Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

Anleitung zum

Digitallabor (Digilab)

Vorläufige Fassung!

Autor: Prof. Dr. J. Stöckle
Stand: 14.4.2016

Hochschule Karlsruhe – Technik und Wirtschaft
Fakultät für Elektro- und Informationstechnik
Moltkestraße 30 D-76133 Karlsruhe
Tel. (0721)925-1574 Fax (0721)925-1513
E-Mail joachim.stoeckle@hs-karlsruhe.de

Vorwort

Die vorliegende Anleitung zum Digitallabor ermöglicht die selbständige Einarbeitung in die Programmiersprache VHDL und die Realisierung von Aufgabenstellungen mit dem MACH Demoboard MD-12. Ein „Durchlesen“ der Anleitung wird nicht zum Erfolg führen. Die vorgestellten Aufgabenstellungen müssen selbst programmiert und in die Praxis umgesetzt werden. Neben der Simulation ist die praktische Erprobung der Entwürfe mit dem MACH Demoboard eine wesentliche Voraussetzung für das Begreifen der Zusammenhänge zwischen den Programmkonstrukten von VHDL und der PLD-Hardware. Damit verliert die Programmiersprache VHDL relativ rasch ihren zunächst trockenen Charakter. Man erfährt unmittelbar, wie sich Programmkonstrukte praktisch auswirken. Schließlich ist auch die systematische Fehlersuche eine wesentliche Fähigkeit, die man sich nur durch die praktische Umsetzung von Projekten im Laufe der Zeit aneignen kann.

Karlsruhe, im April 2016

Prof. Dr. J. Stöckle

Inhalt

Anhang

1 Einführung und Überblick

Die Komplexität programmierbarer Logikbausteine steigerte sich in den zurückliegenden Jahren laufend. Gleichzeitig wurde die Flexibilität der Bausteinstrukturen verbessert. Die Leistungsaufnahme der Bausteine wurde erheblich reduziert. Trotz dieser deutlichen Leistungssteigerungen blieben die Kosten nahezu unverändert. Aus diesem Grund sind programmierbare Logikbausteine zu einem unverzichtbaren Bestandteil moderner Elektronik geworden.

Im Digitallabor wird der Umgang mit programmierbaren Logikbausteinen vermittelt. Die Studierenden realisieren unterschiedliche Aufgabenstellungen mit Hilfe programmierbarer Logik, z. B. Zähler, eine Ampelsteuerung oder eine Stoppuhr. Das Programm wird mit Hilfe der Hardware-Programmiersprache VHDL und mittels Schaltplaneingabe erstellt. Die Design-Software Lattice Diamond ermöglicht die Simulation des Entwurfs und die eigentliche Programmierung des Logikbausteins. Zur Realisierung der Projekte wird ein Demoboard mit dem Logikbaustein MACH XO2 eingesetzt, das über eine USB-Schnittstelle mit dem PC kommuniziert. Innerhalb des Labors wird weniger Wert auf eine umfassende Vermittlung aller programmtechnischen Details der Programmiersprache VHDL gelegt. Im Vordergrund stehen vielmehr die hardwarenahe Anwendung von VHDL und die digitaltechnische Umsetzung der Programme.

Nach erfolgreichem Abschluss des Digitallabors sind die Studierenden in der Lage, selbständig Projekte geringeren Umfangs mit Hilfe von Schaltplaneingabe und unter Verwendung der Programmiersprache VHDL durchzuführen.

1.1 Programmierbare Logikbausteine

Aus der Theorie der Digitaltechnik ist bekannt, dass alle Schaltnetze lediglich mit den Basisverknüpfungen UND, ODER und Negation realisiert werden können. Alternativ können Schaltnetze lediglich mit Hilfe von NAND-Gattern oder lediglich mit NOR-Gattern erstellt werden. Schaltwerke bestehen aus Speichern (Flipflops) und Schaltnetzen. Durch geeignete Verbindungen von Gattern und Flipflops lassen sich somit alle digitaltechnischen Aufgabenstellungen verwirklichen. Programmierbare Logikbausteine bestehen aus einer mehr oder weniger großen Anzahl von Gattern und Flipflops, die auf vielfältige Art und Weise miteinander verknüpft werden können. Die Verbindungen der Bausteine untereinander sind programmierbar. Im Laufe der Zeit wurden unterschiedliche Technologien entwickelt, die im Folgenden kurz vorgestellt werden.

Gegeben sei die disjunktive Normalform (DNF) einer Verknüpfung der Variablen a, b, c und d:

$$y = (a \wedge b) \vee (\overline{c} \wedge d) \vee (\overline{a} \wedge b \wedge \overline{c})$$

Auf Basis der DNF wurden programmierbare Logikbausteine entwickelt, die eine unmittelbare Realisierung der DNF ermöglichen. Abb. 1.1 zeigt die Struktur eines derartigen Bausteins für 5 Eingangsvariable (a, b, c, d, e) und eine Ausgangsvariable (y). Die innerhalb des Bausteins vorhandenen Negierer erzeugen die ggf. benötigten negierten Eingangssignale. Eine programmierbare UND-Matrix realisiert die UND-Verknüpfungen. Eine ODER-Verknüpfung mit 8 Eingängen bildet das Ausgangssignal y. Die Punkte markieren die zur Realisierung der obigen DNF erforderlichen UND-Verknüpfungen. Diese UND-Verknüpfungen sind programmierbar. Die dargestellte Struktur ermöglicht die Realierung aller denkbaren Verknüpfungen von 5 Eingangsvariablen, die sich aus maximal 8 Mintermen (UND-Verknüpfungen) zusammensetzen.

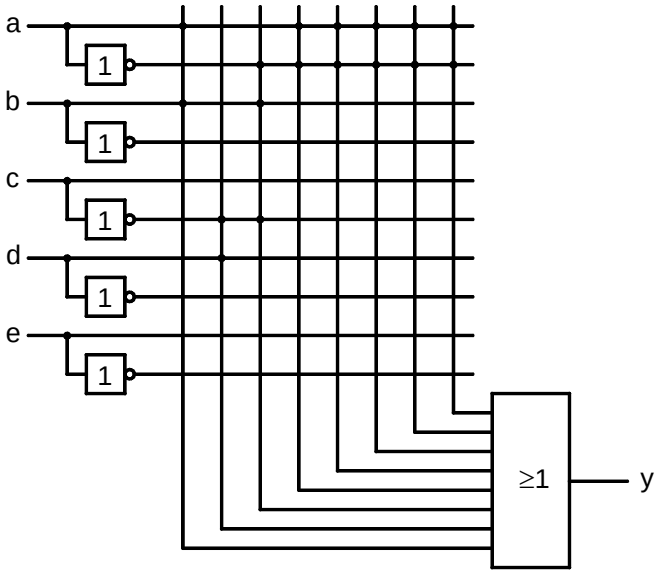


Abb. 1.1 Struktur eines programmierbaren Logikbausteins mit programmierbarer UND-Matrix

Im Beispiel besitzt der programmierbare Logikbaustein 5 Eingänge, die mit maximal 8 UND-Termen zu einer DNF verknüpft werden können. Da die UND-Verknüpfung gelegentlich auch durch einen Punkt gekennzeichnet wird, spricht man bei den UND-Termen auch von Produkttermen. Die UND-Terme können gemäß Abb. 1.2 realisiert werden.

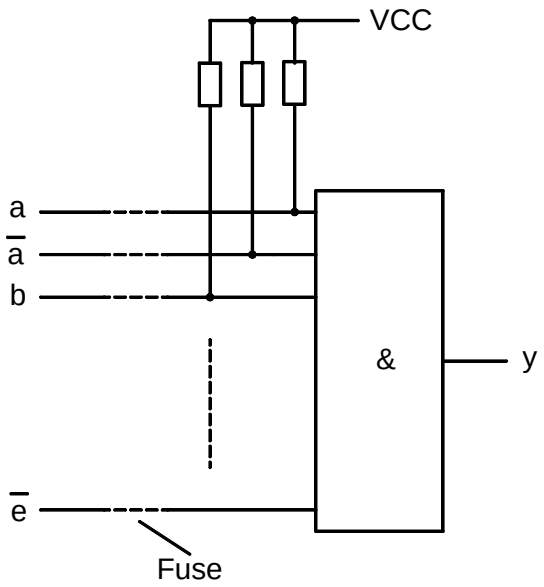


Abb. 1.2 Realisierung eines UND-Terms

Das UND-Gatter verfügt im vorliegenden Beispiel über 10 Eingänge. Alle 5 Eingangsvariablen sind zusammen mit den negierten Eingangsvariablen auf das UND-Gatter geführt. Die Eingänge sind über Pullup-Widerstände mit der Spannungsversorgung VCC verbunden. Die Programmierung des UND-Terms erfolgt durch Unterbrechung der Leitungen nicht benötigter Signale. Da der

zugehörige Eingang des UND-Gatters über den Pullup-Widerstand auf den logischen Zustand 1 gebracht wird, ist er für die UND-Verknüpfung wirkungslos. Werden eine Eingangsvariable und ihre Negierte gleichzeitig auf das UND-Gatter geführt, so ergibt sich y = 0. Der UND-Term ist dann für die ODER-Verknüpfung ohne Bedeutung.

Bei den ersten programmierbaren Logikbausteinen wurde der Signalweg unterbrochen, indem Sicherungen (Fuses) durchgeschmolzen wurden (Fuse blown). Die Programmierung konnte deshalb nicht rückgängig gemacht werden. Später erfolgte die Programmierung dadurch, dass die Signalwege mit Multiplexern geschaltet wurden. Wie die Signalwege der Multiplexer geschaltet werden sollten, wurde in einem EEPROM abgelegt. Derartige Bausteine konnten durch Löschen und erneutes Programmieren des EEPROMs mehrfach programmiert werden. Die Haltbarkeit der Daten im EEPROM betrug ursprünglich ca. 10 Jahre. Bei heutigen EEPROMs rechnet man mit einer Haltbarkeit der Daten von mindestens 50 Jahren. Auch die Programmierung moderner Bausteine erfolgt in EEPROMs.

Programmierbare Logikbausteine werden auch als PLD (Programmable Logic Device) bezeichnet. Ist das PLD mit einer programmierbaren UND-Matrix und einem festen ODER-Gatter ausgestattet, so spricht man von einem PAL (Programmable Array Logic). Ist das PAL mehrfach programmierbar, so bezeichnet man es als GAL (Generic Array Logic). Ein PLA (Programmable Array Logic) besteht aus einer programmierbaren UND-Matrix und einer programmierbaren ODER-Matrix. Die Bezeichnungen kommen allerdings nicht immer konsequent zur Anwendung.

Zur Realisierung von Schaltwerken muss das PLD auch Flipflops aufweisen. Ein Flipflop ist in einer Makrozelle untergebracht, die über einen Ein- und einen Ausgang verfügt. Die Makrozelle wird über die programmierten Schaltnetze angesteuert. Entweder gelangt das Signal über das Flipflop nach aussen oder das Flipflop wird umgangen. Die Ausgänge von Flipflops können wiederum als Eingänge für das programmierbare Schaltnetz dienen. Alle Signalwege werden mit Multiplexern geschaltet.

Moderne PLD’s bestehen aus einer programmierbaren Schaltnetzstruktur, Makrozellen, Multiplexern zur Schaltung der Signalwege und einem EEPROM zur Speicherung des Programms. Mit der Zunahme der Integrationsdichte integrierter Bausteine wuchs die Anzahl der Makrozellen bzw. Ein/Ausgänge und die Komplexität der möglichen internen Verknüpfungen mit Multiplexern. Derartige komplexe PLD’s werden als CPLD (Complex PLD) bezeichnet.

Eine Alternative zur Realisierung programmierbarer Logikbausteine besteht darin, dass die Schaltnetze über Wahrheitstabellen definiert werden und diese Wahrheitstabellen unmittelbar über Lookup Tables (LUT) umgesetzt werden. In Abhängigkeit von der Anzahl der Eingangsvariablen spricht man von einer LUT4-, LUT5- oder LUT6-Struktur. Eine LUT-Struktur und eine Makrozelle kann in einem Block zusammengefasst werden. Zahlreiche identische Blöcke können in einem integrierten Baustein untergebracht werden. Die Programmierung erfolgt durch die Programmierung der LUT-Strukturen innerhalb der Blöcke und der Verbindungen zwischen den Blöcken mit Hilfe von Multiplexern. Bausteine, die nach diesem Prinzip arbeiten, werden als FPGA bezeichnet (Field Programmable Gate Array). Diese Bezeichnung macht zudem deutlich, dass der Baustein nicht im Werk des Herstellers mit Hilfe eines geeigneten Programmiergeräts, sondern vor Ort programmiert werden kann. Hierzu muss der Baustein über eine geeignete Schnittstelle verfügen. In den vergangenen Jahren hat sich hierfür die JTAG-Schnittstelle durchgesetzt.

Tab. 1.1 gibt einen Überblick über die Entwicklung programmierbarer Logikbausteine in den vergangenen Jahren. Bei gleichbleibendem Preis nahm die Komplexität deutlich zu. Außerdem wurde die Leistungsaufnahme der Bausteine immer weiter verringert.

Jahr	Typischer Baustein	Struktur	Gehäuse	Makrozellen	Programmierung
1990	PALCE16V8	PLD	DIP20, PLCC20	8	Programmier-gerät
1998	MACH 4A	CPLD	TQFP, BGA, PLCC	32-512	JTAG
2012	MACH XO2	PLD/FPGA	TQFP, BGA, QFN	256-6864 (LUT+FF)	JTAG

Tab. 1.1 Entwicklung der programmierbaren Logikbausteine

Die Strukturen moderner programmierbarer Logikbausteine lassen sich nicht mehr konsequent den vorgestellten Strukturen (PLD bzw. FPGA) zuordnen. Moderne Bausteine bestehen aus einer Mischung unterschiedlicher Strukturen. Die Signalwege werden stets mit Multiplexern geschaltet. Das Programm wird in einem EEPROM abgelegt.

Zahlreiche Hersteller befassten sich in der Vergangenheit oder/und heute mit der Herstellung programmierbarer Logikbausteine: AMD, ATMEL, Cypress Semiconductor, Lattice, National Semiconductor, NXP (Philips), ST Microelectronics, Texas Instruments (TI), VANTIS, XILINX. Die Produkte von AMD und VANTIS wurden durch Lattice übernommen. TI erwarb National Semiconductor. Die wesentlichen Hersteller programmierbarer Logik sind heute Lattice und XILINX.

1.2 Programmierung der Bausteine

Die gewünschte logische Schaltung muss in einer maschinenlesbaren Form definiert werden, bevor sie mit einer geeigneten Software in eine programmierfähige Datei umgesetzt werden kann. Hierzu bestehen unterschiedliche Möglichkeiten. Einerseits kann die Schaltung mit einem Schaltplan-Editor (Schematic Editor) erstellt werden. Andererseits ist eine Festlegung der Schaltung mit Hilfe einer Programmiersprache (Hardware Description Language = HDL) möglich. Oft werden beide Möglichkeiten gleichzeitig genutzt. Drei Sprachen werden heute eingesetzt:

ABEL Advanced Boolean Equation Language
Verilog
VHDL Very High Speed Integrated Circuits Description Language

ABEL ist die erste Sprache, die zur Definition logischer Schaltungen eingesetzt wurde. Die Sprache ist relativ einfach strukturiert und deshalb rasch erlernbar. Mit ABEL ist eine hardwarenahe Definition logischer Schaltungen möglich. ABEL unterstützt die Eingabe von Wahrheitstabellen und die Definition der Logik durch Boolesche Algebra. Außerdem können Schaltwerke mit Hilfe von Zustandsmaschinen (State Machines) komfortabel festgelegt werden. Zur Realisierung komplexerer Aufgabenstellungen ist ABEL jedoch weniger geeignet. Für Projekte geringeren Umfangs wird ABEL weiterhin eingesetzt, da der Vorteil der kurzen Einarbeitungszeit oft von wesentlicher Bedeutung ist.

Die Sprachen Verilog und VHDL bieten die Möglichkeit zur komfortablen Definition logischer Aufgabenstellungen von geringstem bis zu größtem Umfang. Auf Grund des hierfür erforderlichen Formalismus ist die Einarbeitungszeit im Vergleich zu ABEL wesentlich höher. Verilog wird vorwiegend in USA eingesetzt. In Europa findet vor allem VHDL Verwendung.

Nach der maschinenlesbaren Definition einer logischen Schaltung mittels Schaltplan und/oder Programmiersprache (HDL) erfolgt im nächsten Schritt die Simulation des Entwurfs. Hierzu werden der entworfenen Schaltung unterschiedliche Signalverläufe zugeführt, um das Verhalten der Schaltung zu überprüfen. Die Signalverläufe können mit allen 3 oben beschriebenen Sprachen definiert werden. Man spricht von der Definition von Testvektoren.

Im Anschluss an eine erfolgreiche Simulation wird die entworfene Schaltung in den vorgegebenen Logikbaustein eingepasst (Fit Design) und eine Datei erzeugt, die zur Programmierung dient (JEDEC-Datei).

Eine Programmier-Software liest die JEDEC-Datei und überträgt die Daten über eine JTAG-Schnittstelle in den programmierbaren Logikbaustein.

Bei Bedarf ist eine Analyse des programmierten Bausteins innerhalb der erstellten Schaltung möglich (In Circuit Analyse, Verifikation). Über die JTAG-Schnittstelle kann auf interne sowie externe Signale des Logikbausteins zugegriffen werden.

Die Hersteller der programmierbaren Logikbausteine bieten kostenfreie Design-Software an, mit denen die beschriebenen Aufgaben durchgeführt werden können. Abb. 1.3 zeigt die Entwurfsschritte bei der Schaltungsentwicklung auf der Basis programmierbarer Logikbausteine im Überblick.

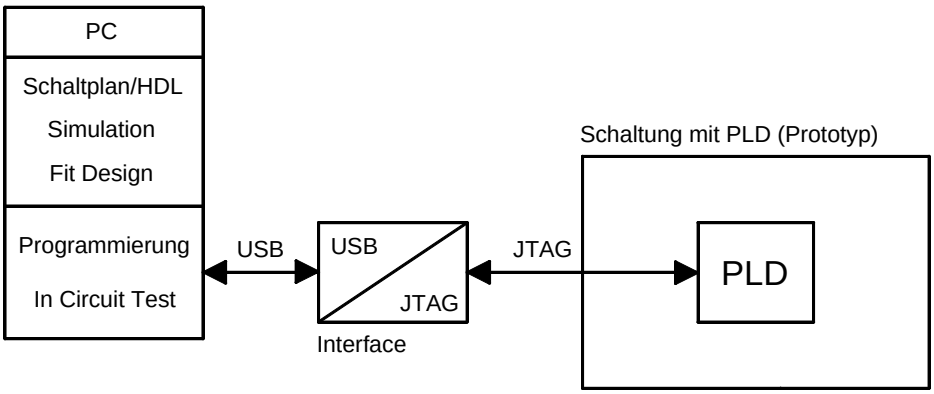


Abb. 1.3 Entwurfsschritte bei programmierbarer Logik

Da Personal Computer keine JTAG-Schnittstelle besitzen, wird die USB-Schnittstelle verwendet. Ein externer Interface-Baustein setzt die Signale der USB-Schnittstelle in die Anforderungen der JTAG-Schnittstelle um. Die JTAG-Signale werden dem auf der entwickelten Schaltung befindlichen PLD zugeführt. Die Kommunikation zwischen PC und PLD erfolgt bidirektional, so dass im PLD befindliche Daten durch den PC ausgelesen werden können. Auch zum In Circuit Test des PLD innerhalb der entwickelten Schaltung ist eine bidirektionale Kommunikation erforderlich. Das USB/JTAG-Interface kann auch in die Schaltung integriert werden, so dass diese mit der Umgebung über eine USB-Schnittstelle kommuniziert.

Die JTAG-Schnittstelle wurde ursprünglich für den Test umfangreicherer digitaler Schaltungen in einer Zusammenarbeit mehrerer Unternehmen entwickelt. Im Zusammenhang damit steht die Bezeichnung JTAG (Joint Test Action Group). Umfangreiche digitale Schaltungen können auf Basis der JTAG-Schnittstelle in Verbindung mit geeigneter Testsoftware nach der Fertigstellung automatisch auf Funktionalität überprüft werden. Man spricht in diesem Zusammenhang auch von Boundary Scan.

Seit ca. 1998 wird die JTAG-Schnittstelle auch zur Programmierung von Mikrocontrollern und programmierbaren Logikbausteinen eingesetzt. Inzwischen ist sie zur Standardschnittstelle für diese Aufgaben geworden.

Bei der JTAG-Schnittstelle handelt es sich um eine synchrone serielle Schnittstelle mit den folgenden 4 Signalen:

TDI Test Data In
TDO Test Data Out
TCK Test Clock
TMS Test Mode Select

Die Verwendung im Zusammenhang mit einem PLD ist in Abb. 1.4 dargestellt.

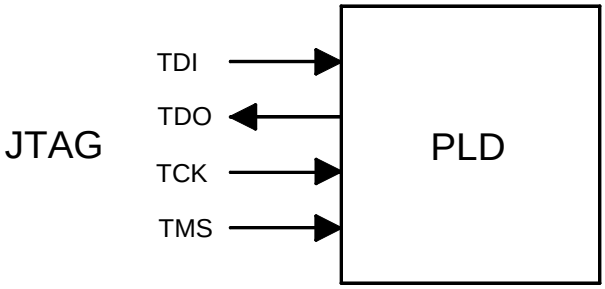


Abb. 1.4 PLD mit JTAG-Schnittstelle

2 Programmierbare Logikbausteine aus der Familie MACH XO2

Bei den programmierbaren Logikbausteinen vom Typ MACH XO2 handelt es sich um eine Bausteinfamilie der Firma Lattice, die im Jahr 2012 neu auf den Markt kam. Die Bausteinfamilie besteht aus zahlreichen Varianten unterschiedlicher Komplexität, so dass eine optimale Anpassung an individuelle Aufgabenstellungen möglich ist. Der Einstieg ist mit dem Baustein MACH XO2-256 möglich. Dieser Baustein wird im Digitallabor eingesetzt. Er ist momentan (April 2013) zu einem Preis von ca. 4 € pro Stück erhältlich. Der Baustein MACH XO2-7000 bietet die umfangreichste Funktionalität. Zwischen diesen beiden Ausführungen gibt es zahlreiche Varianten.

Die Bausteine werden in den folgenden SMD-Gehäusevarianten angeboten: BGA (Ball Grid Array), TQFP (Thin Quad Flat Package), QFN (Quad Flat No Leads Package). Innerhalb dieser Gehäuseformen gibt es verschiedene Ausführungen, die sich in der Anzahl der Anschlüsse unterscheiden. Der im Digitallabor eingesetzte Typ hat die Gehäuseform TQFP100 mit 100 Anschlüssen.

Die Bausteine werden für den industriellen Temperaturbereich -40 … 100°C und den kommerziellen Temperaturbereich 0 … 85°C angeboten. Außerdem gibt es 6 verschiedene Klassen in Abhängigkeit von der maximal zulässigen internen Taktfrequenz (speed grades). Bei Klasse 1 ist intern ein Takt von 104 MHz zulässig. In Klasse 6 sind 388 MHz intern möglich.

Die ZE-Typen weisen einen extrem geringen Leistungsbedarf auf. ZE- und HE-Typen werden mit 1,2 V gespeist. Die HC-Typen können mit 2,5 V oder 3,3 V versorgt werden.

Die Bestellnummer des im Digitallabor eingesetzten Bausteins lautet:

LCMXO2-256HC-1TG100C

In der Bestellnummer sind die folgenden Eigenschaften codiert:

Typ	HC
Versorgungsspannung	2,5/3,3 V
Klasse	1
Gehäuse	TQFP100
Temperaturbereich	0 … 85°C

Der Strombedarf des Bausteins liegt bei ca. 1 mA (ohne Beschaltung der Ein/Ausgänge). Während der Programmierung erhöht sich der Strombedarf auf ca. 15 mA. Die nebenstehende Abbildung zeigt den Logikbaustein im TQFP100-Gehäuse.

Abb. 2.1 MACH XO2-256 im TQFP100-Gehäuse

Das Blockschaltbild des Logikbausteins ist in Abb. 2.2 wiedergegeben. Der Baustein verfügt über 56 Ein/Ausgänge. Die Ein/Ausgänge können durch Programmierung sowohl als Eingang oder als Ausgang festgelegt werden. Zahlreiche Ein/Ausgänge verfügen über Mehrfachfunktionen, d. h. sie können als binäre Ein/Ausgänge oder als Teil einer seriellen Schnittstelle verwendet werden. Zur Programmierung wird normalerweise die JTAG-Schnittstelle verwendet. Zwei weitere Schnittstellen genügen dem I^2C-Standard (I^2C = IIC = Inter Integrated Circuit). Außerdem ist eine SPI-Schnittstelle vorhanden (SPI = Serial Peripheral Interface). Die Programmierung ist alternativ auch über eine I^2C- oder die SPI-Schnittstelle möglich. Bei Verwendung von Ein/Ausgängen als Schnittstellensignale reduziert sich die Anzahl der binären Ein/Ausgänge entsprechend.

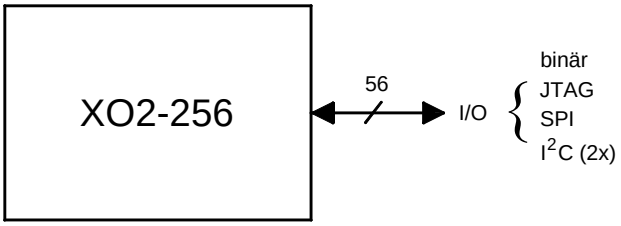


Abb. 2.2 Blockschaltbild des MACH XO2-256

In Abb. 2.3 ist der Aufbau des Logikbausteins detaillierter dargestellt. Auf jeder Seite des Bausteins befindet sich ein Block mit programmierbaren Ein/Ausgängen (PIO = Programmable I/O). Am oberen Rand des Bausteins sind die Signale der JTAG-Schnittstelle zugänglich. Der Kern des Logikbausteins wird gebildet durch den programmierbaren Logikbereich mit 32 PFU’s (PFU = Programmable Function Unit). Außerdem befinden sich im Kern des Bausteins Blöcke, die spezifischen Aufgaben zugeordnet sind. Im CFG-Block (CFG = Configuration Flash Memory) wird die Konfiguration des Bausteins gespeichert. Es handelt sich hierbei um ein Flash E^2PROM mit einer Datenhaltbarkeit von mindestens 20 Jahren (bei maximal 85°C). Im Block OSC befindet sich der interne Oszillator, der Frequenzen in einem weiten Frequenzbereich (2,08 … 133 MHz in Stufen) erzeugen kann. Außerdem ist ein Extended Function Block (EFB) vorhanden, in dem verschiedene spezifische Funktionen zusammengefasst sind, z. B. die Ansteuerung der seriellen Schnittstellen.

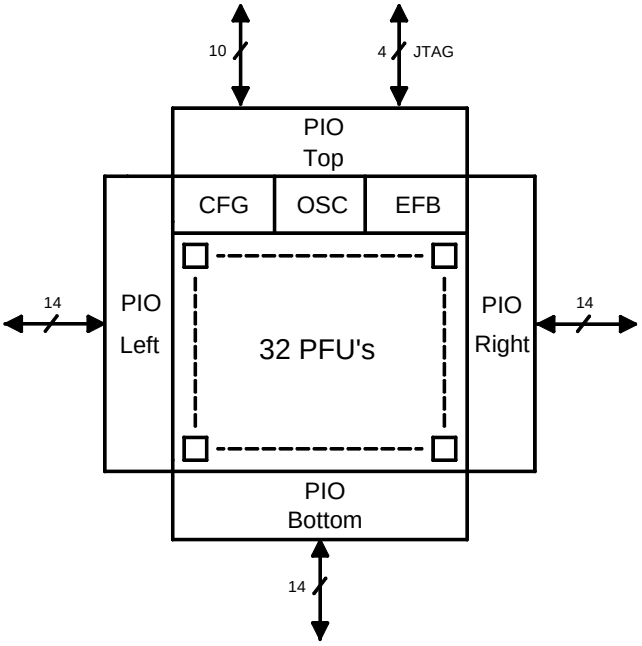


Abb. 2.3 Aufbau des XO2-256

Die Ein/Ausgangssignale können mit unterschiedlichen Logikpegeln arbeiten. Der Logikpegel eines Signals wird durch Programmierung festgelegt. Die Logikpegel sind abhängig von der Versorgungsspannung des PIO-Blocks (V_{CCIO}). Eine unterschiedliche Wahl der Versorgungsspannungen der PIO-Blöcke ist möglich. Auch die Versorgungsspannung des Bausteinkerns (V_{CC}) kann unabhängig gewählt werden. In Tab. 2.1 ist eine Auswahl der Logikpegel mit ihren Eigenschaften zusammengestellt. Die angegebenen Ausgangspegel beziehen sich auf einen Ausgangsstrom von maximal 4 mA.

I/O Standard	V_{IL} [V]		V_{IH} [V]		V_{OL} [V]	V_{OH} [V]	V_{CCIO} [V]
	min.	max.	min.	max.	max.	min.	typ.
LVCMOS33	-0,3	0,8	2,0	3,6	0,4	V_{CCIO}-0,4	3,3
LVCMOS25	-0,3	0,7	1,7	3,6	0,4	V_{CCIO}-0,4	2,5
LVCMOS18	-0,3	0,35 V_{CCIO}	0,65 V_{CCIO}	3,6	0,4	V_{CCIO}-0,4	1,8
LVCMOS15	-0,3	0,35 V_{CCIO}	0,65 V_{CCIO}	3,6	0,4	V_{CCIO}-0,4	1,5
LVCMOS12	-0,3	0,35 V_{CCIO}	0,65 V_{CCIO}	3,6	0,4	V_{CCIO}-0,4	1,2

Tab. 2.1 Logikpegel (Auswahl)

Die Zuordnung eines Logikpegels zu einem bestimmten Ein/Ausgang erfolgt durch Programmierung. Bei Eingängen ist es zudem möglich, einen Pullup-Widerstand oder einen Pulldown-Widerstand (ca. 10 kOhm) vorzusehen. Außerdem können Eingänge über einen Schmitt-Trigger verfügen. Bei Ausgängen ist es möglich, die Eigenschaft „Open Drain“ (Open Collector) oder „Tri State“ zu wählen. Alle genannten Eigenschaften werden durch Programmierung festgelegt.

Der programmierbare Logikbereich des XO2-256 setzt sich aus insgesamt 32 PFU’s zusammen. Jede PFU wiederum besteht aus 4 Slices entsprechend Abb. 2.4.

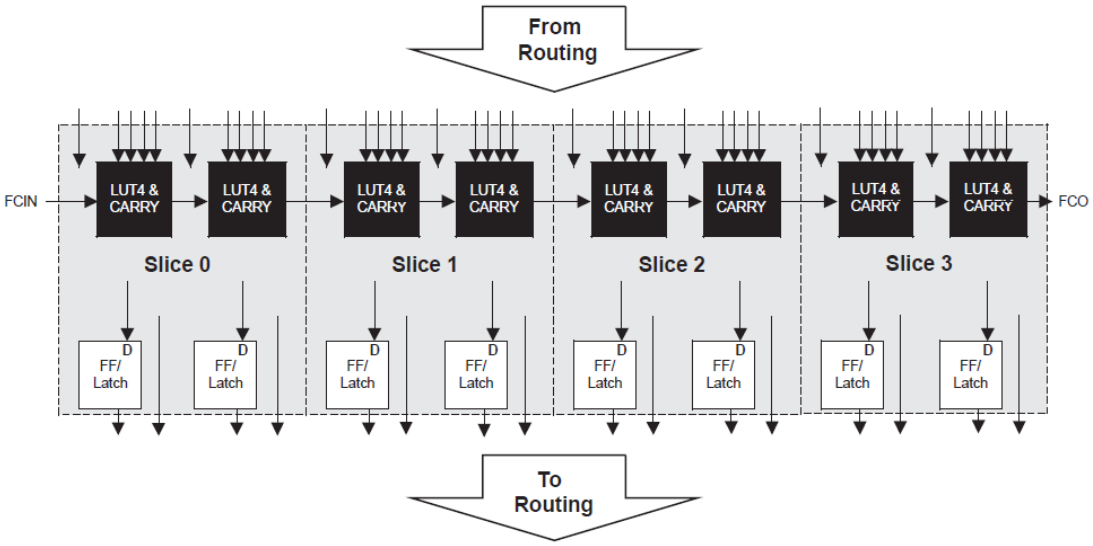


Abb. 2.4 Aufbau einer PFU

Ein Slice wiederum beinhaltet zwei Flipflops und zwei LUT4-Strukturen. Insgesamt umfasst der Logikbaustein also 32 x 8 = 256 LUTs und dieselbe Anzahl von Flipflops. Die Anzahl der LUTs bzw. Flipflops ist der Grund für die Bezeichnung des Bausteins mit XO2-256.

Da ein Slice 2 LUTs beinhaltet, sind 128 Slices vorhanden. Zahlreiche Multiplexer schalten die Signalwege zwischen den Ein- und Ausgängen der Slices (Routing). Die durch die Multiplexer festgelegten Signalwege werden durch Programmierung definiert.

Die innere Struktur eines Slice ist in Abb. 2.5 wiedergegeben. Jede LUT4-Struktur verfügt über 4 Eingänge. Mit einem LUT4 können alle denkbaren Schaltnetze mit 4 Eingangsvariablen realisiert werden.

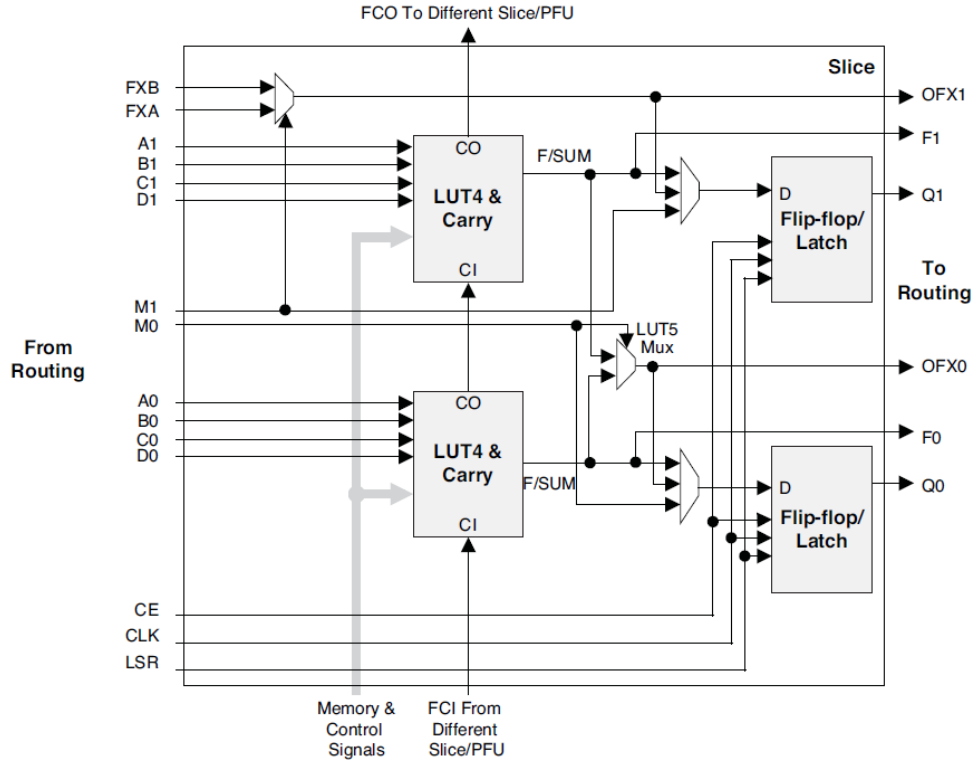


Abb. 2.5 Struktur eines Slice

Die Ausgänge Q0 und Q1 der beiden D-Flipflops werden direkt nach außen geführt. Die Signale CE (Clock Enable), CLK (Clock) und LSR (Local Set/Reset) gelangen gleichzeitig an beide Flipflops. Über mehrere Multiplexer können dem D-Eingang des unteren Flipflops entweder der Ausgang des unteren LUT4, der Ausgang des oberen LUT4 oder das externe Signal M0 zugeführt werden. Der D-Eingang des oberen Flipflops erhält alternativ einen der beiden LUT4-Ausgänge oder das externe Signal M1.

Die Ausgänge der LUT’s können auch direkt oder über Multiplexer nach außen gelangen (Signale F0/OFX0 bzw. F1/OFX1). Auf diese Weise ist es möglich, umfangreichere Schaltnetze zu verwirklichen.

Abb. 2.6 zeigt das Blockschaltbild des EFB (Extended Function Block). Dieser innerhalb des PLD angeordnete Block verfügt über feste Hardware-Strukturen zur Realisierung der Kommunikation über die seriellen Schnittstellen. Außerdem befindet sich ein 16 Bit-Zähler (Timer/Counter) im EFB. Bei Bedarf können diese Hardware-Strukturen verwendet werden, ohne dass für die betreffenden Aufgaben PFU’s verloren gehen.

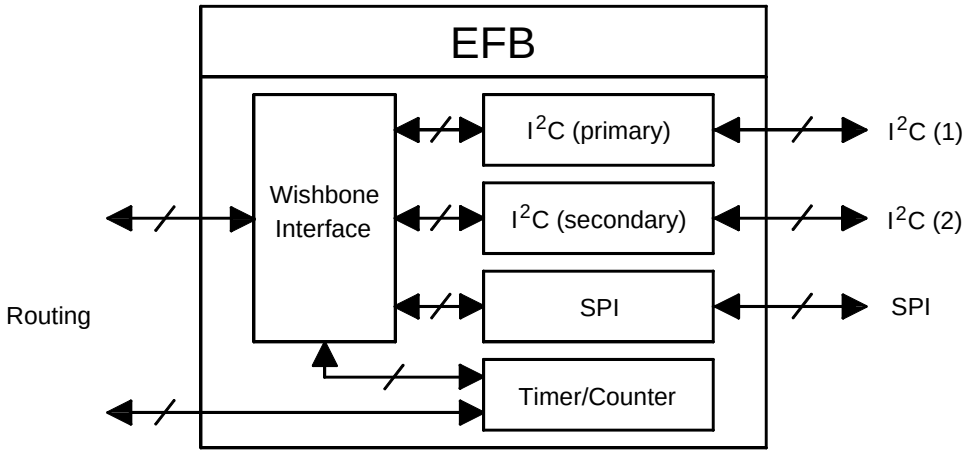


Abb. 2.6 Extended Function Block (EFB)

Die Signale der seriellen Schnittstellen sind unmittelbar über Ein/Ausgänge des XO2-256 zugänglich (Alternativfunktionen der Ein/Ausgänge). Beim Timer/Counter sind die Ein/Ausgangssignale über Routing erreichbar. Der 16-Bit-Timer/Counter benötigt verschiedene Parameter, über die die Funktionsweise festgelegt wird. So muss z. B. der maximal erreichbare Zählwert (Teiler) vorgegeben werden (1 … 65535). Außerdem ist festzulegen, welches Eingangssignal der Zähler erhalten soll. Auch die Parameter der seriellen Schnittstellen müssen vorgegeben werden. Alle Parameter können über ein Wishbone-Interface aus dem Routing-Bereich übermittelt werden. Bei einem Wishbone-Interface handelt es sich um eine standardisierte Vorgehensweise zur Datenübermittlung. Die Parameter des Timer/Counter-Bausteins sind auch direkt über den Routing-Bereich zugänglich.

3 Das MACH Demoboard MD-12

3.1 Überblick über das Demoboard

Das MACH Demoboard MD-12 dient zur Einarbeitung in programmierbare Logikbausteine der Familie MACH XO2. Es ist mit dem einfachsten Vertreter dieser Familie, dem Logikbaustein MACH XO2-256, bestückt. In Abb. 3.1 ist das Demoboard abgebildet. Die Bezeichnung der Schalter, Taster, Steckverbinder und Anzeigeelemente ist Abb. 3.2 zu entnehmen.

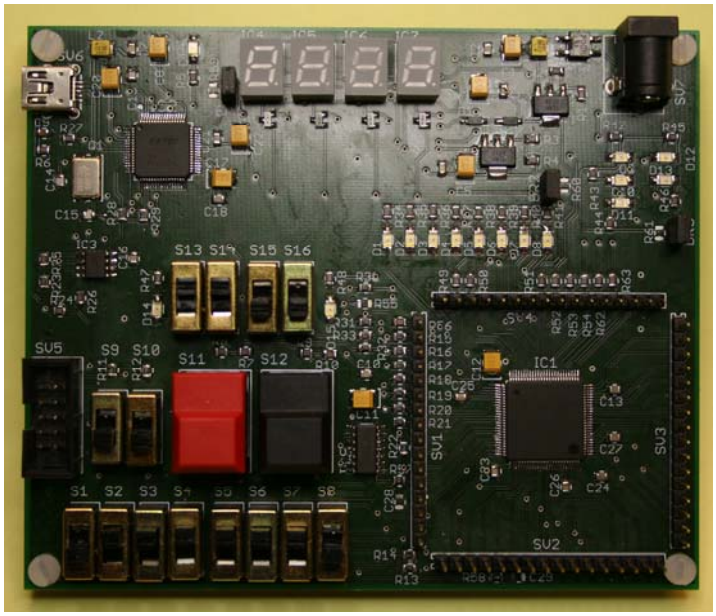


Abb. 3.1 MACH Demoboard MD-12

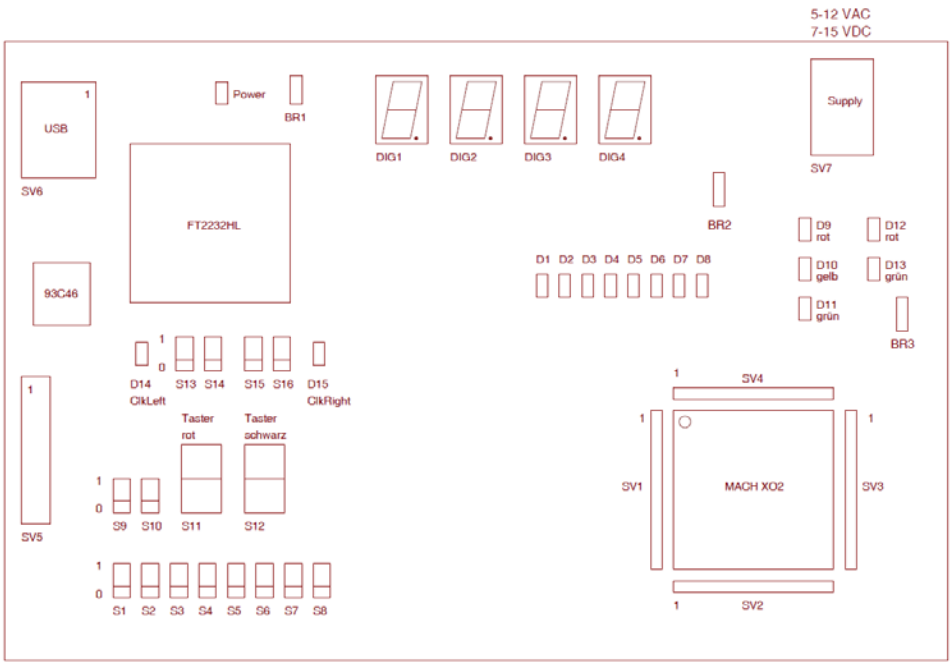


Abb. 3.2 Bezeichnung der Bauelemente des Demoboards

Das Demoboard wird über ein USB-Kabel mit einem PC verbunden. Eine Mini USB-Buchse ist links oben auf dem Demoboard vorhanden. Die Speisung des Demoboards erfolgt über die USB-Schnittstelle. Das Board kann auch mit Hilfe eines Netzteils gespeist werden. Als Versorgungsspannung wird Gleichspannung im Bereich 7-15 V oder Wechselspannung im Bereich 5-12 V verwendet. Bei Gleichspannung ist die Polarität ohne Bedeutung. Die Netzteilversorgung kann gleichzeitig mit dem USB-Anschluss erfolgen. Das Netzteil muss einen Strom von maximal 100 mA zur Verfügung stellen können. Eine grüne LED links oben signalisiert das Vorhandensein der Versorgungsspannung.

Die Programmierung des Boards erfolgt über die USB-Schnittstelle. Nach erfolgreicher Programmierung kann das USB-Kabel entfernt werden. Das Board kann dann über das Netzteil betrieben werden. Die Programmierung des Logikbausteins bleibt ohne Spannungsversorgung ca. 20 Jahre erhalten.

Da der programmierbare Logikbaustein XO2-256 nicht über eine USB-Schnittstelle verfügt, wird die vorhandene JTAG-Schnittstelle mit Hilfe des Schnittstellenwandlers FT2232HL in eine USB-Schnittstelle umgesetzt. Abb. 3.3 zeigt das Blockschaltbild der Schaltung. In einem E^2PROM vom Typ 93C46 werden bei der erstmaligen Inbetriebnahme des Demoboards die Daten gespeichert, die für eine Kommunikationsaufnahme über die USB-Schnittstelle benötigt werden, z. B. der Strombedarf des Demoboards. Unmittelbar nach Anschluss des Demoboards an die USB-Schnittstelle eines PC’s werden diese Daten ausgelesen.

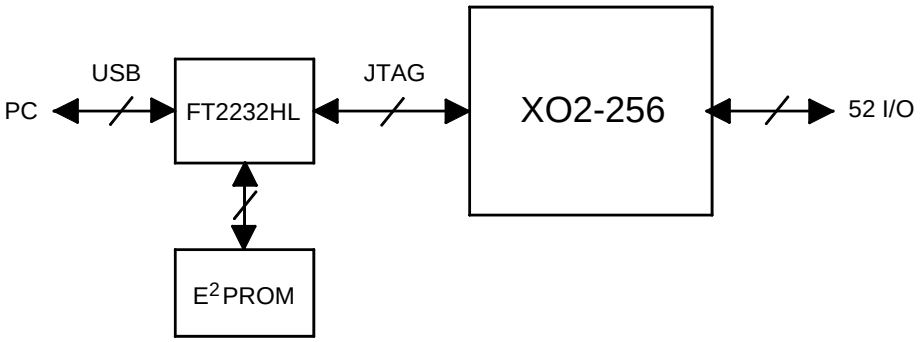


Abb. 3.3 Umsetzung zwischen USB- und JTAG-Schnittstelle

In Abb. 3.4 ist die Beschaltung der Ein/Ausgänge des Logikbausteins XO2-256 dargestellt. Der rote Taster S11 dient zur Erzeugung einzelner Impulse des Taktsignals ClkLeft. Der Taster ist entprellt. ClkLeft wird mit der LED D14 angezeigt. Mit den Schaltern S13 und S14 wird das Taktsignal ausgewählt. Mit dem schwarzen Taster S12 werden einzelne Impulse des Takts ClkRight erzeugt. Der Taster ist entprellt. ClkRight wird mit der LED D15 angezeigt. Mit den Schaltern S15 und S16 wird die Art des Taktsignals ClkRight bestimmt. Unmittelbar nach der Inbetriebnahme des Demoboards ist die Takterzeugung nicht funktionstüchtig. Der Logikbaustein muss hierzu mit dem Modul ClkGen programmiert werden. Weitere Hinweise zur Takterzeugung finden sich in Abschnitt 3.2.

Wird die Takterzeugung nicht benötigt, so ist eine Programmierung des Moduls ClkGen nicht erforderlich. Die Taster S11 und S12 sowie die Schalter S13, … , S16 können dann als Eingangssignale Verwendung finden. S11 und S12 sind durch eine Hardware-Schaltung entprellt. Alle anderen Schalter sind nicht entprellt.

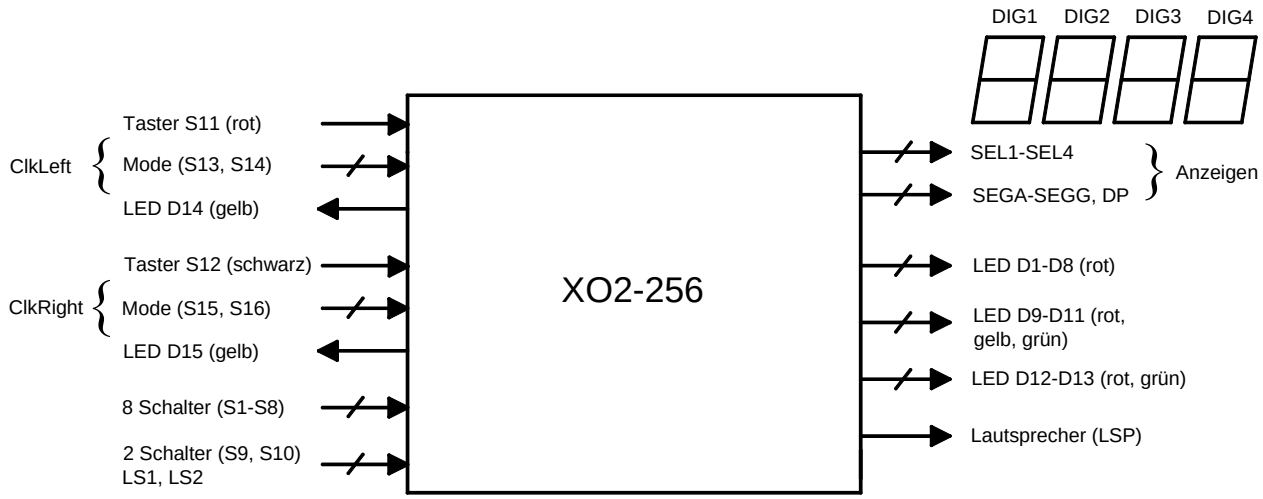


Abb. 3.4 Beschaltung der Ein/Ausgänge des Logikbausteins XO2-256 beim Demoboard MD-12

Neben den Tastern, Schaltern und LED’s, die für die Takterzeugung verwendet werden, sind 8 Schalter (S1-S8) zur Signaleingabe vorhanden. Ebenfalls zur Signaleingabe dienen die Schalter S9 und S10. Die Schalter S9 und S10 sind parallel geschaltet zu den Lichtschrankensignalen LS1 und LS2. LS1 und LS2 sind am Steckverbinder SV5 zugänglich. LS1 (Start) und LS2 (Stopp) erzeugen low-aktive Signale zum Start und Stopp eines Zählers mit Hilfe von zwei Lichtschranken. Voraussetzung für den Betrieb der Lichtschranken ist, dass sich die Schalter S9 und S10 im Zustand 1 befinden.

Mit den Ausgangssignalen SEL1-SEL4 werden die Digits DIG1-DIG4 der Anzeige ausgewählt. Die Signale SEGA-SEGG steuern die Segmente der Anzeige an. Das Signal DP schaltet den Dezimalpunkt. Alle Signale sind high-aktiv.

Die Leuchtdioden D1-D13 dienen zur Anzeige logischer Zustände. Mit dem Signal LSP kann ein externer Lautsprecher angesteuert werden. LSP ist am Steckverbinder SV5 zugänglich.

Mit den Steckbrücken BR1, BR2 und BR3 ist eine Aktivierung bzw. Inaktivierung der Anzeige oder von LED’s möglich. Wenn die Steckbrücken vorhanden sind, sind die zugeordneten Bausteine aktiviert. BR1 dient der Aktivierung der Anzeigen DIG1, … , DIG4. BR2 ist den LED’s D1, … , D8 zugeordnet. Die Steckbrücke BR3 ist verknüpft mit den LED’s D9, … , D13.

Das Demoboard eignet sich zur Realisierung zahlreicher Projekte. Beispiele hierfür sind:

- Lauflicht
- Zähler
- Stoppuhr
- Rechenwerk
- Sichere Datenübertragung mit Hamming-Codierung
- Geschwindigkeitsmessung bei Modellautos
- Frequenzerzeugung für einen externen Lautsprecher
- Timer
- Code-Wandler

3.2 Signalbelegung des Logikbausteins XO2-256

Die Zuordnung der Schalter, Taster, LED’s und Anzeigen zu den Ein/Ausgängen des programmierbaren Logikbausteins XO2-256 ist den nachfolgenden Tabellen zu entnehmen. Alle Signale sind auch auf die Steckverbinder SV1-SV4 (Stiftleisten) geführt. Die Belegung dieser Stiftleisten ist ebenfalls in den Tabellen angegeben. Die Position 1 der Stiftleisten ist Abb. 3.2 zu entnehmen. Im Normalfall sind die Stiftleisten nicht bestückt. Bei Bedarf können sie nachträglich eingelötet werden.

XO2-Pin	Demoboard		XO2-Signal	SV1-Pin
	Signal	Bauteil		
1	DI1	S1	PL2A	2
2	DI2	S2	PL2B	3
3	DI3	S3	PL2C/PCLKT3_2	4
4	DI4	S4	PL2D/PCLKC3_2	5
5			VCC	1
6			GND	16
7	DI5	S5	PL3A	6
8	DI6	S6	PL3B	7
9				
10				
11				
12	DI7	S7	PL3C/PCLKT3_1	8
13	DI8	S8	PL3D/PCLKC3_1	9
14				
15				
16	DO1	D1	PL5A	10
17	DO2	D2	PL5B	11
18				
19				
20	PULSE_S11	S11	PL5C/PCLKT3_0	12
21	DO3	D3	PL5D/PCLKC3_0	13
22			GND	16
23			VCC	1
24	DO4	D4	PL6A	14
25	TEST1		PL6B	15

Tab. 3.1 XO2-Signalbelegung (Left)

XO2-Pin	Demoboard		XO2-Signal	SV2-Pin
	Signal	Bauteil		
26			VCC	1
27	DO5	D5	PB2A/CSSPIN	2
28	DO6	D6	PB2B	3
29				
30				
31	DO7	D7	PB2C/MCLK/CCLK	4
32	DO8	D8	PB2D/SO/SPISO	5
33			GND	16

34	PULSE_S12	S12	PB4A/PCLKT2_0	6
35	DI15	S13	PB4B/PCLKC2_0	7
36				
37				
38	TEST2/USB_12MHZ		PB4C/PCLKT2_1	8
39	DI16	S14	PB4D/PCLKC2_1	9
40	DI18	S16	PB7A	10
41	DI17	S15	PB7B	11
42	LSP	Lautsprecher	PB7C	12
43	DI9	S9	PB7D	13
44			GND	16
45				
46			VCC	1
47				
48	DI10	S10	PB9A/SN	14
49	TEST3		PB9B/SI/SISPI	15
50			VCC	1

Tab. 3.2 XO2-Signalbelegung (Bottom)

XO2-Pin	Demoboard		XO2-Signal	SV3-Pin
	Signal	Bauteil		
51	SEL4	Anzeige DIG4	PR6D	15
52	SEL3	Anzeige DIG3	PR6C	14
53	SEL2	Anzeige DIG2	PR6B	13
54	SEL1	Anzeige DIG1	PR6A	12
55			VCC	1
56			GND	16
57	TEST6		PR5D	11
58	TEST5		PR5C	10
59				
60				
61				
62	DO15	D15	PR5B/PCLKC1_0	9
63	DO14	D14	PR5A/PCLKT1_0	8
64				
65				
66	TEST4		PR3D	7
67	DO13	D13	PR3C	6
68				
69				
70	DO12	D12	PR3B	5
71	DO11	D11	PR3A	4
72			GND	16
73			VCC	1
74	DO10	D10	PR2B	3
75	DO9	D9	PR2A	2

Tab. 3.3 XO2-Signalbelegung (Right)

XO2-Pin	Demoboard		XO2-Signal	SV4-Pin
	Signal	Bauteil		
76	TEST8		PT9D/DONE	15
77	TEST7		PT9C/INITN	14
78				
79			GND	16
80			VCC	1
81	SEG8	Anzeige	PT9B/PROGRAMN	13
82	SEG7	Anzeige	PT9A/JTAGENB	12
83				
84				
85	SEG6	Anzeige	PT8D/SDA/PCLKC0_0	11
86	SEG5	Anzeige	PT8C/SCL/PCLKT0_0	10
87	SEG4	Anzeige	PT8B/PCLKC0_1	9
88	SEG3	Anzeige	PT8A/PCLKT0_1	8
89				
90	TMS	JTAG	TMS	7
91	TCK	JTAG	TCK	6
92				
93			VCC	1
94	TDI	JTAG	TDI	5
95	TDO	JTAG	TDO	4
96				
97				
98	SEG2	Anzeige	PT6B	3
99	SEG1	Anzeige	PT6A	2
100			VCC	1

Tab. 3.4 XO2-Signalbelegung (Top)

VCC = 3,3 V

3.3 Ansteuerung der 7-Segment-Anzeigen

Die Bezeichnung der Segmente bei 7-Segment-Anzeigen ist Abb. 3.5 zu entnehmen. In Tab. 3.5 sind die zugeordneten Signale aufgeführt.

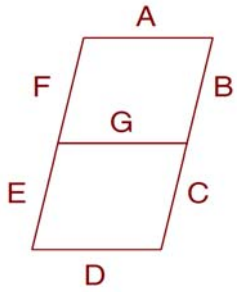


Abb. 3.5 Bezeichnung der Segmente bei 7-Segment-Anzeigen

Signal	Anzeigen-Segment
SEG1	A
SEG2	B
SEG3	C
SEG4	D
SEG5	E
SEG6	F
SEG7	G
SEG8	DP

Tab. 3.5 Signale für die Anzeigen-Segmente

Der Dezimalpunkt befindet sich jeweils rechts unten neben der 7-Segment-Anzeige. Die Signale SEG1, … , SEG8 sind parallel auf die 4 Anzeigen DIG1, … , DIG4 geführt. Zur Aktivierung der Anzeigen dienen die Signale SEL1, … , SEL4. Im Fall SEL1 = 1 ist z. B. DIG1 aktiv. Wenn alle Signale SEL1, … , SEL4 aktiviert sind, so leuchten alle Anzeigen. Alle Anzeigen zeigen dann dasselbe Zeichen an. Um 4 verschiedene Zeichen zur Anzeige zu bringen, muss ein Multiplex-Betrieb der Anzeigen erfolgen, d. h. in rascher Folge wird eine Anzeige aktiviert und gleichzeitig über die Segmentsignale der anzuzeigende Wert übertragen. Wenn die Aktivierung der Anzeigen in rascher Abfolge durchgeführt wird, erscheint der abgezeigte Wert stabil. Bewährt hat sich eine Frequenz von 200 Hz für die Umschaltung zwischen DIG1, … , DIG4.

3.4 Takterzeugung

Zur Verfügung stehen die Taktsignale ClkLeft und ClkRight. Jeder Takt kann wahlweise unterschiedliche Frequenzen aufweisen. Auch Einzelimpulse können erzeugt werden. Die Wahl der Betriebsart von ClkLeft erfolgt mit den Schaltern S13 und S14. Bei ClkRight wird die Betriebsart mit den Schaltern S15 und S16 gewählt. Der Takt ClkLeft wird an LED D14 angezeigt. Die LED D15 dient zur Anzeige von ClkRight. Einzelimpulse von ClkLeft werden mit dem roten Taster S11 erzeugt. Der schwarze Taster S12 dient zur Erzeugung von Einzelimpulsen im Takt ClkRight. Zur Takterzeugung ist das Modul ClkGen erforderlich. Dieses Modul muss in das PLD programmiert werden (siehe Abschnitt 5.2). Ohne das Modul ClkGen ist die Takterzeugung nicht funktionsfähig. Tab. 3.6 gibt einen Überblick über die Betriebsarten von ClkLeft. In Tab. 3.7 sind die Betriebsarten von ClkRight angegeben.

ModeLeft	ModeLeft1 S13	ModeLeft0 S14	ClkLeft D14
0	0	0	Impuls
1	0	1	0,3 Hz
2	1	0	1 Hz
3	1	1	3 Hz

Tab. 3.6 Betriebsarten von ClkLeft

ModeRight	ModeRight1 S15	ModeRight0 S16	ClkRight D15
0	0	0	Impuls
1	0	1	1 Hz
2	1	0	3 Hz
3	1	1	100 Hz

Tab. 3.7 Betriebsarten von ClkRight

Das Blockschaltbild der Takterzeugung ist in Abb. 3.6 dargestellt. Weitere Hinweise zur Einbindung des Moduls ClkGen in ein eigenes Programm finden sich in den Kapiteln 7.2.2 und 8.1.

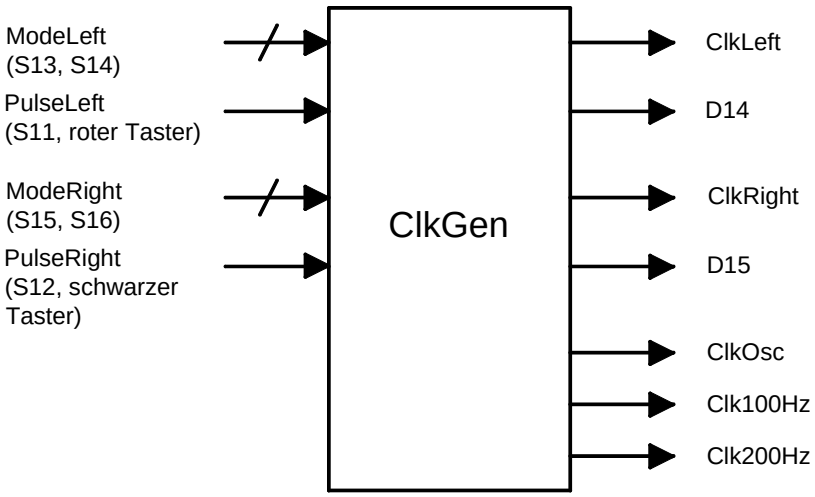


Abb. 3.6 Takterzeugung mit dem Modul ClkGen

Der Basistakt des Moduls ClkGen wird durch den PLD-internen Oszillator erzeugt (2,08 MHz). Neben ClkLeft und ClkRight stellt das Modul auch diesen Basistakt ClkOsc sowie das 100 Hz-Taktsignal Clk100Hz und das 200 Hz-Taktsignal Clk200Hz zur Verfügung.

3.5 Anschluss externer Signale

Über den Steckverbinder SV5 (siehe Abb. 3.2) können zwei etxerne Lichtschranken (LS1, LS2) und ein Lautsprecher (LSP) angeschlossen werden. Die Belegung des Steckverbinders SV5 ist Tab. 3.8 zu entnehmen.

Kontakt	Belegung	Kontakt	Belegung
1	LS1	2	LS2
3	LSP	4	
5		6	
7	VCC (3,3 V)	8	VCC (3,3 V)
9	GND	10	GND

Tab. 3.8 Belegung Steckverbinder SV5 (links unten)

Die Schaltung zum Anschluss der Lichtschranken ist in Abb. 3.7 dargestellt. Voraussetzung für den Betrieb der Lichtschranken ist, dass sich die Schalter S9 und S10 in der Position 1 befinden. Die Widerstände befinden sich auf dem Demoboard.

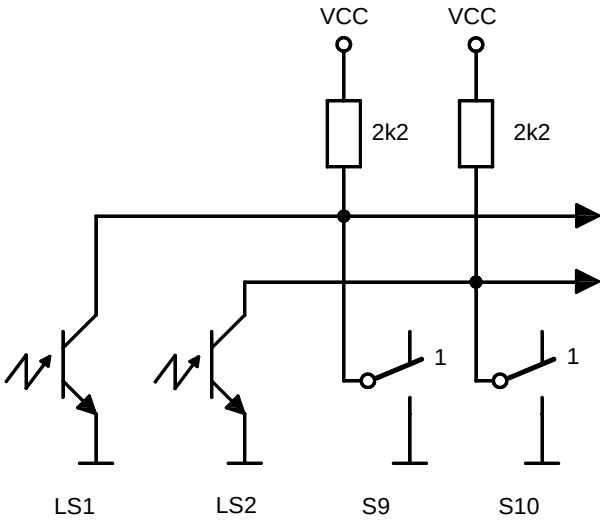


Abb. 3.7 Schaltung für Lichtschrankenbetrieb

Wenn Licht auf den Fototransistor fällt, schaltet der Transistor durch und das Signal nimmt den Wert 0 an. Im Ruhezustand weist das Signal den Wert 1 auf.

4 Installation der Diamond-Software

Die Programmierung des PLD MACH XO2-256 erfolgt mit der Entwicklungsumgebung Diamond der Firma Lattice Semiconductor. Die Software wird kostenfrei angeboten von der Firma Lattice Semiconductor und kann von der Homepage www.latticesemi.com heruntergeladen werden. Momentan ist die Version Diamond 3.7 aktuell (April 2016). Mit der Version 3.7 der Software Diamond wurden umfangreiche Änderungen der Bedienungsoberfläche eingeführt, die in dieser Dokumentation nicht berücksichtigt werden konnten. Aus diesem Grund wird im Digitallabor die Version 3.1.096 verwendet.

Die Software Diamond gibt es in einer 32 Bit- und einer 64 Bit-Ausführung für Windows-Betriebssysteme (Win XP, Win 7, Vista). Win 8 wird nicht unterstützt (siehe Anhang A7). Die Software umfasst insgesamt ca. 1,5 GB. Der Download erfolgt vom Downloadbereich der Hochschule (ca. 25 Min. bei 10 MBit/s).

Bei Win XP und Vista wird die 32 Bit-Ausführung verwendet. Um bei Win 7 festzustellen, ob das Betriebssystem mit 32 oder mit 64 Bit arbeitet, führen Sie die folgenden Schritte durch:

Start > Systemsteuerung > System

Unter "Systemtyp" wird angegeben, ob ein 32 Bit- oder ein 64 Bit-Betriebssystem vorliegt.

Im Fall eines 32 Bit-Betriebssystems laden Sie die Datei 3.1.0.96_Diamond.zip vom Downloadbereich der Hochschule und speichern Sie die Software auf Ihrer Festplatte, z. B. im Verzeichnis C:\Diamond. Bei einem 64 Bit-Betriebssystem laden Sie die Datei 3.1.0.96_Diamond_x64.zip vom Downloadbereich der Hochschule und speichern sie auf der Festplatte. Entpacken Sie die Datei und erstellen Sie die ausführbare Datei *.exe im selben Verzeichnis. Starten Sie die ausführbare Datei mit einem Doppelklick. Der Installationsvorgang beginnt. Das Programm wird im Verzeichnis C:\lscc installiert.

Wählen Sie während der Installation die folgenden Optionen:

- √ Diamond for Windows
- √ FPGAs
- √ Synplify Pro for Lattice
- √ Active HDL Lattice Edition
- √ Programmer Drivers
- √ Node Lock License
- √ Shortcut on Desktop
- √ Parallel/USB-Port Driver

Führen Sie nach Abschluss der Installation einen Neustart Ihres PC durch.

Die Dokumentation zur Software findet sich nach der Installation im Verzeichnis C:\lscc\diamond\3.1_x64\docs (64 Bit-Betriebssystem). Leider ist die Dokumentation stark fehlerbehaftet und relativ schwer verständlich, so dass eine Einarbeitung mit Hilfe dieser Dokumente nahezu unmöglich ist. Gelegentlich ist jedoch ein Rückgriff auf diese Dokumente unumgänglich. In Anhang A2 sind diese Dokumente aufgeführt.

Auf der Homepage von Lattice Semiconductor finden sich weitere Dokumente, z. B. Technical Notes. Außerdem finden sich dort FAQs. Auch über die Help-Funktion des Programms sind weitere Informationen zugänglich. Die Help-Funktion ist nur funktionsfähig, wenn der Rechner eine Verbindung zur Homepage von Lattice Semiconductor herstellen kann.

Auf die Dokumente kann auch unmittelbar nach dem Start der Software von der Startseite der Software aus zugegriffen werden. Auf Dokumente, die sich nach der Installation der Software auf der Festplatte befinden, erfolgt der Zugriff unmittelbar. Für die meisten Dokumente ist jedoch eine Internet-Verbindung zur Lattice-Homepage erforderlich.

Wenn Sie die Software nun durch Doppelklick auf das Icon Diamond 3.1 starten, erfolgt die Meldung, dass der Pfad für eine Lizenzdatei angegeben werden soll.

Zur Beantragung der Lizenz bei Lattice Semiconductor benötigen Sie die physikalische Adresse Ihrer Network Interface Card (NIC) bzw. Ihres Netzwerkadapters. Diese Adresse wird auch als MAC-Adresse (Media-Access-Control-Adresse) bezeichnet. Zur Ermittlung der NIC öffnen Sie das MS DOS-Befehlseingabefenster und geben ein:

ipconfig/all

Die Eigenschaften der installierten LAN-Adapter werden aufgeführt. Notieren Sie sich die physikalische Adresse des Ethernet Network Adapters (Festnetzverbindung). Es handelt sich hierbei um eine 12-stellige Hexadezimalzahl, z. B.

B4-B5-2F-71-E9-87

Achten Sie darauf, dass es sich um die physikalische Adresse des Festnetzadapters handelt und nicht um die Adresse eines WLAN-Adapters! Erfahrungsgemäß gibt es Probleme mit der Lizenzierung, wenn die Adressen von WLAN-Adaptern benutzt werden. Gehen Sie nun auf die Homepage der Firma Lattice. Richten Sie einen Web Account ein. Danach loggen Sie sich ein. Wählen Sie:

Support > Lattice Software Licenses > Lattice Diamond Design Software > Request a Free License

Geben Sie die 12-stellige NIC-Adresse ohne Bindestriche (!) ein und wählen Sie “Generate License”. Nach wenigen Minuten erhalten Sie eine E-Mail, in deren Anhang sich die Datei license.dat befindet. Kopieren Sie diese Datei unverändert (!) in das Verzeichnis C:\lscc\Diamond\3.1\license (32-Bit-Betriebssystem) bzw. C:\lscc\Diamond\3.1_x64\license (64-Bit-Betriebssystem). Nun müsste Diamond problemlos starten. Die Lizenzdauer beträgt 1 Jahr. Sollten Probleme mit der Lizenzierung auftreten, so finden Sie auf der Lattice-Homepage Hinweise zur Behebung der Probleme. Bei manchen Internet-Providern werden in die übermittelten Anhänge Steuerzeichen eingetragen, die Probleme bei der Lizenzierung verursachen. Lassen Sie sich die Lizenzdatei dann über Ihren Hochschul-Account zusenden.

Bei dauerhaften Problemen mit der Lizenzierung prüfen Sie die Umgebungsvariable LM_LICENSE_FILE. Der Zugriff erfolgt über

Start > Systemsteuerung > System > Erweiterte Systemeinstellungen > Erweitert > Umgebungsvariablen

Im Fall der Standard-Installation muss diese Umgebungsvariable lauten (64 Bit-Betriebssystem):

C:\lscc\diamond\3.1_x64\license\license.dat

Achtung! Wenn Sie diese Umgebungsvariable bearbeiten, ändert sich u. U. die Richtung der Schrägstriche (Slash in Backslash und umgekehrt). Die Lizenzierung kann misslingen, wenn sich irgendwo auf dem PC eine weitere Datei mit dem Namen license.dat befindet.

5 Arbeiten mit Lattice Diamond und VHDL

Die Entwicklungsumgebung Diamond ermöglicht die Programmerstellung mit Verilog oder/und VHDL sowie mit Schaltplaneingabe, die Simulation des Designs, den Programmdownload in das PLD sowie die Verifikation des Designs nach Programmierung in das PLD. Innerhalb des Digitallabors erfolgt die Programmerstellung mit VHDL und durch Schaltplaneingabe. Außerdem wird der Entwurf simuliert und in das PLD auf dem MACH Demoboard programmiert.

Achtung! Bei der Diamond Entwicklungsumgebung handelt es sich um eine amerikanische Software. Es besteht deshalb die Gefahr, dass die deutschen Umlaute ä, ö, ü, Ä, Ö, Ü und das Zeichen ß Störungen verursachen können. Achten Sie deshalb bei allen Eingaben darauf, dass diese Zeichen nicht verwendet werden. Dies gilt auch für die Eingabe von Kommentaren! Auch in Datei- und Verzeichnisnamen dürfen diese Zeichen nicht verwendet werden. Datei- und Verzeichnisnamen dürfen nicht mit Ziffern oder Sonderzeichen beginnen.

5.1 Erste Schritte mit Diamond

Starten Sie die Diamond Entwicklungsumgebung durch Doppelklick auf das Icon Diamond. Die nebenstehende Abbildung zeigt die Bildschirmdarstellung der Startseite (Start Page).

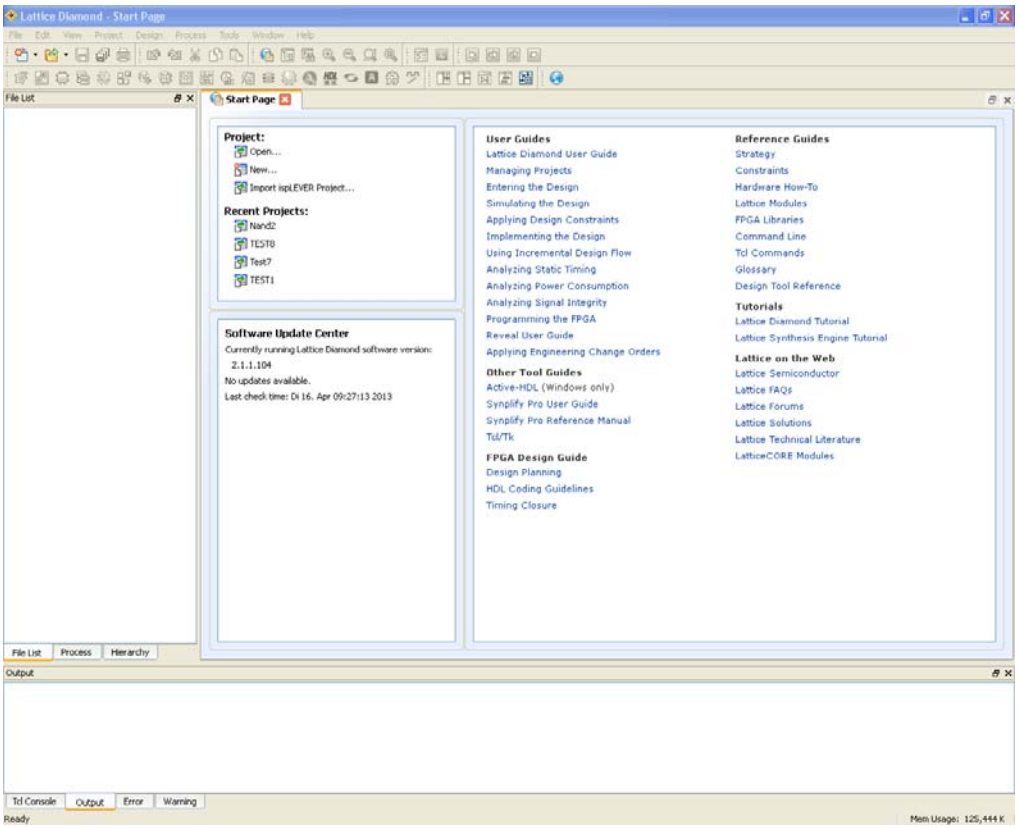


Abb. 5.1 Diamond Start Page

Rechts findet sich die Start Page, auf der verschiedene Dokumente angeboten werden. Die meisten Dokumente sind nur über eine Internet-Verbindung zur Lattice-Homepage zugänglich. Links unten in der Start Page finden sich Informationen zum Update. Wenn eine Internet-

Verbindung besteht, wird automatisch geprüft, ob ein Update verfügbar ist. Das Update kann ggf. sofort installiert werden. Links oben in der Start Page werden zuletzt durchgeführte Projekte angezeigt, die unmittelbar gestartet werden können. Links neben der Start Page befindet sich das Fenster File List, in dem bei Bearbeitung eines Projekts die zum Projekt gehörigen Dateien angezeigt werden. Mit den Tasten unterhalb dieses Fensters kann die Anzeige auf Process oder Hierarchy umgeschaltet werden. Bei Wahl von Process werden die innerhalb eines Projekts durchführbaren Aktionen dargestellt.

Am oberen Rand des Bildschirms sind Pulldown-Menüs zugänglich, über die verschiedene Aufgaben ausgelöst werden können. Für häufig vorkommende Aufgaben sind Schaltflächen mit entsprechenden Symbolen vorhanden.

Der untere Rand des Bildschirms ist das Output-Fenster, in dem Warnungen und Fehler angezeigt werden, die bei der Durchführung eines Projekts anfallen. Durch Ziehen mit der Maus kann das Fenster vergrößert oder verkleinert werden. Am unteren Rand befinden sich Tasten, mit denen die anzuzeigenden Informationen ausgewählt werden können. Nach Wahl der Taste Tcl Console ist die Eingabe von Befehlen möglich.

Bei der Benutzung von Diamond wird gelegentlich auf ispLEVER hingewiesen. Hierbei handelt es sich um eine Entwicklungsumgebung der Firma Lattice, die seit dem Jahr 1998 zur PLD-Programmierung eingesetzt wurde. Bei Diamond besteht die Möglichkeit, Daten aus ispLEVER zu übernehmen. Die Bausteinfamilie MACH XO2 wird durch ispLEVER nicht unterstützt.

Beenden Sie Diamond mit

File > Exit

5.2 VHDL-Grundlagen

VHDL wurde erstmals definiert im Jahr 1987 in der Norm IEEE 1076-1987 (VHDL-1987). Später erfolgten Erweiterungen und Ergänzungen dieser Norm in den Jahren 1993 (VHDL-1993), 2002 (VHDL-2002) und 2008 (VHDL-2008). Die Programmierumgebungen der PLD-Hersteller unterstützen unterschiedliche VHDL-Fassungen. Diamond 3.1 unterstützt VHDL-2008. In Anhang A3 sind die wesentlichen Eigenschaften von VHDL zusammengestellt.

Aus der Sicht von VHDL entspricht ein PLD einem Funktionsblock, der Bits mit der Umwelt austauscht (Abb. 5.2). Interne Variable und Konstante können unterschiedliche Datentypen aufweisen, z. B. Bit, Integer (ganze Zahlen), Float (Floatpointzahlen = Gleitkommazahlen), Felder, etc. Integer-Zahlen können nur positiv sein, positive und negative Werte annehmen, in einem bestimmten Zahlenbereich liegen oder sich nur aus einer bestimmten Anzahl Bits zusammensetzen. Felder können aus allen Datentypen gebildet werden. Für die Kommunikation mit der Umwelt müssen die externen Bit-Signale in die internen Datentypen umgewandelt werden und umgekehrt.

Mit den internen Variablen und Konstanten können alle erdenklichen mathematischen Operationen durchgeführt werden. Neben den Grundrechenarten (Addition, Subtraktion, Multiplikation, Division) sind auch logische Operationen (Negation, Und, Oder, Exor, Nand, Nor, Äquivalenz) und Modulo-Division möglich.

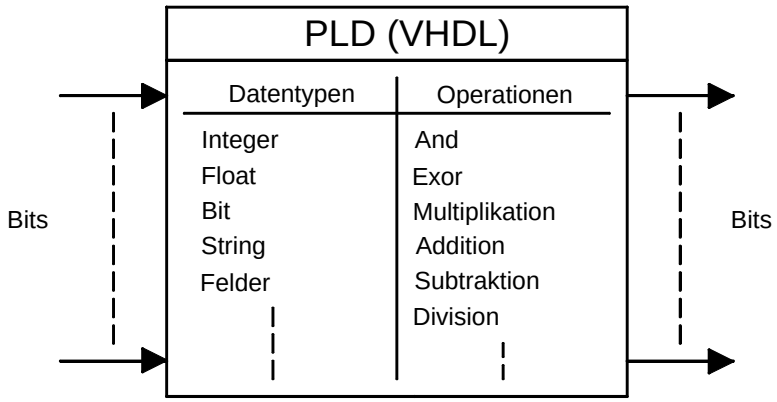


Abb. 5.2 Datenverarbeitung im PLD unter VHDL

Die interne Datenverarbeitung im PLD kann in mehrere Funktionsblöcke aufgegliedert werden, die miteinander Signale austauschen (Abb. 5.3). Die Definition der Funktionsblöcke und die Festlegung der Signalverknüpfungen können mit VHDL erfolgen. Es ist aber auch möglich, diese Eigenschaften mit einem Schaltplan-Editor festzulegen. Größere Funktionsblöcke können in kleinere Funktionsblöcke zerlegt werden. Die Funktionsweise eines Funktionsblocks kann mit VHDL oder mit Schaltplan-Editor definiert werden.

In den folgenden Kapiteln erfolgt eine schrittweise Einführung in die Arbeitsweise mit Diamond und VHDL. Zunächst werden verschiedene Aufgabenstellungen nur unter Verwendung von VHDL gelöst. Später wird neben VHDL auch der Schaltplan-Editor eingesetzt.

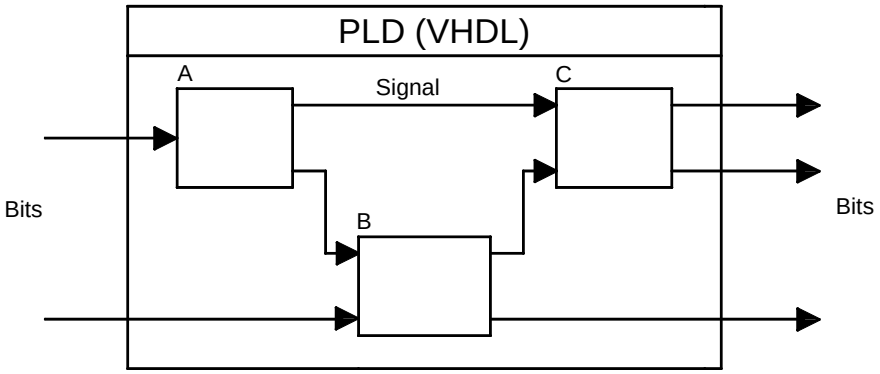


Abb. 5.3 VHDL-Funktionsblöcke und Signale

5.3 Das erste Design (NAND mit 2 Eingängen)

Als erste Aufgabe soll ein NAND-Gatter mit 2 Eingängen unter Verwendung von VHDL mit dem MACH Demoboard realisiert werden (Abb. 5.4). Die Eingangssignale A und B werden mit den Schaltern S1 und S2 erzeugt. Das Ausgangssignal y wird mit der LED D1 angezeigt.

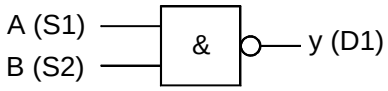


Abb. 5.4 NAND mit 2 Eingängen

Zwischen dem Ausgangssignal und den Eingangssignalen besteht der Zusammenhang:

$y = \overline{A \wedge B}$

Starten Sie Diamond und wählen Sie

File > New > Project

Im eingeblendeten Fenster New Project wählen Sie unter Project

Name: Nand2
Location: C:/Diamond/Projekte/Nand2

Das Projekt trägt den Namen Nand2 (NAND mit 2 Eingängen) und wird im angegebenen Verzeichnis abgelegt. Jedes Projekt muss ein eigenes Verzeichnis erhalten.

Die Eintragungen unter Implementation erfolgen automatisch. Nun wählen Sie

Next >

Im nächsten Fenster markieren Sie

√ Copy source to implementation directory

und wählen

Next >

Unter Select Device wird der PLD-Baustein ausgewählt:

Family:	MACH XO2
Performance grade:	4
Package type:	TQFP100
Operating conditions:	Commercial
Device:	LCMXO2-256HC

Unter Part name wird automatisch die Bestellnummer des ausgewählten Bausteins angezeigt. Wählen Sie

Next >

und markieren Sie

√ Lattice LSE

Hierdurch wählen Sie, dass als Synthesis Tool die Lattice Synthesis Engine (LSE) eingesetzt wird. Als Synthesis Tool bezeichnet man ein Programm, das das vom Anwender eingegebene Design (mit VHDL und/oder Schaltplan) in PLD-Strukturen umsetzt.

Wählen Sie

Next >

und

Finish

Abb. 5.5 zeigt den nun vorhandenen Diamond Bildschirm.

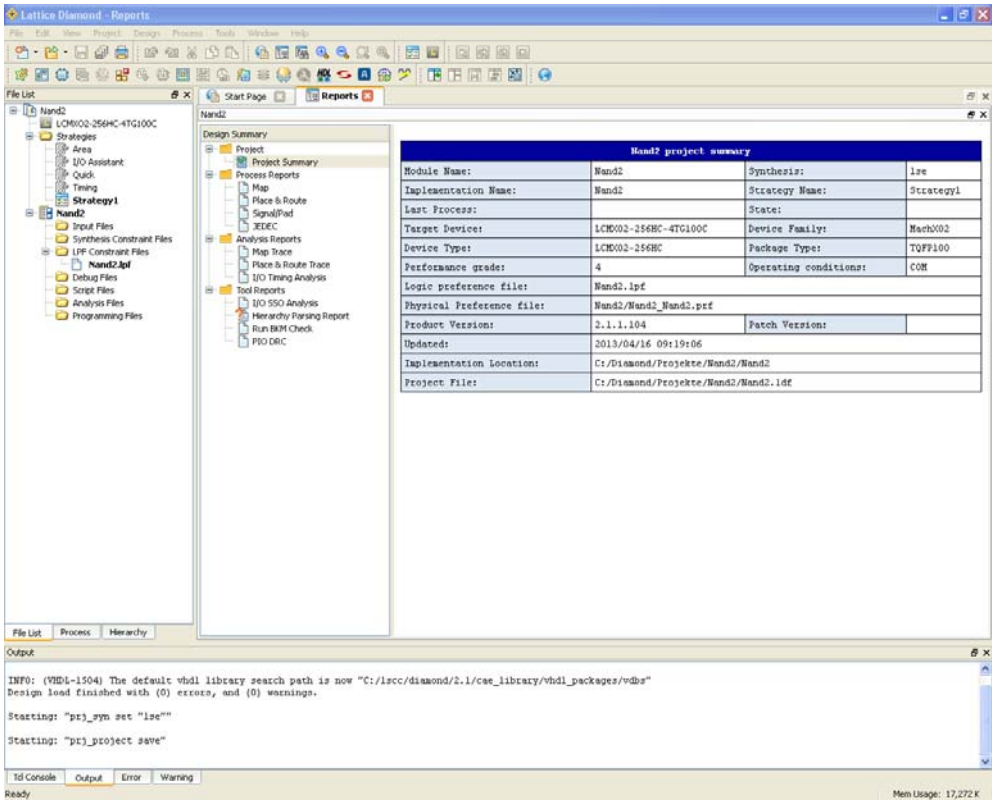


Abb. 5.5 Diamond Bildschirm nach Definition des Projekts NAND2

Das Fenster Start Page wurde durch das Fenster Reports ersetzt. Die Start Page ist aber weiterhin vorhanden und kann bei Bedarf angezeigt werden. Im Reports Fenster werden die wesentlichen Projektdaten dargestellt. Die Projektdaten werden in der Datei Nand2.ldf abgelegt (LDF = Lattice Definition File). Mit x kann ein Fenster geschlossen werden.

Im Fenster Design Summary werden die innerhalb des Projekts bereits durchgeführten Aktionen angezeigt. Durch Klicken auf die dort angebotenen Reports können diese geöffnet werden.

Unter File List werden die zum Projekt gehörenden Dateien aufgeführt. Dort findet sich die Datei Nand2.lpf (LPF = Lattice Project File). In dieser Datei werden die Constraints abgelegt. Hierunter versteht man u. a. die Zuordnung der Ein/Ausgänge des PLD zu den unter VHDL oder im Schaltplan gewählten Bezeichnungen.

Am oberen Rand des Fensters File List findet sich die Bezeichnung des ausgewählten PLD. Darunter werden verschiedene Strategies angeboten. Unter Strategy versteht man die Art und Weise wie ein Design in die Strukturen des PLD umgesetzt wird. Z. B. ist es möglich, weniger auf geringe Propagation Delay Times zu achten oder es können möglichst geringe Propagation Delay Times gefordert werden. Im Digitallabor wird stets Strategy1 verwendet.

Mit

File > Close Project

können Sie das aktuelle Projekt schließen. Das Projekt Nand2 rufen Sie wieder auf mit

File > Open > Project > Nand2.ldf

Öffnen

Alternativ können Sie das Projekt Nand2 öffnen durch Doppelklick auf Nand2 unter Recent Projects.

Nun wird die VHDL-Datei erzeugt. Öffnen Sie hierzu das Projekt Nand2 und wählen Sie

File > New > File

Im eingeblendeten Fenster machen Sie die folgenden Angaben:

Categories:	Source Files
Source Files:	VHDL Files
Name:	Nand2

Markieren Sie

√ Add to Implementation

und betätigen Sie

New

Abb. 5.6 zeigt den anschließend vorhandenen Diamond Bildschirm. Im rechten Fenster wird der Inhalt der Datei Nand2.vhd angezeigt. Da die Datei noch keinen Inhalt aufweist, ist der Bildschirm leer. Im Fenster File List wird unter Input Files die neu erzeugte VHDL-Datei Nand2.vhd aufgeführt. Durch Betätigung von x können Sie das Fenster Nand2.vhd schließen. Erneut öffnen können Sie das Fenster durch Doppelklick auf die Datei Nand2.vhd im Fenster File List.

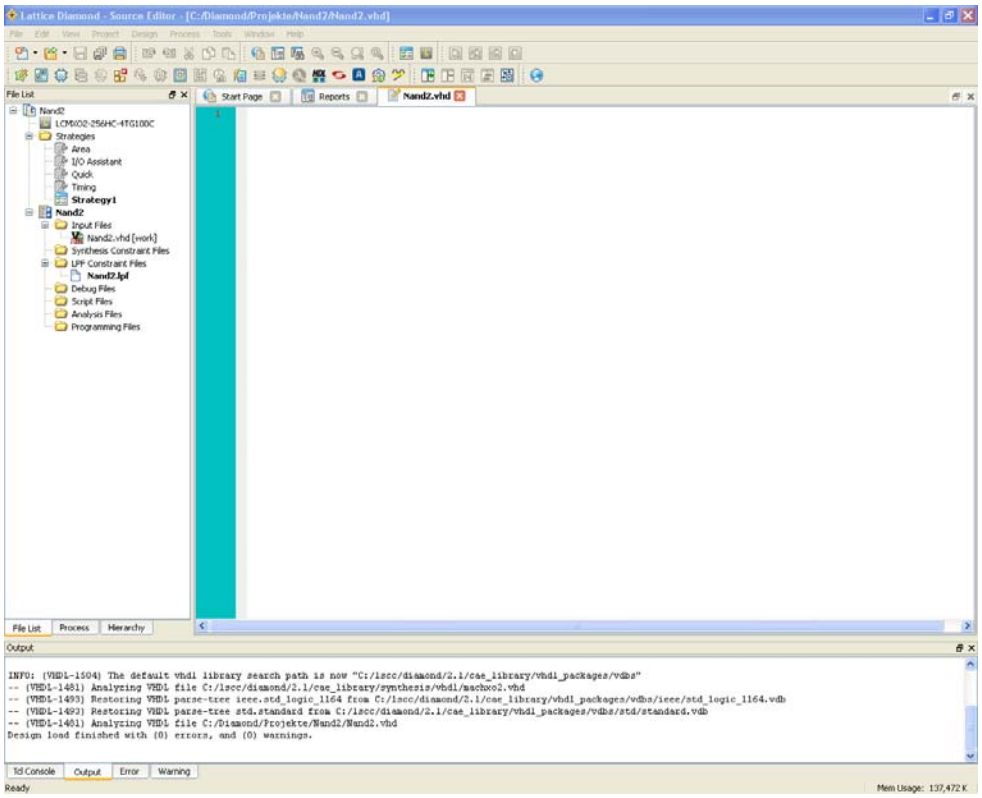


Abb. 5.6 Diamond Bildschirm nach Erstellen der ersten VHDL-Datei

Geben Sie nun das folgende VHDL-Programm ein.

```
-- ************************************
-- Projekt: Nand2
-- Autor:   Stoeckle
-- Datum:   16.4.2013
-- Beschreibung:
--
-- Nand mit den Eingaengen A und B und dem Ausgang y.
-- A: S1, B: S2, y: D1
--
-- Historie:
--
-- 16.4.2013   Stoeckle   1. Entwurf
-- ************************************

library ieee;
use ieee.std_logic_1164.all;

entity nand2 is
   port (
      A, B: in std_logic;    -- Eingaenge (S1, S2)
      y: out std_logic       -- Ausgang (D1)
      );
end entity nand2;
```

```
architecture behavior of nand2 is

begin

   main: process (A, B) is  -- main wird durchgefuehrt, wenn A oder B sich aendert
   begin
      y <= A nand B;
   end process main;

end architecture behavior;
```

Abb. 5.7. Programm Nand2

Das Programm enthält einen Programmkopf, in dem die Eigenschaften des Programms festgehalten werden. Der Programmkopf enthält Angaben über das Erstellungsdatum und den Autor des Programms. Ausserdem werden die Aufgaben des Programms und die Ein/Ausgänge beschrieben. Unter Historie wird angegeben, wie das Programm im Lauf der Zeit verändert wurde. Bei all diesen Angaben handelt es sich um Kommentar. Kommentare sind bei VHDL dadurch gekennzeichnet, dass sie mit den Zeichen `--` beginnen. Kommentare können in jeder Spalte beginnen. Erstrecken sich Kommentare über mehrere Zeilen, so muss jede Zeile mit den Zeichen `--` beginnen. Achten Sie darauf, dass im Kommentar keine Umlaute verwendet werden!

Im Anschluss an den Kommentar wird angegeben, welche Bibliothek benutzt werden soll, um die nachfolgende Syntax zu interpretieren.

Unter dem Schlüsselwort entity werden die Ein/Ausgänge definiert. Alle Ein/Ausgänge sollen vom Typ Bit bzw. Standard Logic sein. Es ist darauf zu achten, dass die letzte Zeile innerhalb von entity nicht mit einem Semikolon abgeschlossen wird. Der Grund hierfür liegt darin, dass die entity-Deklaration auch wie folgt durchgeführt werden kann:

```
port (A, B: in std_logic; y: out std_logic);
```

Die Schreibweise in Abb. 5.7 ist jedoch vorzuziehen, da die Deklaration übersichtlicher ist und Kommentare angegeben werden können.

Unter architecture wird angegeben, welche Aufgabe durchgeführt werden soll. In diesem Fall trägt architecture die Bezeichnung behavior. Innerhalb von architecture wird der Prozess process (A, B) deklariert. Er trägt die Bezeichnung main. process (A, B) wird nur aktiv, wenn A oder B sich ändert. Zur Deklaration der NAND-Verküpfung wird der Operator nand verwendet. Die Zuordnung zu einem Ausgang erfolgt mit den Zeichen <=.

Schließen Sie den Texteditor durch Eingabe von x und speichern Sie die Datei Nand2.vhd. Unmittelbar nach der Speicherung prüft Diamond die Syntax des eingegebenen Programms und gibt ggf. Fehlermeldungen im Output Fenster aus.

Um eine Fehlermeldung zu erzeugen, schreiben Sie die Deklaration des Ausgangs im obigen Programm wie folgt:

```
y: out std_logic;
```

Wie oben bereits erwähnt, darf die Zeile nicht mit einem Semikolon abgeschlossen werden. Schließen Sie den Editor mit x und beobachten Sie die Meldungen im Output Fenster. Nach Doppelklick auf die erste Fehlermeldung öffnet sich der Editor und im Bereich des Fehlers erfolgt eine Markierung (Abb. 5.8).

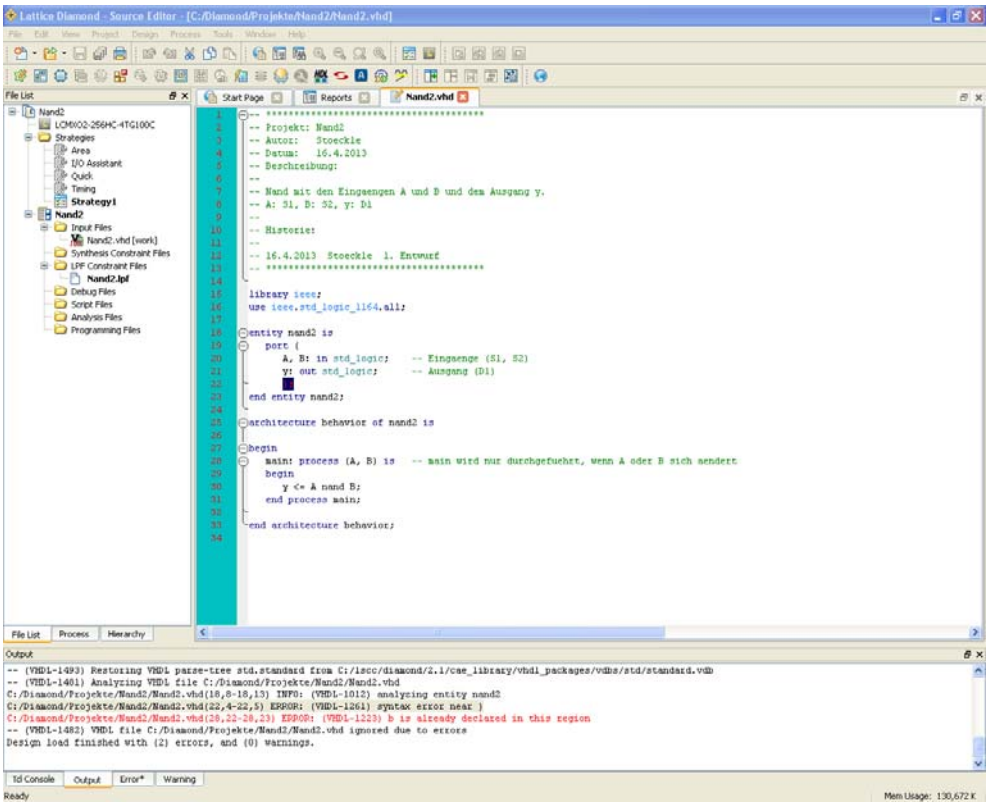


Abb. 5.8 Diamond Bildschirm mit Fehlermeldung

Entfernen Sie das Semikolon und schließen Sie die Datei nand2.vhd. Im Output Fenster dürfen nun keine Fehlermeldungen mehr vorhanden sein.

Wechseln Sie links in das Fenster Process. Führen Sie oben einen Doppelklick auf Synthesize Design aus. Im Fenster Reports werden Ihnen nun neue Reports angezeigt. Wählen Sie

Tools > HDL Diagram

oder betätigen Sie die HDL-Schaltfläche. Im rechten Fenster sehen Sie nun die Funktionsblöcke des Projekts Nand2 blau hinterlegt. Durch Betätigung von Run BKM Check (Haken) erfolgt eine Überprüfung des Projekts. Bei erfolgreicher Überprüfung erscheinen die Funktionsblöcke grün hinterlegt.

Führen Sie einen Rechtsklick auf den Block nand2 durch und wählen Sie View Entity Symbol. Das Symbol des Projekts Nand2 mit allen Ein/Ausgängen wird angezeigt. Führen Sie einen Rechtsklick auf den Block behavior durch und wählen Sie View Connectivity. Der Prozess main mit seinen Eingängen A und B und seinem Ausgang y wird angezeigt (Abb. 5.9).

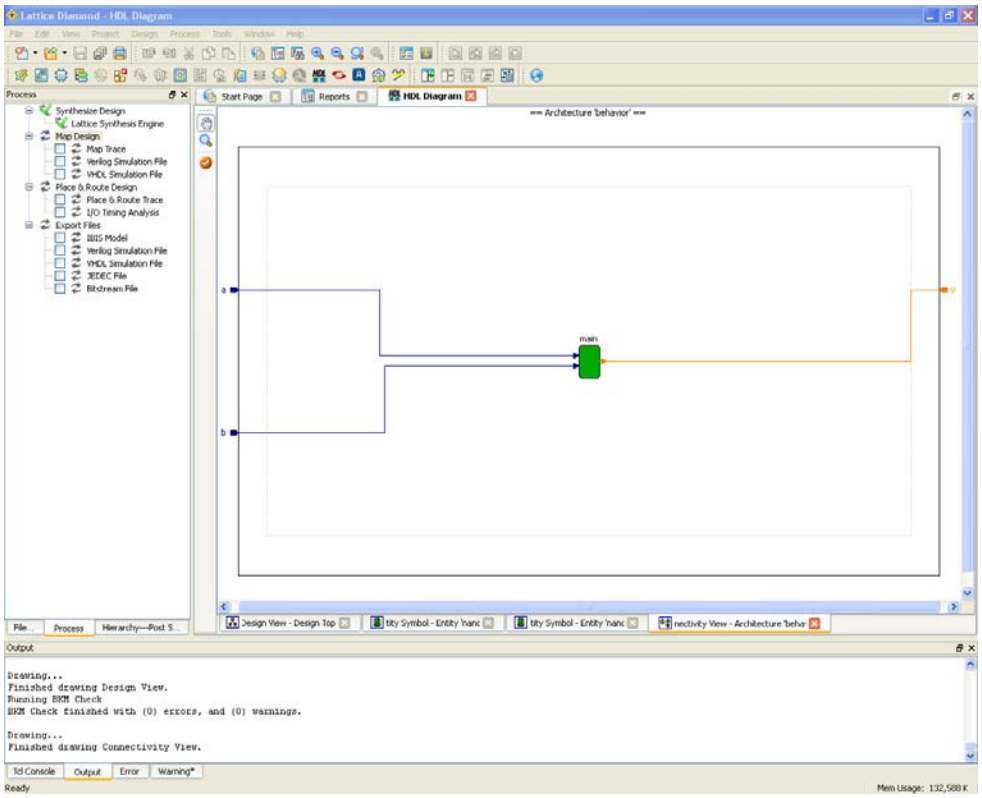


Abb. 5.9 Diamond Bildschirm nach View Connectivity

Offenbar werden die vorgegebenen Bezeichnungen mit den Großbuchstaben A und B ignoriert und die Signale werden mit kleinen Buchstaben markiert. Tatsächlich unterscheidet Diamond nicht zwischen Groß- und Kleinschreibung. Die Verwendung großer Buchstaben ist allerdings sinnvoll für eine übersichtlichere Programmgestaltung.

Wenn Sie auf der linken Seite das Fenster Hierarchy anzeigen, werden Sie an oberster Stelle der Hierarchie das Modul Nand2 finden. Wenn ein Projekt aus mehreren Modulen besteht, wird ihr hierarchischer Zusammenhang in diesem Fenster angezeigt. Es muss immer ein Top Level Module geben. Dieses Modul befindet sich auf der höchsten Hierarchieebene. Die Signale werden nach unten hierarchisch weitergegeben.

5.3.1 Simulation des Entwurfs

Betätigen Sie die Schaltfläche HDL Diagram und zeigen Sie die grün hinterlegten Funktionsblöcke an. Führen Sie einen Rechtsklick auf das Modul nand2 durch und wählen Sie VHDL Test Bench Template. Im Anschluss daran sehen Sie die in Abb. 5.10 wiedergegebene Bildschirmdarstellung.

Im File Fenster findet sich unter den Input Files von Projekt Nand2 nun auch die Datei nand2_tb.vhd. Hierbei handelt es sich um die automatisch erzeugte Test Workbench im VHDL-Code. Nach Doppelklick auf diese Datei wird der Inhalt der Test Workbench im Editorfenster dargestellt (Abb. 5.10).

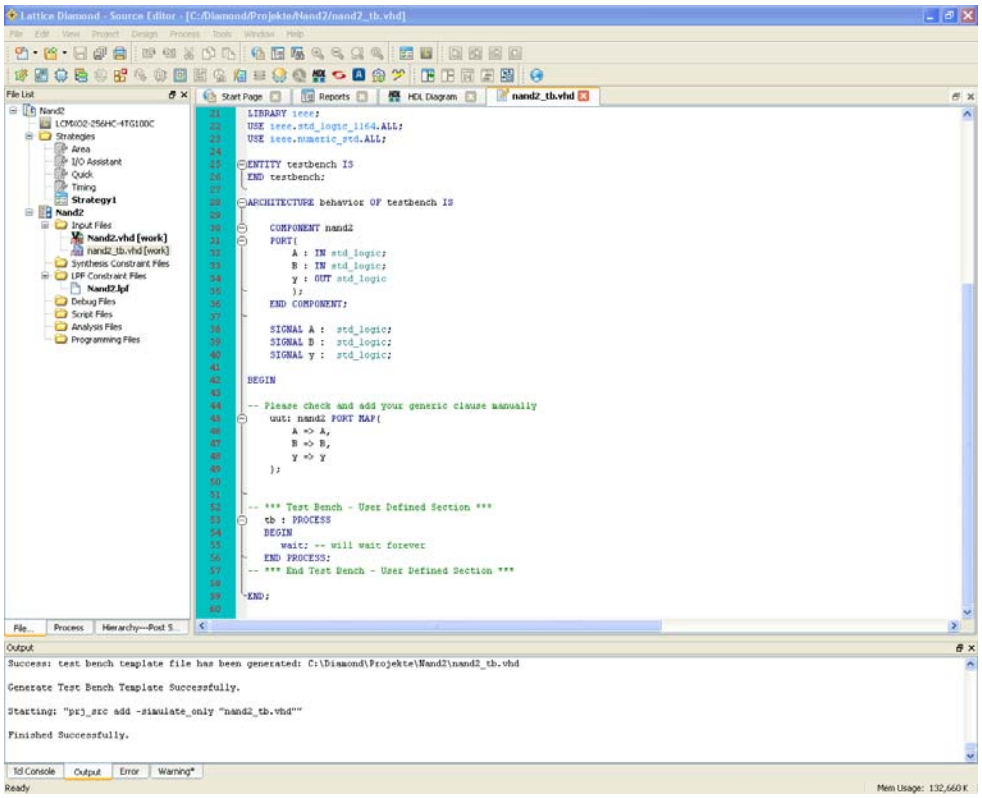


Abb. 5.10 Diamond Bildschirm nach Erzeugung der Test Workbench

Im unteren Bereich der Workbench-Datei befindet sich ein Bereich, der bearbeitet werden muss (User Defined Section). Der Prozess tb beinhaltet die Testsignale (Testvektoren), mit denen das Projekt Nand2 getestet werden soll. Ergänzen Sie den Prozess tb wie folgt:

```
tb: process
begin
   A <= '0';
   B <= '0';
   wait for 100 ns;
   A <= '0';
   B <= '1';
   wait for 100 ns;
   A <= '1';
   B <= '0';
   wait for 100 ns;
   A <= '1';
   B <= '1';
   wait;
end process;
```

Abb. 5.11 Definition der Testsignale (Testvektoren)

Auf diese Weise werden den Eingangssignalen A und B alle denkbaren Kombinationen zugeordnet, so dass die Funktion des Projekts Nand2 für alle diese Kombinationen überprüft werden kann. Der Befehl

```
wait for 100 ns;
```

bewirkt, dass bei der Simulation 100 ns gewartet wird. Es können auch andere Wartezeiten vorgegeben werden. Neben der Einheit ns sind auch us oder ms möglich. Der Wait-Befehl ist nur sinnvoll für Simulationen, da sich im PLD keine Hardware-Struktur befindet, die die Realisierung von Wartezeiten ermöglicht.

Der Befehl

```
wait;
```

bewirkt unendliches Warten.

Nach Eingabe der Testsignale schließen Sie die Testdatei. Im Output-Fenster dürfen keine Fehlermeldungen auftreten.

Zum Starten der Simulation wählen Sie

Tools > Simulation Wizard

Sie können auch die entsprechende Schaltfläche betätigen (Rechtecksignal mit Zauberstab). Im foglenden Fenster geben Sie ein:

Project name: Test

Sonst nehmen Sie keine Veränderungen vor. Betätigen Sie mehrfach Next und schließlich Finish. Im Anschluss daran wird der Simulator-Bildschirm gemäß Abb. 5.12 dargestellt.

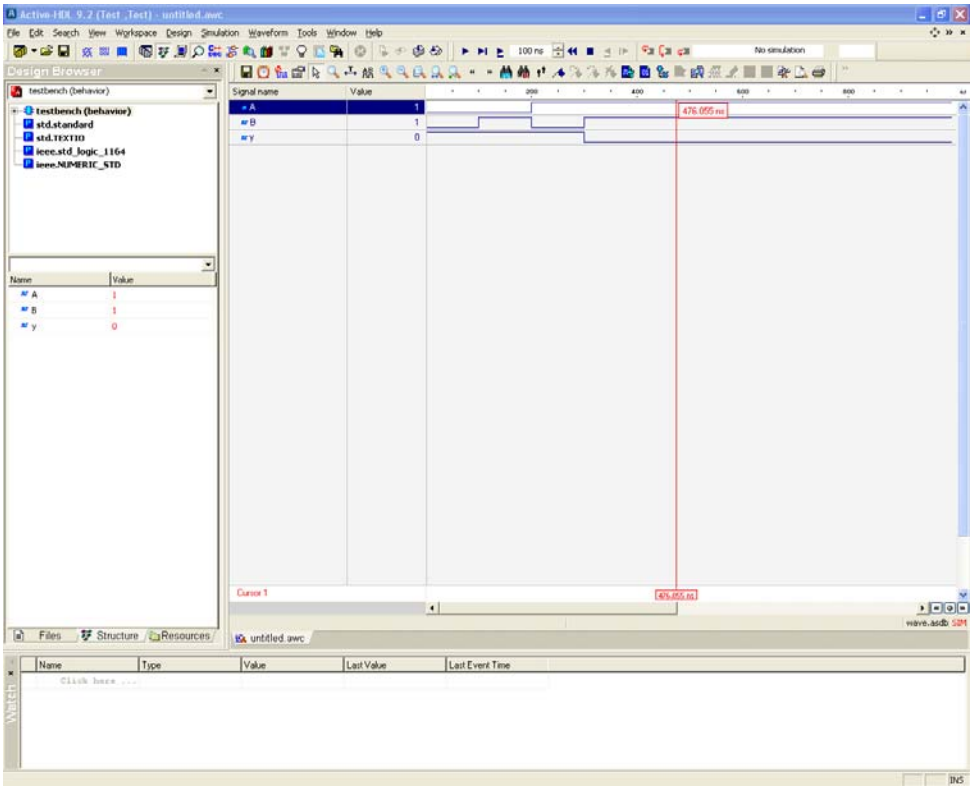


Abb. 5.12 Simulator-Bildschirm (Active-HDL)

Im Fenster rechts wird das Timing-Diagramm der Signale A, B und y dargestellt. Durch Klicken der linken Maustaste können Sie den Cursor setzen. Bei Festhalten der linken Maustaste können Sie den Cursor kontinuierlich hin- und herbewegen. Die Zeit wird am oberen und unteren Ende des Cursors eingeblendet. Als Einheit wird ps verwendet. Wenn eine andere Zeiteinheit gewünscht wird, kann dies durch einen Rechtsklick auf die untere Zeitanzeige erfolgen. Danach Time Unit auswählen und die gewünschte Zeiteinheit einstellen.

Aus den dargestellten Signalen entnimmt man das einwandfreie Verhalten entsprechend der gewünschten Nand-Funktion. Im Fenster links unten wird der Wert der Signale bei Beendigung der Simulation angezeigt.

Der aktuelle Wert der Signale A, B und y wird in der Spalte Value angezeigt. Betätigt man die rechte Maustaste im Timing-Fenster, so können weitere Funktionen gewählt werden (z. B. Zoom, weiterer Cursor, Signal finden). Zoom kann auch mit den entsprechenden Schaltflächen in der Bedienleiste über dem Timing-Diagramm ausgelöst werden (Zoom +, Zoom -, Zoom Fit).

Für eine spätere Auswertung der Simulation kann die Simulation gespeichert werden:

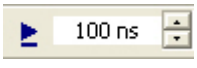
File > Save as …

Die Simulation wird beendet mit

File > Exit

Im Fenster oben rechts wird angezeigt, bis zu welchem Zeitpunkt die Simulation durchgeführt wurde, z. B. 1000 ns.

In Tab. 5.1 sind die Aufgaben einiger Schaltflächen zusammengestellt.

Schaltfläche	Funktion
	Restart Simulation. Die Simulation beginnt zum Zeitpunkt 0 ns.
	Simulation bis zum Ende durchführen. Alle Testvektoren abarbeiten.
	Run for 100 ns (Standard).
	Run until. Simulation bis zum eingegebenen Zeitpunkt durchführen. Nach Betätigung der Schaltfläche wird ein Fenster eingeblendet, in dem der Zeitpunkt eingegeben werden kann.
	End Simulation. Die Simulation wird beendet und das Timing-Diagramm wird gelöscht. Eine erneute Simulation ist nicht möglich. Schaltfläche nicht benutzen!

Tab. 5.1 Funktion wichtiger Schaltflächen während der Simulation

Durch Doppelklick im Timing-Fenster kann das Fähnchen mit der Zeitangabe am oberen Ende des Cursors ein- und ausgeschaltet werden.

5.3.2 Programmierung des PLD

Damit das Projekt in das PLD eingepasst werden kann, müssen die Pins für die Signale A, B und y festgelegt werden. Vorgabe ist, dass es sich hierbei um die Schalter S1 und S2 sowie das LED D1 handelt. Wählen Sie

Tools > Spreadsheet View

Eine Tabelle wird geöffnet, in der alle Signale eingetragen sind, die im aktuellen Projekt Verwendung finden. Alternativ können Sie das Spreadsheet auch durch Betätigen der entsprechenden Schaltfläche (ganz links, Stern mit Stab) öffnen.

Wählen Sie Port Assignments und klicken Sie mit der rechten Maustaste auf IO_TYPE in der Zeile All Ports. Wählen Sie LVCMOS33. Alle vorhandenen Signale weisen nun die Definition LVCMOS33 auf.

Wählen Sie Pin Assignments. Aus Tab. 3.1 entnehmen Sie, dass der Schalter S1 auf Pin Nr. 1 bzw. auf den Anschluss PL2A (Pad Name) des PLD geführt ist. Klicken Sie unter Pin 1 mit der rechten Maustaste auf das Feld Signal Name und wählen Sie das Signal A aus (Assign Signals). Schalter S2 ist gemäß Tab. 3.1 auf Pin Nr. 2 bzw. auf den Anschluss PL2B des PLD geführt. Klicken Sie unter Pin 2 mit der rechten Maustaste auf das Feld Signal Name und wählen Sie das Signal B aus (Assign Signals). Die LED D1 ist gemäß Tab. 3.1 auf Pin Nr. 16 bzw. auf den Anschluss PL5A geführt. Klicken Sie unter Pin 16 mit der rechten Maustaste auf das Feld Signal Name und wählen Sie das Signal y aus (Assign Signals). Nun sind alle im VHDL-Programm definierten Signale Anschlüssen des PLD zugeordnet. Schließen Sie das Spreadsheet und bestätigen Sie, dass die Änderungen in der Datei Nand2.lpf gesichert werden.

Nun kann das Projekt in das PLD eingepasst werden. Hierzu führen Sie im Fenster Process einen Doppelklick auf MAP Design aus. Anschließend führen Sie im Fenster Process einen Doppelklick auf Place & Route Design aus. Aus dem zugehörigen Report können Sie entnehmen, dass zur Realisierung des Projekts 3 von 56 PIOs (5%) und 1 von 128 Slices (< 1%) benutzt werden.

Zur Programmierung des PLD wird eine JEDEC-Datei benötigt. Markieren Sie im Fenster Process unter Export Files das JEDEC File und führen Sie anschließend einen Doppelklick auf Export Files aus.

Jede erfolgreich durchgeführte Aktion wird im Fenster Process und im Fenster Design Summary durch einen grünen Haken markiert.

Vor der eigentlichen Programmierung des PLD über die USB-Schnittstelle soll noch ein Blick auf die physikalische Umsetzung des Designs im PLD erfolgen. Wählen Sie

Tools > Floorplan View

Es folgt die in Abb. 5.13 wiedergegebene Bildschirmdarstellung. Man erkennt alle 128 Slices, über die das PLD verfügt. Benutzte Slices (1) sind blau markiert.

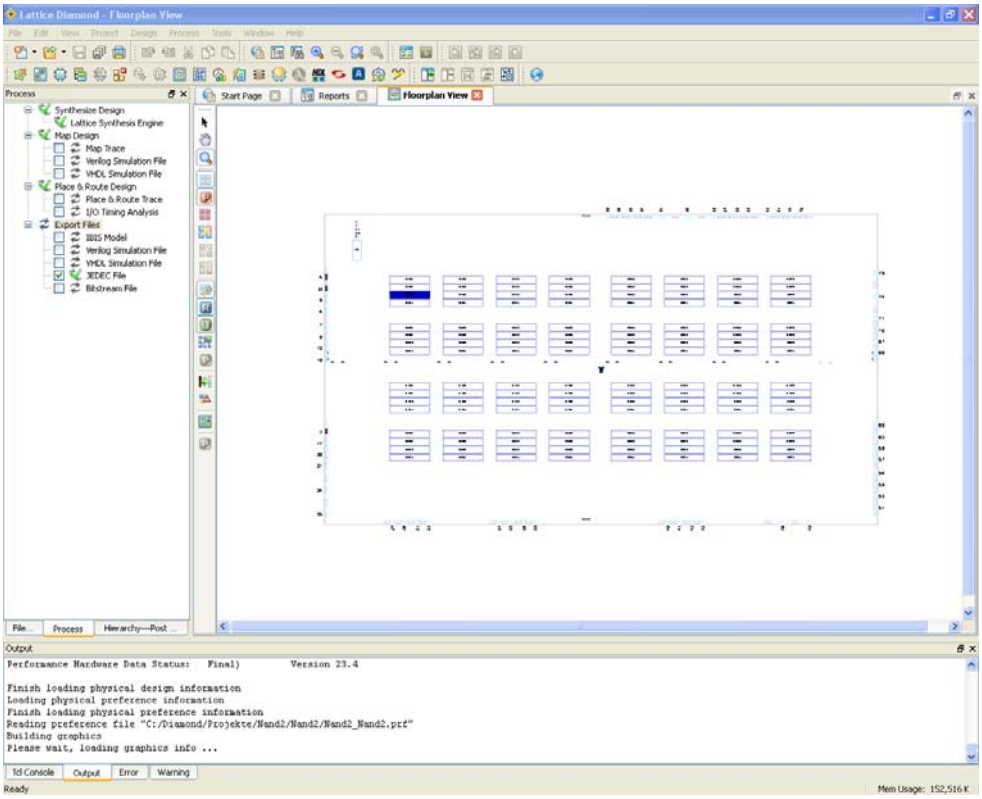


Abb. 5.13 Floorplan View

Wählen Sie

Tools > Physical View

Es folgt die in Abb. 5.14 wiedergegebene Bildschirmdarstellung, aus der sich die Verbindungen innerhalb des PLD und zu den Ein/Ausgängen ergeben.

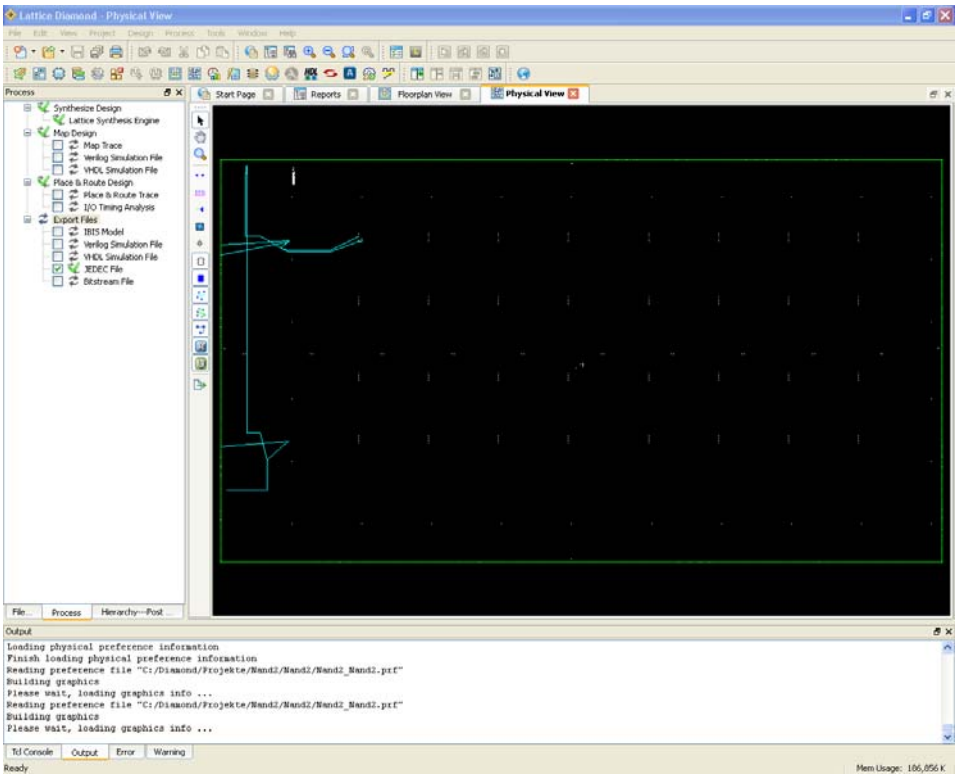


Abb. 5.14 Physical View

Um das PLD auf dem MACH Demoboard zu programmieren, verbinden Sie nun das Demoboard mit Hilfe eines USB-Kabels mit einer USB-Schnittstelle des Computers. Unmittelbar nach dem Anschluss des Demoboards hören Sie ein akustisches Signal, mit dem die korrekte Funktion der Kommunikation mit dem Board bestätigt wird. Die Spannungsanzeige am oberen Rand des Demoboards (grüne LED) muss aufleuchten.

Zur Programmierung wählen Sie

Tools > Programmer

Alternativ können Sie den Programmiervorgang auch durch Betätigung der entsprechenden Schaltfläche starten (IC mit rosa Pfeil). In dem eingeblendeten Fenster führen Sie keine Änderungen durch und betätigen die Taste OK. Im Programmer Fenster wählen Sie die Schaltfläche Programmieren (IC mit grünem Pfeil). Das PLD wird programmiert. Die erfolgreiche Programmierung wird signalisiert durch die grün markierte Fläche PASS.

Testen Sie die Nand-Funktion mit dem Demoboard.

Achten Sie beim Programmieren darauf, dass im rechten Fenster unter "Cable-Settings" die folgenden Einstellungen angegeben sind:

Cable:	HW-USBN-2B (FTDI)
Port:	FTUSB-0
Use default I/O settings:	√

Im mittleren Fenster muss die zu programmierende Datei vom Typ *.JED ausgewählt sein.

5.4 Schaltnetz mit 3 NAND (NAND32)

Das zu erstellende Schaltnetz soll aus 3 Nands mit jeweils 2 Eingängen bestehen, die gemäß Abb. 5.15 verknüpft sind. Als Basis wird das Nand mit 2 Eingängen verwendet, das in Abschnitt 5.3 erstellt wurde (NAND2).

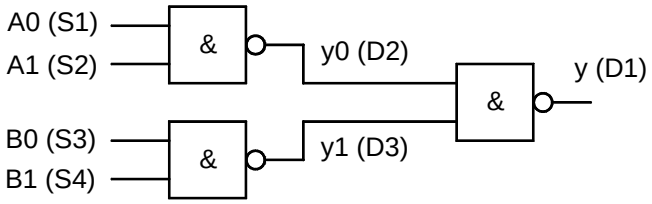


Abb. 5.15 Schaltnetz mit 3 NAND2 (NAND32)

Die zugehörige Wahrheitstabelle ist in Tab. 5.2 dargestellt. Mit den Schaltern S1, S2, S3 und S4 werden die Eingangsvariablen A0, A1, B0, B1 erzeugt. Die Ausgangsvariable y wird mit der LED D1 angezeigt. Die Variablen y0 und y1 werden mit D2 und D3 angezeigt.

A0	A1	B0	B1	y0	y1	y
0	0	0	0	1	1	0
0	0	0	1	1	1	0
0	0	1	0	1	1	0
0	0	1	1	1	0	1
0	1	0	0	1	1	0
0	1	0	1	1	1	0
0	1	1	0	1	1	0
0	1	1	1	1	0	1
1	0	0	0	1	1	0
1	0	0	1	1	1	0
1	0	1	0	1	1	0
1	0	1	1	1	0	1
1	1	0	0	0	1	1
1	1	0	1	0	1	1
1	1	1	0	0	1	1
1	1	1	1	0	0	1

Tab. 5.2 Wahrheitstabelle NAND32

Erzeugen Sie zunächst ein neues VHDL-Projekt mit dem Namen NAND32 im Verzeichnis C:/Diamond/Projekte/Nand32. Erstellen Sie eine VHDL-Datei mit der Bezeichnung NAND32.vhd. Geben Sie das nebenstehende VHDL-Programm ein.

```
-- ***************************************************
-- Projekt: NAND32
-- Autor:   Stoeckle
-- Datum:   17.4.2013
-- Beschreibung:
--
-- 3 NANDs mit jeweils 2 Eingaengen werden verbunden.
-- 1. NAND (Gate0): A0, A1 -> y0
-- 2. NAND (Gate1): B0, B1 -> y1
-- 3. NAND (Gate2): y0, y1 -> y
```

```
--
-- A0, A1: S1, S2
-- B0, B1: S3, S4
-- y: D1
-- y0: D2
-- y1: D3
--
-- Historie:
--
-- 17.4.2013  Stoeckle  1. Entwurf
-- ************************************************

library ieee;
use ieee.std_logic_1164.all;
use work.all;

entity nand32 is
   port (
      A0, A1, B0, B1: in std_logic;
      y, y0, y1: out std_logic
      );
end entity nand32;

architecture struct of nand32 is

signal y0_sig: std_logic;
signal y1_sig: std_logic;

begin
   Gate0: entity work.nand2(behavior)
          port map (A0, A1, y0_sig);
   Gate1: entity work.nand2(behavior)
          port map (B0, B1, y1_sig);
   Gate2: entity work.nand2(behavior)
          port map (y0_sig, y1_sig, y);

   output: process (y0_sig, y1_sig) is
   begin
      y0 <= y0_sig;
      y1 <= y1_sig;
   end process output;

end architecture struct;
```

Abb. 5.16 Programm Nand32

Mit der Befehlszeile

```
use work.all;
```

wird festgelegt, dass alle Programme, die sich im aktuellen Verzeichnis befinden, verwendet werden können.

Im Anschluss daran erfolgt die entity-Deklaration mit den Ein/Ausgängen. In der architecture struct werden 3 Instanzen der Funktion NAND2 erstellt (Gate0, Gate1, Gate2). Die Ein/Ausgänge dieser Instanzen werden nach port map angegeben.

Die Signale zwischen den Instanzen müssen mit dem Schlüsselwort signal deklariert werden. Als Signale werden verwendet y0_sig und y1_sig. Ein/Ausgänge können in der port map Liste verwendet werden. Zur Ausgabe von y0 und y1 dient der Prozess output. Hier erfolgt die Zuordnung der Signale zu den Ausgängen.

Damit Sie das Programm nicht immer neu eingeben müssen, können Sie Programmteile aus einem vorhandenen Programm kopieren:

Markieren mit linker Maustaste > rechte Maustaste > Copy

Einfügen erfolgt mit:

rechte Maustaste > Paste

Speichern Sie nun die Datei nand32.vhd. Im Output Fenster werden 3 Warnings angezeigt, da die Funktion nand2 nicht vorhanden ist.

Kopieren Sie nun die Datei nand2.vhd in das Verzeichnis C:/Diamond/Projekte/Nand32. Bevor diese Datei innerhalb des Projekts Nand32 benutzt werden kann, muss sie zum Projekt hinzugefügt werden. Dies erfolgt mit

File > Add > Existing File

und Auswahl der Datei nand2.vhd. Öffnen Sie nand2.vhd und schließen Sie die Datei wieder. Fehlermeldungen dürfen nicht auftreten. Nach Öffnen und Schließen von nand32.vhd dürfen keine Warnings mehr auftreten.

Führen Sie nun die folgenden Schritte aus:

Synthesize Design
HDL Diagram
Run BKM Check

Im HDL Diagram sehen Sie die 3 Blöcke nand32, struct und nand2. Rechtsklick auf struct und Wahl von Connectivity führt zur Anzeige des in Abb. 5.17 dargestellten Schaltplans.

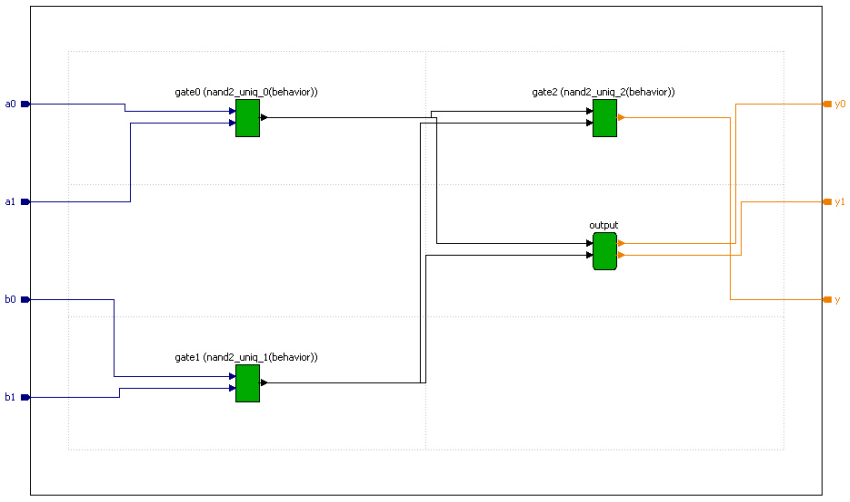


Abb. 5.17 Schaltplan von Nand32

Im Schaltplan erkennt man die 3 Instanzen Gate0, Gate1 und Gate2 von Nand2 und den Prozess output zur Erzeugung der Ausgänge aus den internen Verbindungssignalen.

Tragen Sie nun im Spreadsheet die Pins des PLD ein und wählen Sie:

Map Design
Place & Route Design
Export File (JEDEC)

Programmieren Sie das PLD im Demoboard und überprüfen Sie die Funktionsweise.

5.5 Code-Umsetzer (Decoder)

Zu realisieren ist der in Abb. 5.18 dargestellte Decoder, der 4 Bit Dualzahlen in 4 Bit Zahlen im Graycode umsetzt. Die zugehörige Wahrheitstabelle ist in Tab. 5.3 angegeben.

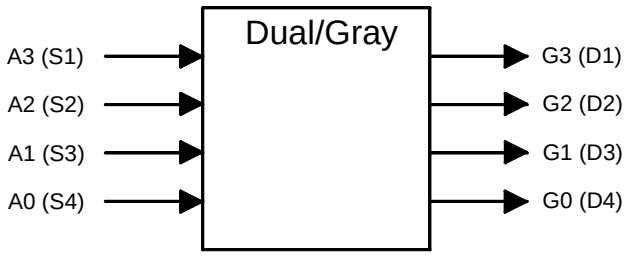


Abb. 5.18 Decoder zur Umsetzung vom Dualcode in den Graycode

A3	A2	A1	A0	G3	G2	G1	G0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	1
0	1	1	0	0	1	0	1
0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1
1	0	1	0	1	1	1	1
1	0	1	1	1	1	1	0
1	1	0	0	1	0	1	0
1	1	0	1	1	0	1	1
1	1	1	0	1	0	0	1
1	1	1	1	1	0	0	0

Tab. 5.3 Wahrheitstabelle für die Umsetzung vom Dualcode in den Graycode

Die Schaltfunktionen G0, G1, G2 und G3 können durch logische Verknüpfungen aus den Eingangsvariablen gebildet werden. VHDL bietet jedoch die Möglichkeit zu einer kompakteren Schreibweise auf der Basis von Feldern. Der VHDL-Code lautet in diesem Fall:

```
entity decoder is
   port (
      A: in std_logic_vector (3 downto 0);
      G: out std_logic_vector (3 downto 0)
      );
end entity decoder;


architecture behavior of decoder is
begin
   dec: process (A) is
   begin
      case A is
         when "0000" => G <= "0000";
         when "0001" => G <= "0001";
         when "0010" => G <= "0011";
         when "0011" => G <= "0010";
         when "0100" => G <= "0110";
         when "0101" => G <= "0111";
         when "0110" => G <= "0101";
         when "0111" => G <= "0100";
         when "1000" => G <= "1100";
         when "1001" => G <= "1101";
         when "1010" => G <= "1111";
         when "1011" => G <= "1110";
         when "1100" => G <= "1010";
         when "1101" => G <= "1011";
         when "1110" => G <= "1001";
         when "1111" => G <= "1000";
         when others => null;
      end case;
   end process dec;
end architecture behavior;
```

Abb. 5.19 VHDL-Programm für den Decoder

Die Definition des Felds A mit downto bewirkt, dass die Bits in der Reihenfolge A(3), A(2), A(1), A(0) von links nach rechts gelesen werden. In der case-Anweisung ist others erforderlich. Der Fall others tritt nie ein. Ohne Angabe von others erfolgt jedoch eine Fehlermeldung. Dem Fall others wird der Do Nothing-Befehl null zugewiesen.

Erzeugen Sie ein neues Projekt mit dem Namen GRAYDEC und erstellen Sie eine VHDL-Datei mit demselben Namen. Geben Sie das Programm ein, programmieren Sie das Demoboard und überprüfen Sie die Funktionsweise des Decoders.

Aus dem Place & Route Report entnimmt man, dass zur Realisierung der Aufgabe 8 von 56 PIOs (14%) und 2 von 128 Slices (1%) verwendet wurden.

5.6 Vergleicher (Komparator)

Zu realisieren ist ein Vergleicher für die beiden positiven 4 Bit-Zahlen A und B gemäß Abb. 5.20.

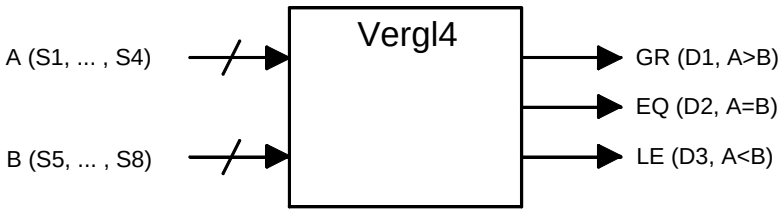


Abb. 5.20 Vergleicher für zwei positive 4 Bit-Zahlen

A wird mit den Schaltern S1, … , S4 erzeugt. Zur Eingabe von B werden die Schalter S5, … , S8 verwendet. Das Ergebnis des Vergleichs wird durch die Ausgangssignale GR, EQ und LE signalisiert. Es ist jeweils nur eines dieser Signale aktiv. Wenn A > B, ist GR = 1. Im Fall A < B ist LE = 1. Wenn A = B, ist EQ = 1.

Eine Wahrheitstabelle ist in diesem Fall nicht sinnvoll, da bei 8 Eingangsvariablen insgesamt 256 Zeilen erforderlich wären. Damit ein Zahlenvergleich in VHDL erfolgen kann, müssen die mit den Schaltern erzeugten Bits in Integer-Zahlen umgewandelt werden. Diese Integer-Zahlen haben einen Bereich von 0 … 15. Insgesamt also 16 Möglichkeiten. Es ist sinnvoll, dafür einen spezifischen Datentyp zu deklarieren, nämlich Int16. Diamond überprüft dann den Entwurf auf die Einhaltung dieses Zahlenbereichs und erzeugt Fehlermeldungen bei Bereichsverletzungen.

Die Umsetzung der eingegebenen Bitvektoren A bzw. B in Integer-Zahlen erfolgt durch sukzessive Multiplikation mit der Zahl 2. Nach 4 Schritten liegt die Integer-Zahl vor. Dieses Vorgehen erinnert an das sukzessive Abarbeiten von Befehlen innerhalb eines Computers. Dies ist aber nicht der Fall. Bei VHDL dient diese Darstellung lediglich zur maschinenlesbaren Eingabe der Aufgabenstellung. Die Realisierung des Vergleichers erfolgt mit einem Schaltnetz. Der VHDL Code ist in Abb. 5.21 dargestellt.

```
entity vergl4 is
   port (
      A, B: in std_logic_vector (3 downto 0);
      GR, EQ, LE: out std_logic
      );
end entity vergl4;


architecture behavior of vergl4 is

begin
   verg: process (A, B) is
   type int16 is range 0 to 15;   -- Typ fuer eingeschraenkte Int-Variable
   variable a_int: int16;
   variable b_int: int16;
   variable i: int16;
   variable sum: int16;

   begin

      -- Wandlung von A in a_int

      sum := 0;
      for i in 3 downto 0 loop    -- Schleife
         if A(i) = '0' then
```

```
            sum := sum * 2;
         else
            sum := sum * 2 + 1;
         end if;
      end loop;
      a_int := sum;

      -- Wandlung von B in b_int

      sum := 0;
      for i in 3 downto 0 loop    -- Schleife
         if B(i) = '0' then
            sum := sum * 2;
         else
            sum := sum * 2 + 1;
         end if;
      end loop;
      b_int := sum;

      -- Vergleich

      if a_int > b_int then
         GR <= '1';
         EQ <= '0';
         LE <= '0';
      end if;

      if a_int < b_int then
         GR <= '0';
         EQ <= '0';
         LE <= '1';
      end if;

      if a_int = b_int then
         GR <= '0';
         EQ <= '1';
         LE <= '0';
      end if;

   end process verg;

end architecture behavior;
```

Abb. 5.21 VHDL-Programm für den Vergleich von zwei positiven 4 Bit-Zahlen

Die Wandlung der Bitvektoren in Integer-Zahlen erscheint umständlich. Einfacher wäre eine Formulierung etwa in folgender Form:

```
      for i in 3 downto 0 loop    -- Schleife
         sum := sum * 2 + A(i);
      end loop;
```

Dies wäre jedoch nicht zulässig, da sum vom Datentyp int16 ist und A(i) vom Datentyp std_logic.

Die Variablen sind nur innerhalb des Prozesses verg definiert und können deshalb auch nur innerhalb dieses Prozesses verwendet werden.

Erzeugen Sie ein neues Projekt mit dem Namen VERGL4 und erstellen Sie eine VHDL-Datei mit demselben Namen. Geben Sie das Programm ein. Erzeugen Sie eine VHDL Testbench und ergänzen Sie den Prozess tb wie folgt:

```
tb: process
begin
   A <= "0000";   -- EQ = 1
   B <= "0000";
```

```
   wait for 100 ns;
   A <= "0001";    -- LE = 1
   B <= "0011";
   wait for 100 ns;
   A <= "0100";    -- GR = 1
   B <= "0011";
   wait for 100 ns;
   A <= "1110";    -- LE = 1
   B <= "1111";
   wait;
end process;
```

Abb. 5.22 Definition der Testsignale (Testvektoren)

Starten Sie den Simulator und überprüfen Sie die Funktion. Programmieren Sie das Demoboard und überprüfen Sie die Funktionsweise des Vergleichers am Demoboard.

Aus dem Place & Route Report entnimmt man, dass zur Realisierung der Aufgabe 11 von 56 PIOs (19%) und 4 von 128 Slices (3%) verwendet wurden.

Um den Vergleich negativer Zahlen zu ermöglichen, muss die Umwandlung der Bitvektoren in die Integer-Zahlen modifiziert werden. Außerdem sind die Datentypen der Integer-Zahlen anzupassen. Abb. 5.23 zeigt das VHDL-Programm in diesem Fall. Die entity-Deklaration wird nicht verändert. Negative Zahlen werden im Zweierkomplement dargestellt. Bei 4 Bit umfasst der Zahlenbereich -8 … 7, d. h. 16 Kombinationen. Innerhalb des Programms sind Variable vorhanden, die den Wert +15 annehmen können (sum). Die Variable sum wiederum wird an die Variablen a_int bzw. b_int übergeben. Dies ist nur möglich bei identischen Datentypen. Deshalb wird der gemeinsame Datentyp int24 verwendet, der den Bereich -8 … 15 überstreicht, d. h. 24 Werte (inkl. 0).

```
architecture behavior of vergl4k is

begin
   verg: process (A, B) is
   type int24 is range -8 to 15;   -- Typ fuer eingeschraenkte Int-Variable
   variable a_int: int24;
   variable b_int: int24;
   variable i: int24;
   variable sum: int24;

   begin

      -- Wandlung von A in a_int

      sum := 0;
      for i in 3 downto 0 loop    -- Schleife
         if A(i) = '0' then
            sum := sum * 2;
         else
            sum := sum * 2 + 1;
         end if;
      end loop;
      if A(3) = '0' then          -- positive Zahlen
         a_int := sum;
      else
         a_int := sum - 16;       -- negative Zahlen
      end if;

      -- Wandlung von B in b_int

      sum := 0;
      for i in 3 downto 0 loop    -- Schleife
         if B(i) = '0' then
            sum := sum * 2;
```

```
         else
            sum := sum * 2 + 1;
         end if;
      end loop;
      if B(3) = '0' then         -- positive Zahlen
         b_int := sum;
      else
         b_int := sum - 16;      -- negative Zahlen
      end if;

      -- Vergleich

      if a_int > b_int then
         GR <= '1';
         EQ <= '0';
         LE <= '0';
      end if;

      if a_int < b_int then
         GR <= '0';
         EQ <= '0';
         LE <= '1';
      end if;

      if a_int = b_int then
         GR <= '0';
         EQ <= '1';
         LE <= '0';
      end if;

   end process verg;

end architecture behavior;
```

Abb. 5.23 VHDL-Programm für den Vergleich positiver und negativer 4 Bit-Zahlen

Erzeugen Sie ein neues Projekt mit dem Namen VERGL4K und erstellen Sie eine VHDL-Datei mit demselben Namen. Geben Sie das Programm ein. Erzeugen Sie eine VHDL Testbench und ergänzen Sie den Prozess tb wie folgt:

```
tb: process
begin
   A <= "0000";    -- EQ = 1
   B <= "0000";
   wait for 100 ns;
   A <= "0001";    -- LE = 1
   B <= "0011";
   wait for 100 ns;
   A <= "0100";    -- GR = 1
   B <= "0011";
   wait for 100 ns;
   A <= "1110";    -- LE = 1
   B <= "1111";
   wait for 100 ns;
   A <= "0010";    -- GR = 1
   B <= "1010";
   wait;
end process;
```

Abb. 5.24 Testbench für den Vergleich positiver und negativer Zahlen

Starten Sie den Simulator und überprüfen Sie die Funktion. Programmieren Sie das Demoboard und überprüfen Sie die Funktionsweise des Vergleichers am Demoboard.

Aus dem Place & Route Report entnimmt man, dass zur Realisierung der Aufgabe 11 von 56 PIOs (19%) und 4 von 128 Slices (3%) verwendet wurden.

5.7 Verwendung des Numeric Standard Package

Oft kommt es vor, dass die Ein- und Ausgänge eines Projekts Bitvektoren vom Typ std_logic_vector sind. Diese Bitvektoren sind in vielen Fällen als positive Dualzahlen oder als vorzeichenbehaftete Zahlen im Zweierkomplement-Format zu interpretieren. Damit unter VHDL Zahlenoperationen durchgeführt werden können, müssen die Bitvektoren in Zahlen umgewandelt werden. In Kapitel 5.6 wurde gezeigt, wie die Typumwandlung durchgeführt werden kann. Komfortabler kann die Typumwandlung erfolgen, wenn das Numeric Standard Package eingesetzt wird. Es handelt sich hierbei um eine VHDL-Funktionsbibliothek, die speziell auf diese Aufgabenstellungen zugeschnitten ist (siehe Kapitel A3.16). Um das Numeric Standard Package zu verwenden, muss am Anfang des Programms die folgende Deklaration erfolgen:

```
library ieee;
use ieee.numeric_std.all;
```

Durch Verwendung des Numeric Standard Package können die Typumwandlungen von und in Bitvektoren kürzer und effizienter gestaltet werden.

5.8 Archivieren von Diamond-Projekten

Am Beispiel des Projekts NAND2 wird im Folgenden die Archivierung eines Projekts beschrieben. Das Projekt wird archiviert mit

File > Archive Project

Datei: Nand2.zip

Save

Um das archivierte Projekt auf einem anderen PC bearbeiten zu können, ist dort zunächst ein entsprechendes Verzeichnis zu erstellen, z. B. C:\Diamond\Projekte\Nand2. In dieses neue Verzeichnis wird die Datei Nand2.zip kopiert. Danach Diamond starten und Wahl von

File > Open > Archived Projects …

und Nand2.zip auswählen.

Es kann sein, dass nicht alle Prozesse als durchgeführt gekennzeichnet sind (grüner Pfeil fehlt). Dann die Schritte Synthesize, Map and Route etc. erneut durchführen. Beim Programmieren der JEDEC-Datei kann es vorkommen, dass diese nicht gefunden wird. Innerhalb des Programmers dann den richtigen Pfad für die JEDEC-Datei festlegen.

Achtung! Es ist nicht sinnvoll, ein Projekt von einem Computer auf einen anderen Computer zu übertragen, indem der komplette Inhalt des Projektverzeichnisses kopiert wird. In den Dateien sind Informationen über die Installation des Computers enthalten. Jeder Computer verfügt über eine andere Installation, so dass eine korrekte Funktion von Diamond mit einem auf diese Weise übertragenen Projekt nicht sichergestellt ist.

6 Schaltplaneingabe (Schematic Entry)

Ein Projekt kann mit Hilfe eines Schaltplans definiert werden. Es ist auch möglich, lediglich Teile eines Projekts mit einem Schaltplan zu definieren. Die übrigen Teile des Projekts werden dann mit einem VHDL-Programm bestimmt.

Zur Erstellung des Schaltplans verfügt Diamond über einen Schaltplan-Editor (Schematic Editor). Der Schaltplan beinhaltet allgemein bekannte Funktionsbausteine der Digitaltechnik, wie z. B. Nand, Und, Oder, Negation, Multiplexer, Komparator, Flipflops, etc., die im Schaltplan miteinander verbunden werden. Eigene Funktionsbausteine können erstellt werden. Die Funktion der Bausteine wird durch ein VHDL-Programm definiert oder kann wiederum mit Hilfe eines Schaltplans bestimmt werden. Eine Bibliothek mit Standard-Funktionsbausteinen ist vorhanden.

6.1 Das Projekt Nand2S

Das aus Kapitel 5.3 bekannte Projekt Nand2 (Nand mit 2 Eingängen) soll mit Hilfe eines Schaltplans erstellt werden.

Erstellen Sie ein neues Projekt mit dem Namen Nand2S und erzeugen Sie einen Schaltplan durch Auswahl von

File > New > File … > Schematic Files

Wählen Sie den Namen Nand2S.sch für den Schaltplan. Danach wird die Zeichenfläche für den Schaltplan angezeigt. Schließen Sie den Schaltplan mit x. Unter File List wird nun die Datei Nand2S.sch aufgeführt. Durch Doppelklick auf diese Datei wird der Schaltplan-Editor gestartet und die Zeichenfläche wird angezeigt.

Auf der linken Seite der Zeichenfläche befindet sich eine Leiste mit Symboltasten zur Ausführung der Zeichenaufgaben. In Tab. 6.1 sind die Bezeichnungen dieser Symboltasten und die zugeordneten Aufgaben zusammengestellt.

Mit der Pfeiltaste (Select) werden Objekte ausgewählt, bevor sie z. B. verschoben werden können. Nach Auswahl eines Objekts werden nach Betätigung der rechten Maustaste weitere Aktionen angeboten, z. B. Löschen des Objekts (Clear).

Ein Objekt kann auch gelöscht werden, indem es mit der Pfeiltaste markiert wird und die Taste Del (Entf) betätigt wird.

Mit der Pan-Taste kann die Zeichenfläche verschoben werden: Nach Auswahl von Pan die linke Maustaste festhalten und verschieben.

Mit der Zoom-Taste wird der anzuzeigende Bereich definiert. Nach Betätigung der rechten Maustaste innerhalb der Zeichenfläche werden weitere Zoom-Möglichkeiten angeboten, z. B. Zoom Fit.

Einige Zeichenfunktionen werden durch Doppelklick beendet (z. B. Wire). Ein Abbruch ist mit ESC möglich.

Die Verwendung eines Gitters (Grid) hat sich bewährt. Zum Einschalten des Gitters wählen Sie

Tools > Options > Schematic Editor > Graphic

√ Show Grid

Symboltaste	Bezeichnung	Aufgabe
	Select	Auswahl eines Objekts
	Pan	Verschieben der Zeichenfläche
	Zoom Area	Auswahl des darstellbaren Bereichs mit einem Rechteckfenster.
	Wire	Zeichnen einer Verbindung (Net)
	Net Name	Anzeige des Namens einer Verbindung oder Vergabe eines Namens.
	Bus Tap	Bus-Anschluss wählen
	IO Port	Eingang oder Ausgang definieren
	Symbol	Symbol einfügen
	Instance Name	Name einer Instanz vergeben
	Arc	Bogen zeichnen
	Circle	Kreis zeichnen
	Line	Linie zeichnen
	Rectangle	Rechteck zeichnen
	Text	Text eingeben
	Highlight	Suchen einer Verbindung (Net)
	Move	Objekt verschieben
	Drag	Ziehen

Tab. 6.1 Symboltasten des Schematic Editor

In Kapitel 5.3 wurde das VHDL-Programms eines Nands mit 2 Eingängen erstellt. Dieses Nand soll nun in den Schaltplan aufgenommen werden. Hierzu muss ein Funktionsblock des VHDL-Programms erstellt werden. Die Ein- und Ausgänge dieses Funktionsblocks wurden innerhalb der Entity-Deklaration des VHDL-Programms festgelegt.

Um den benötigten Funktionsblock zu erzeugen, schließen Sie das aktuelle Projekt und öffnen das Projekt Nand2. Betätigen Sie die HDL-Taste. Führen Sie Run BKM Check (Haken) durch. Auf dem oberen Funktionsblock Nand2 drücken Sie die rechte Maustaste und wählen Sie Generate Schematic Symbol. Das Symbol des Funktionsblocks Nand2 wird in der Datei Nand2.sym im Verzeichnis des Projekts Nand2 abgelegt.

Schließen Sie das Projekt Nand2. Kopieren Sie die Dateien Nand2.sym und Nand2.vhd des Projekts Nand2 in das Verzeichnis des Projekts Nand2S. Öffnen Sie das Projekt Nand2S. Übernehmen Sie die Datei Nand2.vhd in das Projekt durch Wahl von

File > Add > Existing File

und Auswahl von Nand2.vhd.

Starten Sie den Schematic Editor durch Doppelklick auf Nand2S.sch. Nach Betätigung der Taste Symbol werden Bibliotheken und Verzeichnisse angeboten, in denen sich Symbole befinden. Im aktuellen Verzeichnis [local] befindet sich das Symbol Nand2. Klicken Sie auf dieses Symbol und übernehmen Sie das Symbol in den Schaltplan.

Die Eingänge und der Ausgang des Nand-Gatters müssen mit einer kurzen Verbindungsleitung versehen werden. Verwenden Sie hierzu die Taste Wire.

Nun werden die Verbindungsleitungen bezeichnet. Wählen Sie die Taste Net Name und geben Sie für den ersten Eingang den Namen S1 ein. Klicken Sie auf das linke Ende der Verbindungsleitung zum ersten Eingang. Der Name wird platziert. Für den zweiten Eingang wählen Sie den Namen S2 und für den Ausgang die Bezeichnung D1.

Betätigen Sie nun die Taste IO Port und wählen Sie Input. Markieren Sie den Eingang S1. Der Eingang S1 verfügt nun über einen Eingangspfeil. Genauso verfahren Sie mit dem Eingang S2. Beim Ausgang D1 wählen Sie nach Betätigung der Taste IO Port Output und markieren Sie D1. Der Ausgang D1 erhält einen Ausgangspfeil. Wenn mehrere Eingänge bzw. Ausgänge vorhanden sind, können sie gleichzeitig markiert und mit einem Ein- bzw. Ausgangspfeil versehen werden.

Betätigen Sie die Taste Instance Name und geben Sie als Namen für die erstellte Instanz des Nand-Gatter Nand0 ein. Klicken Sie mit der linken Maustaste auf das Nand-Gatter. Die Bezeichnung wird im Schaltplan eingetragen. In Abb. 6.1 ist der nun vorhandene Schaltplan wiedergegeben.

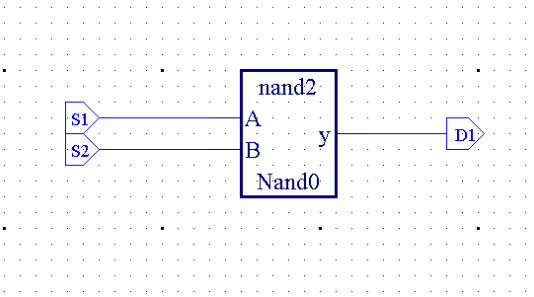


Abb. 6.1 Schaltplan des Nand-Gatters

Schließen Sie den Schaltplan Editor und führen Sie den Prozess Synthesize aus. Tragen Sie im Spreadsheet die Pins ein. Erzeugen Sie die JEDEC-Datei. Programmieren Sie das PLD des Demoboards und erproben Sie die Funktion der Schaltung mit dem Demoboard.

Ist eine Schaltplanseite für einen komplexeren Schaltplan nicht ausreichend, so können weitere Schaltplanseiten (Sheets) erstellt werden. Wählen Sie hierzu

Edit > Sheet > New

und geben Sie unter Sheet Num die Seitennummer der neuen Schaltplanseite ein. Die Standardgröße einer Seite beträgt 559 x 432 mm (ANSI C: 22 x 17 inch). Diese Größe ist für die meisten Anwendungen optimal.

Durch Eingabe von

Edit > Sheet > Delete

können Seiten gelöscht werden.

6.2 Das Projekt Nand3S

Das Projekt Nand3 aus Kapitel 5.4 soll nun auch mit Hilfe eines Schaltplans erstellt werden. Erzeugen Sie das Projekt Nand3S und erstellen Sie eine Schaltplan-Datei mit demselben Namen. Gehen Sie vor wie im vorigen Abschnitt beschrieben und platzieren Sie im Schaltplan drei Instanzen des Funktionsblocks Nand2. Bezeichnen Sie die Instanzen mit Nand0, Nand1 und Nand2. Wählen Sie als Eingänge die Schalter S1, S2, S3 und S4. Der Ausgang ist wiederum die LED D1. In Abb. 6.2 ist der Schaltplan wiedergegeben.

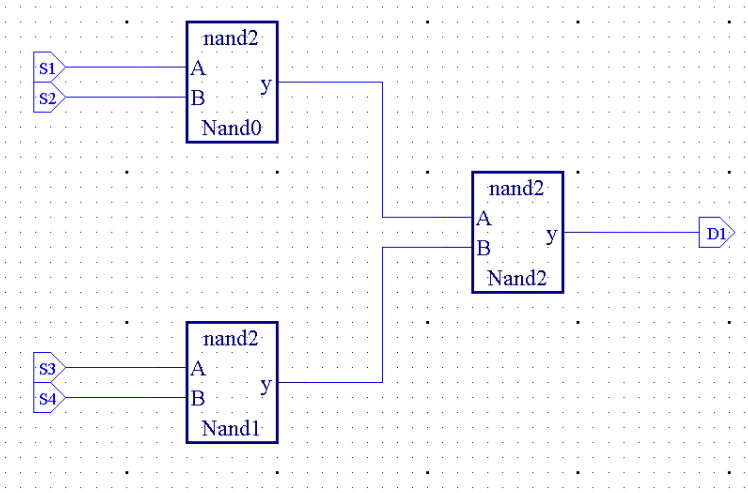


Abb. 6.2 Schaltplan mit 3 Nand-Gattern

Schließen Sie den Schaltplan-Editor. Führen Sie den Prozess Synthesize durch und geben Sie im Spreadsheet die Pins des PLD an. Erzeugen Sie die JEDEC-Datei. Programmieren Sie das PLD des Demoboards und erproben Sie die Schaltung mit dem Demoboard.

Bisher wurden als Nand-Funktionsblöcke die selbst erstellten Dateien Nand2.vhd und Nand2.sym verwendet. Alternativ können vorgefertigte Funktionsblöcke verwendet werden, die sich in der Bibliothek lattice.lib befinden. Um dies zu erproben, öffnen Sie den Schaltplan Editor mit dem Schaltplan Nand3S.sch und löschen Sie die vorhandene Schaltung. Betätigen Sie die Taste Symbol und wählen Sie aus der Bibliothek das Nand-Gatter mit 2 Eingängen (nd2). Platzieren Sie 3 Instanzen des Nand-Gatters auf dem Schaltplan. Erstellen Sie die Verbindungen zwischen den Nand-Gattern, bezeichnen Sie die Ein/Ausgänge und benennen Sie die Instanzen wie im vorigen Beispiel. Abb. 6.3 zeigt den neuen Schaltplan.

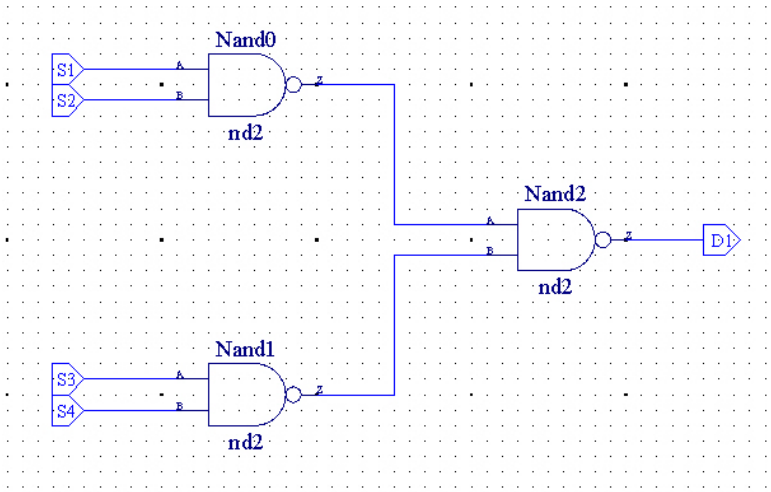


Abb. 6.3 Schaltplan mit den Nand-Gattern aus der Bibliothek lattice.lib

Schließen Sie den Schaltplan Editor. Führen Sie den Prozess Synthesize durch. Machen Sie die Eintragungen im Spreadsheet. Erstellen Sie die JEDEC-Datei. Programmieren Sie das PLD des Demoboards und erproben Sie die Schaltung mit dem Demoboard.

Wird ein Signal an einer anderen Stelle innerhalb eines Schaltplans benötigt und kann keine Verbindungsleitung zwischen den beiden Punkten hergestellt werden, z. B. weil die Verbindung zu lang wäre und der Schaltplan dadurch unübersichtlich würde oder weil sich das Signal über mehrere Seiten des Schaltplans erstreckt, so werden die zu verbindenden Leitungsstücke mit demselben Namen bezeichnet. Diamond erkennt dies und berücksichtigt die Verbindung. Anhand des Beispiels aus Abb. 6.3 wird dies nun gezeigt. Die Verbindungen zwischen den Ausgängen der Gatter Nand0 sowie Nand1 und den Eingängen von Gatter Nand3 werden nicht gezeichnet. Der Ausgang von Gatter Nand0 wird mit A bezeichnet. Der Ausgang von Gatter Nand1 erhält die Bezeichnung B. Die Eingänge von Gatter Nand2 tragen die Bezeichnungen A und B. Damit sind die gewünschten Verbindungen hergestellt. In Abb. 6.4 ist der modifzierte Schaltplan wiedergegeben.

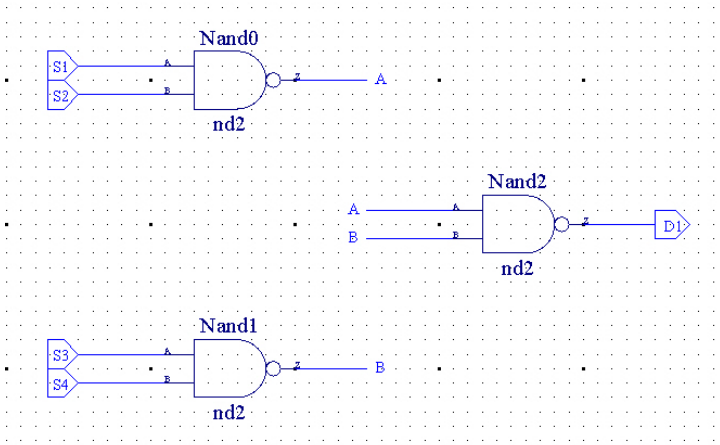


Abb. 6.4 Schaltplan aus Abb. 6.3 mit nicht gezeichneten internen Verbindungen A und B

6.3 Manuelle Erstellung von Symbolen

In Kapitel 6.1 wurde aus dem VHDL-Programm Nand2.vhd automatisch die Symboldatei Nand2.sym erstellt. Ein Symbol kann auch mit dem Symbol Editor erstellt werden. Es handelt sich hierbei um ein Grafikprogramm, das ähnliche Funktionen bietet wie der Schematic Editor. Um den Symbol Editor zu erproben, wird das Projekt Nand3S verwendet. Löschen Sie zunächst den vorhandenen Schaltplaninhalt. Löschen Sie die Datei Nand2.sym aus dem Projektverzeichnis von Nand3S. Die Datei Nand2.vhd verbleibt im Projektverzeichnis. Sie beschreibt die Funktionalität des Nands. Für dieses Nand wird nun manuell ein Symbol gestaltet.

Erzeugen Sie eine neue Symboldatei durch Wahl von

File > New > File … > Other Files > Symbol Files

und Eingabe des Dateinamens Nand2.sym.

Im Anschluss daran öffnet sich die Zeichenfläche des Symbol Editors. Schließen Sie den Symbol Editor mit x. Die Datei Nand2.sym wird in der File List nicht aufgeführt. Um den Symbol Editor für diese Datei neu zu starten, wählen Sie

File > Open > File …

und zeigen Sie alle Dateien *.sym an. Durch Öffnen von Nand2.sym wird der Symbol Editor gestartet.

Die Verwendung eines Gitters (Grid) hat sich bewährt. Um das Gitter einzuschalten, wählen Sie

Tools > Options > Symbol Editor > Graphic

√ Show Grid

Öffnen Sie den Symbol Editor mit der Datei Nand2.sym. Zeichnen Sie als Symbol für das Nand-Gatter ein Rechteck (Taste Rectangle). Im Bereich des Ausgangs zeichnen Sie einen Kreis (Taste Big Bubble). Wählen Sie die Taste Pin und positionieren Sie die Pins für die beiden Eingänge und den Pin für den Ausgang des Nand-Gatters. Verbinden Sie die Pins mit dem Gattersymbol durch Linien (Taste Line). Das Zeichnen der Linien beenden Sie mit ESC.

Markieren Sie den ersten Eingangspin (Taste Select). Führen Sie einen Rechtsklick durch und wählen Sie Object Properties aus. Klicken Sie auf + links von Pins und dann auf die punktierte Linie darunter.

Machen Sie die folgenden Angaben:

Pin Name A
Polarity INPUT

Gehen Sie entsprechend vor beim zweiten Eingang und beim Ausgang. Beim Ausgang ist dem Parameter Polarity der Wert OUTPUT zuzuordnen.

Mit Hilfe der Taste Text bringen Sie im Inneren des Rechtecks noch das fehlende Symbol & an. Schließen Sie den Symbol Editor.

Öffnen Sie den Schematic Editor durch Doppelklick auf die Datei Nand3S.sch. Erzeugen Sie mit der Taste Symbol 3 Instanzen des selbst definierten Nand-Gatters. Verbinden Sie die Instanzen gemäß Kapitel 6.2 und fügen Sie die Eingänge S1 und S2 sowie den Ausgang D1 hinzu.

Probleme:

Die Namen der Instanzen können nicht vergeben werden. Wie werden Symbole definiert, damit die Namen der Instanzen angezeigt werden können?

Beim automatisch erzeugten Symbol sind die Texte <Type> und <InstName> enthalten. Wie wirkt sich dies aus?

Beim automatisch erzeugten Symbol hat das Rechteck eine größere Linienbreite. Wie wird dies definiert?

Empfehlung: Bis zur Klärung der Probleme Symbole automatisch aus einem VHDL-Programm erzeugen. Mit dem Symbol Editor ggf. Veränderungen der automatisch erzeugten Symbole vornehmen.

6.4 Verwendung von Busverbindungen

Innerhalb einer Busverbindung werden mehrere Signale zusammengefasst. Auf diese Weise ist eine übersichtliche Gestaltung von Schaltplänen möglich. Das Erstellen von Busverbindungen soll anhand eines Beispiels erläutert werden. Es ist ein Schaltnetz zu erstellen, das die folgende Schaltfunktion realisiert:

$$y = \overline{A0} \wedge \overline{A1} \wedge \overline{A2} \wedge \overline{A3} \wedge \overline{A4} \wedge \overline{A5}$$

Die Variablen A0, … , A5 werden mit den Schaltern S1, … , S6 des Demoboards erzeugt. Es gelten die Zuordnungen S1 = A5, S2 = A4, … , S6 = A0. Die Ausgangsvariable y wird mit der LED D1 zur Anzeige gebracht.

Erstellen Sie ein neues Projekt mit dem Namen Bus und einen Schaltplan mit demselben Namen. Starten Sie den Schematic Editor. Platzieren Sie auf der linken Seite 6 Negierer untereinander aus der Bibliothek lattice.lib (Element inv). Versehen Sie die Ein- und Ausgänge der Negierer mit kurzen Leitungsstücken. Bezeichnen Sie die Eingangsleitungen von oben nach unten mit A0, … , A5. Ordnen Sie die Bezeichnungen jeweils links von den Leitungen an. Bringen Sie die Pfeilmarkierungen für die Eingänge an. Bezeichnen Sie die Ausgangsleitungen der Negierer von oben nach unten mit B0, … , B5. Ordnen Sie die Bezeichnungen oberhalb oder unterhalb der Leitungen an.

Platzieren Sie mit etwas Abstand rechts neben den Negierern zwei Und-Gatter mit jeweils 3 Eingängen aus der Bibliothek lattice.lib (Element and3). Rechts von den beiden Und-Gattern ordnen Sie ein Und-Gatter mit zwei Eingängen aus der Bilbliothek lattice.lib an (Element and2). Verbinden Sie die Ausgänge der Und-Gatter mit 3 Eingängen mit den beiden Eingängen des Und-Gatters mit zwei Eingängen. Bringen Sie am Ausgang des Und-Gatters mit zwei Eingängen eine kurze Leitung an, die Sie mit y bezeichnen. Bringen Sie die Pfeilmarkierung für den Ausgang y an.

Es fehlen noch die Verbindungen zwischen den Ausgängen der Negierer und den Eingängen der Und-Gatter mit den 3 Eingängen. Zeichnen Sie eine Leitung (Taste Wire), die neben dem Ausgang B0 beginnt, an den anderen Negiererausgängen vorbeiführt und an den Eingängen der Und-Gatter mit den 3 Eingängen endet (siehe Abb. 6.4). Diese Leitung soll in eine Busverbindung überführt werden. Wählen Sie die Taste Bus Tap. Betätigen Sie die linke Maustaste im Bereich der Leitung B0 auf dem Bus und halten Sie die Maustaste fest. Verbinden Sie den entstandenen Busanschluss mit dem Signal B0. Zur Kennzeichnung der Busverbindung hat der Bus nun eine stärkere Linienbreite. Führen Sie die Signale B1, … , B5 ebenfalls auf den Bus.

Mit der Taste Net Name können Sie den Bus mit einem Namen versehen. Wählen Sie die Bezeichnung InvBus.

Klicken Sie mit der rechten Maustaste auf den Bus und wählen Sie Object Properties. Die Bussignale werden angezeigt.

Zum Anschluss der Eingänge der Und-Gatter an den Bus wählen Sie die Taste Bus Tap. Betätigen Sie auf der Höhe des obersten Eingangs die linke Maustaste auf dem Bus und ziehen Sie bei gedrückter Maustaste die Verbindungslinie bis zum Eingang des Und-Gatters. Benennen Sie die Leitung mit Hilfe der Taste Net Name mit dem Bussignal B0. Schließen Sie alle übrigen Eingänge der Und-Gatter an den Bus an. In Abb. 6.4 ist der entstandene Schaltplan wiedergegeben.

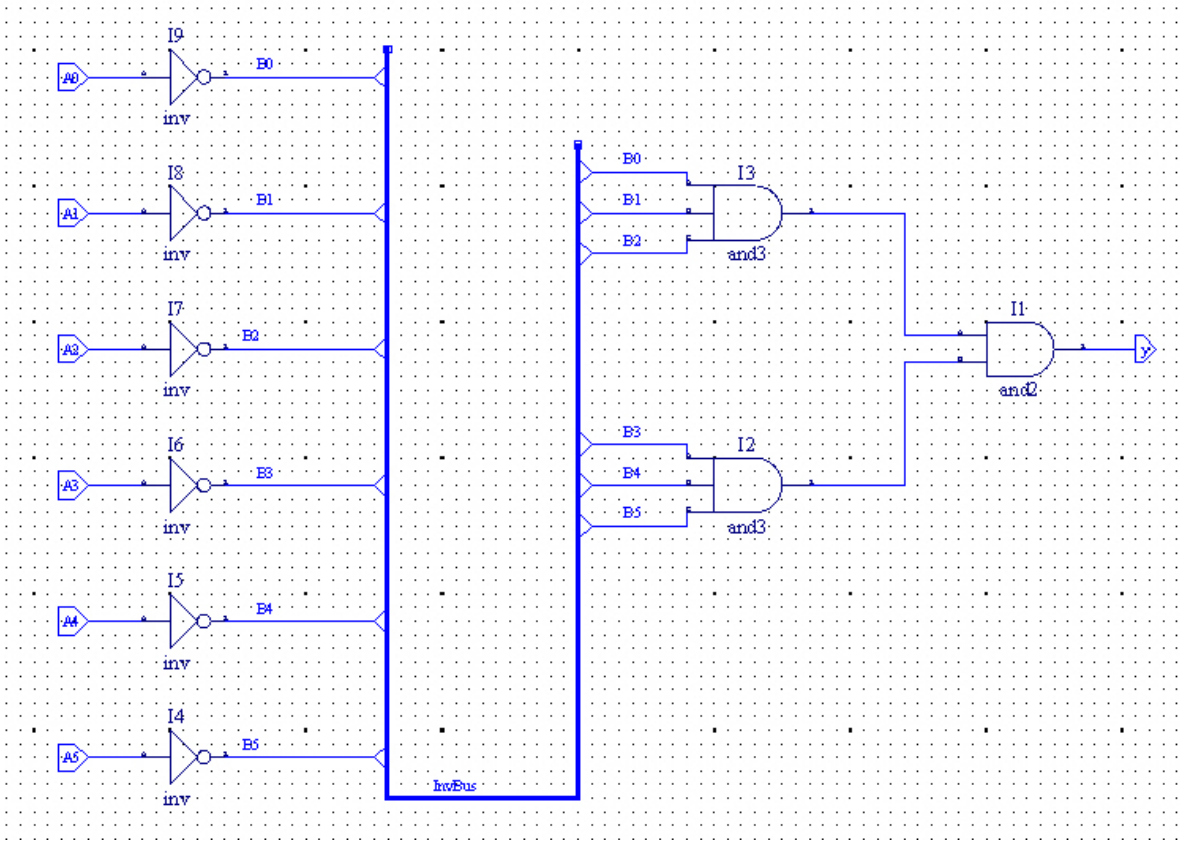


Abb. 6.4 Schaltplan mit Busverbindung

Zum Verfolgen einer Leitung auf dem Schaltplan dient die Taste Highlight (Glühbirne). Markieren Sie die zu prüfende Leitung an einer beliebigen Stelle. Das betreffende Leitungsstück wird rot

gekennzeichnet. Nach Betätigung der Taste Highlight wird der gesamte Verlauf der ausgewählten Leitung auf dem Schaltplan rot gekennzeichnet. Die Kennzeichnung wird gelöscht durch Klicken auf die linke Maustaste in einem leeren Schaltplanbereich.

Wird für eine Leitung ein bestimmter Name gewählt, so wird diese Leitung automatisch zum Bus. Ein Beispiel für einen solchen Namen ist die Bezeichnung A[3:0]. Wird einer Leitung dieser Name zugeordnet, so wird diese Leitung zum Bus mit den Signalen A[0], A[1], A[2] und A[3]. Die Signale, die mit diesem Bus verbunden werden, müssen die Namen A[0], A[1], … besitzen.

6.5 Verwendung der Bibliothek lattice.lib

Innerhalb eines Schaltplans kann mit Add > Symbol ein Symbol aus einer vorhandenen Bibliothek in den Schaltplan übernommen werden. Die Bibliothek lattice.lib beinhaltet zahlreiche Funktionsbausteine der Digitaltechnik wie z. B. Negierer, Und-Gatter mit 2 Eingängen, Und-Gatter mit 3 Eingängen, Oder-Gatter mit 2 Eingängen, Addierer, etc. Nach Betätigung von Add > Symbol und Auswahl der Bibliothek lattice.lib werden die in der Bibliothek enthaltenen Symbole und eine kurze Beschreibung ihrer Funktion angezeigt. Eine Dokumentation zu den Funktionsbausteinen ist nicht verfügbar. Ausführlichere Informationen zu den Bibliothekselementen erhält man mit Help und Suche nach dem Namen des Bibliothekselements. Die Symbole aus der Bibliothek lattice.lib genügen nicht europäischen Normen. In Tab. 6.2 ist eine kleine Auswahl der in der Bibliothek lattice.lib enthaltenen Funktionsbausteine zusammengestellt. Im Zusammenhang mit Schaltwerken werden weitere Funktionsbausteine behandelt (siehe Kap. 7.4).

Name	Funktion
and2	Und-Gatter mit 2 Eingängen
and3	Und-Gatter mit 3 Eingängen
gsr	Global Set/Reset
inv	Negierer (Inverter)
mux41	Multiplexer mit 4 Eingängen und einem Ausgang
nd2	Nand mit 2 Eingängen
nr2	Nor mit 2 Eingängen
or2	Oder mit 2 Eingängen
or3	Oder mit 3 Eingängen
vhi	High (logischer Zustand 1)
vlo	Low (logischer Zustand 0)
xor2	Exor mit 2 Eingängen

Tab. 6.2 Funktionsbausteine aus der Bibliothek lattice.lib

Die Funktionsbausteine vhi und vlo dienen zur statischen Erzeugung der Logikpegel Low bzw. High (logische Zustände 0 bzw. 1). Damit dem Eingang eines Funktionsbausteins der logische Zustand 0 bzw. 1 fest zugeordnet wird, muss der Funktionsbaustein vlo bzw. vhi neben dem betreffenden Eingang angeordnet und mit diesem verbunden werden.

Der Funktionsbaustein gsr dient zur Erzeugung eines Global Set/Reset (GSR). Das Symbol besitzt lediglich einen Eingang. Der Eingang ist Low-aktiv. Weist der Eingang den Zustand 0 auf, so wird ein Global Set/Reset verursacht. Hierbei werden bestimmte Flipflops mit dem Zustand 0 bzw. 1 initialisiert. Außerdem werden bestimmte Funktionsbausteine im Extended Function Block (EFB) initialisiert. Eine ausführliche Beschreibung des Resetverhaltens des PLD's XO2 findet sich in

Anhang A6. Der Funktionsbaustein gsr muss in jedem Projekt vorhanden sein, da sonst eine zuverlässige Funktion des Projekts nicht gewährleistet ist. Wenn kein externer Eingang für GSR benötigt wird, wird der Eingang des Funktionsbausteins gsr mit vhi verbunden.

Nach Betätigen der Taste HDL wird ein Blockschaltbild des Schaltplans angezeigt. Durch Rechtsklick auf den Block Schematic und Auswahl von "Goto Source Definition" erhält man das VHDL-Programm, mit dem die im Schaltplan definierte Schaltung erzeugt wird. Voraussetzung hierfür ist, dass für das Projekt unter dem Parameter "Schematic HDL Type" die Option VHDL gewählt wurde (siehe nächstes Kapitel).

In dem automatisch aus dem Schaltplan erzeugten VHDL-Programm findet sich am Anfang der folgende Eintrag

```
library MACHXO2;
use MACHXO2.components.all;
```

Hiermit wird die Bibliothek lattice.lib ausgewählt, in der sich die oben beschriebenen Funktionsbausteine befinden. Diese Funktionsbausteine können auch ohne Verwendung eines Schaltplans innerhalb eines Projekts benutzt werden, indem sie in einem VHDL-Programm als Instanz deklariert werden. Die Vorgehensweise kann dem automatisch aus einem Schaltplan erzeugten VHDL-Programm entnommen werden (siehe auch Kapitel 6.7).

Im Zusammenhang mit der Verwendung von Bibliothekselementen aus der Bibliothek lattice.lib treten Warnungen auf, mit denen auf die Verwendung von "Black Boxes" hingewiesen wird. Die Warnungen haben ihre Ursache darin, dass der VHDL-Code von Bibliothekselementen nicht zugänglich ist. Die Bibliothekselemente sind mittels Netzlisten definiert. Da dies keinen Einfluss auf die Funktionalität eines Projekts hat, können die Warnungen ignoriert werden.

6.6 Simulation von Schaltplänen

Nach Betätigung der Taste HDL wird das Blockschaltbild des Projekts angezeigt. Bei VHDL-Programmen konnte durch Betätigung der rechten Maustaste auf den obersten Block die Funktion "VHDL Test Bench Template" ausgewählt werden. Bei reinen Schaltplanprojekten wird diese Möglichkeit standardmäßig nicht angeboten. In der Standardeinstellung ist die Erzeugung einer Verilog Test Bench vorgesehen.

Um die Standardeinstellung zu ändern, wird ein Rechtsklick auf den Projektnamen im Fenster File List (ganz oben) durchgeführt und Properties ausgewählt. Rechts von Schematic HDL Type ist kein Eintrag vorhanden. Nach Linksklick in dieses leere Feld wird die Option VHDL angeboten. Diese Option muss gewählt werden, damit im HDL-Blockschaltbild nach einem Rechtsklick auf den obersten Block "VHDL Test Bench Template" ausgewählt werden kann.

Es kann vorkommen, dass die Option VHDL nicht sofort übernommen wird und weiterhin die Verilog Test Bench angeboten wird. In diesem Fall das Projekt schließen (File > Close Project) und erneut öffnen.

6.7 Verwendung von Schaltplänen und VHDL-Programmen in einem Projekt

Ein Projekt kann sich aus mehreren Schaltplänen und/oder VHDL-Programmen zusammensetzen. In jedem Schaltplan und jedem VHDL-Programm werden Ein/Ausgänge definiert. Welche Ein/Ausgänge werden in die Spreadsheet-Tabelle übernommen?

Die Antwort lautet: Alle Ein/Ausgänge, die in der Top-Level Unit definiert werden, werden in die Spreadsheet-Tabelle übernommen. Diese Datei wird im Fenster File List fett markiert. Es kann sich hierbei um einen Schaltplan oder ein VHDL-Programm handeln.

Wenn ein Schaltplan die Top-Level Unit ist, so werden die in diesem Schaltplan definierten Ein/Ausgänge als Ein/Ausgänge des PLD verwendet. Diese Ein/Ausgänge werden in der Spreadsheet-Tabelle angezeigt. Die Ein/Ausgänge der anderen Module innerhalb des Projekts (Schaltplan oder VHDL-Programm) dienen lediglich zum internen Datenaustausch innerhalb des PLD.

Oft ist die gewünschte Datei bereits automatisch als Top-Level Unit gekennzeichnet. Ist dies nicht der Fall, so kann aus den vorhandenen Dateien ausgewählt werden, welche Top-Level Unit sein soll. Hierzu ist im Fenster File List ein Rechtsklick auf den Projektnamen (unterhalb von Strategy1) durchzuführen. Nach Auswahl von Set Top-Level Unit wird im Feld rechts von Top-Level Unit der Name der Datei angezeigt, die momentan Top-Level Unit ist. Ist nur eine Datei vorhanden, so ist das Feld leer. Soll eine Änderung erfolgen, so ist das Feld mit einem Linksklick anzuwählen. Aus den vorhandenen Dateien kann nun diejenige Datei ausgewählt werden, die Top-Level Unit sein soll.

Anhand des Beispiels NandTest wird nun gezeigt, wie aus einem VHDL-Programm, das Top-Level Unit ist, ein Schaltplan aufgerufen wird. Erzeugen Sie zunächst ein neues Projekt mit dem Namen NandTest. Erstellen Sie innerhalb dieses Projekts den Schaltplan Nand2, der lediglich über ein Nand-Gatter mit zwei Eingängen verfügen soll. Die Eingänge werden mit A und B bezeichnet. Der Ausgang erhält die Bezeichnung y. Bringen Sie die Ein/Ausgangspfeile für die Ein/Ausgänge an.

Der Schaltplan soll nun aus einem übergeordneten VHDL-Programm mit dem Namen NandTest aufgerufen werden. Erzeugen Sie ein VHDL-Programm mit dem Namen NandTest und übernehmen Sie das in Abb. 6.5 wiedergegebene Programm.

```
entity NandTest is
   port (
   S1, S2: in std_logic;
   D1: out std_logic
   );
end entity NandTest;


architecture behavior of NandTest is

   component nand2 port (
      A, B: in std_logic;
      y: out std_logic
      );
   end component nand2;

begin

   Gate0: nand2 port map (A => S1, B => S2, y => D1);

end architecture behavior;
```

Abb. 6.5 VHDL-Programm NandTest

Unter entity werden die Ein/Ausgänge des PLD deklariert, nämlich die Schalter S1 und S2 sowie die LED D1. Die Ein/Ausgänge des Schaltplans werden mit einer component-Deklaration erfasst. Vom Schaltplan wird die Instanz Gate0 erzeugt. In der zugehörigen Port Map werden die Ein/Ausgänge von Gate0 zugeordnet.

Nach Synthesize Process befinden sich in der Spreadsheet-Tabelle die Einträge S1, S2 und D1. Achten Sie darauf, dass die Datei NandTest.vhd Top-Level Unit ist! Wird eine VHDL Test Bench erzeugt, so ist in dieser Datei der gewünschte Verlauf für die Signale S1 und S2 festzulegen.

Achten Sie darauf, dass sich die Deklaration

```
use work.all;
```

am Anfang des VHDL-Programms befindet!

6.8 Verwendung von Symbolen mit Busverbindungen

Module bzw. Symbole können als Ein/Ausgänge Busverbindungen besitzen. Im Schaltplan müssen diese Busverbindungen Namen in der Form A[3:0] bzw. B[7:4] aufweisen. Der Bus A besitzt in diesem Fall die Signale A[3], A[2], A[1] und A[0]. Der Bus B verfügt über die Signale B[7], B[6], B[5] und B[4].

Die Verwendung von Symbolen mit Busverbindungen wird im Folgenden anhand des Projekts Sum1 demonstriert. Sum1 soll ein Schaltnetz sein, dessen Aufgabe es ist, die Anzahl der Einsen innerhalb des Bytes A zu bestimmen. Die ermittelte Anzahl von Einsen wird mit der 7-Segment-Anzeige DIG1 angezeigt.

Die Anzahl der Einsen im Byte A wird durch zweimalige Verwendung des Moduls Count1 bestimmt. Count1 bestimmt die Einsen innerhalb von jeweils 4 Bits. Das Ausgangssignal von Count1 ist die Summe der Einsen der 4 Eingangsbits, die zwischen 0 und 4 liegen kann. Das Ausgangssignal ist deshalb als Bus S[2:0] ausgeführt.

Mit dem Addierer Add2x3 werden die Sumensignale von zwei Count1-Modulen addiert. Das Resultat wird über einen 4-Bit-Bus nach außen gegeben. Abb. 6.6 zeigt das Schaltbild des Projekts Sum1. Das Eingangsbyte A wird mit den Schaltern S1 (A7), … , S8 (A0) erzeugt.

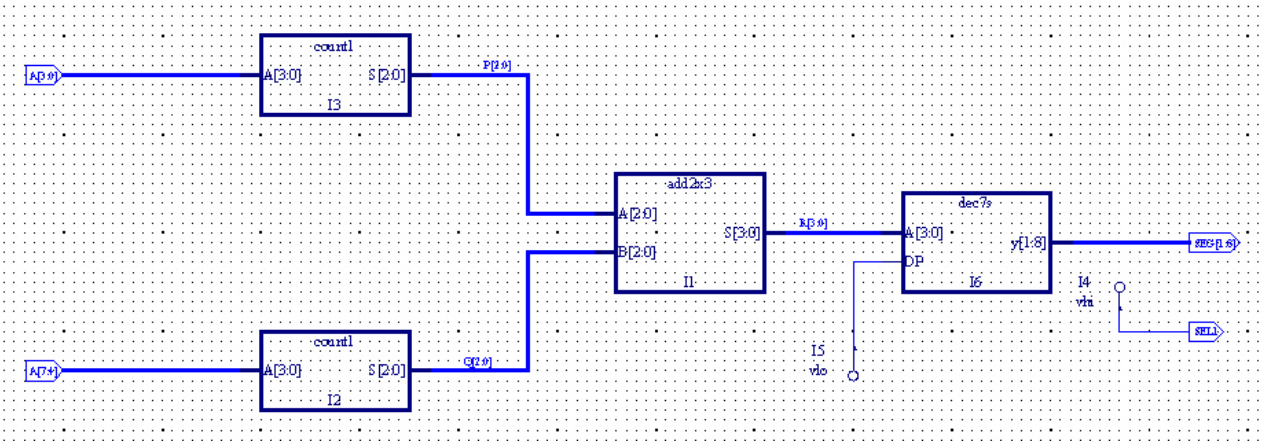


Abb. 6.6 Schaltplan des Projekts Sum1

Im Bus A[3:0] sind die Signale A3, … , A0 enthalten. Dem Bus A[7:4] sind die Signale A7, … , A4 zugeordnet. Der oberen Instanz von Count1 wird der Bus A[3:0] zugeführt. Der Bus A[7:4] ist mit dem Eingang der unteren Instanz von Count1 verbunden. Zwischen der oberen Instanz von Count1 und dem Modul Add2x3 befindet sich der Bus P[2:0]. Die Verbindung zwischen der unteren Instanz von Count1 und dem Modul Add2x3 erfolgt mit dem Bus Q[2:0]. Das Modul Dec7S dient zur Umsetzung vom 4-Bit-Dualcode in den 7-Segment-Code der Anzeige. Der Dezimalpunkt ist ausgeschaltet. Mit SEL1 = 1 wird die Anzeige DIG1 aktiviert. Die logischen Zustände 1 bzw. 0 werden mit den Modulen vhi bzw. vlo erzeugt, die sich in der Bibliothek lattice.lib befinden.

Abb. 6.7 zeigt das VHDL-Programm für Count1. In Abb. 6.8 ist das VHDL-Programm für den Addierer Add2x3 wiedergegeben.

```
entity Count1 is
   port (
   A: in std_logic_vector (3 downto 0);
   S: out std_logic_vector (2 downto 0)
   );
end entity Count1;


architecture behavior of Count1 is

begin

   main: process (A) is
   type int16 is range 0 to 15;
   variable i, sum: int16;
   begin
      sum := 0;
      for i in 0 to 3 loop
         if A(i) = '1' then
            sum := sum + 1;
         end if;
      end loop;
      S <= std_logic_vector (to_unsigned (integer (sum), 3));
   end process main;

end architecture behavior;
```

Abb. 6.7 VHDL-Programm für Count1

```
entity Add2x3 is
   port (
   A, B: in std_logic_vector (2 downto 0);
   S: out std_logic_vector (3 downto 0)
   );
end entity Add2x3;


architecture behavior of Add2x3 is

begin

   main: process (A, B) is
   variable sum: unsigned (3 downto 0);
   begin
      sum := unsigned ('0' & A (2 downto 0)) + unsigned ('0' & B);
      S <= std_logic_vector (sum);
   end process main;

end architecture behavior;
```

Abb. 6.8 VHDL-Programm für Add2x3

Erstellen Sie zunächst die Dateien Count1.vhd und Add2x3.vhd im Verzeichnis Sum1. Betätigen Sie die Taste HDL und erzeugen Sie durch Rechtsklick auf den entsprechenden Block und Auswahl von "Generate Schematic Symbol" die Symbole für die Module Count1 und Add2x3. Zeichnen Sie anschließend den Schaltplan Sum1. Erzeugen Sie die JEDEC-Datei und programmieren Sie das PLD des Demoboards. Überprüfen Sie die Funktion der Schaltung mit dem Demoboard.

Beachten Sie, dass die Elemente eines Felds bzw. Vektors oder Busses unterschiedlich bezeichnet werden, obwohl dieselben Elemente gemeint sind. Das Feld A besitzt z. B. die Elemente A3, … , A0. In der entity-Deklaration eines VHDL-Programms lautet die Deklaration:

```
A: in std_logic_vector (3 downto 0);
```

Auf die einzelnen Elemente von A wird innerhalb des VHDL-Programms zugegriffen z. B. durch A(3).

Ein Bus trägt z. B. den Namen A[3:0]. Der Zugriff auf die einzelnen Bussignale erfolgt z. B. durch A[3]. Wird von einem VHDL-Programm mit der obigen Vektordeklaration ein Symbol erzeugt, so besitzt das Symbol als Eingang einen Bus mit dem Namen A[3:0].

Beim Programm Add2x3 sind bei der Berechnung von sum bewusst zwei Alternativen für die Benutzung des &-Operators angegeben. In beiden Fällen wird aus einem 3-Bit-Vektor durch Hinzufügen einer 0 auf der linken Seite ein 4-Bit-Vektor erzeugt.

7 Schaltwerke

7.1 Flipflops

Ein wesentlicher Bestandteil von Schaltwerken sind Flipflops. Im Folgenden wird die Realisierung einiger typischer Flipflops mit VHDL betrachtet.

7.1.1 RS-Flipflop

Zunächst soll ein RS-Flipflop mit VHDL definiert werden. Abb. 7.1 zeigt das Schaltbild des Flipflops. Zur Erzeugung von R wird Schalter S1 verwendet. S wird mit Schalter S2 erzeugt. Mit der LED D1 wird der Zustand Q des Flipflops angezeigt. Der irreguläre Zustand R = S = 1 darf nicht auftreten. In Abb. 7.2 ist das zugehörige VHDL-Programm wiedergegeben. Der Prozess ff wird nur aktiv, wenn sich eines der Signale R bzw. S ändert. Im Fall R = S = 0 erfolgt innerhalb des Prozesses keine Aktion. Der Zustand von Q wird nicht verändert (Speichern).

Abb. 7.1 RS-Flipflop ohne Taktsteuerung

```
entity rsff is
   port (
   R, S: in std_logic;
   Q: out std_logic
   );
end entity rsff;


architecture behavior of rsff is

begin
   ff: process (R, S) is
   begin
      if R = '1' then
         Q <= '0';
      end if;
      if S = '1' then
         Q <= '1';
      end if;
   end process ff;

end architecture behavior;
```

Abb. 7.2 VHDL-Programm für das RS-Flipflop

Erstellen Sie ein neues Projekt mit dem Namen RSFF und erzeugen Sie eine VHDL-Datei mit demselben Namen. Geben Sie das Programm ein. Erzeugen Sie eine Testbench und ergänzen Sie den Prozess tb wie folgt:

```
tb: process
begin
   R <= '0';
   S <= '0';
   wait for 100 ns;
   S <= '1';
   wait for 100 ns;
   S <= '0';
   wait for 100 ns;
   R <= '1'
   wait for 100 ns;
   R <= '0'
   wait;
end process;
```

Abb. 7.3 Testbench für das RS-Flipflop

Führen Sie eine Simulation durch. Der durch die Simulation erhaltene Signalverlauf ist in Abb. 7.4 wiedergegeben. Erproben Sie das Flipflop am Demoboard.

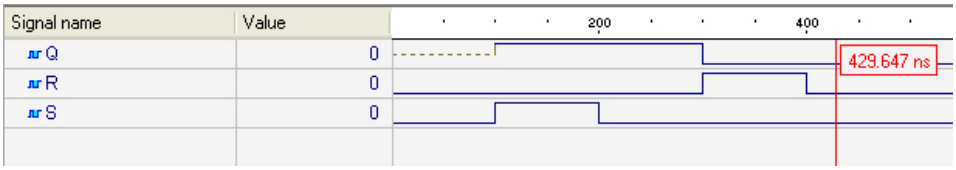


Abb. 7.4 Simulation des RS-Flipflops

Achtung! Hardware-Problem!

Spreadsheet: R und S werden als Clock erkannt. Warum?

Hardware: S = 1 bewirkt Q = 1. Nach der Rücknahme S = 0 folgt gleichzeitig Q = 0. Speichern erfolgt nicht. Die Flipflop-Funktion wird nicht korrekt ausgeführt.

Da Diamond die mit VHDL definierte Flipflop-Funktion nicht korrekt umsetzt, wird das RS-Flipflop nun mit einem alternativen Verfahren definiert. Bekanntlich kann ein RS-Flipflop mit Hilfe von zwei NOR-Gattern gemäß Abb. 7.5 aufgebaut werden.

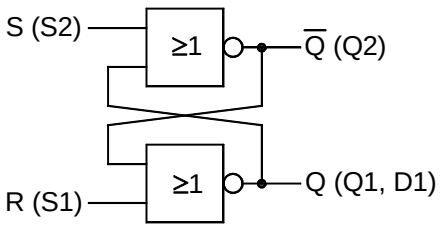


Abb. 7.5 RS-Flipflop mit zwei NOR-Gattern

Diese Struktur wird nun unmittelbar mit einem VHDL-Programm definiert. Das Programm ist in Abb. 7.6 wiedergegeben.

```
entity rsff_nor is
   port (
   R, S: in std_logic;
   Q: out std_logic
   );
end entity rsff_nor;


architecture behavior of rsff_nor is

begin

   ff: process (R, S) is

   variable Q1, Q2: std_logic;

   begin
      Q1 := R nor Q2;
      Q2 := S nor Q1;
      Q <= Q1;
   end process ff;

end architecture behavior;
```

Abb. 7.6 Programm für die Realisierung des RS-Flipflops mit NOR-Gattern

Erstellen Sie ein neues Projekt mit dem Namen RSFF_NOR und erzeugen Sie eine VHDL-Datei mit demselben Namen. Geben Sie das obige Programm ein und führen Sie die Simulation mit der Testbench aus Abb. 7.3 durch. Es ergibt sich das in Abb. 7.4 dargestellte Timing-Diagramm.

Erzeugen Sie die JEDEC-Datei und programmieren Sie das PLD des Demoboards. Überprüfen Sie die Funktion des RS-Flipflops mit dem Demoboard. Die Funktion eines RS-Flipflops wird nun korrekt ausgeführt.

Das RS-Flipflop soll nun mit dem Taktsignal CLK ergänzt werden (siehe Abb. 7.5). Nur im Fall CLK = 1 werden die Eingänge R und S berücksichtigt. Das zugehörige VHDL-Programm ist in Abb. 7.6 wiedergegeben.

Abb. 7.7 RS-Flipflop mit Taktsteuerung

```
entity rsff_pz is
   port (
   R, S, CLK: in std_logic;
   Q: out std_logic
   );
end entity rsff_pz;
```

```
architecture behavior of rsff_pz is

begin
   ff: process (R, S, CLK) is
   begin
      if CLK = '1' then
         if R = '1' then
            Q <= '0';
         end if;
         if S = '1' then
            Q <= '1';
         end if;
      end if;
   end process ff;

end architecture behavior;
```

Abb. 7.8 VHDL-Programm für das RS-Flipflop mit positiver Taktzustandssteuerung

Erzeugen Sie ein neues Projekt mit dem Namen RSFF_PZ und eine VHDL-Datei mit demselben Namen. Übernehmen Sie das obige VHDL-Programm und führen Sie die Simulation durch. Ergänzen Sie die Testbench wie folgt:

```
tb: process
begin
   R <= '0';
   S <= '0';
   CLK <= '0';
   wait for 100 ns;
   S <= '1';
   wait for 100 ns;
   CLK <= '1';
   wait for 100 ns;
   CLK <= '0';
   wait for 100 ns;
   S <= '0';
   wait for 100 ns;
   R <= '1';
   wait for 100 ns;
   CLK <= '1';
   wait for 100 ns;
   R <= '0';
   wait for 100 ns;
   CLK <= '0';
   wait for 100 ns;
   S <= '1';
   wait for 100 ns;
   CLK <= '1';
   wait for 100 ns;
   S <= '0';
   wait for 100 ns;
   R <= '1';
   wait for 100 ns;
   R <= '0';
   wait;
end process;
```

Abb. 7.9 Testbench für das positiv zustandsgesteuerte RS-Flipflop

Führen Sie die Simulation durch und überprüfen Sie das Timing-Diagramm. Abb. 7.8 zeigt den Verlauf des Timing-Diagramms. Programmieren Sie das PLD auf dem Demoboard und testen Sie die Schaltung mit dem Demoboard.

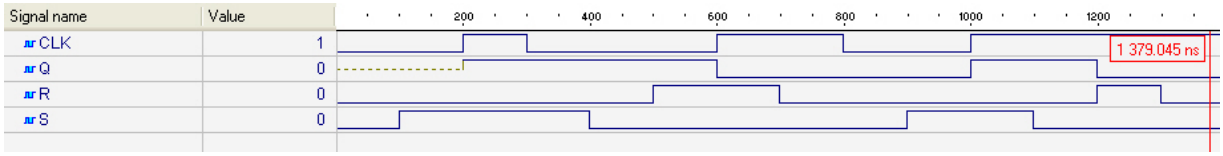


Abb. 7.10 Simulation des RS-Flipflops mit positiver Taktzustandssteuerung

Zunächst erkennt man das gewöhnliche Verhalten eines taktzustandsgesteuerten RS-Flipflops, bei dem S = 1 bzw. R = 1 erst nach Auftreten von CLK = 1 übernommen werden. Bei 1200 ns ist zunächst Q = 1 und CLK = 1. Während der Takt aktiv ist, tritt R = 1 auf, was dann unmittelbar zum Rücksetzen des Flipflops auf Q = 0 führt.

Bei den bisherigen Entwürfen darf der irreguläre Zustand R = S = 1 nicht angelegt werden, da sich das Flipflop nicht definiert verhalten würde. Dies lässt sich vermeiden, indem das VHDL-Programm wie folgt verändert wird:

```
if CLK = '1' then
   if R = '1' then
      Q <= '0';
   else
      if S = '1' then
         Q <= '1';
      end if;
   end if;
end if;
```

Im Fall R = S = 1 würde nun R = 1 ausgeführt, da diese Abfrage zunächst erfolgt (Rücksetzvorrang). Vertauscht man die Reihenfolge der Abfragen von R und S, so ergibt sich Setzvorrang. Bei der nachfolgenden Änderung würde sich Signalerhalt ergeben.

```
if CLK = '1' then
   if R = '1' and S = '0' then
      Q <= '0';
   else
      if S = '1' and R = '0' then
         Q <= '1';
      end if;
   end if;
end if;
```

Damit das RS-Flipflop positiv flankengetriggert ist, muss das VHDL-Programm wie folgt verändert werden:

```
architecture behavior of rsff_pf is

begin
   ff: process (CLK) is
   begin
      if CLK'event and CLK = '1' then    -- positive Flankentriggerung
         if R = '1' then
            Q <= '0';
         end if;
         if S = '1' then
            Q <= '1';
         end if;
      end if;
   end process ff;
```

```
end architecture behavior;
```

Abb. 7.11 VHDL-Programm für das positiv flankengetriggerte RS-Flipflop

Erzeugen Sie ein neues Projekt mit dem Namen RSFF_PF und erstellen Sie die Datei RSFF_PF.VHD. Geben Sie das obige VHDL-Programm ein und führen Sie die Simulation durch. Verwenden Sie die Testbench aus Abb. 7.7. Abb. 7.10 zeigt das Resultat der Simulation. Veränderungen von R und S während CLK = 1 wirken sich nun nicht mehr sofort aus, sondern erst bei der nächsten positiven Taktflanke. Bei 1200 ns erfolgt deshalb kein Reset des Flipflops.

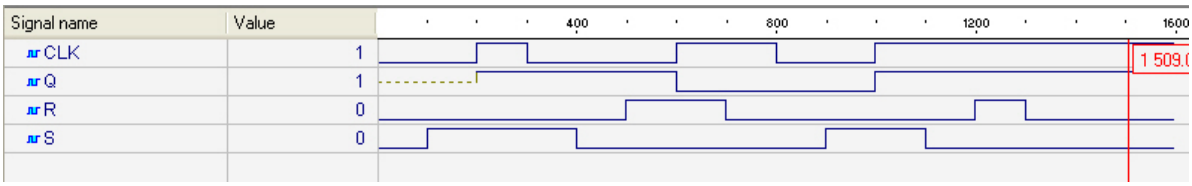


Abb. 7.12 Simulation de RS-Flipflops mit positiver Taktflankensteuerung

Programmieren Sie das PLD auf dem Demoboard und überprüfen Sie die Funktion der Schaltung mit dem Demoboard.

Für negative Flankentriggerung muss die obige if-Anweisung durch die folgende Anweisung ersetzt werden:

```
if CLK'event and CLK = '0' then   -- negative Flankentriggerung
```

7.1.2 D-Flipflop

Zunächst soll ein positiv zustandsgesteuertes Einspeicher-D-Flipflop mit VHDL definiert werden. Abb. 7.11 zeigt das Schaltbild des D-Flipflops. Das Takstsignal CLK wird mit dem roten Taster S11 erzeugt.

Abb. 7.13 D-Flipflop

Im Fall CLK = 1 wird der D-Eingang übernommen. In Abb. 7.12 ist das VHDL-Programm wiedergegeben. Die Bezeichnung dff_pz dient zur Kennzeichnung als positiv zustandsgesteuertes D-Flipflop.

```
entity dff_pz is
   port (
      CLK, D: in std_logic;
      Q: out std_logic
      );
```

```
end entity dff_pz;


architecture behavior of dff_pz is

begin

   dff: process (CLK, D) is
   begin
      if CLK = '1' then
         Q <= D;
      end if;
   end process dff;

end architecture behavior;
```

Abb. 7.14 VHDL-Programm für das D-Flipflop

Erzeugen Sie ein neues Projekt mit dem Namen DFF_PZ und erstellen Sie eine VHDL-Datei mit demselben Namen. Geben Sie das Programm ein. Erzeugen Sie eine VHDL Testbench und ergänzen Sie den Prozess tb wie folgt:

```
tb: process
begin
   CLK <= '0';
   D <= '0';
   wait for 100 ns;
   CLK <= '1';
   wait for 100 ns;
   CLK <= '0';
   wait for 100 ns;
   D <= '1';
   wait for 100 ns;
   CLK <= '1';
   wait for 100 ns;
   CLK <= '0';
   wait for 100 ns;
   D <= '0';
   wait for 100 ns;
   D <= '1';
   wait for 100 ns;
   CLK <= '1';
   wait for 100 ns;
   D <= '0';
   wait;
end process;
```

Abb. 7.15 Testbench für das D-Flipflop

Starten Sie den Simulator und überprüfen Sie die Funktion. Programmieren Sie das Demoboard und überprüfen Sie die Funktionsweise des D-Flipflops am Demoboard. Abb. 7.14 zeigt das Timing-Diagramm, das die Simulation erzeugt. Ab 600 ns ist D = 0 für 100 ns. Dies wirkt sich auf Q jedoch nicht aus, da CLK = 0. Ab 900 ns ist D = 0. Dies wirkt sich sofort auf Q aus, da CLK = 1.

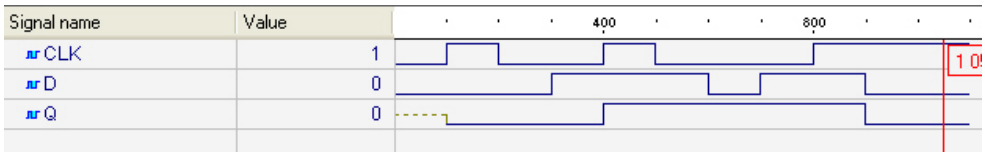


Abb. 7.16 Simulation des D-Flipflops

Programmieren Sie das PLD im Demoboard und überprüfen Sie die Funktion der Schaltung am Demoboard.

Nun soll das D-Flipflop einen asynchronen Reset-Eingang erhalten. Dieser Eingang wird mit RD (Reset Direct) bezeichnet (siehe Abb. 7.15). Das modifizierte VHDL-Programm ist in Abb. 7.16 wiedergegeben.

Abb. 7.17 D-Flipflop mit asynchronem Reset

```
entity dff_pz_rd is
   port (
      CLK, D, RD: in std_logic;
      Q: out std_logic
      );
end entity dff_pz_rd;


architecture behavior of dff_pz_rd is

begin
   dff: process (CLK, D, RD) is
   begin
      if RD = '1' then
         Q <= '0';
      else
         if CLK = '1' then
            Q <= D;
         end if;
      end if;
   end process dff;

end architecture behavior;
```

Abb. 7.18 VHDL-Programm für das D-Flipflop mit asynchronem Reset

Erzeugen Sie ein neues Projekt mit dem Namen DFF_PZ_RD und erstellen Sie eine VHDL-Datei mit demselben Namen. Geben Sie das obige Programm ein und führen Sie die Simulation durch. Ergänzen Sie zuvor die Testbench wie nachfolgend angegeben.

```
tb: process
begin
   CLK <= '0';
   D <= '0';
   RD <= '0';
   wait for 100 ns;
   CLK <= '1';
   wait for 100 ns;
   CLK <= '0';
   wait for 100 ns;
   D <= '1';
   wait for 100 ns;
   CLK <= '1';
   wait for 100 ns;
   CLK <= '0';
```

```
   wait for 100 ns;
   D <= '0';
   wait for 100 ns;
   D <= '1';
   wait for 100 ns;
   CLK <= '1';
   wait for 100 ns;
   RD <= '1';
   wait for 100 ns;
   RD <= '0';
   wait for 100 ns;
   D <= '0';
   wait for 100 ns;
   CLK <= '0';
   wait for 100 ns;
   D <= '1';
   wait for 100 ns;
   CLK <= '1';
   wait for 100 ns;
   CLK <= '0';
   wait for 100 ns;
   RD <= '1';
   wait for 100 ns;
   RD <= '0';
   wait;
end process;
```

Abb. 7.19 Testbench für das D-Flipflop mit asynchronem Reset

In Abb. 7.18 ist das Timing-Diagramm dargestellt, das sich aus der Simulation ergibt. Ab 900 ns ist RD = 1 für 100 ns. Dies wirkt sich sofort auf Q aus und erzeugt Q = 0. Danach ist weiterhin CLK = 1. Deshalb ist sofort wieder Q = D = 1. Bei 1600 ns verhält es sich anders. Ab 1600 ns ist RD = 1 für die Dauer von 100 ns. Dies wirkt sich sofort aus und verursacht Q = 0. Ab 1700 ns ist RD = 0. Nun ist aber CLK = 0. Deshalb wirkt sich D = 1 nicht auf Q aus.

Abb. 7.20 Simulation des positiv zustandsgesteuerten D-Flipflops mit asynchronem Reset

Programmieren Sie das PLD im Demoboard und erproben Sie die Schaltung am Demoboard.

Achtung! Hardware-Problem!

D = 1, CLK = 0, Q = 1: RD = 1 bewirkt Q = 0. Wenn danach wieder RD = 0 folgt, ergibt sich sofort Q = 1. RD löscht das Flipflop also nicht.

Damit das D-Flipflop als positiv flankengetriggertes Flipflop arbeitet, ist das VHDL-Programm wie folgt zu ändern:

```
architecture behavior of dff_pf_rd is

begin

   dff: process (CLK, RD) is
```

```
   begin
      if RD = '1' then
         Q <= '0';
      else
         if CLK'event and CLK = '1' then
            Q <= D;
         end if;
      end if;
   end process dff;

end architecture behavior;
```

Abb. 7.21 VHDL-Programm für das positiv flankengetriggerte D-Flipflop mit asynchronem Reset

Erstellen Sie ein neues Projekt mit dem Namen DFF_PF_RD und erzeugen Sie eine VHDL-Datei mit demselben Namen. Übernehmen Sie das obige Programm und erstellen Sie die Testbench gemäß Abb. 7.20. Wenn Sie nun die Simulation ausführen, ergibt sich das in Abb. 7.21 dargestellte Timing-Diagramm.

```
tb: process
begin
   D <= '0';
   CLK <= '0';
   RD <= '0';
   wait for 100 ns;
   CLK <= '1';
   wait for 100 ns;
   CLK <= '0';
   wait for 100 ns;
   D <= '1';
   wait for 100 ns;
   CLK <= '1';
   wait for 100 ns;
   CLK <= '0';
   wait for 100 ns;
   RD <= '1';
   wait for 100 ns;
   RD <= '0';
   wait;
```

Abb. 7.22 Testbench für das positiv flankengesteuerte D-Flipflop mit asynchronem Reset

Abb. 7.23 Simulation des D-Flipflops mit positiver Flankensteuerung und asynchronem Reset

Programmieren Sie das PLD auf dem Demoboard und erproben Sie die Schaltung mit dem Demoboard.

Das D-Flipflop mit asynchronem Reset kann ergänzt werden durch ein asynchrones Preset-Signal SD (Set Direct). Das zugehörige Blockschaltbild ist in Abb. 7.22 dargestellt.

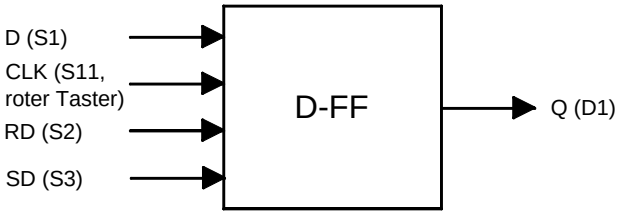


Abb. 7.24 D-Flipflop mit positiver Flankensteuerung und asynchronem Reset sowie Preset

Das zugehörige VHDL-Programm ist in Abb. 7.23 wiedergegeben. Der irreguläre Zustand RD = SD = 1 ist zulässig, da in diesem Fall Reset erfolgt. Erstellen Sie ein neues Projekt mit dem Namen DFF_PF_RD_SD und erzeugen Sie eine VHDL-Datei mit demselben Namen. Übernehmen Sie das Programm und führen Sie die Simulation durch unter Verwendung der in Abb. 7.24 wiedergegebenen Testbench.

```
architecture behavior of dff_pf_rd_sd is

begin

   dff: process (CLK, RD) is
   begin
      if RD = '1' then
         Q <= '0';
      else
         if SD = '1' then
            Q <= '1';
         else
            if CLK'event and CLK = '1' then
               Q <= D;
            end if;
         end if;
      end if;
   end process dff;

end architecture behavior;
```

Abb. 7.25 VHDL-Programm für positiv flankengetriggertes D-Flipflop mit asynchronem Reset und Preset

```
tb: process
begin
   D <= '0';
   CLK <= '0';
   RD <= '0';
   SD <= '0';
   wait for 100 ns;
   CLK <= '1';
   wait for 100 ns;
   CLK <= '0';
   wait for 100 ns;
   D <= '1';
   wait for 100 ns;
   CLK <= '1';
   wait for 100 ns;
   CLK <= '0';
   wait for 100 ns;
   RD <= '1';
   wait for 100 ns;
   RD <= '0';
   wait for 100 ns;
   SD <= '1';
```

```
wait for 100 ns;
SD <= '0';
wait for 100 ns;
SD <= '1';
wait for 100 ns;
RD <= '1';
wait;
```

Abb. 7.26 Testbench für das positiv flankengetriggerte D-Flipflop mit asynchronem Reset und Preset

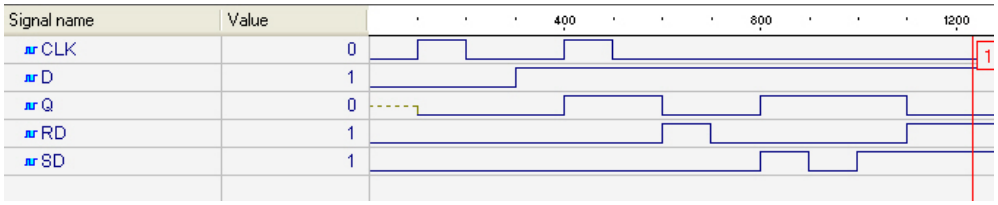


Abb. 7.27 Simulation für das positiv flankengetriggerte D-Flipflop mit asynchronem Reset und Preset

Im Timing-Diagramm erkennt man ab 1100 ns den irregulären Betrieb RD = SD = 1. Das Flipflop wird zurückgesetzt, da Reset im vorliegenden Fall dominant ist.

Programmieren Sie das PLD auf dem Demoboard und erproben Sie die Schaltung am Demoboard.

7.1.3 T-Flipflop

Betrachtet wird ein positiv flankengetriggertes T-Flipflop mit asynchronem Reset gemäß Abb. 7.26. Das zugehörige VHDL-Programm ist in Abb. 7.27 wiedergegeben.

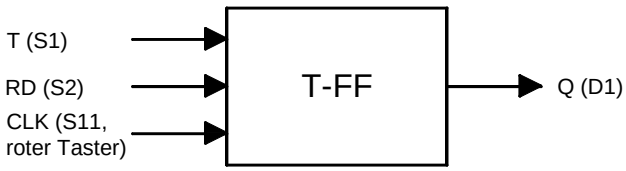


Abb. 7.28 Positiv flankengetriggertes T-Flipflop mit asynchronem Reset

```
architecture behavior of tff_pf_rd is

signal Qint: std_logic;    -- interner Speicher

begin

   ff: process (CLK, RD) is
   begin
      if RD = '1' then
         Qint <= '0';
      else
         if CLK'event and CLK = '1' then
            if T = '1' then
```

```
                Qint <= not Qint;
            end if;
         end if;
      end if;
   end process ff;

   Q <= Qint;

end architecture behavior;
```

Abb. 7.29 VHDL-Programm für das positiv flankengetriggerte D-Flipflop mit asynchronem Reset

Zum Speichern des internen Zustands wird das Signal Qint verwendet. Realisieren Sie die Testbench gemäß Abb. 7.28 und führen Sie die Simulation durch. Das Timing-Diagramm ist in Abb. 7.29 wiedergegeben.

```
tb: process
begin
   T <= '0';
   RD <= '0';
   CLK <= '0';
   wait for 100 ns;
   RD <= '1';
   wait for 100 ns;
   RD <= '0';
   wait for 100 ns;
   CLK <= '1';
   wait for 100 ns;
   CLK <= '0';
   wait for 100 ns;
   T <= '1';
   wait for 100 ns;
   CLK <= '1';
   wait for 100 ns;
   CLK <= '0';
   wait;
```

Abb. 7.30 Testbench für das positive flankengetriggerte T-Flipflop mit asynchronem Reset

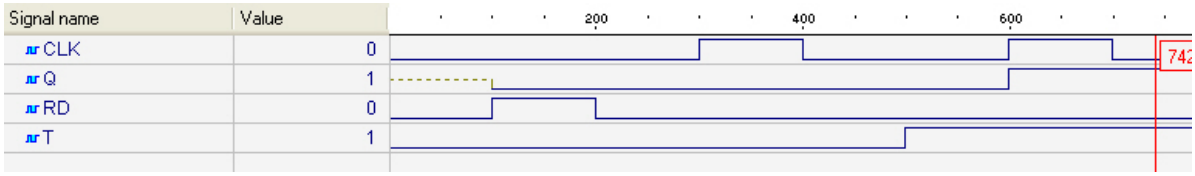


Abb. 7.31 Simulation des positiv flankengetriggerten T-Flipflops mit asynchronem Reset

Programmieren Sie das PLD auf dem Demoboard und erproben Sie die Schaltung mit dem Demoboard.

7.1.4 JK-Flipflop

Es soll ein positiv flankengetriggertes JK-Flipflop mit asynchronem Reset gemäß Abb. 7.30 realisiert werden. Das zugehörige VHDL-Programm ist in Abb. 7.31 wiedergegeben.

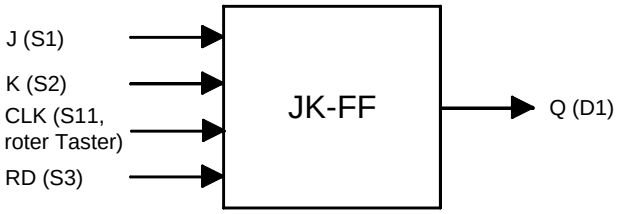


Abb. 7.32 Positiv flankengetriggertes JK-Flipflop mit asynchronem Reset

```
architecture behavior of jkff_pf_rd is

signal Qint: std_logic;                         -- interner Speicher
signal input: std_logic_vector (1 downto 0);  -- Input

begin
   input (1) <= J;
   input (0) <= K;

   ff: process (CLK, RD) is
   begin
      if RD = '1' then
         Qint <= '0';
      else
         if CLK'event and CLK = '1' then
            case input is
               when "11" => Qint <= not Qint;
               when "10" => Qint <= '1';
               when "01" => Qint <= '0';
               when others => null;
            end case;
         end if;
      end if;
   end process ff;

   Q <= Qint;

end architecture behavior;
```

Abb. 7.33 VHDL-Programm für das positiv flankengetriggerte JK-Flipflop mit asynchronem Reset

Führen Sie eine Simulation des JK-Flipflops mit der Testbench gemäß Abb. 7.32 durch. Das resultierende Timing-Diagramm ist in Abb. 7.33 wiedergegeben.

```
tb: process
begin
   J <= '0';
   K <= '0';
   RD <= '0';
   CLK <= '0';
   wait for 100 ns;
   J <= '1';
   wait for 100 ns;
   CLK <= '1';
   wait for 100 ns;
   CLK <= '0';
   wait for 100 ns;
   J <= '0';
   wait for 100 ns;
   K <= '1';
   wait for 100 ns;
   CLK <= '1';
```

```
wait for 100 ns;
CLK <= '0';
wait for 100 ns;
K <= '0';
wait for 100 ns;
J <= '1';
wait for 100 ns;
CLK <= '1';
wait for 100 ns;
RD <= '1';
wait for 100 ns;
J <= '0';
wait for 100 ns;
RD <= '0';
wait for 100 ns;
CLK <= '0';
wait;
```

Abb. 7.34 Testbench für das positiv flankengetriggerte JK-Flipflop mit asynchronem Reset

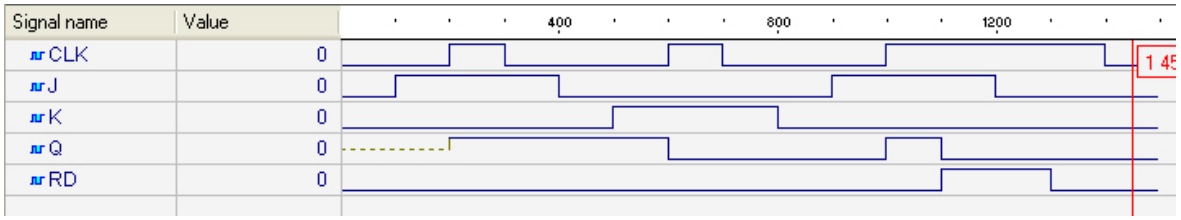


Abb. 7.35 Simulation des positiv flankengetriggerten JK-Flipflops mit asynchronem Reset

Programmieren Sie das PLD des Demoboards und überprüfen Sie die Funktion des JK-Flipflops mit dem Demoboard.

7.2 4-Bit-Zähler

Es soll ein 4-Bit-Zähler gemäß Abb. 7.34 realisiert werden. Der Zähler soll aufwärts Zählen bis zum Erreichen der höchsten Zahl 15. Danach beginnt der Zähler wieder bei 0. Mit dem asynchronen Reset RD = 1 kann der Zähler auf den Anfangszustand 0 gesetzt werden. Der Zähler soll positiv flankengetriggert arbeiten. Der aktuelle Zählerstand wird mit den LED’s D1, … , D4 angezeigt.

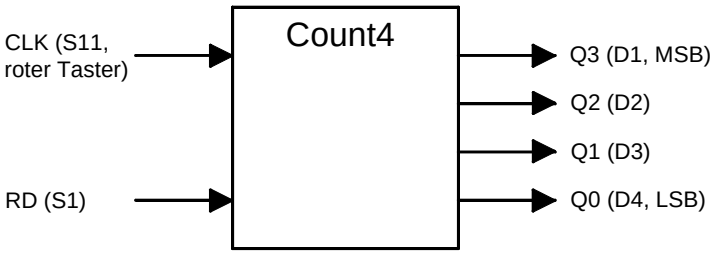


Abb. 7.36 4-Bit-Zähler

Das VHDL-Programm des Zählers ist in Abb. 7.35 angegeben. Erzeugen Sie ein neues Projekt mit dem Namen Count4 und erstellen Sie eine VHDL-Datei mit demselben Namen. Geben Sie das Programm ein. Erstellen Sie eine Testbench gemäß Abb. 7.36 und führen Sie die Simulation durch. Das resultierende Timing-Diagramm ist in Abb. 7.37 wiedergegeben.

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

entity count4 is
   port (
      CLK, RD: in std_logic;
      Q: out std_logic_vector (3 downto 0)
      );
end entity count4;


architecture behavior of count4 is

begin
   count: process (CLK, RD) is

   type int15 is range 0 to 15;
   variable count_value: int15;
   variable sum: int15;
   variable i: int15;
   variable rest: int15;
   variable y: std_logic_vector (3 downto 0);

   begin

      -- Reset

      if RD = '1'
         count_value := 0;
      else

         -- Zaehler

         if CLK'event and CLK = '1' then
            if count_value < 15 then
               count_value := count_value + 1;
            else
               count_value := 0;
            end if;
         end if;
      end if;

      -- Umsetzung Integer in Bit Vektor

      sum := count_value;
      for i in 0 to 3 loop
         rest := sum mod 2;
         sum := sum/2;
         if rest = 1 then
            y(i) := '1';
         else
            y(i) := '0';
         end if;
      end loop;

      Q <= y;

   end process count;

end architecture behavior;
```

Abb. 7.37 VHDL-Programm für den 4-Bit-Zähler

```
tb: process
begin
   RD <= '0';
   CLK <= '0';
   wait for 100 ns;
   CLK <= '1';          --1
   wait for 100 ns;
   CLK <= '0';
   wait for 100 ns;
   CLK <= '1';          --2
   wait for 100 ns;
   CLK <= '0';
   wait for 100 ns;
   CLK <= '1';          --3
   wait for 100 ns;
   CLK <= '0';
   wait for 100 ns;
   CLK <= '1';          --4
   wait for 100 ns;
   CLK <= '0';
   wait for 100 ns;
   CLK <= '1';          --5
   wait for 100 ns;
   CLK <= '0';
   wait for 100 ns;
   CLK <= '1';          --6
   wait for 100 ns;
   CLK <= '0';
   wait for 100 ns;
   RD <= '1';           --0
   wait for 100 ns;
   CLK <= '1';          --0
   wait for 100 ns;
   CLK <= '0';
   wait for 100 ns;
   RD <= '0';
   wait for 100 ns;
   CLK <= '1';          --1
   wait for 100 ns;
   CLK <= '0';
   wait for 100 ns;
   CLK <= '1';          --2
   wait for 100 ns;
   CLK <= '0';
   wait;
```

Abb. 7.38 Testbench für den 4-Bit-Zähler

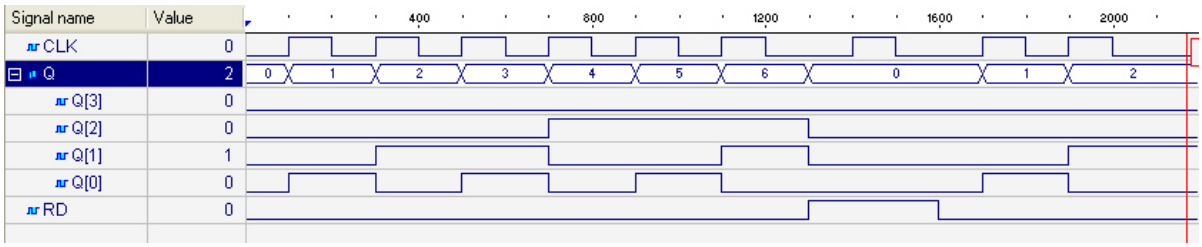


Abb. 7.39 Simulation des 4-Bit-Zählers

Dem Timing-Diagramm entnimmt man, dass ab 1400 ns CLK = 1 ist. Trotzdem bleibt der Zählerstand bei 0, da RD = 1 ist. Erst bei 1700 ns ist RD = 0 und der Zähler wird bei CLK = 0/1 inkrementiert.

Programmieren Sie den Zähler in das PLD auf dem Demoboard und erproben Sie den Zähler mit dem Demoboard.

7.2.1 4-Bit-Zähler mit 7-Segment-Anzeige

Der Zählerstand soll nun nicht mehr alleine durch die LED’s D1, … , D4 angezeigt werden, sondern die Anzeige soll gleichzeitig hexadezimal mit der 7-Segment-Anzeige DIG1 erfolgen (0, … , F). Das Blockschaltbild der Schaltung ist in Abb. 7.40 angegeben.

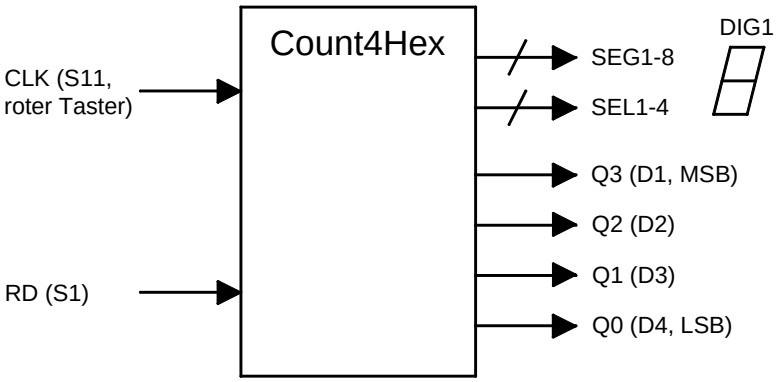


Abb. 7.40 4-Bit-Zähler mit hexadezimaler und dualer Ausgabe

Damit die Ausgabe hexadezimal mit der 7-Segment-Anzeige erfolgen kann, muss der Dualcode in den 7-Segment-Code umgewandelt werden. Dies erfolgt mit dem Code-Umsetzer Dec7S. Das VHDL-Programm für das Modul Dec7S ist in Abb. 7.41 angegeben. Die Zuordnung der Segmente der 7-Segment-Anzeige findet sich in Kap. 3.3. Dec7S ist in der Programmsammlung VHDL4Digilab.zip enthalten, die vom internen Downloadbereich der Fakultät heruntergeladen werden kann.

Das obige Modul Count4 und das Modul Dec7S werden gemäß Abb. 7.40 zusammengeschaltet. Es muss ein übergeordnetes Modul erzeugt werden, das die Module Count4 und Dec7S aufruft und die Signale schaltet. Das übergeordnete Modul erhält die Bezeichnung Count4Hex.

```
entity dec7s is
   port (
   A: in std_logic_vector (3 downto 0); -- Dualcode
   DP: in std_logic;                    -- Dezimalpunkt
   y: out std_logic_vector (1 to 8)
   );
end entity dec7s;


architecture behavior of dec7s is

begin
   dec: process (A, DP) is
   variable segments: std_logic_vector (1 to 8);
   begin
      case A is
         when "0000" => segments := "11111100"; --0
         when "0001" => segments := "01100000"; --1
         when "0010" => segments := "11011010"; --2
         when "0011" => segments := "11110010"; --3
         when "0100" => segments := "01100110"; --4
         when "0101" => segments := "10110110"; --5
         when "0110" => segments := "10111110"; --6
         when "0111" => segments := "11100000"; --7
         when "1000" => segments := "11111110"; --8
         when "1001" => segments := "11110110"; --9
         when "1010" => segments := "11101110"; --A
```

```
         when "1011" => segments := "00111110"; --B
         when "1100" => segments := "10011100"; --C
         when "1101" => segments := "01111010"; --D
         when "1110" => segments := "10011110"; --E
         when "1111" => segments := "10001110"; --F
         when others => null;
      end case;
      if DP = '1' then
         y <= segments or "00000001";
      else
         y <= segments;
      end if;
   end process dec;
end architecture behavior;
```

Abb. 7.41 VHDL-Programm für den Code-Umsetzer Dec7S

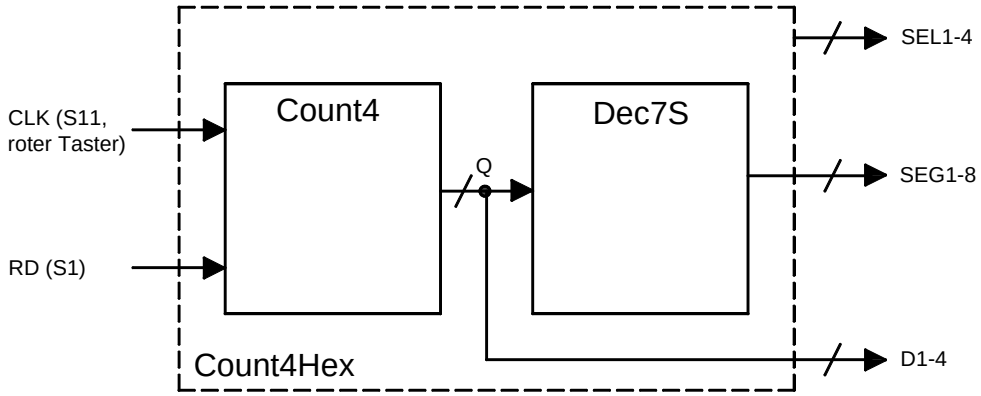


Abb. 7.42 Zusammenschaltung der Module Count4 und Dec7S

Erzeugen Sie ein neues Projekt mit dem Namen Count4Hex und erstellen Sie eine VHDL-Datei mit demselben Namen. Geben Sie das VHDL-Programm gemäß Abb. 7.43 ein.

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

entity Count4Hex is
   port (
      CLK, RD: in std_logic;
      D: out std_logic_vector (1 to 4);
      SEL: out std_logic_vector (1 to 4);
      SEG: out std_logic_vector (1 to 8)
      );
end entity Count4Hex;


architecture structure of Count4Hex is

   signal Qint: std_logic_vector (3 downto 0);

begin

   Counter: entity work.Count4 (behavior)
      port map (CLK => CLK, RD => RD, Q => Qint);

   Decoder: entity work.Dec7S (behavior)
      port map (A => D, DP => '0', y => SEG);
```

```
   Main: process (Qint) is
   begin
      SEL(1) <= '1';        -- Aktivieren der Digits
      SEL(2) <= '0';
      SEL(3) <= '0';
      SEL(4) <= '0';

      D(1) <= Qint(3);      -- Duale Anzeige
      D(2) <= Qint(2);
      D(3) <= Qint(1);
      D(4) <= Qint(0);
   end process Main;

end architecture structure;
```

Abb. 7.43 VHDL-Programm für den Zähler Count4Hex

Die Angabe

```
use work.all;
```

bewirkt, dass andere Programme im selben Verzeichnis aufgerufen werden können.

Nach dem Speichern von Count4Hex erfolgt die Meldung, dass die Programme Count4 und Dec7S nicht gefunden werden. Sie müssen noch in das aktuelle Verzeichnis kopiert werden. Bevor sie im aktuellen Projekt verwendet werden können müssen sie mit

File > Add > Existing File

und Auswahl der entsprechenden Datei in das aktuelle Projekt übernommen werden.

Erzeugen Sie die JEDEC-Datei und programmieren Sie das PLD auf dem Demoboard. Erproben Sie die Schaltung mit dem Demoboard.

Betätigen Sie die HDL-Schaltfläche und erzeugen Sie die graphische Darstellung des Programms. In Abb. 7.44 ist das Schaltbild dargestellt, das Sie durch Auswahl von "connectivity" nach rechtem Mausklick erhalten.

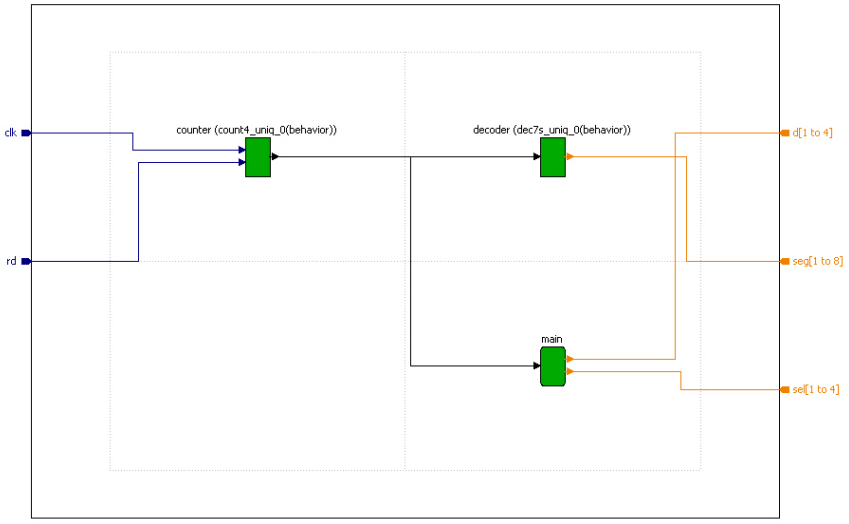


Abb. 7.44 Schaltbild von Coount4Hex nach Wahl von "connectivity"

Man erkennt die Instanz Counter, die vom Programm Count4 erstellt wurde, und die Instanz Decoder, der das Programm Dec7S zugrunde liegt. Der Prozess Main erzeugt aus dem internen Signal Qint die Signale D1, … , D4 zur Ansteuerung der LED's und die Signale SEL1, … , SEL4, die zur Aktivierung der Digits benötigt werden.

Innerhalb des Programms Count4Hex wäre es möglich, auf den Prozess Main zu verzichten. Abb 7.45 zeigt das entsprechend modifizierte Programm ohne den Prozess Main.

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

entity Count4Hex is
   port (
      CLK, RD: in std_logic;
      D: out std_logic_vector (1 to 4);
      SEL: out std_logic_vector (1 to 4);
      SEG: out std_logic_vector (1 to 8)
      );
end entity Count4Hex;


architecture structure of Count4Hex is

   signal Qint: std_logic_vector (3 downto 0);

begin

   Counter: entity work.Count4 (behavior)
      port map (CLK => CLK, RD => RD, Q => Qint);

   Decoder: entity work.Dec7S (behavior)
      port map (A => D, DP => '0', y => SEG);

   SEL(1) <= '1';       -- Aktivieren der Digits
   SEL(2) <= '0';
   SEL(3) <= '0';
   SEL(4) <= '0';

   D(1) <= Qint(3);     -- Duale Anzeige
   D(2) <= Qint(2);
   D(3) <= Qint(1);
   D(4) <= Qint(0);

end architecture structure;
```

Abb. 7.45 VHDL-Programm für den Zähler Count4Hex ohne den Prozess Main

Die Signale SEL1, … , SEL4 und D1, … , D4 werden nun unmittelbar in der Architektur structure erzeugt. Führen Sie die entsprechenden Programmänderungen durch und erzeugen Sie mit der HDL-Schaltfläche das Schaltbild durch Wahl von "connectivity". In Abb. 7.46 ist das Schaltbild dargestellt. Vorhanden sind wiederum die Instanzen Counter und Decoder. Die Erzeugung der Signale D1, … , D4 und SEL1, … , SEL4 ist nun jedoch detailliert dargestellt. In der ersten Relaisierung des Programms Count4Hex mit dem Prozess Main wurden diese Signale innerhalb des Prozesses Main erzeugt. Durch Verwendung des Prozesses Main ergibt sich eine übersichtlichere Gestaltung des Schaltbilds. Die Verwendung von Prozessen für spezifische Aufgaben ist deshalb vorzuziehen. Innerhalb einer Architektur können mehrere Prozesse definiert werden.

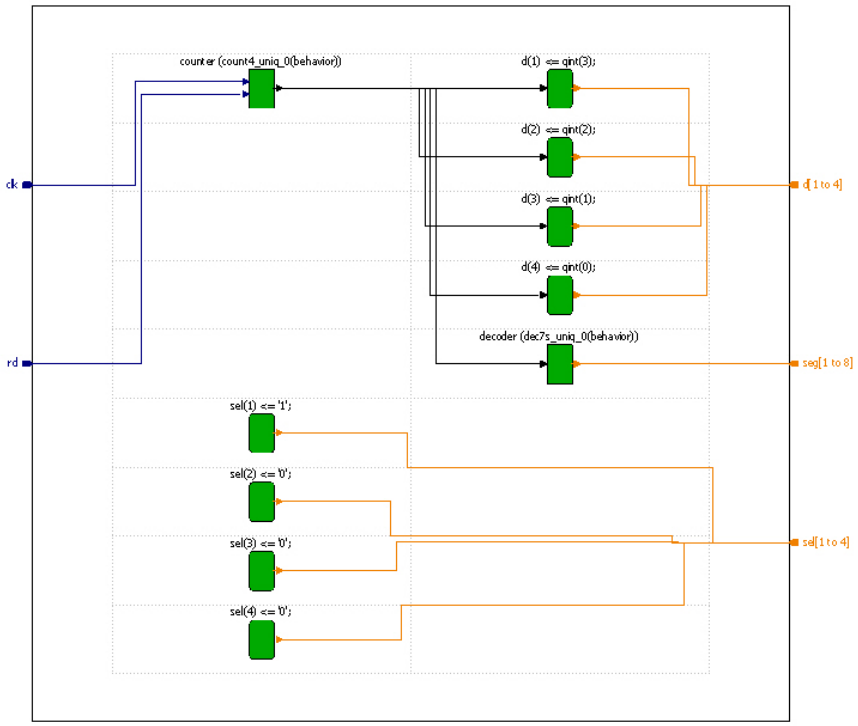


Abb. 7.46 Schaltbild für das Programm Count4Hex ohne den Prozess Main

7.2.2 4-Bit-Zähler mit automatischer Takterzeugung

Der Takt CLK des Zählers wurde bisher manuell durch Betätigung des roten Tasters S11 erzeugt. Nun soll die Erzeugung des Taktsignals CLK automatisch mit unterschiedlichen Frequenzen erfolgen. Zur Takterzeugung verfügt das PLD über einen internen Oszillator mit einer Frequenz von 2,08 MHz. Aus dieser Frequenz können mit entsprechenden Teilern andere Frequenzen erzeugt werden, z. B. 1 Hz, 3 Hz, 100 Hz, etc. In Kap. 3.4 wurde die Takterzeugung bereits beschrieben. Zur Takterzeugung muss das Modul ClkGen in das Projekt integriert werden. Das Blockschaltbild dieses Moduls ist in Abb. 7.47 dargestellt.

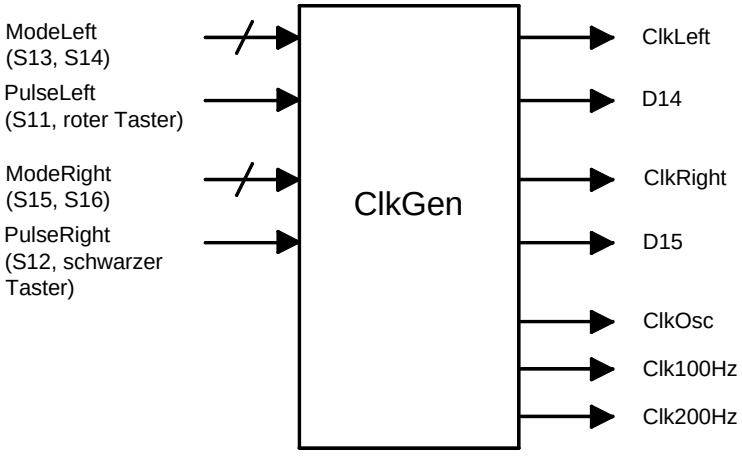


Abb. 7.47 Takterzeugung mit dem Modul ClkGen

Das Modul ClkGen ist im Programmpaket VHDL4Digilab enthalten. Die Auswahl der Frequenzen mit ModeLeft und ModeRight ist in Kap. 3.4 erläutert.

Im Folgenden wird die Verwendung des Moduls ClkGen am Beispiel des Programms Count4Hex gezeigt. Das Programm Count4Hex ist abzuändern, wie in Abb. 7.48 angegeben. Es wird der Prozess Main verwendet.

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

entity Count4Hex is
   port (
      RD: in std_logic;
      SCLK: in std_logic_vector (11 to 16);         -- Schalter fuer ClkGen
      D14, D15: out std_logic;                      -- LEDs fuer ClkGen
      D: out std_logic_vector (1 to 4);
      SEL: out std_logic_vector (1 to 4);
      SEG: out std_logic_vector (1 to 8)
      );
end entity Count4Hex;


architecture structure of Count4Hex is

   signal Qint: std_logic_vector (3 downto 0);
   signal ClkLeft, ClkRight, Clk100Hz, ClkOsc: std_logic;     -- Signale fuer ClkGen
   signal ModeLeft, ModeRight: std_logic_vector (1 downto 0);

begin

   Counter: entity work.Count4 (behavior)
      port map (CLK => ClkLeft, RD => RD, Q => Qint);

   Decoder: entity work.Dec7S (behavior)
      port map (A => D, DP => '0', y => SEG);

   Clock: entity work.ClkGen (behavior)
      port map (ClkModeLeft => ModeLeft, ClkModeRight => ModeRight, PulseLeft => SCLK(11),
         PulseRight => SCLK(12), SignalLeft => D14, SignalRight => D15,
         ClkLeft => ClkLeft, ClkRight => ClkRight, Clk100Hz => Clk100Hz,
         ClkOsc => ClkOsc);

   Main: process (Qint) is
   begin
      SEL(1) <= '1';       -- Aktivieren der Digits
      SEL(2) <= '0';
      SEL(3) <= '0';
      SEL(4) <= '0';

      D(1) <= Qint(3);     -- Duale Anzeige
      D(2) <= Qint(2);
      D(3) <= Qint(1);
      D(4) <= Qint(0);
   end process Main;

   ClockMode: process (SCLK) is
   begin
      ModeLeft <= SCLK(13) & SCLK(14);
      ModeRight <= SCLK(15) & SCLK(16);
   end process ClockMode;

end architecture structure;
```

Abb. 7.48 VHDL-Programm für den Zähler Count4Hex mit dem Modul ClkGen

Mit den Tastern S11 (rot) und S12 (schwarz) erfolgt die manuelle Impulserzeugung. Die Schalter S13, … , S16 dienen zur Einstellung von ModeLeft und ModeRight. Diese Schalter werden im Feld SCLK zusammengefasst. ModeLeft und ModeRight werden im Prozess ClockMode bestimmt. Mit dem Verkettungs-Operator & (siehe Anhang A3.14) werden die von den Schaltern erzeugten Bits zum Feld ModeLeft bzw. ModeRight zusammengefasst.

Die von ClkGen erzeugten Taktsignale sind ClkLeft, ClkRight, Clk100Hz, Clk200Hz und ClkOsc. In der Deklaration der Instanz Clock des Moduls ClkGen werden diese Signale mit den Ausgangssignalen des Moduls ClkGen verknüpft. Innerhalb dieser Deklaration werden auch die LED's D14 und D15 angegeben, mit denen die Taktsignale ClkLeft und ClkRight optisch signalisiert werden. Da das PLD nur über einen internen Oszillator verfügt, darf vom Modul ClkGen nur eine Instanz erzeugt werden.

Der interne Oszillator arbeitet mit einer Frequenz von 2,08 MHz. Der Oszillator ist ein fester Bestandteil des PLD (siehe Abb. 2.3). Zum Teilen der Oszillatorfrequenz kann der Extended Function Block (EFB) verwendet werden, der ebenfalls ein fester Bestandteil des PLD ist (siehe Abb. 2.3 und Abb. 2.6). Der EFB verfügt über einen 16 Bit-Counter/Timer (siehe Abb. 2.6). Der Oszillator und der EFB müssen so parametriert werden, dass eine geeignete Basisfrequenz erzeugt wird. Der Counter/Timer wird mit dem Teiler 5200 parametriert, so dass sich eine Basisfrequenz von 200 Hz ergibt. Zu beachten ist die zusätzliche Teilung durch 2, da im Counter/Timer-Ausgangssignal bei jedem Nulldurchgang des Counters ein Flankenwechsel erfolgt. Diese Einstellungen werden mit dem Programm IPexpress durchgeführt, das über Tools > IPexpress gestartet wird. Für das Digitallabor wurden die Einstellungen für den Oszillator und den Counter/Timer im Modul CT zusammengefasst. Die folgenden Dateien sind in die aktuelle Projektdatei zu kopieren:

CT.vhd
CT.ipx
CT.lpc

Danach müssen die Dateien mit

File > Add > Existing File

in das Projekt aufgenommen werden. Die genannten Dateien befinden sich im Programmpaket VHDL4Digilab. Durch Doppelklick auf CT.ipx wird IPexpress gestartet und die aktuellen Parameter des Moduls werden angezeigt.

Ebenfalls im Programmpaket VHDL4Digilab enthalten ist das Programm ClkGen.vhd, das in das aktuelle Projektverzeichnis kopiert werden muss. Danach ist das Programm mit File > Add > Existing File in das aktuelle Projekt aufzunehmen.

Im obigen Programm werden alle von ClkGen erzeugten Signale deklariert. Sie können durch andere Module verwendet werden. Beim Modul Counter wird dessen Taktsignal CLK mit ClkLeft verbunden. Werden nicht alle Signale benötigt, die ClkGen erzeugt, so ist ihre Deklaration nicht erforderlich. Im vorliegenden Fall wird lediglich ClkLeft benötigt. Die Deklaration der Signale ClkRight, Clk100Hz, Clk200Hz und ClkOsc wäre deshalb nicht erforderlich und die Deklaration der Instanz Clock würde wie folgt lauten:

```
Clock: entity work.ClkGen (behavior)
   port map (ClkModeLeft => ModeLeft, ClkModeRight => "00", PulseLeft => SCLK(11),
      PulseRight => '0', SignalLeft => D14, ClkLeft => ClkLeft);
```

Lediglich ClkLeft wird benötigt. Alle anderen von ClkGen erzeugten Signale sind deshalb in der Port Map nicht aufgeführt. Jedem Eingangssignal des Moduls ClkGen muss ein gültiger Wert zugewiesen werden. Eingangssignale können nicht "offen" bleiben. ClkLeft wird mit der LED D14 signalisiert.

Modifizieren Sie das Programm für den Zähler Count4Hex gemäß Abb. 7.48 und erzeugen Sie die JEDEC-Datei. Programmieren Sie das PLD auf dem Demoboard. Erproben Sie den Zähler mit der automatischen Takterzeugung. Wählen Sie mit den Schaltern S13 und S14 verschiedene Taktfrequenzen (Single Step, 0,3 Hz, 1 Hz, 3 Hz).

Hinweis: Beim Durchführen der Prozesse Place & Route und Map können Warnungen auftreten. Diese Warnungen weisen auf nicht verbundene Blöcke hin. Die Ursache hierfür ist, dass innerhalb von ClkGen Funktionsbausteine aus der Bibliothek lattice.lib verwendet werden, deren VHDL-Code nicht verfügbar ist. Die Warnungen können ignoriert werden. Eine Beeinträchtigung der Funktion des Zählers erfolgt nicht.

7.3 8-Bit-Schieberegister

Zu erstellen ist ein 8 Bit-Schieberegister gemäß Abb. 7.47. Das Taktsignal CLK wird mit dem roten Taster S11 erzeugt. Der Inhalt des Schieberegisters wird mit den LEDs D1, … , D8 zur Anzeige gebracht. Bei jeder positiven Flanke von CLK wird um eine Stelle nach rechts geschoben. Das Bit rechts außen geht verloren. Links wird das Bit SIN nachgeschoben. Mit RD = 1 wird das Schieberegister gelöscht.

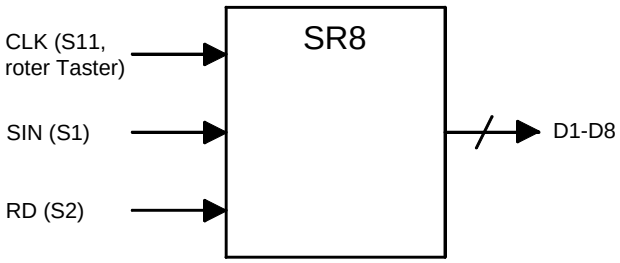


Abb. 7.49 8 Bit-Schieberegister

In Abb. 7.50 ist das Programm für das Schieberegister wiedergegeben. Es werden 8 Instanzen des D-Flipflops mit positiver Taktflankensteuerung und asynchronem Reset verwendet (DFF_PF_RD). Die Instanzen werden mit DFF0, … , DFF7 bezeichnet. SIN ist auf den D-Eingang von DFF7 geführt. Die anderen Flipflops sind entsprechend der Struktur eines Schieberegisters verknüpft. Der Ausgang von DFF7 wird mit der LED D1 angezeigt. Die LED D2 zeigt den Ausgang von DFF6 an, etc.

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

entity SR8 is
   port (
      RD, SIN, CLK: in std_logic;
      D: out std_logic_vector (1 to 8)
```

```
      );
end entity SR8;


architecture behavior of SR8 is

   signal Q: std_logic_vector (7 downto 0);

begin

   DFF0: entity work.dff_pf_rd (behavior)
      port map (CLK => CLK, RD => RD, D => Q(1), Q => Q(0));

   DFF1: entity work.dff_pf_rd (behavior)
      port map (CLK => CLK, RD => RD, D => Q(2), Q => Q(1));

   DFF2: entity work.dff_pf_rd (behavior)
      port map (CLK => CLK, RD => RD, D => Q(3), Q => Q(2));

   DFF3: entity work.dff_pf_rd (behavior)
      port map (CLK => CLK, RD => RD, D => Q(4), Q => Q(3));

   DFF4: entity work.dff_pf_rd (behavior)
      port map (CLK => CLK, RD => RD, D => Q(5), Q => Q(4));

   DFF5: entity work.dff_pf_rd (behavior)
      port map (CLK => CLK, RD => RD, D => Q(6), Q => Q(5));

   DFF6: entity work.dff_pf_rd (behavior)
      port map (CLK => CLK, RD => RD, D => Q(7), Q => Q(6));

   DFF7: entity work.dff_pf_rd (behavior)
      port map (CLK => CLK, RD => RD, D => SIN, Q => Q(7));

   SR: process (Q) is                 -- Ausgabe des SR-Inhalts
   begin
      D(1) <= Q(7);
      D(2) <= Q(6);
      D(3) <= Q(5);
      D(4) <= Q(4);
      D(5) <= Q(3);
      D(6) <= Q(2);
      D(7) <= Q(1);
      D(8) <= Q(0);
   end process SR;

end architecture behavior;
```

Abb. 7.50 Programm für das Schieberegister SR8

Erstellen Sie ein neues Projekt mit dem Namen SR8 und eine VHDL-Datei mit demselben Namen. Geben Sie das Programm gemäß Abb. 7.50 ein. Erstellen Sie eine Testbench, die 8 Impulse CLK = 1 mit einer Dauer von jeweils 100 ns erzeugt. Zwischen den Impulsen ist CLK = 0 für jeweils 100 ns. Vor Beginn der Taktung wählen Sie RD = 1 für die Dauer von 100 ns. Danach ist RD = 0. 50 ns vor der ersten positiven Flanke von CLK setzen Sie SIN = 1. 50 ns nach der ersten positiven Flanke von CLK wählen Sie SIN = 0. Überprüfen Sie anhand der Simulation die Funktionsweise des Schieberegisters. Abb. 7.51 zeigt das Timing-Diagramm.

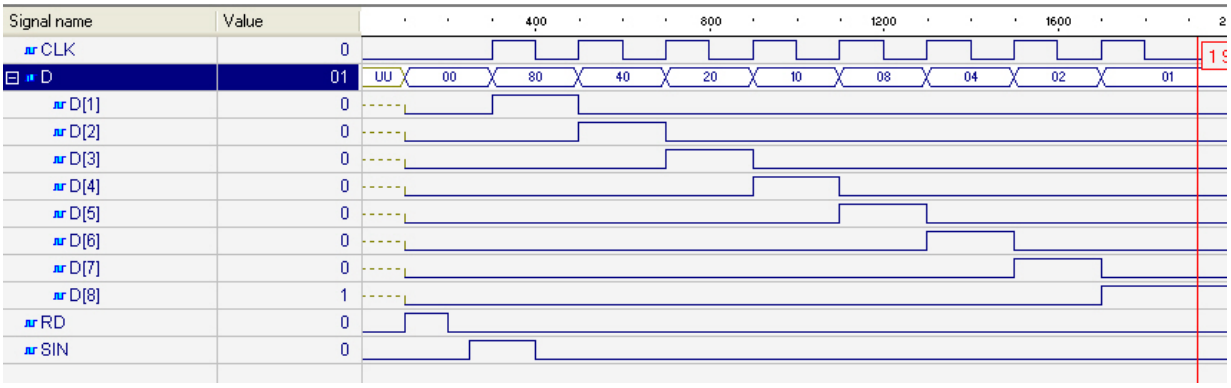


Abb. 7.51 Simulation des Schieberegisters

Erzeugen Sie die JEDEC-Datei und programmieren Sie das PLD auf dem Demoboard. Überprüfen Sie die Funktionsweise des Schieberegisters mit dem Demoboard.

VHDL bietet die Möglichkeit, ein Schieberegister kompakter zu definieren. In Abb. 7.52 ist diese Programmvariante wiedergegeben. Mit dem Verkettungs-Operator & (Anhang A3.14) wird ein Teil des Felds Q ausgeblendet und mit dem seriellen Eingang SIN verkettet.

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

entity SR8 is
   port (
      RD, SIN, CLK: in std_logic;
      D: out std_logic_vector (1 to 8)
      );
end entity SR8;


architecture behavior of SR8 is

   signal Q: std_logic_vector (7 downto 0);

begin

   SR: process (RD, CLK) is

   begin

      if RD = '1' then
         Q <= "00000000";
      else
         if CLK'event and CLK = '1' then
            Q <= SIN & Q(7 downto 1);
         end if;
      end if;

      D(1) <= Q(7);
      D(2) <= Q(6);
      D(3) <= Q(5);
      D(4) <= Q(4);
      D(5) <= Q(3);
      D(6) <= Q(2);
      D(7) <= Q(1);
      D(8) <= Q(0);
```

```
   end process SR;

end architecture behavior;
```

Abb. 7.52 Programm für das Schieberegister SR8 unter Verwendung des Verkettungsoperators &

Programmieren Sie diese Programmvariante und führen Sie die Simulation durch. Überprüfen Sie die Funktion des Schieberegisters mit dem Demoboard.

7.4 Schaltplanentwurf eines 4-Bit-Schieberegisters

Innerhalb eines Schaltplans können Bibliothekselemente der Bibliothek lattice.lib verwendet werden. Außer den einfachen Schaltfunktionen Und, Oder, Negation, etc., sind in dieser Bibliothek auch zahlreiche unterschiedliche Flipflops enthalten. Bei der Auswahl eines Bibliothekselements mit der Taste Symbol werden zu dem Bibliothekselement Informationen über dessen Funktionsweise angegeben. Allerdings sind diese Informationen sehr knapp gehalten. Eine ausführlichere Beschreibung erhält man mittels Help und Suche nach dem Namen des Bibliothekselements.

Da mit einem D-Flipflop alle Schaltwerke realisiert werden können, werden im Folgenden lediglich zwei D-Flipflops aus der Bibliothek lattice.lib näher betrachtet. Ihre Eigenschaften sind in Tab. 7.1 zusammengestellt.

Name	Funktion	Beschreibung
fd1s3ax	D-Flipflop	Positiv flankengetriggert. GSR (Clear).
fd1s3dx	D-Flipflop	Positiv flankengetriggert. GSR (Clear). Asynchroner Reset (Clear Direct, CD = 1).

Tab. 7.1 D-Flipflops aus der Bibliothek lattice.lib

Mit GSR (Global Set/Reset) wird die Eigenschaft bezeichnet, dass die Flipflops nach der Initialisierung des PLD, z. B. nach dem Einschalten der Versorgungsspannung, einen bestimmten Zustand annehmen, z. B. Q = 0 (Clear).

Zur Realisierung des Schieberegisters erstellen Sie ein Projekt mit dem Namen SR4 und erzeugen Sie einen Schaltplan mit demselben Namen. Platzieren Sie im Schaltplan 4 Instanzen des D-Flipflops fd1s3dx. Stellen Sie die für ein Schieberegister erforderlichen Verbindungen her. Bezeichnen Sie die Ausgänge der Flipflops mit Q3, … , Q0. Q3 ist links außen. Q3, … , Q0 werden mit den LED's D1, … , D4 angezeigt. Der Eingang des Schieberegisters SIN wird mit dem Schalter S1 erzeugt. Das Taktsignal CLK wird verbunden mit dem Taster S11. Das Resetsignal RES wird auf Schalter S2 geführt. Abb. 7.53 zeigt den Schaltplan des Schieberegisters.

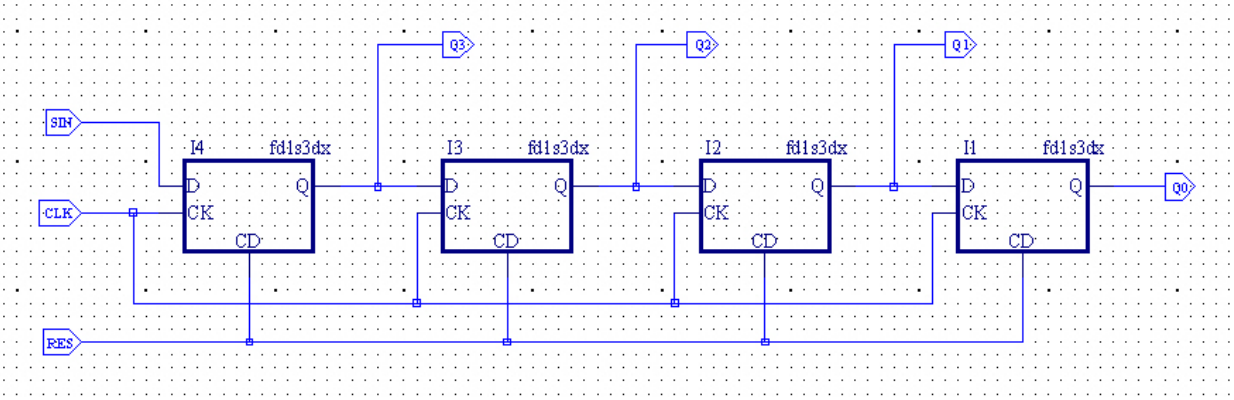


Abb. 7.53 Schaltplan des 4-Bit-Schieberegisters

Erzeugen Sie eine JEDEC-Datei und programmieren Sie das PLD des Demoboards. Überprüfen Sie die Funktion des Schieberegisters mit dem Demoboard.

Wenn Sie die HDL-Taste betätigen, erhalten Sie das Blockschaltbild des Projekts. Nach Rechtsklick auf einen Block und Auswahl von "Goto Source Definition" wird das VHDL-Programm des Schaltplans angezeigt. Es wird die Bibliothek MACHXO2 verwendet. Die Deklaration lautet:

```
library MACHXO2;
use MACHXO2.components.all;
```

Nach Angabe dieser Deklaration können Bibliothekselemente in VHDL-Programmen aufgerufen werden, ohne einen Schaltplan zu erstellen.

8 Ergänzende Themen

8.1 Multiplex-Betrieb der 7-Segment-Anzeigen

Mit den vier 7-Segment-Anzeigen DIG1, … , DIG4 können bis zu vierstellige Dezimalzahlen zur Anzeige gebracht werden. Um den PLD-Bedarf gering zu halten, erfolgt der Betrieb der Anzeigen im Multiplex-Verfahren. Identische Segmente der Anzeigen sind miteinander zu den Segmentsignalen SEG1, … , SEG8 verbunden. SEG8 dient zur Anzeige des Dezimalpunkts (DP). Mit den Signalen SEL1, … , SEL4 wird die gewünschte Anzeige aktiviert. Wenn die 4 Ziffern in rascher Folge aktualisiert werden, so nimmt der Betrachter eine statische Anzeige wahr.

In Abb. 8.1 ist die Ansteuerschaltung für die vier 7-Segment-Anzeigen dargestellt. Es wird eine 4-stellige Dezimalzahl angezeigt. Die Stellen der Dezimalzahl werden mit T (Tausender), H (Hunderter), Z (Zehner) und E (Einer) bezeichnet. Jede Ziffer liegt im 8-4-2-1-Code vor.

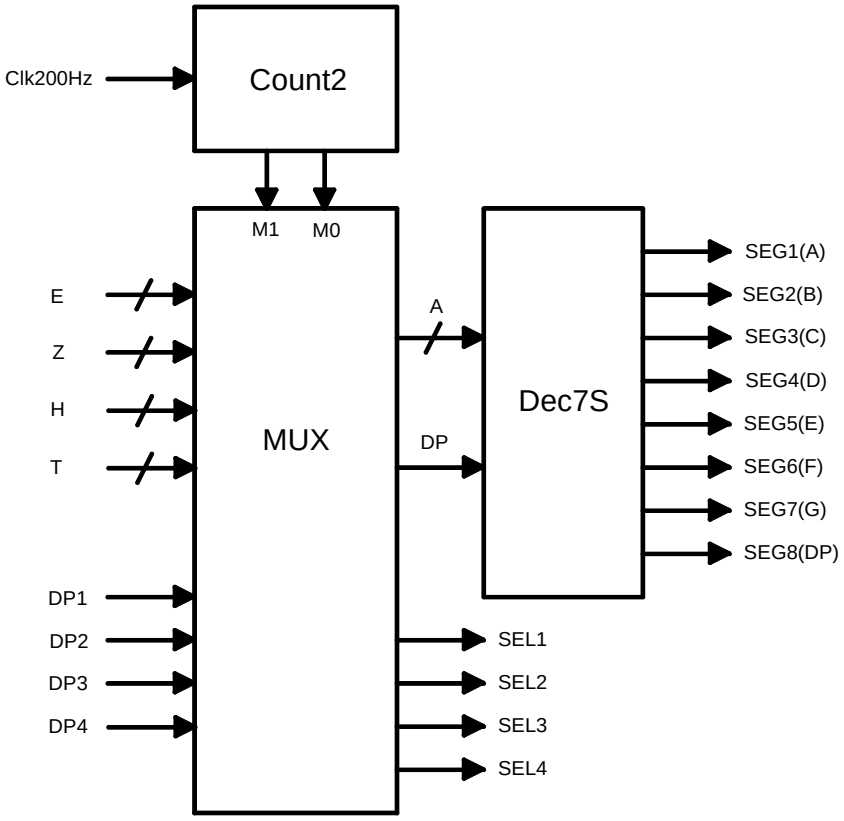


Abb. 8.1 Multiplex-Betrieb der 7-Segment-Anzeigen

Der Multiplexer schaltet in rascher Folge die Ziffern E, Z, H, T auf den Code-Umsetzer Dec7S. Die durchgeschaltete Ziffer wird bestimmt durch die Steuersignale M0 und M1 des MUX. Die Steuersignale werden erzeugt durch den 2-Bit-Zähler Count2, der durch den Takt Clk200Hz angesteuert wird. Im Fall M = 0 wird T durchgeschaltet. T muss auf der links aussen befindlichen Anzeige DIG1 angezeigt werden. Deshalb wird das Signal SEL1 aktiviert. Bei M = 1 wird H durchgeschaltet. H wird mit DIG2 angezeigt. Deshalb muss SEL2 aktiviert werden. Entsprechend wird mit Z und E verfahren. Die Erzeugung der Signale SEL1, … , SEL4 erfolgt durch den MUX

abhängig von M. Die Dezimalpunktsignale DP1, … , DP4 werden ebenfalls durch den MUX in Abhängigkeit von M umgeschaltet.

8.2 Start/Stop-Schaltungen

Im Zusammenhang mit der Realisierung von Schaltwerken kommt es häufig vor, dass ein Zähler bei einem Ereignis gestartet wird und der Zählvorgang nach Auftreten eines weiteren Ereignisses beendet wird. Im Folgenden wird angenommen, dass der Zähler über den Enable-Eingang EN verfügt. Sobald das Startsignal Start eine positive Flanke aufweist, soll EN = 1 sein. Nach Auftreten einer positiven Flanke im Stoppsignal Stop soll EN = 0 sein. Die Signale Start und Stop weisen nie gleichzeitig den Zustand 1 auf. In Abb. 8.2 ist die Realisierung der Aufgabe unter Verwendung eines RS-Flipflops dargestellt.

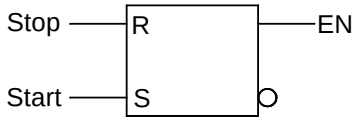


Abb. 8.2 Start/Stop mit RS-Flipflop

Bei Verwendung des RS-Flipflops ist darauf zu achten, dass der irreguläre Fall Start = Stop = 1 nie eintritt. Der irreguläre Fall darf auch nicht kurzzeitig auftreten! Nach dem Einschalten der Versorgungsspannung befindet sich das Flipflop in einem beliebigen Zustand.

Hinweis: Mit VHDL definierte RS-Flipflops werden durch den Diamond Synthese-Algorithmus unter Umständen nicht als Flipflop erkannt (siehe Kapitel 7.1.1).

Auch ein D-Flipflop kann zur Realisierung einer Start/Stop-Schaltung verwendet werden (siehe Abb. 8.3).

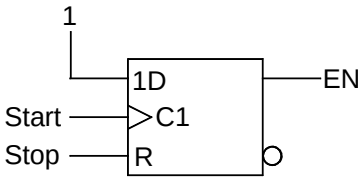


Abb. 8.3 Start/Stop mit D-Flipflop

Die positive Flanke im Signal Start bewirkt EN = 1. Mit Stop = 1 wird EN = 0 gesetzt.

Eine weitere Alternative zur Lösung der Aufgabe ist in Abb. 8.4 dargestellt. In diesem Fall wird ein T-Flipflop eingesetzt.

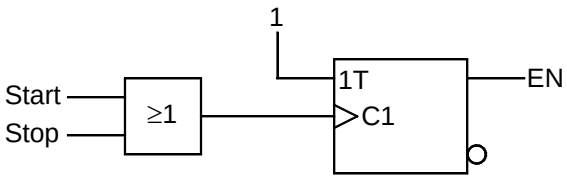


Abb. 8.4 Start/Stop mit T-Flipflop

Bei jeder positiven Flanke der Signale Start bzw. Stop wechselt EN seinen Zustand.

An der Stelle der beiden Signale Start und Stop, die mit einem Oder-Gatter verknüpft werden, kann ein einziges Signal Start/Stop zur Ansteuerung des Takteingangs in Abb. 8.4 verwendet werden.

Eine weitere Alternative zur Erzeugung des Signals EN mit einem einzigen Start/Stop-Signal ist in Abb. 8.5 dargestellt. In diesem Fall wird ein D-Flipflop eingesetzt.

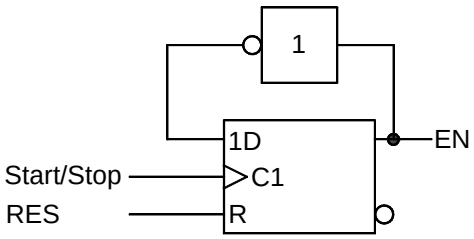


Abb. 8.5 Start/Stop-Schaltung mit einem Start/Stop-Signal

Bei jeder positiven Flanke des Signals Start/Stop wird das Signal EN gesetzt bzw. zurückgesetzt. Mit dem Signal RES ist ein asynchroner Reset möglich.

8.3 Glitch-Effekte bei Zählern

Als Glitch-Effekt bezeichnet man die Erzeugung kurzzeitiger Störimpulse oder Störzustände, die auch bei synchronen Zählern auf Grund unterschiedlicher Signallaufzeiten entstehen können. Für Glitches werden auch die Begriffe Hazards oder Spikes verwendet. Um zu zeigen, wie Glitches bei einem synchronen Zähler entstehen können, wird ein Modulo 6-Zähler betrachtet. Der Zähler durchläuft zyklisch die Zahlenfolge 0, 1, 2, 3, 4, 5, 0, 1, 2, … . Die Zustandsänderungen sollen bei positiven Flanken des Taktsignals CLK erfolgen. Wegen der vorhandenen 6 Zustände sind 3 Flipflops erforderlich. Tab. 8.1 zeigt die Zustandsfolgetabelle des Zählers.

Q2	Q1	Q0	$Q2^+$	$Q1^+$	$Q0^+$
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	0	0	0
1	1	0	x	x	x
1	1	1	x	x	x

Tab. 8.1 Zustandsfolgetabelle für den Modulo 6-Zähler

Durch Minimieren mit KV-Diagrammen erhält man die folgenden disjunktiven Minimalformen für das Übergangsschaltnetz:

$$Q2^+ = \left(Q2 \wedge \overline{Q0}\right) \vee \left(Q1 \wedge Q0\right)$$

$$Q1^+ = \left(Q1 \wedge \overline{Q0}\right) \vee \left(\overline{Q2} \wedge \overline{Q1} \wedge Q0\right)$$

$$Q0^+ = \overline{Q0}$$

Abb. 8.6 zeigt das VHDL-Programm für den Zähler unter Verwendung dieses Übergangsschaltnetzes.

```
entity CountMod6 is
   port (
      CLK, RD: in std_logic;
      CP: out std_logic;
      y: out std_logic_vector (2 downto 0)
      );
end entity CountMod6;


architecture behavior of CountMod6 is

signal Q: std_logic_vector (2 downto 0);

begin

   Count: process (RD, CLK) is

   begin
      if RD = '1' then
         Q <= "000";
      else
         if CLK'event and CLK = '1' then
            Q(2) <= ((Q(2) and not (Q(0))) or (Q(1) and Q(0))) after 20 ns;
            Q(1) <= ((Q(1) and not (Q(0))) or ((not (Q(2))) and (not (Q(1))) and Q(0))) after 10 ns;
            Q(0) <= (not (Q(0))) after 30 ns;
         end if;
      end if;
   end process Count;

   Output: process (Q) is
   begin
      if Q = "101" then
         CP <= '1';
      else
         CP <= '0';
      end if;
```

```
    y <= Q;
    end process Output;

end architecture behavior;
```

Abb. 8.6 VHDL-Programm für den Modulo 6-Zähler

Im Prozess Output wird das Signal CP erzeugt, das genau dann den Zustand 1 annehmen soll, wenn der Zählerstand 5 erreicht wird.

Es liegt ein synchrones Schaltwerk vor. Der Zustand des Zählers ändert sich bei jeder positiven Flanke des Taktsignals CLK. Die Flipflops weisen jedoch unterschiedliche Schaltzeiten auf, so dass sie nicht absolut gleichzeitig schalten. Um diesen Effekt innerhalb der Simulation deutlich zeigen zu können, wurden den Flipflops unterschiedliche Schaltzeiten zugewiesen (20, 10, 30 ns). Das Resultat der Simulation zeigt Abb. 8.7.

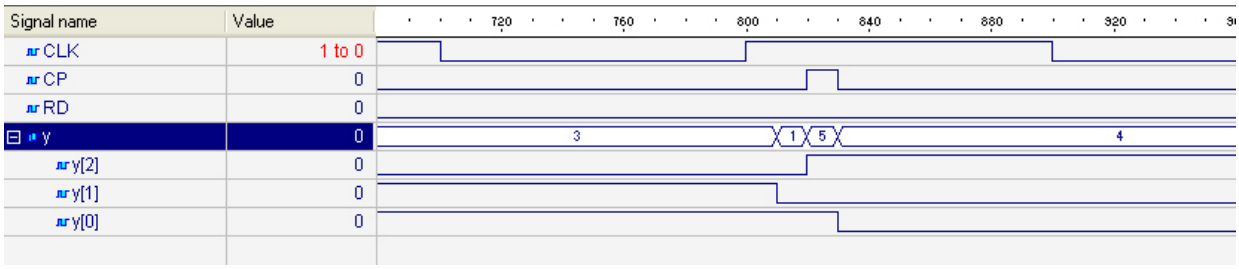


Abb. 8.7 Simulation des Modulo 6-Zählers mit Glitch-Effekten

Die Simulation zeigt, dass die gewünschte Zählfolge 0, 1, 2, 3, 4, 5, 0, 1, … durch den Glitch-Effekt bei 800 ns unterbrochen wird und kurzzeitig falsche Zählerstände eingefügt werden. So folgt auf den Zustand 3 nicht unmittelbar 4, sondern es ergibt sich die Folge 3, 1, 5, 4. Die falschen Zustände 1 und 5 werden eingefügt. In Abhängigkeit von der nachfolgenden Schaltung kann sich dies störend auswirken. Folgt z. B. ein Komparator, der auf den Zustand 5 wartet, so wird dieser Zustand bereits vor dem Auftreten von Zustand 4 erkannt. Siehe hierzu auch den nachfolgenden Abschnitt.

Im vorliegenden VHDL-Programm wurde im Prozess Output ein Komparator nachgeschaltet, der bei jeder Änderung des Zählerstands Q prüft, ob der Wert 5 vorliegt. Ist dies der Fall, so wird das Signal CP = 1 gesetzt. Auf Grund des falschen Zwischenwerts Q = 5 bei 820 ns ergibt sich kurzzeitig CP = 1.

Glitch-Effekte sind temperaturabhängig, da die Laufzeiten der Flipflops temperaturabhängig sind. In Abhängigkeit von der Temperatur können deshalb unterschiedliche Fehlzustände eingefügt werden.

Durch einen sorgfältigen Schaltungsentwurf sind Glitch-Effekte zu vermeiden. Innerhalb der Simulation können sie nicht erkannt werden, da sie abhängig von den real vorhandenen Laufzeiten der Flipflops im PLD sind. Diese Laufzeiten sind nicht nur abhängig vom PLD-Typ, sondern sie unterliegen auch beim selben PLD-Typ fertigungsbedingten Streuungen.

Hinweis: Das VHDL-Programm aus Abb. 8.6 kann in der vorliegenden Form nicht in ein PLD programmiert werden. Die definierten Verzugszeiten dienen nur der Simulation! Im PLD können die vorgegebenen Verzugszeiten nicht realisiert werden.

8.4 Frequenzteiler

Frequenzteiler dienen zur Herabsetzung von Frequenzen auf niedrigere Frequenzen. Ein Frequenzteiler 1:100 setzt z. B. die Eingangsfrequenz 100 Hz in die Ausgangsfrequenz 1 Hz um. Für den Zusammenhang zwischen den Frequenzen gilt:

$$f_{out} = \frac{f_{in}}{N}$$

Für die Periodendauern folgt

$$\frac{1}{T_{out}} = \frac{1}{NT_{in}}$$

und somit

$$T_{out} = NT_{in},$$

wobei N das Teilerverhältnis ist.

Bei der Realisierung eines Frequenzteilers mit einem Zähler ist darauf zu achten, dass das Ausgangssignal nicht nur kurzzeitig einen Impuls beim Erreichen des vorgegebenen Endwerts aufweist. Die Verarbeitung extrem kurzer Impulse durch nachfolgende Schaltungen kann Probleme verursachen. Oft wird deshalb ein Puls/Pausenverhältnis von 1 angestrebt.

Wird der Frequenzteiler gemäß Abb. 8.8 realisiert, so können die im vorigen Abschnitt beschriebenen Glitch-Effekte zu Unregelmäßigkeiten im Ausgangssignal führen. Die Periodendauer ist nicht konstant. Das Puls/Pausenverhältnis schwankt. Die Ursache hierfür liegt darin, dass der Komparator laufend den aktuellen Zählerstand Z prüft und somit auch auf Glitches reagiert.

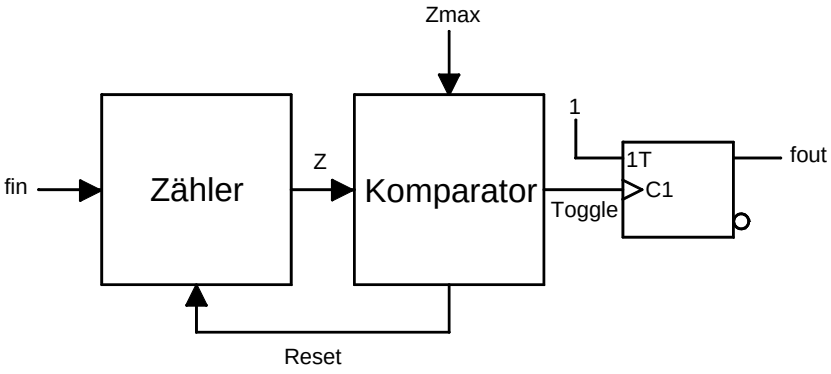


Abb. 8.8 Frequenzteiler mit Gefahr zu Störungen infolge Glitch-Effekten

Der Vergleich von Zählerstand Z und vorgegebenem Endwert Zmax muss deshalb stets in ausreichendem zeitlichem Abstand von dem Zeitpunkt erfolgen, zu dem der Zähler inkrementiert. Inkrementiert der Zähler bei einer positiven Flanke, so kann der Komparator z. B. lediglich unter der Voraussetzung CLK = 0 aktiv sein. In Abb. 8.9 ist das VHDL-Programm eines Frequenzteilers mit dem Teilerverhältnis N = 10 wiedergegeben. Das Puls/Pausenverhältnis beträgt 1. Glitch-

Effekte können bei diesem Entwurf nicht auftreten, da der Vergleich unmittelbar zum Zeitpunkt der positiven Flanke erfolgt.

```
entity FT10 is
   port(
   CLKIN, RD: in std_logic;
   CLKOUT: out std_logic
   );
end entity FT10;


architecture behavior of FT10 is

begin

   Count: process (CLKIN, RD) is

   type int16 is range 0 to 15;
   variable Q: int16;
-- variable Q: unsigned (3 downto 0);    -- alternative Definition

   begin
      if RD = '1' then
         Q := 0;
--       Q := "0000";                -- alternativ
         CLKOUT <= '0';
      else
         if (CLKIN'event and CLKIN = '1') then
            if Q < 10 then
               Q := Q + 1;
            else
               Q := 0;
--             Q := "0000";           -- alternativ
            end if;
            if Q <= 5 then
               CLKOUT <= '0';
            else
               CLKOUT <= '1';
            end if;
         end if;
      end if;
--    if Q <= 5 then                 -- Falsche Stelle fuer den Vergleich!
--       CLKOUT <= '0';
--    else
--       CLKOUT <= '1';
--    end if;
   end process Count;

end architecture behavior;
```

Abb. 8.9 VHDL-Programm für einen Frequenzteiler mit dem Teilerverhältnis N = 10

8.5 Realisierung eines Schaltwerks auf Zählerbasis

Zahlreiche Schaltwerke lassen sich unter Verwendung eines Zählers und eines Ausgangsschaltnetzes realisieren. Da der Zähler über endlich viele Zustände verfügt, spricht man auch von einer "Finite State Machine" oder kurz "State Machine". Siehe hierzu auch Kapitel 8.6.

Anhand des folgenden Beispiels wird die Realisierung eines Schaltwerks auf Zählerbasis erläutert. Der Puls P soll zyklisch erzeugt werden. In Abhängigkeit von M (Mode) soll der Puls eine unterschiedliche Länge aufweisen. Das Schaltwerk erhält die Bezeichnung PGen. Das Blockschaltbild des Schaltwerks ist in Abb. 8.10 dargestellt.

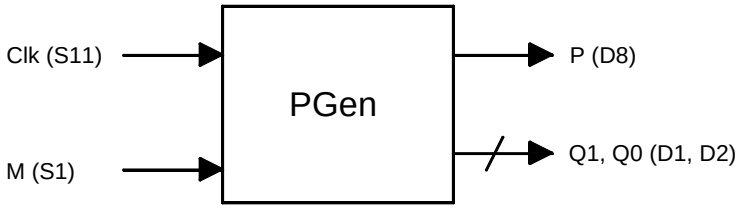


Abb. 8.10 Blockschaltbild des Schaltwerks PGen

Abb. 8.11 zeigt das Impulsdiagramm für das Schaltwerk. Im Fall M = 0 ist P = 1 nur während dem Zustand Q = 1. Ist M = 1, so ist der Impuls P = 1 während der Zustände Q = 1 und Q = 2.

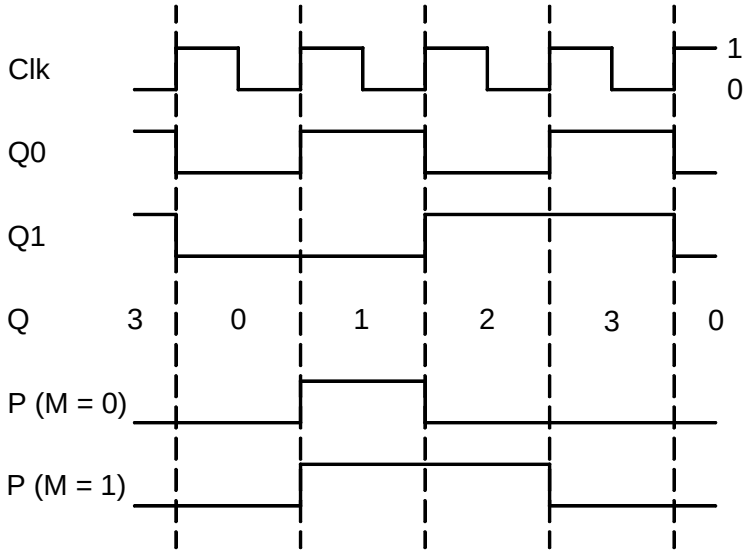


Abb. 8.11 Impulsdiagramm für das Schaltwerk PGen

In Abb. 8.12 ist das VHDL-Programm für das Schaltwerk wiedergegeben. Das Programm besteht aus einem Zähler, der zyklisch im Bereich 0, … , 3 zählt. Das Ausgangsschaltnetz ist mit einer case-Anweisung realisiert. Jedem Zustand ist ein Wert des Ausgangssignals P zugeordnet. Im Zustand Q = 2 ist das Ausgangssignal abhängig von M.

```
entity PGen is
   port(
   CLK, M: in std_logic;
   P: out std_logic;
   Q: out std_logic_vector (1 downto 0)
   );
end entity PGen;


architecture behavior of PGen is

begin

   main: process (CLK, M) is

   variable Qint: unsigned (1 downto 0);

   begin
```

```
      if (CLK'event and CLK = '1') then
         Qint := Qint + 1;
      end if;

      case Qint is
         when "00" => P <= '0';
         when "01" => P <= '1';
         when "10" =>
            if M = '0' then
               P <= '0';
            else
               P <= '1';
            end if;
         when "11" => P <= '0';
         when others => null;
      end case;

      Q <= std_logic_vector (Qint);

   end process main;

end architecture behavior;
```

Abb. 8.12 VHDL-Programm für das Schaltwerk PGen

Erstellen Sie ein neues Projekt mit dem Namen PGen und erzeugen Sie eine VHDL-Datei mit demselben Namen. Übernehmen Sie das Programm aus Abb. 8.11. Verwenden Sie die in Abb. 8.10 angegebenen Schalter und LED's. Erzeugen Sie die JEDEC-Datei und programmieren Sie das PLD des Demoboards. Erproben Sie die Funktion mit dem Demoboard.

Offenbar entscheidet im Zustand Q = 2 das Eingangssignal M darüber, welchen Zustand das Ausgangssignal P annimmt. In Abhängigkeit vom Eingangssignal M kann sich also das Ausgangssignal P ändern, obwohl sich der Zustand Q des Schaltwerks nicht verändert. Es liegt deshalb ein Mealy-Automat vor.

8.6 Finite State Machines

Schaltwerke sind dadurch gekennzeichnet, dass mehrere Zustände aufeinander folgen, wobei die Abfolge der Zustände von den Eingängen abhängig sein kann. In jedem Zustand werden den Ausgängen Werte zugeordnet. Die Werte der Ausgänge sind abhängig vom Zustand. Sie können zudem abhängig sein von den Eingängen, z. B. beim Mealy Automaten. Da derartige Schaltwerke eine endliche Zahl interner Zustände besitzen, werden sie als Finite States Machines bezeichnet.

8.6.1 Puls-Erzeugung mit Finite State Machine

Betrachtet man den Automaten aus dem vorigen Kapitel als Finite State Machine, so durchläuft er die 4 Zustände S0 (Q = 0), S1 (Q = 1), S2 (Q = 2) und S3 (Q = 3) stets in derselben zyklischen Folge. Bei Finite State Machines kann der Folgezustand jedoch abhängig vom Eingangssignal sein. Um dies zu erreichen, wird der Automat so modifiziert, dass in Abhängigkeit vom Eingangssignal M 3 oder 4 Zustände durchlaufen werden. Abb. 8.13 zeigt das Zustandsfolgediagramm in diesem Fall. Da die Finite State Machine im vorliegenden Beispiel 4 Zustände besitzt, erhält sie die Bezeichnung FSM4.

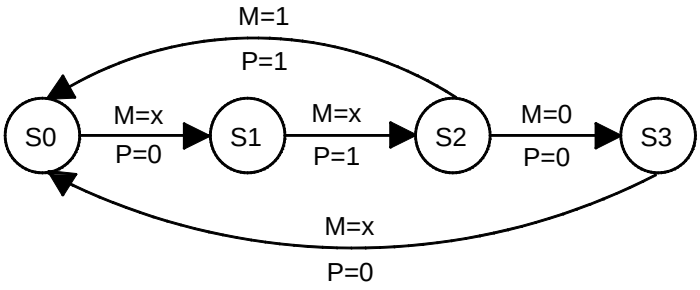


Abb. 8.13 Finite State Machine FSM4

Dadurch, dass in Abhängigkeit von M 3 oder 4 Zyklen durchlaufen werden, beträgt die Periodendauer des erzeugten Puls-Signals 3 oder 4 Clock-Zyklen. Im Fall M = 3 weist das Puls-Signal die Folge 0, 1, 1 auf, bei M = 4 wird die Folge 0, 1, 0, 0 abgegeben.

In Abb. 8.14 ist das VHDL-Programm angegeben, mit dem die Finite State Machine realisiert wird. Das Programm kann als Beispiel für die Realisierung beliebiger Finite State Machines dienen, indem es um die jeweils benötigten States und Ein- bzw. Ausgänge erweitert wird.

```
entity FSM4 is
   port(
   CLK, M: in std_logic;
   P: out std_logic;
   Q: out std_logic_vector (1 downto 0)
   );
end entity FSM4;


architecture behavior of FSM4 is

   type state_type is (S0, S1, S2, S3);
   signal state, next_state: state_type;

begin

   state_logic: process (state, M) is
   begin
      case state is
         when S0 =>
            Q <= "00";
            P <= '0';
            next_state <= S1;
         when S1 =>
            Q <= "01";
            P <= '1';
            next_state <= S2;
         when S2 =>
            Q <= "10";
            if M = '0' then
               P <= '0';
               next_state <= S3;
            else
               P <= '1';
               next_state <= S0;
            end if;
         when S3 =>
               Q <= "11";
               P <= '0';
               next_state <= S0;
         when others =>
               P <= '0';
               next_state <= S0;
      end case;
   end process state_logic;
```

```
   state_change: process (CLK) is
   begin
      if rising_edge (CLK) then
         state <= next_state;
      end if;
   end process state_change;

end architecture behavior;
```

Abb. 8.14 VHDL-Programm für die Finite State Machine FSM4

Erstellen Sie das Projekt FSM4 und erzeugen Sie die JEDEC-Datei. Programmieren Sie das PLD des Demoboards und erproben Sie die Funktion der Schaltung. Verwenden Sie die folgenden Zuordnungen zu den Schaltern und LED's:

CLK	S11
M	S1
P	D8
Q	D1 (Q1), D2 (Q0)

Der Vektor Q dient zur Anzeige des aktuellen Zustands über die LED's D1 und D2. Mit D1 wird das MSB von Q signalisiert. Mit der Anweisung `when others ...` wird erreicht, dass im Störungsfall P = 0 ausgegeben wird und der Zustand S0 folgt.

Die Zustände S0, S1, S2 und S3 müssen codiert werden. Hierzu werden mindestens 2 Bits benötigt. Beim obigen Programm hat Diamond die Zustände selbständig codiert. In welcher Form die Codierung erfolgte, ist nicht nachprüfbar.

Die Art der Codierung wirkt sich auf die Störfestigkeit des Automaten und auf den Realisierungsaufwand aus. Will man eine bestimmte Codierung erreichen, so muss die Codierung der Zustände vorgegeben werden. Die Codierung kann im Dualcode oder in einem anderen Code, z. B. im Gray-Code, erfolgen.

Im folgenden Beispiel werden die 4 Zustände mit 4 Bits im 1-aus-4-Code codiert.

```
subtype state_type is std_logic_vector (3 downto 0);
constant S0: state_type := "0001";
constant S1: state_type := "0010";
constant S2: state_type := "0100";
constant S3: state_type := "1000";
```

Diese Anweisungen werden im obigen Programm anstatt der Anweisung `type state_type ...` eingefügt.

Vergleicht man den Realisierungsaufwand für die beiden Programmvarianten, so stellt man fest, dass für die erste Variante 3 von 128 Slices erforderlich sind, was einer PLD-Ausnutzung von 2% entspricht. Bei Verwendung des 1-aus-4-Codes werden 4 von 128 Slices benötigt, entsprechend einer PLD-Ausnutzung von 3%.

Das Beispiel zeigt, dass die Art der Codierung der Zustände S0, S1, S2 und S3 einen Einfluss auf die PLD-Ausnutzung haben kann.

8.6.2 Münzwechsler als Finite State Machine

Der in der Vorlesung "Digitaltechnik" vorgestellte Münzwechsler kann als Finite State Machine realisiert werden. Es liegen die folgenden Zustände vor:

S0 Keine Schulden
S1 1 € eingeworfen
S2 2 € eingeworfen

Die Eingaben sind:

E1 1 €
E2 2 €
E3 Wechselanforderung

Die Ausgaben sind:

A1 10 x 10 Ct-Münzen
A2 20 x 10 Ct-Münzen
A3 1 € zurück
A4 2 € zurück

In Abb 8.15 ist das Zustandsfolgediagramm des Münzwechslers wiedergegeben.

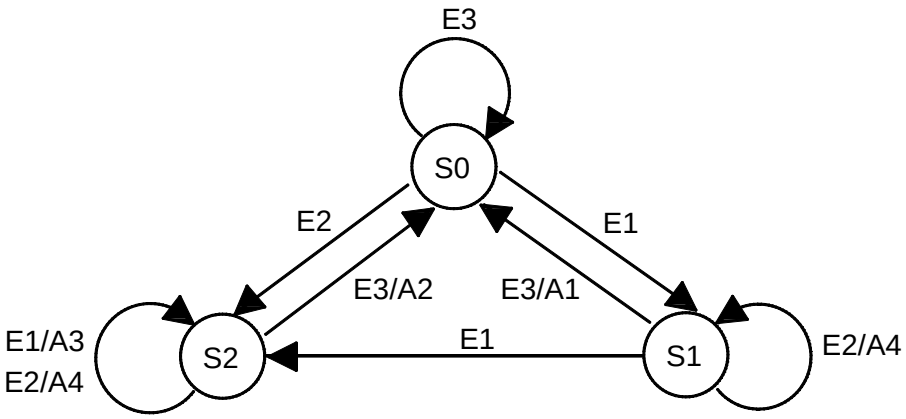


Abb. 8.15 Zustandsfolgediagramm des Münzwechslers

Zur Realisierung des Münzwechslers mit dem Demoboard werden die Eingaben mit den Schaltern erzeugt. Die Ausgaben werden mit Leuchtdioden signalisiert. In Tab. 8.1 ist die Zuordnung der Eingaben und Ausgaben zu den Schaltern und LED's des Demoboards angegeben.

E1	E2	E3	A1	A2	A3	A4	P1	P0	Q1	Q0
S1	S2	S3	D1	D2	D3	D4	D5	D6	D7	D8

Tab. 8.1 Zuordnung der Eingaben und Ausgaben zu den Schaltern und Leuchtdioden des Demoboards

Damit die interne Zustandsfolge im Automaten von außen verfolgt werden kann, wird der aktuelle Zustand mit dem Vektor Q (Bits Q1 und Q0) nach außen gegeben. Der Folgezustand wird mit dem Vektor P (Bits P1 und P0) nach außen signalisiert. Die Zuordnung zu den Zuständen S0, S1 und S2 ist Tab. 8.2 zu entnehmen.

P/Q1	P/Q0	Zustand
0	0	S0
0	1	S1
1	0	S2

Tab. 8.2 Signalisierung der Zustände und Folgezustände

Das VHDL-Programm der Finite State Machine ist in Abb. 8.16 dargestellt.

```
entity Change is
   port(
   CLK, RES: in std_logic;
   E: in std_logic_vector (3 downto 0);
   A: out std_logic_vector (4 downto 1);
   P: out std_logic_vector (1 downto 0);
   Q: out std_logic_vector (1 downto 0)
   );
end entity Change;


architecture behavior of Change is

type state_type is (S0, S1, S2);
signal state, next_state: state_type;
signal Aint: std_logic-vector (4 downto 1);   -- Merker fuer Ausgabe

begin

   state_logic: process (state, E) is
   begin
      case state is
         when S0 =>
            Q <= "00";
            if E = "001" then
               next_state <= S1;
            elsif E = "010" then
               next_state <= S2;
            elsif E = "100" then
               next_state <= S0;
            else
               next_state <= S0;
            end if;
            Aint <= "0000";
         when S1 =>
            Q <= "01";
            if E = "001" then
               next_state <= S2;
            elsif E = "010" then
               next_state <= S1;
               Aint <= "1000";          -- 2 Euro zurueck
            elsif E = "100" then
               next_state <= S0;
               Aint <= "0001";          -- 10 x 10 Ct ausgeben
            else
               next_state <= S1;
            end if;
         when S2 =>
            Q <= "10";
            if E = "001" then
               next_state <= S2;
               Aint <= "0100";          -- 1 Euro zurueck
            elsif E = "010" then
```

```
               next_state <= S2;
               Aint <= "1000";          -- 2 Euro zurueck
            elsif E = "100" then
               next_state <= S0;
               Aint <= "0010";          -- 20 x 10 Ct ausgeben
            else
               next_state <= S2;
               Aint <= "0000";          -- keine Eingabe -> keine Ausgabe
            end if;
         when others =>
            Q <= "11";
            next_state <= S0;
            Aint <= "0000";
      end case;
      if next_state = S0 then
         P <= "00";
      elsif next_state = S1 then
         P <= "01";
      elsif next_state = S2 then
         P <= "10";
      else
         P <= "11";
      end if;
   end process state_logic;

   state_change: process (CLK, RES) is
   begin
      if RES = '1' then
         state <= S0;
      else
         if falling_edge (CLK) then
            state <= next_state;
         end if;
      end if;
      if CLK = '1' then
         A <= Aint;                   -- Ausgabe nur bei CLK = 1
      else
         A <= "0000";
      end if;
   end process state_change;

end architecture behavior;
```

Abb. 8.16 VHDL-Programm des Münzwechslers mit Resetsignal RES

Das Programm für den Münzwechsler ist ähnlich aufgebaut wie das Programm für die Finite State Machine zur Puls-Erzeugung (FSM4, Kap. 8.6.1). Der Prozess state_change wurde allerdings modifiziert. Würde man den Prozess state_change aus Kap. 8.6.1 auf den vorliegenden Fall übertragen, so würde sich das nachfolgende Programm ergeben:

```
   state_change: process (CLK) is
   begin
      if falling_edge (CLK) then
         state <= next_state;
      end if;
      if CLK = '1' then
         A <= Aint;                   -- Ausgabe nur bei CLK = 1
      else
         A <= "0000";
      end if;
   end process state_change;
```

Abb. 8.17 Prozess state_change ohne Berücksichtigung eines unzulässigen Anfangszustands

Der Automat mit dem Prozess state_change nach Abb. 8.17 ist jedoch nicht lauffähig, da nach dem Einschalten der Versorgungsspannung das Signal state keinen der zulässigen Zustände S0, S1 oder S2 annimmt. Aus diesem Grund wird dem Signal next_state ebenfalls kein zulässiger Zustand zugeordnet. Der Automat startet nicht. Der Prozess state_change muss deshalb so verändert werden, dass nach dem Einschalten der Versorgungsspannung oder unmittelbar nach dem Programmieren des Demoboards das Signal state einen zulässigen Zustand annimmt. Aus diesem Grund wird das Eingangssignal RES verwendet, das mit dem Schalter S8 erzeugt wird. Im Fall RES = 1 wird das Signal state mit S0 initialisiert. Danach arbeitet der Automat mit RES = 0 wie vorgesehen.

In der Praxis ist eine Betätigung des Schalters S8 allerdings überhaupt nicht erforderlich. Er verbleibt in der Position RES = 0. Die Lattice Synthesis Engine (LSE) erkennt beim Übersetzen des Programms, dass ein asynchrones Reset-Signal vorhanden ist und bewirkt deshalb eine Initalisierung des Signals state mit dem Zustand S0 durch Anwendung des Global Set/Reset-Mechanismus (GSR).

Alternativ kann der definierte Anfangszustand state = S0 auch mit dem in Abb. 8.18 dargestellten Prozess erreicht werden.

```
state_change: process (CLK) is
begin
   if falling_edge (CLK) then
      if (state /= S0) and (state /= S1) and (state /= S2) then
         state <= S0;
      else
         state <= next_state;
      end if;
   end if;
   if CLK = '1' then
      A <= Aint;                    -- Ausgabe nur bei CLK = 1
   else
      A <= "0000";
   end if;
end process state_change;
```

Abb. 8.18 Alternative Erzeugung des Anfangszustands S0

Bei Betätigung des Tasters wird der Automat in den Zustand S0 versetzt, wenn das Signal state keinen zulässigen Zustand aufweist. Ist dies z. B. nach dem Einschalten des Demoboards der Fall, so beginnt der Automat nach der erstmaligen Betätigung des Tasters mit dem Zustand S0.

8.7 Effiziente Realisierung von PLD-Projekten

Ein PLD-Projekt ist so zu realisieren, dass der Entwurf in das PLD passt. Das Projekt kann lediglich über die im PLD vorhandene Anzahl von LUT's und Flipflops verfügen. Im Fall des PLD's XO2-256 sind 128 Slices vorhanden. Sind mehr Slices für ein Projekt erforderlich, so ist ein leistungsfähigeres PLD auszuwählen, was mit höheren Kosten verbunden ist.

VHDL bietet die Möglichkeit zur Verwendung zahlreicher unterschiedlicher Datentypen. Eigene Datentypen können definiert werden. Zahlreiche verschiedene Realsierungen eines Projekts sind möglich. Hierbei zeigen sich i. a. erhebliche Unterschiede im Realisierungsaufwand.

Wird innerhalb eines Programms die Variable Aint benötigt und soll diese Variable lediglich Werte im Bereich -8, … , 7 annehmen, so kann Aint z. B. wie folgt alternativ definiert werden:

```
variable Aint: signed (3 downto 0);

variable Aint: integer;

type int16k is range -8 to 7;
variable Aint: int16k;
```

Bei der ersten und der letzten Definition von Aint werden für die Variable 4 Bit reserviert. Wird jedoch Aint vom Typ Integer definiert, so sind wesentlich mehr Bits zur Speicherung erforderlich. Gemäß VHDL-Standard werden für Integer-Variable 32 Bit reserviert. Bei allzu großzügigem Umgang mit dem Datentyp Integer sind die Möglichkeiten eines PLD's deshalb rasch erschöpft. Am Beispiel des Projekts ClkGenTest soll dies gezeigt werden. Das Projekt verwendet das Modul ClkGen. Erzeugen Sie ein neues Projekt mit dem Namen ClkGenTest und einen Schaltplan mit demselben Namen. Kopieren Sie die zu ClkGen gehörigen Dateien in das Projektverzeichnis und fügen Sie diese Dateien mit File > Add zum Projekt hinzu.

Erzeugen Sie im Schaltplan eine Instanz von ClkGen und schließen Sie an die Eingänge des Moduls die Signale ClkModeLeft [1:0], ClkModeRight [1:0], PulseLeft und PulseRight an. Ordnen Sie den Ausgängen SignalLeft und SignalRight Ausgangssignale mit denselben Namen zu. Abb. 8.13 zeigt das Schaltbild.

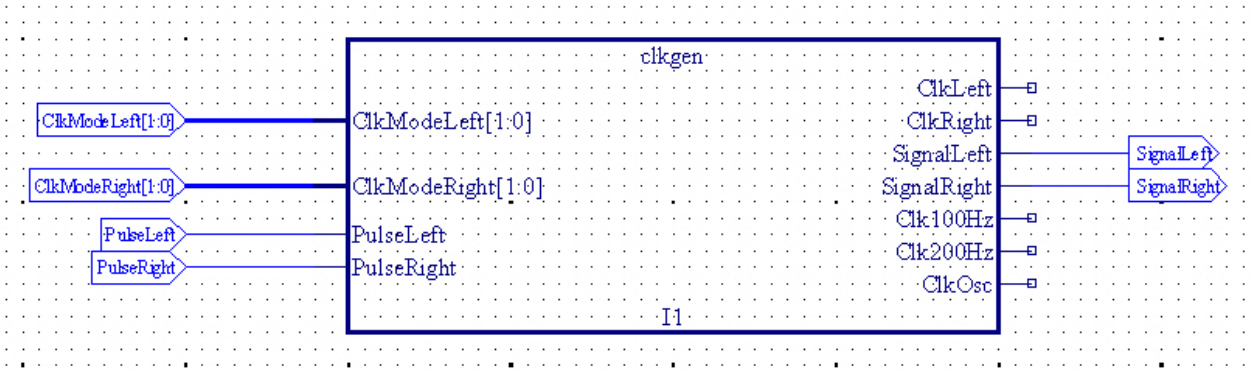


Abb. 8.13 Schaltbild des Projekts ClkGenTest

Definieren Sie die Schalter und LED's mit dem Spreadsheet und erzeugen Sie die JEDEC-Datei. Programmieren Sie das Demoboard und überprüfen Sie die Funktion der Schaltung mit dem Demoboard. Im Report von Place and Route ist die PLD-Ausnutzung angegeben. Es werden 23 von 128 Slices verwendet. Die Ausnutzung beträgt also 17%.

Verändern Sie nun den Typ der Variablen Counter03Hz im Programm ClkGen.vhd von int_range_334 auf integer. Erzeugen Sie die JEDEC-Datei und erproben Sie die Funktion mit dem Demoboard. Die Takterzeugung erfolgt weiterhin wie gewünscht. Im Report von Place and Route finden Sie nun jedoch eine PLD-Ausnutzung von 51 von 128 Slices, d. h. 39%. Die PLD-Ausnutzung hat sich mehr als verdoppelt, nur weil für eine Variable der volle Integer-Bereich (32 Bit) verwendet wird.

Machen Sie die Änderung des Datentyps der Variablen Counter03Hz rückgängig und weisen Sie der Variablen wieder den Datentyp int_range_334 zu.

9 Laborübungen

9.1 Laborübung 1

9.1.1 Teil 1: Bildung des Zweierkomplements einer 4 Bit-Zahl

Die Zahl A besteht aus den Bits A3, … , A0, die mit den Schaltern S1, … , S4 eingegeben werden. A3 ist das MSB (S1). Es soll das Zweierkomplement von A gebildet werden. Das Zweierkomplement Z setzt sich aus den Bits Z3, … , Z0 zusammen. Z3, … , Z0 werden mit den LEDs D1, … , D4 angezeigt. Z3 ist das MSB (D1). Erstellen Sie das VHDL-Programm und überprüfen Sie es mit dem Simulator. Programmieren Sie das PLD auf dem Demoboard und weisen Sie die Funktion der Schaltung nach. Die folgende Abbildung zeigt das Blockschaltbild der Schaltung.

Abb. 9.1 Zweierkomplement-Umsetzer

9.1.2 Teil 2: Vergleicher für zwei positive 4 Bit-Zahlen

Erstellen Sie einen Vergleicher für die beiden positiven 4 Bit-Zahlen A und B. A besteht aus den Bits A3, … , A0, die mit den Schaltern S1, … , S4 erzeugt werden. A3 ist das MSB (S1). B besteht aus den Bits B3, … , B0, die mit den Schaltern S5, … , S8 erzeugt werden. B3 ist das MSB (S5). Erzeugen Sie die Signale GR (Greater), EQ (Equal) und LE (Less). Die Signale haben die folgende Bedeutung:

A > B: GR = 1
A = B: EQ = 1
A < B: LE = 1

Es ist immer nur ein Signal aktiv. Die Ausgabe der Signale erfolgt mit den LEDs D1, D2 und D3 (D1 = GR, D2 = EQ, D3 = LE).

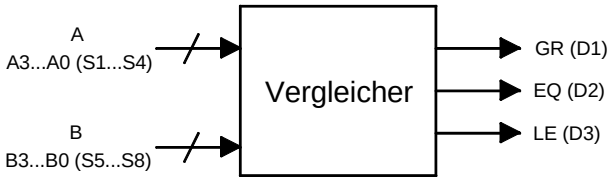


Abb. 9.2 Vergleicher für zwei 4 Bit-Zahlen

Erstellen Sie ein VHDL-Programm. Simulieren Sie das Programm. Programmieren Sie das PLD auf dem Demoboard und weisen Sie die Funktion der Schaltung nach.

9.1.3 Teil 3: Vergleicher für positive und negative 4 Bit-Zahlen

Ändern Sie den Vergleicher so, dass die 4 Bit-Zahlen sowohl positiv als auch negativ sein können. Zur Darstellung der negativen Zahlen wird das Zweierkomplement verwendet. Die Zahlen überstreichen also den Wertebereich -8 … 7. Erstellen Sie auch hierfür ein VHDL-Programm. Führen Sie die Simulation durch und programmieren Sie das PLD auf dem Demoboard. Weisen Sie die Funktion der Schaltung nach.

9.2 Laborübung 2

9.2.1 Teil 1: Addierer für zwei 3 Bit-Zahlen

Die 3 Bit-Zahlen A und B werden im Zweierkomplement dargestellt. A besteht aus den Bits A2, … , A0. A überstreicht den Zahlenbereich -4 … +3. B besteht aus den Bits B2, … , B0. Die Bits A2, … , A0 werden mit den Schaltern S1, … , S3 erzeugt. A2 ist das MSB (S1). Die Bits B2, … , B0 werden mit den Schaltern S4, … , S6 erzeugt. B2 ist das MSB (S4). Das Resultat der Addition ist die Zahl Z, die ebenfalls im Zweierkomplement dargestellt wird. Z besteht aus den Bits Z3, … , Z0, die mit den LEDs D1, … , D4 angezeigt werden. Z3 ist das MSB (D1). Erstellen Sie ein VHDL-Programm. Führen Sie die Simulation durch. Programmieren Sie das PLD im Demoboard und weisen Sie die Funktion der Schaltung nach.

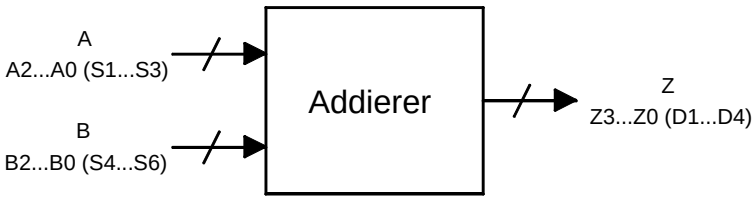


Abb. 9.3 Addierer für zwei 3 Bit-Zahlen

9.2.2 Teil 2: Subtrahierer für zwei 3 Bit-Zahlen

Anstatt der Addition aus dem vorigen Teil soll nun eine Subtraktion der beiden Zahlen erfolgen.

9.2.3 Teil 3: Multiplizierer für zwei 3 Bit-Zahlen

Anstatt der Addition aus Teil 1 soll nun eine Multiplikation ausgeführt werden. Im Fall einer Bereichsüberschreitung wird bei einem positiven Resultat die maximal mit Z darstellbare positive Zahl angezeigt. Ist das Resultat bei einer Bereichsüberschreitung negativ, so wird die minimal darstellbare negative Zahl angezeigt.

9.2.4 Teil 4: Dividierer für zwei 3 Bit-Zahlen

Anstatt der Addition aus Teil 1 soll eine Division ausgeführt werden. Im Fall einer Division mit 0 wird die maximal darstellbare positive Zahl angezeigt.

9.2.5 Teil 5: Rechenwerk für zwei 3 Bit-Zahlen

Die in den vorangehenden Teilen erstellten Schaltungen sollen zu einem Rechenwerk zusammengefasst werden. Mit den Steuereingängen C0 und C1 wird die Funktion des Rechenwerks gesteuert. C0 und C1 werden mit S9 und S10 erzeugt. Es soll gelten:

C1 (S9)	C0 (S10)	Funktion
0	0	Addierer
0	1	Subtrahierer
1	0	Multiplizierer
1	1	Dividierer

Tab. 9.1 Steuersignale C0 und C1

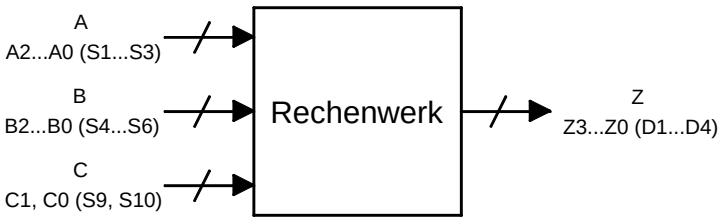


Abb. 9.4 Rechenwerk

Verwenden Sie die Schaltungen aus den vorigen Teilen. Realisieren Sie einen Multiplexer, der durch C0 und C1 gesteuert wird. Erstellen Sie die VHDL-Programme und führen Sie eine Simulation durch. Programmieren Sie das PLD im Demoboard und weisen Sie die Funktion der Schaltung nach.

9.3 Laborübung 3

9.3.1 Teil 1: Hamming Codierer (Coder)

Die 5 Bit-Dualzahl A soll mittels Hamming-Codierung so gesichert werden, dass sie ein-Fehler-korrigierbar ist. Das Datenwort A besteht aus den Bits A0, … , A4. A0 ist das LSB, A4 ist das MSB. A wird mit den Schaltern S1, … , S5 erzeugt (S1 = A4, … , S5 = A0). Es werden 4 Kontrollbits benötigt. Die Kontrollbits werden mit K0, … , K3 bezeichnet. K0 ist das LSB, K3 ist das MSB. Die Kontrollbits werden mit den LEDs D1, … , D4 signalisiert. Tab. 9.2 zeigt die Bit-Zuordnungen im Überblick.

Datenbits (Infobits)					Kontrollbits			
A4	A3	A2	A1	A0	K3	K2	K1	K0
S1	S2	S3	S4	S5	D1	D2	D3	D4
x1	x2	x3	x4	x5	y1	y2	y3	y4

Tab. 9.2 Bit-Zuordnungen

Erstellen Sie ein Schaltnetz gemäß dem nebenstehenden Blockschaltbild, das die Kontrollbits erzeugt.

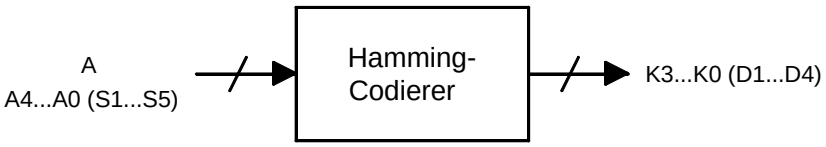


Abb. 9.5 Hamming-Codierer

Bestimmen Sie die Kontrollbits zu den in Tab. 9.3 angegebenen Infobits und vergleichen Sie diese mit den theoretisch ermittelten Werten.

Infobits						Kontrollbits			
Hex.	A4	A3	A2	A1	A0	K3	K2	K1	K0
17H	1	0	1	1	1				
0DH	0	1	1	0	1				
1CH	1	1	1	0	0				
0EH	0	1	1	1	0				
15H	1	0	1	0	1				

Tab. 9.3 Zu übertragende Infobits mit den Kontrollbits

9.3.2 Teil 2: Hamming-Decodierer (Decoder)

Entwerfen Sie ein Schaltnetz für den Empfänger der in Teil 1 codierten Daten. Der Empfänger bildet aus den empfangenen Infobits A0e, … , A4e erneut die Kontrollbits K0, … , K3 und vergleicht diese mit den empfangenen Kontrollbits K0e, … , K3e. Durch diesen Vergleich stellt der Empfänger fest, ob ein Bit falsch übertragen wurde und welches Bit ggf. korrigiert werden muss. Mit den LEDs D1, … , D5 wird das ggf. korrigierte Datenwort angezeigt (D1 = A4k, … , D5 = A0k). D6 leuchtet, wenn überhaupt ein Übertragungsfehler vorliegt (D6 = UEF). D7 leuchtet zusätzlich, wenn eines der Kontrollbits falsch übertragen wurde (D7 = KF). Die nebenstehende Abbildung zeigt das Blockschaltbild des Schaltnetzes.

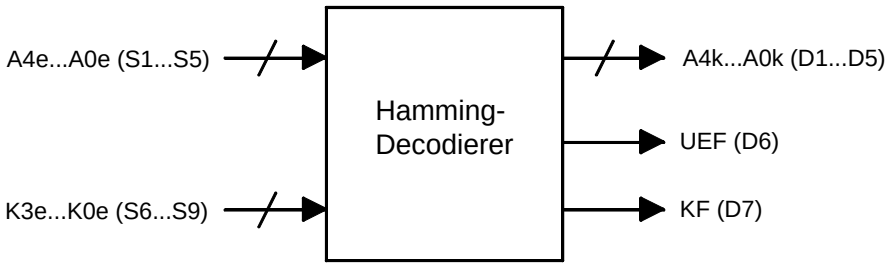


Abb. 9.6 Hamming-Decodierer

Die empfangenen Bits werden mit den Schaltern S1, … , S9 eingegeben. Verwenden Sie die folgenden Zuordnungen:

A4e	A3e	A2e	A1e	A0e	K3e	K2e	K1e	K0e
S1	S2	S3	S4	S5	S6	S7	S8	S9

Tab. 9.4 Bit-Zuordnungen

Testen Sie die Funktion des Hamming-Decodierers mit den in Tab. 9.5 angegebenen empfangenen Bits und bestimmen Sie die korrekten Datenworte.

Hex.	A4e	A3e	A2e	A1e	A0e	K3e	K2e	K1e	K0e	A4k	A3k	A2k	A1k	A0k	UEF	KF
	S1	S2	S3	S4	S5	S6	S7	S8	S9	D1	D2	D3	D4	D5	D6	D7
16DH	1	0	1	1	0	1	1	0	1							
52H	0	0	1	0	1	0	0	1	0							
13BH	1	0	0	1	1	1	0	1	1							
0DCH	0	1	1	0	1	1	1	0	0							
1C9H	1	1	1	0	0	1	0	0	1							

Tab. 9.5 Empfangene Bits und korrigiertes Datenwort

Definieren Sie Ihre Schaltung mit Hifle der Schaltplaneingabe. Gliedern Sie die Aufgabe in mehrere Funktionsblöcke und füllen Sie die Funktionsblöcke mit VHDL-Programmen. Für den Teil 2 der Aufgabe kann der in Teil 1 erstellte Funktionsblock „Hamming-Codierer“ wieder verwendet werden.

Führen Sie eine Simulation durch und programmieren Sie das PLD auf dem Demoboard. Weisen Sie die Funktion der Schaltung mit dem Demoboard nach.

9.4 Laborübung 4

9.4.1 Teil 1: Lauflicht

Erstellen Sie ein Schaltwerk, das mit den Dioden D1, … , D8 ein Lauflicht realisiert. Es soll eine Diode leuchten. Verwenden Sie als Taktsignal den Takt ClkLeft, der mit dem Modul ClkGen erzeugt wird. Bei jeder positiven Taktflanke von ClkLeft wird die leuchtende LED um eine Stelle nach rechts bewegt. Nach der LED D8 folgt die LED D1. Testen Sie die Funktion mit dem Demoboard. Erzeugen Sie die Taktflanken zunächst mit dem roten Taster. Wählen Sie danach verschiedene Frequenzen. Erstellen Sie einen Schaltplan mit Funktionsblöcken, deren Funktion mit VHDL definiert wird.

9.4.2 Teil 2: Lauflicht mit wählbarer Laufrichtung

Modifizieren Sie das Schaltwerk aus Teil 1 so, dass die Laufrichtung abhängig vom Signal DIR ist. Es soll gelten:

DIR = 0: Laufrichtung nach rechts
DIR = 1: Laufrichtung nach links

Erzeugen Sie DIR mit dem Schalter S1. Testen Sie die Funktion mit dem Demoboard.

9.4.3 Teil 3: Lauflicht mit wählbarer Laufrichtung und optional inverser Darstellung

Ergänzen Sie das Schaltwerk aus Teil 2 so, dass das Lauflicht in Abhängigkeit vom Signal INV entweder wie bisher oder invers realisiert wird. Bei inverser Darstellung bleibt eine LED dunkel, alle anderen leuchten. Die dunkle LED bewegt sich nach links bzw. nach rechts. Für INV soll gelten:

INV = 0: Normale Darstellung
INV = 1: Inverse Darstellung

Erzeugen Sie INV mit dem Schalter S2. Testen Sie die Funktion mit dem Demoboard. Achten Sie darauf, dass die LED's beim Umschalten von INV bzw. DIR nicht sprungartig ihre Position verändern!

9.4.4 Teil 4: 4 Bit-Zähler

Entwerfen Sie einen Binärzähler für 4 Binärstellen. Erstellen Sie einen Funktionsblock mit den in Abb. 9.7 angegebenen Ein/Ausgängen.

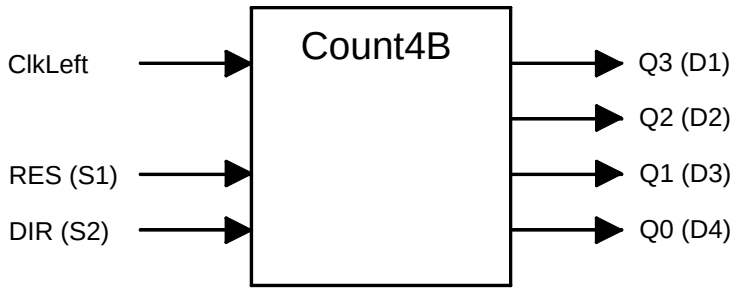


Abb. 9.7 4 Bit-Binärzähler

Die Signale RES und DIR sollen die folgenden Funktionen besitzen:

RES = 0: zählen
RES = 1: Reset (Q = 0)

DIR = 0: aufwärts zählen
DIR = 1: abwärts zählen

Der Zählvorgang soll "Turn around" erfolgen, d. h. wird beim Aufwärtszählen die größe Zahl 0FH erreicht, so ist der nächste Schritt Q = 0. Beim Abwärtszählen folgt nach Q = 0 die größte Zahl 0FH. Entwerfen Sie den Zähler als Funktionsblock und erstellen Sie einen Schaltplan, in dem der Funktionsblock verwendet wird.

9.5 Laborübung 5

9.5.1 Teil 1: Ampelsteuerung

Realisieren Sie einen Automaten zur Ansteuerung einer Fahrzeugampel und einer Fußgängerampel. Die LED's für die beiden Ampeln befinden sich auf dem Demoboard auf der rechten Seite. Im Ruhezustand leuchten die Fahrzeugampel grün und die Fußgängerampel rot. Will ein Fußgänger die Fahrbahn überqueren, so betätigt er die Anforderungstaste S11 (roter Taster). Danach erfolgen folgende Schritte:

Fahrzeug-Ampel	Fußgänger-Ampel	Dauer [s]	Phase
grün	rot	unbegrenzt	Ruhe
gelb	rot	2	Nach Anf. für Überq.
rot	rot	2	
rot	grün	5	Überquerung
rot	rot	2	Nach Überquerung
rot + gelb	rot	2	
grün	rot	unbegrenzt	Ruhe

Tab. 9.6 Ampelphasen

Realisieren Sie den Automaten als Zähler mit einem Ausgangsschaltnetz. Verwenden Sie als Taktsignal ClkRight in der Betriebsart 1 Hz. Zur Erprobung der Schaltung können Sie mit dem schwarzen Taster S12 einzelne Impulse erzeugen. Achten Sie beim Entwurf der Schaltung darauf, dass auch sehr kurze Betätigungen des Anforderungstasters erkannt werden.

Realisieren Sie den Automaten sowohl als Mealy- als auch als Moore-Automaten. Wie wirken sich die unterschiedlichen Automatentypen aus? Welcher Automatentyp ist zur Realisierung einer Ampelsteuerung ungeeignet?

9.5.2 Teil 2: Ansteuerung einer 7-Segment-Anzeige

Mit der 7-Segment-Anzeige DIG4 (ganz rechts auf dem Demoboard) sollen die Ziffern 0, … , 0FH angezeigt werden. Die Eingabe der Ziffern erfolgt im Dualcode. Erzeugen Sie die Bits A3, … , A0 des Dualcodes mit den Schaltern S1, … , S4. S1 ist das MSB. Verwenden Sie zur Umsetzung vom Dualcode in den 7-Segment-Code das Modul Dec7S.

9.5.3 Teil 3: Aufwärts/Abwärts-Zähler mit 7-Segment-Anzeige

Im Teil 4 der Laborübung 4 wurde ein 4-Bit-Binärzähler entwickelt. Verknüpfen Sie den Binärzähler mit der im vorigen Teil entworfenen Ansteuerung der 7-Segment-Anzeige DIG4. Programmieren Sie das PLD des Demoboards und weisen Sie die Funktion der Schaltung nach.

9.6 Laborübung 6

9.6.1 Teil 1: 4-Dekaden-Zähler

Entwerfen Sie einen 4-Dekaden-Zähler, dessen Zählerstand mit den 7-Segment-Anzeigen DIG1, … , DIG4 angezeigt wird. Der Zähler soll die Eingänge RES und EN besitzen. Es gelten die folgenden Zuordnungen:

RES = 0: Zählen (wenn EN = 1)
RES = 1: Reset (Anzeige = 0000)

EN = 0: Kein Zählen
EN = 1: Zählen

Verwenden Sie als Takt ClkRight. Erzeugen Sie RES mit dem Schalter S1 und EN mit dem Schalter S2. Die 7-Segment-Anzeigen DIG1, … , DIG4 werden im Multiplex betrieben (siehe Abschnitt 8.1).

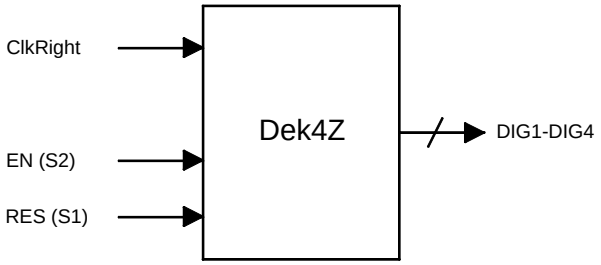


Abb. 9.8 Blockschaltbild des 4-Dekaden-Zählers

Verwenden Sie 4 Instanzen des Funktionsblocks Dek1Z (Zähler für eine Dekade) zum Aufbau des 4-Dekaden-Zählers. Die folgende Abbildung zeigt das Blockschaltbild von Dek1Z.

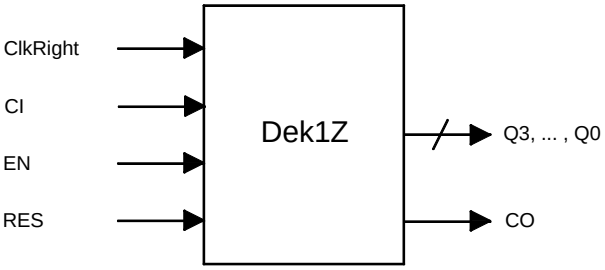


Abb. 9.9 Blockschaltbild des Ein-Dekaden-Zählers

EN und RES besitzen dieselbe Funktion wie oben. Der Ausgang CO (Carry Out) ist aktiv, wenn der Zähler den Stand 9 erreicht und CI = 1 (CI = Carry In) ist.

Programmieren Sie den Zähler in das PLD des Demoboards und überprüfen Sie die Funktion der Schaltung.

9.6.2 Teil 2: Stoppuhr

Wenn Sie den Dekadenzähler aus Teil 1 mit ClkRight = 100 Hz takten, inkrementiert der Zähler im Abstand von 10 ms. Aktivieren Sie den Dezimalpunkt in der Mitte der vier 7-Segment-Anzeigen, so dass er Sekunden und Millisekunden trennt. Sie haben nun eine Stoppuhr. Start und Stopp der Stoppuhr erfolgen mit dem Schalter S2 (EN).

Start und Stopp der Stoppuhr sollen mit dem roten Taster S11 erfolgen. Ergänzen Sie die vorhandene Schaltung so, dass das Signal EN durch Betätigen des roten Tasters aktiviert bzw. inaktiviert wird.

9.6.3 Teil 3: Messung der Fahrtzeit bei Modellautos

Zur Messung der Fahrtzeit von Modellautos ist ein U-Profil von ca. 1 m Länge vorhanden. Nach dem Start des Autos durchläuft es eine Infrarot-Lichtschranke. Kurz vor dem Ende der Rennstrecke fährt das Auto durch eine zweite Lichtschranke. Die beiden Lichtschranken erzeugen die Signale LS1 (Start) und LS2 (Stop). Beide Signale sind low-aktiv. Das Verbindungskabel der Rennstrecke wird mit dem Stecker auf der linken Seite des Demoboards verbunden. Achten Sie darauf, dass sich Pin 1 von Stecker und Buchse an derselben Stelle befinden. Pin 1 des Steckers auf dem Demoboard ist links oben. An dieser Stelle muss sich die farbige Markierung des Flachbandkabels befinden.

LS1 ist mit Schalter S9 gekoppelt, LS2 mit Schalter S10 (siehe Abschnitt 3.5). Damit LS1 und LS2 zum Demoboard durchgeschaltet werden, müssen sich beide Schalter in der Stellung 1 (oben) befinden. Mit S9 = 0 kann das Startsignal simuliert werden. Das Stoppsignal kann mit S10 = 0 simuliert werden. Die Betätigung der Schalter ist bei angeschlossenem Flachbandkabel möglich.

Modifizieren Sie die Stoppuhr so, dass sie mit dem Schalter S9 gestartet und mit dem Schalter S10 gestoppt wird.

Nun können Sie ein Modellauto starten und die Zeit bestimmen, die es für die Strecke zwischen den Lichtschranken benötigt.

Anhang

A1 Technische Daten des MACH Demoboards

Bedienelemente	2 Taster (entprellt), 14 Schalter (nicht entprellt)
Anzeigen	15 LED, 4 x 7-Segment-Anzeigen (Multiplex-Betrieb)
Taktsignale	Single Step, 0,3 Hz, 1 Hz, 3 Hz, 100 Hz, 200 Hz, 2,08 MHz
PLD	MACH XO2-256
Spannungsversorgung	
USB	5 VDC, max. 80 mA
extern	7-15 VDC/5-12 VAC, max. 80 mA
Gewicht	ca. 800 g
Abmessungen	120 x 100 mm
Temperaturbereich	0 … 70°C

A2 Literatur

[1] Lattice Semiconductor: MACH XO2 Family Handbook.
www.latticesemi.com

[2] Lattice Semiconductor: MACH XO2 Family Data Sheet.
www.latticesemi.com

[3] Ashenden, Peter J.: The Designer’s Guide to VHDL. Morgan Kaufmann Publishers, 3. Auflage, 2006.

[4] Reichardt, Jürgen: Lehrbuch Digitaltechnik. Eine Einführung mit VHDL. Oldenbourg, München, 2011.

[5] Dokumentation auf der Festplatte nach der Installation von Lattice Diamond (Auswahl):

Verzeichnis	Datei	Inhalt
...\docs\manuals	DiamondUG.pdf	Diamond User Guide
	fpga_library.pdf	Lattice Bibliothek
	programming_ug.pdf	Programming User Guide
	release notes.pdf	Hinweise zur aktuellen Version
...\docs\tutorial	fpga_design_tutor.pdf	Lattice Diamond Tutorial
	LSE_tutor.pdf	Lattice Synthesis Engine Tutorial

A3 Programmiersprache VHDL

Im Folgenden werden wesentliche Bestandteile der Programmiersprache VHDL dargestellt, die zur Durchführung des Digitallabors benötigt werden. Es ist nicht beabsichtigt, die Eigenschaften von VHDL vollständig wiederzugeben.

VHDL wurde erstmals definiert im Jahr 1987 in der Norm IEEE 1076-1987 (VHDL-1987). Später erfolgten Erweiterungen und Ergänzungen dieser Norm in den Jahren 1993 (VHDL-1993), 2002 (VHDL-2002) und 2008 (VHDL-2008). Die Programmierumgebungen der PLD-Hersteller unterstützen unterschiedliche VHDL-Fassungen. Die nachfolgend angegebenen VHDL-Eigenschaften werden von der Entwicklungsumgebung Lattice Diamond 3.1 unterstützt (VHDL-2008).

A3.1 Programmrahmen

```
entity <entity_name> is
   port (<Eingänge> in <Typ>; <Ausgänge> out <Typ>);
end <entity_name>;

architecture <architecture_name> of <entity_name> is
begin
   <process_name>: process is
   begin
   .
   .
   .
   wait on <Liste>;
   end process <process_name>;
end architecture <architecture_name>;
```

Beispiel:

```
entity nand2 is         -- Nand mit 2 Eingaengen
   port (
      A, B: in std_logic;
      y: out std_logic
      );
end nand2;

architecture behavior of nand2 is
   gate: process is
      y <= A nand B;
      wait on A, B;
   end process gate;
end architecture behavior;
```

Kommentar wird mit zwei voranstehenden Bindestrichen eingeleitet und kann an beliebiger Stelle im Programm auftreten. Der Kommentar erstreckt sich auf den Rest der Zeile.

Es können innerhalb einer architecture mehrere Prozesse definiert werden.

Achtung! Die im Prozess definierten Befehle werden nicht sequenziell abgearbeitet, sondern gleichzeitig! Im Gegensatz zur sequenziellen Abarbeitung der Befehle bei einem Mikrocontroller-Programm! Ergebnisse eines Befehls können deshalb nicht als Operand in einem nachfolgenden Befehl verwendet werden.

Nach Ausführung der Befehle des Prozesses wird auf eine Änderung der in der wait-Anweisung aufgeführten Signale gewartet. Alternativ kann die Liste in der process-Anweisung auftreten. Die wait-Anweisung muss dann entfallen.

Beispiel:

```
architecture behavior of nand2 is
   gate: process (A, B) is
      y <= A nand B;
   end process gate;
end architecture behavior;
```

A3.2 Zahlen und Zeichen

Integer

```
23
0
146
```

Floatpoint

```
23.1
0.0
3.14159
46E5
1E+12
19e00
1.234E09
98.6E+21
34.0e-08
```

Dual, Hexadezimal, Oktal

Die folgenden Darstellungen entsprechen der Hexadezimalzahl 0FDH mit der Zahlenbasis 2, 16 und 8:

```
2#11111101#
16#FD#
16#0fd#
8#0375#
```

Die Zahl 0,5 kann wie folgt dargestellt werden:

```
2#0.100#
8#0.4#
12#0.6#
```

Buchstaben

```
'A'
```

Strings

```
"A String"
```

Bit Strings

```
B"010011"
b"010011"
```

Gruppierungen sind möglich:

```
b"0101_0110_1100"
```

Dies entspricht:

```
H"56C"
```

Dieselbe Zahl oktal:

```
b"010_101_101_100"
O"2554"
```

A3.3 Datentypen

Integer

```
type <Datentyp> is range <Anfangswert> to <Endwert>;
```

Beispiele:

```
type year is range 0 to 2100;
type month is range 1 to 12;
type day_of_month is range 1 to 31;
type int16 is range 0 to 15;    -- Bereich fuer positive 4-Bit-Zahlen
type int16k is range -8 to 7;   -- Zweierkomplement mit 4 Bit
```

Der Datentyp `integer` ist vordefiniert. Er bezieht sich auf positive und negative ganze Zahlen inkl. 0, die mit 32 Bit im Zweierkomplement darstellbar sind. Die größte Zahl ist somit $2^{31}-1$. Die kleinste Zahl ist -2^{31}.

Floating-Point (Real)

```
type <Datentyp> is range <Anfangswert> to <Endwert>;
```

Beispiele:

```
type input_level is range -10.0 to 10.0;
type probability is range 0.0 to 1.0;
```

Der Datentyp `real` ist vordefiniert. Er bezieht sich auf das IEEE 32-Bit-Floatpoint-Format. Der Wertebereich umfasst $-1.0 \cdot 10^{38}$ bis $1.0 \cdot 10^{38}$.

Physikalische Typen

```
type <Datentyp> is range <Anfangswert> to <Endwert>
   units
   <Einheit(en)>
   end units <Datentyp>;
```

Beispiele:

```
type resistance is range 0 to 1E9
   units
      ohm;
   end units resistance;

type resistance is range 0 to 1E9
   units
      ohm;
      kohm = 1000 ohm;
      Mohm = 1000 kohm;
   end units resistance;

type time is range
   units
      fs;
      ps = 1000 fs;
      ns = 1000 ps;
      us = 1000 ns;
      ms = 1000 us;
      sec = 1000 ms;
      min = 60 sec;
      hr = 60 min;
   end units time;
```

Der Datentyp time ist vordefiniert.

Aufzählungstypen

```
type <Datentyp> is (<Aufzählung>);
```

Beispiele:

```
type octal_digit is ('0', '1', '2', '3', '4', '5', '6', '7');
variable last_digit: octal_digit;
last_digit := '0';

type logic_level is (unknown, low, undriven, high);
```

```
variable state: logic_level;
state := unknown;
```

Boolean

Vordefiniert ist:

```
type boolean is (false, true);
```

Eine Variable vom Typ boolean kann nur die Werte false oder true annehmen. Variable vom Typ boolean sind das Ergebnis von Vergleichen.

True liegt vor bei den Vergleichen:

```
123 = 123
'A' = 'A'
7 ns = 7 ns
123 < 456
789 ps <= 789 ps
'1' > '0'
```

False ergibt sich bei den Vergleichen:

```
123 = 456
'A' = 'z'
7 ns = 2 us
96 >= 102
2 us < 4 ns
'X' < 'X'
```

Typ Bit

Vordefiniert ist:

```
type bit is ('0', '1');
```

Beispiel:

```
variable A: bit;
A := '1';
```

Standard Logic

In der Bibliothek std_logic_1164 sind hardwarenahe Datentypen deklariert. Damit diese Datentypen verwendet werden können, muss am Anfang des Programms die Bibliothek wie folgt deklariert werden:

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

Häufig verwendete Datentypen sind std_ulogic, std_ulogic_vector, std_logic und std_logic_vector. Ihre Verwendung ergibt sich aus den nachfolgenden Beispielen. Weitere Hinweise finden sich in Abschnitt A3.13.

```
type std_ulogic is ('U',  -- Uninitialized
                    'X',  -- Forcing unknown
                    '0',  -- Forcing zero
                    '1',  -- Forcing one
                    'Z',  -- High impedance
                    'W',  -- Weak unknown
                    'L',  -- Weak zero
                    'H',  -- Weak one
                    '-'); -- Don't care
```

Dieser Datentyp ist erforderlich, um neben den Zustanden 0 und 1 eines Signals auch andere Eigenschaften wie z. B. einen hochohmigen Ausgang (Hi Z) verwenden zu können. Der hochohmige Ausgang wird vor allem bei Busverbindungen benötigt, wenn mehrere Ausgänge auf dieselbe Leitung geschaltet werden. U bedeutet, dass das Signal nicht initialisiert wurde. 0 und 1 bedeuten einen Logikpegel entsprechend 0 bzw. 1. L entspricht einem Pulldown und H einem Pullup mit einem Widerstand. X entsteht, wenn zwei Ausgänge 0 und 1 auf eine Leitung einspeisen. W tritt auf, wenn zwei Ausgänge L und H einspeisen.

Es ist nicht festgelegt, welcher Zustand auftritt, wenn zwei verschiedene Signale vom Typ std_ulogic zusammengeschaltet werden. Wird bei einer Busverbindung ein Teilnehmer aktiviert, so weisen die Ausgänge aller anderen Teilnehmer den Zustand Z auf. Der aktivierte Teilnehmer bestimmt den Zustand auf der Leitung. Die Leitung wird also beschaltet mit mehreren Ausgängen im Zustand Z und einem Ausgang im Zustand 0 bzw. 1. Der hierbei entstehende Buszustand ist im Fall von std_ulogic nicht gelöst (unresolved → ulogic). Eine Fehlermeldung signalisiert den Konflikt.

Das Problem ist gelöst (resolved) mit dem Subtyp std_logic:

```
subtype std_logic is resolved std_ulogic;
```

Der Typ std_logic umfasst dieselben Elemente wie std_ulogic. Allerdings ist genau festgelegt, welcher Zustand sich beim Zusammenschalten von Ausgängen mit unterschiedlichen Zuständen ergibt (Abschnitt A3.13).

Beispiel:

```
variable A: std_logic_vector (3 downto 0);
A := "1010";
```

In diesem Fall ist A(3) = 1, A(2) = 0, A(1) = 1, A(0) = 0. Alternativ wäre die folgende Deklaration möglich:

```
A(3) := '1';
A(2) := '0';
A(1) := '1';
A(0) := '0';
```

Eine alternative Definition des Vektors ist:

```
variable A: std_logic_vector (0 to 3);
```

```
A := "0101";
```

Die Reihenfolge der Bits ist nun umgekehrt. A(0) ist links aussen und A(3) rechts aussen.

Subtypen

Aus einem vorhandenen Datentyp kann ein Subtyp gebildet werden.

Beispiel:

```
subtype small_int is integer range -128 to 127;
```

Die Verwendung erklärt sich mit dem folgenden Programmauszug:

```
variable deviation: small_int;
variable adjustment: integer;
deviation := deviation + adjustment;
```

Subtyp (small_int) und Basistyp (integer) können in derselben Anweisung auftreten.

Vordefinierte Subtypen:

```
subtype natural is integer range 0 to <highest integer>;
subtype positive is integer range 1 to <highest integer>;
```

A3.4 Konstante und Variable

```
constant <name> : <type> := <value>;
```

Beispiele:

```
constant number_of_bytes: integer := 4;
constant e: real := 2.71828;
```

```
variable <name> : <type> [:= <value>];
```

Der Wert der Variablen muss in der Deklaration nicht angegeben werden. Die eckige Klammer kann entfallen.

Beispiele:

```
variable counter: integer := 5;
```

oder

```
variable counter: integer;
counter := 5;
```

A3.5 Signale

Signale dienen zur Verbindung der Ein-/Ausgänge von Modulen. Auch die in der entity-Deklaration angegebenen Ein-/Ausgänge sind Signale.

Beispiele:

```
signal int_clk: bit;
signal partial_product, full_product: integer;
```

Zuweisung von Werten an Signale:

```
int_clk <= '0';
partial_product <= 3;
```

A3.6 Operatoren

Operator	Operation	Linker Operand	Rechter Operand	Datentyp Ergebnis
not	Negation		bit, boolean, auch als Feld	wie Operand
and	Und	bit, boolean, auch als Feld	wie links	wie Operand
or	Oder			
nand	Nand			
nor	Nor			
xor	Antivalenz (Exor)			
xnor	Äquivalenz			
+	Addition	integer, float	wie links	wie Operand
-	Subtraktion			
*	Multiplikation			
/	Division			
&	Verkettung (concatenation)	Feld Feld Element vom Typ eines Feldelements Element vom Typ des Resultat-Felds	Feld Element vom Typ eines Feldelements Feld Element vom Typ des Resultat-Felds	Feld Feld Feld Feld
**	Potenz	integer, float	integer	wie linker Operand
abs	Absolutwert		integer, float	wie Operand
mod	Modulo	integer	integer	integer
rem	Rest	integer	integer	integer
sll	shift left logical	Feld von bit, boolean	integer	wie linker Operand
srl	shift right logical			
sla	shift left arithmetic			
sra	shift right arithmetic			
rol	rotate left			
ror	rotate right			
=	gleich	alle	wie links	boolean
/=	ungleich			
<	kleiner			
<=	kleiner gleich			
>	größer			
>=	größer gleich			

Tab. A3.1 Operatoren (Auswahl)

Feld = eindimensionales Feld (Vektor)

Beispiele:

```
y := abs a;

y := a and b;

y := a mod 2;

y := a/2;
```

A3.7 if-Anweisung

```
[label:] if <expression> then
            <statements>
         else
            <statements>
         end if [label];
```

Beispiele:

```
if en = '1' then
   stored_value := data_in;
end if;

if sel = '0' then
   result <= input_0;
else
   result <= input_1;
end if;
```

A3.8 case-Anweisung

```
[label:] case <expression> is
            when <choices> => <statements>;
            .
            .
            when others => <statements>;
         end case [label];
```

Beispiel:

```
case A is
   when "0000" => B <= "1010";
   when "0001" => B <= "1100";
   when "0010" => B <= "1101";
   when "0100" => B <= "1110";
   when others => B <= "0000";
end case;
```

Die Abfrage auf others muss immer vorhanden sein. Wenn in diesem Fall nichts geschehen soll, lautet die Zeile:

```
when others => null;
```

A3.9 for-Anweisung

```
[label:] for <identifier> in <discrete_range> loop
            <statements>
         end loop [label];
```

Beispiel:

```
for i in 0 to 3 loop
   A(i) <= '0';
end loop;
```

Achtung! Die Befehle werden nicht sequenziell ausgeführt wie bei einem Mikroprozessor. Die Schleife dient lediglich zur maschinenlesbaren Definition der Aufgabe.

A3.10 Felder

Im Folgenden werden nur eindimensionale Felder (Vektoren) betrachtet.

```
type <identifier> is array (<discrete_range>) of <element_type>;
```

Beispiele:

```
type bit_vector_8 is array (0 to 7) of bit;
variable A: bit_vector_8;
```

Definiert das Feld A mit den Elementen A(0), … , A(7). Der Zugriff auf die Elemente ist wie folgt möglich:

```
A(0) := '0';
A(1) := '1';
.
.
```

Alternativ kann auf A kompakt zugegriffen werden in der Form:

```
A := "10001110";
```

A(0) ist links außen, A(7) ist rechts außen.

Im Fall der Definition

```
type bit_vector_8 is array (7 downto 0) of bit;
```

wäre A(7) links außen und A(0) rechts außen.

A3.11 Ereignisse

Im Zusammenhang mit der Realisierung von Flipflops ist die Definition eines Zeitpunkts durch eine positive bzw. negative Taktflanke von Bedeutung. Dies geschieht wie folgt:

```
if (CLK'event and CLK = 1) then  -- positive Flanke

if (CLK'event and CLK = 0) then  -- negative Flanke
```

Alternativ können die Ereignisse "positive Flanke" und "negative Flanke" auch wie folgt definiert werden:

```
if rising_edge (CLK) then  -- positive Flanke

if falling_edge (CLK) then  -- negative Flanke
```

A3.12 Befehle für die Test Bench

Im Programm für die Test Bench werden die Testvektoren deklariert. In diesem Zusammenhang müssen oft Zeiten vorgegeben werden. Der Befehl

```
wait for 100 ns;
```

fügt eine Wartezeit von 100 ns ein. Beliebige Zeiten können vorgegeben werden.

Soll ein Befehl erst nach einer bestimmten Zeit ausgeführt werden, so kann dies wie folgt deklariert werden:

```
A := 5 after 5 ns;
```

Der Variablen A wird der Wert 5 erst nach einer Wartezeit von 5 ns zugewiesen, z. B. nach dem Ereignis "positive Flanke". Auf diese Weise kann die Erprobung eines Entwurfs mit realistischen Propagation Delay Times durchgeführt werden.

Achtung! Befehle mit Wartezeiten dienen nur zur Simulation! Wartezeiten können auf diese Weise nicht im PLD realisiert werden.

A3.13 IEEE Standard 1164

In der Bibliothek std_logic_1164 sind hardwarenahe Datentypen deklariert. Damit diese Datentypen verwendet werden können, muss am Anfang des Programms die Bibliothek wie folgt deklariert werden:

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

Mit dem Typ bit können lediglich die Zustände 0 und 1 erfasst werden. Logische Ausgänge können jedoch noch weitere Zustände annehmen, wie z. B. Hi Z (hochohmig) bei Tristate-Ausgängen. Zur Beschreibung dieser Signale dient der Typ std_ulogic:

```
type std_ulogic is ('U',  -- Uninitialized
                    'X',  -- Forcing unknown
                    '0',  -- Forcing zero
                    '1',  -- Forcing one
                    'Z',  -- High impedance
                    'W',  -- Weak unknown
                    'L',  -- Weak zero
                    'H',  -- Weak one
                    '-'); -- Don't care
```

Der hochohmige Ausgang Z wird vor allem bei Busverbindungen benötigt, wenn mehrere Ausgänge auf dieselbe Leitung geschaltet werden. U bedeutet, dass das Signal nicht initialisiert wurde. 0 und 1 bedeuten einen Logikpegel entsprechend 0 bzw. 1. L entspricht einem Pulldown und H einem Pullup mit einem Widerstand. X entsteht, wenn zwei Ausgänge 0 und 1 auf die Leitung einspeisen. W tritt auf, wenn zwei Ausgänge L und H einspeisen (Abb. A3.1). Vor dem Zusammenschalten ist A = H und B = L. Nach dem Zusammenschalten resultiert C = W.

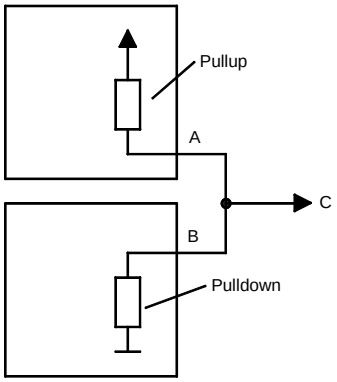


Abb. A3.1 Entstehung des Zustands W durch Zusammenschalten von Ausgängen mit den Zuständen H und L

Es ist nicht festgelegt, welcher Zustand auftritt, wenn zwei verschiedene Signale vom Typ std_ulogic zusammengeschaltet werden. Wird bei einer Busverbindung ein Teilnehmer aktiviert, so weisen die Ausgänge aller anderen Teilnehmer den Zustand Z auf. Der aktivierte Teilnehmer bestimmt den Zustand auf der Leitung (0 bzw. 1). Die Leitung wird also beschaltet mit mehreren Ausgängen im Zustand Z und einem Ausgang im Zustand 0 bzw. 1. Der hierbei entstehende Buszustand ist im Fall von std_ulogic nicht gelöst (unresolved → ulogic). Eine Fehlermeldung signalisiert den Konflikt.

Das Problem ist gelöst (resolved) mit dem Subtyp std_logic:

```
function resolved (s: std_ulogic_vector) return std_ulogic;
subtype std_logic is resolved std_ulogic;
type std_logic_vector is array (natural range <>) of std_logic;
```

Mit Elementen vom Typ std_logic kann ein Feld vom Typ std_logic_vector deklariert werden.

Der Typ std_logic umfasst dieselben Elemente wie std_ulogic. Allerdings ist genau festgelegt, welcher Zustand sich beim Zusammenschalten von Ausgängen mit unterschiedlichen Zuständen ergibt (Resolution Table, Tab. A3.2).

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

Tab. A3.2 Resolution Table

Im IEEE Standard 1164 wird aus Gründen der Austauschbarkeit von Programmen empfohlen, die Datentypen std_ulogic und std_ulogic_vector nicht zu verwenden.

A3.14 Verkettung (concatenation)

Felder können verkettet werden. Hierzu dient der &-Operator.

Beispiel:

```
variable A, B: std_logic_vector (1 downto 0);
variable C: std_logic_vector (3 downto 0);

C := A & B;
```

Der Inhalt der Felder A und B wird zum Feld C verkettet. A = "01" und B = "10" führt zu C = "0110".

Beispiel:

```
variable A3, A2, A1, A0: std_logic;
variable B: std_logic_vector (3 downto 0);

B := A3&A2&A1&A0;
```

Die Bits A0, … , A3 werden in das Feld B übernommen. Im Fall A0 = 1, A1 = 1, A2 = 0, A3 = 1 ergibt sich B = "1011".

A3.15 Typumwandlung

Bestimmte Operationen können nur mit denselben Datentypen erfolgen oder erfordern bestimmte Datentypen. Wenn der geforderte Datentyp nicht vorliegt, ist eine Typumwandlung notwendig.

Beispiel:

```
type int16 is range 0 to 15;
type int32 is range 0 to 31;
variable A, B: int16;
variable C: int32;

A := 5;
B := 6;
C := A + B;
```

Das Programm verursacht die Fehlermeldung

```
ERROR: 0 definitions of operator "+" match here
```

Die Ursache für die Fehlermeldung liegt darin, dass die beiden Variablen A und B vom Typ int16 sind, die Variable C aber den Typ int32 besitzt. Bei einer Addition müssen alle beteiligten Variablen denselben Typ aufweisen (siehe auch Tab. A3.1). Im vorliegenden Beispiel ist deshalb eine Typumwandlung erforderlich. Die Addition muss wie folgt formuliert werden:

```
C := int32(A) + int32(B);
```

Zur Durchführung der Addition werden A und B in den Typ int32 gewandelt.

Beispiel:

```
variable A, y: std_logic_vector (3 downto 0);

y := not A + 1;
```

Die Operation verursacht mehrere Fehlermeldungen. A ist ein Bitvektor. not A ist ebenfalls ein Bitvektor, nämlich das Einerkomplement von A. Additionen können aber nur mit Zahlen vom Typ Integer oder Float durchgeführt werden.

Beispiel:

```
variable A: std_logic_vector (3 downto 0);
type int16 is range 0 to 15;
variable Aint: int16;

Aint := int16(A);
```

Die Operation verursacht die Fehlermeldung:

```
ERROR: cannot convert type std_logic_vector to type int16
```

Beispiel:

```
variable A: std_logic;
type int16 is range 0 to 15;
variable Aint: int16;

Aint := int16(A);
```

Die Operation verursacht die Fehlermeldung:

```
ERROR: cannot convert type std_ulogic to type int16
```

Beispiel:

```
variable A: std_logic;
variable B: bit;

B := A;
```

Die Operation verursacht die Fehlermeldung:

```
ERROR: type error near a ; expected type bit
```

A ist vom Typ std_logic, B vom Typ bit. Eine direkte Zuweisung ist deshalb nicht möglich.

Beispiel:

```
variable A: std_logic;
variable B: bit;

B := bit(A);
```

Die Operation verursacht die Fehlermeldung:

```
ERROR: cannot convert type std_logic to type bit
```

Die Variable A vom Typ std_logic kann nicht durch direkte Typumwandlung in eine Variable vom Typ bit umgesetzt werden.

Im Fall der letzten beiden Beispiele kann man sich abhelfen durch die folgenden Anweisungen:

```
if A = '0' then
   B := '0';
else
   B := '1';
end if;
```

Die obigen Fehlermeldungen werden verursacht, wenn lediglich die Bibliothek std_logic_1164 verwendet wird.

A3.16 IEEE Standard 1076.3 (Numeric Standard Package)

Im Jahr 1997 wurde der IEEE Standard 1076.3 verabschiedet. Vergleichbar mit dem IEEE Standard 1164 handelt es sich hierbei ebenfalls um ein VHDL Synthesis Package, das weitere Typdefinitionen und Operatoren beinhaltet. Die im Paket enthaltenen Typdefinitionen und Operatoren beziehen sich vorwiegend auf Felder mit Elementen vom Typ std_logic. Das Paket wird auch als Numeric Standard Package bezeichnet. Damit es in einem Projekt eingesetzt werden kann, muss es wie folgt deklariert werden:

```
library ieee;
use ieee.numeric_std.all;
```

Mehrere Synthesis Packages können gleichzeitig verwendet werden. Eine Aktualisierung des Standards 1076.3 erfolgte bis zum heutigen Zeitpunkt nicht. Weiterhin ist die Version aus dem Jahr 1997 gültig. Diamond unterstützt den Standard 1076.3.

Die Datentypen Unsigned und Signed sind im Numeric Standard Package wie folgt definiert:

```
type unsigned is array (<discrete_range>) of std_logic;
type signed is array (<discrete_range>) of std_logic;
```

Unsigned bezieht sich auf positive Zahlen einschließlich 0, wohingegen der Datentyp Signed sowohl positive und negative Zahlen einschließlich 0 umfasst.

Beispiel:

```
signal Z: unsigned (3 downto 0);
```

Durch Verwendung der Datentypen Signed bzw. Unsigned sind Typumwandlungen von Bitvektoren in Zahlen und umgekehrt besonders einfach realisierbar.

A3.16.1 Wandlung von Bitvektoren in Signed bzw. Unsigned und umgekehrt

Beispiel:

```
signal A: std_logic_vector (3 downto 0);
signal Z: unsigned (3 downto 0);

Z <= unsigned (A);
A <= std_logic_vector (Z);
```

Beispiel:

Aus dem Bitvektor A soll das Zweierkomplement gebildet werden. Das Resultat liegt als Bitvektor Z vor.

```
signal A, Z: std_logic_vector (3 downto 0);

Z <= std_logic_vector (unsigned (not (A)) + 1);
```

Vom Bitvektor A wird das Einerkomplement gebildet. Das Einerkomplement wird als positive Zahl interpretiert. Zum Einerkomplement wird der Wert 1 addiert. Vom Resultat wird wieder ein Bitvektor gebildet. Im Fall A = 0000 ergibt sich als Einerkomplement K1(A) = 1111. Als positive Zahl interpretiert entspricht dies 15. Nach der Addition von 1 erhält man 16. Da sich der Ergebnisvektor Z auf 4 Bits beschränkt, folgt als Resultat Z = 0000.

Alternativ kann das Zweierkomplement wie folgt gebildet werden:

```
signal A, Z: std_logic_vector (3 downto 0);

Z <= std_logic_vector (- signed (A));
```

Wenn die Zahlen A und Z über eine unterschiedliche Bitbreite verfügen, muss auf identische Bitbreite ergänzt werden.

Beispiel:

```
signal A: std_logic_vector (2 downto 0);
signal Z: unsigned (3 downto 0);

Z <= unsigned ('0' & A);
```

Vor der Typumwandlung wurde das Feld auf 4 Stellen erweitert. Da A als positive Zahl interpretiert werden soll, wurde links eine 0 angefügt. Wenn A eine vorzeichenbehaftete Zahl ist, so muss die Erweiterung mit dem Vorzeichenbit erfolgen.

Beispiel:

```
signal A: std_logic_vector (2 downto 0);
signal Z: signed (3 downto 0);

Z <= unsigned (A(2) & A);
```

A3.16.2 Wandlung von Integer in Signed bzw. Unsigned

Um eine Zahl vom Typ Signed bzw. Unsigned aus einer Integer-Zahl zu erhalten, ist die folgende Typumwandlung naheliegend:

```
variable A: signed (3 downto 0);
type int16k is range -8 to 7;
variable Z: int16k;

A := signed (Z);
```

Obwohl der Zahlenbereich für beide Typen identisch ist, tritt die folgende Fehlermeldung auf:

```
ERROR: cannot convert type int16k to type signed
```

Zur Umwandlung von Integer-Zahlen in Zahlen der Typen Signed bzw. Unsigned dienen die Operationen to_signed bzw. to_unsigned. Subtypen, wie z. B. int16k, werden jedoch nicht unterstützt.

Beispiel:

```
variable A: signed (3 downto 0);
variable Z: integer;

A := to_signed (Z, 4);
```

Innerhalb der Funktion ist anzugeben, über wieviele Bits die Signed- bzw. Unsigned-Zahl verfügt. Ist Z ein Subtyp des Typs Integer, z. B. int16k, so kann die Typumwandlung wie folgt durchgeführt werden:

```
A := to_signed (integer(Z), 4);
```

A3.16.3 Wandlung von Signed bzw. Unsigned in Integer

Für die Umwandlung einer Signed- bzw. Unsigned-Zahl in eine Integer-Zahl ist die folgende Operation naheliegend:

```
variable A: signed (3 downto 0);
type int16k is range -8 to 7;
variable Z: int16k;

Z := int16k (A);
```

Die Operation führt jedoch zur Fehlermeldung:

```
ERROR: cannot convert type signed to type int16k
```

Auch eine Typumwandlung in den Datentyp Integer ist auf diese Weise nicht möglich.

Für die Typumwandlung von Signed- bzw. Unsigned-Zahlen in Integer-Zahlen wird deshalb die Funktion to_integer verwendet.

Beispiel:

```
variable A: signed (3 downto 0);
variable Z: integer;

Z := to_integer (A);
```

Wenn Z vom Subtyp int16k ist, lautet die Typumwandlung wie folgt:

```
variable A: signed (3 downto 0);
type int16k is range -8 to 7;
variable Z: int16k;

Z := int16k(to_integer (A));
```

Da es sich bei den Datentypen Signed bzw. Unsigned um Zahlen handelt, sind Additonen bzw. Subtraktionen unmittelbar möglich.

Beispiel:

```
variable A, B: signed (3 downto 0);
A := "0101";
B := A + 2;
```

Die Zuweisung

```
A := 5;
```

verursacht allerdings die Fehlermeldung

```
ERROR: type signal does not match with the integer literal
```

Die ganze Zahl 5 müsste mit Hilfe der Funktion to_signed in den Datentyp Signed gewandelt werden.

A3.16.4 Ausschneiden von Feldbereichen

Variable vom Typ std_logic_vector, Signed und Unsigned setzen sich aus einer vorgegebenen Anzahl von Elementen (Bits) zusammen. Bei Operationen mit Variablen dieser Datentypen ist auf die vorgegebene Feldgröße (Dimension) zu achten. Eine Zuweisung von Feldern mit unterschiedlichen Diemensionen ist grundsätzlich nicht möglich. Die Zuordnung vorgegebener Elemente und von Teilbereichen der Felder ist jedoch machbar.

Beispiel:

```
variable A: std_logic_vector (7 downto 0);
variable B: std_logic_vector (3 downto 0);

A := B;
```

Diese Zuweisung verursacht die Fehlermeldung:

```
ERROR: expression has 4 elements; expected 8
```

Eine Zuweisung eines Felds mit 4 Elementen zu einem Feld mit 8 Elementen ist auf diese Weise nicht möglich. Es muss klar festgelegt werden, welche Elemente der Felder einander zugewiesen werden sollen. Die folgenden Zuweisungen arbeiten fehlerfrei:

```
A(3 downto 0) := B;
A(4 downto 1) := B;
A(7 downto 4) := B;
```

Durch diese Anweisungen wird eindeutig festgelegt, wie die Elemente von Feld A den Elementen von Feld B zugeordnet werden sollen.

Die Anweisung

```
A(0 to 3) := B;
```

verursacht allerdings die Fehlermeldung:

```
ERROR: slice direction differs from its index subtype range
```

Die nachfolgenden Zuweisungen sind wiederum machbar:

```
B := A(3 downto 0);
B := A(7 downto 4);
B := A(6 downto 3);
```

Wohingegen die Anweisung

```
B := A(3 to 6);
```

ebenfalls die nachfolgende Fehlermeldung verursacht:

```
ERROR: slice direction differs from its index subtype range
```

Will man die Elemente A(3), ... , A(6) in dieser Reihenfolge dem Feld B zuordnen, d. h. A(3) links und A(6) rechts, so ist eine elementeweise Zuordnung erforderlich:

```
B := A(3)&A(4)&A(5)&A(6);
```

A3.17 Schiebeoperationen

In der Tab. A3.1 sind die folgenden Schiebeoperatoren aufgeführt:

sll	shift left logical
srl	shift right logical
sla	shift left arithmetic
sra	shift right arithmetic
rol	rotate left
ror	rotate right

Im Folgenden wird die Verwendung dieser Operatoren an Hand von Beispielen erläutert.

Beispiel:

```
B"10001010" sll 3 = B"01010000"
```

Um 3 Bits nach links schieben. Rechts wird 0 nachgeschoben.

Beispiel:

```
B"10010111" srl 2 = B"00100101"
```

Um 2 Bits nach rechts schieben. Links wird 0 nachgeschoben.

Beispiel:

```
B"00001100" sla 2 = B"00110000"
```

Um 2 Bit nach links schieben. Das Bit rechts außen wird nachgeschoben.

```
B"00010001" sla 2 = B"01000111"
```

Um 2 Bits nach links schieben. Das Bit rechts außen wird nachgeschoben.

Beispiel:

```
B"01001011" sra 3 = B"00001001"
```

Um 3 Bits nach rechts schieben. Das Bit links außen wird nachgeschoben.

```
B"10010111" sra 3 = B"11110010"
```

Um 3 Bits nach rechts schieben. Das Bit links außen wird nachgeschoben.

Beispiel:

```
B"10010011" rol 1 = "00100111"
```

Um 1 Bit nach links schieben. Das Bit links außen wird rechts nachgeschoben.

Beispiel:

```
B"10010011" ror 1 = B"11001001"
```

Um 1 Bit nach rechts schieben. Das Bit rechts außen wird links nachgeschoben.

A3.18 Die component-Deklaration

Mit Hilfe der component-Anweisung ist es möglich, Funktionsbausteine aus anderen Programmen oder aus Bibliotheken zu übernehmen. Innerhalb des eigenen Programms muss darauf hingewiesen werden, dass ein externer Funktionsbaustein verwendet wird und welche Übergabeparameter vorhanden sind.

```
component <identifier> is
   [generic (<generic_interface_list>);]
   [port (<port_interface_list>);]
end component [<identifier>];
```

Unter `generic` werden die Übergabeparameter aufgeführt, mit denen der Funktionsbaustein arbeiten soll. Die auszutauschenden Signale werden unter `port` zusammengefasst. Die Klammern [ ] geben an, dass diese Angaben entfallen können.

Beispiel:

```
component DFF is
   generic (Tpd: delay_length);
   port (
      Clk: in std_logic;
      Clr: in std_logic;
      D: in std_logic;
      Q: out std_logic
      );
end component DFF;
```

Im Beispiel wird ein D-Flipflop mit seinen Ein- und Ausgängen deklariert. Außerdem wird die Propagation Delay Time Tpd angegeben, mit der das Flipflop arbeiten soll.

Das folgende Beispiel zeigt, wie ein 4-Bit-Register unter Verwendung des Funktionsbausteins DFF aufgebaut wird.

Beispiel:

```
entity REG4 is
   port (
      Clk, Clr: in std_logic;
      D: in std_logic_vector (3 downto 0);
      Q: out std_logic_vector (3 downto 0)
      );
   end entity REG4;


architecture struct of REG4 is

   component DFF is
      generic (Tpd: delay_length);
      port (
```

```
         Clk: in std_logic;
         Clr: in std_logic;
         D: in std_logic;
         Q: out std_logic
         );
   end component DFF;

begin

   DFF0: component DFF
      generic map (Tpd => 2 ns);
      port map (Clk => Clk, Clr => Clr, D => D(0), Q => Q(0));

   DFF1: component DFF
      generic map (Tpd => 2 ns);
      port map (Clk => Clk, Clr => Clr, D => D(1), Q => Q(1));

   DFF2: component DFF
      generic map (Tpd => 2 ns);
      port map (Clk => Clk, Clr => Clr, D => D(2), Q => Q(2));

   DFF3: component DFF
      generic map (Tpd => 2 ns);
      port map (Clk => Clk, Clr => Clr, D => D(3), Q => Q(3));

end architecture struct;
```

Es werden die 4 Instanzen DFF0, DFF1, DFF2 und DFF3 der Komponente DFF erzeugt. Jede Instanz dient zum Speichern eines Bits.

A4 Signaltabelle für den Programmentwurf

XO2-Pin	Signal/Bauteil	Signalname im Projekt
1	S1	
2	S2	
3	S3	
4	S4	
7	S5	
8	S6	
12	S7	
13	S8	
16	D1	
17	D2	
20	S11	
21	D3	
24	D4	
27	D5	
28	D6	
31	D7	
32	D8	
34	S12	
35	S13	
39	S14	
40	S16	
41	S15	
42	LSP	
43	S9	
48	S10	
51	SEL4	
52	SEL3	
53	SEL2	
54	SEL1	
62	D15	
63	D14	
67	D13	
70	D12	
71	D11	
74	D10	
75	D9	
81	SEG8	
82	SEG7	
85	SEG6	
86	SEG5	
87	SEG4	
88	SEG3	
98	SEG2	
99	SEG1	

A5 Programmpaket VHDL4Digilab

Das Programmpaket VHDL4Digilab.zip kann vom internen Downloadbereich der Fakultät heruntergeladen werden. Es beinhaltet verschiedene Programme, deren Funktion in der nachfolgenden Tabelle beschrieben ist.

Programm	Aufgabe	Beschrieben in	Datei(en)
Dec7S	Code-Umsetzer (Decoder) vom Dualcode in den 7-Segment-Code	Kap. 7.2.1	Dec7S.vhd
ClkGen	Modul zur Takterzeugung	Kap. 3.4, 7.2.2	ClkGen.vhd
CT	Erzeugung des Basistakts 100 Hz	Kap. 7.2.2	CT.vhd, CT.ipx, CT.lpc
Nand2	Nand mit 2 Eingängen	Kap. 5.3	Nand2.vhd

Tab. A5.1 Inhalt des Programmpakets VHDL4Digilab

Die Dateien müssen in das aktuelle Projektverzeichnis kopiert werden. Mit

File > Add > Existing File

werden die Dateien in das aktuelle Projekt übernommen.

Die Datei Nand2.vhd dient als Vorlage zur Erstellung von VHDL-Programmen.

A6 Verhalten des PLD XO2-256 bei einem Reset

Nach dem Anlegen der Versorgungsspannung führt das PLD einen Reset durch. Hierbei wird das im EEPROM befindliche Anwenderprogramm ausgelesen und das PLD entsprechend strukturiert. Anschließend wird ein GSR (Global Set/Reset) durchgeführt. Hierbei werden bestimmte Speicherelemente in einen definierten Anfangszustand versetzt. GSR wirkt sich auch auf Funktionsbausteine aus, die sich im EFB (Extended Function Block) befinden, z. B. Counter/Timer. Durch entsprechende Parametrierung dieser Funktionsbausteine kann verhindert werden, dass sich GSR auf diese Funktionsbausteine auswirkt.

Aktionen bei GSR:

- Set/Reset von Flipflops, Zählern etc. aus der Lattice-Bibliothek mit GSR-Eigenschaft.
- Kein Set/Reset von Flipflops, Zählern etc. aus der Lattice-Bibliothek mit Eingang für asynchrones Set/Reset.
- Set/Reset selbst definierter Flipflops, Zähler etc. ohne asynchronen Set/Reset.

Bei Funktionsbausteinen aus der Lattice-Bibliothek ist angegeben, ob GSR unterstützt wird. Flipflops sind gekennzeichnet durch GSR (Clear) bzw. GSR (Preset). Im Fall GSR (Clear) nimmt das Flipflop bei einem GSR den Zustand 0 an. Bei Flipflops mit der Kennzeichnung GSR (Preset) bewirkt GSR den Anfangszustand 1.

Die für GSR erforderlichen Signale verlaufen im Hintergrund und sind für den Anwender nicht sichtbar.

Achtung! Momentan ist nicht bekannt, wie ein VHDL-Programm für ein Flipflop zu erstellen ist, so dass GSR (Clear) oder GSR (Preset) erfolgt. Bei mit VHDL definierten Zählern erfolgt kein GSR.

Der Funktionsbaustein GSR befindet sich in der Lattice-Bibliothek. Das Verhalten eines Projekts ist abhängig davon, ob dieser Funktionsbaustein verwendet wird und wie das Eingangssignal des Funktionsbausteins gewählt wird (siehe Tab. A6.1).

Verwendung des Funktionsbausteins GSR:	Beliebiger asynchroner Reset verursacht:
Ohne GSR.	Stop interner Oszillator. GSR.
Mit GSR. GSRIN = 1.	Interner Oszillator läuft weiter. Kein GSR.

Tab. A6.1 Verhalten bei asynchronem Reset in Abhängigkeit von der Verwendung des Funktionsbausteins GSR

Ein asynchrones Reset-Signal eines beliebigen Funktionsbausteins verursacht GSR, wenn der Funktionsbaustein GSR im Projekt nicht verwendet wird. Derselbe GSR wird bei Verwendung des Funktionsbausteins GSR ausgelöst, wenn dessen Eingangssignal 0 ist.

Der Funktionsbaustein GSR kann in einem Schaltplan platziert werden. Er kann aber auch innerhalb eines VHDL-Programms als Instanz definiert werden:

```
library MACHXO2;
use MACHXO2.components.all;

entity ...
   port (
      GSRIN: in std_logic;

architecture ...

component gsr is
   port (GSR: in std_logic);
end component gsr;

begin

   GSR0: component gsr
      port map (GSR => GSRIN);
```

Abb. A6.1 Deklaration der Instanz GSR0 in einem VHDL-Programmm (Programmauszug)

Der Funktionsbaustein GSR muss in einem Projekt stets vorhanden sein, da sonst eine korrekte Funktion asynchroner Reset-Signale nicht gewährleistet ist!

Der Funktionsbaustein PUR (Power Up Reset) aus der Lattice-Bibliothek ist unter VHDL nicht funktionsfähig.

Der Funktionsbaustein SGSR aus der Lattice-Bibliothek verursacht einen synchronen Reset. SGSR hat die Eingänge CLK und GSR. Die Funktionsbausteine GSR und SGSR dürfen nicht gemeinsam verwendet werden! Im Folgenden wird der Eingang GSR des Funktionsbausteins SGSR mit GSRIN bezeichnet. GSRIN ist Low-aktiv. Im Fall GSRIN = 0 erfolgt sofort GSR mit den oben erläuterten Eigenschaften. Wenn danach GSRIN = 1 ist, so wird GSR nicht sofort zurückgenommen. Die Rücknahme von GSR erfolgt erst nach zwei positiven Flanken des Taktsignals CLK.

A7 Test des MACH-Demoboards

Besteht der Verdacht, dass das Demoboard Fehlfunktionen aufweist, so kann die Funktionsweise mit der Hilfe von Testprogrammen überprüft werden. Die Testprogramme sind in der Datei MD12_Test_140115.zip enthalten. Diese Datei kann vom Downloadbereich der Hochschule geladen werden. Die Datei beinhaltet die Archivdateien MD12_Test1.zip und MD12_Test2.zip. Zur Durchführung der Tests wird die Datei MD12_Test1.zip in das Verzeichnis C:\MD12_Test\Test1 kopiert. Die Datei MD12_Test2.zip wird in das Verzeichnis C:\MD12_Test\Test2 kopiert. Diamond starten.

Test1

File > Open > Archived Project ...

MD12_Test1.zip wählen.

Das PLD auf dem Demoboard programmieren.

Hinweis: Es kann sein, dass der Programmer die JEDEC-Datei nicht findet. Im Programmer dann den Pfad für die JEDEC-Datei anpassen.

Die Schalter S1-S8 aktivieren die LED's D1-D8.

Die Schalter S9-S10 aktivieren die LED's D12-D13.

Die Taster S11-S12 aktivieren die LED's D14-D15.

Die Schalter S13-S14 aktivieren die LED's D9-D10. Für die LED D11 gilt:

D11 = S15 xnor S16 (Äquivalenz).

Alle Schalter und LED's prüfen.

Schalter/Taster	LED's
S1-S8	D1-D8
S9-S10	D12-D13
S11-S12	D14-D15
S13-S14	D9-D10
S15-S16	D11=S15 xnor S16 (Äquivalenz)

Tab. A7.1 Verknüpfung von Schaltern/Tastern und LED's

Test2

File > Open > Archived Project ...

MD12_Test2.zip wählen.

Das PLD auf dem Demoboard programmieren.

Hinweis: Es kann sein, dass der Programmer die JEDEC-Datei nicht findet. Im Programmer dann den Pfad für die JEDEC-Datei anpassen.

Dieses Testprogramm dient zur Überprüfung der 7-Segment-Anzeigen. Mit den Schaltern S1-S8 werden die Segmente inkl. Dezimalpunkt ausgewählt (A-G, DP). Die Anzeigen DIG1-DIG4 werden mit den Schaltern S13-S16 aktiviert. Die Schalter S13-S16 können gleichzeitig aktiviert werden.

A8 Häufige Fehlerursachen

(1) Fehler 999

Ursache: Programm wurde im Verzeichnis C:\lscc oder einem Unterverzeichnis davon erstellt.

Abhilfe: Anderen Programmordner wählen, z. B. C:\Diamond\Projekte

(2) Schaltflächen (Tabs) Process, Hierarchy, etc., werden nicht angezeigt.

Ursache: Veränderung der Programmoberfläche.

Abhilfe:

Window > Load Default Layout

(3) Im Spreadsheet werden bei Zuordnung eines Signalnamens durch Assign Signal stets zwei Pins belegt.

Ursache: Versehentlich wurde der Logikpegel LVCMOS33D gewählt.

Abhilfe: LVCMOS33 für alle Pins wählen.

(4) Keine Kommunikation mit dem Demoboard.

Ursache: Treiber für die USB-Schnittstelle wurde nicht korrekt installiert.

Abhilfe: Siehe Dokumentation im Downloadbereich.

(5) Falsches PLD gewählt.

Ursache: Bei der Erstellung des Projekts wurde ein falscher PLD-Typ ausgewählt.

Abhilfe: Im Fenster File List Doppelklick mit der linken Maustaste auf das PLD und Auswahl des richtigen PLD.

Hinweis: Es kann sein, dass das neue PLD zwar eingestellt wird, die neuen PLD-Eigenschaften aber im Spreadsheet nicht berücksichtigt werden. In diesem Fall Projekt schließen und erneut öffnen. Wenn das Problem weiterhin besteht, neues Projekt erstellen und die vorhandenen Dateien übernehmen.

(6) Pins werden im Spreadsheet nicht angeboten.

Ursache: Falsches PLD gewählt.

Abhilfe: siehe (5).

(7) Löschen von Signalen im Schaltplan bewirkt Störungen im Spreadsheet.

Ursache: Die im Schaltplan gelöschten Signale sind im Spreadsheet unter Signal Name nicht mehr eingetragen. Es erfolgen jedoch Fehlermeldungen, dass die Signale nicht auffindbar wären.

Abhilfe: Neues Projekt erstellen. Die Dateien *.vhd, *.sch und *.sym in das Verzeichnis des neuen Projekts kopieren und die Dateien *.vhd und *.sch mit File > Add übernehmen. Process Synthesize durchführen. Spreadsheet definieren.

Alternativ: Im Projekt einen neuen Schaltplan erstellen und den Inhalt des vorhandenen Schaltplans in den neuen Schaltplan kopieren. Bisherigen Schaltplan mit Remove entfernen. Das Spreadsheet weist nun keine Fehler mehr auf.

Alternativ: Vor dem Löschen eines Signals im Schaltplan das Signal im Spreadsheet löschen. Erst danach das Signal im Schaltplan löschen.

Alternativ: In der Datei *.lpf die nicht mehr vorhandenen Signale löschen. Aufruf der Datei durch Doppelklick auf den Dateinamen. Datei *.lpf speichern. Danach arbeitet der Spreadsheet Editor wieder einwandfrei.

(8) Beim Speichern eines Schaltplans tritt unter Windows 8 die folgende Meldung auf:

Qt: Untested Windows Version 6.2 detected!
Done: Error code 1

Ursache: Windows 8 wird durch Lattice Diamond nicht unterstützt.

Abhilfe: Win XP oder Win 7 im Kompatibilitätsmodus benutzen.

(9) Modulo-Funktion wird nicht korrekt durchgeführt.

Die Funktion mod n wird nur für n = 2, 3, ... , 9 korrekt durchgeführt. Ab n = 10 treten Fehler auf. Die Simulation arbeitet einwandfrei.

Ursache: Probleme bei Place & Route mit LSE (Lattice Synthesis Engine).

Abhilfe: Modulo-Funktion nicht benutzen. Modulo-Funktion mit logischen Abfragen realisieren.

Hinweis: Ab Diamond 3.1 arbeitet die Modulo-Funktion einwandfrei.

(10) Asynchroner Reset funktioniert nicht korrekt (bei Flipflops, Zählern, etc.).

Ursache: Funktionsbaustein GSR (Global Set/Reset) fehlt im Projekt.

Abhilfe: Funktionsbaustein GSR in Schaltplan aufnehmen und mit H beschalten.

(11) Aldec HDL Simulator startet nicht unter Windows 8.1.

Meldung: Untested Windows Version 6.2 detected!

Ursache: Windows 8 wird durch Lattice Diamond nicht unterstützt.

Abhilfe: Kompatibilitätsmodus Windows 7 oder Windows XP verwenden.

(12) Als ZIP-Datei archivierte Projekte können nicht auf einem anderen Computer bearbeitet werden.

Meldungen (nach Synthesize mit LSE (Lattice Synthesize Engine)):

ERROR: File is Empty
Done: error code 2

Abhilfe: Projekt neu erstellen und die VHD-Dateien mit "Add Existing File" in das Projekt aufnehmen.

(13) Division mit 10 (dezimal) erfolgt fehlerhaft.

Meldung: -

Abhilfe: Nacheinander mit 5 und 2 dividieren.

A9 Einschränkungen bei Lattice Diamond

Das Programm Lattice Diamond unterstützt nicht alle Eigenschaften von VHDL. Die folgenden Einschränkungen sind bekannt:

(1) Keine Unterstützung des Datentyps Floating-Point (Real).

A10 Testatblatt

Name	Vorname

Übung Nr.	Datum	Ausarbeitung	Versuch
1			
2			
3			
4			
5			
6			