IMS
Institut für Mikroelektronische Systeme

Leibniz
Universität
Hannover

Masterlabor Mechatronik II
Grundlagen des HW-Entwurfs für FPGAs
Einführung und Grundlagen

Laborumdruck

Till Fiedler, Tim Oberschulte

Prof. Dr.-Ing. Holger Blume

Wintersemester 2022/23

Revision: 13. Oktober 2022

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation und Ziele

In diesen Versuchen soll verdeutlicht werden, wie Schaltungen unterschiedlicher Komplexität mittels einer Hardware-Beschreibungssprache auf rekonfigurierbare Logikbausteine komfortabel und flexibel implementiert werden können. Hierfür sollen die Teilnehmer mehrere aufeinander aufbauende Versuche bearbeiten und so einige Grundkenntnisse im Umgang mit der Hardware-Beschreibungssprache VHDL erwerben. Im Rahmen dieses Umdrucks wird deshalb neben der ersten Versuchsbeschreibung auch eine Einführung in VHDL gegeben. Weiterhin wird zum besseren Verständnis der Hintergründe des Labors und der einzelnen Versuche ebenfalls auf den grundlegenden Aufbau der verwendeten Hardware, den sogenannten Field Programmable Gate Arrays (FPGA), eingegangen.

1.2 Bedeutung rekonfigurierbarer Logik

Bei der Wahl einer geeigneten Implementierungsform zur Realisierung eines digitalen Systems bieten sich verschiedene Möglichkeiten an, die sich bezüglich ihrer Eigenschaften unterscheiden. Diese sind für die Kosten eines Systems entscheidend und spannen einen so genannten „mehrdimensionalen Entwurfsraum“ auf. In Abbildung 1 sind für verschiedene Implementierungsformen drei dieser Eigenschaften (Flexibilität, Rechenleistung, Verlustleistung) aufgetragen [1].

Dabei lässt sich erkennen, dass die programmierbaren General-Purpose Prozessoren (GPP), Digitalen Signalprozessoren (Digital signal processor, DSP) und Application-Specific-Instruction-Set-Prozessoren (ASIP) bezüglich ihrer Flexibilität, die hier als Reziprokwert der erforderlichen Redesign/Re-Programmier-Zeit für eine spezielle Aufgabe aufgetragen ist, zunächst die attraktivsten Lösungen darstellen. Sie sind im Allgemeinen in einer Hochsprache (C/C++) programmierbar. Das bedeutet, dass z.B. Systemänderungen aufgrund neuer Anforderungen schnell im Source-Code geändert werden können. Bezüglich der erreichbaren Rechenleistung

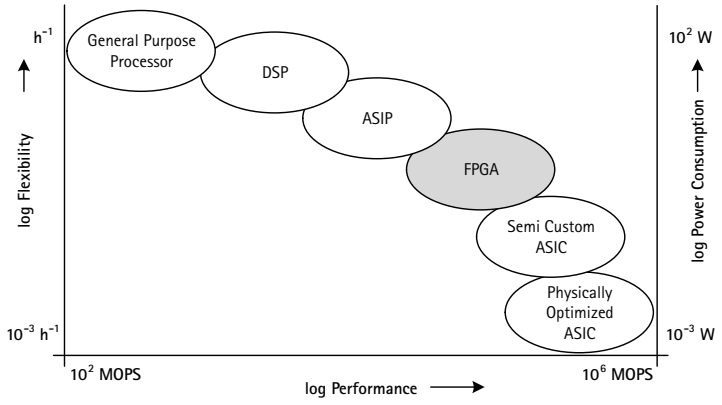

Abbildung 1: Ausschnitt aus dem Entwurfsraum digitaler Systeme nach [1].

und Verlustleistungsaufnahme, die insbesondere für mobile Systeme (Handys, Laptops, etc.) auf Grund daraus resultierender Akku-Laufzeiten von entscheidender Bedeutung ist, bieten die programmierbaren Implementierungsformen ungünstigere Eigenschaften verglichen mit festverdrahteten Implementierungsformen (Standardzellen-ASIC, physikalisch optimiertes ASIC). FPGAs befinden sich zwischen diesen Extrema des Entwurfsraums und stellen für viele Systemimplementierungen einen guten Kompromiss bezüglich der drei aufgeführten Kriterien dar. Einerseits haben sie ein hohes Maß an Rechenleistung bei moderater Verlustleistungsaufnahme, andererseits sind sie aufgrund ihrer Rekonfigurierbarkeit sehr flexibel.

Moderne Hardware-Plattformen sind häufig heterogen aufgebaut. Das bedeutet, dass mehrere der genannten Implementierungsformen in Form von Architekturblöcken, beispielsweise auf einem Chip als System-on-Chip (SoC), miteinander kombiniert werden. Auf diese Weise können SoCs an die komplexen Anforderungen von modernen digitalen Systemen, bezüglich der physikalischen Kosten, der Flexibilität und erforderlichen Performance angepasst werden.

1.3 Field Programmable Gate Arrays

Unter einem FPGA (Field Programmable Gate Array) versteht man einen speicherkonfigurierbaren Logikbaustein, der vom Anwender so programmiert werden kann, dass die gewünschte Logikfunktion wie bei einem fest verdrahteten Logikbaustein zur Verfügung steht. Mit Hilfe eines FPGAs lassen sich daher sehr komplexe Funktionen mit hoher Logiktiefe implementieren, wie z.B. arithmetische Grundschaltungen und komplexe Schaltwerke. FPGAs werden vor allem in der Prototypenentwicklung und in rekonfigurierbaren Systemen wie z.B. Hardware-Emulatoren eingesetzt. Da inzwischen preisgünstige FPGA-Bausteine erhältlich sind, werden FPGAs auch zunehmend in Serienfertigungen verwendet.

Prinzipiell bestehen FPGAs aus drei wesentlichen, konfigurierbaren Grundelementen. Es handelt sich dabei um Logikblöcke, Eingangs- und Ausgangs-Blöcke (I/O-Blöcke) und programmierbare Verbindungen. Die Grundelemente sind in einer regulären Matrix-Struktur auf dem Substrat angeordnet (Abbildung 2). Diese feinstufige Architekturform wird als Channeled-Array-Architektur bezeichnet.

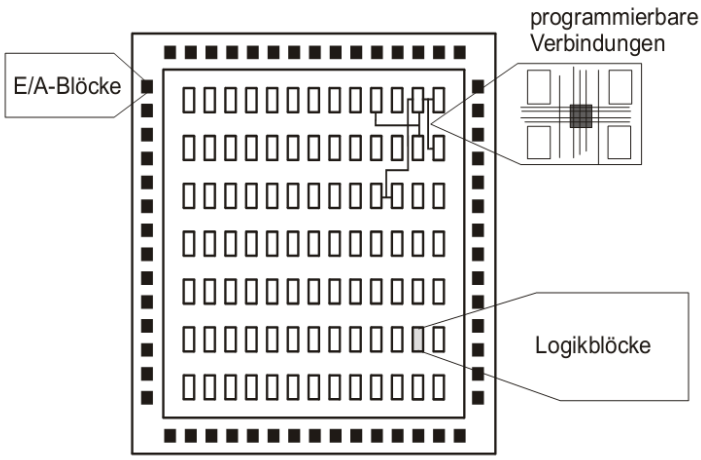

Abbildung 2: Grundstruktur eines FPGA.

Ein Logikblock enthält als wesentliche Elemente „Look-Up-Tabellen“ (LUTs) zur Realisierung einfacher Schaltnetze und Flip-Flops. Die LUTs sind aus SRAM-Zellen aufgebaut, deren Inhalt mittels eines Multiplexers ausgelesen wird. Die Anzahl k der Multiplexereingänge legt die Größe der LUTs fest. Wie in Abbildung 3 dargestellt, enthält eine k-Input LUT 2^k SRAM-Zellen. Somit lässt sich in einer k-Input LUT jede der möglichen booleschen Funktion aus k Eingangsvariablen implementieren. Im Gegensatz zu einer Realisierung mit UND- bzw. ODER-Gattern ist die benötigte Siliziumfläche damit unabhängig von der booleschen Funktion. Je nach FPGA-Hersteller variiert der Wert von k zwischen 3 und 16. Üblicherweise werden in SRAM-basierten FPGAs 4-Input LUTs verwendet.

Zur Verschaltung von Logik- und I/O-Blöcken steht dem Anwender ein Netz aus Verbindungselementen (Abbildung 4) zur Verfügung, die in horizontalen und vertikalen Kanälen zusammengefasst sind. Zur Verknüpfung von Leitungssegmenten werden programmierbare Schalter eingesetzt. Es handelt sich dabei um sogenannte Pass-Transistoren. Die Konfiguration der Pass-Transistoren erfolgt mit Hilfe einer SRAM-Zelle, die mit dem

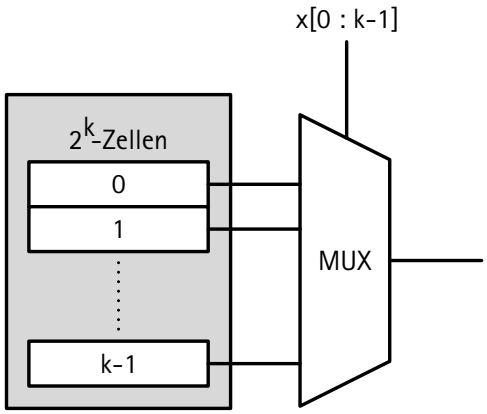

Abbildung 3: Look-Up-Table.

Gate verbunden ist. Enthält die Zelle eine logische „1“ ist der Transistor durchgeschaltet und die Leitungssegmente sind verbunden. Im Fall einer „0“ besteht keine Verbindung. In den Kreuzungspunkten der horizontalen und vertikalen Verdrahtungskanäle befinden sich konfigurierbare Verbindungsmatrizen. Sie ermöglichen den Wechsel eines Signals von einer horizontalen auf eine vertikale Richtung und umgekehrt.

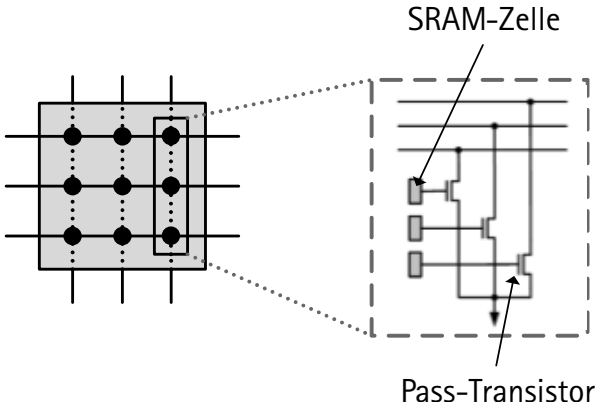

Abbildung 4: Programmierbare Verbindungen durch Pass-Transistoren. Die Programmierung findet durch die angeschlossene SRAM-Zelle statt.

Die Verwendung konfigurierbarer Elemente erlaubt eine große Flexibilität, stellt jedoch hohe Anforderungen an die Aufteilung der kompletten Schaltung in Teilfunktionen und deren anschließende Verdrahtung. Besonders bei langen Verbindungen auf dem Chip können sich aufgrund hoher Leitungskapazitäten und großer Widerstände durch die Pass-Transistoren hohe Verzögerungszeiten ergeben.

Neben der Logikzelle existieren typischerweise eine Vielzahl weiterer Grundelemente (z.B. dedizierte Multiplizierer-Bausteine) auf einem FPGA. An dieser Stelle sollen jedoch lediglich die Grundstrukturen von FPGAs erläutert werden. Es sei jedoch darauf hingewiesen, dass viele weitere Aspekte bei der Arbeit mit FPGAs relevant sein können. Im Kapitel 4 dieses Umdruckes ist für den interessierten Leser ein Auszug tiefergehender Literatur bereitgestellt.

1.4 Entwurfsablauf

Der Entwurfsprozess von digitalen Schaltungen für ein FPGA lässt sich in mehrere Schritte unterteilen. Die einzelnen Teilschritte des Entwurfsablaufs sind in Abbildung 5 neben den dazugehörigen Programmen/Werkzeugen für das im Rahmen dieses Labors verwendete FPGA dargestellt.

Im Allgemeinen beginnt der Entwurfsprozess bei einem so genannten „Top-Down“ Ansatz mit der Formulierung des Problems. Für diese Problemstellung wird ein Algorithmus oder Schaltung hergeleitet, der die gestellte Aufgabe löst. Die Beschreibung der Schaltung kann z.B. mit Hilfe einer Hardware-Beschreibungssprache

(Hardware-Description-Language, HDL) erfolgen. HDLs sind u.a. Verilog und VHDL (Very High Speed Integrated Circuit HDL). Alternativ kann die Beschreibung auch grafisch in Form eines Schaltplans beschrieben werden. Für die grafischen Beschreibungen gibt es kein einheitliches Format, da es sich in der Regel um eine Visualisierung von HDL-Code handelt. Grafische oder textuelle Beschreibung von Hardware kann sowohl für eine FPGA-Programmierung als auch für einen ASIC Standard-Zellen-Entwurf genutzt werden. In diesem Labor wird ausschließlich die Hardware-Beschreibungssprache VHDL verwendet.

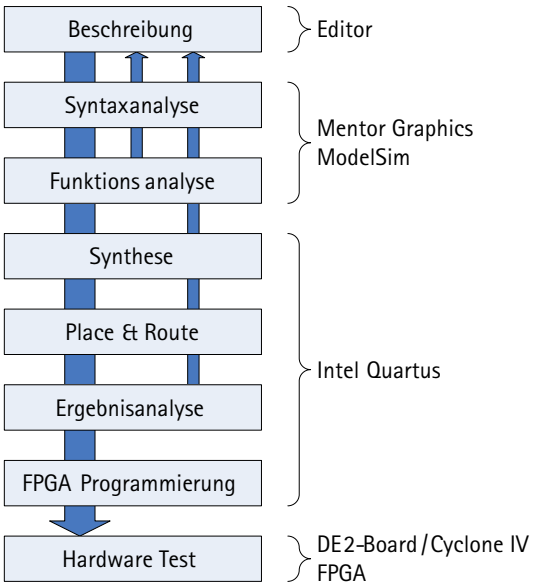

Abbildung 5: Entwurfsablauf zur Abbildung einer Schaltung auf ein FPGA.

Anschließend an die Beschreibung wird der VHDL-Code einer **Syntaxanalyse** unterzogen. Wenn die Beschreibung syntaktisch korrekt ist, wird vom Programmierenden eine Verifikation des programmierten Algorithmus mit Hilfe einer funktionalen Simulation durchgeführt (**Funktionsanalyse**). Dabei wird der reine VHDL-Code simuliert, ohne Rücksicht auf die zugrundeliegende Hardware. Timing-Informationen werden noch nicht betrachtet.

Nachdem die Funktionalität verifiziert wurde, erfolgt bei einer **Synthese** die Extraktion einer gatterbasierten Netzliste, welche bereits Informationen der zu Grunde liegenden Technologie berücksichtigt. In dieser Netzliste wird angegeben, wie viele Logikzellen benötigt, wie diese programmiert, und untereinander verbunden werden. Es ist jedoch noch nicht festgelegt, wo sich die Logikzellen auf dem FPGA befinden und welche Routing-Ressourcen dafür genutzt werden.

Im **Place & Route**-Schritt wird die extrahierte Netzliste auf den FPGA abgebildet. Dabei werden zum einen die erforderlichen Logikelemente auf dem FPGA platziert, zum anderen werden die Logikelemente über die zur Verfügung stehenden Verbindungsstrukturen (Routing-Ressourcen) miteinander verbunden.

Bei der **Ergebnisanalyse** werden Parameter der vorherigen Schritte überprüft und mit eventuell angegebenen Zielvorgaben verglichen. Ein Beispiel dafür ist die Timing-Analyse. Dabei wird geprüft, welches die maximale Taktfrequenz der Schaltung ist und ob eine Vorgabe, ein sog. Timing-Constraint, eingehalten werden kann. Können Vorgaben grundsätzlich nicht eingehalten werden, gibt es die Möglichkeiten die Zielvorgaben abzuändern oder die Beschreibung der digitalen Schaltung zu optimieren. Wird die Beschreibung geändert, müssen alle vorherigen Schritte des Entwurfsablaufs erneut durchgeführt werden.

Entsprechen die Ergebnisse den Zielvorgaben, kann mit der entsprechenden Hardware und Software die Programmierung des FPGAs (**FPGA Programmierung**) vorgenommen werden. Hier kann nun die korrekte Funktion der digitalen Schaltung auf dem FPGA durch einfach Hör- oder visuelle Tests, sowie mit dafür vorgesehenen Messgeräten (z.B. Logik-Analysator) verifiziert werden (**Hardware Test**).

Da die durchzuführenden Versuche im Rahmen dieses Labors bezüglich ihrer Komplexität überschaubar sind, wird in diesem Labor auf eine vorige Simulation der auf dem FPGA zu implementierenden Schaltungen verzichtet. Statt der Nutzung des in der Abbildung 5 ebenfalls dargestellten Programmes Mentor Graphics ModelSim sollen die Teilnehmer die in VHDL beschriebene Schaltung über die Entwicklungsumgebung Intel Quartus Prime direkt auf das FPGA programmieren und anhand seines Verhaltens verifizieren und bewerten.

1.5 Das DE2-Board

Alle Versuche in diesem Labor werden auf dem *Altera DE2 Development and Education Board* (DE2-Board, [10]) durchgeführt. Die Hauptkomponente ist ein *Cyclone IV* FPGA der Firma Altera (heute Intel). Die FPGAs dieser Familie sind in einer 60 nm Technologie gefertigt und vereinen einen kostengünstigen Preis mit hoher Performance. Die folgende Abbildung zeigt das Board von oben.

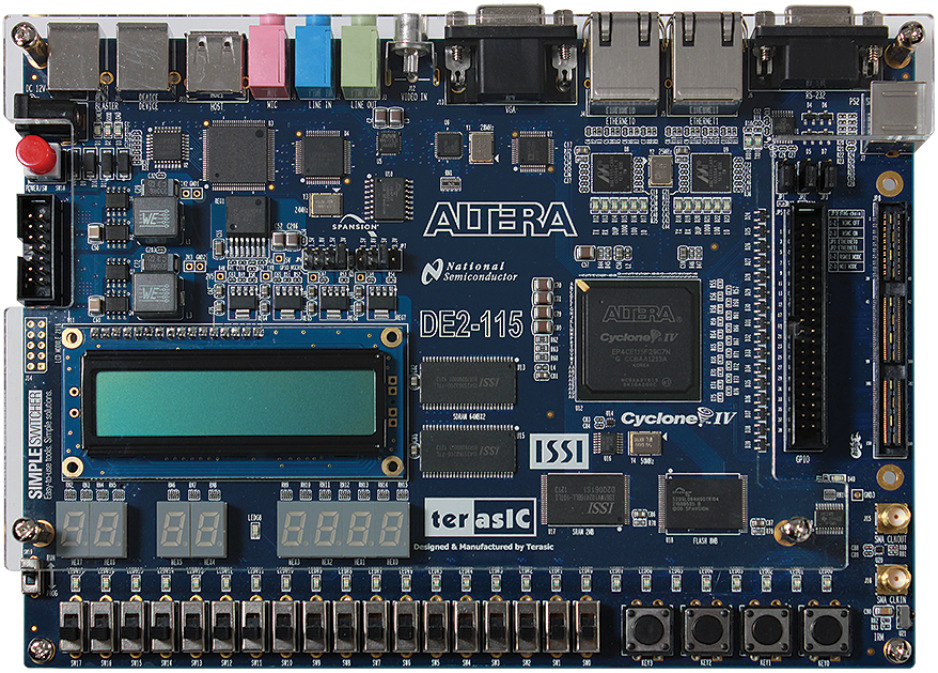

Abbildung 6: DE2-Board.

Zusätzlich ist das DE2-Board mit einer Vielzahl an Peripherie-Komponenten ausgestattet. Dazu gehören diverse Speicher (SRAM, SDRAM, FLASH), ein LC-Display, 7-Segment-Anzeigen, diverse Ein- und Ausgabe-Schnittstellen (Audio, Video, Netzwerk, serielle Kommunikation), Pfostenwannen zum Anschluss externer Komponenten, diverse Knöpfe, Schalter und LEDs. Dies ermöglich es, das DE2-Board für unterschiedlichste Anwendungen einfach und komfortabel einzusetzen.

Kapitel 2

Einführung in VHDL

Für alle Versuche dieses Labors wird die Hardware-Beschreibungssprache VHDL verwendet. Dieses Kapitel gibt eine kurze Einführung in die Grundlagen dieser Sprache. Dabei wird neben dem Aufbau einer VHDL-Datei auch auf einige Besonderheiten bei bestimmten Implementierungen, wie beispielsweise bei Clock-Signalen oder endlichen Zustandsautomaten, eingegangen. Details können auch unter [2] nachgelesen werden.

2.1 Struktur einer VHDL-Komponente

Der grundlegende Aufbau einer Komponente in VHDL besteht aus Einbinden von Bibliotheken, einer „*Entity*“ und einer „*Architecture*“. Bibliotheken können Datentypen, Funktionen und Komponenten enthalten, die nach Einbindung einer Bibliothek verwendet werden können. So müssen einfach logische und arithmetische Funktionen wie eine Addition nicht immer neu definiert werden. Die folgende Abbildung zeigt die Einbindung von Bibliotheken in eine VHDL Datei. Dies geschieht üblicherweise am Beginn einer Datei.

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
```

Abbildung 7: Einbinden von Bibliotheken in eine VHDL-Datei. Mit diesen drei Zeilen beginnt fast wie jede VHDL-Datei.

Verwenden Sie grundsätzlich für alle Versuche nur die beiden angegebenen Bibliotheken, sofern nichts anderes im Umdruck des jeweiligen Versuches erwähnt wird.

Die *Entity* beschreibt die Ein- und Ausgangsports einer Komponente, die Schnittstelle zu anderen Komponenten. Dabei wird jeder Port mit einem eindeutigen Namen, der Richtung und dem Datentyp angegeben. Für die Portrichtung stehen die Schlüsselwörter `in`, `out` und `inout` zur Verfügung. Ein Eingang wird mit `in` und ein Ausgang mit `out` deklariert. Handelt es sich um einen bi-direktionalen Port, so kann dieses mit `inout` deklariert werden. Auf die Datentypen wird im Einzelnen in Abschnitt 2.3 eingegangen. Grundlegend muss ein Port mindestens ein Bit breit sein. Mehr Bit breite Ports können dabei über sogenannte Vektoren definiert werden. Abbildung 8 zeigt die Struktur einer Entity in VHDL.

```
1  entity NAME_DER_ENTITY is
2  port(
3    port_name1 : in  std_ulogic;
4    port_name2 : in  std_ulogic_vector(BREITE_PORT2 - 1 downto 0);
5
6    port_name3 : out std_ulogic;
7    port_name4 : out std_ulogic_vector(BREITE_PORT4 - 1 downto 0) -- hier darf kein ; hin
8  );
9  end NAME_DER_ENTITY;
```

Abbildung 8: Struktur einer Entity in VHDL. Hier werden die Ein- und Ausgänge mit ihren dazugehörigen Datentypen definiert. Hinter dem letzten Signal darf kein Semikolon stehen.

Der Name der *Entity* ist der Bezeichner für die gesamte Komponente. Grundsätzlich wird empfohlen *Entity*-Namen dem Namen der Datei anzupassen. Unter Umständen können Programme Probleme damit haben, wenn der Name der *Entity* nicht dem Dateinamen entspricht. Heißt die Datei z.B. „counter.vhdl“, so sollte die *Entity* ebenfalls „counter“ heißen. Beachten Sie, dass nach der Beschreibung des letzten Signals kein Semikolon folgen darf.

Während die *Entity* nur die Ein- und Ausgangsports einer „Blackbox“ beschreibt, wird in der *Architecture* das Verhalten innerhalb der Komponente beschrieben. Dabei können Signale definiert und miteinander verschaltet, verknüpft oder verrechnet werden. Signale in VHDL sind die hauptsächlichen Objekte die ein digitales System beschreiben und gleichbedeutend mit „Drähten“. Sie repräsentieren Verbindungen zwischen den Elementen einer digitalen Schaltung. Je nachdem wie Signale verschaltet werden, übersetzt ein Synthesewerkzeug (z.B. Quartus) die Verschaltungen in Multiplexer, Flip-Flops oder logische Gatter.

Das `begin` in Zeile 8 in Abbildung 9 trennt den sogenannten Deklarationsabschnitt von dem Beschreibungsabschnitt einer *Architecture*.

Im Deklarationsabschnitt werden ausschließlich Signale und z.B. andere Komponenten deklariert. Im Beschreibungsabschnitt können diese Signale dann mit den Ein- und Ausgangsports der *Entity* und untereinander verschaltet werden.

```
1  architecture NAME_DER_ARCHITEKTUR of NAME_DER_ENTITY is
2  -- Deklarationsabschnitt
3
4   -- Signaldeklarationen
5   signal signal1 : std_ulogic;
6   signal signal2 : std_ulogic_vector(BREITE_SIGNAL2 - 1 downto 0);
7
8  begin
9  -- Beschreibungsabschnitt
10 ...
11 end NAME_DER_ARCHITEKTUR;
```

Abbildung 9: Struktur einer Architecture in VHDL. Sie besteht aus einem Deklarationsabschnitt, in dem Signale definiert werden können, und einem Beschreibungsabschnitt, welcher die Signal und die Ein- und Ausgaben der Entity verschaltet.

Eine *Architecture* ist immer genau einer *Entity* zugeordnet. Eine *Entity* kann unter Umständen aber mehrere *Architectures* besitzen. Daher muss der Name der *Architecture* für eine Entity eindeutig gewählt werden. Eine Verknüpfung mit dem Dateinamen ist hier nicht notwendig.

Grundsätzlich kann eine VHDL Datei mehrere *Entities* oder *Architectures* enthalten. Diese sind dann völlig voneinander unabhängige Komponenten. Aus Gründen der Übersichtlichkeit wird allerdings empfohlen in einer Datei nur eine Komponente zu beschreiben.

2.2 Hierarchische Designs

VHDL bietet die Möglichkeit das eigene Design modular aufzubauen und somit ein Problem in mehrere kleine Probleme zu zerlegen. Dieses Prinzip nennt sich „*Teile und Herrsche*“ (Divide and Conquer). So können VHDL-Komponenten die selber erstellt oder mit Bibliotheken eingebunden wurden mehrfach wiederverwendet werden.

Um eine Komponente in die eigene *Architecture* einzubinden, muss zunächst im Deklarationsabschnitt der Architecture die Komponente mit allen Ein- und Ausgangspots deklariert werden. Ein Beispiel dafür ist in Abbildung 10 gegeben.

```
1  -- Komponentendeklaration
2  component NAME_DER_ENTITY_AUS_DER_KOMPONENTE
3  port(
4    port1 : in  std_ulogic;
5    port2 : out std_ulogic_vector(BREITE_PORT2 - 1 downto 0) -- hier darf kein ; hin
6  );
7  end component;
```

Abbildung 10: Deklaration einer Komponente im Deklarationsabschnitt einer *Architecture*.

Um die deklarierte Komponente nun verwenden zu können, muss diese instanziiert und angeschlossen werden. Dabei sind beliebig viele Instanzen einer Komponente möglich, der Name einer Instanz muss dabei allerdings eindeutig sein. Eine Instanziierung erfolgt grundsätzlich im Beschreibungsabschnitt einer *Architecture*.

```
1  NAME_INSTANZ: NAME_DER_ENTITY_AUS_DER_KOMPONENTE
2  port map(
3    port1 => signal1,
4    port2 => signal2  -- hier darf kein , hin
5  );
```

Abbildung 11: Instanziierung einer Komponente. Dabei werden Signale der aktuellen *Architecure* mit den Ein- und Ausgängen der Komponente verbunden.

Sowohl bei der Komponentenbeschreibung als auch bei der Instanziierung darf, wie auch bei den Ports einer Entity, nach dem letztem Port kein Semikolon bzw. Komma folgen.

Die Komponente auf der höchsten Ebene, die zwar andere Komponenten einbindet, selber aber nicht eingebunden wird, bezeichnet man als *Top-Level*. Die Ein- und Ausgangsports der *Top-Level*-Komponente werden z.B. bei einer Emulation auf dem DE2-Board mit den physikalischen Pins des FPGAs verbunden.

2.3 Datentypen

Die Beschreibungssprache VHDL ist sehr stark typisiert. Der Typ eines Signals gibt an, wie ein Signal bei einer Berechnung interpretiert werden soll z.B. als vorzeichenlose oder vorzeichenbehaftete Zahl.

In diesem Abschnitt sollen nachfolgend nur die Datentypen betrachtet werden, die für das Labor von Bedeutung sind. Andere Datentypen sollen nicht verwendet werden, sofern dies im Umdruck nicht anderweitig beschrieben ist:

`std_ulogic` Beschreibt ein 1 Bit breites Signal. Beispiele hierfür sind das Clock- oder Reset-Signal.

`std_ulogic_vector` Im Gegensatz zum `std_ulogic`-Datentyp wird mit dem Vektor ein mehr Bit breites Signal deklariert. Die Bitbreite und Bitordnung wird bei der Deklaration dahinter in Klammern angegeben. Für ein 8-Bit breites Signal mit der Big-Endian-Bitordnung sieht die Deklaration wie folgt aus:

```
signal SIGNAL_NAME_1 : std_ulogic_vector(7 downto 0);
```

Das Schlüsselwort `downto` bestimmt dabei die Bitordnung. In diesem Fall steht das höchstwertigste Bit (Most Significat Bit, MSB) an erster Stelle. Um die Ordnung umzudrehen kann das Schlüsselwort `to` verwendet werden, entsprechend dazu müssen dann auch die Zahlen vertauscht werden.

Um einem `std_ulogic_vector` z.B. Nullen zuzuweisen können die einzelnen Bits bei einer Zuweisung angegeben werden:

```
SIGNAL_NAME_1 <= "00000000";
```

Alternativ kann diese Zuweisung auch durch einen sogenannten `other`-Befehl ersetzt werden, welche automatisch die gesamte Breite des Vektors abdeckt:

```
SIGNAL_NAME_1 <= (others => '0');
```

`unsigned` Um Vektoren als vorzeichenlose Zahl zu interpretieren kann dieser Datentyp verwendet werden. Um ein als `std_ulogic_vector` deklariertes Signal als vorzeichenlose Zahl zu interpretieren, muss das Schlüsselwort entsprechend mit angegeben werden:

```
... <= unsigned(SIGNAL_NAME_1);
```

Generell können alle Signale auf diese Art und Weise uminterpretiert werden („Type-Casting“). Dazu muss der Datentypen vor das anders zu interpretierende Signal, welches in runden Klammern steht, einfügt werden.

Muss ein Signal häufig als vorzeichenlose Zahl interpretiert werden, so kann diese auch direkt als solches deklariert werden:

```
signal SIGNAL_NAME_2 : unsigned(7 downto 0);
```

Signale vom Typ `unsigned` können mit Konstanten oder anderen Signalen verrechnet oder verglichen werden. Welche Arten von Vergleichen und Rechenoperationen möglich sind, wird im Abschnitt 2.5 erläutert.

Um eine Konstante direkt in diesen Datentyp zu konvertieren kann die Funktion `to_unsigned`(Konstante, Bitbreite) verwendet werden. Beispiel:

```
SIGNAL_NAME_2 <= to_unsigned(13, 8);
```

`signed` Dieser Datentyp ist ähnlich zum Datentyp `unsigned`. Signale können genauso uminterpretiert oder deklariert werden. Der Unterschied besteht darin, dass das MSB als Vorzeichen interpretiert wird. Negative Zahlen werden in VHDL grundsätzlich als 2er-Komplement dargestellt.

`integer` Dieser Datentyp stellt Ganzzahlen dar. Allerdings werden bei der Deklaration keine konkreten Bitbreiten, sondern Zahlenbereiche angegeben. Die Interpretation und das Ergebnis in Hardware der angegebenen Zahlenbereiche ist sehr vom Synthesewerkzeug abhängig, wodurch keine Signale mit diesem Datentyp deklariert werden sollten. Benutzt werden können Integer um einzelne Bit von Vektoren dynamisch auszuwählen. In diesem Fall muss eine vorzeichenlose Zahl in einen Integer umgewandelt werden:

```
... <= SIGNAL_NAME1(to_integer(SIGNAL_NAME_2));
```

`natural` Ähnlich wie `interger`, allerdings stellt `natural` nur positive Ganzzahlen dar.

2.4 Prozesse

Prozesse sind ein grundlegendes Element zur Beschreibung von Verhalten der Hardware und dienen zur Modellierung algorithmischer Vorgänge. Eine *Architecture* kann beliebig viele Prozesse beinhalten. Wichtig dabei ist aber zu beachten, dass Prozesse nebenläufig sind, also immer parallel abgearbeitet werden. Die nachfolgende Abbildung zeigt die grundlegende Struktur eines Prozesses:

Das Schlüsselwort für einen Prozess in VHDL ist `process`. Der Name vor dem Prozess ist rein optional und kann weggelassen werden. Wird kein Name verwendet entfällt auch der Doppelpunkt. Die sogenannte „*Sensitivity List*“ soll grundsätzlich alle Signale enthalten, die innerhalb des Prozesses gelesen werden. Ändert sich der

```vhdl
PROZESS_NAME : process(SENSITIVITY_LIST)
begin
-- Anweisungen
...
end process;
```

Abbildung 12: Struktur eines Prozesses in VHDL.

Wert von einem Signal in der *Sensitivity List* während einer Simulation, so wird der Prozess aktiviert und neu durchlaufen. Fehlen Signale in der *Sensitivity List* so kann es zu Fehlverhalten der beschriebenen Hardware bei der Simulation kommen. Für die Emulation hingegen spielt es keine Rolle, welche Signale in einer *Sensitivity List* aufgeführt sind.

Wichtig: Die Anweisungen innerhalb eines Prozesses werden zwar sequentiell durchlaufen, es ist aber für jedes Signal nur die jeweils letzte Wertzusweisung gültig.

In den beiden nachfolgenden Abschnitten werden die beiden grundlegenden Verwendungsarten (Kombinatorisch und Sequentiell) von Prozessen erläutert.

2.4.1 Kombinatorisch

Kombinatorische Prozesse enthalten grundsätzlich nur asynchrone Zuweisungen die unabhängig von einem Taktsignal sind. Somit werden in kombinatorischen Prozessen nur logische oder arithmetische Operationen, sowie De-/Multiplexer beschrieben.

```vhdl
...
  signal n_reg, n_next : unsigned(SIGNALBREITE -1 downto 0);
  signal a, b, c       : std_ulogic;
...
begin
...
  comb : process(n_reg, b) -- in Klammern: sensitivity list
  begin
    -- asynchrone Zuweisung
    n_next <= n_reg + to_unsigned(1, SIGNALBREITE);

    a <= '0';
    if b = '1' then
        a <= c;
    end if;
  end process;
...
```

Abbildung 13: Beispiel für einen kombinatorischen Prozess.

Sobald sich das Signal `n_reg` ändert, wird dem Signal `n_next` das um Eins inkrementierte Signal `n_reg` zugewiesen. Außerdem wird dem Signal `a` abhängig vom Signal `b` '0' oder der Wert des Signals `c` zugewiesen.

2.4.2 Sequentiell

Sequentielle Prozesse dienen der Beschreibung von Flip-Flops. Daher ist in ihnen eine Wertänderung immer nur zu einer Taktflanke möglich. Ob die steigende oder fallende Flanke gewählt wird, kann frei definiert werden.

Im Rahmen dieses Labors sollen Wertänderungen jedoch grundsätzlich auf steigenden Taktflanken stattfinden.

```
1  regs : process(clock, reset)  -- in Klammern: sensitivity list
2  begin
3    if reset = '1' then -- asynchroner high-aktiver Reset
4      n_reg <= (others => '0');
5    elsif rising_edge(clock) then -- steigende Taktflanke
6      if enable = '1' then
7        -- sequentielle Anweisung
8        n_reg <= n_next;
9      end if;
10   end if;
11 end process;
```

Abbildung 14: Beispiel für einen sequentiellen Prozess.

Ist im Falle von Abbildung 14 die `reset`-Bedingung gültig, so wird dieser Pfad genommen und das Signal `n_reg` auf den Wert Null zurückgesetzt. Da `n_reg` sonst nur ein Wert bei einer steigenden Taktflanke zugewiesen wird, wird aus diesem Signal ein Register mit der in Zeile 2 von Abbildung 13 deklarierten Bitbreite erstellt.

Die Abfrage der steigenden Taktflanke ist in Zeile 5 von Abbildung 14 zu finden. Die Anweisung `rising_edge(clock)` spezifiziert dabei Änderung des Signals von dem Wert '0' auf '1'.

Das Signal `enable` braucht in diesem Fall generell nicht in die *Sensitivity List* eingefügt werden, da bei einem Flip-Flop eine Wertänderung stets nur zu einer steigenden Taktflanke erfolgen kann.

Die in Abbildung 14 angegebene `reset`-Bedingung soll für alle Versuche des Labors so verwendet werden.

2.4.3 Clock-Management

Für eine Schaltung auf einem FPGA wird in der Regel ein globaler Takt durch eine externe Beschaltung, z.B. einen Schwingquarz, erzeugt. Dieser wird an bestimmte Clock-Eingänge des FPGAs angeschlossen, die eine besonders gute, d.h. schnelle und gleichmäßige Verteilung dieses Taktsignals über den gesamten FPGA gewährleisten.

Für gewisse Schaltungen kann es erforderlich sein, weitere Taktsignale mit geringerer Taktfrequenz auf dem FPGA zur Verfügung zu stellen. Diese so genannten Sub-Takte werden vom Systemtakt abgeleitet, sind jedoch durch die dafür nötige Schaltlogik und die schlechtere Verdrahtung über nicht spezialisierte Taktleitungen nicht mehr synchron zum Systemtakt. Diese Verschiebung gegenüber dem Systemtakt heißt Clock-Skew und kann zu unerwünschten Schaltzuständen in der Schaltung führen. Dieses Kapitel beschreibt, wie Sub-Takte erzeugt und die daraus entstehenden Timing-Probleme gelöst werden können.

Reduktion der Taktrate

Die einfachste Methode, einen reduzierten Takt zu generieren ist es, einen Zähler zu implementieren und das gewünschte Bit dieses Zählers für die Erzeugung des neuen Taktes zu verwenden:

Das Signal `clock_vector_reg` ist hierbei vom Typ `std_ulogic_vector(n downto 0)`. Die Größe dieses Signals beeinflusst den maximalen Faktor (Potenzen von Zwei), mit dem der Takt reduziert wird. So hat zum Beispiel `clock_vector_reg[1]` die 4-fache und `clock_vector_reg[2]` die 8-fache Periodendauer von `clock` (Abbildung 16).

Die hierbei entstehenden Zeitunterschiede der Signalflanken, hervorgerufen durch Signallaufzeiten, können zu Problemen führen, wenn beide Taktsysteme in einem FPGA verwendet werden. Auf diese so genannten Timing-Violations wird im nächsten Abschnitt näher eingegangen.

Clock-Skew und Timing-Violations

Um die korrekte logische Funktion einer Schaltung zu gewährleisten, müssen die Daten für bestimmte Zeiten vor und nach der steigenden Taktflanke an den Eingängen der nachgeschalteten Register anliegen. Diese

```vhdl
...
regs : process (clock, reset)
begin
  if reset = '1' then
    clock_vector_reg <= (others => '0');
  elsif rising_edge(clock) then
    clock_vector_reg <= clock_vector_next;
  end if;
end process;

comb : process (clock_vector_reg)
begin
  clock_vector_next <= std_ulogic_vector(unsigned(clock_vector_reg) + 1);
end process;
...
```

Abbildung 15: Beispielprozesse zur Beschreibung eines Zählers zur Taktreduktion. Der Wert von `clock_vector_reg` wird jeden Takt um eins erhöht.

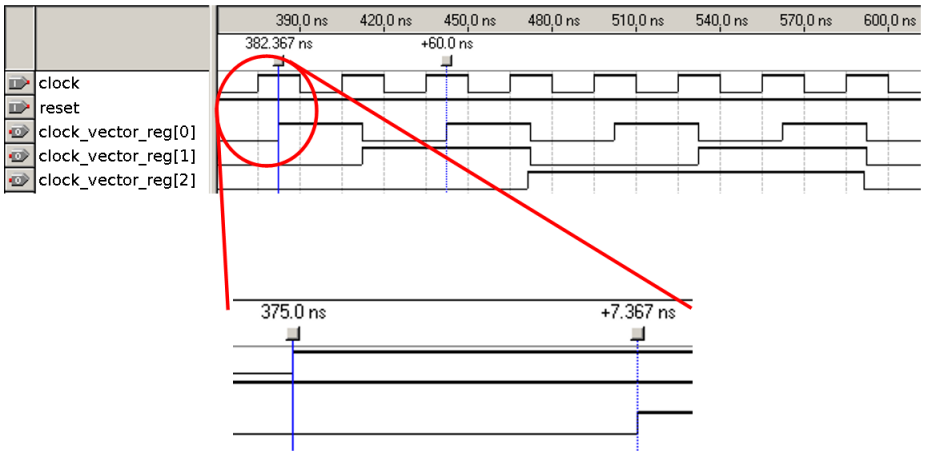

Abbildung 16: Verzögerung durch Signallaufzeiten.

Zeitbedingungen werden *Setup* und *Hold* genannt. Die *Setup*-Zeit (t_{setup}) ist die Länge des Zeitintervalls vor der aktiven Taktflanke, in dem sich das Eingangsdatum nicht mehr ändern darf. Die *Hold*-Zeit (t_{hold}) definiert den Zeitbereich nach einer steigenden Taktflanke, in welcher das anliegende Eingangsdatum noch konstant zu halten ist. Eine *Violation* liegt dann vor, wenn sich die Daten innerhalb der *Setup*- oder *Hold*-Zeiten ändern. Dies kann leicht auftreten, wenn der Takt durch eingefügte Logikschaltungen gegenüber den Daten verzögert wird. Des Weiteren definiert die sogenannte *Propagation Delay Time* (t_{pd}) die Verzögerungszeit eines Registers, die zum Zeitpunkt steigender Taktflanke vergeht, bis das Eingangsdatum am Ausgang anliegt.

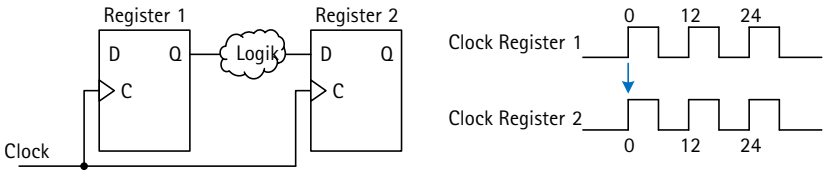

Abbildung 17: Reguläre Taktumgebung, $t_{\text{hold,required}} = 0\,\text{ns}$. Zwischen den beiden Takteingängen existiert keine Verzögerung.

Abbildung 17 zeigt eine logische Schaltung mit einem Takt. *Hold-Violations* treten nicht auf, Verletzungen der *Setup-Zeit* werden durch Einhaltung der maximalen Taktfrequenz vermieden. Der maximale Takt ist durch folgenden Zusammenhang beschrieben: $t_{\text{pd}} + t_{\text{hold}} + t_{\text{setup}} \leq t_{\text{clk}}$.

In Abbildung 18 wird der Takt am zweiten Register durch Logikschaltung und Verdrahtung verzögert. Auf diese Weise können *Hold-Violations* oder sogenannte *Races* entstehen.

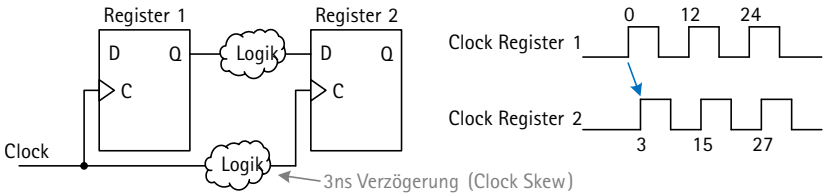

Abbildung 18: Durch Logik veränderte Taktgebung. Der Takteingang am zweiten Register ist um 3 ns gegenüber Register 1 verzögert.

Steuerung des Clock-Signals

Dieser Abschnitt befasst sich mit der Steuerung des Taktsignals. Die simple, aber aus den bereits erwähnten Gründen ungünstige Methode ist das so genannte *Clock-Gating* (vgl. Abbildung 19). Das Taktsignal wird hier über ein UND-Gatter mit einem Steuersignal verknüpft. Deshalb empfiehlt es sich, bei synchronen Schaltungen mit dem *Enable*-Eingang des Flip-Flops die Übernahme der Daten zu steuern.

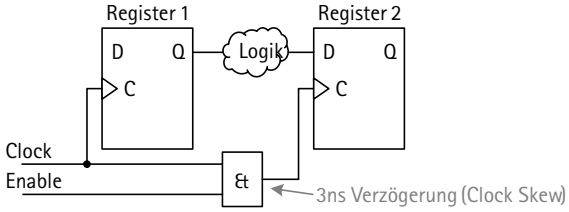

Abbildung 19: Gated Glock. Durch das zusätzliche Gatter wird eine Verzögerung des zweiten Takteingangs von 3 ns eingeführt.

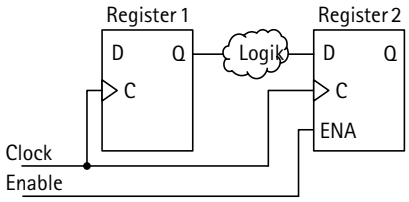

Abbildung 20: Clock-Enable. Durch die Verwendung des *Enable*-Eingangs wird keine zusätzliche Verzögerung eingeführt.

Erzeugen eines Clock-Enable-Signals

Eine Möglichkeit dieses *Enable*-Signal zu generieren besteht darin, einen Zähler zyklisch bis zu einer bestimmten Zahl zählen zu lassen und jeweils bei null das Signal `enable` für einen Taktzyklus auf '1' zu setzen. Das nachfolgende Beispiel (Abbildung 21) zeigt, wie aus dem 50 MHz Takt ein 1 Hz *Enable*-Signal generiert werden kann. Hierbei ist `FREQUENCY` eine Konstante, die auf den Wert der Clock-Frequenz in Hz gesetzt wird.

```vhdl
...
regs : process(clock, reset)
begin
  if reset = '1' then
    clock_counter_reg <= (others => '0');
    clock_enable_reg    <= '0';
  elsif rising_edge(clock) then
    clock_counter_reg <= clock_counter_next;
    clock_enable_reg    <= clk_enable_next;
  end if;
end process;

comb : process(clock_counter_reg)
begin
  if clock_counter_reg = FREQUENCY-1 then
    clock_counter_next <= (others => '0');
    clock_enable_next    <= '1';
  else
    clock_counter_next <= clock_counter_reg + 1;
    clock_enable_next    <= '0';
  end if;
end process;
...
```

Abbildung 21: Beispielprozesse zur Reduktion der Taktrate. Das Signal `clock_enable_reg` wird durch einen Zähler so erzeugt, dass es alle `FREQUENCY` Takte auf 1 gesetzt wird.

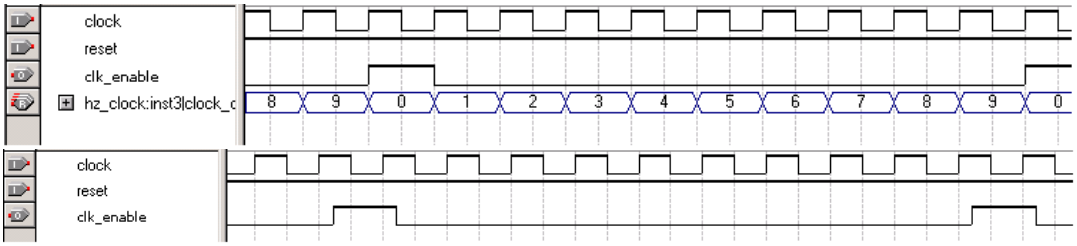

Abbildung 22: Enable-Signal (`clock_enable`) funktionale Simulation (oben) und Timing–Simulation (unten) mit `FREQUENCY` = 10.

Verwendung eines Clock-Enable-Signals

Eine geeignete Methode, einen kontrollierten, verlangsamten Takt zu verwenden, ist die Implementierung eines *Enable*-Signals, das direkt an das Flipflop angeschlossen wird.

```vhdl
process(clock, reset)
begin
  if reset = '1' then
    -- Register zurücksetzen
  elsif rising_edge(clock) then
    if clock_enable = '1' then
      -- Registerzuweisungen
    end if;
  end if;
end process;
```

Abbildung 23: Struktur einer Clock-Enable Schaltung. Die Daten werden bei einer stiegenden Taktflanke nur übernommen, wenn `clock_enable = '1'` ist.

Der VHDL-Code in Abbildung 23 zeigt das Gerüst eines solchen Aufbaus. Bitte beachten Sie, dass die *if*-Struktur hier keinen *else*-Teil hat.

2.5 Arithmetik in VHDL

Die Verwendung der Bibliotheken „*ieee.std_logic_1164.all*“ und „*ieee.numeric_std.all*“ erlauben es arithmetische und logische Operationen durch die einfache Verwendung von z.B. einem „+“-Zeichen durchzuführen.

Dabei sind in der `ieee.std_logic_1164`-Bibliothek Datentypen wie `std_ulogic` und `std_ulogic_vector` sowie logische Operationen wie z.B. UND-Verknüpfungen definiert.

Die `ieee.numeric_std`-Bibliothek baut auf der `ieee.std_logic_1164`-Bibliothek auf und definiert weitere Datentypen wie `signed` und `unsigned` und dazu eine Vielzahl von arithmetischen Operationen wie Additionen, Multiplikationen und Shifts. In Tabelle 2.1 ist ein Auszug der zur Verfügung stehenden logischen, relationalen und arithmetischen Operatoren dargestellt.

Gruppe	**Operatoren**	**Datentypen**	**Beispiele**
Logische Operatoren	`not`, `and`, `or`, `nand`, `nor`, `xor`	`std_ulogic`, `std_ulogic_vector`	`a <= not b;` `c <= d and e;`
Relationale Operatoren	$=$ (`=`), $\neq$ (`/=`), $<$ (`<`), $\leq$ (`<=`), $>$ (`>`), $>$ (`>`), $\geq$ (`>=`)	`std_ulogic`, `std_ulogic_vector`, `unsigned`, `integer`	`if a >= b then...` `if c /= d then...`
Arithmetische Operatoren	`+`, `-`	`unsigned`, `signed`, `integer`	`a <= b + c - d;`

Tabelle 2.1: Auszug der verfügbaren Operatoren mit Beispielen.

In der Bibliothek sind die erlaubten Eingangsdatentypen und die dazugehörigen Ausgangsdatentypen definiert. Werden z.B. zwei vorzeichenlose Zahlen arithmetisch mit einander verknüpft, so ist das Ergebnis ebenfalls immer vorzeichenlos.

Auf die Realisierung von arithmetischen Operationen aus logischen Gattern soll hier nicht weiter eingegangen werden. Alle definierten logischen und arithmetischen Operationen sowie Vergleiche und deren zulässige Datentypen können unter [3] und [4] eingesehen werden.

Wichtig zu beachten bei arithmetischen Operationen ist der möglich darzustellende Wertebereich mit einem Signal. Werden z.B. zwei vorzeichenlose 8-Bit Zahlen miteinander addiert, so kann das Ergebnis durchaus mehr als 8 Bit zur Darstellung benötigen. Dementsprechend müssen die Bitbreiten der beiden Eingangsoperanden für die Addition im Vorfeld auf die Bitbreite der Ergebniszahl erweitert werden. Dafür steht die Funktion `resize` zur Verfügung (Abbildung 24).

```
1 function resize(ARG: signed/unsigned, NEW_SIZE: natural)
2 return signed / unsigned;
```

Abbildung 24: Definition der Funktion `resize`.

Je nachdem ob die der Eingangsdatentyp vorzeichenlos oder -behaftet ist, wird der Vektor mit Nullen oder Einsen aufgefüllt. Der Parameter „*NEW_SIZE*“ stellt die neue Länge des Vektors in Bit dar und muss als positive Ganzzahl (`natural`) angegeben werden. Der Zahlenwert selber bleibt bei dieser Operation unverändert. Ein Beispiel zur Verwendung der `resize`-Funktion ist in Abbildung 25 dargestellt.

```
1 signal a : signed(3 downto 0);
2 signal b : signed(7 downto 0);
3 ...
4 b <= resize(a,8); -- 1000 -> 11111000
```

Abbildung 25: Beispiel zur Verwendung der `resize`-Funktion. Bei vorzeichenbehafteten Zahlen werden die Werte dem 2er-Komplement entsprechend erweitert.

2.6 Beschreibung endlicher Zustandsautomaten

Für sequentielle Abläufe oder Steueraufgaben jeglicher Art, können endliche Zustandsautomaten eingesetzt werden. Diese bieten den Vorteil einer hohen Abstraktionsebene, aber gleichzeitig eine gute Umsetzbarkeit in Hardware. Viele Synthese-Programme wie zum Beispiel Quartus Prime können beschriebene Zustandsautomaten erkennen, hoch optimiert in Hardware umsetzen und auch grafisch wieder anzeigen.

Der Zustandsautomat aus Abbildung 26 enthält zwei Zustände (S_1 und S_2). Der Zustand S_1 wird gehalten, bis das Eingangssignal „*test_sig*“ auf „1“ wechselt. Der Zustand S_2 wird nach einem Takt wieder verlassen und zurück in den Zustand S_1 gewechselt. Der schwarze Punkt an Zustand S_1 markiert dabei den Anfangszustand nach einem Reset der Hardware.

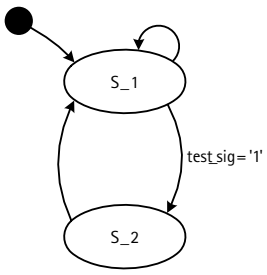

Abbildung 26: Zustandsautomat mit zwei Zuständen.

In Abbildung 27 wird der beschriebene Zustandsautomat in VHDL abgebildet. In Zeile 5 wird zunächst ein Datentyp erstellt und definiert, welche Werte dieser Datentyp annehmen kann.

Bei der Synthese werden diese abstrakten Werte durch Bit-Werte ersetzt. In Zeile 6 erstellt werden entsprechende Signale vom definierten Datentyp erstellt, die die Zustände repräsentieren sollen.

Generell besteht die Beschreibung eines Zustandsautomaten, wie auch Register, aus der Zwei-Prozess-Struktur. Im sequentiellen Prozess (`fsm_seq`) wird der Folgezustand dem aktuellen Zustand zugewiesen. Im kombinatorischen Prozess (`fsm_comb`) wird der Folgezustand anhand von Fallunterscheidungen und dem aktuellen Zustand bestimmt. Um Latches zu vermeiden bietet es sich an zunächst für alle Register und Ausgangssignale des Zustandsautomaten Standardzuweisungen zu definieren (z.B. Zeile 22).

```
1  ...
2   signal test_sig    : std_ulogic;
3   signal test_output : std_ulogic;
4
5   type state_t is (S_1, S_2);
6   signal state, state_nxt : state_t;
7
8  begin
9
10  fsm_seq : process(clock, reset)
11  begin
12    if reset = '1' then
13      state <= S_1;
14    elsif rising_edge(clock) then
15      state <= state_nxt;
16    end if;
17      end process;
18
19  fsm_comb : process(state, test_sig)
20  begin
21    -- Standardzuweisungen
22    state_nxt <= state;
23    test_output <= '0';
24
25    case state is
26      when S_1 =>
27        -- Signalzuweisungen
28        ...
29        if test_sig = '1' then
30          state_nxt <= S_2;
31        end if;
32
33      when S_2 =>
34        -- Signalzuweisungen
35        test_output <= '1';
36        ...
37        state_nxt <= S_1;
38
39
40    end case;
41
42  end process;
43  ...
```

Abbildung 27: Beschreibung eines Zustandsautomaten in VHDL.

Der eigentliche Automat wird mit einer case-Struktur definiert. Wichtig dabei ist, dass diese case-Struktur alle Werte, die der Datentype des Signals `state` annehmen kann. Ein „*when others* =>“ wird hingegen nicht benötigt. Innerhalb der Zustände können Signalzuweisungen (z.B. Zeile 35) und Verzweigungen (z.B. Zeile 29) eingefügt werden. Achten Sie darauf, dass in einem VHDL-Prozess immer nur die letzte Zuweisung eines bestimmten Signales ausgeführt wird.

In Ihrer Implementierung sollten die Zustände sinnvoller benannt werden.

Kapitel 3

Einführung in die Entwicklungsumgebung Quartus

Quartus ist eine vielseitige Software zur Programmierung von FPGAs der Firma Intel. Dem Benutzer wird eine grafische Oberfläche zum direkten Zusammenschalten einzelner VHDL-Module, ein Texteditor mit Hervorhebung von VHDL Schlüsselwörtern, verschiedene Konfigurationsmenüs zum Anpassen FPGA-spezifischer Einstellungen (z.B. Zuweisung von Pins) und eine Schnittstelle zur Programmierung eines FPGAs zur Verfügung gestellt. Ziel dieses Tutorials ist es, den Umgang mit der Programmierumgebung Quartus für das Labor zu erlernen. Hierfür orientiert sich dieser Abschnitt an der Herangehensweise für den ersten Versuch. Für andere Versuche müssen einige Schritte entsprechend angepasst werden.

3.1 Starten von Quartus

Quartus lässt sich mit einem Doppelklick auf der entsprechenden Verknüpfung auf dem Desktop starten. Das Symbol der Verknüpfung ist in Abbildung 28 gezeigt.

Abbildung 28: Verknüpfungssymbol von Quartus.

Die Software benötigt dabei einen Moment zum Starten.

3.2 Laden eines vorhandenen Projekts

Nach dem Starten der Software, erscheint eine grafische Benutzeroberfläche (engl. *graphical user interface*, GUI), bei der die meisten Elemente ausgegraut/deaktiviert sind. Ein Bild der grafischen Oberfläche ist in Abbildung 29 dargestellt. Am oberen Rand des Bildschirms sind verschiedene Menüleisten angebracht, auf der linken Seite befinden sich der „Project Navigator“ sowie die „Tasks“ und in der Mitte des Bildschirms wird blauer Dialog mit dem Titel „Getting Started With Quartus II Software“ angezeigt.

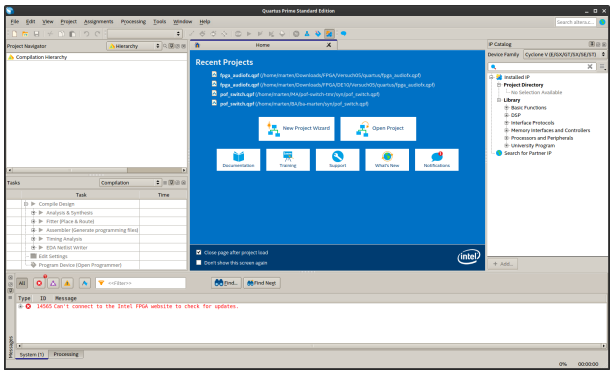

Abbildung 29: Grafische Benutzeroberfläche von Quartus direkt nach dem Start des Programms.

Im Rahmen dieses Labors sollen für die einzelnen Versuchsteile bereits angelegte Template-Projekte verwendet werden. Um ein vorhandenes Projekt zu öffnen, muss im blauen Dialog auf der linken Seite das Feld „Open Existing Project“ mit einem Klick der linken Maustaste betätigt werden. Es öffnet sich ein Durchsuchen-Dialog mit dem Titel „Open Project“ zum Suchen der entsprechenden Projekt-Datei, wie in Abbildung 30 zu sehen. Zum Öffnen des Projekts für den ersten Versuch, muss in den Ordner „Dokumente → mechlab → lab1“ gewechselt werden. In diesem Ordner befindet sich die Datei „Seg_Counter.qpf“, welche die Projekt-Datei für den ersten Versuch darstellt. Die Abkürzung .qpf steht dabei für „Quartus Project File“.

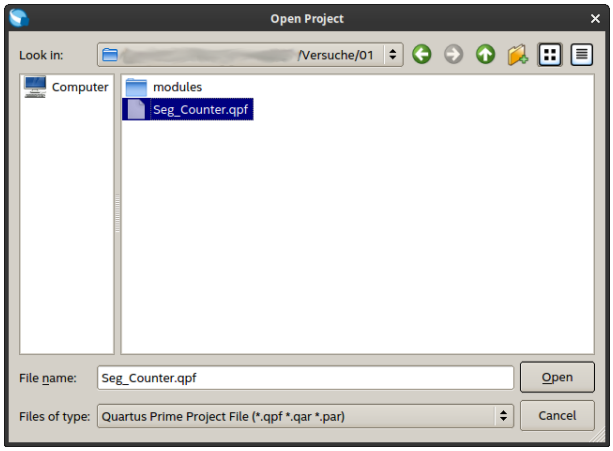

Abbildung 30: Durchsuchen-Dialog zum Öffnen einer Projekt-Datei.

Mit einem Doppelklick der linken Maustaste auf die Projekt-Datei im Durchsuchen-Dialog lässt sich das Quartus-Projekt öffnen. Dieser Schritt benötigt nur einen kurzen Moment.

3.3 Editieren von VHDL-Modulen

Um eine Übersicht über das Projekt zu bekommen, muss zunächst im „Project Navigator“ der Reiter „Files“ ausgewählt werden (siehe Abbildung 31).

Mit einem Doppelklick der linken Maustaste auf die Datei „versuch.bdf“ im „Project Navigator“ öffnet sich eine grafische Übersicht von mehreren zusammengeschalteten VHDL-Modulen (Abbildung 32). In diesem Fall sind drei VHDL-Module miteinander verschaltet: „ClockGen“, „Counter“ und „SegmentDecoder“. Diese Blöcke stellen die Entity eines VHDL-Moduls dar, also nur die Ein- und Ausgänge.

Des Weiteren befinden sich ganz links virtuelle Eingangs- und ganz recht virtuelle Ausgangspins. Diese virtuellen Pins sind bereits mit physikalisch vorhandenen Pins des FPGAs verbunden, um z.B. eine 7-Segment-Anzeige anzusteuern. Mit welchem physikalischen Pin ein virtueller Pin verbunden ist, wird in der Box links oder rechts neben dem Pin angezeigt. Werden in einer dieser Boxen nur Punkte angezeigt, so passen nicht alle Pin-Namen in diese Box. Ein virtueller Pin kann zudem mit mehreren physikalischen Pins verbunden sein. Im Falle eine 7-Segment-Anzeige ist hier nur ein virtueller Pin vorhanden, der aber entsprechend mit allen sieben physikalischen Pins des FPGAs verbunden ist, die mit der 7-Segment-Anzeige auf dem DE2 Board verbunden

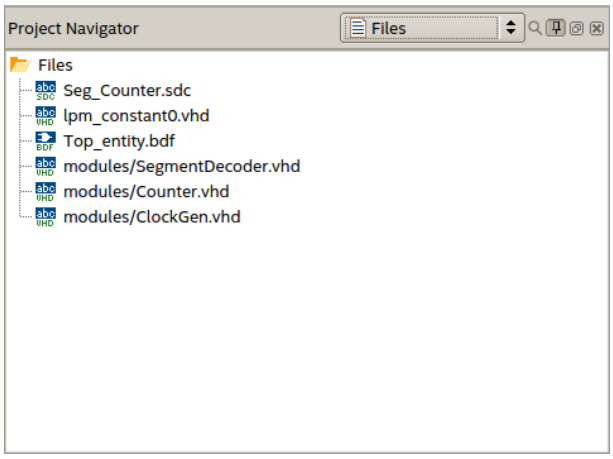

Abbildung 31: Project Navigator.

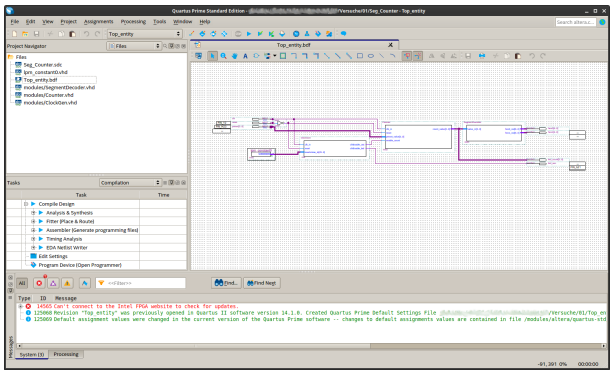

Abbildung 32: Zusammengeschaltete VHDL-Module in Quartus.

sind. Die Verdrahtung zwischen den einzelnen VHDL-Modules wird hier mit dunkelroten Linien dargestellt. Ist die Linie dünn, so wird genau ein Bit über diese Linie verbunden, wie beispielsweise ein Taktsignal (Clock). Ist die Linie dick, so werden damit mehr als ein Bit verbunden (Bus). Im weiteren Verlauf des Labors muss aber niemals etwas an den virtuellen Ein-, Ausgangspins oder an der Verdrahtung der Module geändert werden.

Um ein VHDL-Module zu bearbeiten, kann man die entsprechende VHDL-Datei im „Project Navigator“ auswählen und mit einem Doppelklick der linken Maustaste öffnen. Alternativ ist es auch möglich direkt mit einem Doppelklick auf einen Block die dazugehörige VHDL-Datei zu öffnen. Es öffnet sich dabei ein neuer Reiter im Hauptfenster der Software, der den Inhalt der entsprechenden VHDL-Datei im Texteditor anzeigt (Abbildung 33).

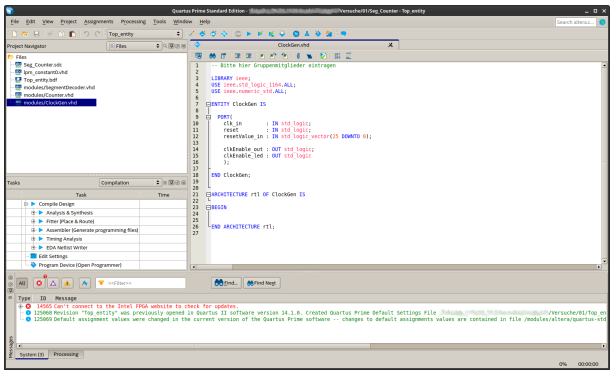

Abbildung 33: Anzeige einer VHDL-Datei im Quartus Prime-Texteditor.

Dort kann nun die VHDL-Datei erweitert werden. Um Änderungen zu speichern kann die Tastenkombination „Strg + S“ benutzt werden oder über die obere Menüleiste der Menüpunkt „File → Save“ ausgewählt werden. Ist dieser Menüpunkt ausgegraut/deaktiviert, so ist die aktuelle Version der Datei bereits gespeichert. Mit einem einen Klick der linken Maustaste auf dem „X“ am rechten Rand des Reiters, kann dieser entsprechend

wieder geschlossen werden.

Quartus kommt mit einem simplen Code-Editor. Neben dem Speichern und Laden von Dateien nimmt dieser ein Syntaxhighlighting vor.

3.4 Design erstellen

Um nun ein VHDL-Design zu erstellen, muss in der oberen Menüleiste der Menüpunkt „Processing → Start Compilation“ betätigt werden. Quartus beginnt nun damit, alle VHDL-Module entsprechend ihrer Zusammenschaltung in eine Bit-Datei zu übersetzen, mit der das FPGA anschließend programmiert werden kann. Der Fortschritt wird dabei in den „Tasks“ angezeigt (Abbildung 34).

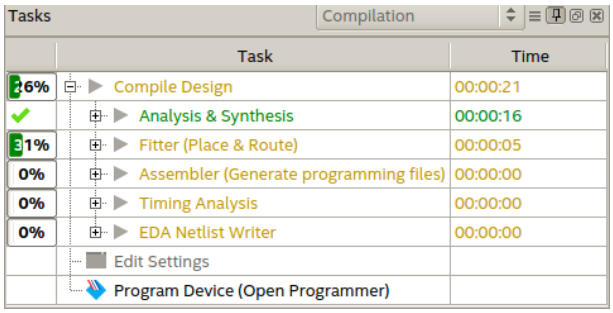

Abbildung 34: Fortschrittsanzeige einer Übersetzung.

Im obersten Punkt „Compile Design“ wird der Gesamtfortschritt der Übersetzung angezeigt. Ist die Anzeige bei 100% und vor jedem der Unterpunkte ein grüner Haken vorhanden (siehe Unterpunkt „Analysis & Synthesis“ in Abbildung 34), konnte das Design erfolgreich übersetzt werden. Das Ergebnis des letzten Schritts ist eine .sof-Datei (SRAM Object File), mit der das FPGA programmiert wird.

Eventuelle Warnungen oder Fehler werden direkt während der Übersetzung in einer Nachrichten-Box am unteren Rand des Bildschirms ausgegeben (siehe Abbildung 29). Zusätzlich sind dort am unteren Rand der Box verschiedene Reiter vorhanden mit denen sich Nachrichten nach ihrem Typ filtern lassen. Fehler müssen grundsätzlich immer behoben werden, da diese so schwerwiegend sind, dass das Design nicht übersetzt werden kann. Zudem sollten nach Möglichkeit auch (kritische) Warnungen behoben werden, da diese möglicherweise auf ungewolltes Verhalten im Design hinweisen.

Nach erfolgreichem Abschluss einer Übersetzung wird zudem ein neuer Reiter im Hauptfenster mit einer Zusammenfassung angezeigt. Die dort angezeigten Daten enthalten detaillierte Informationen über Geschwindigkeit und Ressourcenverbrauch des übersetzten Designs, auf die in diesem Tutorial nicht weiter eingegangen werden soll.

3.5 FPGA programmieren

Um das FPGA zu programmieren muss zunächst der „Programmer“ gestartet werden. Dieser befindet sich in der oberen Menüliste unter „Tools → Programmer“. Mit einem Klick auf den Menüpunkt öffnet sich ein Fenster (Abbildung 35).

Über diesen Dialog lässt sich das FPGA programmieren. Dabei müssen im Regelfall keine weiteren Einstellungen vorgenommen oder verändert werden. Es ist lediglich wichtig darauf zu achten, dass rechts neben der Schaltfläche „Hardware Setup“ der „USB-Blaster [USB-0]“ als Verbindungsschnittstelle zum DE2 Board angezeigt wird.

Über die Schaltfläche „Start“ auf der linken Seite des Dialogs wird das FPGA mit der rechts daneben angegebenen Datei programmiert. Der Fortschritt der Programmierung wird oben rechts im Dialog bei „Progress:“ angezeigt. Ist die Anzeige dort wie in Abbildung 35 bei 100 %, wurde das FPGA erfolgreich programmiert und der Dialog kann wieder geschlossen werden. Eventuelle Änderungen sollten beim Schließen des Dialogs nicht gespeichert werden.

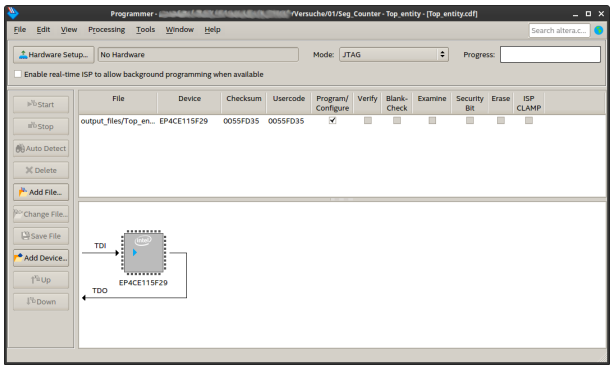

Abbildung 35: Dialog „Programmer“.

Kapitel 4

Tipps und Tricks

Um das Bearbeiten der folgenden Versuche ein wenig zu vereinfachen sollten Sie folgende Tipps und Tricks im Hinterkopf behalten. Zum Einen dienen diese dazu den Code übersichtlicher zu halten und das finden von Fehlern zu vereinfachen. Zum Anderen helfen diese Konventionen dabei, Fehler von Anfang an zu vermeiden.

Benennung von Registersignalen Zur Darstellung eines Registers empfiehlt es sich zwei Signale zu verwenden. Dabei stellt eines den aktuellen Wert des Registers und eines den neu Wert – welcher zur nächsten Taktflanke übernommen wird – dar. Um diese beiden Signale von „normalen“ Kontroll- und Datensignalen unterschieden zu können, können diese z.B. wie folgt benannt werden: `x_reg` oder `x_ff` für das Signal des aktuellen Wertes und `x_nxt` oder `x_next` für das Signal des folgenden Wertes.

Lesen und Schreiben von Signalen Aufgrund der Art und Weise wie der VHDL-Code interpretiert wird, kann ein Signal nur innerhalb von einem einzigen Prozess oder nur außerhalb der Prozesse im Beschreibungsabschnitt der Architektur beschrieben werden. Der Wert des Signals kann dafür in jedem anderen Prozess gelesen werden. Wir das Signal in einem Prozess beschrieben, sollte im selben Prozess nicht aus diesem gelesen werden.

Datentypen sinnvoll wählen Durch den Umstand, dass VHDL eine stark typisierte Beschreibungssprache ist, kommt es häufig zu Situationen, in denen viele Signalkonvertierungen stattfinden müssen. Dies kann unter Umständen durch die richtige Wahl des Datentypens eines Signals vereinfacht werden. Folgendes Beispiel zeigt dieses deutlich:

```vhdl
signal counter_reg, counter_nxt : std_ulogic_vector(8 downto 0);
...
if counter_reg >= std_ulogic_vector(to_unsigned(128, counter_reg'length)) then
    counter_nxt <= std_ulogic_vector(unsigned(counter_reg) + 1);
end if;
```

↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓

```vhdl
signal counter_reg, counter_nxt : unsigned(8 downto 0);
...
if counter_reg >= to_unsigned(128, counter_reg'length) then
    counter_nxt <= counter_reg + 1;
end if;
```

Faustregel: Finden häufig arithmetische Operationen/Vergleiche statt – wie zum Beispiel bei Zählern (welche sehr häufig zur Verwendung kommen) – bieten sich ein Datentyp mit direkter Zahleninterpretation an.

Zustände sinnvoll benennen Gerade große Zustandsautomaten können schnell unübersichtlich werden. Deshalb sollten Zustände sinnvoll benannt werden.

Standardzuweisungen für Signale Um Latches kategorisch auszuschließen sollte allen Signalen, die in einem Prozess beschrieben werden, am Anfang dieses ein Standardwert zugewiesen werden. Dadurch wird verhindert, dass ein Fälle gibt, in denen das Signal nicht beschrieben wird, wodurch das Synthesetool in diesem Fall ein Latch synthetisieren würde. Außerdem hilft es dabei, den Code ingesamt ordentlicher

zu halten: Dadurch, dass der Standardwert zu beginn des Prozesses gesetzt wird, muss das Signal nur noch in davon abweichenden Fällen beschrieben werden, was zu kürzerem, übersichtlicheren und verstehbareren Code führen kann.

```vhdl
case state is
    when idle =>
        a <= '0';
        b <= '1';
        c <= '1';
    when start =>
        a <= '1';
        b <= '0';
        c <= '1';
    when finish =>
        a <= '0';
        b <= '0';
        c <= '0';
end case;
```

```vhdl
a <= '0';
b <= '0';
c <= '1';
case state is
    when idle =>
        b <= '1';
    when start =>
        a <= '1';
    when finish =>
        c <= '0';
end case;
```

Groß- und Kleinschreibung Generell wird in VHDL nicht zwischen Groß- und Kleinschreibung unterschieden. Es empfiehlt sich allerdings bei einer der beiden Varianten zu bleiben, um den Code übersichtlicher zu halten.

Kommentare In VHDL können Zeilenkommentare eingefügt werden. Dies erfolgt durch einen doppelten Strich (--). Blockkommentare sind nicht möglich.

Feste Werte Zuweisungen von festen Werten zu `std_ulogic` erfolgen mit einfachen ('0') und bei `std_ulogic_vector` sowie `unsigned`/`signed` mit doppeltem Anführungszeichen ("0000"). Sollen Zahlenwerte für `unsigned`/`signed` zugewiesen werden, muss die entsprechende Konvertierungsfunktion verwendet werden.

Ordentlicher Reset Der Zustand der Implementierung wird in den Registern gespeichert. Um nach einem Reset immer ein deterministisches Verhalten zu erhalten, sollten alle Register durch ein Reset-Signal auf einen sinnvollen Standardwert zurückgesetzt werden können.

Literaturverzeichnis

[1] Blume, H.; Feldkämper, H.; Noll, T.G: „Model based exploration of the design space for heterogeneous systems on chip“, Journal of VLSI Signal Processing 40, 2005, pp. 19 – 34

[2] Doulos: „VHDL Golden Reference Guide“, 2003

[3] Std_logic_1164 Standard VHDL Syntheses Package: standard_vhdl_packages http://www.csee.umbc.edu/portal/help/VHDL/packages/std_logic_1164.vhd

[4] Numeric_Std Standard VHDL Syntheses Package: vhdl_packages http://www.csee.umbc.edu/portal/help/VHDL/numeric_std.vhdl

[5] Altera Corporation: „Configuring Altera FPGAs“, http://www.altera.com/literature/hb/cfg/cfg_cf51001.pdf, 2012

[6] Altera Corporation: „Cyclone IV FPGA Device Family Overview“, http://www.altera.com/literature/hb/cyclone-iv/cyiv-51001.pdf, 2014

[7] Altera Corporation: „Introduction to Quartus II Software“, http://www.altera.com/literature/manual/quartus2_introduction.pdf 2011

[8] Altera Corporation: „Quartus II Handbook“, http://www.altera.com/literature/lit-qts.jsp, 2014

[9] Altera Corporation: „DE2 Development and Education Board User Manual“, 2007

[10] Altera DE2 Development and Education Board User Manual Download: ftp://ftp.altera.com/up/pub/Webdocs/DE2_UserManual.pdf, 2006