von einem Bit. Durch einen Doppelklick auf den neuen Draht öffnet sich sein Eigenschafts-Fenster. Durch einen Namenszusatz können Sie jetzt den Draht zu einem Bus machen. Schreiben Sie in Klammern

hinter dem Namen, welche Bits verwendet werden, z.B. ist `Test(3:0)` ein 4-Bit breiter Bus.





Mit `Add Bus Tap` können von einem Bus einzelne Bit ausgewählt werden. Verbinden Sie dazu das `Bus Tap` mit dem gewünschten Bus. Über den Namen kann ein bestimmtes Bit ausgewählt werden, z.B. wird über den Namen `Test(3)` das höchstwertige Bit des Busses `Test` ausgewählt.



Nachdem Sie ihre Schaltung fertiggestellt haben, kann die resultierende Entity wie folgt verwendet werden.

- Wechseln Sie wieder zum Design-Fenster des Projekts (Reiter `Design` unterhalb des Symbol-Fensters), wo die Entitäten und Dateien ihres aktuellen Projekts angezeigt werden. Es gibt nun zwei Arten den VHDL-Quellcode der Schemata-Datei einzusehen.

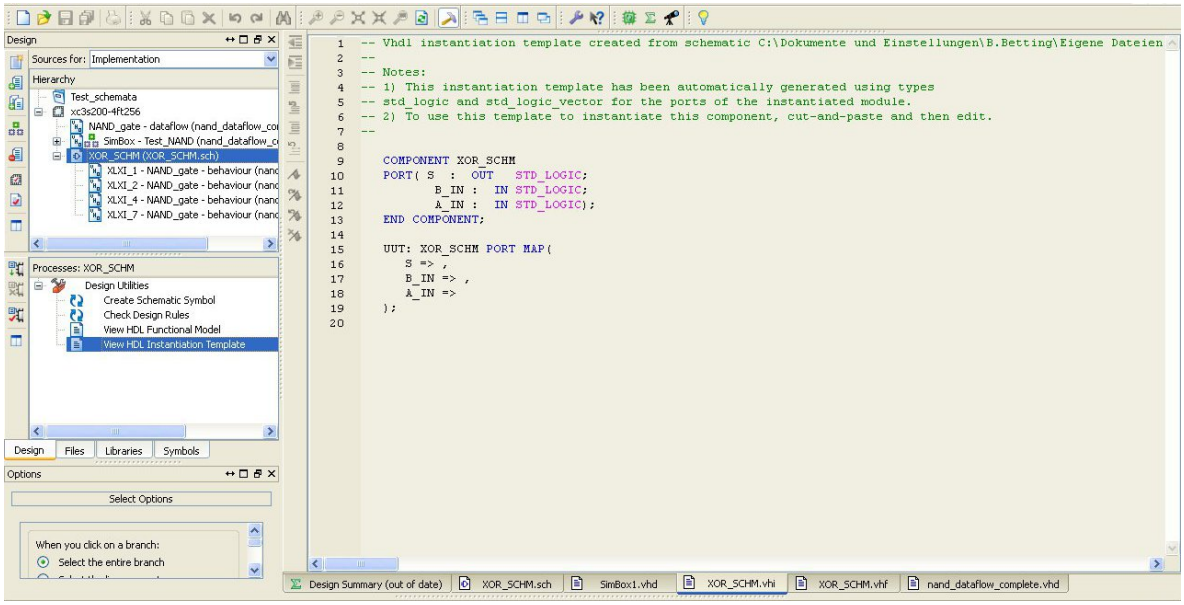
- Die Erste umfasst den gesamten Code, also Entity und Architektur.
- Die Zweite bietet eine Ansicht des zugehörigen Instanz-Templates, d.h. den Component-Block welcher benötigt wird, wenn Sie die Entity in einer höheren Entity instanziieren möchten, wie zum Beispiel in einer Testumgebung (SimBox). Das Instanz-Template wird dann wie gewohnt in die Architektur der Simbox eingefügt und instanziiert.

Für die Ansicht des gesamten Quellcodes gehen Sie wie folgt vor:

- Auswahl der schematisch beschriebenen Entity aus dem Design-Fenster
- Im Prozess-Fenster wählen Sie dann den Punkt `Design Utilities` → `View HDL Functional Model`

Für die Ansicht des Instanz-Templates gehen Sie wie folgt vor:

- Auswahl der schematisch beschriebenen Entity aus dem Design-Fenster
- Im Prozess-Fenster wählen Sie dann den Punkt `Design Utilities` → `View HDL Instantiation Template`



Wichtig: In beiden Varianten ist nur eine Einsicht des Codes möglich und keine Bearbeitung. Dazu muss der Code kopiert und in einer neuen Datei abgelegt werden (in den Praktikumsversuchen ist dies allerdings nicht notwendig).

6 Laufzeiteffekte in Schaltnetzen

Bisher haben wir nur verzögerungsfreie Schaltungen simuliert, d.h. wenn zum Zeitpunkt t eine Signaländerung am Eingang einer Schaltung anlag, hat diese auch zum Zeitpunkt t darauf am Ausgang reagiert. Dies ist natürlich kein reales Verhalten, denn jedes Gatter braucht eine gewisse Zeitspanne, um eine Signaländerung zu verarbeiten. Es gibt hierbei zwei Modelle:

- ideale Verzögerung um die Zeitspanne t: Jede Signaländerung wirkt sich erst nach der Zeitspanne t am Ausgang aus.
- träge Verzögerung um die Zeitspanne t: Signaländerungen, die nur eine Zeitspanne $s < t$ andauern, werden verschluckt. Nur Signaländerungen, die mindestens die Zeitspanne t überstehen, werden ideal verzögert.

Auch VHDL kennt ideale und träge Verzögerung. Dies ist wichtig für die Simulation, da diese Verzögerungen in der Realität auftreten und deshalb auch modellierbar sein müssen. Kennt man die Verzögerungswerte der Technologie, in der man sein Design implementiert, kann man dies bei der Simulation berücksichtigen. In VHDL gibt es jedoch keine Laufzeitglieder, die Verzögerung wird durch Sprachkonstrukte bei der Signalzuweisung implementiert.

Die Architektur eines AND-Gatters mit einer trägen Verzögerung von 10 Nanosekunden sieht so aus:

Listing 4: AND mit Verzögerung

```
architecture dataflow of AND_gate is
begin
  c <= a and b after 10 ns ;
end Dataflow;
```

Das Schlüsselwort ***after*** beschreibt eine träge Verzögerung.

Für eine ideale Verzögerung sieht die Signalzuweisung dagegen wie folgt aus:

Listing 5: Signalzuweisung bei einer idealen Verzögerung

```
c <= transport a and b after 10 ns ;
```

7 Addierer

Addierer berechnen die arithmetische Summe boolescher Vektoren, wobei auch Überträge entstehen können. Daher werden Addierer sowohl durch die Anzahl der zu addierenden Bits, als auch durch die Art der Berechnung des Übertrages klassifiziert. Sie können durch Kaskadierung (Hintereinanderschaltung) von 1-Bit-Volladdierern realisiert werden, die ihrerseits wiederum Halbaddierer enthalten.

7.1 Halb- und Volladdierer

Halbaddierer sind Addierer, die aus zwei booleschen Werten a und b die mod 2-Summe $s = (a + b) mod\ 2$ und den Übertrag (Carry) $c = a * b$ berechnen.

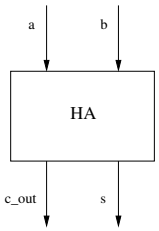


Abbildung 1: Graphische Darstellung eines Halbaddierers

Volladdierer weisen neben dem Übertragsausgang (carry out) noch einen Übertragseingang (carry in) auf, wodurch sie sich u.a. zur Kaskadierung eignen.

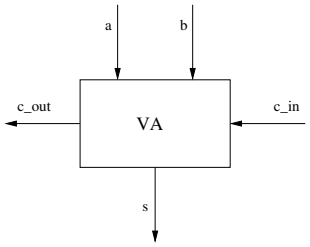


Abbildung 2: Graphische Darstellung eines Volladdierers

8 Vorbereitungsaufgaben

Die folgenden Aufgaben dienen der Vorbereitung der Praktikumsaufgaben und sind teilweise Ausgangsbasis für eine VHDL-Implementierung. Bearbeiten Sie diese Aufgaben vor dem Praktikumstermin.

Aufgabe 1. Weshalb wird die Sensitivitätsliste eines Prozesses benötigt?

Aufgabe 2. Geben Sie die Wahrheitstabelle der Implikationsfunktion an und die dazu gehörige boolesche Gleichung als minimale DNF (disjunktive Normalform).

Aufgabe 3. Informieren Sie sich über Multiplexer. Wie viele Ausgänge kann ein Multiplexer haben? Bei n Steuereingängen hat ein Multiplexer wie viele Dateneingänge?
Beschalten Sie einen 2:1 Multiplexer ohne Verwendung weiterer Gatter um die Implikationsfunktion zu realisieren.

Aufgabe 4. In VHDL ist es möglich, eine Schaltung durch ihre Komponenten und deren Vernetzung untereinander zu beschreiben. Konstruieren Sie ein AND-Gatter mit fünf Eingängen aus maximal vier AND-Gattern mit zwei Eingängen.

Aufgabe 5. Gehen Sie davon aus das es 10 Nanosekunden dauert bis ein gegebenes AND-Gatter (mit zwei Eingängen) bei einem Signalwechsel das neue Ergebnis berechnet und an seinem Ausgang anzeigt. Die Zeit für den Signaltransport zwischen zwei Gattern, also vom Ausgang des ersten Gatters zum Eingang des zweiten Gatters, wird nicht berücksichtigt. Geben Sie für jeden Eingang der Schaltung aus Aufgabe 4 die Zeit an, die benötigt wird, bis ein Wechsel des jeweiligen Eingangs auch einen Wechsel am Ausgang der Gesamtschaltung bewirkt.

Aufgabe 6. Bestimmen Sie mit Hilfe zweier KV-Diagramme die minimale DNF für die Ausgänge c_out und s des Halbaddierers. Übertragen Sie zunächst die Wertetabellen für die Summe und den Übertrag in zwei KV-Diagramme. Führen Sie eine graphische Minimierung durch und ermitteln Sie die minimalen DNFs für die Summe und den Übertrag.

Aufgabe 7. Im Folgenden soll ein 1-Bit-Volladdierer entworfen werden. Er bildet die Summe aus zwei Eingangsbits und dem Carry-out der vorherigen Stufe. Außerdem liefert er ein Carry-out für die nachfolgende Stufe. Beschreiben Sie sein Verhalten durch eine Wahrheitstabelle. Erstellen Sie die KV-Diagramme für die Summe ***s*** und für ***c_out***. Bestimmen Sie die daraus resultierenden minimalen DNFs.

Aufgabe 8. Überlegen Sie sich wie ein Addierer für zwei n-Bit Vektoren aus mehreren Voll- und Halbaddierern aufgebaut werden kann.

9 Praktikumsaufgaben

Aufgabe 1. Legen Sie ein neues Projekt an und implementieren Sie die minimale DNF der Implikationsfunktion aus der Vorbereitungsaufgabe mittels Datenflussbeschreibung und Verhaltenbeschreibung möglichst kompakt. Testen Sie ihre VHDL Module anschließend im Simulator.
Hinweis: Ein Beispiel für eine Verhaltensbeschreibung finden Sie in Kapitel 3.

Aufgabe 2. Erstellen Sie eine Datenflussbeschreibung von von einem OR Gatter mit zwei Eingängen und einem Inverter. Erstellen Sie anschließend daraus eine Strukturbeschreibung für die Implikationsfunktion.

Aufgabe 3. Erweitern Sie die verwendeten Architekturen von Inverter- und OR-Gatter um eine ideale Verzögerung von jeweils 10ns. Erweitern Sie danach ihre Testumgebung um einen Wechsel der Eingangsbelegungen von $a = 1$, $b = 1$ zu $a = 0$, $b = 0$. Beobachten Sie den Ausgabewert und finden Sie eine Erklärung für den scheinbar falschen Signalverlauf.

Aufgabe 4. Legen Sie ein neues Projekt an und erstellen Sie eine algorithmische Beschreibung für einen 2:1 Multiplexer. Testen Sie ihn anschließend in der Simbox.

Aufgabe 5. Versehen Sie ihren Multiplexer mit einer Verzögerungszeit von 10ns und beschalten Sie ihn in der Simboxumgebung um die Implikationsfunktion zu realisieren (siehe Vorbereitungsaufgabe 3). Was passiert hier in der Testumgebung bei einem Wechsel der Eingangsbelegungen von $a = 1$, $b = 1$ zu $a = 0$, $b = 0$?

Aufgabe 6. Legen Sie ein neues Projekt an, erstellen eine Architektur für ein AND-Gatter mit zwei Eingängen, versehen die Schaltung mit einer idealen Verzögerungszeit von 10ns und erzeugen das zugehörige Schaltsymbol (siehe Kapitel 2.2). Öffnen Sie nun den grafischen Editor und erstellen aus ihrem AND-Gatter ein neues AND-Gatter mit 5 Eingängen. Testen Sie ihr VHDL Modul anschließend im Simulator.

Aufgabe 7. Ausgehend von ihrer Entdeckung aus Aufgabe 3 überlegen Sie sich einen Wechsel der Eingabebelegungen der auch bei dieser Schaltung für einen falschen Signalverlauf führt. Überprüfen Sie ihre Vermutung im Simulator.
Hinweis: Der Signalverlauf beim Wechsel der Eingangsbelegungen sollte kurz eine 1 am Ausgang anzeigen obwohl bei beiden Eingangsbelegungen eine 0 am Ausgang erwartet wird.

Aufgabe 8. Legen Sie für diese Aufgabe ein neues Projekt an. Beschreiben Sie die Entity und die Architektur eines Halbaddierers gemäß der ermittelten Gleichungen für die Ausgänge `c_out` und `s`. Simulieren Sie den Halbaddierer mit allen möglichen Eingangsbelegungen.

Aufgabe 9. Erstellen Sie eine Datenflussbeschreibung für den Volladdierer gemäß den Ergebnissen aus den Vorbereitungsaufgaben. Simulieren Sie den Entwurf.

Aufgabe 10. Es sollen eine 5-Bit lange Zahl und eine 8-Bit lange Zahl miteinander addiert werden. Wieviele Volladdierer und Halbaddierer werden benötigt, wenn möglichst wenig Volladdierer verwendet werden sollen? Wie lang ist der Ausgabevektor?
Erstellen Sie eine Strukturbeschreibung. Verwenden Sie dafür ihre Beschreibung für den Halb- und den Volladdierer. Testen Sie ihre Schaltung anschließend.

Hinweis: Den Wert von Vektoren kann man sich im Simulator auch als Dezimalzahl anzeigen lassen.