Praktikum
Grundlagen von Hardwaresystemen
Wintersemester 2016/17

Versuch 1: VHDL-Grundlagen

18. 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

Kapitel 1

Einleitung

Die Entwicklung von mikroelektronischen Schaltungen auf hohem Abstraktionsniveau wird durch den starken Anstieg der Entwicklungskomplexität bei sehr stark reduzierten Entwicklungszeiten bestimmt.

Um konkurrenzfähig zu bleiben, reichen herkömmliche Wege über graphische Schaltplaneingaben auf Logikelementebene in der Industrie schon seit längerem nicht mehr aus. Aus diesem Grund kommen Hardwarebeschreibungssprachen zum Einsatz, um Schaltungen zu entwerfen. Zudem besteht die Möglichkeit den entworfenen Quellcode mit zur Verfügung stehenden Simulationsprogrammen zu testen und das Verhalten der Schaltung schon vor der Synthese zu überprüfen. Diese Beschreibungssprachen werden in zwei Kategorien unterteilt:

- Hardwarebeschreibungssprachen
- Modellierungs- und Simulationssprachen

Eine ***Hardwarebeschreibungssprache*** (engl. Hardware Description Language, HDL) bezeichnet eine Sprache innerhalb einer allgemeinen Klasse von Beschreibungssprachen in der Elektronik. Mit ihr können die Funktion von Schaltungen und ihr Design beschrieben, sowie durch Simulation getestet werden. Sie drücken ein zeitliches Verhalten und/oder eine (räumliche) Schaltkreisstruktur in normalem Text aus. Zu dieser Klasse gehören die Hardwarebeschreibungssprachen, die in Tabelle 1 aufgelistet sind.

Eine ***Modellierungs- und Simulationssprache*** wie SystemC ist insbesondere für die Entwicklung von komplexen elektronischen Systemen, die sowohl Hardware- als auch Softwarekomponenten enthalten, geeignet. Im Gegensatz zu reinen Hardwarebeschreibungssprachen wird SystemC vorrangig zur Modellierung auf noch höheren Abstraktionsebenen eingesetzt, womit Simulationen um den Faktor 100 bis 1000 schneller durchgeführt werden können. Selbst längere Programme, z.B. die Firmware eingebetteter Systeme, die auf der beschriebenen Hardware ablaufen, können mitsimuliert werden.

In diesem Praktikum wird eine Hardwarebeschreibungssprache benötigt, die für die Beschreibung und Simulation digitaler Systeme und deren Umgebung geeignet ist. Wir haben uns für VHDL, welche 1983 vom amerikanischen Departament of Defense initiiert wurde, entschieden.

Ein besonderer Vorteil der Benutzung der Hardwarebeschreibungssprachen liegt in der gemeinsamen Nutzung einheitlicher Modellierung für Simulation und Synthese. Die Simulation dient der Verifikation der Entwurfsidee; die Synthese setzt die Hardwarebeschreibung

HDL	Standardisierung	Anbieter
VHDL	IEEE 1076-1987	EDA
Verilog	IEEE 1364-1995	EDA
GHDL		(Genrad)
VHDL-AMS	(IEEE 1976.1)	(EDA)
Verilog	(OVI)	(EDA)
MAST		Analogy
FAS		Anacad/MGC
Spectre-HDL		Cadence
IRENE		IMAG
KARL		Universität Kaiserslautern 1979
AHPL		Universität Arizona 1974
DACAPO		DOSIS GmbH Dortmund

Tabelle 1.1: Hardwarebeschreibungssprachen

automatisch in eine Netzliste um. Letztere ist die Grundlage für die Implementierung der Schaltung auf Hardwareplattformen wie ASIC, FPGA und CPLD.

Die Zielarchitektur des Praktikums ist ein FPGA von Xilinx. Den FPGA-Markt teilen sich im Wesentlichen die Firmen Xilinx und Altera. Daneben gibt es noch einige weitere Nischenanbieter wie Actel, Lattice und QuickLogic. Xilinx wurde ausgewählt, da neben der Marktbedeutung auch bereits entsprechende Software und Erfahrung an der Professur vorhanden sind.

Kapitel 2

Grundlagen

In diesem Versuch werden grundlegende VHDL-Konzepte erläutert. Am Beispiel eines NAND- und eines XOR-Gatters wird der VHDL-Sprachumfang schrittweise und problemorientiert eingeführt. Im Einzelnen werden dabei die folgenden Punkte behandelt:

- Methoden der Schaltungsbeschreibung:
  - Konzept: Trennung zwischen Schnittstelle (*Port*), Entwurfseinheit (*Entity*) und Verhalten (*Architecture*)
  - Beschreibungsstile von Schaltungen:
    - strukturelle Beschreibung
    - Datenflussbeschreibung
    - algorithmische Beschreibung
- Definition von Signalen
- Bedienung der VHDL-Entwurfsumgebung (Compiler und Simulator)
- Modellierung einfacher boolescher Schaltnetze
- Beschreibung von Halb- und 1-Bit-Volladdierern

2.1 VHDL

2.1.1 Entity, Architektur und Signale

Die nachfolgend aufgeführten VHDL-Elemente sind als grundlegende Strukturelemente jeder VHDL-Beschreibung anzusehen.

2.1.1.1 Entity

In der mit ***Entity*** bezeichneten Entwurfseinheit werden die Schnittstellen eines VHDL-Funktionsblocks nach außen beschrieben. Unter der ***Schnittstelle*** einer Schaltung versteht man die Menge aller Signalleitungen, die diese Schaltung mit ihrer Umgebung verbinden, inklusive der Information über die Signaleigenschaften wie z.B. Wertebereiche und Signalrichtungen. Beim Vergleich eines Board-Designs mit einem VHDL-Quellcode stellt eine Entity den zu

bestückenden IC-Gehäusetyp dar, der durch die Anzahl, die Eigenschaften und die Bezeichnung der Anschlüsse definiert ist. Die Deklaration der Anschlüsse innerhalb der Entity erfolgt mit Hilfe der ***Port***-Anweisung. Dabei beschreibt jedes Element der Port-Anweisung einen oder mehrere Anschlüsse.

Entity-Syntax

```
entity <Entityname> is
  port (<Deklaration der Ein- und Ausgänge>);
end <Entityname>;
```

Listing 2.1: Deklaration eines NAND-Gatters

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

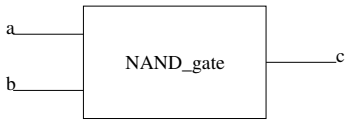

Abbildung 2.1: Graphische Darstellung der Entity eines NAND-Gatters

Die Entity einer Schaltung (mit zwei Eingängen und einem Ausgang) kann man auch graphisch wie in Abbildung 2.1 darstellen. Dabei sind `a` und `b` die Eingangssignale und `c` das Ausgangssignal der Schaltung. Alle drei Signale sind vom Typ `std_logic`, was bedeutet, dass es sich um einzelne Logiksignale handelt.

2.1.1.2 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 neun 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. 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.

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.

2.1.1.3 Architektur

Die ***Architektur*** beschreibt das Innenleben, d.h. die Funktionalität des VHDL-Codes. Jeder Entity muss (mindestens) eine Architektur zugeordnet sein. In obiger Vorstellung beschreibt also die Architektur, welche Funktion bzw. welcher Chip sich in dem Gehäuse befindet.

Syntax einer Architektur:

`architecture` <Architekturname> `of` <Entityname> `is`
[Lokale Deklarationen]
`begin`
<VHDL-Anweisungen>
`end` <Architekturname>`;`

Der Deklarationsteil definiert Typen, Unterprogramme, Konstanten und Signale, die in dieser Architektur benötigt werden, während die VHDL-Anweisungen das eigentliche Innenleben beschreiben. Für diese Beschreibung gibt es drei Stile: Verhalten, Datenfluss und Struktur. Innerhalb des Anweisungsteils kann ein Stil oder eine beliebige Kombination dieser Stile benutzt werden.

Verhalten Die Grundstruktur der Verhaltensbeschreibung ist der ***Prozess***.

Syntax eines Prozesses:

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

Listing 2.2: 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)

Hinweis: Signalzuweisungen sind erst am Ende des Prozesses gültig!

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

Listing 2.3: 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 2.4: 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;
```

In Praktikumsaufgabe 5 wird die Strukturbescheibung für den Entwurf eines XOR-Gatters verwendet.

2.1.2 Simulation einer VHDL-Schaltung

Für die Simulation der entworfenen Schaltung werden sogenannte Stimuli (Testbench) verwendet. Dazu wird eine (Test-)Entity mit beliebigem Namen, im Folgenden `SimBox`, erstellt, die selbst keine Ein- und Ausgänge besitzt. Die zu testende Schaltung bekommt dann eine Abfolge von Werten an den Eingängen angelegt und es wird beobachtet, wie sich dabei deren Ausgängen verhalten und ob dies der Spezifikation entspricht, die logische Grundlage des Entwurfs ist. Die SimBox instantiiert die zu testende Komponente, und versorgt die Eingänge zu festgelegten Zeiten mit Testwerten, so dass man die Reaktion der Ausgänge beobachten kann.

Um das Beispiel des NAND-Gatters aufzugreifen: In der Architektur der SimBox muss das NAND-Gatter mit seinen Ein- und Ausgängen als Komponente deklariert werden. Außerdem sind interne Signale erforderlich, die dann an den Ein- und Ausgängen (ports) der Komponente angeschlossen werden. Die Stimulidatei sieht dann folgendermaßen aus:

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

-- Die SimBox ist die "ganze Welt", es existieren
-- keine Ein- und Ausgaenge
```

```
entity SimBox is
end SimBox;

-- Architecture der SimBox

architecture Test_NAND of SimBox is

-- Hier muss das NAND-Gatter mit
-- den gleichen Ports und Portnamen
-- wie in der Entity als Komponente deklariert werden

component NAND_gate
  port (
    a,b : in  STD_LOGIC;
    c   : out STD_LOGIC
  );
end component;

-- hier werden die Signale deklariert,
-- die in der Simbox verwendet werden

signal a_test, b_test, c_test : STD_LOGIC;

begin

  -- Dies ist die Instanziierung von
  -- my_NAND_gate der "Klasse" NAND_gate

  my_NAND_gate : NAND_gate
  port map (
    a => a_test,      -- explizite Zuordnung der Testsignale
    b => b_test,      -- zu den Ports der Entity
    c => c_test
  );

  -- Hier werden den Signalen konkrete Werte zugewiesen.
  -- Die Zeiten beziehen sich auf den Start der Simulation.
  a_test <=
    '0' after 0 ns,
    '1' after 2 ns;

  b_test <=
    '0' after 0 ns,
    '1' after 4 ns;

end Test_NAND;
```

2.1.3 Konfiguration

Mittels der Konfiguration kann der Entwickler zwischen verschiedenen Design-Alternativen und -Versionen auswählen. Dabei bestimmt eine Konfiguration, welche Realisierung für eine Entity, benutzt wird, d.h. welche Architektur in einer Entity eingesetzt wird. Dazu muss in der Architektur der SimBox, bevor eine Entity instanziiert wird, angegeben werden welche Architektur die Instanz verwenden soll. work ist dabei das Arbeitsverzeichnis des Compilers und des Simulators (siehe Listing 2.5).

Syntax einer *Konfiguration:*
for <Instanz> : <Entityname> use entity work.<Entityname> (Architekturname)

Listing 2.5: Konfiguration eines NAND-Gatters

```
-- Innerhalb der Architecture der SimBox
architecture Test_NAND of SimBox is
.
.
.
-- Konfiguration der verwendeten Architecture "dataflow"
for my_NAND_gate : NAND_gate use entity work.NAND_gate(dataflow)
.
.
.
begin
.
.
.
my_NAND_gate : NAND_gate
.
.
.
end Test_NAND;
```

Wenn das Design die Bereitstellung multipler Architekturen für Entities vorsieht , d.h. eine Entity besitzt mehrere Architekturen, dann ist es notwendig, dass für jede instanziierte Komponente die zu verwendende Architektur angegeben wird. Wird keine Architektur angegeben, so wird der Compiler stets die Architektur als Default ansehen, welche im Programmcode sequenziell an letzter Stelle steht.

2.1.4 Das gesamte Projekt

Unser erstes NAND-Simulationsprojekt besteht nun aus zwei Teilen:

- Entity und Architektur
- Simulationsumgebung SimBox mit Stimuli und Konfiguration

```
-- Deklaration und Implementierung der NAND-Entity
```

```
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;

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

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;

architecture structure of NAND_gatter 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;
```

```
-- Deklaration und Implementierung der Simulationsumgebung

Library IEEE;
use IEEE.std_logic_1164.all;

-- Die SimBox ist die "ganze Welt", es existieren
-- keine Ein- und Ausgaenge
entity SimBox is
end SimBox;


architecture Test_NAND of SimBox is

-- Hier muss das NAND-Gatter mit
-- den gleichen Ports und Portnamen
-- wie in der Entity als Komponente deklariert werden

component NAND_gate
  port (
    a,b : in  STD_LOGIC;
    c   : out STD_LOGIC
  );
end component;

-- hier werden die Signale deklariert,
-- die in der Simbox verwendet werden

signal a_test, b_test, c_test: STD_LOGIC;

--Konfiguration der SimBox
-- hier werden die verwendeten Architectures der Entity-Instanzen
-- angegeben. Bei nur einer Architektur je Entity
-- ist dieser Block !OPTIONAL!

for my_NAND_gate : NAND_gate use entity work.NAND_gate(behaviour);

begin

  -- Dies ist die Instanziierung von
  -- my_NAND_gate der "Klasse" NAND_gate
```

```
  my_NAND_gate : NAND_gate
  port map (
    a => a_test,      -- explizite Zuordnung der Testsignale
    b => b_test,      -- zu den Ports der Entity
    c => c_test
  );

  -- Hier werden den Signalen konkrete Werte zugewiesen.
  -- Die Zeiten beziehen sich auf den Start der Simulation.
  a_test <=
  '0' after 0 ns,
  --'0' after 2 ns,  -- nicht notwendig, ein Signal behaelt
                     -- seinen Wert bis ein neuer zugewiesen wird
  '1' after 4 ns;

  b_test <=
  '0' after 0 ns,
  '1' after 2 ns,
  '0' after 4 ns,
  '1' after 6 ns;

end Test_NAND;
```

2.2 Einführung in Xilinx ISE WebPack

In diesem Praktikum verwenden sie als Entwicklungsumgebung das ISE WebPack des Logik-IC Herstellers Xilinx. Diese ermöglicht es Ihnen Ihre VHDL-Entwürfe zu implementieren, simulieren und mittels Synthese auf reale FPGA-Hardware als Zielarchitektur zu übertragen. In den ersten drei Praktikumsversuchen werden zunächst grundlegende digitalen Schaltungen in VHDL realisiert und mittels Simulation verifiziert. In den verbleibenden drei Versuchen werden komplexere Schaltungen folgen und auf realer Hardware getestet.
Legen sie bitte für jeden Praktikumstag ein neues Verzeichnis an. Es empfiehlt sich, ein einheitliches Bezeichnungsschema durchzuhalten, z. B. `versuch1` für den ersten Versuch, `versuch2` für den zweiten usw. Am Besten organisiert man die einzelnen Entwürfe in Projekten. Die folgenden Abschnitte geben eine kurze Einführung in die grundlegenden Funktionen und Arbeitsschritte mit ISE Webpack.

2.2.1 ISE Webpack Starten, Erzeugen von Projekten

Für das Starten sowie Anlegen eines neuen Projekts gehen sie wie folgt vor:

- Doppelklicken sie auf das **Xilinx ISE Design Suite**-Icon auf dem Desktop. Dadurch wird das ISE Webpack gestartet.
- Mit dem Menüpunkt **File => New Project** wird ein neues Projekt erstellt.

- Auf `Create New Project` muss unter `Project Name` dem Projekt ein beliebiger, möglichst selbsterklärender Name gegeben werden. Unter Location wählen sie das Verzeichnis Ihres gegenwärtigen Accounts unter `C:\Benutzer\Name des Accounts`, da sie lediglich in diesem Verzeichnis Schreibrechte besitzen! Achten sie darauf, das sie keine Leerzeichen in Datei- oder Verzeichnisnamen verwenden. Der `Top-Level Source Type` ist `HDL` (Hardware Description Language). Mit `Next` geht es zur nächsten Dialogseite (Geräte Eigenschaften).

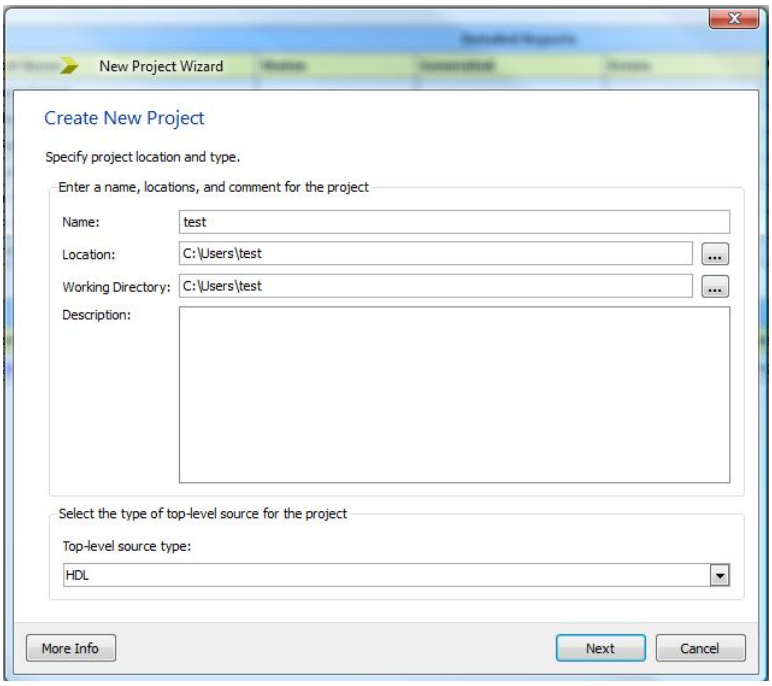

Abbildung 2.2: Projekt Namen und Verzeichnis angeben

Hier wird jetzt definiert, auf welchem FPGA das Projekt dann laufen. Für die im Praktikum verwendete Hardware verwenden sie folgende Einstellungen:

- Wählen sie unter `Evaluation Development Board None Specified` aus
- Die `Product Category` ist `All`
- Wählen sie unter `Family` den `Spartan6`
- Und als `Device` den `XC6SLX16`
- Mit dem `Package CSG324`
- Und setzen sie den `Speed` auf `-3`
- Wählen sie unter `Synthesis ToolXST (VHDL/Verilog)`
- Und als `Simulator ISim (VHDL/Verilog)`
- Die `Preferred Language` ist `VHDL`

- `Property Specification in Project File`
- Die Box `Manual Compile Order` sollte leer sein
- Wählen sie als `VHDL Source Analysis Standard` `VHDL-93` aus
- Und die Box `Enable Message Filtering` sollte ebenfalls frei sein

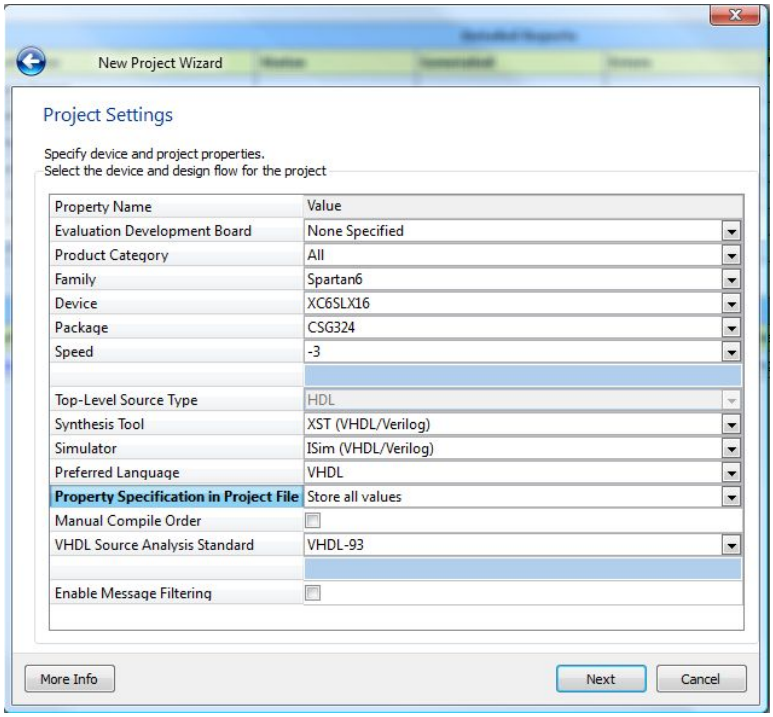

Abbildung 2.3: Projekt Eigenschaften einstellen

Jetzt ist Ihr Projekt richtig konfiguriert und mit `Next` gelangen sie zur nächsten Dialogseite. Hier bekommen sie noch mal eine Übersicht über die Einstellungen Ihres Projekts. Schließen sie nun den Assistenten mit dem Button `Finish`.

Nach der Erzeugung des Projekts, kann direkt mit der Implementierung von Entwürfen begonnen werden. Dazu muss zunächst ein neue VHDL-Datei angelegt werden, in welchem die Entities Platz finden.

2.2.2 Anlegen und Einfügen von VHDL Dateien

2.2.2.1 Anlegen von Dateien

Für die Erzeugung von VHDL-Dateien gehen sie wie folgt vor:

- Am linken oberen Rand des Bildschirms finden sie das Design-Fenster. Hier werden alle Komponenten des aktuell geöffneten Projekts hierarchisch aufgelistet. Um nun eine Datei zu erzeugen klicken sie zunächst mit der rechten Maustaste auf das FPGA-Design (xc6slx16-3csg324). Dieses befindet sich direkt unterhalb des Projekt-Ordners in der Baumstruktur des Design-Fensters. In dem sich öffnenden Kontext wählen sie dann `New Source` aus. Der zugehörige Dialog wird geöffnet.

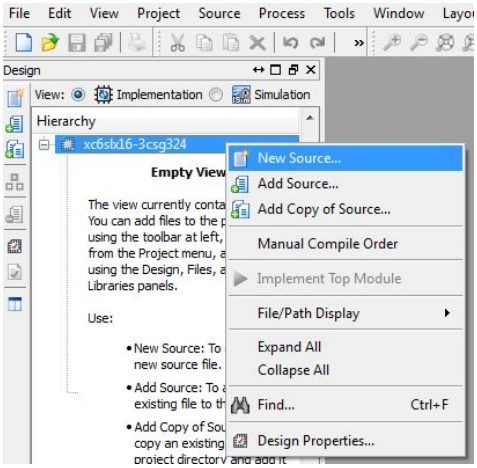

Abbildung 2.4: Neue Datei unter New Source hinzufügen

- Auf `Select Source Type` wählen sie als Datei-Typ `VHDL Module` aus dem Dialog aus. Unter `File Name` geben sie einen Namen für die Datei an. Empfehlenswert ist ein Bezeichner der möglichst mit dem Namen der Entity übereinstimmt. Das Verzeichnis unter `Location` ist das des aktuellen Projekts und wird allgemein beibehalten. Mit `Next` zur nächsten Dialogseite.

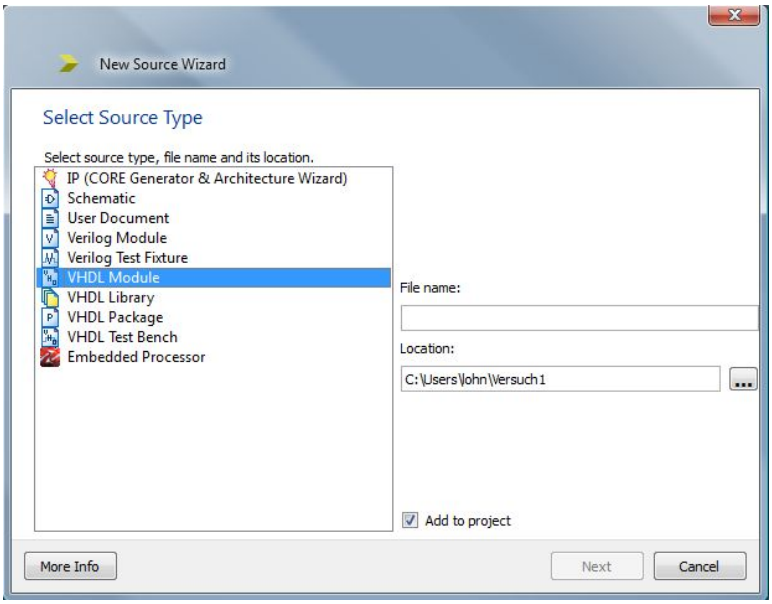

Abbildung 2.5: Name und Dateityp angeben

- Auf `Define Module` tragen sie unter `Entity Name` den gewünschten Namen Ihrer Entity ein (auch hier möglich selbsterklärend). Mit `Architecture Name` geben sie ein Namen für die Architektur an. Hier empfiehlt es sich eine Kombination aus Entity Name und Beschreibungsstil (strukturelle Beschreibung, Datenflussbeschreibung, algorithmische Beschreibung) zu wählen um eine bessere Übersichtlichkeit bei mehreren Architekturen zu

gewährleisten. Innerhalb der darunter liegenden Liste können Signalleitung (Ein- und Ausgänge der Entity) automatisch durch den Wizard generiert werden. Für den Anfang ist es aber ratsam allen VHDL-Code möglichst selbst zu implementieren um die Syntax und das Konstrukt besser zu erlernen. Auf `Next` gehts weiter.

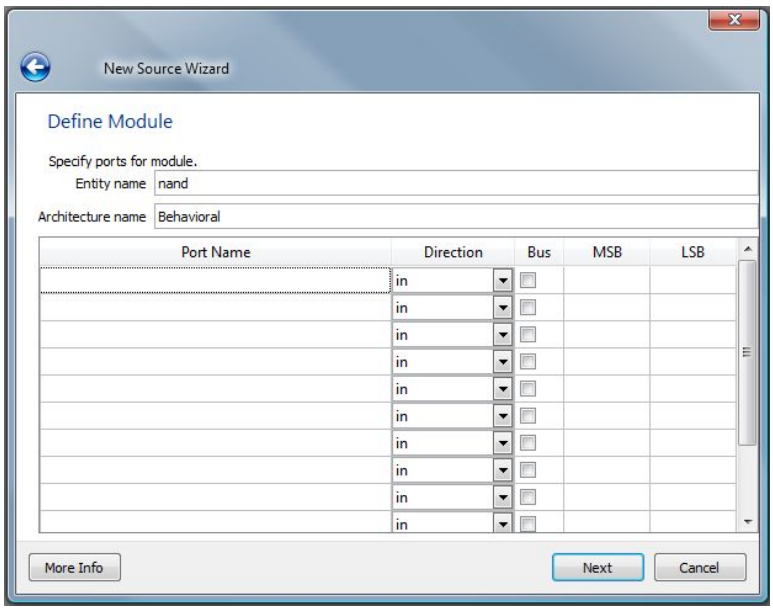

Abbildung 2.6: Entity- und Architekturname sowie bereits definierte Signalleitungen

- Es folgt die Zusammenfassung der Dateieigenschaften. Mit `Finish` bestätigen und die Datei ist angelegt. Im Design-Fenster ist die nun erzeugte Datei zu sehen. Um zu editieren Doppelklicken sie darauf.

2.2.2.2 Hinzufügen von Dateien

Beim Hinzufügen von bereits vorhandenen VHDL-Dateien sollte immer darauf geachtet werden, dass sich diese auch im Verzeichnis des aktuellen Projekts (work) befinden bzw. abgelegt werden, um Pfadfehler beim Kompilieren der Datei zu vermeiden. Für das Hinzufügen von bestehenden Dateien sollten sie daher immer wie folgt vorgehen:

- Klicken sie zunächst mit der rechten Maustaste auf das FPGA-Design (xc6slx16-3csg324) im Design-Fenster. Aus dem sich öffnenden Kontextmenü wählen sie den Punkt `Add Copy of Source` aus (keinesfalls nur `Add Source`). Der Windows FileChooser öffnet sich.

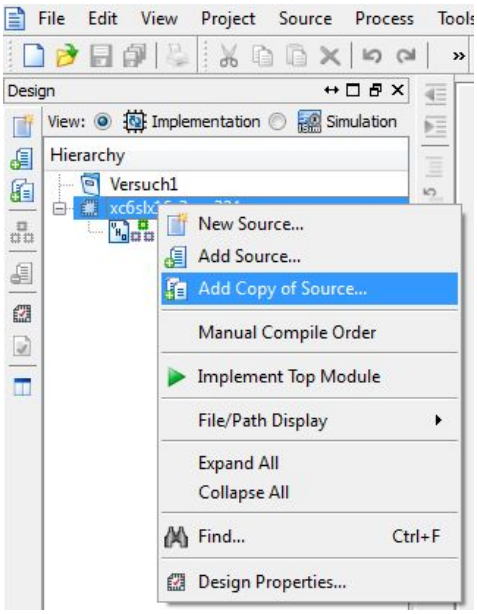

Abbildung 2.7: Existierende Datei mit Add Copy of Source hinzufügen

- Wählen sie die gewünschte Datei aus dem jeweiligen Verzeichnis aus und bestätigen sie mit `OK`. Die Datei wurde nun dem Projekt hinzugefügt und in das aktuelle Projektverzeichnis (work) kopiert bzw. abgelegt.

Bei der späteren Synthese von Entwürfen (Versuche 3 - 6) ist darauf zu achten, dass die richtige VHDL-Datei als Top Modul ausgewiesen ist, welche die Hauptkomponente (Main Entity) beinhaltet. Dazu klicken sie einfach mit der rechten Maustaste auf die gewünschte Datei mit der darin befindlichen Entity und wählen `Set as Top Module`.

2.2.3 Simulation von Entwürfen

Für die Simulation eines Entwurfs verwendet ISE Webpack den von Xilinx mitgelieferten Simulator ISim. Um ISim zu starten gehen sie wie folgt vor:

- Wählen sie unter `View` die `Simulation` aus.

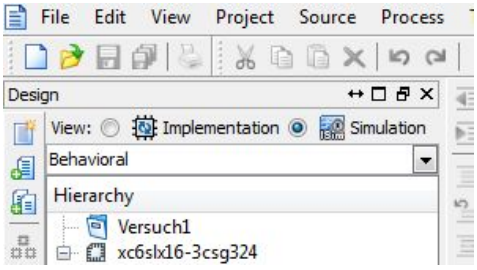

Abbildung 2.8: Prozessvorgang Simulation auswählen

- Markieren sie nun in der Hierarchie die SimBox, die sie simulieren wollen.

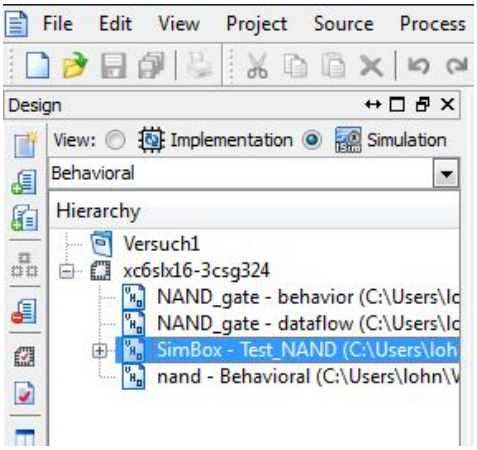

Abbildung 2.9: Die markierte Auswahl wird simuliert

- Im Prozess-Fenster welches direkt unter dem Design-Fenster liegt ist nun der Menüpunkt `Simulate Behavioral Model` verfügbar. Mit einem Doppelklick darauf starten sie die Kompilierung der Datei. Sofern keine syntaktischen Fehler vorhanden sind wird im Anschluß ISim gestartet. Sollten Fehlermeldungen ausgegeben werden, müssen diese zunächst korrigiert und dieser letzte Schritt wiederholt werden.

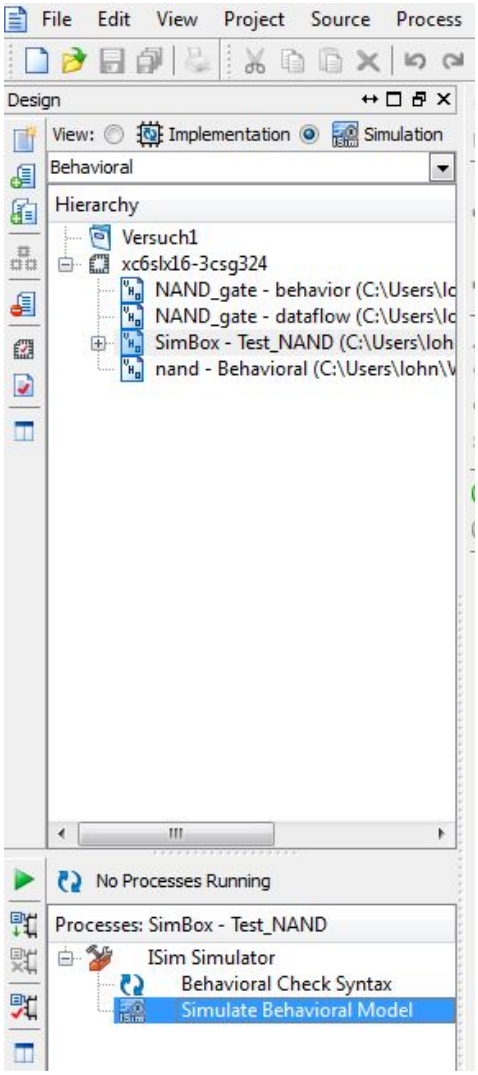

Abbildung 2.10: Starten der Simulation

2.2.3.1 Verwendung von ISim

ISim hebt sich in Bezug auf die Anwenderfreundlichkeit deutlich von anderen frei verfügbaren Simulatoren wie zum Beispiel ModelSim, etc. ab. Die Oberfläche setzt sich im wesentlichen aus den drei Komponenten Instanz-, Objekt- und WaveForm-Fenster zusammen (von links nach rechts). Zentrales Element ist das WaveForm-Fenster in welchem die zeitlichen Signalverläufe abgebildet sind. Im mittleren Objekt Fenster können die Signale des aktuellen Entwurfs ausgewählt werden, für welche ein Verlauf im WaveForm erstellt werden soll. Die Standardeinstellung sieht die Darstellung aller Signale vor und fügt diese deshalb automatisch hinzu. Sie können aber dennoch Signale nach belieben zum WaveForm Fenster hinzufügen oder entfernen, indem sie diese aus dem Objekt Fenster mit Rechtsklick auswählen und mit dem Menüpunkt `Add to Wave Configuration` bestätigen. Das Instanz Fenster am rechten Bildschirmrand zeigt die Hierarchie der Instanzen des gegenwärtigen Entwurfs.

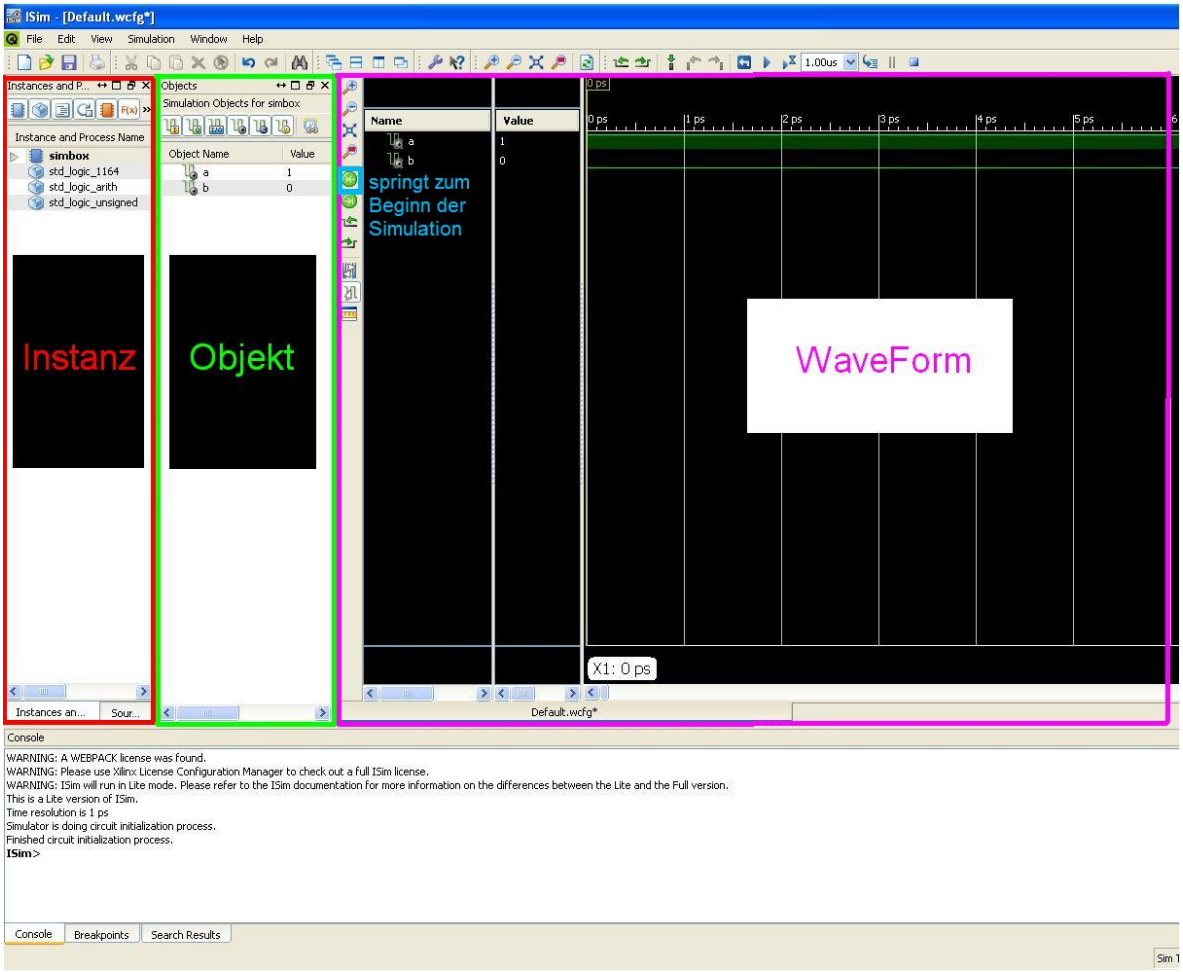

Abbildung 2.11: v.l.n.r. Instanz-, Objekt- und WaveForm-Fenster

Simulation starten
Eine Simulation wird entweder über den Menüpunkt **Simulate => Run All** oder über den Konsolenbefehl run `<Laufzeit>` gestartet werden. Bei der letzteren Variante muss eine Zeit-

spanne inklusive der Zeiteinheit übergeben werden d.h. zum Beispiel run 100ns (Laufzeit der Simulation 100 Nanosekunden).

Mit dem grünen Pfeil nach links (im Bild blau markiert) gelangen sie zum Beginn der simulierten Zeit.

Simulation beenden
Simulationsergebnisse können zu jedem Zeitpunkt über den Menüpunkt **Simulate => Restart** oder über den Konsolenbefehl *restart* rückgesetzt werden.

Sollte die Analyse der Simulaton abgeschlossen sein, so kann ISim direkt geschlossen werden.

2.3 Multiplexer

Ein Multiplexer ist ein auswählendes Schaltnetz, dessen Funktionalität dem einer einfachen Datenweiche entspricht. Mittels Steuereingängen wird einer von mehreren Dateneingängen selektiert und auf den Ausgang durchgeschaltet.

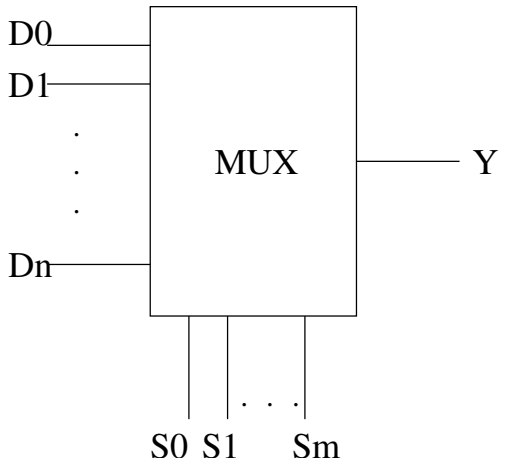

Abbildung 2.12: Blockschaltbild eines Multiplexers mit (n+1)-Eingängen, (m+1)-Steuereingängen und einem Ausgang

In Abbildung 2.12 sind die Dateneingänge mit D0, ..., Dn, die Steuereingänge mit S0, ..., Sm und der Ausgang mit Y bezeichnet. Das folgende Beispiel eines 4:1 Multiplexers (Abbildung 2.13) zeigt das genaue Verhalten in Abhängigkeit der beiden Steuersignale S0 und S1. Die Ausgabe Y des Multiplexers wird durch die Wahrheitstabelle 2.1 beschrieben. Dabei wird jeweils ein Dateneingang durch das entsprechende Steuerwort mit UND-verknüpft und auf den entsprechenden Ausgang geschaltet. Daraus ergibt sich die Schaltfunktion

$$Y = (\overline{S1} \wedge \overline{S0} \wedge D0) \vee (\overline{S1} \wedge S0 \wedge D1) \vee (S1 \wedge \overline{S0} \wedge D2) \vee (S1 \wedge S0 \wedge D3)$$

S1	S0	Y
0	0	D0
0	1	D1
1	0	D2
1	1	D3

Tabelle 2.1: Schaltfunktion eines 4:1 Multiplexers

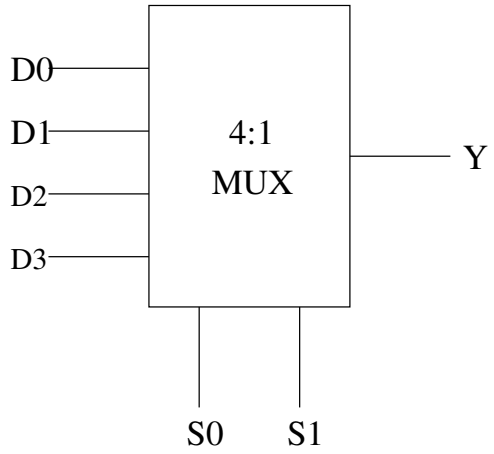

Abbildung 2.13: Blockschaltbild eines Multiplexers mit 4-Eingängen, 2-Steuereingängen und einem Ausgang

2.4 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.

2.4.1 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.

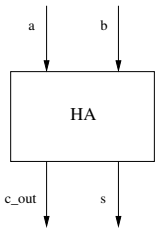

Abbildung 2.14: Graphische Darstellung eines Halbaddierers

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

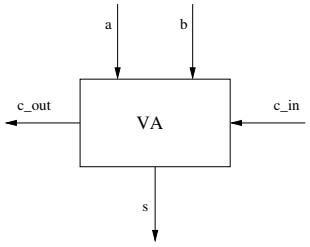

Abbildung 2.15: Graphische Darstellung eines Volladdierers

Kapitel 3

Anmerkungen und Tipps

- VHDL unterscheidet *nicht* zwischen Gross- und Kleinschreibung.
- In VHDL werden Kommentare mit `--` eingeleitet.
- Namen und Bezeichner müssen mit einem Buchstaben beginnen. Die nachfolgenden Zeichen können Buchstaben, Ziffern oder aber der Unterstrich `_` sein.
- Vollständige VHDL-Anweisungen werden mit einem Semikolon abgeschlossen.
- Die *Entity*-Beschreibung enthält keinerlei Information über die innere Struktur oder das Verhalten einer Schaltung. Sie beschreibt lediglich die Ein- und Ausgänge.
- Während sich die `library`-Anweisung auf alle Entwurfseinheiten einer VHDL-Datei bezieht, ist bei der `use`-Anweisung zu beachten, dass diese vor jeder einzelnen entity wiedeholt werden muss, die diese Datentypen verwendet.
- Die Zuweisung eines Signalwerts vom Typ `std_logic` erfolgt durch den Operator `<=`:

```
Y <= '0';
```

- Die Zuweisung von `std_logic_vector` Werten erfolgt durch Einbettung in Anführungszeichen

```
E <= "1010";
```

- Bei der Zuweisung der konkreten Werte zu den Signalen werden die Wert-Zeit-Tupel durch Kommata getrennt. Nur am Schluss steht ein Semikolon, um die VHDL-Anweisung abzuschließen.

```
   a_test <=
'0' after 0 ns,
'1' after 2 ns;
```

- Der Name *SimBox* ist kein Schlüsselwort; Sie können also statt dessen jeden beliebigen anderen Namen für diese Anordnung aus Stimuli-Generator und unserer eigentlichen Schaltung verwenden.
- Tipp für die Vorbereitungsaufgabe 5: Erweitern Sie die DNF geeignet, z.B. mit $a\overline{a} = 0$.

- Zum Wechseln der Belegung alle 2ns können Sie folgendes benutzen:

```
a_test <= not a_test after 2 ns;
```

Kapitel 4

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. Erläutern Sie die Funktionalität der Kernkomponenten Entity, Architektur, Konfiguration und Stimuli eines regulären VHDL-Schaltungsentwurfs. Beschreiben Sie dabei auch in welchem Zusammenhang die Komponenten untereinander stehen.

Aufgabe 2. Finden Sie die Fehler in den folgenden Quellcodes:

```
1. Library IEEE
2. use ieee.std_logic_1164.all;
3.
4. entity X_gate is
5.   port(a : in STD_LOGIC;
6.              b : in std_LOGIC;
7.        c : out std_logic;);
8. end X
9.
10. architecture dataflow of X IS
11. begin
12.   c = A and b ;
13. END;
```

```
1. Library IEEE;
2. use IEEE.std_logic_1164.all;
3.
4. entity SimBox is
5. end;
6.
7.
8. architecture Test_X of SimBox is
9.
10. component X_gate
11. port
```

```
12. (
13.   a : in  STD_LOGIC;
14.     b : in  STD_LOGIC;
15.   c : out STD_LOGIC
16. );
17. end;
18.
19. signal a_test, b_test, c_test: STD_logic
20.
21. for my_X_gate : x_gate use entity work.X_gate(structure);
22.
23. begin
24.
25. my_X_gate : X_gate;
26. port map
27. (
28.   a = a_test,
29.   b = b_test,
30.   c = c_test
31. );
32.
33. a_test <=
34.   '0' after 0 ns;
35.   '1' after 2 ns;
36.
37. b_test <=
38.   '0' after 0 ns,
39.   '1' after 4 ns;
40.
41. end Test_X_gate;
```

Aufgabe 3. Listen Sie die in VHDL verfügbaren logischen Operatoren auf. Dies kann in Form einer Tabelle mit folgenden Spalten erfolgen: Operator, Bedeutung und Beispiel.

Aufgabe 4. Wozu ist die Sensitivitätsliste eines Prozesses notwendig? Muss diese Liste immer vorhanden sein?

Aufgabe 5. Geben Sie die Wahrheitstabelle eines Äquivalenz-Gatters, die dazu gehörige boolesche Gleichung (lässt sich leicht aus dem KV-Diagramm ablesen) und die minimale DNF (disjunktive Normalform) an.

Aufgabe 6. In VHDL ist es auch möglich, eine Schaltung durch ihre Komponenten und deren Vernetzung untereinander zu beschreiben. Konstruieren Sie ein XOR-Gatter ausschließlich unter der Verwendung eines 4:1 Multiplexer. Verwenden Sie dabei die beiden Eingänge des Gatters als Steuersignale S1, S0 des Multiplexers.

Aufgabe 7. 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 8. 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 $\boldsymbol{s}$ und für $\boldsymbol{c\_out}$. Bestimmen Sie die daraus resultierenden minimalen DNFs.

Kapitel 5

Praktikumsaufgaben

Aufgabe 1. Erzeugen Sie ein neues ISE Webpack Projekt für diesen Versuch. Gehen Sie dazu vor wie in 2.2.1 beschrieben. Laden Sie die VHDL-Beschreibung des NAND-Gatters, sowie die zugehörige SimBox von der Webseite des Praktikums runter und fügen Sie diese dem Projekt hinzu (siehe Abschnitt2.2.2.2).

Aufgabe 2. Simulieren Sie Ihre VHDL-Beschreibung des NAND-Gatters mittels ISim (siehe Abschnitt 2.2.3).

Mit den Lupensymbolen können Sie im WaveForm-Fenster herein- und herauszoomen. Bei der Simulation werden die als Stimuli definierten Testwerte zu den entsprechenden Zeitpunkten angelegt, und das Ergebnis im WaveForm-Fenster angezeigt. Alternativ können Sie auch schrittweise simulieren.

Aufgabe 3. Erstellen Sie eine Entity mit dem Namen `mux_gate` und eine zugehörige Architekture namens `mux_behavior` für einen gewöhnlichen 4:1 Multiplexer. Verwenden Sie dabei die *process*-Umgebung, um die Funktionalität des Multiplexers mittels einfachem *if-then-else*-Konstrukt zu modellieren. Passen Sie auch die Simulationsumgebung und die Konfiguration entsprechend an. Speichern Sie die neuen Dateien unter den Namen `mux_gate.vhd` und `mux_gate_simbox.vhd` (siehe Abschnitt 2.2.2.1) und fügen Sie sie dem bestehenden Projekt hinzu. Simulieren Sie anschließend die Schaltung des Multiplexers (mit allen möglichen Eingabewerten) in der gleichen Art und Weise wie beim NAND-Gatter.

Hinweis: Der Name einer Entity, z.B. SimBox, darf im kompletten Projekt nur einmal vorkommen.

Aufgabe 4. Mit der in der Vorbereitung bestimmten minimalen DNF soll nun von Ihnen eine Entity mit dem Namen `eq_gate` und eine zugehörige Architektur namens `eq_dataflow` für ein Äquivalenz-Gatter geschrieben werden. Verwenden Sie hierfür direkt die DNF mittels Datenflussbeschreibung, um die Äquivalenz-Funktion zu modellieren. Speichern Sie die entsprechenden Dateien ebenfalls separiert unter den Namen `eq_gate.vhd` und `eq_gate_simbox.vhd` (siehe Abschnitt 2.2.2.1) und fügen Sie sie dem bestehenden Projekt hinzu. Simulieren Sie anschließend die Schaltung vollständig.

Aufgabe 5. Ergänzen Sie die Datei `eq_gate.vhd` um eine weitere Architektur `eq_structure`, in die Sie Ihre Multiplexer-basierte Schaltung des Äquivalenz-Gatters eintragen. Verwenden

Sie dazu ausschließlich den 4:1 Multiplexer aus Aufgabe 3. Ändern Sie nun die Konfigurationsdatei so, dass sie die neu geschriebene Architektur benutzt, und simulieren Sie anschließend die Schaltung. Gibt es Veränderungen in den Ergebnissen im Vergleich zu den vorigen Simulationen?

Aufgabe 6. Beschreiben Sie die Entity und die Architektur eines Halbaddierers gemäß der ermittelten Gleichungen für die Ausgänge `c_out` und `s`. Ergänzen Sie die Simulationsumgebung und die Konfiguration. Simulieren Sie den Halbaddierer mit allen möglichen Eingangsbelegungen. Legen Sie für diese Aufgabe ein neues Projekt an.

Aufgabe 7. Erstellen Sie eine Datenflussbeschreibung für den Volladdierer gemäß den Ergebnissen aus den Vorbereitungsaufgaben. Simulieren Sie den Entwurf. Ergänzen Sie für diese Aufgabe das Projekt aus Aufgabe 6 um weitere VHDL-Dateien. Eine für den Volladdierer und eine für die Simulationsumgebung.