CPLDs

Vom Platinenlayout zum ersten Projekt

Jakob Holderbaum

Erstellt am 9. August 2008

Berufskolleg Olsberg
Paul-Oeventrop Str. 7
59939 Olsberg

Inhaltsverzeichnis

1 Vorwort

Dieser Bericht soll eine Einführung in die Programmierung eines bestimmten Typs der programmierbaren Bausteine, den sogenannten „CPLDs“ (*Complex Programmable Logic Devices*), bieten. Er bietet dem Leser einen Einblick in die Technik und den allgemeinen Aufbau solcher Bausteine, um später mit der Programmierung der CPLDs beginnen zu können. Dazu wird die *Hardware Description Language* „VHDL“ verwendet und näher erläutert. Des Weiteren erfolgt eine kurze Vorstellung der verwendeten Entwicklungsumgebung „ISE Design Tool“, welche der CPLD-Hersteller „Xilinx“ kostenfrei zur Verfügung stellt.

Nach dieser umfassenden Einführung in die Thematik folgt der praktische Teil des Projektes: Es muss ein sogenannter „JTAG Programmer“ entworfen werden, um den CPLD programmieren zu können. Zusätzlich wird eine Test- und Programmierplatine benötigt, auf der der Chip angebracht werden kann. Als Schaltungsvorlagen werden die Platinenlayouts von www.ulrichradig.de verwendet, welche ebenfalls in kostenloser Ausführung zum Herunterladen angeboten werden.
In der praktischen Anwendung wird ein „CPLD XC9536-15PC44“ der Firma „Xilinx“ verwendet.

Nach Fertigung und Test dieses Programmers wird ein erstes, kleines Projekt implementiert, um die einzelnen Arbeitsschritte und -vorgänge besser kennenzulernen. Abschließend werden dann noch einige praxisbezogene Beispiel-Projekte vorgestellt und erläutert.

Ich hoffe, dass dieses Dokument zum Verstandnis und zur Verinnerlichung der Materie beitragen kann.

2 Programmierbare Bausteine

2.1 Was ist ein programmierbarer Baustein?

Ein programmierbarer Baustein wird in der Digitaltechnik als PLD bezeichnet. PLD steht für die englische Bezeichnung *Programmable Logic Device*.
Prinzipiell kann man mit ihnen jede Schaltung umsetzen, deren Funktionsweise in boolescher Form ausdrückbar ist. Weiterhin lassen sich auch Formen von RTL-Schaltungsbeschreibungen (*Register-Transfer-Logik*) umsetzen. Bekanntes Beispiel einer RTL-Beschreibung ist die Finite-State-Machine. [1, S. 66]

2.2 Welche Arten von PLDs gibt es?

PLDs lassen sich grob in zwei Typen unterteilen. Es gibt auf der einen Seite Bausteine, welche aus einer UND-ODER-Matrix bestehen und auf der anderen Seite Bausteine die aus einer gewissen Anzahl logischer Blöcke bestehen, welche untereinander verknüpft werden können.

Zu der ersten Ausführung lassen sich zum Beispiel folgende Bausteine zählen:

- Programmable Read Only Memory (PROM)
  Der PROM enthält ein festes UND-Array, welches mit einem programmierbaren ODER-Array verknüpft wird.
- Programmable Array Logic (PAL) bzw. GAL
  Der PAL enthält ein programmierbares UND-Array welches in ein festes ODER-Array geführt wird
  Der GAL ist im Gegensatz zum PAL wiederbeschreibbar und dadurch erheblich flexibler.
- Programmierbare logische Anordnung (PLA)
  Ein PLA stellt in gewisser Weise eine Kombination der oben genannten Bausteine dar.
  Er enthält ebenfalls ein UND- und ein ODER-Array, mit dem Unterschied, dass beide programmierbar sind.

Die zweite, oben genannte Ausführung wird unter anderem von folgenden Bausteinen implementiert:

- Complex Programmable Logic Device (CPLD)
  Ein CPLD enthält verschiedene Blöcke. Ein Block ist ein vollwertiger PLA. Dieser kann mit den Ein- und Ausgangsblöcken oder programmierbaren Rückkopplungen verbunden werden. Zusätzlich stellt dieser Baustein für alle Ein- bzw. Ausgänge Flipflops zur Verfügung.
- Field Programmable Gate Array (FPGA)
  Prinzipiell ist ein FPGA einem CPLD sehr ähnlich. Er ist lediglich in seiner Gesamtheit wesentlich komplexer.

Hier enthalten die Blöcke Flipflops und Lookup-Tables. Außerdem stellt er komplexere Verknüpfunsmöglichkeiten zwischen den Blöcken zur Verfügung.

(Vergleiche dazu: [4])

2.3 Der CPLD XC9536

In dieser Versuchsreihe wird ein CPLD des Typs „XC9536“ der Firma „Xilinx“ verwendet.

Dieser zeichnet sich im Allgemeinen durch folgende Eigenschaften aus:

- Maximale Count- bzw. Toggle-Frequenz der Flipflops $f_{CNT} = 100MHz$
- 36 Zellen mit insgesamt 800 verwendbaren Gattern
- Bis zu 34 mögliche Pins zur Ein- bzw. Ausgabe (I/O Pins)
- Zwischen 5 und $15ns$ „pin-to-pin delay“
- Bis zu 10.000 Wiederbeschreibungen möglich

Die Typenbezeichnung des CPLDs beinhaltet einige wichtige Angaben über dessen Funktionsweise:

XC9536-15PC44		
XC95	$\longrightarrow$	CPLD-Familie: XC9500
36	$\longrightarrow$	Anzahl der Makrozellen
-15	$\longrightarrow$	„pin-to-pin delay“ in ns
PC	$\longrightarrow$	Package Typ
44	$\longrightarrow$	Anzahl der Pins

(Vergleiche dazu: [9])

2.3.1 Anwendungsgebiete

CPLDs sind gegenüber FPGAs deutlich schneller in der Implementierung kombinatorischer Schaltungen. Nachteilig ist, dass Pipelining durch Fehlen zusätzlicher Flipflops neben denen der ein und Ausgänge oft nicht möglich ist.
Aufgrund der hohen Bitbreite werden CPLDs gerne im Bereich der Decoder verwendet, wo schnelle Schaltungen oft zwingend erforderlich sind.

2.3.2 Erläuterung der Architektur

Um die Funktionsweise eines CPLDs zu erläutern, folgt eine schematische Darstellung dessen Architektur:

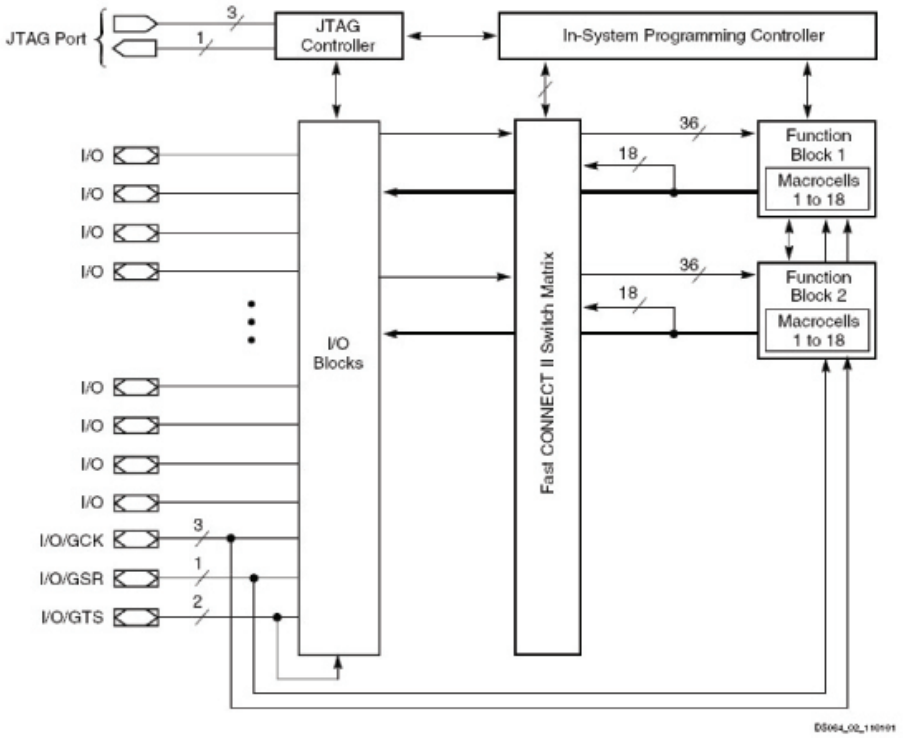

Abbildung 1: Architektur eines XC9536 Bausteins [8]

JTAG Controller Das „JTAG Protokoll“ wurde von der Joint Test Action Group entwickelt und als IEEE 1149.1 Standard definiert. Es dient zur Programmierung und zum Debuggen von PLDs, während sie in die Schaltung integriert sind. [5] Der „JTAG Controller“ interpretiert dieses Protokoll und kommuniziert mit dem „In-System Programming Controller“.

In-System Programming Controller Das sogenannte „In-System Programming“ bedeutet, dass das Gerät in eingebautem Zustand programmiert werden kann. Dieser Programmer arbeitet mit einem „4-Pin JTAG Protokoll“, welches durch den vorgeschalteten „JTAG Controller“ interpretiert wird. Wird der CPLD programmiert, werden alle Pins auf einen „Pull-Up-Widerstand“ geschaltet, um versehentliche Beschaltungen zu vermeiden.

Function Block (FB) Der FB spielt eine zentrale Rolle in der Architektur des CPLDs. Er enthält 18 voneinander unabhängige Makrozellen. Eine solche Zelle kann entweder eine kombinatorische- oder eine Register-Funktion implementieren. Die Ausgänge der 18 Zellen führen in die „FastCONNECT Switch Matrix“ und mit ihren korrespondierenden „Output Enabled“ Signalen in die I/O-Blocks. Der FB ist über eine 36 Bit breite Leitung mit

der Switch Matrix verbunden. Auf dieser liegen Signale von einzelnen I/O-Blöcken oder rückgekoppelten Makrozellen.

Macrocell Eine Makrozelle enthält wahlweise einen D- oder einen T-Flipflop (Flipflop mit „Toggle-Eingang“). Dieser kann überbrückt werden, wenn die Zelle eine rein kombinatorische Funktion implementiert. Des Weiteren ist jeder Flipflop an einen GCK (Global Clock) und einen GSR (Global Set/Reset) angeschlossen. Darüber können diese dann gesteuert werden. Im Einschaltmoment des CPLDs werden alle Register auf '0' gesetzt, wenn dies durch den Programmierer nicht anders definiert wurde.

FastCONNECT Switch Matrix Die Switch Matrix nimmt sowohl die Zustände der Makrozellen als auch die der I/O Blocks entgegen. Dadurch kann die Matrix jede Makrozelle mit jedem Signal treiben.

I/O Block (IOB) Die IOBs stellen ein Interface zwischen den Pins und der Switch Matrix dar. Sie sind kompatibel zu $5V$ CMOS/TTL und $3,3V$, was eine hohe Flexibilität gewährleistet. Der IOB gibt das anliegende Signal nicht direkt weiter, sondern verwendet ein selbst erzeugtes, korrespondierendes Signal, das auf Basis der Versorgungsspannung erzeugt wird.

(Vergleiche dazu: [9])

3 VHSIC Hardware Description Language

3.1 Ganz allgemein: Hardwarebeschreibungssprachen

Eine Hardwarebeschreibungssprache, kurz „HDL“, dient der möglichst effizienten und zeitnahen Entwicklung digitaler Schaltungen, was bei heutigen Geschwindigkeiten in der Hardwareentwicklung zwingend erforderlich ist.
Durch die Verwendung einer solchen Sprache entwirft man die Schaltung auf einem höheren Abstraktionsniveau. Es wird nicht mehr nur die reine boolesche Logik definiert, sondern das Verhalten, ähnlich einer imperativen Programmiersprache, wie beispielsweise C, beschrieben. Der geschriebene Code wird dann zu einer Netzliste compiliert. Eine Netzliste ist eine textuelle Beschreibung aller Verbindungen zwischen den Modulen des CPLDs. Dieser Vorgang wird in der Hardwareentwicklung allerdings nicht als Compilierung sondern als **Synthetisierung** bezeichnet.

Diese Abstraktion in der Entwicklung von Hardware bringt erhebliche Vorteile aber auch Nachteile mit sich.

Zu den Vorteilen zählen sicher die verkürzten Entwicklungszeiten und die effizientere Planung der Hardware. Durch das hohe Abstraktionsniveau kann es allerdings schnell dazu kommen, dass der Entwickler nicht synthetisierbare Logik beschreibt oder völlig unoptimierte Logik erzeugt. Die Entstehung und Behebung solcher Probleme werden nach und nach in den folgenden Kapiteln erläutert.

Zu den bekanntesten Sprachen gehören „ABEL“, „Verilog HDL“ und „VHDL“. „ABEL“ ist eine ältere Hardwarebeschreibungssprache die in der Industrie kaum noch Anwendung findet. „Verilog HDL“ und VHDL haben sich heute zu einem weltweiten Standart etabliert, wobei man unterscheiden muss, dass „Verilog HDL“ eher im amerikanischen und „VHDL“ eher im europäischen Raum verwendet wird. Dies ist auch der Hauptgrund aus dem in diesem Bericht „VHDL“ zur Entwicklung des CPLDs verwendet wird.

3.2 Ganz spezifisch: VHDL

Die Hardwarebeschreibungssprache „VHDL“ bringt einen großen Umfang an Funktionen mit sich um sehr effizient die Funktionsweise einer digitalen Schaltung zu beschreiben. Bei der Entwicklung mit „VHDL“ spricht man von sogenannten Modulen. Dieses besteht immer aus einer definierten Schnittstelle aus Ein- und Ausgängen sowie einer beschriebenen Funktionsweise.

In VHDL entspricht ein Modul einer einzelnen Datei mit der Extension `vhd`. Es enthält die folgenden, elementaren Konstrukte:

`library:` einbinden wichtiger Bibliotheken, vergleichbar mit der `#include`-Anweisung der Programmiersprache C.

`package:` erzeugen eigener Bibliotheken, welche Funktionen enthalten (In diesem Bericht wird nicht näher darauf eingegangen)

`entity:` enthält die Schnittstellenbeschreibung des Moduls. Man könnte sagen, dass hier die Ein- und Ausgänge einer (noch-) Blackbox definiert werden.

`architectur:` Verhaltensbeschreibung einer `entity`. Das Konstrukt enthält sowohl sequentielle- als auch Registerlogik. Eine `entity` kann unterschiedliche `architectures` besitzen, wenn durch das `configuration`-Konstrukt eine eindeutige Zuordnung stattfindet. Das ist allerdings nur bei mehr als einer `architecture` notwendig.

Aus diesen Elementen lässt sich das Grundgerüst eines einfachen VHDL-Moduls erzeugen:

```
1
2  library <lib_name>;
3  use <lib_name>.<package1_name>;
4  use <lib_name>.<package2_name>;
5
6  entity <entity_name> is
7     Port (   <port1_name>  : <Richtung>  <Typ>;
8              <port2_name>  : <Richtung>  <Typ>);
9  end <entity_name>;
10
11 architecture <architecture_name> of <entity_name> is
12
13 <signal Definitionen>
14
15 begin
16
17    <sequentielle- und Registerlogik>
18
19 end <architecture_name>;
```

Dieses Modul stellt den Rahmen einer Schaltung dar. In den folgenden Kapiteln wird nun gezeigt, wie sich dieser Rahmen „befüllen“ lässt, um eine echte Funktion zu beschreiben.

3.3 Die Schnittstellenbeschreibung

Der Aufbau einer Schnittstelle (`entity`) in VHDL setzt sich aus den Bezeichnungen der sogenannten `ports` (Ein- bzw. Ausgänge), deren Richtung und deren Typ zusammen. Es gibt vier verschiedene Richtungsangaben:

`in:` definiert diesen `Port` als Eingang.

`out:` definiert diesen `Port` als Ausgang.

`inout:` legt einen `bidirektionalen Port`fest, der sowohl als Ein- als auch als Ausgang verwendet werden kann.

`buffer:` definiert ebenfalls einen `bidirektionalen Port`, welcher allerdings nur von jew. einer Quelle getrieben werden kann.

(Vergleiche dazu: [2, S. 179])

Für den allgemeinen Gebrauch sind jedoch die ersten beiden Richtungsangaben völlig ausreichend.

Die Typangabe des `Ports` legt, vereinfacht ausgedrückt, dessen Bitbreite fest. Das IEEE empfiehlt die Verwendung der `STD_LOGIC`-Typen aus dem `IEEE-package`. Diese bieten zwei unterschiedliche Typen:

`STD_LOGIC:` ist ein einfacher boolescher Wert, also mit einem einzelnen Pin zu vergleichen.

`STD_LOGIC_VECTOR:` stellt einen mehrere Bit breiten `Bus` dar. Es gibt zwei unterschiedliche Varianten, die Bitbreite festzulegen:

`STD_LOGIC_VECTOR(n downto 0):` Der Bitvektor stellt ein Bitwort der angegebenen Breite dar (3 downto 0 = 4 Bit). Die Angabe durch `downto` sagt lediglich aus, dass sich das MSB[1] auf der linken Seite des Wortes befindet. Das entspricht der standardisierten Darstellung einer binären Zahl.

`STD_LOGIC_VECTOR(0 to n):` Diese Angabe definiert ebenfalls die Breite des Bitwortes auf die selbe Art und Weise. Hierbei befindet sich lediglich das LSB[2] auf der linken Seite.

Beispiel einer Schnittstellenbeschreibung eines NAND-Gatters:

```
1 library IEEE;
2    use IEEE.STD_LOGIC_1164.ALL;
3
4 entity nand_gate is
5    Port ( A : in  STD_LOGIC;
6           B : in  STD_LOGIC;
7           Q : out STD_LOGIC );
8 end nand_gate;
```

Da die beiden Eingänge hier differenziert betrachtet werden, macht es keinen Sinn einen Vektor zu verwenden. Deren Einsatz ist dann sinnvoll, wenn die anliegende Bitkombination als ganzes, beispielsweise als Zahl, interpretiert wird.

3.3.1 Definition von Vektoren

Definiert man einen Ein- oder Ausgang als Vektor, kann man diesen (mehreren Pin breiten) Port als Zahl betrachten und auf einer gewissen Abstraktionsebene

[1] Most Significant Bit
[2] Least Significant Bit

rechnen. Statt der Definition von jeweils vier einzelnen Ein- und Ausgängen zur Umrechnung einer 4 Bit breiten Zahl, werden zwei Vektoren erzeugt. Dadurch kann man diese direkt als Zahlen behandeln und ist nicht mehr dazu gezwungen auf Bit-Ebene zu arbeiten.

```
1 entity example is
2    Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
3           Q : out STD_LOGIC_VECTOR (3 downto 0) );
4 end example;
```

Die in Klammern gefasste Angabe hinter der Deklaration definiert die Lage des „Most-“ und „Least-Significant-Bit“. Die Bedeutung dieser Angabe wurde bereits in dem vorhergehenden Kapitel beschrieben.

3.4 Die Verhaltensbeschreibung

Die Verhaltensbeschreibung eines VHDL-Moduls setzt sich im Prinzip aus zwei Teilen zusammen. Der untere Teil, unter dem `begin`-Statement ist der eigentliche Programmcode. Darüber befinden sich die Variablen- und Signaldefinitionen. Dieser Aufbau ist vergleichbar mit dem eines herkömmlichen C-Programms. Dort werden ebenfalls zuerst die verwendeten Variablen vor dem eigentlichen Code deklariert.

3.4.1 Die Signaldeklaration

In der Hardwareentwicklung mit VHDL ist darauf zu achten, grundsätzlich keine Variablen zu verwenden, wenn der Code synthetisiert werden soll. Variablen sind nur in Ausnahmefällen synthetisierbar und sollten lediglich zur Simulation dienen.

Ein Signal wird folgendermaßen deklariert:

```
1 signal <name> : <type> := <value>;
```

Die Typisierung von Signalen erfolgt exakt wie die der im vorhergehenden Kapitel erläuterten Ports. Hier sind ebenfalls Vektoren möglich. Die Wertezuweisung am Ende der Deklaration ist allerdings optional.

3.4.2 Deklaration eigener Daten- bzw. Signaltypen

VHDL bietet die Möglichkeit eigene Signaltypen zu definieren. Diese sind mit den Enumerations aus anderen Programmiersprachen zu vergleichen. Eine Enumeration ist im Prinzip ein Datentyp der nichts weiter als ganzzahlige vorzeichenlose Werte speichern kann. Er bietet jedoch ferner die Möglichkeit Konstanten zu definieren die für die jeweiligen Zahlenwerte stellvertretend sind. Dazu ein einfaches Beispiel. Folgender Code definiert ein 4 Bit breites Signal und gibt ihm den Wert 2:

```
signal sig0 : STD_LOGIC_VECTOR(3 downto 0) := 2;
```

Möchte man nun diesen Vektor als Zustandsspeicher verwenden, müsste man eine Liste anfertigen, welche jeden Zustand einem numerischen Wert zuweist. Hier kommt die Enumeration ins Spiel. Diese wird folgendermaßen deklariert (über den Signaldeklarationen):

```
type <Type_Name> is (<Wert0>,<Wert1>,..);
```

Die in den Klammern definierten Worte sind stellvertretend für einen Zahlenwert. Das erste in der Liste auftauchende Wort für den Wert 0, das zweite für 1 und so weiter...

Der Deklarationskopf eines einfachen Zustandautomatens könnte dann wie folgt aussehen:

```
type STATES is (default,state0,state1);

signal currentState  : STATES := default;
signal nextState     : STATES := state0;
```

Die Zuweisungen der Werte statt eines numerischen können überall im Quelltext bei Signalen des definierten Typs durchgeführt werden.

3.4.3 Der eigentliche Code

Der eigentliche Code des Moduls beginnt unter dem `begin`-Statement. Her können einfache Signal- und Portzuweisungen definiert werden. Als eine Besonderheit, die VHDL von Programmiersprachen unterscheidet, ist hier zu erwähnen, dass alle Zuweisungen parallel behandelt werden und nicht sequentiell Ablaufen. Dies ist eine der großen Unterschiede von Hardwarebeschreibungssprachen, die den Umstieg von einer Programmiersprache zum teil stark erschweren können.

Als Beispiel für einen einfache Verhaltensbeschreibung soll die `architecture` des Nand-Gatters implementiert werden:

```
architecture Behavioural of nand_gate is
begin
   Q <= A nand B;
end Behavioural;
```

Die Bezeichnung „Behavioural“ der `architecture` ist eine Vorgabe der Xilinx ISE. Im Normalfall kann der Entwickler selbst über die Namensgebung der einzelnen Elemente entscheiden.

Als zweites Beispiel eine einfache `architecture` welche ein Signal verwendet:

```
architecture Behavioural of example is

```

```
3 signal sig0 : STD_LOGIC := '0';
4
5 begin
6    sig0  <= A and B;
7    Q     <= sig0 or C;
8 end Behavioural;
```

An diesem Beispiel lässt sich erkennen, dass Signale im Prinzip wie Ein- und Ausgangs-Ports behandelt werden können. Durch die Parallelität ist der Code gleichbedeutend zu `Q <= ( A and B ) or C;`.

Diese Form der Hardwarebeschreibung bietet im Prinzip immer noch keine sehr hohe Abstraktion, da den Ausgängen lediglich boolesche Ausdrücke zugewiesen werden. Aus diesem Grund gibt es sogenannte bedingte- und selektive Signalzuweisungen. Diese stellen eine Alternative zu if- und switch-case-Konstrukten dar, da diese nicht im eigentlichen Code verwendet werden können.

3.4.4 Arbeiten mit Vektoren

Vektoren werden in VHDL im Prinzip wie Arrays in C oder C++ behandelt. Durch Angabe eines Indizes kann gezielt auf die einzelnen Bits zugegriffen werden:

```
1 vec0(0) <= '1';
2 vec0(1) <= vec1(3);
```

Neben dieser Verwendung lassen sich Vektoren auch als Ganzzahlen betrachten, was das Rechnen deutlich abstrahiert und damit vereinfacht:

```
1 vec0 <= vec1 + 12;
2 vec2 <= vec1 + vec3;
```

Dies stellt den grundlegenden Umgang mit Vektoren in VHDL dar.

3.4.5 Verzweigende Strukturen

Die bedingte Signalzuweisung wird in folgender Syntax verfasst:

```
1 <Signal_oder_Port>     <= <Ausdruck0> when <Bedingung>
2                       else <Ausdruck1>;
```

Dies Bedeutet, dass `<Ausdruck0>` zugewiesen wird, wenn `<Bedingung>` gegeben ist. Ist dies nicht der Fall, wird `<Ausdruck1>` zugewiesen. Diese Form der Zuweisung ermöglicht schon eine gewisse Abstraktion. So kann zum Beispiel

`Q <= (A and B) and not C;` als
`Q <= (A and B) when C='0' else '0';`

dargestellt werden.

Die bedingte Zuweisung ist mit einem if-Konstrukt einer Programmiersprache zu vergleichen.

Die selektive Signalzuweisung wird durch folgende Syntax ausgedrückt:

```
with <beobachteter_Ausdruck> select
   Q3 <= <Ausdruck0> when <Wert0>,
         <Ausdruck1> when <Wert1>,
         ...;
```

Dies Bedeutet, dass der jeweilige Ausdruck zugewiesen wird, wenn sein zugehöriger Wert mit dem beobachteten Ausdruck übereinstimmt. Hiermit lässt sich Beispielsweise `Q <= A nand B` folgender Maßen darstellen:

```
with (A and B) select
   Q3 <= '0' when '1',
         '1' when others;
```

Die selektive Zuweisung ist mit dem switch-case-Konstrukt einer Programmiersprache zu vergleichen.

Damit wären im Prinzip die Grundlagen der parallelen Hardwareentwicklung in VHDL erläutert. Ein zweites, großes Thema sind die so genannten Prozesse.

3.5 Der Prozess

Ein Prozess wird in VHDL innerhalb einer Verhaltensbeschreibung deklariert. Alle in einer solchen Beschreibung deklarierten Prozesse werden wie auch alle andern Anweisungen parallel ausgeführt. Der in einem Prozess selbst enthaltene Code wird allerdings sequentiell abgearbeitet.

Einen solchen Prozess kann man sich im Prinzip als eine eigene Logikwolke innerhalb der gesamten Logikwolke vorstellen. Er kann neben den Ports des Moduls auf alle deklarierten Variablen und Signale zugreifen, und zusätzlich eigene definieren.

3.5.1 Aufbau eines Prozesses

Ein Prozess ist im Prinzip so ähnlich wie eine `architecture` aufgebaut.

```
<process_name> : process (<sensitivity_list>)
begin
end process <process_name>;
```

Wie auch bei der `architecture` wird der Code unter, und die Variablen- sowie Signaldeklarationen über dem `begin`-Statement platziert. Der Prozess-Name hat für die letztendliche Synthtisierung keinen Nutzen, er dient lediglich der Übersichtlichkeit während der Entwicklung, des Debuggings und der Simulation.

Der in einem Prozess definierte Code wird nicht permanent ausgeführt. Die Sensitivity-List enthält eine durch Komata getrennte Liste von Signalen und

Ports, auf die der Prozess „lauscht“. Sobald sich der Zustand einer der Signale oder Ports ändert, wird der sequentielle Code des Prozesses ausgeführt. Dieser Vorgang wird solange wiederhohlt, bis alle Signale innerhalb des Prozesses stabil sind, das heißt einen festen Wert haben. Hier liegt auch der große Unterschied zwischen Variablen und Signalen. Wird in einem Prozess einem Signal ein Wert zugewiesen, liegt dieser erst nach Beendigung des Prozesses an. Das heißt er kann erst im nächsten Durchlauf abgefragt werden. Dieses Verhalten weisen Variablen nicht auf, weshalb die Entwicklung mit ihnen statt Signalen augenscheinlich deutlich bequemer zu sein scheint. Weißt man einer Variable einen Wert zu, ist dieser sofort zugewiesen. Ein kleines Beispiel:

```
1  entity test is
2
3  Port (   A : in  STD_LOGIC;
4           Q : out STD_LOGIC );
5
6  end test;
7
8  architecture Behavioural of test is
9
10 signal sig0 : STD_LOGIC := '0';
11
12 begin
13
14    proc0 : process (A)
15    begin
16       sig0  <= A;
17       Q     <= sig0;
18    end process;
19
20 end Behavioural;
```

Simuliert man diesen Code indem man einen Takt auf Port A gibt, wird der Zustand A immer um eine Periodendauer verschoben zugewiesen. Das hängt damit zusammen, dass dem Ausgang Q immer der Wert des Signals sig0, welches dem des vorherigen Durchlaufs entspricht, zugewiesen wird. Dieses Verhalten sollte bei dem Entwurf von Hardware immer bedacht werden. Trotz dieser Schwierigkeiten m Umgang mit Signalen kann die Verwendung von Variablen zum Verlust der Synthetisierbarkeit der Schaltung führen!

In Prozessen dürfen keine bedingten oder selektiven Signalzuweisungen stattfinden. An deren Stelle treten die aus anderen Programmiersprachen bekannten if- und switch-case-Konstrukte.

Grundsätzlich gibt es zwei Arten von Prozessen: die rein kombinatorischen Prozesse und die Prozesse die neben ihrer enthaltenen Logik ein Register implementieren. In VHDL ist es nicht möglich, ein Register explizit zu erzeugen. Dies geschieht während der Synthetisierung automatisch. Daraus resultiert ein-

er der Gründe, aus denen man Prozesse sehr umsichtig programmieren sollte. Ein Grundsatz in der Hardwareentwicklung lautet: „Think Hardware!“. Damit soll ausgedrückt werden, dass man auch auf hoher Abstraktionsebene immer im Hinterkopf haben sollte, was auf Gatterebene gerade passiert. Das verhindert leichtsinnsfehler.

3.5.2 Konstrukte innerhalb von Prozessen

Ein Prozess kennt, im Gegensatz zu der herkömmlich parallelen Hardwarebeschreibung sowohl if- als auch switch-case-Konstrukte. Diese sind nach üblichen Schemata defniert.

Das if-Konstrukt wird durch folgende Syntax implementiert:

```
if <Ausdruck0> then
   <Code>
elsif <Ausdruck1> then
   <Code>
else
   <Code>
end if;
```

Ausdrücke werden durch die selbe Syntax wie auch bei booleschen Zuweisungen realisiert.

Das switch-case-Konstrukt ist folgendermaßen definiert:

```
case <Ausdruck> is
   when <Wert> =>
   when others =>
end case;
```

Der Zustand „others“ entspricht, wie sein Name vermuten lässt dem default-Fall eines switch-case-Konstrukts in C oder C++.

Bei Verwendung einer dieser Methoden sollte beachtet werden, dass unterschiedliche Hardware daraus synthetisiert wird. Ein if-else-Konstrukt erzeugt eine hierarchische Anordnung der Gatter. Das heißt, dass durch jedes weitere elsif eine weitere Gatterebene angefügt wird. Durch diesen Aufbau entsteht eine unter Umständen langsamere Logik, die aber mit Priotitäten arbeitet. Dadurch werden bei Zutreffen einer Bedingung die restlichen nicht mehr abgefragt.
Das case-Konstrukt entspricht einer parallelen Lösung, in der alle Möglichkeiten gleichzeitig abgefragt werden. Die Entscheidung zwischen den beiden Konstrukten sollte sich danach richten, ob ein bestimmtes Ergebnis sehr häufig und andere eher selten erwartet werden. Lässt sich eine solche Hierarchie nach Warscheinlichkeit aufstellen, sollte das if-else-Konstrukt verwendet werden. Hier muss natürlich die festgelegte Reihenfolge eingehalten werden (der warscheinlichste

Fall in das vordere if, usw..). Lässt sich diese Einstufung nicht vornehmen, ist das case-Konstrukt die richtige Wahl.

Schleifen bergen in der Hardwareentwicklung einige Probleme in sich. Man muss zwischen zwei Arten von Schleifen unterscheiden: den statischen und den dynamischen Schleifen. Eine statische Schleifen ist eine Schleifenform, bei der die Anzahl der Schleifensurchläufe zur Laufzeit bekannt ist. Diese erzeugen bei jedem Durchlauf neue Gatterlogik. Dynamische Schleifen beinhalten spezielle werte- oder ereignis-abhängige Abbruchbedingungen, welche wenn überhaupt durch Zustandsautomaten realisiert werden und in der Regel nicht sythetisierbar sind.

Statische Schleifen lassen sich durch das for-Konstrukt in VHDL erzeugen (Die Angabe des Labels ist optional):

```
1 <label> : for <Index> in <Bereich> loop
2    <Code>    -- <Index> kann innerhalb der
3              -- Schleife wie eine Variable verwendet werden.
4 end loop <label>;
```

Solche Schleifen lassen sich sehr gut nutzen, um einzelne Bits eines Vektors anzusprechen. Ein Beispiel dafür könnte folgendermaßen aussehen:

```
1 <label> : for i in (3 downto 0) loop
2    Q(i) <= '0';
3 end loop <label>;
```

Dazu muss allerdings gesagt werden, dass die obige Funktion deutlich einfacher und kürzer durch folgende Zuweisung realisiert werden kann:

```
Q <= (others => '0').
```

Es ist immer darauf zu achten, dass der Code, welcher zum Abbruch der Schleife führt (durch `exit` oder `next`) frei von dynamischen Faktoren wie Signalen, Variablen oder Ports gehalten wird damit das Modul synthetisierbar bleibt.

3.5.3 Der kombinatorische Prozess

Ein kombinatorischer Prozess zeichnet sich durch mehrere Dinge aus:

1. Alle Signale und Ports die innerhalb des Prozesses abgefragt werden stehen in der Sensitivity-List
2. Alle Abzweigungen enden in einer Zuweisung
3. Bei einer Verzweigung sind **alle**(!) Kombinationen abgedeckt
4. Es gibt keine Flankenerkennung

Der Entwurf eines solchen Prozesses erfordert eine sorgfältige Planung der darin enthaltenen Zuweisungen. Wird beispielsweise der Zustand eines Signals oder Ports abgefragt, welcher sich nicht in der Sensitivity-List befindet, ist der Prozess dazu gezwungen diesen Wert über ein Register zur Verfügung zu stellen. Dies kann differenzen zwischen Simulation und synthtisierter Netzliste oder falsche Werte der Abfrage erzeugen. Um diesen Fehler zu vermeiden, sollte stets darauf geachtet werden alle Signale und Ports einzutragen deren Werte innerhalb des Prozesses verwendet werden.

Da der Prozess im Prinzip eine Logikwolke implementiert, besitzt er sowohl Ein- als auch Ausgänge. Ist der Prozess nun kombinatorischer Natur, muss er immer (!) definieren, welche Werte die ausgehenden Signale und Ports besitzen, da es sonst zu undefinierten Zuständen kommen kann. Diese Fehlerquelle lässt sich geschickt umgehen, indem man in der obersten Zeile des Prozesses für alle ausgehenden Signale und Ports Default-Werte definiert. Das könnte Beispielsweise so aussehen:

```
1  proc0 : process (A,B,C)
2  begin
3
4  -- default
5  Q0 = '1';
6  Q1 = '0';
7
8
9  if A='1' and B='0' and C='1' then
10 Q0 = '0';
11 Q1 = '1';
12 end if;
13
14 end process;
```

Die Zuweisung an $Q0$ und $Q1$ würden ohne die default Zuweisung nur dann stattfinden, wenn der spezielle im Kopf des if-Konstrukt abgefragte Zustand der Signale gegeben wäre. Da es sich um einen registerlosen Prozess handeln soll, würde bei Nichtzuweisung der Signale nicht der ursprüngliche Wert des Signals beibehalten sondern in einen undefinierten Zustand gesprungen werden. Durch die sequentielle Abarbeitung des Codes entsteht durch diese „doppelte“ Zuweisung kein wirklicher Zeitverlust, da immer nur die letzte Zuweisung wirklich, physikalisch zugewiesen wird. Die Verwendung der Default-Zuweisungen erspart die Ausprogrammierung aller Kombinationen beziehungsweise der else-Abzweigungen und verhindert somit Fehler.

Es ist nicht möglich auf Flanken an den eingehenden Signalen und Ports zu lauschen, da eine Flankenerkennung lediglich durch Flip Flops realisiert werden kann. Diese Einschränkung ist allerdings keine sehr hinderliche Sache, da Flankenerkennung im Prinzip nur von taktgesteuerten Schaltungen verwendet wird.

Ein weiterer Punkt ist, dass immer darauf geachtet werden muss, keine Rückkopplungen zu erzeugen. Eine solche würde dazu führen, dass sich der Prozess immer wieder selbst aufruft und dadurch nie zur Ruhe kommt. Derartige Rückkopplungen lassen sich vermeiden, in dem man keinem in der Sensitivity-Liste stehenden Signalen innerhalb des Prozesses einen Wert zuweist.

3.5.4 Der Register-Prozess

Ein Register-Prozess weißt wiederum einige prägnante Merkmale auf:

1. Nur der Takt (und ggf. der asynchrone Reset) stehen in der Sensitivity-List
2. Bei einer Verzweigung müssen nicht **alle**(!) Kombinationen abgedeckt
3. Es gibt genau eine Flankenerkennung

Ein solcher Register-Prozess ist immer nach dem gleichen Grundschema aufgebaut:

```
1  reg: process (clk)
2  begin
3
4  if clk'event and clk='1' then
5
6     <code>
7
8  end if
9
10 end process;
```

Dieser Prozess implementiert ein auf positive Taktflanken getriggertes Register. Die Bedingung des if-Konstruktes fragt im Prinzip zwei Dinge ab: zum einen, ob sich das Signal `clk` verändert (durch `clk'event`) und zum anderen ob der Wert nun High ist. Würde auf Low abgefragt, wäre das Register auf negative Taktflanken getriggert. Wenn die Abfrage lediglich auf das `clk'event` beschränkt wird, würde es auf jede Flanke reagieren.
An diesem Beispiel lässt sich auch die Eigenschaft der Register-Prozesse erkennen, nicht zwangsläufig alle möglichen Bedingungen in einem if-Konstrukt zu erfassen.

Diesem einfachen Register fehlt allerdings noch eine entscheidende Fähigkeit: die Möglichkeit es zurückzusetzten. Bei der Implementierung eines Register-Resets in VHDL gibt es zwei Umsetzungsmöglichkeiten. Einen synchronen oder einen asynchronen Reset. Man kann nicht pauschal sagen welcher Reset wann am besten ist. Ein wichtiger Aspekt ist der, wie der Reset verwendet wird. Der synchrone Reset tritt erst in Kraft, wenn er aktiviert ist und am Takteingang der Triggerimpuls anliegt. Der asynchrone hingegen setzt das Register sofort

zurück. Arbeitet man in der gesamten Schaltung nur auf Basis eines gemeinsamen Taktes, ist es durchaus sinnvoll den Reset synchron zu realisieren, da die Werte im Normalfall erst nach den jeweiligen Taktimpulsen benöigt werden. Soll die Schaltung in einer Umgebung agieren in der Signale und Werte asynchron abgefragt werden, kann ein sofort wirkender Reset sinnvoll sein.

Der synchrone Reset lässt sich folgendermaßen Umsetzten:

```
1  reg: process (clk)
2  begin
3
4  if clk'event and clk='1' then
5
6     if reset='1' then
7        <Reset_Code>
8     else
9        <Set_Code>
10    end if;
11
12 end if
13
14 end process;
```

Der asynchrone hingegen nach folgendem Schema:

```
1  reg: process (clk,reset)
2  begin
3
4  if reset='0' then
5
6     if clk'event and clk='1' then
7        <Set_Code>
8     end if;
9
10 elsif
11    <Reset_Code>
12 end if;
13
14 end process;
```

Die beiden oben dargestellten Methoden stellen jeweils high-aktive Resets dar. Um diese umzukehren, sprich low-aktiv zu setzten, müssen alle `reset='0'` durch `reset='1'` und umgekehrt ersetzt werden.

Anhand dieser Grundlagen sollten Sie nun in der Lage sein, die später folgenden Beispiel-Anwendungen richtig zu verstehen und deren Funktionsweise nachzuvollziehen.

4 Fertigung des Programmers

4.1 Der JTAG-Programmer

Der JTAG-Programmer wird an den Rechner selbst über den LPT-Port angeschlossen. Er stellt die Verbindung zwischen dem programmierenden Computer und der CPLD-Platine her. Diese CPLD-Platine wird über eine 6-adrige Leitung an den Programmer angeschlossen.

4.1.1 Funktionsweise

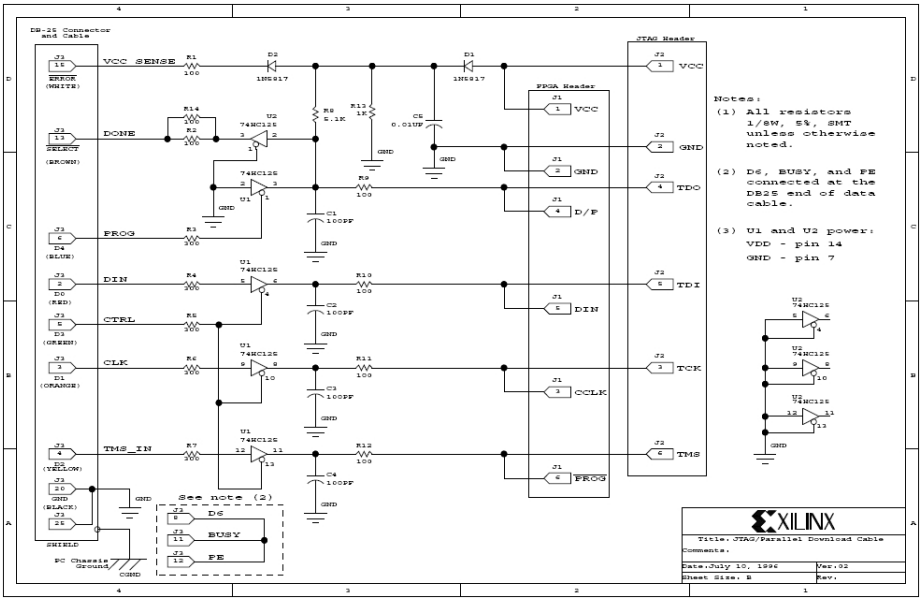

Abbildung 2: Schaltplan des JTAG-Programmers [6]

Prinzipiell leitet die Platine die Signale der LPT-Schnittstelle an die JTAG-Ausgänge weiter. Durch die Treiber auf den Leitungen wird das Signal entkoppelt und auf den korrekten Pegel gebracht. Die in Richtung Masse angebrachten Kondensatoren dienen der Filterung von Wechselspannungsanteilen auf der Leitung.

4.1.2 Die Stückliste

Die Stückliste beinhaltet alle Bauteile, die zur Fertigung der Platine notwendig sind:

Tabelle 1: Stückliste der JTAG-Programmer-Platine

Anzahl	Bezeichnung	Abk.	Wert	Einheit
1	LPT Male	X	-	-
1	Steckverbindung	JP	6	Pin
1	Kondensator	C	0,1	μF
4	Kondensator	C	100	pF
2	Widerstand	R	1	kΩ
5	Widerstand	R	300	Ω
7	Widerstand	R	100	Ω
2	Diode	D	1N4148	Typ
2	IC	IC	74HC125N	Typ

4.1.3 Das Layout der Platine

Das Layout (Abbildung 19) und die bestückte Platine (Abbildung 18) sind im Anhang „Abbildungen“ auf Seite 63 zu finden.

4.2 Die XC9536 Test- und Programmierplatine

Diese Platine dient als Schnittstelle zum CPLD selbst. Auf ihr befindet sich ein Sockel, der diesen Chip halten kann. Des Weiteren sind Pins und zum Teil auch LEDs an den I/O- und CLK-Pins angebracht, um die implementierten Verhaltensbeschreibungen zu testen.

4.2.1 Funktionsweise

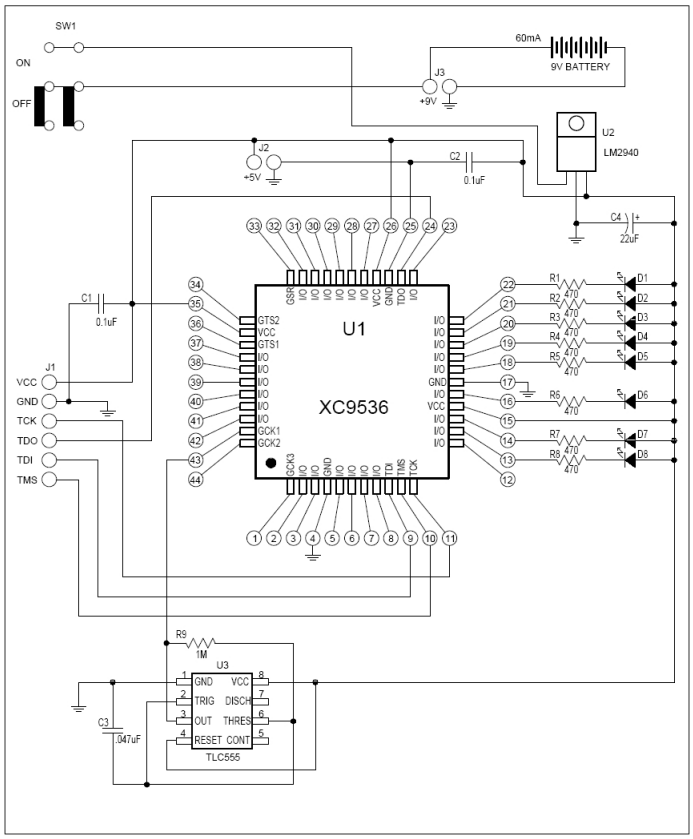

Abbildung 3: Schaltplan der XC9536 Test- und Programmierplatines [7]

Die eingehende Spannung wird durch den Festspannungsregler U2 auf konstanten $5V$ gehalten. Diese Spannung liegt an Pin 15 und 26 an. Die LEDs auf der rechten Seite des CPLDs sind über Widerstände an einigen I/O-Pins angeschlossen. Ist einer dieser Pins auf Low, entsteht eine Verbindung über den Pullup-Widerstand des CPLDs auf Masse, welche die Leuchtdioden zum Leuchten anregt. Damit ensteht eine lowaktive Anzeige des Ausgangs. Der NE555, gekennzeichnet durch U3, erzeugt mithilfe des Kondensators eine Frequenz von

$\sim 16Hz$, welche an Pin 43 des CPLDs geführt wird. Die restlichen Kondensatoren Richtung Masse dienen auch hier der Entstörung.

4.2.2 Die Stückliste

Die Stückliste beinhaltet alle Bauteile, die zur Fertigung der Platine notwendig sind:

Tabelle 2: Stückliste der XC9536-Platine

Anzahl	Bezeichnung	Abk.	Wert	Einheit
1	Spannungsklemmen	X1	2	Pin
1	Steckverbindung	JP5	6	Pin
4	Steckverbindung	JP1-JP4	11	Pin
1	Steckplatz f. XC9536	U1	44	Pin
1	Kondensator	C3	0,047	μF
1	Kondensator	C4	22	μF
2	Kondensator	C1/C2	0,1	μF
1	Widerstand	R9	1	MΩ
8	Widerstand	R1-R8	470	Ω
8	LED rot	D1-D8	1,2	V
1	Spg.-Festregler	U2	LM2940	Typ

4.2.3 Das Layout der Platine

Das Layout (Abbildung 21) und die bestückte Platine (Abbildung 20) sind im Anhang „Abbildungen“ auf Seite 63 zu finden.

Da dieses Layout der bestückten Oberfläche lediglich eine Nummerierung der Bauteile besitzt, sind in der obigen Stückliste die Bauteil-Bezeichnungen angegeben.

Alle in diesem Kapitel verwendeten Layouts und Schaltungen befinden sich auf der beigefügten CD im Verzeichnis `/layouts`.

4.3 Das Arbeiten mit den beiden Platinen

Bei der Einsetzung des CPLDs in die Fassung muss auf die korrekte Steckrichtung geachtet werden(gekennzeichnet durch einen Pfeil). Im nächsten Schritt muss der JTAG-Programmer an den LPT-Port des PCs angeschlossen und über das sechs-adrige Kabel mit der Test-Platine verbunden werden. Wenn man einen Bytestream aus der ISE Entwicklungsumgebung auf den CPLD brennen möchte, erkennt diese die angeschlossene Programmer-Platine automatisch und führt den Brennvorgang durch. Während des Programmierens ist auf eine kontinuierliche, externe Spannungsversorgung des CPLDs zu achten, da die Schnittstelle des PCs diese Aufgabe **nicht** übernimmt.
In Abhängigkeit der verlangten Ausgangs-Spannung an den I/O-Pins muss Pin 32 (V_{CCIO}) entweder mit 3,3V oder 5V (TTL) beschaltet werden.

5 Die Entwicklungsumgebung

5.1 Die XILINX Integrated Software Environment (ISE)

Die ISE Entwicklunsumgebung wurde speziell dazu entwickelt, PLDs der Firma Xilinx zu programmieren. Um einen Einblick in die Funktionsweise dieser Software zu bekommen, sollte zuerst die Standardoberfläche näher betrachtet werden.

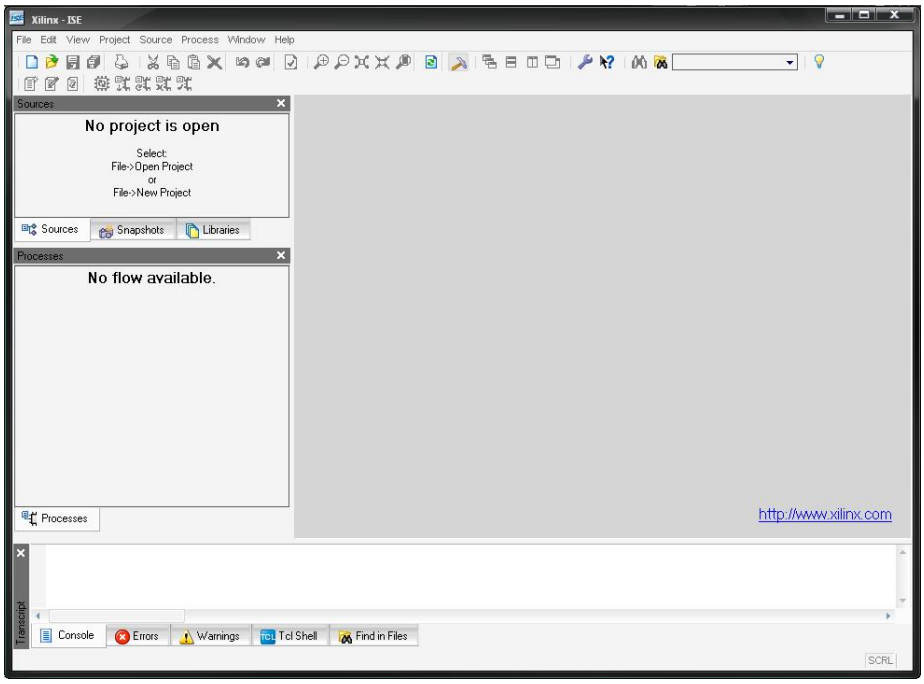

Abbildung 4: ISE Entwicklungsumgebung nach Start der Anwendung

- Sources
  Das Sources-Fenster enthält das geöffnete Projekt, sowie alle darin enthaltenen Dateien

- Processes
  Das Processes-Fenster bietet eine Übersicht über die mit einem mark ierten Element aus dem Sources-Fenster durchführbaren Aktionen und Tasks.

- Transcript
  Das Transcript-Fenster stellt die Konsole für Ausgaben aller Art dar. Hier werden beispielsweise die Meldungen des Synthetisierers („VHDL-Compiler“) ausgegeben.

5.1.1 Erstellen eines neuen Projektes

Um die Erstellung eines neuen Projektes anhand praktischer Beispiele zu erläutern, wird hier ein einfaches Beispiel-Projekt implementiert:
ein simples NAND-Gatter.

Die Erzeugung eines neuen Projekts geschieht durch Wählen des Eintrages **New Project** unter dem Menüpunkt **File**. Dort kann der Name und der Pfad des Projektes angegeben werden. In diesem Fall bezeichnen wir es als „nand“. Der „Top-Level Source Type“ gibt an, welche Form zur Beschreibung der Hardware verwendet werden soll. Da die in diesem Bericht implementierten Projekte ausschließlich in VHDL programmiert werden, sollte hier der Punkt HDL ausgewählt werden.
Nach einem Klick auf „Next“ werden einige Angaben zur hinterher verwendeten Architektur und der HDL verlangt. Dieser Bericht verwendet einen „XC9536 CPLD“, welcher zur Familie der „XC9500 CPLDs“ gehört.
Die Angaben „Speed“ und „Package“ ergeben sich aus der Bezeichnung des CPLDs: aus der Bezeichnung „XC9536-15PC44“ ergibt sich ein Speed von -15 und das Package PC44.
Da die Sprache VHDL verwendet werden soll, muss als „Synthesis Tool“ „XST (VHDL/VERILOG)“ ausgewählt werden. Als „Simulator“ wird der „ISE Simulator (VHDL/VERILOG)“ verwendet. Wie bereits oben erwähnt, ist die „Preferred Language“ VHDL. Die restlichen Einstellungen sollten so belassen werden.
Der Klick auf „Next“ führt zu der nächsten Eingabemaske. Diese bietet dem Benutzer die Möglichkeit, ein neues VHDL-Modul, also eine neue Source-Datei zu erzeugen. Durch einen Klick auf „New Source“ wird das benötigte Dialogfenster geöffnet. Auf diesem müssen die Art, der Name und der Pfad der neuen Datei gewählt werden. Dies ist standardmäßig der Pfad des Projektes und sollte nicht verändert werden. Als Datei-Typ sollte zur Erstellung „VHDL Module“ ausgewählt werden. Da der Begriff „nand“ in VHDL selbst als Ausdruck verwendet wird, kann das Modul so nicht bezeichnet werden. Als selbsterklärender Bezeichner würde sich „nand_gate“ anbieten. Vor dem Klick auf „Next“ sollte sichergestellt werden, dass der Punkt „Add to Project“ aktiviert ist.
Die nächste Eingabemaske erwartet die Schnittstellendefinition. In der ersten Spalte wird die Bezeichnung des Ports eingegeben. Es empfiehlt sich diese ausschließlich in Großbuchstaben zu schreiben, um sie in der späteren Funktionsbeschreibung eindeutig als solche zu identifizieren. In der zweiten Spalte muss die Richtung des Ports ausgewählt werden, sprich, ob er einen Eingang oder einen Ausgang darstellt. Die Bezeichnungen A und B werden für die beiden Eingänge verwendet. Für den Ausgang bietet sich die Bezeichnung Q an. In der zweiten Spalte muss nun, in Abhängigkeit der Funktion, definiert werden, ob es sich um einen Ein- oder Ausgang handelt. Die Aktivierung des Bus bleibt vorerst außen vor. Zu diesen und anderen, tiefer greifenden Eigenheiten der Sprache VHDL ist im nächsten Kapitel mehr zu lesen. Die Bezeichnung im Eingabefeld „Architecture“ sollte nicht verändert werden.
Durch einen erneuten Klick auf „Next“, wird eine Übersicht über die Einstel-

lungen gezeigt, welche mit „Finish“ bestätigt werden. Nach diesem Klick meldet sich das Programm mit der Frage, ob das Projektverzeichnis erstellt werden soll. Nach Bestätigung dieser Meldung befindet sich der Benutzer wieder in der Eingabemaske zur Erstellung neuer Source-Dateien. Da dies bereits geschehen ist, kann auch diese mit einem Klick auf „Next“ verlassen werden. Die nächste Seite ist nur relevant, wenn bereits bestehende Dateien in das Projekt übernommen werden sollen. Da dies nicht der Fall ist, kann die Seite übersprungen werden. Auf der letzten Seite befindet sich noch einmal eine Zusammenfassung aller vorgenommenen Projekteinstellungen, welche durch den Button mit der Aufschrift „Finish“ bestätigt werden können.
Nun wurde das erste neue Projekt mit einem vorbereiteten VHDL-Modul erzeugt.

5.1.2 Implementierung und Verifizierung des Codes

Um nun dem Modul seine Funktionalität zu geben, ist ein wenig VHDL-Code zu ergänzen. Der wichtige Ausschnitt aus dem VHDL-Modul sieht in etwa so aus:

```
1  entity nand_gate is
2      Port ( A : in  STD_LOGIC;
3             B : in  STD_LOGIC;
4             Q : out  STD_LOGIC);
5  end nand_gate;
6
7  architecture Behavioral of nand_gate is
8
9  begin
10
11
12 end Behavioral;
```

Unter dem Schlüsselwort `begin` muss nun folgender Code ergänzt werden:

```
1 Q <= A NAND B;
```

Um die Korrektheit des Codes zu überprüfen, bietet die ISE Umgebung eine Funktion mit der treffenden Bezeichnung „Check Syntax“ an. Markiert man im „Sources-Fenster“ das eben erstellte Modul, finden sich im „Processes-Fenster“ einige Optionen. Eine davon lautet „Implement Design“. Diese kann durch einen Klick auf das kleine [+] expandiert werden. In dem erscheinenden Menü muss „Synthesize-XST“ wiederum expandiert werden. Dort befindet sich dann der Punkt „Check Syntax“. Durch einen Doppelklick auf diesen wird der gesamte Code auf syntaktische Fehler hin untersucht. Die Erfolgs- bzw. Fehlermeldung wird in der unten liegenden Konsole ausgegeben.
Der eben angefügte Code sollte fehlerfrei sein und den Syntaxtest mit Erfolg bestehen.

5.1.3 Simulation des Codes

Xilinx stellt mit seinem „ISE-Simulator“ ein mächtiges Werkzeug zur Verfügung, um „HDL-Code“ effektiv vor der eigentlichen Synthetisierung zu simulieren und zu testen. Um ein fertiges Projekt zu testen, muss eine so genannte „Test Bench“ erzeugt werden. Auf dieser werden beispielsweise Einstellungen bezüglich des Clocks und der Simulationsdauer vorgenommen.
Die Erzeugung geht relativ einfach vonstatten. Um eine neue „Test Bench“ zu erzeugen, muss ein Rechtsklick auf das Projekt im „Sources-Fenster“ ausgeführt werden. In dem erscheinenden Kontextmenü muss nun „New Source“ ausgewählt werden. Das nun geöffnete Fenster ist das selbe, das auch bei der Erstellung des VHDL-Moduls verwendet wurde. Hier muss nun, anstelle des Punktes „VHDL-Module“, der Punkt „Test Bench Wave Form“ ausgewählt werden. Als Bezeichnung genügt ein treffendes „testbench“. Bestätigt man nun diese Eingaben, erscheint eine Auswahl der zu testenden Module. Da in diesem Projekt lediglich eines besteht, fällt eine Entscheidung weg und die Einstellung kann so übernommen werden. Das nächste Fenster ist wieder eine Zusammenfassung aller Einstellungen, welche durch den Button mit der Aufschrift „Finish“ bestätigt werden kann.

Nach Erstellung der „Test Bench“ öffnet sich ein neues Fenster. In diesem können detaillierte Einstellungen zum Verhalten des Moduls vorgenommen werden:

Clock Information Da dieses Projekt ein rein kombinatorisches Gatter implementiert, muss der Punkt „Combinatorial (or internal clock)“ ausgewählt werden. Durch diese Auswahl wird die Box mit der Bezeichnung „Clock Timing Information“ automatisch deaktiviert.

Global Signals Damit die Simulation auf Basis eines CPLDs abläuft, muss hier erst „GSR (FPGA)“ deaktiviert und dann „PRLD (CPLD)“ aktiviert werden.

In der unbeschrifteten Box unten rechts lässt sich noch die Länge der Simulation einstellen. Für unsere Zwecke sollten $1000ns$ aber vollkommen ausreichen. Mit einem Klick auf „Finish“ werden die Einstellungen bestätigt und die „Test Bench“ erzeugt.

Nun öffnet sich eine neue Ansicht, die einer Signaldarstellung über der Zeit ähnelt. Bevor an dieser die Einstellungen vorgenommen werden, sollte allerdings das Projekt einmal gespeichert werden (`[File] > [Save All]` oder `[File] > [Save]`).

Mit dieser „Test Bench“ kann nun das Simulationsverhalten definiert werden. Durch einen Klick auf ein hell markiertes Feld in einer Diagrammlinie lässt sich deren Zustand verändern. Um das Verhalten des Nor- Gatters zu simulieren, sollten die Eingänge folgende Wertetabelle implementieren:

Tabelle 3: Wertetabelle der Test Bench eines Nor-Gatters

B	A
0	0
0	1
1	0
1	1

Damit wären alle Rahmenbedingungen für eine Simulation geschaffen. Um sie zu starten, muss im „Sources-Fenster“ im Auswahlmenü „Sources for“ der Punkt „Behavioural Simulation“ ausgewählt werden. Im Fenster selbst ist dann die Datei „testbench (testbench.tbw)“ zu sehen. Diese sollte markiert werden. Im „Processes-Fenster“ muss nun der Punkt „Xilinx ISE Simulator“ expandiert werden. Durch einen Doppelklick auf das erschienene „Simulate Behavioural Model“ wird die Simulation gestartet.

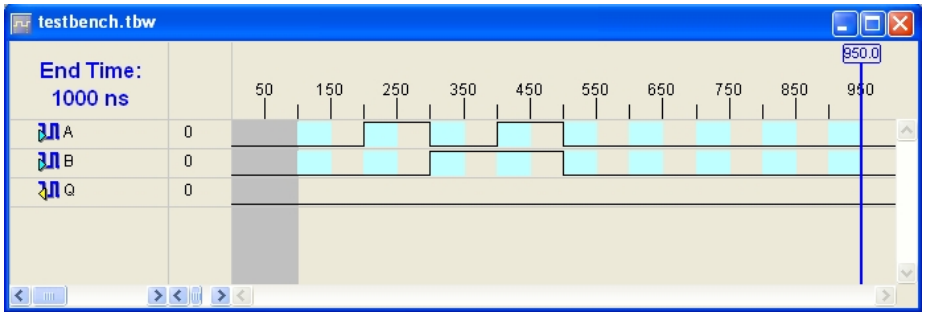

Abbildung 5: Test Bench eines Nor-Gatters

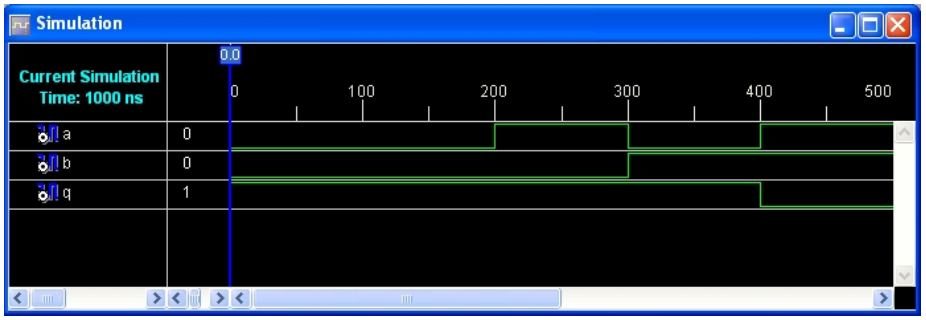

Abbildung 6: Simulation eines Nor-Gatters

5.1.4 Synthetisierung des Codes

Die Synthetisierung des VHDL-Codes wird durch den integrierten VHDL-Synthetisierer vorgenommen. Aus dem VHDL-Code muss eine JEDEC-Datei erzeugt werden, die direkt auf den CPLD gebrannt werden kann. Dieser Vorgang wird durch das Programm „iMPACT“ vorgenommen, welches ebenfalls von Xilinx stammt.

Der erste Schritt in der Synthetisierung ist die Pinbelegung. Alle in VHDL definierten Ports müssen je einem I/O-Pin zugeordnet werden. Wird das VHDL-Modul im „Sources-Fenster“ markiert, findet sich im „Processes-Fenster“ der Punkt „User Constraints“. Der Unterpunkt „Assign Package Pins“ öffnet nach kurzer Ladezeit die benötigte Eingabemaske („Xilinx PACE“).

Im Hauptfenster des Editors befindet sich eine quadratische, schematische Abbildung des Chips. Die einzelnen Pins werden durch die außenliegenden Symbole dargestellt. Bewegt man den Cursor über eins der Symbole werden Funktion und Nummer des Pins angezeigt. Es können lediglich die Pins zugeordnet werden, die als hellgrauer Kreis dargestellt werden und die Typbezeichnung „IO“ besitzen. Um nun die Ports den Pins zuzuordnen, muss die Tabelle im unten rechts gelegenen Fenster mit der Bezeichnung „Design Object List - I/O Pins“ verwendet werden.
In der Spalte „Loc“ kann einfach die Nummer des gewünschten Pins eingetragen werden. Wenn alle Ports zugeordnet sind, erfolgt die Speicherung durch einen Klick auf das Diskettensymbol oben rechts. In der erscheinenden Dialogbox muss „XST Default < >“ ausgewählt werden und mit „OK“ bestätigt werden. Damit ist die Pinbelegung abgeschlossen und das Fenster kann geschlossen werden.

Ein weiterer Punkt ist die Festlegung der sogenannten „Timing Constraints“ (frei übersetzt: „Beschränkungen des Zeitverhaltens“). Diese stellen Richtlinien für den verwendeten Takt sowie die Setup- und Hold-Zeiten der Flipflops dar.
Sie werden bei der letztendlichen Synthetisierung von CPLDs nicht berücksichtigt, sondern nur zur Überprüfung, ob das Design so synthetisierbar ist, verwendet. Sie finden eher Anwendung in der Entwicklung von ASICs oder FPGAs. Der Synthetisierer überprüft vor der Erstellung des „Bitstreams“, ob die eingestellte Frequenz, sowie die festgelegten Signallaufzeiten auf einem Chip des angegebenen Typs realisierbar sind. Ist das nicht der Fall, wird mit einer Fehlermeldung abgebrochen.
Die Kunst in der Verwendung von „Timing Constrains“ besteht darin, dass Laufzeitengpässe erkannt und berücksichtig werden müssen. Um die maximalen Laufzeiten festzulegen, muss der Karteireiter „Ports“ aufgerufen werden. Dort legt man die „Pad to Setup-Zeit“ der Eingänge fest. Diese gibt an, in welchem Zeitraum vor der nächsten Taktflanke der anliegende Pegel stabil sein muss (Es muss gewählt werden, ob sich der Wert auf positive oder auf negative Taktflanken bezieht). Um die Eingabemaske zu erreichen, muss ein Rechtsklick auf die Spalte „Pad to Setup“ in der Zeile des zu dimensionierenden Eingangs ausge-

führt, und in dem dort erscheinenden Kontextmenü „Pad to Setup“ ausgewählt werden.

Das Feld „OFFSET“ beinhaltet die festzulegende Setup-Zeit in der Einheit, die in dem dahinter befindlichen Dropdownfeld steht.

Danach müssen die „Clock to Pad-Zeiten“ der Ausgänge festgelegt werden. Diese dimensionieren den Zeitraum nach einer Taktflanke, nach welchem der am Pin anliegende Pegel stabil sein muss.

Das Einstellen der Frequenz geschieht unter dem Reiter „Global“.

Aus dem VHDL-Code, den „Pin Assignments“ und den „Timing Constraints“ muss nun eine Datei erzeugt werden, welche auf den CPLD gebrannt werden kann. Das Format, welches dazu verwendet wird, nennt sich „jedec“ (`*.jed`) und wird auch als Bitsream bezeichnet. Um diesen Prozess nun zu starten, muss ein Doppelklick auf „Implement Design“ im Processes-Fenster durchgeführt werden. Ist dieser Vorgang abgeschlossen, sollte das Fenster folgendes Erscheinungsbild aufweisen:

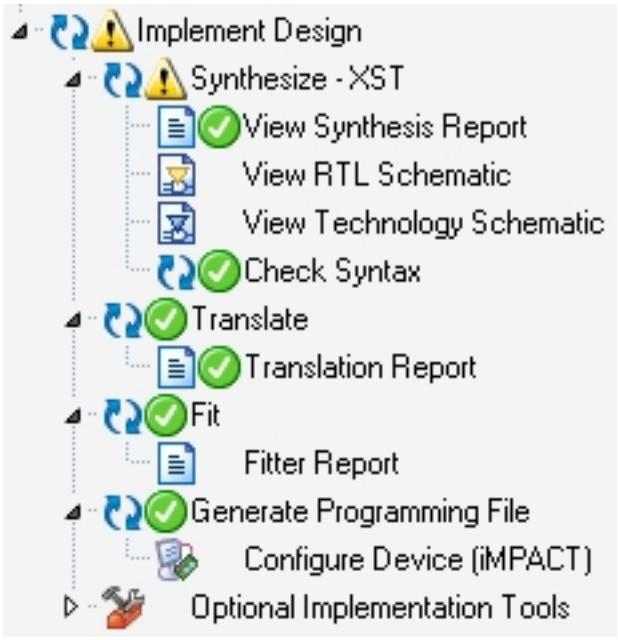

Abbildung 7: Abgeschlossene Implementierung des Chipdesigns

5.1.5 Brennen des CPLDs

Das unter Kapitel 5.1.4 beschriebene Verfahren führte zu der Erstellung eines Bitstreams, welcher nun auf den CPLD gebrannt werden kann. Dieser Vorgang des Brennens erfolgt durch das in der ISE integrierte Werkzeug „iMPACT“. Dieses wird unter `[Implement Design] > [Generate Programming File] > [Configure Device iMPACT]` durch einen Doppelklick aufgerufen.
Es öffnet sich eine Dialogbox, um die Art der Verbindung auszuwählen. Es sollte der Punkt „Configure devices using Boundary-Scan (JTAG)“ mit der Option der automatischen Verbindungssuche („Automatically Connect...“) aktiviert und über „Finish“ bestätigt werden. Der Computer verbindet sich nun automatisch mit dem Gerät. Ist dies abgeschlossen, öffnet sich ein Dateimenü, welches zur Auswahl der jedec-Datei auffordert. Sofern sich mehr als eine Datei mit der Extension `jed` in dem Projektverzeichnis befindet, sollte die, welche den gleichen Namen wie das VHDL-Modul trägt, gewählt werden: `nand_gate.jed`. Durch einen Rechtsklick auf den zentral dargestellten Chip öffnet sich ein Kontextmenü mit verschiedenen Optionen, welche nachfolgend teilweise erläutert werden:

Program: brennt den Bitstream auf den angeschlossenen Chip.

Verify: überprüft, ob der auf dem Chip enthaltene Bitstream mit dem durch das Projekt erzeugten übereinstimmt.

Erase: löscht den gesamten Speicher des Chips.

Blank Check: testet, ob der angeschlossene Chip leer, dh. nicht programmiert ist.

Readback: liest den auf dem Chip befindlichen Bitstream in eine jedec-Datei ein.

Get Device ID: überprüft, ob die ID des verbundenen Chips mit der im Projekt eingestellten übereinstimmt.

Um den Chip nun zu programmieren, muss der Punkt „Program“ ausgewählt werden. Es sollte, sofern der Chip **nicht** leer ist, die Auswahl „Erase Before Programming“ ausgewählt werden, um eine einwandfreie Funktionalität zu gewährleisten. Wird zusätzlich der Punkt „Verify“ aktiviert, wird nach dem Vorgang des Brennens überprüft, ob dieser korrekt durchgeführt wurde. Durch den Klick auf „OK“ sollte der Brennvorgang gestartet werden. Es öffnet sich ein Fenster, welches lediglich einen Fortschrittsbalken enthält. Ist dieser bei 100% angelangt, wird der erfolgreiche Brennvorgang durch die blau hinterlegte Meldung „Program Succeeded“ bekanntgegeben.

Damit wäre ihr erster CPLD programmiert!

6 Implementierung des ersten Projektes

6.1 Vorstellung des Projektes

Als erstes Projekt bietet sich eine einfache, leicht zu implementierende Schaltung an: ein synchroner Zähler. Um das Messen zu vereinfachen, wird dieser auf eine Bandbreite von 3 Bit reduziert.

Er soll bei jeder positiven Taktflanke einen Zählvorgang durchführen, wobei man durch einen zusätzlichen Eingang die Zählrichtung bestimmen kann. Außerdem soll der Zähler keine „Clipping-Funktion“ erhalten, das heißt, er wird, wenn er am Ende seines Zählbereichs angelangt ist, am gegenüberliegenden weiterzählen („Wrapping“). Die bisher bekannte Implementierungsform eines synchronen Zählers besteht in der Verwendung einer „Finite-State-Machine“. In VHDL definiert man eine solche nicht explizit, sondern beschreibt das Verhalten durch einen Register-Prozess.

6.2 Entwurf des VHDL-Moduls

Zu Beginn sollte ein neues ISE-Projekt erzeugt werden. Diesem sollte ein neues VHDL-Modul zugefügt werden mit einem Takteingang (`GCLK`), einer Richtungswahl (`DIR`) und dem letzlichen Ausgang (`OUTPUT - MSB: 2, LSB: 0`).
Nun muss in diesem Modul ein neues Signal erzeugt werden:

```
1 signal counter : STD_LOGIC_VECTOR (2 downto 0)
2   := (others => '0');
```

Dieses ist quasi der Zwischenspeicher für den Zählwert, welcher dann dem Ausgang zugeordnet wird. Da es sich bei `OUTPUT` um einen Ausgang handelt, kann von diesem nicht gelesen werden, weshalb das Signal notwendig ist.
Um nun das Register zu implementieren, muss ein neuer „taktsensitiver Prozess“ erzeugt werden:

```
1 process (GCLK)
2 begin
3 end process;
```

Um nun auch auf eine positive Taktflanke reagieren zu können, muss diese durch ein `if-Konstrukt` abgefragt werden:

```
1 if rising_edge(GCLK) then
2 end if;
```

Innerhalb dieser Bedingung muss nun geprüft werden, in welche Richtung der Zählvorgang stattfinden soll. Auch dies geschieht wieder durch ein `if-Konstrukt`:

```
1 if DIR = '1' then
2 else
3 end if;
```

Damit diese Schaltung ihre eigentliche Funktion erhält, muss der Zählvorgang in dieses letzte `if-Konstrukt` eingebettet werden:

```
if DIR = '1' then
   counter <= counter + 1;
else
   counter <= counter - 1;
end if;
```

Da das Signal, welches den aktuellen Zählerstand enthält, auch ausgegeben werden soll, muss der darin enthaltene Wert dem Ausgang zugeordnet werden. Dieser wird zusätzlich invertiert, da die LEDs auf der Platine low-aktiv sind:

```
OUTPUT <= not counter;
```

Damit wäre das für das Projekt benötigte VHDL-Modul fertig gestellt. Es sollte in seiner Gesamtheit folgendes Bild haben:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;



entity count is
    Port ( GCLK : in  STD_LOGIC;
           DIR : in  STD_LOGIC;
           OUTPUT : out  STD_LOGIC_VECTOR (2 downto 0));
end count;

architecture Behavioral of count is

signal counter : STD_LOGIC_VECTOR (2 downto 0)
  := (others => '0');

begin

   process (GCLK)
   begin
      if rising_edge(GCLK) then
         if DIR = '1' then
            counter <= counter + 1;
         else
            counter <= counter - 1;
         end if;
      end if;
   end process;

   OUTPUT <= not counter;

```

```
34 end Behavioral;
```

Zur Verifizierung sollte nachfolgend ein „Syntax-Check“ durchgeführt werden. Ist dieser erfolgreich abgeschlossen, kann es weiter zur Simulation gehen.

6.3 Simulation des Moduls

Zur Simulation sollte eine neue Testbench erzeugt werden. Für die, in diesem Bericht durchgeführte, Simulation wurden folgende Einstellungen verwendet:

Clock Timing Information	Rising Edge
Clock High Time	$10ns$
Clock Low Time	$10ns$
Input Setup Time	$1ns$
Output Valid Delay	$1ns$
Offset	$100ns$
Global Signals	PRLD(CPLD)
Clock Information	Single Clock (`GCLK`)
Initial Length of Test Bench	$1000ns$

Nach der Erstellung sollte ein Testmuster generiert werden. Da der einzige, veränderbare Eingang die Zählrichtung (`DIR`) ist, sollte diese nach der Hälfte der Simulation auf `High` gesetzt werden. Wird die Simulation dann durchgeführt, sollte folgendes Resultat zu sehen sein:

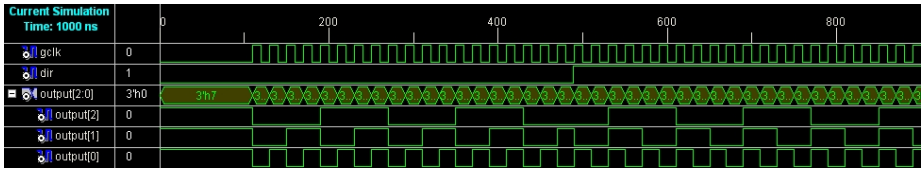

Abbildung 8: Simulation des 3-Bit-Zählers mit variabler Zählrichtung

6.4 Synthetisierung des Moduls

Bevor es zu der eigentlichen Synthetisierung kommt, müssen die „Pin Assignments“ festgelegt werden. Hier ist es dem Benutzer letztlich selbst überlassen, wie er diese zuordnet. Wichtig ist nur, dass der Takt (`GCLK`) auf Pin 5 gelegt wird, da dies der erste „Global Clock“ des CPLDs ist und dort zwangsläufig der Takt anliegen muss. Die „Timing Constraints“ müssen nicht eingestellt werden, da diese auf dem CPLD nicht berücksichtigt werden können.

Nach den Zuordnungen der Pins kann der eigentliche Synthetisierungsvorgang durch einen Doppelklick auf „Implement Design“ beginnen.

6.5 Brennen des CPLDs

Ist dieser erfolgreich abgeschlossen, kann „iMPACT“ für den eigentlichen Brennvorgang gestartet werden. (**Anm.: Wichtig ist, dass der CPLD während des Brennens mit konstanter Spannung versorgt wird.**)
Es muss die eben erzeugte „jedec-Datei“ ausgewählt werden, welche sich im Projektverzeichnis befindet. Durch einen Klick auf „Program“ wird der Brennvorgang gestartet. Wird dies nach Vervollständigung des Fortschrittbalkens durch ein „Program Succeeded“ bestätigt, kann es zum Test der gebrannten Hardware kommen.

6.6 Messung des gebrannten CPLDs

Der in diesem Bericht verwendete CPLD wurde an einem Funktionsgenerator mit $\sim 3kHz$ betrieben. Die an den einzelnen Pins anliegenden Pegel wurden mit einem „Ant8 USB Logic Analysator“ aufgenommen. Es folgen die Messergebnisse für jeweils beide Zählrichtungen:

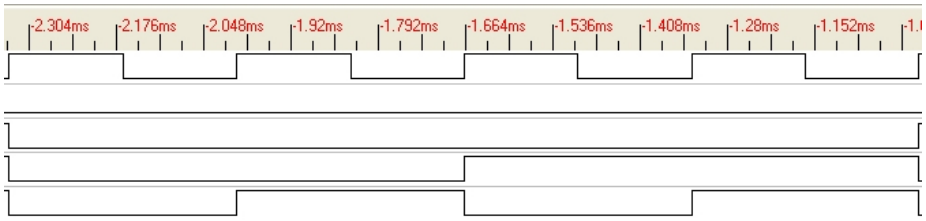

Abbildung 9: Messergebnisse des 3-Bit-Zählers in inkrementeller Richtung

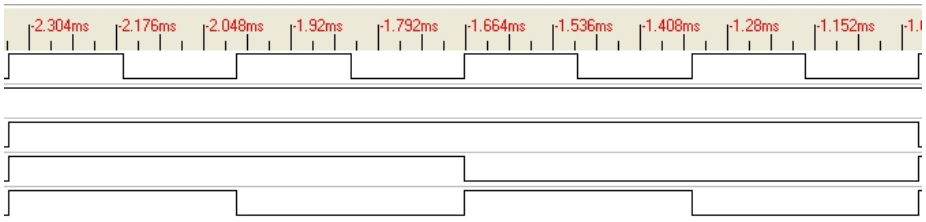

Abbildung 10: Messergebnisse des 3-Bit-Zählers in dekrementeller Richtung

Der oberste Kanal ist die Messung des Taktes, der darunter liegende die der Zählrichtung. Die Restlichen stellen, wie unschwer zu erkennen, den Zählausgang dar.

Damit wäre das erste Projekt, ein synchroner 3-Bit-Zähler, implementiert. Es folgt nun im nächsten Kapitel eine kleine Erweiterung des Zählers sowie weiterführende Beispielprojekte.

7 Weitere Beispielprojekte

7.1 Erweiterung des synchronen Zählers

7.1.1 Erweiterung des VHDL-Moduls

Der Zähler soll um einen synchronen Reset erweitert werden. Dazu sind nur geringfügige Änderungen an dem eigentlichen VHDL-Modul notwendig. Neben der Abfrage der Zählrichtung muss erst geprüft werden, ob der Reset betätigt ist. Es empfiehlt sich, ein neues Projekt zu erstellen und das bestehenden Modul aus dem vorhergehenden Projekt einzubinden. Ist dies geschehen, muss ein zusätzlicher Eingang in die Schnittstellenbeschreibung eingefügt werden.
Aus:

```
entity count is
    Port ( GCLK : in  STD_LOGIC;
           DIR : in  STD_LOGIC;
           OUTPUT : out  STD_LOGIC_VECTOR (2 downto 0));
end count;
```

wird dann:

```
entity count is
    Port (  GCLK : in  STD_LOGIC;
            DIR : in  STD_LOGIC;
            RESET : in STD_LOGIC;
            OUTPUT : out  STD_LOGIC_VECTOR (2 downto 0));
end count;
```

Im nächsten Schritt muss dieser neue Port im eigentlichen Prozess berücksichtigt werden. Dies geschieht durch ein `if-Konstrukt`, welches um die Abfrage der Zählrichtung gelegt wird:

```
if RESET = '1' then
   counter <= (others => '1');
else
      if DIR = '1' then
         counter <= counter + 1;
      else
         counter <= counter - 1;
      end if;
end if;
```

Durch diese Strukturierung ist der verwendete Reset „highaktiv“. Da das Signal `counter` dem eigentlichen Ausgang invertiert zugewiesen wird, wird es im Falle eines Resets komplett auf `High` gesetzt.

7.1.2 Simulation des Moduls

Die Testbench sollte mit den gleichen Einstellungen erstellt werden, die auch bei der Simulation der ersten Zählerversion verwendet wurden. Lediglich das

Testmuster ändert sich ein wenig. Es sollte insgesamt viermal die Richtung geändert werden, und zweimal zurückgesetzt werden, sodass alle Kombinationen abgedeckt sind (Vergleichbar mit zwei Spalten einer Wertetabelle, wenn: Spalte $\widehat{=}$ Eingang).

Die daraus resultierende Simulation sollte folgendermaßen aussehen:

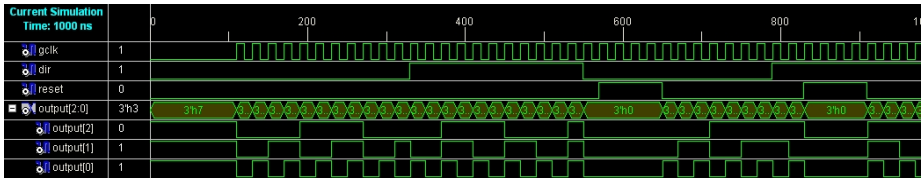

Abbildung 11: Simulation des 3-Bit-Zählers mit variabler Zählrichtung und synchronem Reset

7.1.3 Synthetisierung und Brennen des Moduls

Die Synthetisierung und das Brennen des Projektes geschehen an und für sich analog zu der Vorgehensweise der ersten Version. Es muss zusätzlich noch die Pin-Belegung des Reset-Ports festgelegt werden. Auch hier gibt es wieder eine Richtlinie. Der Reset darf bei dem verwendeten CPLD grundsätzlich nur auf Pin 39 gesetzt werden.

7.1.4 Messung des gebrannten CPLDs

Der CPLD wurde mit der gleichen Frequenz von $\sim 3kHz$ während der Messung betrieben.

Die Zuordnung der Kanäle ist hier ebenfalls identisch zu denen in der vorherigen Messung, mit dem Unterschied, dass der letzte, angefügte Kanal den Reset darstellt:

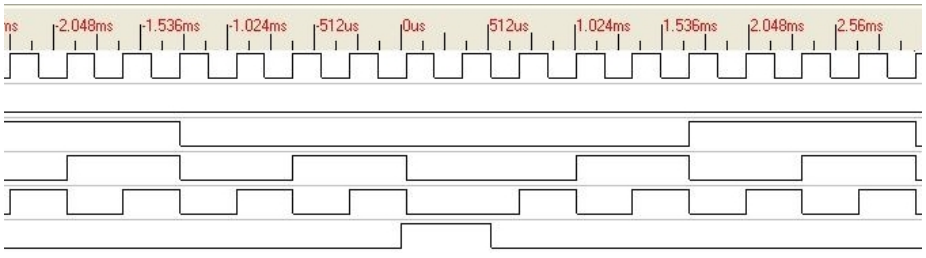

Abbildung 12: Messergebnisse des 3-Bit-Zählers mit synchronem Reset in inkrementeller Richtung

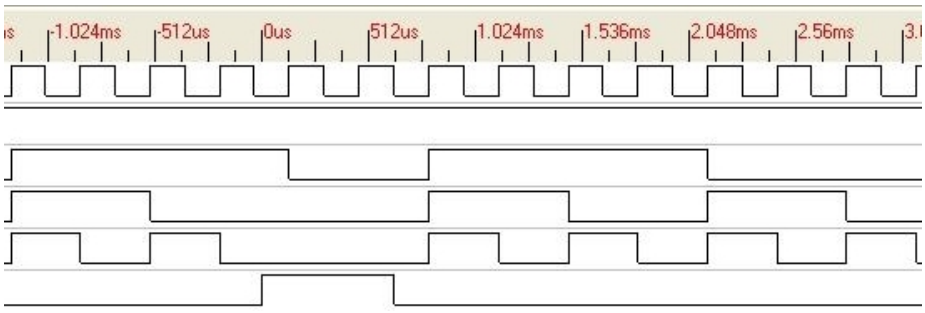

Abbildung 13: Messergebnisse des 3-Bit-Zählers mit synchronem Reset in dekrementeller Richtung

7.2 Parkhaussteuerung

7.2.1 Allgemeine Spezifikationen

Es soll eine einfache Parkhaussteuerung in Form eines Parkleitsystems auf einem CPLD entwickelt werden.

Folgende Spezifikationen sind gegeben:

- Ein Parkplatz mit fünf verfügbaren Parkplätzen
- In der Ein- / Ausfahrt des Parkhauses befinden sich hintereinander zwei Lichtschranken die nach der Durchfahrt eines Fahrzeugs kurzzeitig ein High-Signal ausgeben (standard 5V TTL Pegel). Die weiter innen gelegene Lichtschranke wird mit L_2, die äußere mit L_1 bezeichnet.
- Das Steuer-Modul soll die freien Parkplätze anzeigen sowie einen Fehlerzustand wenn der angezeigte Wert fehlerhaft ist (Under- / Overflow)
- Es soll eine Reset-Möglichkeit des Zählerstandes geben

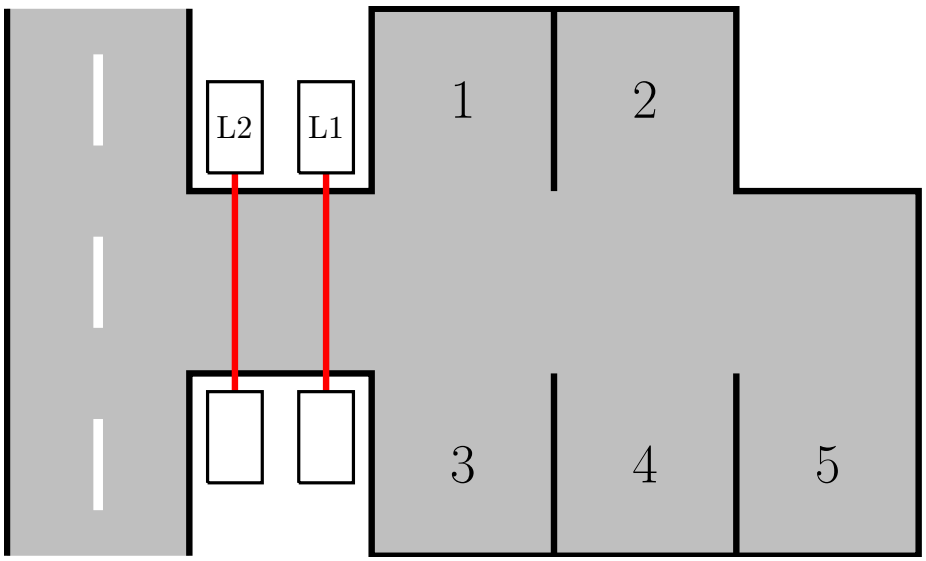

Abbildung 14: Schematische Skizze des Parkhauses

7.2.2 Vorbereitungen und Überlegungen

Der erste Schritt in der Entwicklung eines Chips besteht in der Schnittstellenüberlegung zwischen dem Gerät selbsts und „seiner“ Außenwelt.

- Auf welche Signale reagiert er?
- Welche Signale kann er weitergeben?
- Wie kann man diese Signalmenge mit der geringstmöglichen Anzahl von Pins realisieren?

Eingehende Signale: Die erste Überlegung hierbei sollte sein, ob ein Takt benötigt wird. Da die Steuerung in Form einer Finite-State-Machine realisiert werden soll, ist ein Takt definitiv notwendig.
Der zweite Punkt ist die Möglichkeit des Resets. Da in der Spezifikation von einer Möglichkeit des Resets gesprochen wurde, ist dieser ebenfalls notwendig.
Nach Klärung dieser essentiellen Punke der Schnittstellenvereinbarung besteht der nächste Schritt in der Planung der anwendungsspezifischen Schnittstellen. Da in der Spezifikation nur die Rede von zwei Lichtschranken ist, beschränkt sich die Schnittstelle auf diese.
Aus den nun zusammengestellten Informationen lassen sich vier Eingänge bilden, die hinsichtlich ihrer Anzahl nicht optimiert werden können;

- Takt
- Reset
- Lichtschranke 1
- Lichtschranke 2

Ausgehende Signale: In der Ausgangsspezifikation werden der Zählerstand und ein Fehlerfall aufgelistet. Da lediglich von einem Fehlerfall die Rede ist, wird für diesen auch lediglich ein Ausgang benötigt. Der Zählerstand richtet sich nach der benötigten Anzahl von Bits um den maximalen Wert darzustellen. In diesem Fall währen das drei Bit.

- Fehlerfall
- Zählerstand

Explizite Auflistung der Ein- und Ausgänge: Aus den eben aufgelisteten Ein- und Ausgängen lässt sich nun eine Liste aller Ports inklusive ihrer Bezeichnungen erstellen. Diese Auflistung soll die Grundlage des VHDL-Moduls darstellen:

Takt	clk
Reset	reset
Lichtschranke 1	L1
Lichtschranke 2	L2
Fehlerfall	err
Zählerstand (Bit 0)	cnt[0]
Zählerstand (Bit 1)	cnt[1]
Zählerstand (Bit 2)	cnt[2]

7.2.3 Planung der Umsetzung

Um ermitteln zu können, ob ein Fahrzeug das Parkhaus verlässt oder hineinfährt, stehen lediglich zwei Lichtschranken zur Verfügung. Der CPLD muss also in der Lage sein, anhand der Signale der Lichtschranken die Fahrtrichtung des Fahrzeugs zu erkennen.
Das dazu notwendige Verfahren lässt sich meines Erachtens nach in einfachster Form durch eine FSM realisieren, da die unterschiedlichen **Zustände** der Lichtschranken unterschiedliche Reaktionen erfordern.
Im Prinzip muss das Zustandsdiagramm drei essentielle Zustände enthalten:

- Einen Default-State in dem die Steuerung auf Input wartet
- Einen Zustand der den Zähler inkrementiert
- Einen Zustand der den Zähler dekrementiert

Da sich in dem ersten Projekt (Der einfache Zähler) herausstellte, dass bei der Zählerentwicklung keine FSM von Nöten ist muss die Funktionsweise der zählenden Zustände nicht näher im Diagramm beschrieben werden.

Aus diesen Vorüberlegungen lässt sich folgendes State-Transition-Diagram erzeugen:

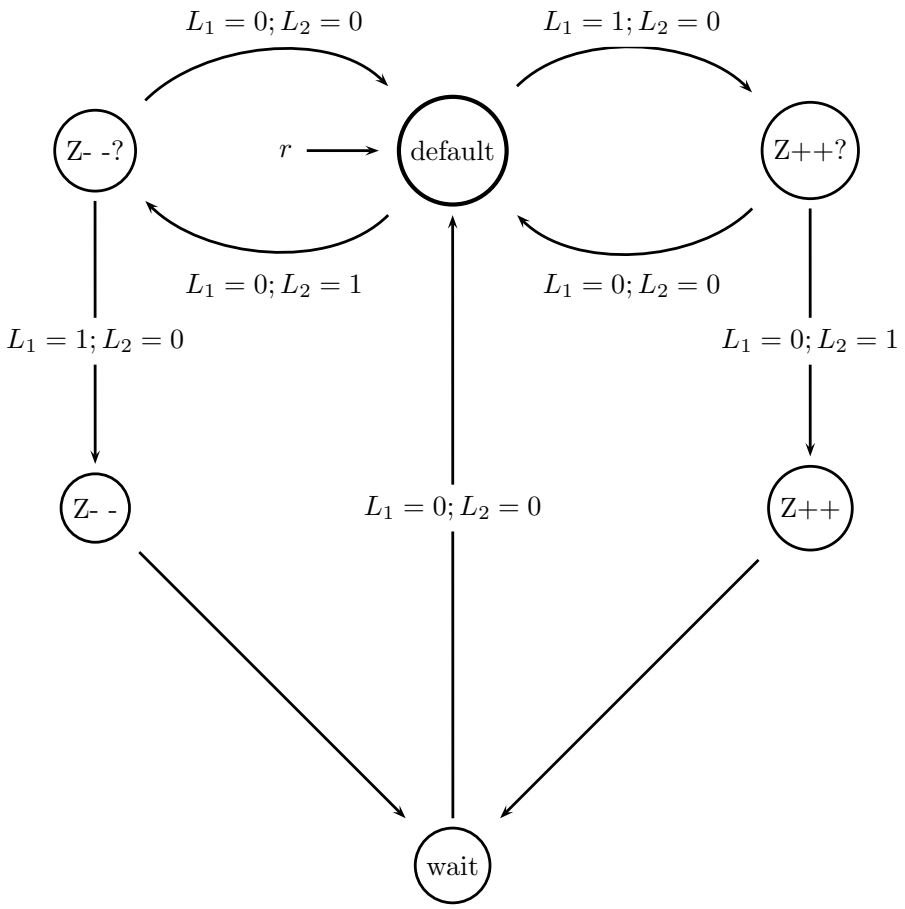

Abbildung 15: State-Transition-Diagram der FSM zur Erkennung der Fahrtrichtung

Erklärung des Diagramms: Wird erst die Lichtschranke L_1 aktiviert, springt die FSM in den Zustand „Z++?“. Dieser wartet darauf, dass die zweite Lichtschranke betätigt wird bevor die Erste auf einen Low-Pegel schaltet. Schalten beide auf einen Low-Pegel, wurde die erste versehentlich ausgelöst oder das Fahrzeug bewegte sich mit einer so niedrigen Geschwindigkeit, dass es von L_1 zu L_2 mehr Zeit brauchte, als der High-Pegel anhält. Daher sollte die Dauer dieses Pegels durch die Potentiometer der Monoflops in einer Größenordnung dimensioniert werden, welche diesen Fehler im Prinzip ausschließt.
Wird die Zweite betätigt, springt der Zustands-Automat in den Zustand „Z++“, welcher den Zähler inkrementieren soll. Damit keine Schleife entsteht, läuft dieser Zustand immer in den Zustand „wait“, welcher darauf wartet, dass sich die Signale der Lichtschranken beruhigt haben (d.h. beide auf Low-Pegel).

Dieser Vorgang läuft genauso in die andere Richtung, mit dem Unterschied, dass die Lichtschranken in entgegengesetzter Reihenfolge abgefragt werden.

7.2.4 Die Umsetzung in VHDL

Das Grundgerüst: Der erste Schritt besteht in der Beschreibung der Schnittstelle. Das VHDL-Modul soll „parkhaus“ heißen:

```
entity parkhaus is
   Port (   reset : in  STD_LOGIC;
            clk   : in  STD_LOGIC;
            L1    : in  STD_LOGIC;
            L2    : in  STD_LOGIC;
            cnt   : out  STD_LOGIC_VECTOR (2 downto 0);
            err   : out  STD_LOGIC);
end parkhaus;
```

Zu einer Schnittstelle gehört immer auch eine *architecture*:

```
architecture Behavioral of parkhaus is
begin
end Behavioral;
```

Um eine Finite-State-Machine in VHDL zu beschreiben, sind vorläufig zwei Signale notwendig: eines, welches den aktuellen Zustand speichert und ein weiteres, welches den folgenden beinhaltet. Damit ein fehleranfälliges Codieren der Zustandsbezeichnungen zu einzelnen Bits vermieden werden kann, wird ein neuer „Datentyp“ für Signale erzeugt, welche verschiedene Werte (die Zustandsnamen) annehmen kann:

```
type states is (  def,   -- Default State
                  Zm,    -- Z-?
                  Zmm,   -- Z-
                  Zp,    -- Z++?
                  Zpp,   -- Z++
                  wt);   -- wait
```

Intern bedeutet diese Enumeration nichts anderes, als das jeder in den Klammern aufgezählte Begriff stellvertretend für eine Zahl, beginnend bei 0 ist. Somit stellt *def* eine 0 und *wt* eine 5 dar. Somit ist ein Signal diesen Typs nichts weiter als ein 3 Bit breiter Vektor.

Zusätzlich zu diesen FSM-spezifischen Angaben benötigen wir noch weitere Register. Um zu ermitteln, wieviele Register (bzw. wieviel Bit) benötigt werden, muss man sich überlegen, welche Werte gespeichert werden, d.h. nach

einem Taktwechsel unverändert sein müssen. Dazu zählt zum einen der Zählerstand und zum anderen der Fehlerfall.

Nun kann der Kopf der *architecture* mit den Signaldefinitionen fertiggestellt werden:

```
architecture Behavioral of parkhaus is
   type states is (  def,   -- Default State
                     Zm,    -- Z-?
                     Zmm,   -- Z-
                     Zp,    -- Z++?
                     Zpp,   -- Z++
                     wt);   -- wait

   -- Speicherung des aktuellen Zustands
   signal state      : states;
   -- Speicherung des folgenden Zustands
   signal state_next    : states;

   -- Register f. Zählerstand
   signal cntbuf : STD_LOGIC_VECTOR(2 downto 0) :="101";
   -- Register f. Fehlerfall
   signal errbuf : STD_LOGIC :='0';
begin
End Behavioural
```

Die technische Umsetzung der FSM ist ebenfalls simpel und strukturiert aufgebaut, da diese lediglich aus zwei Prozessen besteht:

- Ein kombinatorischer Prozess, welcher das Signal *state_ next* abhängig von den Eingangswerten festlegt: „state_transition“
- Ein getakteter Register-Prozess, der die Logik der einzelnen Zustände ausführt: „state_logic“

Der erste Schritt besteht nun darin, die Zustandübergänge in dem kombinatorischen Prozess zu realisieren. Dazu wird zuerst das erforderliche Grundgerüst geschaffen:

```
state_transitions: process(state,L1,L2)
begin
   -- default state
   state_next <= def;

   case state is
      when def    =>
         state_next <= ..;

```

```
10       when Zm  =>
11          state_next <= ..;
12
13       when Zmm    =>
14          state_next <= ..;
15
16       when Zp     =>
17          state_next <= ..;
18
19       when Zpp =>
20          state_next <= ..;
21
22       when wt     =>
23          state_next <= ..;
24
25    end case;
26
27 end process state_transitions;
```

In den einzelnen „case-Fällen“ muss nun durch einige if-Konstrukte der entsprechende Folgezustand definiert werden. Vor der Zuweisung in den if-Konstrukten sollte eine default-Zuweisung erzeugt werden, um die Ausprogrammierung aller möglichen Kombinationen einzusparen:

```
1  case state is
2     when def    =>
3        -- default Zuweisung
4        state_next <= def;
5
6        if    L1='1' and L2='0' then
7
8           state_next <= Zp;
9
10       elsif L1='0' and L2='1' then
11
12          state_next <= Zm;
13       end if;
14
15    when Zm  =>
16       -- default Zuweisung
17       state_next <= Zm;
18
19       if    L1='0' and L2='0' then
20
21             state_next <= def;
22
23       elsif    L1='1' and L2='0' then
24
25             state_next <= Zmm;
```

```
26        end if;
27
28     when Zmm    =>
29        -- default Zuweisung
30        state_next <= wt;
31
32     when Zp     =>
33        -- default Zuweisung
34        state_next <= Zp;
35
36        if    L1='0' and L2='0' then
37
38           state_next <= def;
39
40        elsif    L1='0' and L2='1' then
41
42           state_next <= Zpp;
43        end if;
44
45     when Zpp =>
46        -- default Zuweisung
47        state_next <= wt;
48
49     when wt     =>
50        -- default Zuweisung
51        state_next <= wt;
52
53        if  L1='0' and L2='0' then
54
55           state_next <= def;
56        end if;
57
58  end case;
```

Der nun erstellte kombinatorische Prozess sorgt dafür, dass bei jeder Änderung eines Zustandes oder Einganges der Wert von *next_ state* angepasst wird.

Damit wäre der Rahmen für die FSM geschaffen. Um nun die endgültige Funktion zu erreichen, müssen die Zustände noch „mit Leben gefüllt werden“ und der Wechsel zwischen diesen realisiert werden. Dazu dient der getaktete Prozess. Dieser fragt bei jeder positiven Taktflanke ab, in welchem Zustand sich der Automat befindet und reagiert dementsprechend. Außerdem sorgt er für das Erreichen des nächsten Zustandes. Das dazu nötige Grundgerüst sieht folgender Maßen aus:

```
1  state_logic: process(clk)
2  begin
3     if clk'event and clk='1' then
```

```
4     -- Hauptschleife
5         -- positiver Reset
6         if reset='1' then
7             -- reset code...
8         else
9             -- main code...
10        end if;
11    end if;
12 end process state_logic;
```

Der nächste Schritt besteht meines Erachtens in der Überlegung, welchen Funktionalität der „Reset-Code“ beinhaltet. Der wichtigste Aspekt ist sicher, den aktuellen Zustand auf den Default-State zu setzten. Des weiteren müssen die in den verwendeten Registern gespeicherten Werte zurückgesetzt werden. Der Zählerstand müsste demnach also auf die Zahl fünf (freie Parkplätze) gesetzt werden und der Error-State auf null (da standardmäßig kein Fehler vorhanden ist). Der Code zur Erreichung dieser Funktionalität sieht wie folgt aus:

```
1 state <= def;      -- erreichen des Defaultstates
2 cntbuf   <= "101";    -- den Zähler auf fünf setzen
3 errbuf   <= '0';      -- den Fehlerfall entfernen
```

Nun muss die eigentliche „Hauptschleife“ gefüllt werden. Diese muss zum einen den Zustandswechsel realisieren und zum anderen abfragen, in welchem Zustand sie sich gerade befindet:

```
1  -- state-Register-Transition
2  state <= state_next;
3
4  if state=... then
5
6     - code...
7
8  elsif state=... then
9
10    - code...
11
12 end if;
```

Es gibt zwei Zustände in denen eine echte Funktionalität gegeben ist: Das Hoch- ($Z{+}{+}$) und das Herunterzählen (Z- -). Durch den eingeschränkten Zählbereich sowie den Fehlerabfang reicht es hier nicht, einfach den Zählerstand zu in- beziehungsweise dekrementieren. Der Fehlerfall soll folgendes Verhalten implementieren: Ist der Zähler an seinem Maximal- bzw. Minimalwert angelangt und soll weder de- noch inkrementiert werden, wird statt einer Veränderung des Zählers der Fehlerfall gesetzt. Ist der Fehlerfall gesetzt wird er nur durch einen Zählvorgang in die richtige Richtung aufgehoben. Beispiel: Steht der Zähler auf

fünf und der Error auf eins, kann dieser nur durch einen Dekrementier-Vorgang an den Zähler deaktiviert werden.

In der Funktionsbeschreibung des Zustandes *Z++* müsste also zuerst abgefragt werden, ob der Maximalwert bereits erreicht ist um gegebenfalls den Error-State zu setzen:

```
1 if cntbuf="101" then
2    errbuf <= '1';
```

Danach muss abgefragt werden, ob vielleicht der Fehlerzustand im Minimalwert erreicht wurde. In diesem Fall muss lediglich der Error-State aufgehoben werden:

```
1 elsif cntbuf="000" and errbuf='1' then
2    errbuf <= '0';
```

Treffen weder der eine noch der andere Fall zu, kann der Zähler problemlos erhöht werden:

```
1 else
2    cntbuf <= cntbuf + 1;
3 end if;
```

Das Verhalten des Zustandes *Z-* - ist grundsätzlich identisch, es müssen lediglich die Werte sowie die Zählrichtung verändert werden. Der gesamte Block sieht nun folgendermaßen aus:

```
1  if state=Zpp then
2
3     if cntbuf="101" then
4        errbuf <= '1';
5     elsif cntbuf="000" and errbuf='1' then
6        errbuf <= '0';
7     else
8        cntbuf <= cntbuf + 1;
9     end if;
10
11 elsif state=Zmm then
12
13    if cntbuf="000" then
14       errbuf <= '1';
15    elsif cntbuf="101" and errbuf='1' then
16       errbuf <= '0';
17    else
18       cntbuf <= cntbuf - 1;
```

```
19     end if;
20
21 end if;
```

Um nun die Werte in den Registern an die im Port definierten Ausgänge zu bringen, muss eine weitere Zuweisung unterhalb der Prozesse stattfinden:

```
1 cnt <= cntbuf;
2 err <= errbuf;
```

Da das Testboard des CPLDs low-aktive LEDs besitzt, empfiehlt es sich, die Zuweisung nach der Simulation, sprich vor der eigentlichen Synthetisierung zu invertieren:

```
1 cnt <= not cntbuf;
2 err <= not errbuf;
```

Das nun entstandene VHDL-Modul implementiert die in der Spezifikation verlangten Eigenschaften der Steuerung.

7.2.5 Die Simulation

Zur Simulation muss eine Test Bench angelegt werden. Um eine möglichst große Folge von Signaländerungen darstellen zu können sollte ein schneller Takt gewählt werden:

Clock High Time	$5ns$
Clock Low Time	$5ns$
Input Setup Time	$1ns$
Output Valid Delay	$1ns$

Im Prinzip gibt es zwei kritische Situationen die überprüft werden müssen:

1. Der Übergang vom leeren Parkhaus in den Error-State und wieder zurück.

2. Der Übergang vom vollen Parkhaus in den Error-State und wieder zurück.

Der erste Fall ist relativ schnell simuliert:

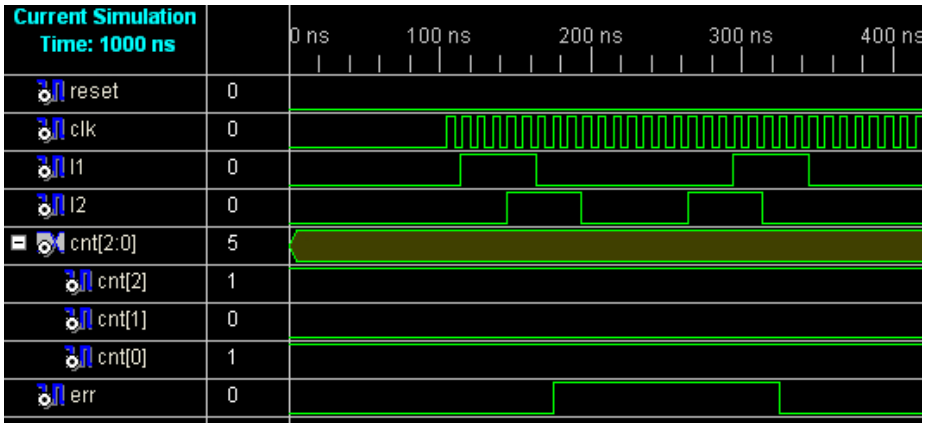

Abbildung 16: Simulation des leeren Parkhauses bei Herausfahren eines Fahrzeuges

Der Zweite erfordert schon etwas mehr Aufwand:

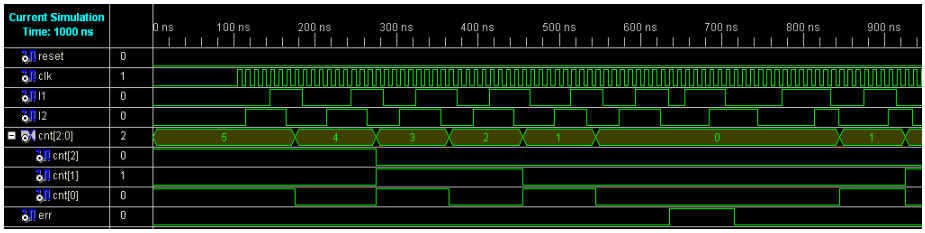

Abbildung 17: Simulation des vollen Parkhauses bei Hereinfahren eines Fahrzeuges

Nach dieser erfolgreichen Simulation geht es an die Synthetisierung:

7.2.6 Die Synthetisierung

Vor dem eigentlichen Synthetisierungsvorgang muss die Pinbelegung durch die „User Constraints“ festgelegt werden. Unter diesem Punkt kann der PACE-Editor durch einen Doppelklick auf `[Floorplan IO]` > `[Pre-Synthesis]` gestartet werden. Bei der Zuordnung der Pins sollte auf die korrekte Zuordnung von Clock (Pin 5) und Reset (Pin 39) geachtet werden. Sind die Pinbelegungen festgelegt und gespeichert, kann PACE beendet werden und zur eigentlichen Synthetisierung fortgeschritten werden. Der Doppelklick auf „Implement Design“ setzt diesen Vorgang in Bewegung.

Ist dieser Vorgang abgeschlossen, sollte der „Fitter Report“ unter dem Punkt „Fit“ einigen Aufschluss über die korrekte Synthetisierung geben. Dort lässt sich in der Tabelle „RESOURCES SUMMARY“ unter dem Punkt „Registers

Used“ erkennen, wieviel Bit an Registern insgesamt verwendet wurden. Unsere Steuerung benötigt laut diesem Report sieben Bit. Das Entspricht drei Bit für den Zähler, drei Bit für den Zustandszähler und ein Bit für den Error-State. Somit scheint dieser Teil der Synthetisierung problemlos funktioniert zu haben. Die Tabelle „PIN RESOURCES“ gibt Aufschluss über die korrekte Pin-Belegung. Dort sollte jeweils in der Zeile GSR/IO (Reset) und GCK/IO (Clock) unter Used die Zahl 1 stehen. Dies bedeutet, dass von beidem jeweils eines benötigt wird. Die nun gesammelten Fakten stimmen soweit mit den erwarteten Ergebnissen überein. Die Netzliste kann nun auf den CPLD gebrannt werden. Dazu sollte in dem Tool iMPACT der Programmier-Vorgang gestartet werden.

7.2.7 Der endgültige Chip

Der gebrannte CPLD sollte nun in den Versuchsaufbau integriert und an die Sensoren angeschlossen werden. In dem hier verwendeten Aufbau verlief dieser Test ohne Probleme und den auf den Messwerten basierenden Erwartungen entsprechend.

Tabellenverzeichnis

Abbildungsverzeichnis

Literatur

[1] Bosse, Dr. Stephan: *Anwendungsspezifische (programmierbare) Digitallogik und VHDL-Synthese*. BSS Lab, Bremen, 2. Auflage, 2007.

[2] Christian Siemers, Axel Sikora: *Taschenbuch Digitaltechnik*. Fachbuchverlag Leipzig, 1. Auflage, 2003.

[3] www.ulrichradig.de: *Ulrich Radig, mikrocontroller and more :: BaseKit Xilinx CPLDs*, 2008. [Online; Stand 30. März 2008].

[4] www.wikipedia.de: *Programmierbare logische Schaltung — Wikipedia, Die freie Enzyklopädie*, 2008. [Online; Stand 30. März 2008].

[5] www.mikrocontroller.net: *Joint Test Action Group*, 2008. [Online; Stand 30. März 2008].

[6] Xilinx: *JTAG/Parallel Download Cable*. Datenblatt, Xilinx, Inc., Juli 1996.

[7] Xilinx: *XC9536 ISP Demo Board*. Datenblatt, Xilinx, Inc., April 1997.

[8] Xilinx: *XC9536 In-System Programmable CPLD*. Datenblatt, Xilinx, Inc., Dezember 1998.

[9] Xilinx: *XC9500 In-System Programmable CPLD Family*. Datenblatt, Xilinx, Inc., September 1999.

A Quelldateien

A.1 Einfacher Zähler [/sources/counter.vhd]:

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.STD_LOGIC_ARITH.ALL;
4  use IEEE.STD_LOGIC_UNSIGNED.ALL;
5
6
7
8  entity count is
9      Port ( GCLK : in  STD_LOGIC;
10            DIR : in  STD_LOGIC;
11            OUTPUT : out  STD_LOGIC_VECTOR (2 downto 0));
12 end count;
13
14 architecture Behavioral of count is
15
16 signal counter : STD_LOGIC_VECTOR (2 downto 0)
17   := (others => '0');
18
19 begin
20
21    process (GCLK)
22    begin
23       if rising_edge(GCLK) then
24          if DIR = '1' then
25             counter <= counter + 1;
26          else
27             counter <= counter - 1;
28          end if;
29       end if;
30    end process;
31
32    OUTPUT <= not counter;
33
34 end Behavioral;
```

A.2 Erweiterter Zähler [/sources/counter_extended.vhd]:

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.STD_LOGIC_ARITH.ALL;
4  use IEEE.STD_LOGIC_UNSIGNED.ALL;
5
6
7
8  entity count is
9      Port ( GCLK : in  STD_LOGIC;
10            DIR : in  STD_LOGIC;
11            RESET : in STD_LOGIC;
12            OUTPUT : out  STD_LOGIC_VECTOR (2 downto 0));
13 end count;
14
15 architecture Behavioral of count is
16
17 signal counter : STD_LOGIC_VECTOR (2 downto 0)
18   := (others => '0');
19
20 begin
21
22    process (GCLK)
23    begin
24       if rising_edge(GCLK) then
25          counter <= (others => '1');
26       else
27          if RESET = '1' then
28             if DIR = '1' then
29                counter <= counter + 1;
30             else
31                counter <= counter - 1;
32             end if;
33          end if;
34       end if;
35    end process;
36
37    OUTPUT <= not counter;
38
39 end Behavioral;
```

A.3 Parkhaussteuerung [/sources/parkhaus.vhd]:

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.STD_LOGIC_ARITH.ALL;
4  use IEEE.STD_LOGIC_UNSIGNED.ALL;
5
6
7
8  entity parkhaus is
9     Port (   reset : in  STD_LOGIC;
10             clk   : in  STD_LOGIC;
11             L1 : in  STD_LOGIC;
12             L2 : in  STD_LOGIC;
13             cnt   : out  STD_LOGIC_VECTOR (2 downto 0);
14             err   : out  STD_LOGIC);
15 end parkhaus;
16
17
18 architecture Behavioral of parkhaus is
19    type states is (  def,   -- Default State
20                      Zm,    -- Z-?
21                      Zmm,   -- Z-
22                      Zp,    -- Z++?
23                      Zpp,   -- Z++
24                      wt);   -- wait
25
26    -- Speicherung des aktuellen Zustands
27    signal state : states;
28    -- Speicherung des folgenden Zustands
29    signal state_next : states;
30
31    -- Register f. Zählerstand
32    signal cntbuf : STD_LOGIC_VECTOR(2 downto 0) :="101";
33    -- Register f. Fehlerfall
34    signal errbuf : STD_LOGIC :='0';
35 begin
36
37 state_transitions: process(state,L1,L2)
38 begin
39    -- default state
40    state_next <= def;
41
42    case state is
43       when def    =>
44          -- default Zuweisung
45          state_next <= def;
46
47          if    L1='1' and L2='0' then
48
```

```
49             state_next <= Zp;
50
51          elsif L1='0' and L2='1' then
52
53             state_next <= Zm;
54          end if;
55
56       when Zm  =>
57          -- default Zuweisung
58          state_next <= Zm;
59
60          if    L1='0' and L2='0' then
61
62                state_next <= def;
63
64          elsif    L1='1' and L2='0' then
65
66                state_next <= Zmm;
67          end if;
68
69       when Zmm    =>
70          -- default Zuweisung
71          state_next <= wt;
72
73       when Zp     =>
74          -- default Zuweisung
75          state_next <= Zp;
76
77          if    L1='0' and L2='0' then
78
79             state_next <= def;
80
81          elsif    L1='0' and L2='1' then
82
83             state_next <= Zpp;
84          end if;
85
86       when Zpp =>
87          -- default Zuweisung
88          state_next <= wt;
89
90       when wt     =>
91          -- default Zuweisung
92          state_next <= wt;
93
94          if  L1='0' and L2='0' then
95
96             state_next <= def;
97          end if;
98
```

```
99    end case;
100
101 end process state_transitions;
102
103 state_logic: process(clk)
104 begin
105    if clk'event and clk='1' then
106    -- Hauptschleife
107       -- positiver Reset
108       if reset='1' then
109          state <= def;      -- erreichen des Defaultstates
110          cntbuf <= "101";   -- den Zähler auf fünf setzen
111          errbuf <= '0';     -- den Fehlerfall entfernen
112       else
113
114          -- state-Register-Transition
115          state <= state_next;
116
117          if state=Zpp then
118
119             if cntbuf="101" then
120                errbuf <= '1';
121             elsif cntbuf="000" and errbuf='1' then
122                errbuf <= '0';
123             else
124                cntbuf <= cntbuf + 1;
125             end if;
126
127          elsif state=Zmm then
128
129             if cntbuf="000" then
130                errbuf <= '1';
131             elsif cntbuf="101" and errbuf='1' then
132                errbuf <= '0';
133             else
134                cntbuf <= cntbuf - 1;
135             end if;
136
137          end if;
138       end if;
139    end if;
140
141 end process state_logic;
142
143
144    -- Simulation
145 -- cnt <= cntbuf;
146 -- err <= errbuf;
147
148    -- Synthese auf Test-Board (LEDs sind Low-Aktiv)
```

```
149     cnt <= not cntbuf;
150     err <= not errbuf;
151
152 end Behavioral;
```

B Abbildungen

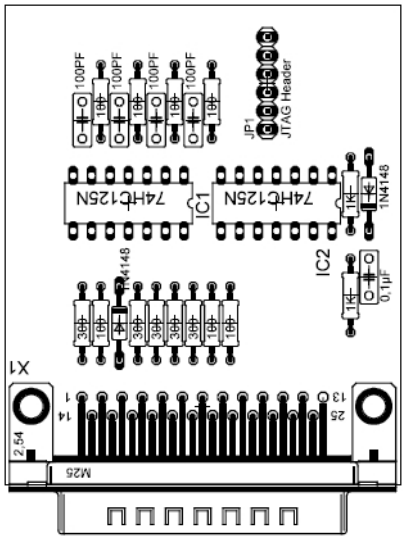

Abbildung 18: Die bestückte JTAG-Programmer Platine [3]

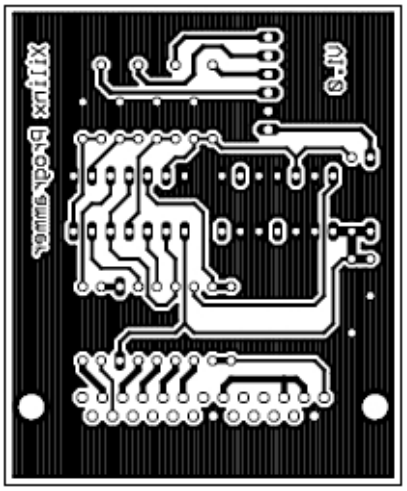

Abbildung 19: Die zu fräsende bzw. zu ätzende Unterseite der JTAG-Programmer Platine [3]

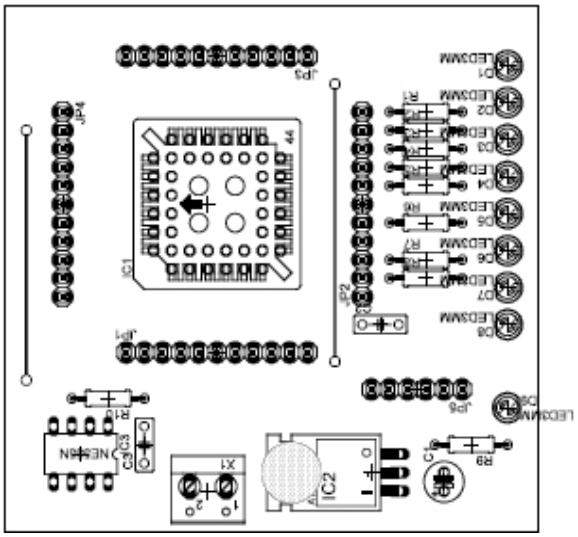

Abbildung 20: Die bestückte XC9536 Test- und Programmierplatine [3]

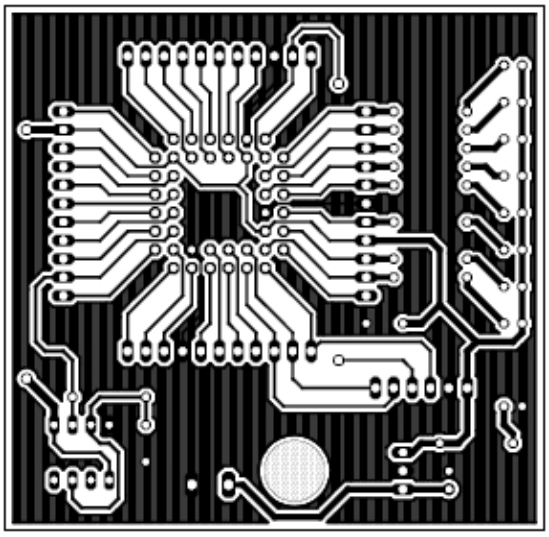

Abbildung 21: Die zu fräsende bzw. zu ätzende Unterseite der XC9536 Test- und Programmierplatine [3]