

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl für Technische Elektronik
Prof. Dr.-Ing. Dr.-Ing. habil. Robert Weigel

PRAKTIKUM ZUR VORLESUNG
Architekturen der Digitalen Signalverarbeitung
ADS

Realisierung einer digitalen Übertragungsstrecke von der
Systemsimulation bis zur fertigen Hardware

A. Schedel / D. Glaser

Übersicht

I	Einführung	1
1	Aufbau	2
2	Konventionen	3
3	Zur Struktur	4
4	Die Hardware	5
II	Versuche	9
5	Sender	11
6	Empfänger	59
7	Gesamtsystem	93
III	Anhang	98
A	MATLAB Tutorial	99
B	ispLEVER	123
C	VHDL-Referenz	125

Inhaltsverzeichnis

I	Einführung	1
1	Aufbau	2
2	Konventionen	3
2.1	Vorbereitung	3
2.2	Warnungen und Hinweise	3
3	Zur Struktur	4
4	Die Hardware	5
4.1	Kurzbeschreibung	5
4.2	Vorbereitung der Hardware	5
II	Versuche	9
5	Sender	11
5.1	Versuch 1: Erste Schritte	12
5.1.1	Konzept	12
5.1.2	Realisierungsmöglichkeiten	12
5.1.3	VHDL: Realisierung	12
5.1.3.1	VHDL-Basics: Libraries	12
5.1.3.2	VHDL-Basics: Die Entity	14
5.1.3.3	VHDL-Basics: Die Architecture	14
5.1.3.4	VHDL-Basics: Concurrent Statements	15
5.1.4	VHDL: Beschreibung der Hardware	15
5.1.5	VHDL: Implementierung in Hardware	16
5.1.6	TEST: Praxis	16
5.2	Versuch 2: Zufallsfolgenerator	18
5.2.1	Konzept	18
5.2.2	Realisierungsmöglichkeiten	18
5.2.2.1	PRN-Schieberegister	18
5.2.3	MATLAB: Programmierung	19
5.2.4	VHDL: Realisierung	20
5.2.4.1	VHDL-Basics: Signale	20
5.2.4.2	VHDL-Basics: Sequentielle Prozesse	21

	5.2.4.3	VHDL-Basics: if-then-else	22
5.2.5		VHDL: Beschreibung der Hardware	23
5.2.6		MODELSIM: Simulation der Beschreibung	23
	5.2.6.1	Einführung in Modelsim	23
5.2.7		TEST: Praxis	25
5.3		Versuch 3: Signalgenerator	29
	5.3.1	Konzepte	29
	5.3.2	Realisierungsmöglichkeiten	29
	5.3.2.1	Direkte Digitale Synthese	29
	5.3.2.2	CORDIC	31
5.3.3		MATLAB: Programmierung	36
5.3.4		Einschub: Maschinenzahlen	38
	5.3.4.1	Zahlendarstellung auf Digitalrechnern	38
	5.3.4.2	Darstellung im Zweierkomplement	38
	5.3.4.3	Zahlendarstellung im Gleitkommaformat	41
5.3.5		VHDL: Realisierung	43
	5.3.5.1	VHDL-Basics: Fixed-Point Arithmetik	43
	5.3.5.2	VHDL-Basics: Konstanten	44
	5.3.5.3	VHDL-Basics: Typenkonvertierung	44
	5.3.5.4	VHDL-Basics: Mehrdimensionale Arrays	44
	5.3.5.5	VHDL-Basics: Zahlentypen	45
	5.3.5.6	VHDL-Basics: Records	45
	5.3.5.7	VHDL-Basics: Komponenten	46
	5.3.5.8	VHDL-Basics: Multiplizierer	47
5.3.6		VHDL: Beschreibung CORDIC-Base	47
5.3.7		MODELSIM: Simulation CORDIC-Base	48
5.3.8		VHDL: Beschreibung 360°-CORDIC	48
5.3.9		MODELSIM: Simulation 360°-CORDIC	49
5.3.10		VHDL: Tonerzeugung	49
5.3.11		MODELSIM: Tonerzeugung	49
5.3.12		TEST: Praxis	50
5.4		Versuch 4: Pegelanzeige	51
	5.4.1	Konzept	51
	5.4.2	Realisierungsmöglichkeiten	51
	5.4.2.1	Pegeldefinition	51
	5.4.3	VHDL: Realisierung	51
	5.4.3.1	VHDL-Basics: Variablen	51
	5.4.3.2	VHDL-Basics: Simulation <-> Synthese	53
5.4.4		VHDL: Beschreibung der Hardware	53
5.4.5		MODELSIM: Simulation der Beschreibung	53
5.4.6		TEST: Praxis	54
5.5		Versuch 5: Modulator	55
	5.5.1	Konzept	55
	5.5.2	Realisierungsmöglichkeiten	55
	5.5.3	MATLAB: Programmierung	55

5.5.4	VHDL: Realisierung	56
5.5.5	VHDL: Modulator mit zwei Signalgeneratoren	56
5.5.6	MODELSIM: Simulation mit zwei Signalgeneratoren	56
5.5.7	VHDL: Modulator mit einem Signalgenerator	57
5.5.8	MODELSIM: Beschreibung mit einem Signalgenerator	57
5.5.9	TEST: Praxistest	58
6	Empfänger	59
6.1	Aufbau	59
6.2	Grundlagen Digitaler Filter	62
6.2.1	Einführung	62
6.2.1.1	Echtzeitsystem zur digitalen Filterung	62
6.2.1.2	Grundlegende Filterfunktionen	62
6.2.1.3	Das Digitalfilter als LTI-System	63
6.2.2	Eigenschaften und Strukturen von FIR-Filtern	65
6.2.2.1	Grundlagen	65
6.2.2.2	Eigenschaften von FIR-Filtern	66
6.2.2.3	Strukturen von FIR-Filtern	68
6.2.3	Entwurf von FIR-Filtern	69
6.2.3.1	Filterentwurf mit der Fenstermethode	69
6.2.3.2	Filterentwurf mit der Optimalmethode	72
6.2.4	Nichtideale Effekte bei digitalen Filtern	73
6.2.4.1	Quantisierungsrauschen durch AD-Wandlung	73
6.2.4.2	Quantisierung der Filterkoeffizienten	74
6.2.4.3	Quantisierte Arithmetik	75
6.2.5	Entwurf digitaler Filter mit MATLAB	79
6.2.5.1	Das Filterdesigntool FDA	79
6.2.6	MATLAB: Filterdesign	82
6.3	Versuch 6: Simulation der Empfängerhardware in MATLAB	84
6.4	Versuch 7: VHDL: Bandpass	87
6.5	Versuch 8: VHDL: Optimierung der Filterhardware	88
6.6	Versuch 9: VHDL: Demodulator	90
6.7	Versuch 10: VHDL: Tiefpass	91
6.8	Versuch 11: VHDL: Signalregeneration	91
7	Gesamtsystem	93
7.1	Versuch 12: Simulation einer nichtidealen Übertragung	94
7.2	Versuch 13: Loop-Test	95
7.2.1	Konzept	95
7.2.2	VHDL: Beschreibung der Hardware	95
7.2.3	TEST: Praxis	95
7.3	Versuch 14: Chat-Session	96
7.3.1	Konzept	96
7.3.2	VHDL: Beschreibung der Hardware	96
7.3.3	TEST: Praxis	96

III Anhang	98
A MATLAB Tutorial	99
A.1 Bedeutung von MATLAB	99
A.2 Die MATLAB-Umgebung	101
A.3 MATLAB im direkten Modus	102
A.3.1 Die MATLAB-Hilfe	102
A.3.2 Elementare Funktionen	102
A.3.3 Umgang mit dem MATLAB-Workspace	103
A.3.4 Matrizen und Vektoren	104
A.3.4.1 Matrizen	104
A.3.4.2 Vektoren	106
A.4 Programmieren in MATLAB	108
A.4.1 MATLAB-Skripte	108
A.4.1.1 Erstellen von MATLAB-Skripten	108
A.4.1.2 Debugging	111
A.4.2 Graphische Ausgabe	112
A.4.3 MATLAB-Funktionen	113
A.4.4 Kontrollstrukturen	113
A.4.4.1 Konditionale Verzweigung	113
A.4.4.2 Schleifen	114
A.5 Zusammenfassung	116
B ispLEVER	123
B.1 Die Oberfläche	123
B.2 Der VHDL-Editor	124
B.3 Wie geht es weiter?	124
C VHDL-Referenz	125
C.1 Einführung	125
C.2 Basiskonstrukte	125
C.2.1 Libraries	125
C.2.2 Entity	125
C.2.3 Architecture	125
C.2.4 Signale und Konstanten	126
C.2.5 Typen	126
C.2.6 Typenkonvertierung	126
C.2.7 Prozesse	127
C.2.8 Kontrollstrukturen	127
C.2.8.1 if-then-else	127
C.2.8.2 when-else	127
C.2.8.3 case-when	127

Teil I

Einführung

1 Aufbau

Im Rahmen dieses Praktikums soll der Entwurf eines Übertragungssystems von der grundlegenden Idee bis hin zur fertigen Hardware-Implementierung erarbeitet werden. Am Ende steht eine Übertragungsstrecke, die einen einfachen Bitstrom über einen Lautsprecher im hörbaren Bereich überträgt. Die Signale sollen über ein Mikrofon aufgenommen und von der nachfolgenden Schaltung demoduliert werden. Sie werden die Strecke zu großen Teilen mit dem Simulationstool Matlab beschreiben und simulieren. Nach erfolgreicher Simulation werden sie das beschriebene Modell in geeigneter Weise in Hardware auf einem FPGA umsetzen, um die tatsächliche Funktion zu validieren. Um das Praktikum abwechslungsreicher zu gestalten wurde vom gewohnten Ablauf

- Simulation
- Verifizierung
- Implementierung auf der Zielhardware

des Systementwurfs abgewichen und alle drei Entwurfs-Schritte zusammenhängend für jedes Modul einzeln durchgeführt.

2 Konventionen

2.1 Vorbereitung

Als Grundlage für dieses Praktikum dient die Vorlesung “Architekturen der digitalen Signalverarbeitung”. Allerdings ist es nicht zwingend notwendig diese besucht zu haben, da der zur erfolgreichen Durchführung des Praktikums notwendige Stoff in den jeweiligen Grundlagenkapiteln der einzelnen Versuche noch einmal aufgearbeitet wird. Sie sollten aber in jedem Fall die entsprechenden Kapitel vor Beginn des Praktikums aufmerksam durcharbeiten, damit sie die eigentliche Versuchszeit zur Realisierung nutzen können.

Sollten sie bisher noch keinen Kontakt mit MATLAB gehabt haben, sollten sie das MATLAB-Tutorial im Anhang A durcharbeiten.

2.2 Warnungen und Hinweise

Warnungen und Hinweise werden im Skript hervorgehoben.



Warnungen Hier besteht Gefahr, etwas zu beschädigen oder sich zu verletzen.



Hinweise Hier wird auf Fehlermöglichkeiten hingewiesen

3 Zur Struktur

Das Skript beinhaltet neben der Praktikumsanleitung ein MATLAB-Tutorium im Anhang A und folgt in der Regel der nachstehenden Struktur.

1. Vorüberlegungen

- Konzepte
- Realisierungsmöglichkeiten finden
- Auswahl einer geeigneten Variante
- Gliederung in Teilprobleme

2. MATLAB

- Programmierung der Module
- Aufbau des Modells/der Simulation
- Simulation des Entwurfs

3. VHDL

- Verhaltensmodelle der Module entwerfen
- Simulation der einzelnen Module
- Simulation des Gesamtentwurfs
- Beschreibung der Hardware (RTL)
- Simulation des RTL Entwurfs
- Implementierung in Hardware
- Praxistest und Fehlersuche
- Versuchsdurchführung

Gegen Mitte des Praktikums wurde der MATLAB-Teil zusammengefasst und die Struktur nur für VHDL beibehalten. Die theoretische Simulation steht am Anfang und kann komplett vollzogen werden. Bei VHDL ist es wichtig, die einzelnen Module getrennt zu verifizieren, da die fehlerfreie Funktion erheblich schwieriger am Gesamtsystem nachzuweisen ist.

4 Die Hardware

4.1 Kurzbeschreibung

Beim SPATES¹ handelt es sich um ein Entwicklungssystem für FPGAs der Serie ECP[6] von Lattice. Weiterhin befinden sich noch Komponenten auf der Leiterplatte, die einerseits für die Funktion des FPGA (Schaltregler für die verschiedenen Versorgungsspannungen, Taktversorgung, usw.), andererseits für die Versuche (AD-, DA-Wandler, Mikrofonverstärker, usw.) notwendig sind.

4.2 Vorbereitung der Hardware

Das SPATES besitzt nur relativ wenige Anschlussmöglichkeiten (vgl. Abb. 4.1 gelb hinterlegt.)



Bei der Handhabung ist Vorsicht geboten. Es handelt sich fast ausschließlich um Bauteile in CMOS Technologie. Zwar besitzen die Schaltkreise inzwischen sehr gute ESD-Schutzschaltungen, dennoch können diese durch unsachgemäße Handhabung zerstört werden. Da die Bauteile teuer sind, bitte vorher das PC-Gehäuse anfassen und sich entladen. Dann sollte nichts passieren.

Zuerst schalten sie Ihr Netzgerät ein und stellen beide Spannungen auf 15V. Wieder ausmachen und eine Brücke wie in Abb. 4.2 einstecken, so dass 30V Versorgungsspannung anliegen. Mit dem Adapterkabel (vgl. Abb. 4.3) verbinden sie das Netzgerät mit dem SPATES. Der Stecker geht manchmal etwas schwerer, allerdings sollte man keine Gewalt oder ein Messer anwenden müssen.

Anschließend können sie das Netzteil wieder einschalten. Zuerst sollten ihnen die verschiedenen Versorgungsspannungs-LEDs auffallen, die eine Funktion der einzelnen Spannungsebenen signalisieren (oder eben auch nicht, dann beim Betreuer melden). Nach einem kurzen Bootvorgang, der durch 2 LEDs direkt neben dem FPGA signalisiert wird, startet auch schon das Diagnose-Programm. Hierbei werden alle LEDs, die 7-Segment-Anzeigen und die Bargraphs kurz angesteuert. Falls der Lautsprecher schon angeschlossen ist, müsste danach ein kurzer Sweep² zu hören sein. Ist dieser Test abgeschlossen, so kann man interaktiv alle weiteren Bedienelemente testen. Dabei gilt die Zuordnung aus Tabelle 4.1.

¹SPATES abk. Signal Processing And Transmission Experiments System

²Durchfahren eines Frequenzbereichs z.B. 20Hz bis 20kHz

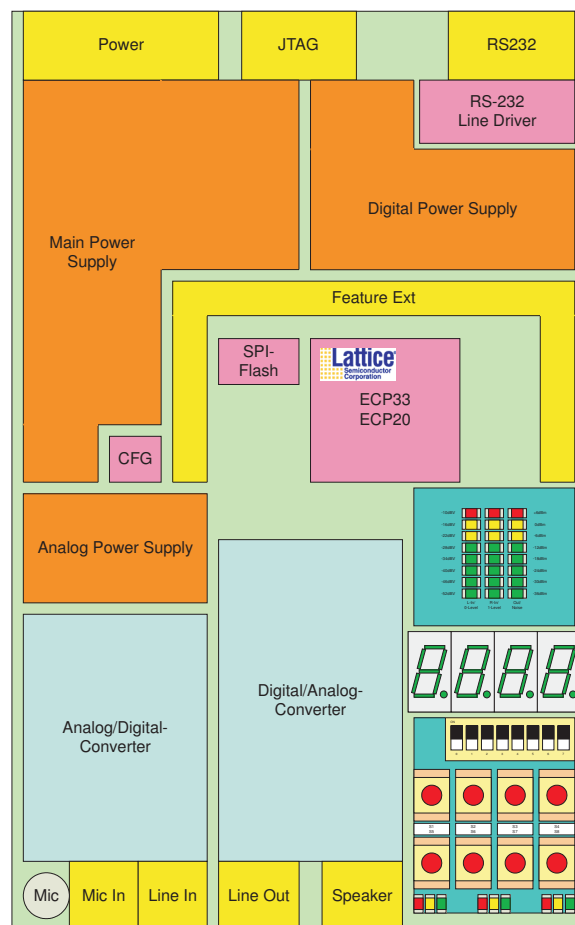


Abbildung 4.1: ADSP-SPATES

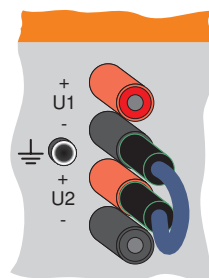


Abbildung 4.2: Netzteil-Brücke

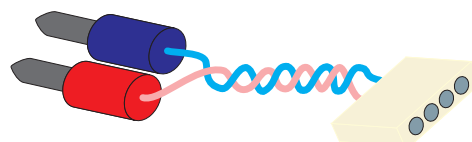


Abbildung 4.3: Power cable

Schalter/Taster	Anzeigeelement
S1	Alle Segmente LD1
S2	Alle Segmente LD2
S3	Alle Segmente LD3
S4	Alle Segmente LD4
S5	Alle Segmente Bargraph 1,2
S6	Status 1R, 1Y, 1G
S7	Status 2R, 2Y, 2G
S8	Status 3R, 3Y, 3G
DIP 1-8	Bargraph 1,2 LEDs 1-8

Tabelle 4.1: Zuordnungen der Taster und Schalter

Teil II

Versuche

5 Sender

Mittels des in Abb. 5.1 dargestellten Systems werden die zu übertragenden Signale erzeugt. Es besteht aus einem Pseudo-Zufallszahlen-Generator, dessen Ausgangssignal mit Hilfe eines einfachen FSK-Modulators moduliert werden soll. Eine detailliertere Beschreibung der einzelnen Komponenten wird in den folgenden Kapiteln erarbeitet.

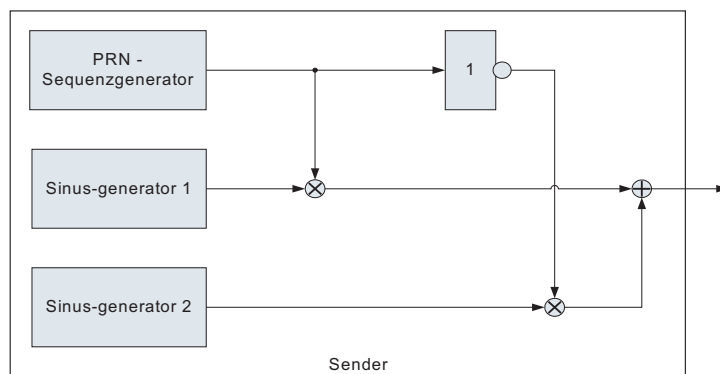


Abbildung 5.1: Sender Blockschaltbild

5.1 Versuch 1: Erste Schritte

5.1.1 Konzept

In diesem Versuch soll eine einfache Umschaltung von verschiedenen Quellen auf eine Senke realisieren. Es soll ein Signal aus 2 Sinusschwingungen, dem AD-Umsetzer und Stumm ausgewählt und dies auf den DA-Umsetzer durchgeschaltet werden können (vgl. Abb. 5.2).

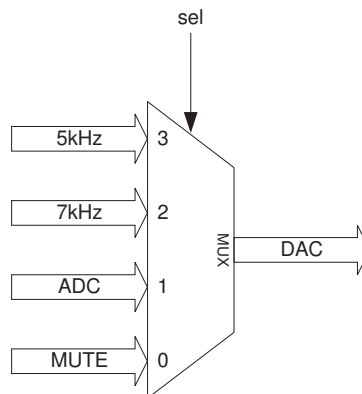


Abbildung 5.2: 8-bit 4-zu-1-Multiplexer

5.1.2 Realisierungsmöglichkeiten

Grundsätzlich hat man in VHDL Möglichkeiten, getaktete (synchrone) und nicht getaktete (asynchrone) Vorgänge zu beschreiben. Dazu benötigen wir wenige grundlegende Konstrukte. In vielen Fällen ist eine asynchrone Schaltung einfach durch ein Register am Ausgang in eine synchrone Schaltung umzuwandeln (vgl. Abb. 5.3). Man sollte allerdings bedenken, dass die Optimierung nur innerhalb einer Hierarchieebene effizient funktioniert. Wegen der besseren Testbarkeit wird der gesamte Code in Module gegliedert und eine Hierarchie aufgebaut. Die ist günstig für die Synthese und die Optimierungsalgorithmen die dieser zugrunde liegen. Werden nun lange Signalwege ohne Register über mehrere Modulgrenzen hinweggeführt, so kann der Algorithmus der Synthese nicht so effizient arbeiten. Daher sollten an den Grenzen der Module nach Möglichkeit Register verwendet werden. Für weitere Informationen zur effizienten Programmierung, siehe [7], Abschnitt „HDL Synthesis Coding Guidelines for Lattice Semiconductor FPGAs“. Diese Empfehlungen treffen nicht nur auf die FPGAs von Lattice zu, sondern sind allgemein anwendbar.

5.1.3 VHDL: Realisierung

5.1.3.1 VHDL-Basics: Libraries

Libraries in VHDL dienen hauptsächlich zur Definition von Typen, Funktionen und für die Definition der Resolution Functions. Typen geben an, wie die Informationen eines Signals oder einer Variablen dargestellt werden und welchen Wertebereich diese besitzen.

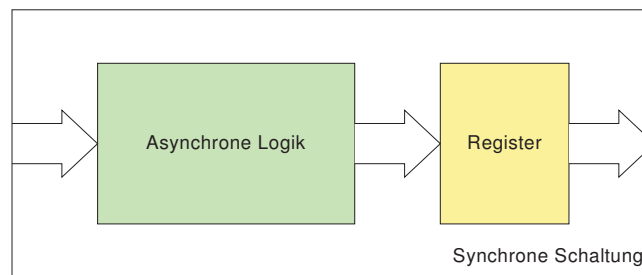


Abbildung 5.3: Synchrone Schaltung

Funktionen sind vergleichbar den Funktionen in Software. Es ist möglich sie mit Parametern aufzurufen und einen Wert zurück zu erhalten. Dies kommt beispielsweise bei der Umwandlung von Typen in andere zum Einsatz. Der meistgebräuchlichste Typ bei der Beschreibung für FPGAs ist der `std_ulogic`- bzw. der `std_logic`-Datentyp, wobei der `std_ulogic` unresolved und der `std_logic` resolved ist. Resolved bedeutet, dass es möglich ist, ein Signal von mehreren Treibern ansteuern zu können, wie es bei einem Bussystem der Fall ist, unresolved Typen können hierzu nicht verwendet werden. Die möglichen Werte der beiden genannten Typen sind Tabelle 5.1 zu entnehmen.

Wert	Bezeichnung	Erklärung
'U'	Not initialized	Dem Signal wurde noch kein Wert zugewiesen
'X'	Forcing Unknown	Das Signal wird gegenläufig getrieben
'0'	Forcing 0	Dies entspricht einem LOW-Pegel
'1'	Forcing 1	Dies entspricht einem HIGH-Pegel
'Z'	High Impedance	Das Signal wird nicht getrieben
'W'	Weak Unknown	Das Signal wird schwach auf Unknown gehalten
'L'	Weak 0	Das Signal wird schwach auf LOW-Pegel gehalten
'H'	Weak 1	Das Signal wird schwach auf HIGH-Pegel gehalten
'-'	Don't care	Der Wert des Signals ist zu ignorieren

Tabelle 5.1: Der Typ `std_logic`

Diese Werte sind in der Library `ieee.std_logic_1164` definiert. Einige weitere wichtige Libraries werden im Folgenden in die Projekte eingebunden:

```
library ieee
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;
use ieee.numeric_bit.all;
```

Teilweise kann deren Quellcode in den Dateien, welche vom Herstellertool mitgeliefert wurden, eingesehen werden. Bei ispLEVER sind diese im Verzeichnis `C:\ispTOOLS5_1\synpbases\lib\vhd\` zu finden.

Weitere Typen lernen Sie zu gegebener Zeit kennen.

5.1.3.2 VHDL-Basics: Die Entity

Da der Multiplexer angeschlossen werden muss, definieren wir eine entity, die gerne mit dem Sockel eines ICs verglichen wird. Eine Entity wird folgendermaßen beschrieben:

```
entity <NAME> is  
  
  [generic ( <VARNAME>: <VARTYPE> := <STD_VALUE>[; <VARNAME> ...]);]  
  port ( <SIGNAL>: <DIRECTION> <TYPE>[; <SIGNAL>:]);  
  
end;
```

Für die *DIRECTION*¹ sind vorläufig nur *in* und *out* interessant.

Damit ist festgelegt, wie ein Modul aussieht und einzubinden ist. Die Generics dienen dazu, das Modul konfigurierbar zu machen und sind nicht notwendig, der *port* hingegen beschreibt die Pins und ist somit unabdingbar. Um nicht jedes Bit einzeln verdrahten zu müssen, wurde ein weiterer Typ eingeführt: *std_logic_vector*

Dabei handelt es sich um ein Array von *std_logic* mit dem ein einfacher Bus dargestellt werden. Die Größe dieses Arrays kann folgendermaßen festgelegt werden:

```
...  
  my_bus: in std_logic_vector(31 downto 0);  
...
```



Üblicherweise werden Vektoren immer absteigend definiert (MSB soll links sein, auch bekannt als *little endian*). Im Praktikum wollen wir uns deshalb auf *downto* beschränken.

5.1.3.3 VHDL-Basics: Die Architecture

Die *architecture* wird benötigt, um der Entity Leben einzuhauchen. Mit VHDL beschreibt man, wie sich das Modul verhalten soll. Bisher wurden nur die Anschlüsse definiert.

```
architecture <NAME> of <ENTITYNAME> is  
  [DECLARATIONS]  
begin  
  [INSTANTIATIONS]  
end;
```

In Fall des Multiplexers bekommt diese den Namen *multiplexer_behavioral* und die Entity *my_multiplexer*. Für die Simulation sind hier in der Beschreibung keine Grenzen gesetzt, solange es sich um korrekte VHDL-Syntax handelt. Will man die Beschreibung

¹Signalrichtung

jedoch synthetisieren und implementieren², so müssen bestimmte Regeln eingehalten werden. Diese Regeln sind jedoch von Tool zu Tool unterschiedlich. Unter Umständen ist es mit manchen Tools möglich, eigentlich unerlaubte Anweisungen dennoch mit vielen Tricks in lauffähige Hardware umzusetzen. Es sollte jedoch unbedingt auf diese Tricks verzichtet werden, da die Portabilität, Zuverlässigkeit und auch die Geschwindigkeit stark darunter leiden können.

5.1.3.4 VHDL-Basics: Concurrent Statements

Generell gibt es, wie weiter oben erwähnt, bei digitaler Hardware zwei Vorgehensweisen: Asynchron und synchron. Der Multiplexer ist im einfachsten Fall eine asynchrone Schaltung. Diese asynchronen Teile bezeichnet man in VHDL als **concurrent statement**, da diese nebenläufig, also alle gleichzeitig, abgearbeitet werden.

Ein solches Statement schreibt man direkt in die architecture. Dem linken Teil wird das Ergebnis des rechten Teils zugewiesen.

```
...
OUT_1 <= IN_1;           -- Einfache Verbindung
OUT_2 <= IN_1 or IN_2;    -- Logische ODER-Verknüpfung
OUT_3 <= not IN_3;        -- Invertierung
...
```

Weiterhin stehen Ihnen alle nachfolgenden logischen Operationen zur Verfügung.

xor Verknüpft den linken mit dem rechten Ausdruck über die XOR-Funktion

and Eine UND-Verknüpfung des linken und rechten Ausdrucks

or Die ODER-Verknüpfung

not Invertiert den rechts stehenden Ausdruck



Auch *concurrent statements* haben eine Verzögerung, wenn Logik im rechten Anweisungsteil enthalten ist.

In Concurrent Statements können auch Multiplexer realisiert werden.

```
...
OUT_1 <= IN_1 when SEL_1 = '1' else
      IN_2 when SEL_2 = '1' else '0';
...
```

5.1.4 VHDL: Beschreibung der Hardware

Aufgabe 1 Zunächst soll ein Multiplexer in VHDL implementiert werden. Sie finden einen vorbereiteten Code-Rahmen im Verzeichnis 01_Multiplexer\my_multiplexer.vhd.

²Abrebitschritte vom VHDL-Code zur lauffähigen Hardwarearchitektur

Der Abschnitt, der in der Datei mit `--WRITE HERE` bezeichnet ist, soll durch Ihren Code ersetzt werden.

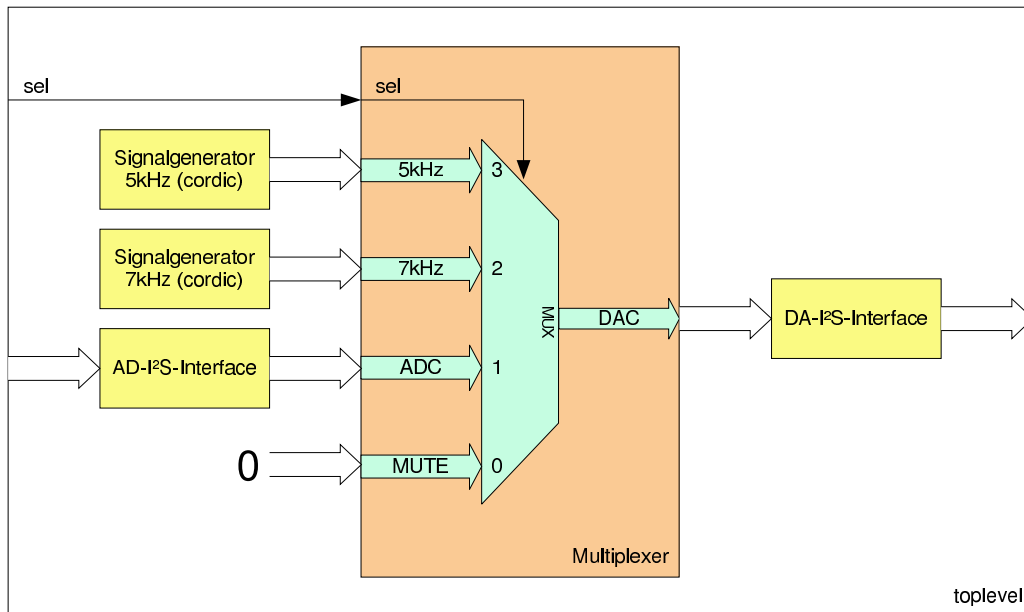





Abbildung 5.4: Der Toplevel des Multiplexers


5.1.5 VHDL: Implementierung in Hardware

Um den VHDL-Code auf dem FPGA auszuführen, muss die Beschreibung auf die FPGA-spezifische Technologie abgebildet, also übersetzt werden. Dies erledigt das Synthese-Tool von ispLEVER.

Aufgabe 2: Speichern sie alle geöffneten Dateien. Anschließend lassen sie das Projekt synthetisieren, indem sie das Toplevel auswählen und den Prozess

 **Synplify Synthesize VHDL File** im Aktionsbereich der Oberfläche starten. Synplify erzeugt eine Netzliste, mit der ispLEVER weiterarbeiten kann. Sind keine Fehler aufgetreten, wählen sie im Hierarchie-Fenster das FPGA  **LFEC20E-3F484C** aus. Für die Programmierung wird ein Bitstream benötigt, den sie auf die Hardware übertragen können. Wählen sie die Aktion  **Generate Bitstream Data** aus, so führt ispLEVER alle notwendigen Prozesse in der richtigen Reihenfolge aus, und erzeugt das Bitfile.

5.1.6 TEST: Praxis

Aufgabe 3: Übertragen sie das Bitfile in das FPGA. Dazu starten sie das Programm ispVM (Symbol  in der Toolbar). (s. Abb 5.5)

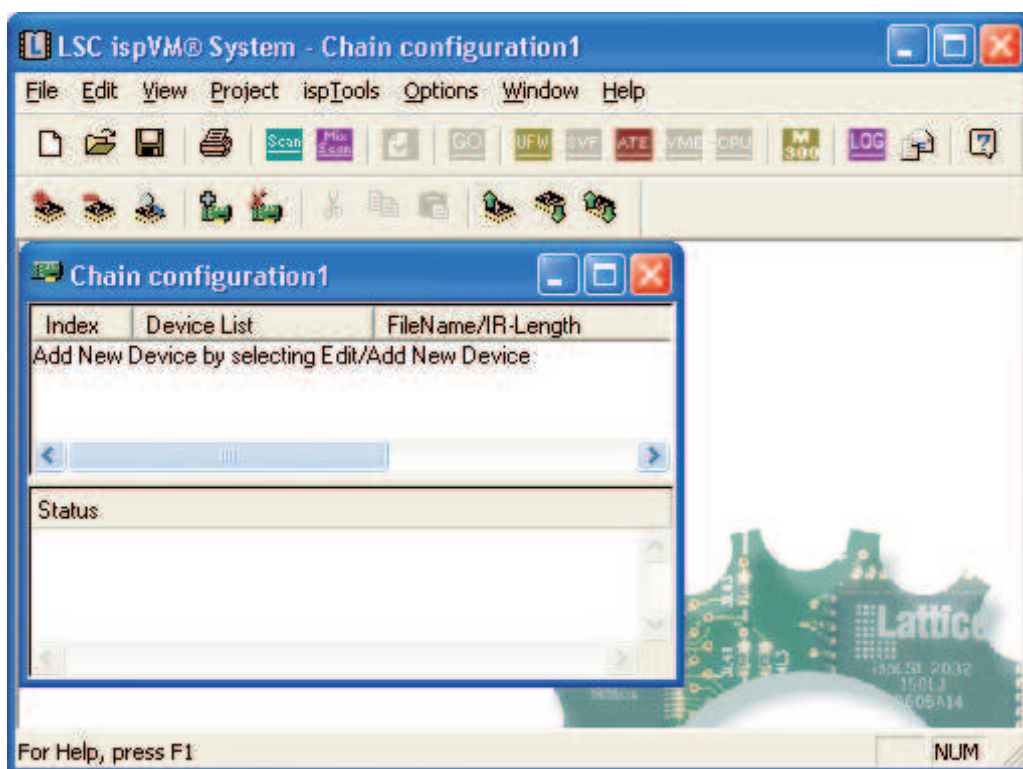


Abbildung 5.5: Das ispVM-Fenster

5.2 Versuch 2: Zufallsfolgenergenerator

5.2.1 Konzept

Zur Validierung von digitalen Schaltungen ist es oft wünschenswert, die Stimuli³ möglichst zufällig aber dennoch reproduzierbar zu erzeugen. Durch die zufälligen Folgen wird so eine möglichst gleichmäßige Verteilung über den Wertebereich erreicht, der eine gute Testabdeckung⁴ ermöglicht. Durch Verwendung von Pseudo-Zufallsfolgen-Generatoren können somit die zu erwartenden Ergebnisse allein durch Kenntnis des Startwertes ermittelt werden.

5.2.2 Realisierungsmöglichkeiten

Es gibt verschiedene Möglichkeiten eine sich ständig wiederholende Zufallsfolge zu erzeugen. Im einfachsten Fall generiert man einen Vektor, der die gewünschte Folge enthält und ruft ihn immer wieder auf. Zugegebenermaßen ist diese Methode alles andere als elegant. Im Praktikum werden wir eine schönere Möglichkeit verwenden, indem wir PRN-Sequenzen⁵ durch ein Schieberegister erzeugen lassen.

5.2.2.1 PRN-Schieberegister

PRN-Sequenzen sind keine echten Zufallszahlen, sondern pseudo Zufallszahlen. Es sind Signale, die den Anschein einer zufälligen Folge von Nullen und Einsen erwecken, aber - bei der gleichen Anfangsbelegung der Schieberegister - reproduzierbare Ergebnisse liefern. Diese Tatsache kann man sich später zur Validierung der empfangenen Daten zunutze machen.

Zur Erzeugung der PRN-Folgen greifen wir auf ein N-Bit-Schieberegister zurück, bei dem der Eingang durch eine XOR-Verknüpfung aus beliebigen Bits des Schieberegisters gebildet wird (vgl. Abb. 5.6).

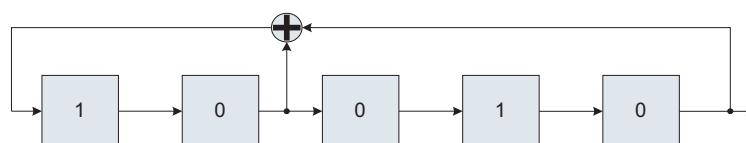


Abbildung 5.6: PRN-Schieberegister

Auf diese Weise lässt sich mit einem N-Bit-Schieberegister eine $2^N - 1$ Bit lange Folge erzeugen. Wie man gut im Beispiel auf Seite 19 erkennen kann, beginnt das Register nach einem vollständigen Durchlauf wieder von vorne (siehe Abb. 5.7). Anwendung in der Praxis findet diese Technik zum Beispiel beim Satellitenortungssystem GPS⁶ zur Erzeugung der

³Steuernde Eingabe eines Systems

⁴Testabdeckung beschreibt das Verhältnis von möglichen Fehlern zu den getesteten Zuständen

⁵Pseudo-Random-Noise Sequenzen

⁶GPS - Global Positioning System

CA-Codes⁷ der einzelnen Satelliten (Detaillierte Informationen zur Verwendung in GPS [9]).

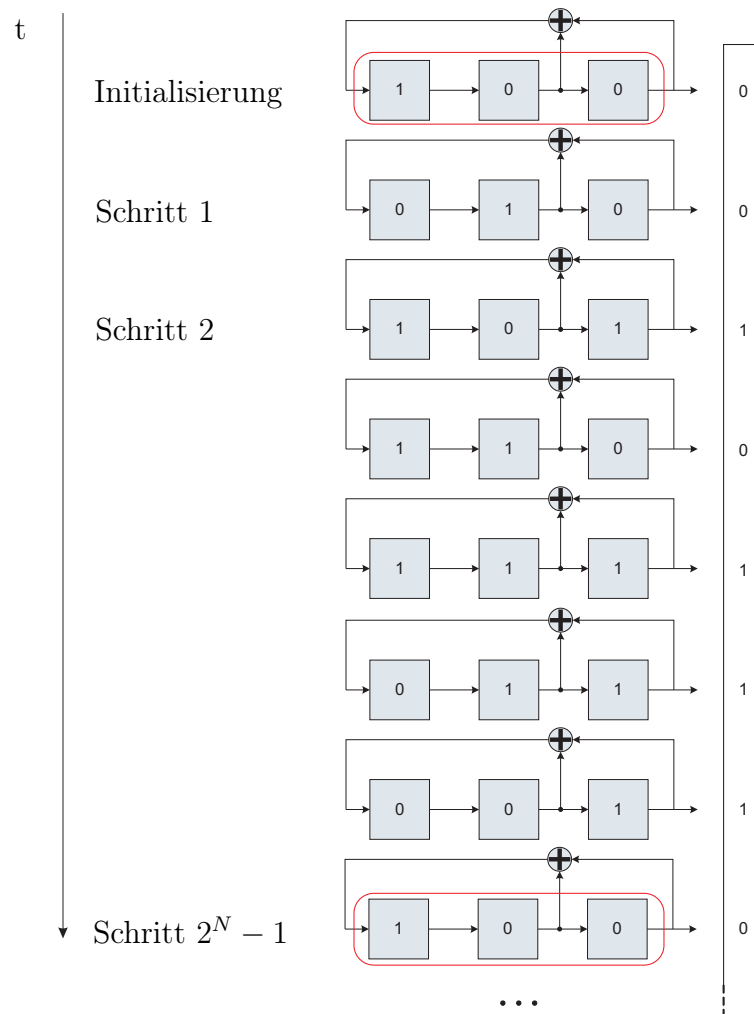


Abbildung 5.7: PRN-Schieberegisterdurchlauf

5.2.3 MATLAB: Programmierung

Vorbemerkung zur MATLAB-Programmierung: Um ein Gefühl für MATLAB zu bekommen werden sie in den ersten Versuchen den Großteil der Skripte und Funktionen vorgegeben bekommen und nur die wichtigen Stellen selbst programmieren müssen. Da sie aber den Umgang mit MATLAB erlernen sollen, werden sie im weiteren Verlauf des Praktikums immer mehr dazu angehalten werden, die notwendigen Skripte und Funktionen selbst zu erstellen. Die ersten Versuche sollen ihnen hierfür als anschauliches Beispiel dienen.

⁷Verwendet zur Identifizierung einzelner Satelliten

Los geht's mit den ersten Gehversuchen in MATLAB: Vervollständigen sie die im Verzeichnis

`MATLAB_src\sender\PRNSequenzgenerator`

vorliegende MATLAB-Funktion *PRNseq.m*, mit deren Hilfe - wie oben beschrieben - ein pseudozufälliger Datenstrom erzeugt werden kann.

Überprüfen sie die Arbeitsweise ihrer Funktion, indem sie diese in einem MATLAB-Skript aufrufen, ihr die nötigen Parameter übergeben und sich den ausgegebenen Vektor als Balkendiagramm darstellen lassen. Dies erreichen sie in MATLAB mit der Plot-Funktion *bar.m*.

Lassen sie sich außerdem die Anzahl der ausgegebenen Einsen und Nullen im Command-Window anzeigen.

5.2.4 VHDL: Realisierung

5.2.4.1 VHDL-Basics: Signale

Bisher wurden ausschließlich Signale verwendet, die durch die Entity bereits vorgegeben bzw. deklariert waren. Sogenannte Signale ermöglichen es, neue Namen zu definieren, die dann mit Werten belegt werden können. Für den Anfang werden lediglich die zwei schon bekannten Typen benötigt.



Bei der Bezeichnung der Signale empfiehlt es sich, dem Signalnamen ein Kürzel voranzustellen, welches Auskunft über dessen Typ gibt. Alle Signale sollten klein geschrieben werden, um sie besser von Ports unterscheiden zu können.

Präfix	Bedeutung
sl	S ignal std_ l ogic
sv	S ignal std_ l ogic_ v ector

Tabelle 5.2: Vorgeschlagene Präfixe der bisher bekannten Typen

Um einem Vektor einen Wert zuweisen zu können, wird eine Möglichkeit benötigt, Arrayelemente verknüpfen zu können, um nicht jedem Element in einer eigenen Anweisung einen Wert zuweisen zu müssen. Hierzu gibt es mehrere Möglichkeiten, von denen je nach Anwendungsfall die geeignete auszuwählen ist.

```
...
architecture ...
...
    signal sv_bus: std_logic_vector(2 downto 0);
begin
...
    sv_bus(0) <= '1';
    sv_bus <= (others => '1'); -- Allen Elementen wird '1' zugewiesen
    sv_bus <= "010";
```

```

sv_bus <= '1' & '0' & '1';
sv_bus <= '0' & "01"
sv_bus(2 downto 1) <= "01";
...
end;

```

Signale haben in der gesamten architecture Gültigkeit.

5.2.4.2 VHDL-Basics: Sequentielle Prozesse

Prozesse sind Teile des Sourcecodes, die synchron ablaufen sollen. Bisher wurden lediglich asynchrone Konstrukte (concurrent statements) verwendet. Besagte Prozesse sind im Grunde nichts anderes als ein großes concurrent statement. Mehrere Prozesse sind daher zueinander nebenläufig, werden also gleichzeitig ausgeführt.



Prozesse sind nicht vergleichbar mit Programmteilen aus dem Software-Sektor, in denen jeder Befehl nacheinander abgearbeitet wird. In Prozessen passiert alles gleichzeitig!

Ein Prozess wird wie folgt beschrieben:

```

...
[<name> :] process [(<SIGNAL1>, <SIGNAL2>)]
begin
...
-- CODE
...
end;
...

```

Der Name des Prozesses ist optional. Die sogenannte *sensitivity list*⁸ gibt die Signale an, auf die der Prozess reagieren soll. Dies ist lediglich für die Simulation von Bedeutung, die den Prozess nur dann abarbeitet, wenn eines der genannten Signale seinen Wert ändert, die Synthese beachtet diese Liste nicht sondern orientiert sich am Code im Prozess selbst.

Um den Prozess synchron auszuführen, benötigt man noch ein Clock- und ggf. ein Reset-Signal. Jeder sequentielle Prozess reagiert zumindest die Flankenänderung des Clock-Signals. Ein Reset ist optional, sollte aber verwendet werden, um einen definierten Zustand nach dem Einschalten zu erhalten.

Ein typischer sequentieller Prozess sieht folgendermaßen aus:

```

...
seq_proc : process (CLK, nRESET)
begin

```

⁸sensitivity list: Signalnamen in Klammern hinter *process*. Bedingte Abarbeitung des Prozesses in der Simulation, bedingt durch die aufgeführten Signale.

```

if nRESET = '0' then
    -- CODE
elseif rising_edge(CLK) then
    -- CODE
end if;
end;

```



Meist verwendet man einen invertierten RESET. Dies folgt daraus, dass alle Signale im Einschaltzustand üblicherweise '0'-Pegel besitzen. Diese Invertierung kennzeichnet man gewöhnlich durch das „n“ vor dem Signalnamen. Es stellt eine weitere freiwillige, aber sinnvolle Konvention für die Programmierung in VHDL dar.

Alternativ für das oben verwendete *rising_edge*-Makro, wird manchmal folgendes schlechter lesbare Konstrukt eingesetzt:

```

...
elsif CLK'event and CLK = '1' then
...

```

Dies folgt daraus, dass einige Tools das *rising_edge*-Makro nicht kennen. Sollte bei der Synthese aufgrund dieses Makros ein Fehler auftreten, wenden sie sich bitte an den Betreuer. Die Meldung müsste aussagen, dass die Synthese diese Funktion nicht kennt.



Wir verwenden in unserem Praktikum das Makro *rising_edge* um die Lesbarkeit des Codes zu verbessern.

5.2.4.3 VHDL-Basics: if-then-else

Im vorherigen Abschnitt wurde bereits das if-then-else-Konstrukt verwendet. Dies ist nicht nur für die Clock-Flanken-Erkennung nützlich sondern ermöglicht auch das Programmieren bedingter Zuweisungen. Allerdings ist es nur in Prozessen zulässig. In concurrent statements muss man sich auf when-else, wie beim Multiplexer verwendet wurde.

Das if-then-else-Konstrukt ist wie folgt definiert:

```

...
if <BEDINGUNG> then
    -- CODE
[elsif <BEDINGUNG> then
    -- CODE ]
[else
    -- CODE ]
end if;
...

```

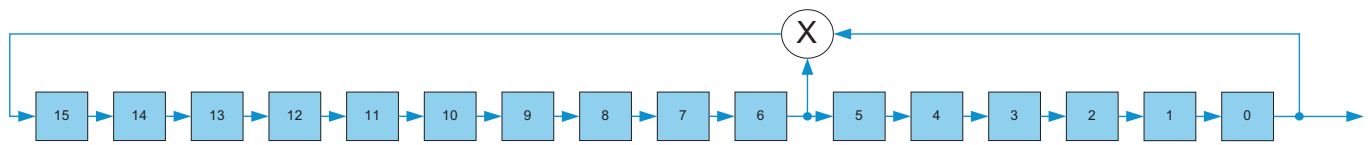


Abbildung 5.8: 16-stufiges Schieberegister mit XOR-Rückkopplung



Immer alle Fälle abdecken, ansonsten *else* verwenden. Ausnahme: Clock-Flankenerkennung.

5.2.5 VHDL: Beschreibung der Hardware

Aufgabe 1: PRN-Schieberegister Es soll ein PRN-Schieberegister verwirklicht werden. In der Datei

02_Randomize\prn_shifter.vhd

finden sie den benötigten, vorgefertigten Sourcecode, um ein solches PRN-Schieberegister zu programmieren. Das Register soll 16 Bit lang sein, und das XOR-Element soll zwischen Ausgang und der 6. Stelle eingefügt werden (vgl. 5.8).

Als kleine Hilfestellung sei noch erwähnt, dass sie Ausgabe-Ports (out) nur zuweisen, nicht jedoch lesen können. Es ist daher nötig, ein Signal zu definieren, mit dem sie im Code arbeiten können und dem Ausgangs-Port dann den Wert dieses Signals zuzuweisen.

Aufgabe 2: Anfangszustand Welchen Anfangswert muss das Schieberegister haben bzw. darf es nicht haben, um die Funktion sicherzustellen

zu

setzen. Dann ergibt sich die Folge: TODO

5.2.6 MODELSIM: Simulation der Beschreibung


5.2.6.1 Einführung in Modelsim

Modelsim ist ein Simulationstool, mit dessen Hilfe sie eine VHDL-Beschreibung in praktisch jeder Abstraktionsebene durchführen können. Es unterstützt die behavioral-Beschreibung, die syntetisierte Netzliste und auch die implementierte Beschreibung auf Gatterebene mit allen Timings. Es ist also eine vollständige Simulation eines Moduls möglich. Die Simulation der kompletten Hardware kann jedoch nur selten erfolgen, da einerseits die Stimuli oft nicht hinreichend genau bekannt sind und die Simulation selbst äußerst viel Zeit und Rechenaufwand beanspruchen würde.




Einzelne Module können schnell mit der Simulation verifiziert werden. Komplette Hardware nur eingeschränkt bzw. gar nicht

Für die Simulation muss eine sogenannte Testbench generiert werden. Viele Hersteller-Tools besitzen dafür ein grafisches Frontend mit, das jedoch bei weitem nicht die Möglichkeiten einer VHDL-Testbench bietet. Daher werden im Praktikum ausschließlich eine VHDL-Dateien verwendet, die keinerlei Ports nach außen haben und das DUT⁹, also unser Modul, als Komponente einbindet. Die Generierung der Stimuli und auch ggf. die Auswertung der Antworten erfolgt ebenfalls in VHDL. In der Simulation sind im Gegensatz zur Synthese alle VHDL-Konstrukte erlaubt, was es uns gestattet, auf einer sehr hohen Abstraktionsebene zu programmieren.

Um nicht für jede zu testende Datei eine eigene Testbench bauen zu müssen, bietet die Entwicklungsumgebung von ispLEVER einen Testbench-Generator an, mit dessen Hilfe schnell eine eigene Testbench erzeugt werden kann. Dazu das zu testende VHDL-Modul im Hierarchie-Fenster von ispLEVER anwählen und im Aktionsfenster den Punkt  **VHDL Test Bench Template** auswählen. Die Ausgabe im Fenster ist anschließend in eine Datei zu kopieren und mit geeigneten Stimuli zu versehen.

Importieren sie nun die Testbench-Datei in ispLever, wobei sich diese unter dem zu testenden VHDL-File einordnet. Um Modelsim zu starten, wählen sie die Testbench in ispLever aus und starten sie die Aktion  **VHDL Functional Simulation** wie in Abb. 5.9

Nun startet Modelsim (vgl. Abb 5.10) und sie können anfangen, ihre Beschreibung zu simulieren. Mit den Standardeinstellungen gestartet, wird bereits $1\mu s$ simuliert, sofern keine syntaktischen Fehler in der Beschreibung vorhanden sind. Sollte dies der Fall sein, so korrigieren sie die fehlerhafte VHDL-Datei und lassen sie diese erneut von Modelsim compilieren. Entweder sie schließen Modelsim und starten es über ispLEVER erneut oder sie recompilieren in Modelsim die Datei neu (vgl. Abb. 5.11). Sie finden die Datei im Reiter **Work** im linken Teil des Modelsim-Fensters und mittels der rechten Maustaste erhalten sie ein Kontextmenü mit dem entsprechenden Befehl **recompile**. Anschließend sollten sie den Befehl **restart** auswählen um die compilierten Versionen in den Speicher zu laden und erneut einen Zeitschritt mittels **run** simulieren lassen.

Alternativ können sie die Befehle auch in das Kommandofenster eintragen und mit Enter bestätigen bzw. mit ; mehrere Befehle voneinander trennen. Mit der -Taste können sie die Historie der Befehle durchblättern. So sind wiederkehrende Befehlsfolgen sehr einfach mit wenigen Tastendrücken zu wiederholen. Beispielsweise mittels `vcom ...; restart; run 10 us` mit einem Befehl, der in der Historie auch schnell wieder gefunden werden kann.

Sollten sie noch weitere Fragen zu Modelsim haben, so wenden sie sich an den Betreuer. Eigentlich sollten keine Probleme mit der Bedienung von Modelsim entstehen. Behalten sie jedoch stets im Hinterkopf, dass Simulation und Hardware nur dann übereinstimmen, wenn einerseits die Timings von der Synthese eingehalten werden können und andererseits die Simulation die von ihr benötigten Parameter auch erhält (z.B. die sensitivity list).

⁹DUT: Device Under Test

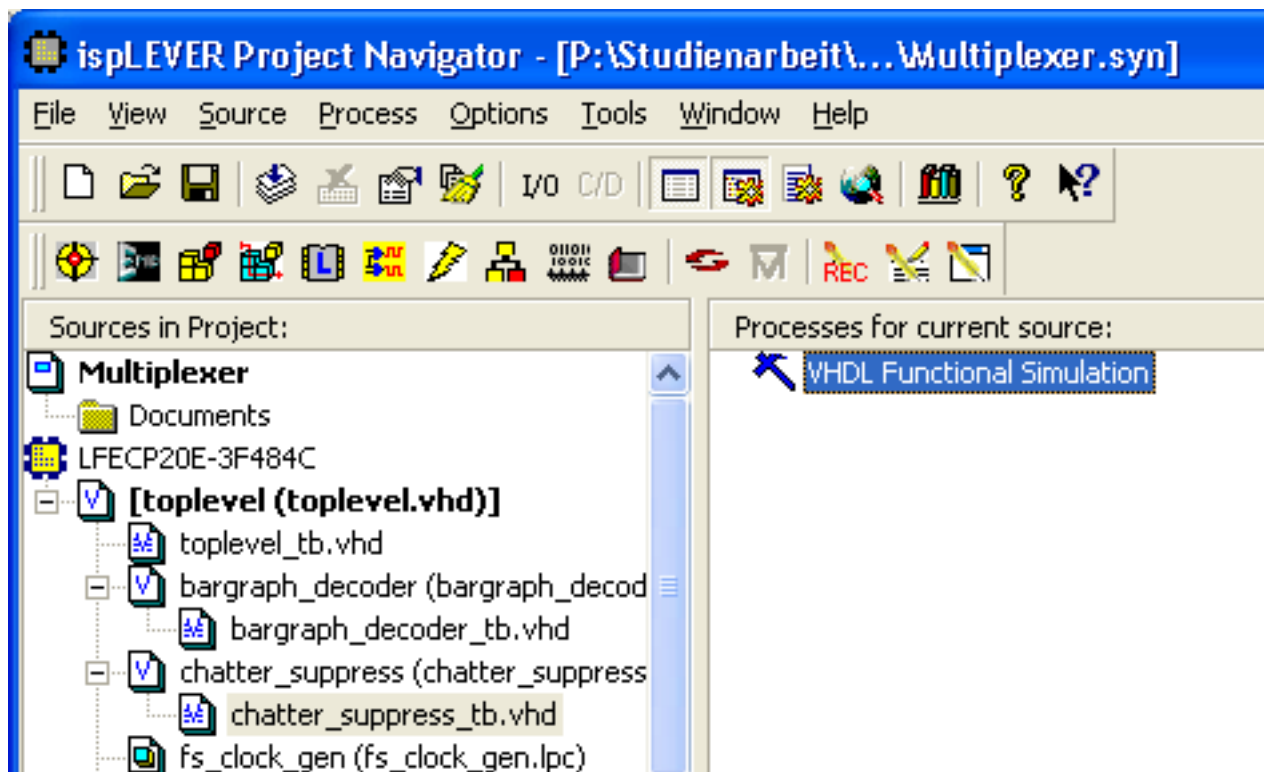


Abbildung 5.9: Testbench simulieren

Würde man beispielsweise das Taktsignal nicht in die sensitivity list eintragen, dann wäre die Simulation nicht funktionsfähig, die Hardware würde allerdings ohne Einschränkungen funktionieren.

Aufgabe 3: Simulation Simulieren sie nun mit Hilfe der oben beschriebenen Prozeduren den von ihnen programmierten Zufallszahlengenerator.

5.2.7 TEST: Praxis

Aufgabe 4: Digitales Rauschen War ihre Simulation erfolgreich, so synthetisieren sie ihre Beschreibung und generieren sie das Bitfile.

Nun wird das generierte File in das FPGA geladen. Mit Hilfe des Tasters S1 kann ein Takt erzeugt werden, während der Inhalt des Schieberegisters HEX-Codiert auf den LED-7-Segment-Anzeigen dargestellt wird. Es besteht alternativ zur Möglichkeit, einen langsamen, automatisch generierten Takt mittels des Schalters 0 des 8-Dip-Switch zu aktivieren.

Jeweils 4 bit (ein BDC-Wort) wurden zu einer Stelle zusammengefasst. Die Anzeige stellt somit Werte zwischen 0 und F dar. Zusätzlich werden alle Bits auf den Pegelanzeigen dargestellt. Die Linke bildet die niederwertigen (7 downto 0) Bits, die Rechte die höherwertigen (15 downto 8).

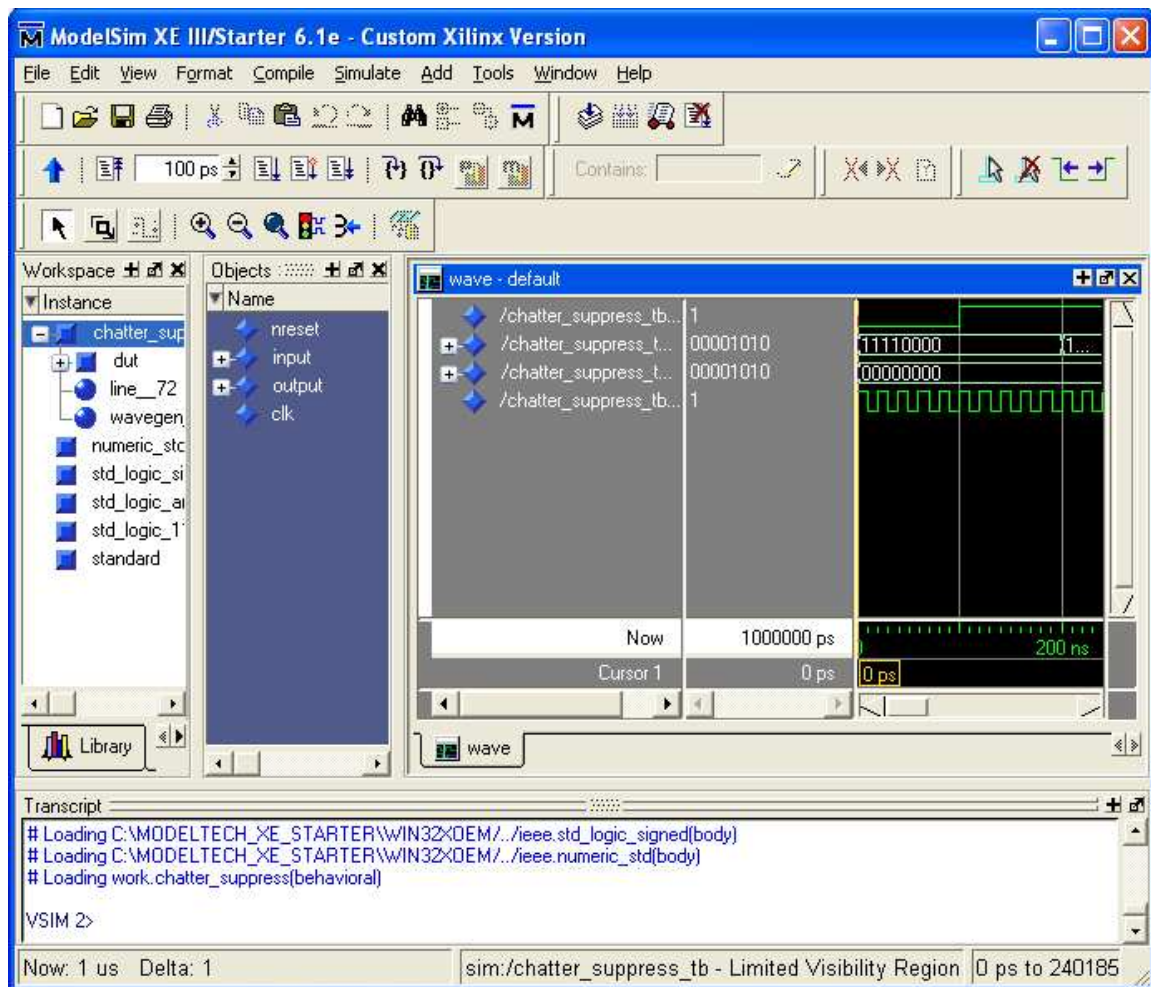


Abbildung 5.10: Modelsim nach dem Start mittels ispLEVER

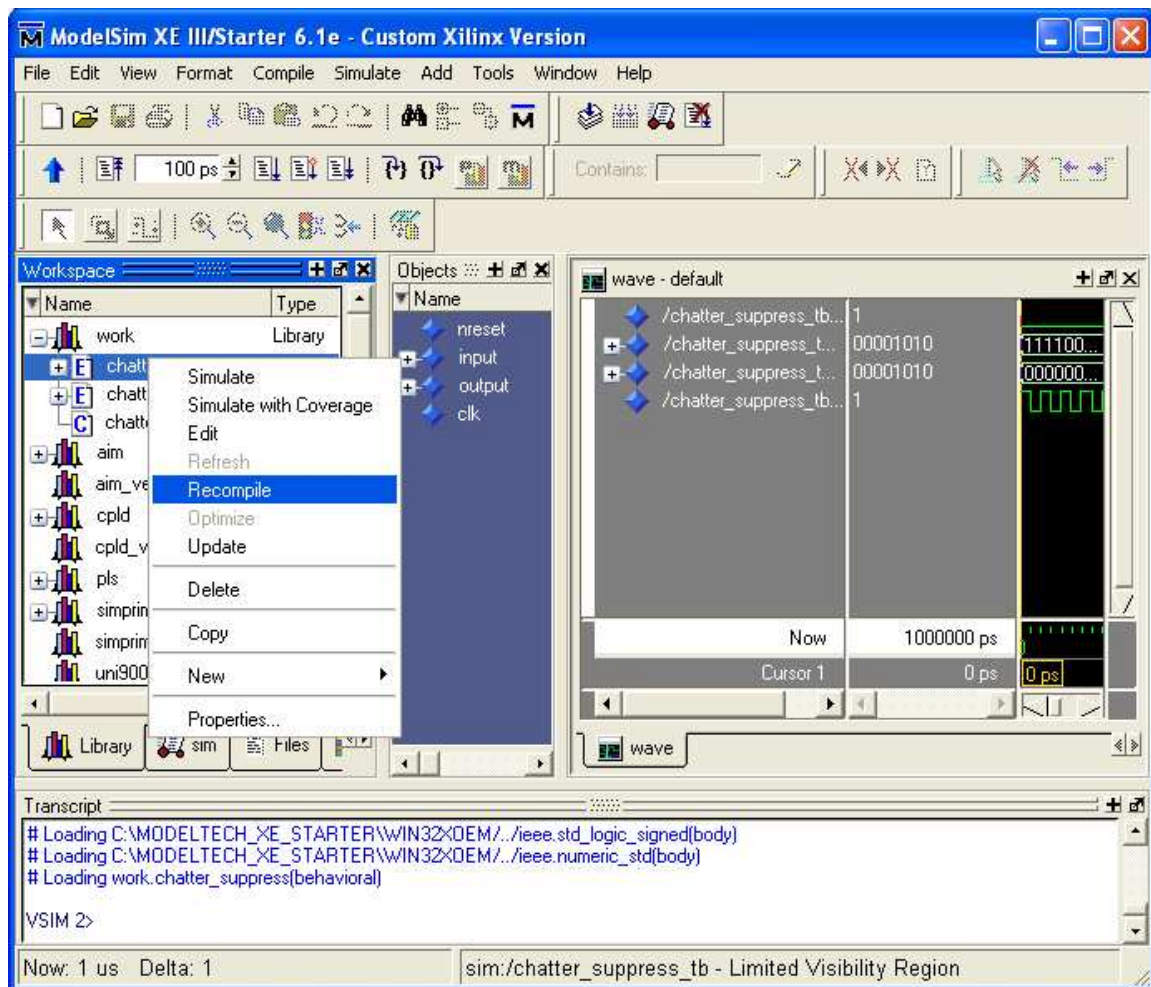


Abbildung 5.11: Erneutes Compilieren einer VHDL-Datei

Aufgabe 5: Qualität des Rauschens Schalten sie nun den automatischen Takt ein. Wie würden sie die Qualität des Rauschens beurteilen? Nur die letzte LED hat hierfür eine Relevanz.

5.3 Versuch 3: Signalgenerator

5.3.1 Konzepte

Um den erzeugten Bitstrom auf die Übertragung vorzubereiten muss man ihn, wie in Abbildung 5.1 dargestellt, noch mit zwei Sinusschwingungen modulieren. Diese können im digitalen Bereich auf verschiedene Arten erzeugt werden.

5.3.2 Realisierungsmöglichkeiten

Beispielhaft soll hier die Signalgenerierung mittels Direkter Digitaler Synthese (DDS) und die Generation per CORDIC-Algorithmus¹⁰ besprochen werden. Beide Ansätze führen zu unterschiedlichen Ergebnissen, worauf im Verlauf dieser Einführung eingegangen werden soll.

5.3.2.1 Direkte Digitale Synthese

Die Direkte Digitale Synthese (DDS) ist ein Verfahren zur Erzeugung einer (meist periodischen) Funktion, deren Funktionswerte in einem Speicher, der Look-Up-Table(LUT) abgelegt sind.

Einfache Variante: Die einfachste Variante besteht aus einem Zähler oder Sägezahn-generator, mit dessen Hilfe nacheinander die Zellen eines Speichers adressiert werden, in dem die Wertetabelle der einzelnen Signalwerte abgelegt ist. In Abbildung 5.12 ist der prinzipielle Aufbau dargestellt.

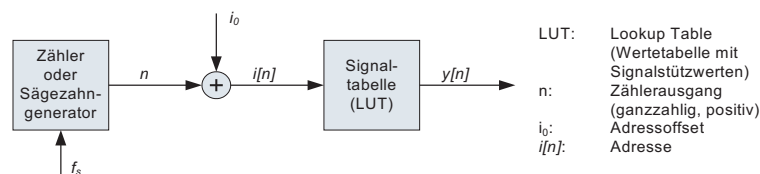


Abbildung 5.12: Direkte Digitale Synthese (DDS)

Die Funktionswerte sind in der LUT abgelegt. Die Zählvariable n adressiert den Speicher $i[n] = n + i_0$ und der ausgelesene Inhalt ergibt den gewünschten Funktionswert $y[n]$. Ist die Zählvariable n durch w_{in} Bits und der Funktionswert $y[n]$ durch w_{out} Bits repräsentiert, so benötigt man (maximal)

$$N_{Sp} = w_{out} 2^{w_{in}} \quad (5.1)$$

Speicherplätze. Mit jedem Takt der Abtastfrequenz f_s wird der Zähler inkrementiert. Kommt es zu einem Überlauf wird ein Statusbit gesetzt, worauf die Variable n erhöht wird und somit auf eine andere Speicherzelle zugegriffen wird.

Mit der oben beschriebenen Struktur sind

¹⁰Coordinate Rotation Digital Computer

- nichtlineare Kennlinien (Anregung mit reinem Aufwärtszähler) sowie
 - periodische Kennlinien (Anregung mit einer Sägezahnfunktion: Zähler mit Überlauf)
- realisierbar.

Erweiterung: Will man Signale mit variablen Abtastfrequenzen und Periodendauern erzeugen ist es nötig, die einfache Schaltung aus Abbildung 5.12 zu erweitern. Die modifizierte Schaltung ist in Abbildung 5.13 dargestellt.

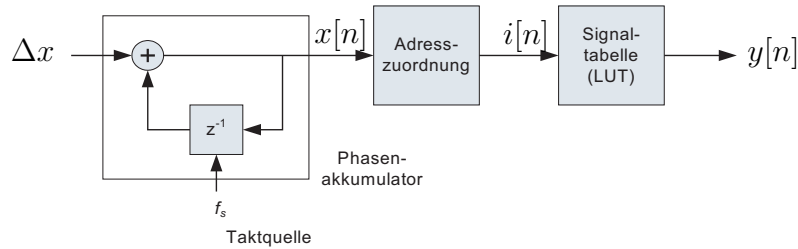


Abbildung 5.13: Erweiterte DDS

Block 'Adresszuordnung': Der Block Adresszuordnung generiert zum diskreten Zeitpunkt n eine Adresse $i[n]$, die angibt, welcher Funktionswert $y[n]$ von der LUT an den Ausgang gelegt werden soll. Die Adresse $i[n]$ wird durch das Intervall Δx_i bestimmt, in dem sich der akkumulierte Phasenwert $x[n]$ befindet.

Bei $2^{w_{in}}$ Stützstellen kann man die Adresse unter der Annahme, dass das Argument $x[n]$ im Fractional-Format vorliegt folgendermaßen berechnen:

$$i[n] = Q_{w_{in}}(2^{w_{in}-1}x[n]) + i_0 \quad (5.2)$$

$x[n]$ wird hierbei auf w_{in} Stellen abgeschnitten. i_0 ist der Adressoffset und $Q_{w_{in}}$ die Quantisierungsfunktion, die das Argument auf w_{in} Stellen vor dem Komma abschneidet.

Beispiel zur Adresszuordnung:

$$w_{in} = 4; x[n] = 0.0010110, i_0 = 00001000 \\ i[n] = 0001. + 00001000 = 00001001 (\text{dezimal} : i[n] = 1 + 8 = 9)$$

Taktquelle: Die Taktquelle bestimmt die Abtastfrequenz des DDS-Systems. Die höchst- te hier erzeugbare Signalfrequenz¹¹ ist $\frac{f_s}{2}$.

Phaseninkrement δx : Das Phaseninkrement bestimmt Phase und Frequenz des Ausgangssignals. Aufgrund der endlichen Wortbreite (x_Q) (LSB) können auch nur endliche Phasen-/ Frequenzauflösungen erreicht werden.

$$\frac{\delta x}{N\delta x_i} = \frac{T_s}{T_0} \implies T_{0Q} = \frac{\delta x_i}{\delta x_Q} NT_s \implies f_{0Q} = \frac{\delta x_Q}{\delta x_i} \cdot \frac{1}{NT_s} \quad (5.3)$$

¹¹vgl. Abtasttheorem

N beschreibt hier die Anzahl der gespeicherten Abtastwerte pro Periode.

Phasenakkumulator: Der Phasenakkumulator ist nichts weiter als ein Addierer, der in jedem Taktzyklus einen vorgegebenen Wert Δx aufaddiert. Die aktuelle Phasenlage berechnet sich zu

$$x[n] = x[n - 1] + \Delta x \quad (5.4)$$

Der Wert Null steht hier für eine Phasenlage von 0° , sein Maximalwert für 360° . Somit liegt mit jedem Taktzyklus eine neue Phasenlage vor, die dann nur noch in einen Amplitudenwert umgerechnet werden muss.



Die Akkumulatorwortbreite muss stets größer oder gleich der Phaseninkrement-Wortbreite sein! Ist das nicht der Fall kommt es mit jeder Addition zu ungewollten Überläufen.

Signaltabelle – Look-Up-Table (LUT): Die Look-Up-Table ist ein Speicher, der eine endliche Anzahl Funktionswerte des zu generierenden, (oft) periodischen Signals in quantisierter Form enthält (Wortbreite w_{out}). Wegen der endlichen Wortbreite kommt es zu Quantisierungsfehlern. Ist die Phasenakkumulatorwortbreite größer als die Adressbreite der Signaltabelle, muss das Argument (die Phase) $x[n]$ gerundet werden. Als allgemeine Faustregel kann man sich merken, dass w_{out} 2 Bit breiter als w_{in} sein muss. Fehler treten auch auf, wenn zwischen den Tabellenwerten interpoliert wird, um die Ausgangswerte zu berechnen (Vorteil: Speicherplatzreduktion).

Nachfolgend ein kurzes Beispiel zur erweiterten DDS:

Es soll mittels DDS ein Sinussignal ($y = \sin(\pi x)$ für $x \in [-1, 1]$) erzeugt werden. Die verwendete LUT ist in Tabelle 5.3 dargestellt.

5.3.2.2 CORDIC

Wie sie im vorhergehenden Abschnitt erkennen konnten liefert die DDS zwar eine speicherplatzeffiziente, aber aufgrund der begrenzten Anzahl von Tabellenwerten relativ ungenaue Lösung. Daher wollen wir uns im Folgenden mit einer eleganteren und vielseitigeren Methode beschäftigen: dem **CORDIC-Algorithmus**¹². Mit ihm lassen sich viele Berechnungen lösen:

- Im CORDIC-Basisverfahren ist
 - die Berechnung der Drehung eines Vektors in einem kartesischen Koordinatensystem und die
 - Berechnung von Betrag und Phase eines Vektors möglich.
- Im erweiterten CORDIC-Basisverfahren sind
 - Multiplikation,
 - Division und

¹²CORDIC steht für “Coordinate Rotation Digital Computer”

x	y
-1.0000	0
-0.8750	-0.38269042968750
-0.7500	-0.70709228515625
-0.6250	-0.92388916015625
-0.5000	-1.00000000000000
-0.3750	-0.92388916015625
-0.2500	-0.70709228515625
-0.1250	-0.38269042968750
0	0
0.1250	0.38269042968750
0.2500	0.70709228515625
0.3750	0.92388916015625
0.5000	1.00000000000000
0.6250	0.92388916015625
0.7500	0.70709228515625
0.8750	0.38269042968750

Tabelle 5.3: DDS Look-Up-Table

– Berechnung hyperbolischer Funktionen durchführbar.

Um eine Sinusschwingung zu erzeugen genügt es, sich mit dem CORDIC-Basisverfahren im Rotation-Mode auseinanderzusetzen. Für ergänzende Informationen zu den anderen Verfahren sei auf [5] verwiesen.

Das CORDIC-Basisverfahren im Rotation Mode: Die Drehung eines Vektors $[x_0, y_0]^T$ um den Winkel θ in einem kartesischen Koordinatensystem führt auf den Vektor $[x_n, y_n]^T$:

$$\begin{bmatrix} x_n \\ y_n \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \end{bmatrix} \quad (5.5)$$

In Abbildung 5.14 wird dies graphisch dargestellt. Unter Verwendung der Identität (aus Formel (5.6))

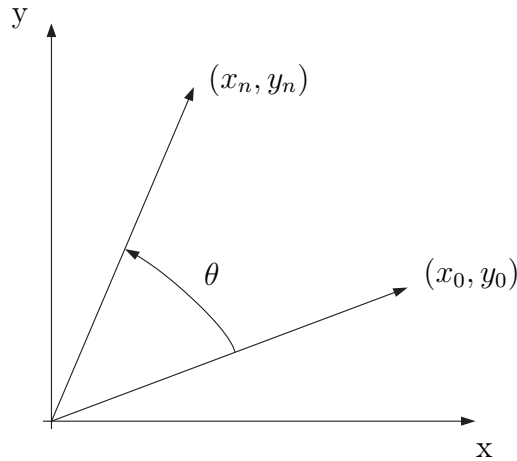
$$\cos(\theta) = \frac{1}{\sqrt{1 + \tan^2(\theta)}} \text{ und } \tan(\theta) = \frac{\sin(\theta)}{\cos(\theta)} \quad (5.6)$$

erhält man die Beziehung:

$$\begin{bmatrix} x_n \\ y_n \end{bmatrix} = \frac{1}{\sqrt{1 + \tan^2(\theta)}} \begin{bmatrix} 1 & -\tan(\theta) \\ \tan(\theta) & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \end{bmatrix} \quad (5.7)$$

Beim CORDIC-Verfahren wird die Drehung um θ durch mehrere Teilrotationen mit bekanntem Teilwinkel α_i realisiert. Durch das festzulegende Vorzeichen σ_i nähert man die Summe der Teilwinkel α_i dem Winkel σ an:

$$\theta \approx \sum_{i=0}^{n-1} \sigma_i \alpha_i \text{ mit } \sigma_i \in \{-1, 1\} \quad (5.8)$$

Abbildung 5.14: Drehung eines Vektors um den Winkel θ

Dies führt zu der Beziehung

$$\begin{bmatrix} x_{i+1} \\ y_{i+1} \end{bmatrix} = \frac{1}{\sqrt{1 + \tan^2(\alpha_i)}} \begin{bmatrix} 1 & -\sigma_i \tan(\alpha_i) \\ \sigma_i \tan(\alpha_i) & 1 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} \quad (5.9)$$

für $i = 0, 1, \dots, n-1$.

Dabei wird α_i mit zunehmendem i abnehmen. Zur Vereinfachung des Matrixprodukts und zur vereinfachten Umsetzung (in Hard- und Software) werden die α_i so gewählt, dass

$$\tan(\alpha_i) = 2^{-i} \quad i = 0, 1, \dots, n-1 \quad (\alpha_i = \arctan(2^{-i})) \quad (5.10)$$

Damit werden die Multiplikationen mit $\tan(\theta)$ zu Schiebeoperationen vereinfacht. Diese lassen sich in digitaler Hardware ohne großen Aufwand realisieren.

Somit kann man schreiben:

$$\begin{bmatrix} x_{i+1} \\ y_{i+1} \end{bmatrix} = k_i \begin{bmatrix} 1 & -\sigma_i 2^{-i} \\ \sigma_i 2^{-i} & 1 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} \quad \text{mit } k_i = \frac{1}{\sqrt{1 + 2^{-2i}}} \quad (5.11)$$

Die α_i werden positiv bewertet, bis die Summe den Wert von θ überschreitet. Danach werden sie negativ bewertet, bis die Summe den Wert von θ wieder unterschreitet. Dies wird entsprechend fortgesetzt. Das Vorzeichen der Differenz

$$z_{i+1} = \theta - \sum_{k=0}^i \sigma_k \alpha_k \quad (5.12)$$

steuert somit das Vorzeichen σ_i der Teilwinkel. Die Hilfsvariable z_i soll während des Iterationsprozesses gegen Null konvergieren (vergleiche Abb. 5.15).

Die Werte für $\alpha_i = \arctan(2^{-i})$ werden üblicherweise in Tabellenform abgelegt und werden schon im Vorfeld je nach gewünschter Genauigkeit berechnet. Ein Beispiel ist in Tabelle 5.4 angegeben.

Fasst man die Teilfaktoren k_i für alle Iterationen zusammen, so benötigt man nur noch einen einzigen, abschließenden Skalierungsschritt, anstatt einzelner Multiplikationen

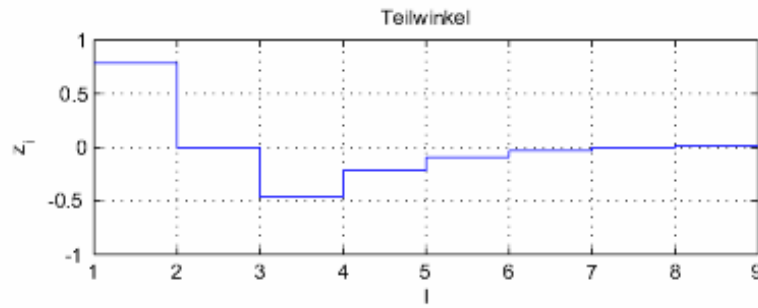


Abbildung 5.15: Schrittweite der Variable z_i

nach jedem Iterationsschritt:

$$k = \prod_{i=0}^{n-1} \frac{1}{\sqrt{1 + 2^{-2i}}} \quad (5.13)$$

Auch der gewünschte Skalierungsfaktor k kann vorab berechnet und abgelegt werden (vgl. Tabelle 5.4).

i	$\alpha_i = \arctan(2^{-i})$	α_i in $^\circ$	k_i	k
0	0.7854	45	0.7071	0.7071
1	0.4636	26.5651	0.8944	0.6325
2	0.2450	14.0362	0.9701	0.6136
3	0.1244	7.1250	0.9923	0.6076
4	0.0624	3.5763	0.9981	0.6074
5	0.0312	1.7899	0.9995	0.6073
6	0.0156	0.8952	0.9999	0.6073
7	0.0078	0.4476	1.0000	0.6073
8	0.0039	0.2238	1.0000	0.6073
9	0.0020	0.1119	1.0000	0.6073
10	0.0010	0.0560	1.0000	0.6073

Tabelle 5.4: Look-Up-Table für α_i und Skalierungsfaktor k

Mit den bisherigen Vereinbarungen gilt für die Iterationsgleichungen:

- Initialisierung:

$$z_0 = \theta$$

- Für $i = 0, 1, \dots, n - 1$:

$$\sigma_i = \begin{cases} +1 & \text{für } z_i \geq 0 \\ -1 & \text{für } z_i < 0 \end{cases} \quad (5.14)$$

$$\begin{aligned} x_{i+1} &= x_i - \sigma_i 2^{-i} y_i \\ y_{i+1} &= y_i + \sigma_i 2^{-i} x_i \\ z_{i+1} &= z_i - \sigma_i \arctan(2^{-i}) \end{aligned} \quad (5.15)$$

Zur Skalierung auf den korrekten Amplitudenwert ist nun noch eine abschließende Multiplikation durchzuführen:

$$kx_n \rightarrow x_n \quad (5.16)$$

$$ky_n \rightarrow y_n \quad (5.17)$$

Fassen wir zusammen: Wird σ_i wie oben beschrieben bestimmt, so konvergiert $z_n \rightarrow 0$ und das Differenzengleichungssystem hat die Lösung:

$$\begin{aligned} x_n &= x_0 \cos(\theta) - y_0 \sin(\theta) \\ y_n &= y_0 \cos(\theta) + x_0 \sin(\theta) \end{aligned} \quad (5.18)$$

Da dies der Drehung eines Zeigers (x_0, y_0) um den Winkel θ nach (x_n, y_n) entspricht, spricht man vom sogenannten **Rotationsmodus** des CORDIC-Algorithmus.

Durch geeignete Wahl des Startwertes $(x_0, y_0) = (x_0, 0)$ kann mit dieser Methode das Produkt eines skalaren Wertes mit dem Sinus bzw. dem Cosinus berechnet werden:

$$\begin{aligned} x_n &= x_0 \cos(\theta) \\ y_n &= y_0 \sin(\theta) \end{aligned} \quad (5.19)$$

Mit der Wahl $(x_0, y_0) = (k, 0)$ und ohne die sonst notwendige Skalierung mit k am Ende des Iterationsprozesses erhält man:

$$\begin{aligned} x_n &= \cos(\theta) \\ y_n &= \sin(\theta) \end{aligned} \quad (5.20)$$

Der CORDIC-Basisalgorithmus eignet sich also im Rotationsmodus

- zur Drehung eines Zeigers um einen vorgegebenen Winkel und
- zur Berechnung der Winkelfunktionen \sin und \cos .

5.3.3 MATLAB: Programmierung

Aufgabe 1: DDS

1. Machen sie sich mit dem MATLAB-Skript zur Direkten Digitalen Synthese (*dds.m*) vertraut¹³.
2. Stellen sie verschiedene Schwingungsfrequenzen f_0 dar und drucken sie die gelieferten Plots für ihre Dokumentation aus.

Aufgabe 2: CORDIC

1. Machen sie sich mit der Funktion *CORDIC.m*¹⁴ vertraut und ergänzen sie die fehlenden Abschnitte.
2. Schreiben sie ein MATLAB-Skript (*CORDICsim.m*), mit dessen Hilfe sie ihre Funktion mit unterschiedlichen Parametern (Iterationsschritte, Drehwinkel, Startwerte) testen können. Was fällt ihnen auf, wenn sie um Winkel größer 100° drehen lassen wollen? Wodurch wird dieser Effekt ausgelöst?

3. Stellen sie die Größen σ_i , α_i (x_0, y_0) und (x_n, y_n) mit Hilfe der MATLAB Plot-Funktionen anschaulich dar und drucken sie ihre Ergebnisse aus.
4. Ergänzen sie die vorliegende Funktion dahingehend, dass auch Winkel $>90^\circ$ richtig verarbeitet werden können.
5. Was fällt ihnen hinsichtlich Rechenaufwand und Genauigkeit auf, wenn sie die beiden Algorithmen (DDS \leftrightarrow CORDIC) miteinander vergleichen?

Aufgabe 3: Sinus-Generator

1. Kopieren sie nun ihre Funktion *cordic.m* in das Verzeichnis

MATLAB_src\sender\03_Sinusgenerator\Oszillator

¹³

MATLAB_src\sender\Sinusgenerator\DDS

¹⁴

MATLAB_src\sender\Sinusgenerator\CORDIC

Hier finden sie außerdem noch das Grundgerüst der Funktion *oszillator.m*. Ergänzen sie diese, um mit Hilfe des CORDIC-Algorithmus eine Sinus- oder Cosinusschwingung zu erzeugen.

5.3.4 Einschub: Maschinenzahlen

Bisher sind wir von einer exakten Darstellung der Abtastwerte (und Filterkoeffizienten) ausgegangen. Reale Anwendungen dagegen besitzen aufgrund der endlichen Wortlänge eine viel geringere Genauigkeit. Durch die Quantisierung der Zahlen ergibt sich eine Vielzahl von möglichen Fehlerquellen. So kommt es zum Beispiel

- zu Quantisierungsfehlern bei der AD-Wandlung,
- zu Quantisierungsfehlern bei der Darstellung von Filterkoeffizienten wie
 - verletzte Entwurfsspezifikationen oder
 - instabile Filter
- zu Arithmetikfehlern innerhalb des Filters, die sich durch
 - mögliche Wortlängenverkürzungen innerhalb des Filters oder
 - mögliche Überläufe nach einer Addition bemerkbar machen.

Um die angesprochenen Fehler verstehen zu können ist es nötig, genauer auf die Darstellung digitaler Zahlen einzugehen.

5.3.4.1 Zahlendarstellung auf Digitalrechnern

Eine digitale Verarbeitung von Zahlen, Abtastwerten u.a. bedingt die Verwendung eines für einen Rechner verständlichen Zahlensystems. Dieses basiert auf einem dualen System mit der Basis zwei - man spricht auch von Maschinenzahlen. Diese können - je nach Einsatzbereich und gewünschter Genauigkeit - in unterschiedlichen Formaten dargestellt werden.

Für hochgenaue Anwendungen eignet sich das Gleit- oder Fließkomma-Format¹⁵. Es bietet eine sehr genaue Darstellung von Zahlen, kombiniert mit einem großen Wertebereich. Da Fließkomma-Prozessoren allerdings wesentlich aufwendiger (und dadurch auch teurer) zu realisieren sind, verwendet man für Anwendung, bei denen es auf niedrige Entwicklungs- und Fertigungskosten ankommt, die ungenauere, aber technisch leicht realisierbare Darstellung im Festkommaformat¹⁶.

5.3.4.2 Darstellung im Zweierkomplement

In Rechenwerken werden duale Zahlen üblicherweise im Zweierkomplement dargestellt, da dieses eine einfache Realisierung von arithmetischen Operationen vorzeichenbehafteter Zahlen erlaubt:

$$x = a_{B-1} \dots a_1 a_0 \bullet \quad (5.21)$$

$$= 2^{B-1} (-a_{B-1} 2^0 + \sum_{i=1}^{B-1} a_{B-1-i} 2^{-i}) \quad (5.22)$$

$$\text{mit } a_i \in 0, 1 \quad (5.23)$$

¹⁵auch Floating-Point-Format

¹⁶Fixed-Point-Format

Es gibt genau 2^B verschiedene Maschinenzahlen, wobei auch die Null eindeutig ausgedrückt ist. Das Bit a_{B-1} dient als Vorzeichenbit ($0 \rightarrow$ positiv, $1 \rightarrow$ negativ). a_0 ist das Bit mit der geringsten Wertigkeit¹⁷. Der fette Punkt hinter dem LSB steht für "point"(Komma). Ist die Wortlänge zum Beispiel 16, so spricht man von Zahlen im Format 16.0, womit man ausdrücken will, dass vor dem Punkt 16 und hinter dem Punkt 0 Bits stehen.

Beispiele:

$$\begin{aligned} 5 &= 0101 \\ 7 &= 0111 \\ -7 &= 1001 \end{aligned}$$

Negative Zahlen berechnen sich im Zweierkomplement aus der positiven Zahl durch Komplementbildung und Addition von Eins.

Darstellung im Fractional-Format: Eine weitere gängige Darstellung ist die Beschreibung von dualen Zahlen im **Fractional-Format**. Darunter versteht man eine Zahl zwischen -1 und 1. Fractional-Zahlen werden folgendermaßen dargestellt:

$$x = a_{B-1} \bullet a_{B-2} \dots a_0 \quad (5.24)$$

$$= -a_{B-1}2^0 + \sum_{i=1}^{B-1} a_{B-1-i}2^{-i} \quad (5.25)$$

mit $a_i \in \{0, 1\}$ und $-1 \leq x \leq 1 - LSB$

Eine Unterscheidungsmöglichkeit zur Darstellung einer ganzen Zahl bietet lediglich die Stellung des Punktes, der sich bei Fractional-Zahlen direkt hinter dem Vorzeichenbit befindet. So ist zum Beispiel das Fractional-Format für eine 16 Bit breite Dualzahl 1.15.

Rechnerintern werden ganze und gebrochene Zahlen gleich dargestellt. Die Stellung des Punktes wird allein durch die Vereinbarung festgelegt.

Üblich ist das Rechnen mit Fractional-Zahlen, weil i.a. angenommen wird, dass alle Abtastwerte im Intervall $[-1..1[$ liegen. Werden die Abtastwerte mit Koeffizienten multipliziert, die betragsmäßig größer als 1 sind, so müssen die Koeffizienten vorher entsprechend skaliert werden.

Beispiel(mit B=4):

$$0.625_d = 0.101_b$$

Die Negation von Fractional-Zahlen geschieht wie bei ganzen Zahlen auch durch Komplementbildung und Addition eines LSB und ergibt sich beim Fractional-Format zu:

$$-x = -\bar{a}_{B-1}2^0 + \sum_{i=1}^{B-1} \bar{a}_{B-1-i}2^{-i} + 2^{-(B-1)} \quad (5.26)$$

mit $a_i \in \{0, 1\}$ und $-1 \leq x \leq 1 - LSB$.

Ausgehend von der einfachen Negation lässt sich eine Subtraktion realisieren, indem man zuerst das Zweierkomplement des Subtrahenten bildet und danach die beiden Zahlen einfach addiert.

¹⁷LSB \rightarrow least significant bit

Darstellbare Zahlen im Zweierkomplement: Prinzipiell ist im Zweierkomplement die Darstellung der Zahl -1 möglich, von +1 dagegen nicht. Verlässt man (z.B. bei einer Addition) den darstellbaren Zahlenbereich (man spricht von einem Überlauf), so folgt der betragsmäßig größten positiven Zahl die kleinste negative Zahl und umgekehrt. Die Quantisierungskennlinie einer 3 Bit Zweierkomplementzahl ist in Abbildung 5.16 dargestellt.

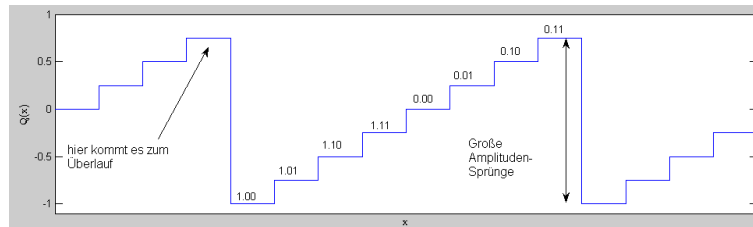


Abbildung 5.16: Quantisierungskennlinie mit Zweierkomplementüberlauf

Wie in der Abbildung sehr gut zu erkennen ist treten im Falle von Überläufen sehr hohe Amplitudensprünge auf, die zu einem stark nichtlinearem Verhalten führen. Allerdings ist das Modulo-2 Überlaufverhalten inhärent und hat den Vorteil, dass Teilüberläufe keine Auswirkungen haben, solange das Gesamtergebniss im Bereich von $[-1,1[$ liegt! Ein Beispiel hierzu finden sie in Tabelle 5.5.

Dezimal	Binär	
0.750	0.110	
+0.500	+0.100	
1.250	1.010	(entspricht -0.75 → Überlauf)
-0.625	+1.011	
0.625	10.101	(entspricht 0.625)

Tabelle 5.5: Beispiel zur Auswirkung von Teilüberläufen

Vor allem bei rekursiven Systemen¹⁸ sollte man eine Modulo-2-Überlaufkennlinie vermeiden, da durch die großen Amplitudensprünge große Grenzzyklen entstehen können (vgl. Kap.6.2.4.2). Zur Realisierung ist eine Überlaufbehandlung in der Arithmetikeinheit notwendig! Eine weitere Möglichkeit ist der Einsatz von Quantisierungskennlinien mit Sättigungsverhalten, wie in Abb. 5.17 gezeigt.

Alignment und Sign-Extension: Moderne DSPs besitzen eine Reihe von Registern und Speicherzellen, die unterschiedliche Wortlängen aufweisen. Beim Transfer von Zweierkomplementzahlen zwischen solchen Speicherplätzen ist zu berücksichtigen, wie ein Datenwort kleiner Wortlänge in einem Speicherplatz großer Wortlänge abgelegt wird.

- Beim Left-Alignment (“Linksbündiges” Einfügen in das Register) werden die niederwertigen Bits mit Nullen aufgefüllt,

¹⁸Systeme mit Rückkopplung, wie IIR-Filter (vgl. Kap.6.2)

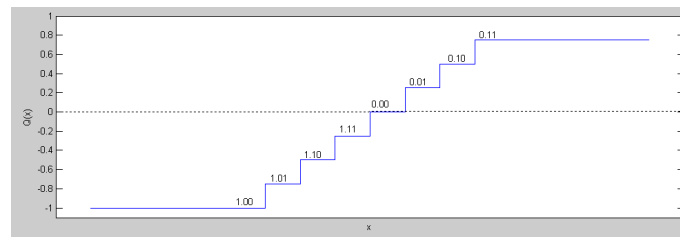


Abbildung 5.17: Quantisierungskennlinie mit Sättigungsverhalten

- Beim Right-Alignment (“Rechtsbündiges” Einfügen in das Register) ist eine Sign-Extension notwendig. Bei negativen Zahlen werden daher die höherwertigen Bits mit Einsen aufgefüllt, bei positiven Zahlen mit Nullen.

Bei der Multiplikation von Zweierkomplementzahlen sind ebenfalls Besonderheiten zu beachten. Hier unterscheidet sich die Wortlänge des Produkts von jener der beiden Operanden. Wird eine Zahl mit N_1 Bits mit einer Zahl mit N_2 Bits multipliziert, so hat das Produkt $N_1 + N_2 - 1$ Bits.

$$N_1 \cdot N_2 \Leftrightarrow N_1 + N_2 - 1 \quad (5.27)$$

Beispiel

Betrachten wird das Produkt zweier 32-Bit-Zahlen. Das Ergebnis hat nach obiger Formel 63 Bit. Je nachdem, ob das Ergebnis Left- oder Right-Aligned in einem 64-Bit-Register abgespeichert wird erhält man ein zusätzliches (redundantes) LSB oder MSB.

Aus diesem Grund wird bei einigen DSPs zwischen der Multiplikation von Integer-Zahlen (32.0) und Fractional-Zahlen (1.31) unterschieden. Bei Fractional-Zahlen muss das Ergebnis left-aligned, bei Integer-Zahlen right-aligned im Register gespeichert werden.

Allgemein gilt, dass die Multiplikation einer Zahl im $I_1.Q_1$ -Format mit einer Zahl im $I_2.Q_2$ -Format ein Ergebnis im $(I_1 + I_2 - 1).(Q_1 + Q_2)$ -Format ergibt.

$$(I_1.Q_1) \cdot (I_2.Q_2) \Leftrightarrow (I_1 + I_2 - 1).(Q_1 + Q_2) \quad (5.28)$$

5.3.4.3 Zahlendarstellung im Gleitkommaformat

Die Darstellung im Gleitkomma- oder Floating-Point-Format ist eine exponentielle Darstellung der Zahlen. Diese ist unerlässlich zur rechnerinternen Verarbeitung sehr großer Zahlen, ebenso bei Anwendungen, die eine extrem hohe Genauigkeit erfordern. Im Gegensatz zur wissenschaftlichen Zahlennotation, die ja mit der Basis Zehn arbeitet, verwendet das Gleitkommaformat Zweierpotenzen.

Eine Gleitkommazahl ist ganz allgemein aus drei Teilen aufgebaut:

- Einem Vorzeichenbit (S)
- der Mantisse (M) und
- dem Exponenten (E).

$$x = (-1)^S \cdot 2^E \cdot M \quad (5.29)$$

Leider gibt es im Gegensatz zu Festkommasignalprozessoren, die einheitlich die Zweierkomplementdarstellung verwenden, noch kein einheitliches Zahlenformat bei Gleitkomma-Signalprozessoren. Zwar verwendet die Mehrheit das genormte 32-Bit-IEEE-Format, aber es gibt immer noch einige Hersteller, die ihre eigenen Standards bevorzugen.

IEEE-Standard P754: Dieser Standard wird - wie oben schon erwähnt - in vielen Signalprozessoren verwendet. Man unterscheidet hier nochmals zwischen zwei unterschiedlichen Genauigkeitsstufen: Single-Precision und Double-Precision.

- Single-Precision
 - kleinstes Format mit 32 Bit
 - Die Zahl wird mit 1 Vorzeichenbit, 8 Exponentenbits und 23 Bits für die Mantisse dargestellt
- Double-Precision
 - 64 Bit Genauigkeit
 - Die Zahl wird mit 1 Vorzeichenbit, 11 Exponentenbits und 52 Bits für die Mantisse dargestellt

Die Darstellung ist in der Regel so skaliert, dass die höchstwertige Stelle der Mantisse 1 ist und daher weggelassen werden kann. In diesem Fall spricht man von der *normalisierten Form*.

Bei der Berechnung des Exponenten wird dessen Vorzeichen durch einen Offset (OS) berücksichtigt. Dieser beträgt bei Single-Precision 127, bei Double-Precision 1023.

$$x = (-1)^S \cdot 2^{E-OS} \cdot (1 + M_f) \quad (5.30)$$

Vorteile des Gleitkommaformates: Mit dem Floating-Point-Format kann ein sehr großer Zahlenbereich abgedeckt werden (Im Single-Precision-FP-Format kann beispielsweise ein Zahlenbereich von $-1.7 \cdot 10^{38} \dots + 1.7 \cdot 10^{38}$ dargestellt werden. Außerdem lassen sich betragsmäßig kleine Zahlen sehr viel präziser darstellen als Festkomma-Zahlen (Die betragsmäßig kleinste 32-Bit-Fractional-Zahl ist $4.7 \cdot 10^{-10}$, beim single-precision Fractional-Format $1.5 \cdot 10^{-39}$).

Man kann aus den Zahlenbeispielen deutlich erkennen, dass 32-Bit-Gleitkomma-Prozessoren einen beträchtlich größeren Dynamikbereich besitzen, als die üblichen 16- oder 24-Bit-Festkomma-Prozessoren. Daher kann man sie als nahezu überlauffrei betrachten. Ihr Nachteil liegt dafür im deutlich komplizierteren Rechenwerk, was sich in der größeren Chipfläche, einem höheren Stromverbrauch und im höheren Preis des Prozessors niederschlägt.

Beurteilungskriterien der Eigenschaften eines Zahlenformats: Im letzten Abschnitt wurden die beiden Kriterien Dynamik und Präzision herangezogen, um die Zahlenformate zu bewerten. Diese sollen nun noch einmal kurz erläutert werden:

- Die Dynamik beschreibt das Verhältnis von größter und kleinster darstellbarer positiver Zahl,
- die Präzision den maximalen Quantisierungsfehler beim Runden.

5.3.5 VHDL: Realisierung

Im Folgenden soll der CORDIC-Algorithmus in VHDL implementiert werden.

5.3.5.1 VHDL-Basics: Fixed-Point Arithmetik

Für den CORDIC ist es notwendig, eine Look-Up-Table zu erzeugen, aus der der Algorithmus die vorberechneten Werte extrahieren kann. Problematisch für eine Hardwarerealisierung des CORDIC ist die Verwendung von rationalen bzw. reellen Zahlen. Eine Möglichkeit besteht darin, die `math_real`-Library zu verwenden. Diese erfordert jedoch eine sehr komplexe Hardware und ist nur in Fällen anzuraten, wenn kein Weg daran vorbei führt oder die Größe und der Leistungsverbrauch eine sehr untergeordnete Rolle spielen. Die Entwicklungsumgebung von `ispLEVER` unterstützt diese Library lediglich zur Syntheszeit. Das bedeutet, dass Berechnungen damit durchgeführt werden können, deren Ergebnis aber nach der Synthese feststehen muss. Bedenkt man den immensen Aufwand für eine Berechnung in Floating-Point Arithmetik, wird schnell klar, dass die Ressourcen der verwendeten Hardware unzureichend sind, wenn mehr als nur den CORDIC implementiert werden soll. Daher wird im Folgenden Fixed-Point-Arithmetik verwendet. Besonders das Problem der Quantisierung und der damit verbundenen Effekte muss genauer betrachtet werden.

Eine Fixed-Point Zahl wird durch eine Integerzahl ausgedrückt, indem man deren Wertebereich umdefiniert. Dies ist nötig, da in VHDL lediglich Integer-Zahlen zur Verfügung stehen. Im Praktikum wird vorerst hauptsächlich mit 8- und 16-Bit-Zahlen gearbeitet, deren Wertebereich sich von -128 bis 128 bzw. von -32768 bis 32767 erstreckt. Dieser Zahlenbereich ist besonders interessant, da das FPGA Multiplizierer mit den Wortbreiten 9, 18 und 36 zur Verfügung stellt. 8 bzw 16-Bit sind optimal, da z.B. nach einer Addition die Wortbreite des Ergebnisses größer ist als die der Eingangswerte und somit eine Reserve vorgehalten werden muss. Dies wurde vom Hersteller dadurch bedacht, dass er die Multiplizierer ein Bit größer macht als nötig und dem Designer damit Spielraum bei der Implementierung gibt.

Der Bereich von -32768 bis 32767 wird umdefiniert zu -1 bis exclusive 1. Damit lässt sich eine Genauigkeit von ca. $3,05 \cdot 10^{-5}$ erreichen, was ausreichend ist. In MANchen Versuchen genügt sogar die Genauigkeit von 8 Bit.

Der genaue Wert der Zahl ergibt sich somit wie folgt.

$$Z = -N_{15} + N_{14} \cdot 2^{-1} + N_{13} \cdot 2^{-2} + \dots + N_{-14} \cdot 2^{-0} \quad (5.31)$$

Mit Hilfe dieser Definition ist es möglich, Addition, Subtraktion und Multiplikation wie gewohnt im Zweierkomplement auszuführen.

5.3.5.2 VHDL-Basics: Konstanten

Da sowohl für die DSS als auch für den CORDIC Look-Up-Tables benötigt werden, stellt sich die Frage, wie die vorausberechneten Werte am effektivsten in das Programm einzubinden sind. Hier bietet VHDL die Möglichkeit, Konstanten zu definieren.

```
...  
    constant sv_lut0 : std_logic_vector(15 downto 0) := "0101010101010101";  
...
```

Um Berechnungen vor der Laufzeit des Codes, also durch die Synthese durchführen zu lassen, genügt es, den gewünschten Ausdruck hinter die Definition zu schreiben.

```
...  
    constant sv_lut1 : std_logic_vector(15 downto 0)  
        := "0101010101010101" + "00000000000000101";
```

Die Synthese wird diesen Ausdruck berechnen und das Ergebnis direkt der Konstanten zuweisen.

5.3.5.3 VHDL-Basics: Typenkonvertierung

Eine elegantere Variante ist die Verwendung von Konvertierungsfunktionen.

conv_std_logic_vector(<VALUE>, <WIDTH>) Konvertiert eine Integerzahl *VALUE* in den Typ `std_logic_vector` mit der angegebenen Breite *WIDTH*.

conv_unsigned(<VALUE>, <WIDTH>) konvertiert eine Integerzahl in eine unsigned Darstellung des Typs `std_logic_vector`.

conv_signed(<VALUE>, <WIDTH>) Konvertiert in eine Signed-Darstellung des `std_logic_vector`-Typs.

Bei `conv_std_logic_vector` ist darauf zu achten, dass die Interpretation eingebundenen Paket (signed oder unsigned) abhängt.

Weitere Konvertierungsroutinen werden im Laufe des Praktikums folgen.

5.3.5.4 VHDL-Basics: Mehrdimensionale Arrays

Bisher wurde lediglich die Festlegung der einzelnen Zeilen behandelt, allerdings ist dies für eine Tabelle recht ineffizient. Hierzu verwendet man ein Array des Typs `std_logic_vector`, um jede Zeile durch einen Index ansprechen zu können, wie es beispielsweise mit den einzelnen Bits des `std_logic_vector` möglich ist. In VHDL ist es erlaubt, mehrdimensionale Arrays beliebiger Tiefe zu erzeugen. Derzeit erlaubt die Synthese aber maximal zweidimensionale Arrays.



Arrays maximal zweidimensionale in Hardware implementieren.
`Std_logic_vector` ist selbst ein eindimensionales Array des Typs `std_logic`, daher maximal eine weitere Dimension zulässig

Eine Möglichkeit eine LUT zu erzeugen sähe also wie folgt aus:

```
...
type tva_LUT is array(0 to 1) of std_logic_vector(3 downto 0);

constant my_lut : tva_LUT := (("0100"),("1000"));
...
```

Der Aufbau der benötigten LUT erfolgt analog zum obigen Beispiel.

5.3.5.5 VHDL-Basics: Zahlentypen

Eine Darstellung von Zahlen erfolgt in Hardware üblicherweise mittels mehrerer einzelner Signale in binärer Form z.B. mittels des Zweierkomplements. Programmieren wäre jedoch mit dieser Methode recht umständlich und die Übersetzung könnte auch gut die Synthese erledigen. Dazu werden an dieser Stelle einige neue Datentypen vorgestellt:

integer übersetzt in 32-bit, Werte zwischen -2^{31} und $2^{31} - 1$

natural übersetzt in 31-bit, Werte zwischen 0 und $2^{31} - 1$ (Vorsicht!!! nicht 32-bit)

positive übersetzt in 31-bit, Werte zwischen 1 und $2^{31} - 1$ (Vorsicht!!! auch nur 31-bit)

Jeden der oben genannten Zahlentypen kann man mit *range* *<FROM>* *to* *<TO>* eingrenzen und so die Synthese anweisen, die minimale Anzahl an Signalen zu verwenden. Bei der Deklaration würde dies wie folgt aussehen:

```
...
signal si_counter : integer range 0 to 40;
...
```

Dies würde die Synthese veranlassen, intern einen 6-Bit-Vektor (0 bis $2^6 - 1 = 63$) zu verwenden.

5.3.5.6 VHDL-Basics: Records

Ähnlich wie in Software-Programmiersprachen können auch in VHDL verschiedene Signale zu einem Record gebündelt werden, um auf diese als Ganzes zugreifen zu können. Ein Record wird über einen eigenen Typ definiert.

```
...
type <REC_NAME> is
  record
    DAY: positive range 1 to 31;
    MONTH : positive range 1 to 12;
    YEAR : integer;
  end record;
...
```

5.3.5.7 VHDL-Basics: Komponenten

Bisher wurde Ihnen die Erstellung eigener Module abgenommen. Bei größeren Modulen empfiehlt es sich aus Gründen der Übersichtlichkeit, dieses in mehrere Untermodule aufzugliedern. VHDL ermöglicht dies, indem man Komponenten erst deklariert und anschließend instantiiert. Die Instantiierung ist nötig, da man dadurch ein Modul mit unterschiedlichen Parametern (Generics, Port-Mapping) mehrmals in einem File verwenden kann, ähnlich den Klassen einer objektorientierten Programmiersprache. Eine Komponente deklariert man wie folgt:

```
...
architecture ...
...
  component <COMP_NAME>
    [generic(
      <GEN_NAME> : <TYPE> [:= <VALUE>] [;
      <GEN_NAME> : <TYPE> [:= <VALUE>]; ...]
    );]
    port(
      <PORT_NAME> : <DIR> <TYPE> [;
      <PORT_NAME> : <DIR> <TYPE>; ...]
    );
  end component;
...
begin
...

```



Es ist ratsam in eigenen Modulen, für Ports lediglich `std_logic` und `std_logic_vector` zu verwenden, da dies der Hardware am nächsten kommt und bei der Synthese sonst oftmals Schwierigkeiten auftreten, über Modulgrenzen hinweg zu optimieren. Für Generics hingegen dürfen nur Integer-Typen (Integer, Boolean, Natural und Positive) verwendet werden.

Hat man die Komponente in seinem Code deklariert, erfolgt die Instantiierung:

```
...
architecture ...
...
begin
...
  <INSTANT_NAME>: <COMP_NAME>
    [generic map (
      <GEN_NAME> => <CONST_NAME> [;
      <GEN_NAME> => <CONST_NAME>; ...]
    );]

```

```

port map (
  <PORT_NAME> => <SIGNAL_NAME> [;
  <PORT_NAME> => <SIGNAL_NAME>; ...]
);
...

```

5.3.5.8 VHDL-Basics: Multiplizierer

In dem verwendeten FPGA sind fertige Multiplizierer bereits integriert. Diese können auf unterschiedliche Art und Weise verwendet werden. Einerseits bietet ispLEVER mit dem Programmmodul IPexpress, mit dem man Module generieren und diese dann in den eigenen Code als Komponente einbinden kann. Oft sind die von IPexpress generierten Module optimiert und bieten manche Features, auf die man sonst keinen Zugriff erlangt. Allerdings kann man ebenso das Synthesetool zur Optimierung nutzen und nach dem Syntheselauf anhand der Ausgaben, die jene erzeugt, entscheiden, ob man mit dem Ergebnis zufrieden ist, oder ob man doch auf IPexpress zurückgreift. In der Ausgabe finden sich zwei Bereiche, aus denen die Verwendung der DSP-Einheiten hervorgeht. Einmal in Form einer Matrix, die die Verwendung der einzelnen Multiplizierer der DSP-Blöcke angibt und zum anderen eine Information über die Konfiguration der DSP-Blöcke (MULTADDSUM, MAC,...).

Der Einfachheit wegen wird im Rahmen des Praktikums auf IPexpress verzichtet, der Multiplizier-Operator in VHDL verwendet und die Synthese übernimmt die Arbeit.

```

...
sv_mult_result <= sv_var_a * sv_var_b; -- std_logic_vector
si_mult_result <= si_var_a * si_var_b; -- integer
...

```

Die Synthese erkennt bei dieser Beschreibung selbstständig, ob ein 9-, ein 18- oder ein 36-Bit-Multiplizierer nötig ist. Diese Arten von Multiplizierern stellt der verwendete FPGA zur Verfügung.



Da die Anzahl der kleineren 9-Bit-Multiplizierer größer ist als die der 18- bzw. 36-Bit-Multiplizierer, ist es hier besonders wichtig, den Wertebereich der Signale so klein wie möglich zu halten. Daher sollten die Integer-Typen bei der Deklaration unbedingt einen möglichst kleinen Wertebereich zugewiesen bekommen.

5.3.6 VHDL: Beschreibung CORDIC-Base


Aufgabe 1: Look-Up Table Erstellen Sie in 03_Siggen\cordic_lut.vhd eine Look-Up Table, die alle benötigten Informationen für einen 10-stufigen CORDIC-Algorithmus beinhaltet. Diese soll in Form eines Moduls aufgebaut sein und als Schnittstelle die Ports aus Tabelle 5.6 besitzen.

Die Werte der LUT kann man entweder durch festes Kodieren der Zahlen oder durch Berechnen während der Synthese mittels des Packages ieee.math_real einfügen. Wer schon

Portname	Direction	Bits	Beschreibung
STEPS	In	4	Die Anzahl der Schritte des Algorithmus
ALPHA_I	Out	16	Der Drehwinkel
TAN_ALPHA_I	Out	16	Der Tangens des Drehwinkels
K_I	Out	16	Der Skalierungsfaktor für diesen Schritt
K_G	Out	16	Der Gesamtskalierungsfaktor für diesen Schritt

Tabelle 5.6: Port des LUT-Moduls

ein wenig mehr Erfahrung in VHDL hat, kann dies gerne versuchen, er genügt jedoch, die Werte mit dem Taschenrechner zu berechnen und in den Quellcode einzutippen. Ein Beispiel, für die elegante Methode ist im VHDL-Code des DDS-Moduls zu finden.



Verwenden sie ausschließlich Integer-Typen bzw. `std_logic_vector` zur Darstellung von Zahlen. Diese können als Einzige von der Synthese in Hardware umgesetzt werden. Für die Synthese existieren keine Fixed-Point Zahlen.

Hinweis Eine LUT kann üblicherweise mittels eines Arrays aufgebaut werden. Versuchen Sie, die gestellte Aufgabe möglichst elegant zu lösen und bedenken Sie, dass die vorangegangenen Grundlagen bei der Lösung des Problems nützlich sein können.

Aufgabe 2.1: CORDIC-Basis-Algorithmus Programmieren Sie nun den CORDIC-Basis-Algorithmus, der es ermöglicht, einen beliebigen Vektor um $\pm 100^\circ$ zu drehen. Binden Sie dazu das in der letzten Aufgabe erstellte Modul `cordic_lut.vhd` ein. Der vorgefertigte Quellcode `cordic_base.vhd` mit den Ports und Generics existiert bereits in ihrem Paraktikumsverzeichnis. Die architecture ist von Ihnen zu programmieren. Die Anzahl der Stufen soll über das Generic *STEPS* von 1 bis 10 gesteuert werden können.

Hinweis: Verwenden Sie einen Zähler, der durch die einzelnen Iterationsstufen zählt und die Hardware steuert.

5.3.7 MODELSIM: Simulation CORDIC-Base

Aufgabe 2.2: CORDIC-Basis-Algorithmus Simulieren sie das fertige Modul und verifizieren sie dessen Fehlerfreiheit. Verwenden sie dazu die Testbench `cordic_base.vhd`. Drucken Sie die Waveform aus Modelsim aus und fügen sie diese ihrem Lösungsordner bei.

5.3.8 VHDL: Beschreibung 360°-CORDIC

Aufgabe 3.1: 360°-CORDIC-Algorithmus Entwerfen Sie ein Modul, dass den CORDIC aus Aufgabe 2 einbindet und auf Drehwinkel von 360° erweitert. Verwenden Sie das File `cordic_full.vhd`, binden sie `cordic_base.vhd` ein und erweitern es um die geforderte Funktionalität.

5.3.9 MODELSIM: Simulation 360°-CORDIC

Aufgabe 3.2: 360°-CORDIC-Algorithmus Simulieren Sie den Code ebenfalls und drucken Sie auch diese Waveform aus.

5.3.10 VHDL: Tonerzeugung

Aufgabe 4.1: Der erste eigene Ton Vervollständigen Sie das Modul `siggen.vhd` so, dass ein Ton mit ca. 9 kHz erzeugt wird. Verwenden Sie den berechneten Vektor als Eingabevektor für den nächsten CORDIC-Durchlauf.

Wie lautet die Formel zur Berechnung der Ausgangsfrequenz des CORDIC-Signalgenerators?

5.3.11 MODELSIM: Tonerzeugung

Aufgabe 4.2: Der erste eigene Ton Simulieren Sie `tone_gen.vhd` mit der vorgefertigten Testbench `siggen_tb.vhd` über einen ausreichend langen Zeitraum (ca. 1000 Perioden). Was stellen Sie fest?

Worauf ist dieses Verhalten zurückzuführen?

Aufgabe 5.1: Verbesserung des CORDIC-Signalgenerators Wie ist `tone_gen.vhd` zu verändern, um diesen Fehler zu beseitigen?

Welche Möglichkeit zur Hardware-Einsparung ergibt sich durch diese Veränderung noch?

Aufgabe 5.2: Simulation der Vebesserungen Simulieren Sie den verbesserten Signalgenerator.

5.3.12 TEST: Praxis

Aufgabe 6 Testen Sie Ihren Signalgenerator mit Hilfe der Hardware.

5.4 Versuch 4: Pegelanzeige

5.4.1 Konzept

Da die Pegel am Mikrofoneingang stark schwanken, ist es wünschenswert, die Verstärkung nachzuführen, um eine maximale Aussteuerung der AD-Wandler zu erhalten. Ebenso ist es nützlich, die Pegel am Ausgang des später verwendeten Filters anzeigen zu können. Da die Funktion einer Pegelanzeige am Einfachsten mit dem Ausgang des Signalgenerators verifiziert werden kann, wird dieser Block an diese Stelle vorgezogen.

5.4.2 Realisierungsmöglichkeiten

Da Pegel leistungsbezogene Angaben sind, muss eine quadratische Mittelwertbildung über die Amplitude der Spannung durchgeführt werden. Die Leistung eines Signals ist definiert durch:

$$\bar{P} = \frac{1}{t_M} \cdot \int_t^{t+t_M} |U^2| \cdot dt = \frac{1}{t_M} \cdot \int_t^{t+t_M} U^2 \cdot dt \quad (5.32)$$

Um eine richtige Anzeige zu erhalten, muss eine Mittelung über einen relativ langen Zeitraum t_M erfolgen. Dieser sollte zwischen 50 und 100 ms betragen, um noch synchron zum Gehörten zu wirken.

5.4.2.1 Pegeldefinition

Pegel werden meist in dBm angegeben. Der die Spannung bei einem Pegel von 3dBm beträgt $\approx 0,31V_{eff}$.

Auf eine Simulation in MATLAB kann hier aufgrund der Einfachheit verzichtet werden.

5.4.3 VHDL: Realisierung

5.4.3.1 VHDL-Basics: Variablen

Bisher wurde die Möglichkeit gezeigt, Werte in Signalen zu speichern. Besitzt eine Architecture hingegen mehrere Prozesse, wird dies schnell unübersichtlich.

In Prozessen darf man sogenannte Variablen deklarieren, die lediglich lokale Gültigkeit besitzen. Im Vergleich zu Signalen ergeben sich jedoch erhebliche Unterschiede, die selbst für erfahrene Programmierer eine Fehlerquelle darstellen.

Der syntaktische Teil, die Deklaration, ist folgendermaßen aufgebaut.

```
<PROCNAME>: process (<SENSITIVITY LIST>)
  variable <VARNAME>: <TYPE>[ := <INITIALIZATION>];
  ...
```

```
begin
    ...
end process <PROCNAME>;
```

Variablen haben, wie bereits erwähnt, nur lokale Gültigkeit. Dies schlägt sich auch in einer schlechten Zugänglichkeit während der Simulation nieder. Auch ändert sich die Syntax der Wertzuweisung. Signalen wird mittels `<=` ein Wert zugewiesen, Variablen hingegen mittels `:=`. Ein Beispiel hierzu:

```
sig_1 <= sig_2;
sig_2 <= var_1;
var_1 := var_2;
var_2 := sig_2;
```

Ein grundlegender Unterschied zwischen Signalen und Variablen ist, dass Signale bis zum nächsten Ereignis ihren Wert behalten, egal in welcher Reihenfolge im Code sie stehen, Variablen hingegen sofort den zugewiesenen Wert erhalten.

Folgendes Beispiel soll dies verdeutlichen:

```
var_1 := var_1 + 5;
var_2 := var_1 + 5;

sig_1 := sig_1 + 5;
sig_2 := sig_1 + 5;
```

Hier erhält `var_2` sofort den Wert von `var_1 + 10`, während `sig_2` den Wert von `sig_1 + 5` erst beim nächsten Ereignis erhält.

Eine Variable in dieser Verwendung stellt ein erhebliches Problem für die Synthese dar und kann diese trotz der Einfachheit der Anweisung sehr komplexen Code erzeugen. Im einfachsten Fall wird Hardware doppelt erzeugt, um die Berechnung vorwegzunehmen. Auch die Geschwindigkeit kann sich erheblich reduzieren.



Reihenfolge der Zuweisung von Variablen beachten, sonst kann es zu unerwartetem Verhalten führen.

Beachtet man bestimmte Regeln, so treten wenig Probleme bei der Verwendung von Variablen auf. Die wichtigste Regel bei Variablen besteht darin, die Zuweisung der Variablen an das Ende des Prozesses zu verlegen, wenn man nicht ausdrücklich plant, das spezielle Verhalten der Variablen zu nutzen um den Code zu verschlanken.

In Software-Programmiersprachen wird davon abgeraten, globale Variablen (bei VHDL vergleichbar mit Signalen) zu verwenden. In VHDL gestaltet sich dies ein wenig anders. Hier wird unerfahrenen Programmierern geraten, nur Signale zu verwenden. Bei überschaubaren Prozessen von wenigen Zeilen ist es jedoch ratsam, auf Variablen zurückzugreifen, um den Aufwand an Signaldeklarationen im Deklarationsteil so gering wie möglich zu halten.

Ein weiterer Grund für die Verwendung von Variablen ist, dass oft temporäre Signale in vielen Prozessen gleiche Namen tragen, allerdings nicht den gleichen Wert besitzen. Einigt man sich hier auf gleiche Namen, so wird das Verständnis des Codes erleichtert, da man sich an bekannten Strukturen orientieren kann. Ein Beispiel ist die Flankenerkennung von Signalen. Hierzu benötigt man ein temporäres Signal (respektive Variable), die den Wert des Signals des letzten Taktzyklusses speichert, um im Nächsten diesen mit dem aktuellen vergleichen zu können. Hier bietet sich beispielsweise der Variablenname `vl_edge_detect` an. So später zu jeder Zeit bekannt, dass es sich hier um die Flankenerkennung handelt.

5.4.3.2 VHDL-Basics: Unterscheidung Simulation <-> Synthese

In Modelsim lassen sich keine Variablen auslesen. Es ist daher unumgänglich, für das Debugging ein Signal zu verwenden. In Hardware darf dieses nicht implementiert werden, und die Synthese ist anzuweisen, dieses zu ignorieren. In VHDL geschieht dies durch folgende Schlüsselworte:

```
--synopsis translate_off
...
--synopsis translate_on
```

Die Anweisungen an Stelle des Platzhalters werden von der Synthese ausgenommen. Somit wird es möglich, Signale und deren Zuweisungen lediglich in der Simulation zu erlauben.

Auch andere Anwendungen existieren, die auf solche Anweisungen angewiesen sind. Beispielsweise die Modelle für die Multiplizierer die in IPExpress generiert wurden. Die Synthese verwendet dann die DSP-Blöcke, die in der Hardware verfügbar sind.

5.4.4 VHDL: Beschreibung der Hardware

Die Pegelanzeige soll Pegel in Abstufungen von 3 dB anzeigen, wobei die rote LED ab einem Pegel von $+1,5\text{dBm}$ leuchten soll, die obere gelbe ab einem Pegel von $-1,5\text{dBm}$, die zweite gelbe ab $-4,5\text{dBm}$, usw.

Aufgabe 1.1: Realisieren sie in VHDL die Pegelanzeige in der Datei `04_Levelmeter\levelmeter.vhd`.

5.4.5 MODELSIM: Simulation der Beschreibung

Aufgabe 1.2: Simulieren sie die zuvor programmierte Pegelanzeige. Erstellen sie hierzu Ihre eigene Testbench mit geeigneten Stimuli. Orientieren sie sich an den bereits verwendeten Testbenches.

5.4.6 TEST: Praxis

Aufgabe 2: Verknüpfen sie nun den Eingang der Pegelanzeige mit dem von AD-Wandler kommenden Mikrofonsignal und testen sie die Anzeige in der Realität. Alternativ schließen sie den Line-In-Eingang an den PC an und spielen mit Winamp ein kurzes Musikstück ab.

5.5 Versuch 5: Modulator

5.5.1 Konzept

Die Übertragung von Daten über eine beliebige Übertragungsstrecke ist nicht ohne weiteres (fehlerfrei) möglich. Würde man die Daten 1:1 über den Kanal schicken, das heißt zum Beispiel durch Codierung der Null-Bits mit einer Spannung von 0 Volt und der Eins-Bits mit 3.3 Volt, wäre das Ergebnis ziemlich ernüchternd. Kleinere Spannungsschwankungen wirken sich in diesem Fall direkt auf das Empfangssignal aus. Um hier Fehler schon im Vorfeld auszuschließen wird das Signal vor der Übertragung moduliert, wodurch die Störanfälligkeit verringert wird.

5.5.2 Realisierungsmöglichkeiten

Um das Praktikum nicht zu kompliziert zu gestalten werden wir uns mit einer einfachen FSK-Modulation¹⁹ begnügen. Hierbei werden Nullen und Einsen durch zwei Sinusschwingungen unterschiedlicher Frequenz dargestellt. Für eine logische Null wird die eine, für eine Eins die andere Frequenz übertragen. Sie haben im Verlauf der ersten Kapitel die nötigen Baugruppen bereits kennengelernt. Nun werden sie den kompletten FSK-Sender, wie im Blockschaltbild in Abb. 5.18, aufbauen.

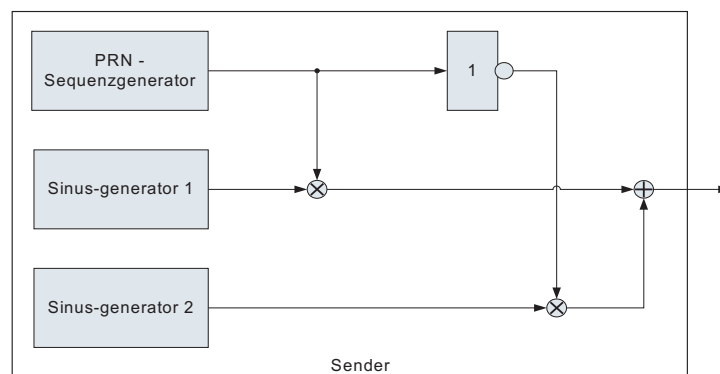


Abbildung 5.18: Sender Blockschaubild

5.5.3 MATLAB: Programmierung

Aufgabe 1: Schreiben sie ein MATLAB-Skript, das unter Verwendung der bisher programmierten Funktionen *PRNseq.m*, *cordic.m* und *oszillator.m* die oben beschriebene Funktion erfüllt. Stellen sie die Ergebnisse graphisch dar. Legen sie alle verwendeten Skripten und die erhaltenen Ergebnisse im Verzeichnis

MATLAB_src\Sender\Modulator

ab.

¹⁹Frequency Shift Keying

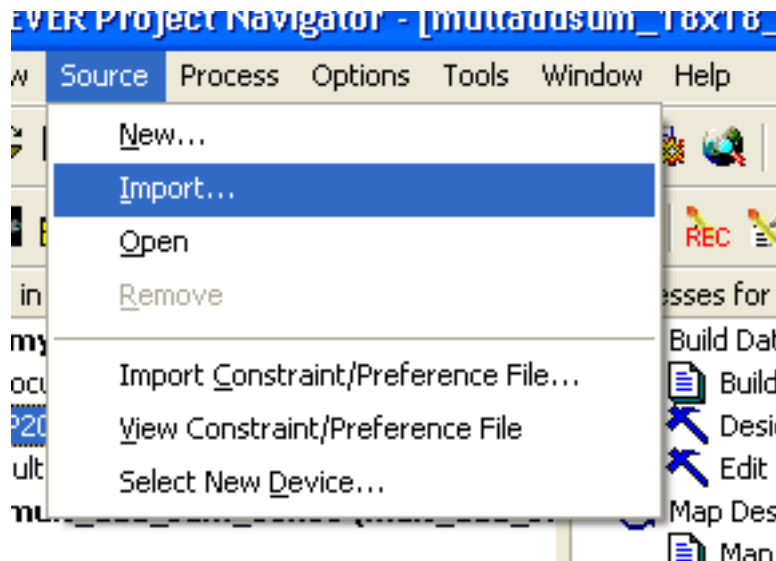


Abbildung 5.19: VHDL-Datei importieren

Aufgabe 2: Wie kann man die gleiche Funktion mit deutlich geringerem Hardwareaufwand realisieren? Implementieren sie ein einfacheres Lösungskonzept in MATLAB!

5.5.4 VHDL: Realisierung

5.5.5 VHDL: Modulator mit zwei Signalgeneratoren

Um den kompletten Modulator in Hardware aufzubauen haben Sie schon alle benötigten Module in den vorherigen Aufgaben erstellt.

Aufgabe 1.1: Setzen sie den Modulator mit Hilfe der erstellten Module in VHDL zusammen

(05_Modulator\stud_toplevel.vhd Importieren sie die benötigten Module in das aktuelle Projekt (vgl. 5.19 und 5.20). Erstellen sie die Komponentendeklaration und instanziiieren sie zwei Signalgeneratoren mit unterschiedlichen Parametern. Die beiden Modulationsfrequenzen, die sie verwenden sollen, erhalten Sie von Ihrem Betreuer.

5.5.6 MODELSIM: Simulation der Beschreibung mit zwei Signalgeneratoren

Aufgabe 1.2: Simulieren Sie den zuvor erstellten Modulator mit Hilfe eines Digital-signals, das von einem 4-Bit-PRN-Schieberegister mit XOR-Feedback über der letzten

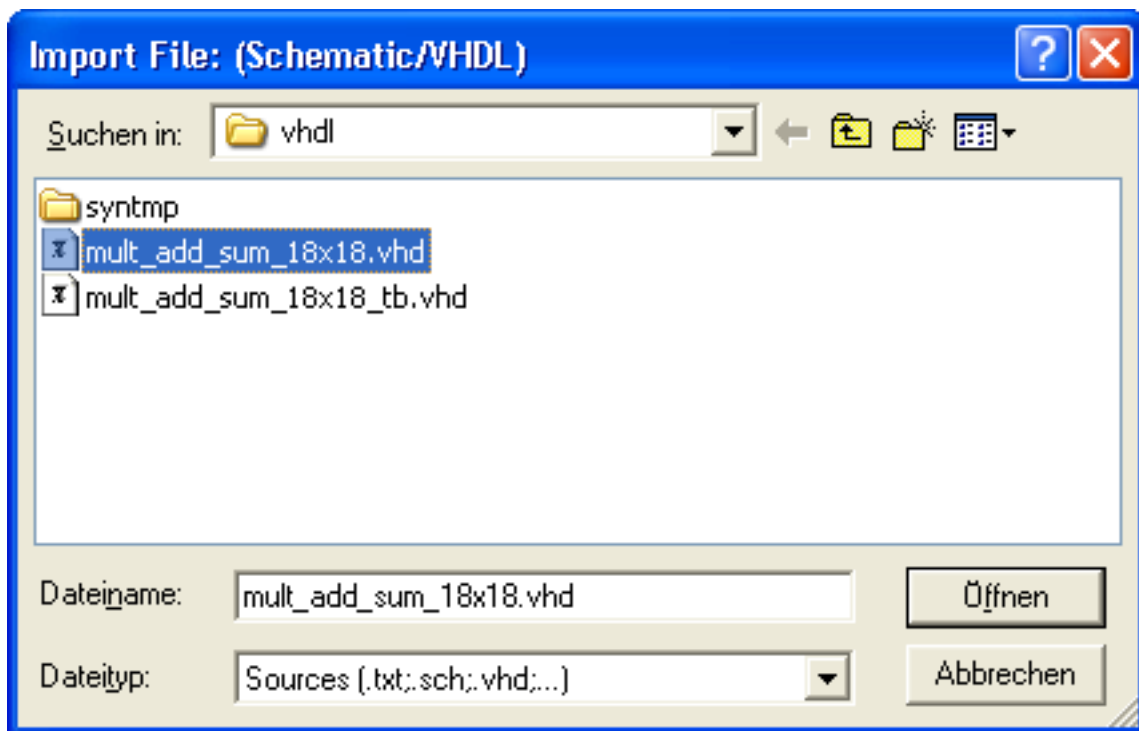


Abbildung 5.20: Importdialog

(MSB) Stufe

```
sv_prn_reg(3) <= sv_prn_reg(1) xor sv_prn_reg(0)
```

erzeugt wird. Als Anfangsbelegung verwenden sie "1001". Der Schiebetakt soll bei etwa 1Hz liegen.

Aufgabe 2: Betrachten sie das modulierte Signal zu den Umschaltzeitpunkten genauer. Was stellen Sie fest?

5.5.7 VHDL: Modulator mit einem Signalgenerator

Aufgabe 3: Verbessern Sie das Verhalten, indem Sie nur einen Signalgenerator verwenden und dessen Drehwinkel über einen Multiplexer verändern.

5.5.8 MODELSIM: Simulation der Beschreibung mit einem Signalgenerator

Simulieren Sie auch dies und kontrollieren nun das Verhalten im Übergangsbereich.

Was stellen sie bezüglich der Stetigkeit des Ausgangssignals fest?

Warum ist dies günstig für unsere Anwendung?

5.5.9 TEST: Praxistest

Testen Sie nun Ihren Entwurf auf der Hardware.

6 Empfänger

6.1 Aufbau

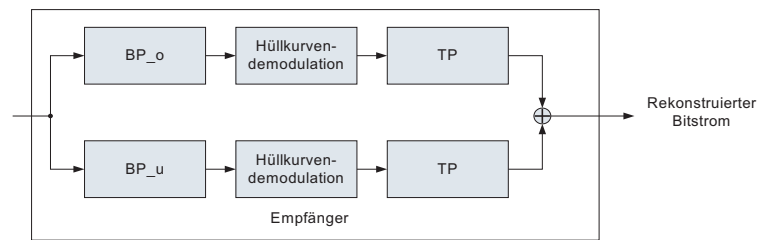


Abbildung 6.1: Blockschaltbild der Empfängerstufe

Der Empfänger ist aus zwei identischen Zweigen aufgebaut. Jeder Zweig besteht, wie in Abb. 6.1 dargestellt, aus einem Bandpass, einem Hüllkurvendemodulator und einem Tiefpass. Am Eingang wird das FSK-modulierte Signal (vgl. Abb. 6.2 des Senders) eingespeist und auf die beiden Empfangszweige aufgeteilt.

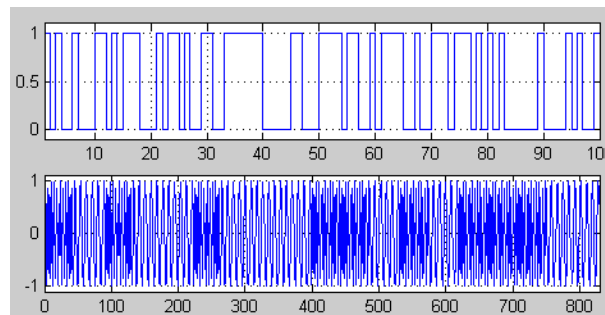


Abbildung 6.2: Bitstrom und Sendesignal

Jeder Bandpass hat die Aufgabe, eine der beiden Signalfrequenzen zu separieren, so dass pro Zweig nur noch eine einzige Frequenz detektiert werden kann. Somit liefert also in einem Zweig nur noch die der Null zugeordnete Frequenz einen Beitrag, im anderen Zweig die der Eins zugeordnete Frequenz. Dies ist in Abbildung 6.3 dargestellt.

Um nun wieder ein eindeutiges Signal zu erhalten, wird eine Hüllkurvendemodulation durchgeführt. Hier wird durch eine einfache Quadrierung des Signals die negative Halbwelle des Sinussignals nach oben “geklappt” (vgl. Abb. 6.4) und danach mit einem Tiefpass geglättet. Das Ergebnis ist in Abb. 6.5 dargestellt.

Zum Abschluss werden die beiden Signale noch von einander subtrahiert (vgl. Abb. 6.6). Im Idealfall sollte das auf diese Weise rekonstruierte Signal dem zu Beginn erzeugten Bitstrom entsprechen.

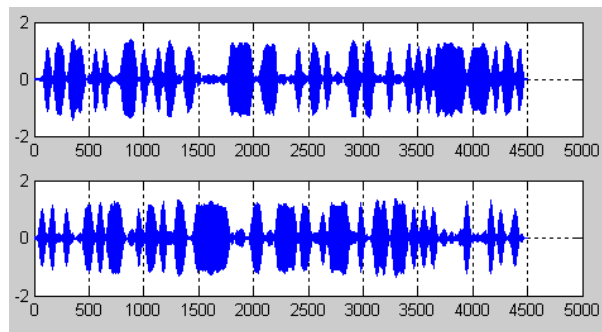


Abbildung 6.3: Bandpassgefilterte Signale

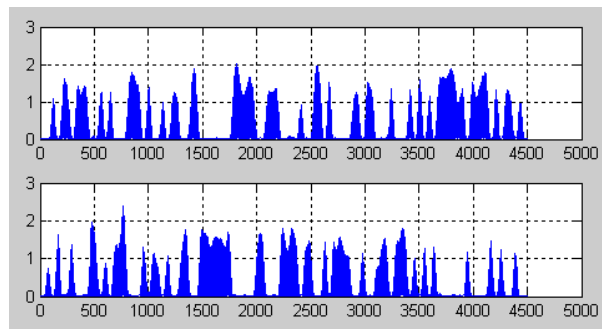


Abbildung 6.4: Bandpassgefiltertes und quadriertes Signal

Sie werden nun in den folgenden Kapiteln nach und nach die einzelnen Blöcke des Empfängers selbst entwerfen und die Übertragungsstrecke sowohl als Simulation als auch in Hardware aufbauen.

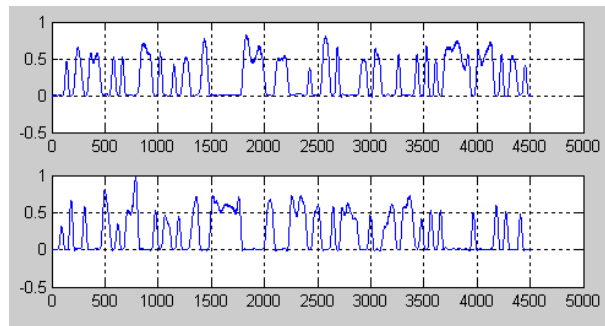


Abbildung 6.5: Tiefpassgefiltertes Signal

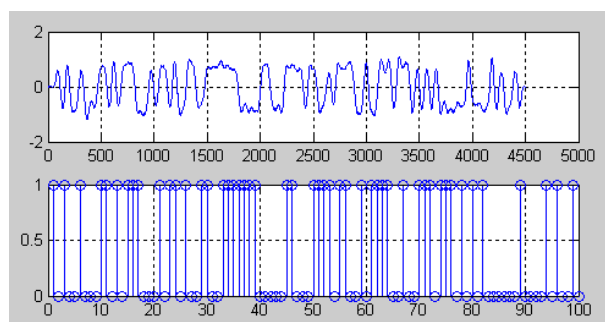


Abbildung 6.6: Rekonstruiertes Empfangssignal

6.2 Grundlagen Digitaler Filter

6.2.1 Einführung

Der Entwurf, die Realisierung und der Einsatz von digitalen Filtern ist das klassische Anwendungsgebiet der digitalen Signalverarbeitung. In der Theorie sind digitale Filter schon seit Beginn der 1970er Jahre bekannt. Allerdings konnten sie erst mit Aufkommen der DSPs in den 1980er Jahren an Popularität gewinnen.

Unter einem Filter versteht man ein System, das gewisse Frequenzkomponenten im Vergleich zu anderen verändert, beispielsweise unterdrückt, verstärkt oder in ihrer Phase verschiebt. Im Bereich der digitalen Filter spielen stabile und kausale LTI-Systeme, die durch rationale Übertragungsfunktionen mit reellen Koeffizienten beschrieben werden können, die wichtigste Rolle. Solche Filter werden meistens als *lineare Digitalfilter* bezeichnet.

6.2.1.1 Echtzeitsystem zur digitalen Filterung

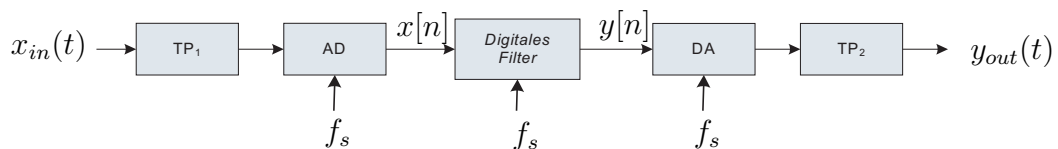


Abbildung 6.7: Echtzeitsystem zur digitalen Filterung

Im Bild 6.7 wird der grundsätzliche Aufbau eines Echtzeitsystems zur digitalen Filterung dargestellt. Das zeitkontinuierliche Eingangssignal $x_{in}(t)$ wird zunächst auf ein analoges Tiefpassfilter TP_1 geführt, das der Bandbegrenzung dient (Antialiasing). Das bandbegrenzte Signal wird durch den folgenden AD-Wandler abgetastet und in die „digitale Welt“ übersetzt. Das Signal besteht nun aus einer Folge von Abtastwerten, die dem digitalen Filter zugeführt werden. Die Filterfunktion wird von einem Digitalrechner übernommen, der aus den Eingangswerten $x[n]$ die Ausgangsfolgewerte $y[n]$ berechnet. Der DA-Wandler setzt das vom Filter ausgegebene Signal in eine zeitkontinuierliche, treppenförmige Spannung um. Anschließend wird es mit einem Tiefpassfilter TP_2 geglättet und der wertekontinuierliche Verlauf zurückgegeben.

Die Tiefpässe TP_1 und TP_2 sind Bandbegrenzungsfilter zur Unterdrückung hoher Frequenzen. Wünscht man zusätzlich eine Unterdrückung der DC-Komponenten, wie zum Beispiel in der Audiosignalverarbeitung, dann werden sie durch Bandpassfilter ersetzt.

6.2.1.2 Grundlegende Filterfunktionen

Für unsere Anwendung sind lediglich die klassischen Filter, wie Tiefpass-, Bandpass- und Hochpassfilter von Bedeutung. Die Funktion der Filter wird durch die schematische Darstellung der Amplituden-Frequenzgänge in Abb. 6.8 verdeutlicht.

Die Parameter f_{passx} nennt man Durchlassfrequenzen, die f_{stopx} Sperrfrequenzen. Die drei Frequenzbereiche, die durch die Durchlass- und Sperrfrequenzen begrenzt werden

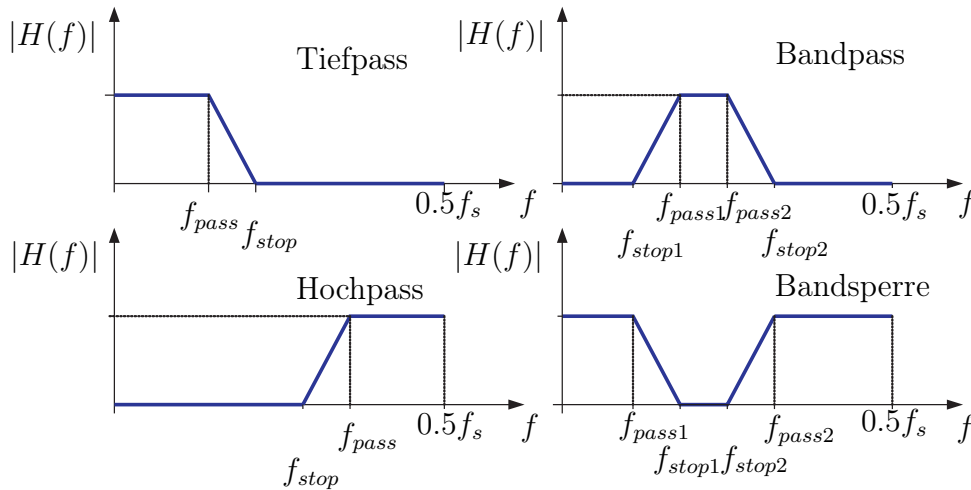


Abbildung 6.8: Filterfrequenzgänge

heissen Durchlass-, Übergangs- und Sperrbereich.

6.2.1.3 Das Digitalfilter als LTI-System

Digitale Filter sind stabile, kausale LTI-Systeme, die sich mit einer rationalen Übertragungsfunktion beschreiben lassen:

$$H_z(z) = \frac{Y_z(z)}{X_z(z)} = \frac{b_0 + b_1 z^{-1} + \dots + b_N z^{-N}}{1 + a_1 z^{-1} + \dots + a_M z^{-M}} \quad (6.1)$$

Die *Übertragungsfunktion* $H(z)$ definiert das Übertragungsverhalten des Systems und die größere der beiden natürlichen Zahlen N und M legt ihre Ordnung fest. Sind alle rekursiven Koeffizienten a_i gleich Null, so spricht man von einem *nichtrekursiven LTI-System* oder einfach von einem *FIR-Filter*¹, andernfalls von einem *rekursiven LTI-System* oder einem *IIR-Filter*².

Unter dem *Frequenzgang* versteht man die Übertragungsfunktion, ausgewertet auf dem Einheitskreis der z -Ebene:

$$H(e^{j2\pi fT}) = H(z)|_{e^{j2\pi fT}} \quad (6.2)$$

Den Betrag $|H(e^{j2\pi fT})|$ nennt man *Amplitudengang*, den Winkel $\angle H(e^{j2\pi fT})$ *Phasengang*. Die negative Ableitung des Phasengangs ist die *Gruppenlaufzeit*:

$$\tau_g(e^{j2\pi fT}) = -\frac{1}{2\pi} \frac{d\angle H(e^{j2\pi fT})}{df} \quad (6.3)$$

Die Gruppenlaufzeit charakterisiert die Verzögerung eines LTI-Systems. Eine weitere, weniger gebräuchliche, Größe ist die *Phasenlaufzeit*:

$$\tau_p(e^{j\Omega}) = -\frac{\arg(H(\Omega))}{\Omega} = -\frac{\angle H(e^{j\Omega})}{\Omega} \text{ mit } \Omega = 2\pi fT \quad (6.4)$$

¹Finite-Impulse-Response Filter

²Infinite-Impulse-Response Filter

Mit diesem Parameter kann die Reaktion eines LTI-Systems auf eine sinusförmige Eingangsgröße der Frequenz Ω im eingeschwungenen Zustand in der Form

$$y[n] = |H(\Omega)| \cos(\Omega(n - \tau_p)) \quad (6.5)$$

geschrieben werden.

Die Phasenlaufzeit ist somit gleich der Anzahl der Abtastintervalle, mit der eine Sinusschwingung beim Durchlaufen eines zeitdiskreten LTI-Systems verzögert wird.

Die Gruppenlaufzeit dagegen ist gleich der Anzahl der Abtastintervalle, um die die Hüllkurve eines modulierten Systems verzögert wird.

Transformieren wir die Übertragungsfunktion in den Zeitbereich, so erhält man die Impulsantwort $h[n]$:

$$H(z) \bullet \longrightarrow h[n] \quad (6.6)$$

Im Frequenzbereich ist das Ausgangssignal gleich dem Eingangssignal multipliziert mit der Übertragungsfunktion,

$$Y(z) = H(z)X(z) \quad (6.7)$$

im Zeitbereich gleich dem Eingangssignal gefaltet mit der Impulsantwort.

$$y[n] = h[n] * x[n] \quad (6.8)$$

Setzen wir für $H(z)$ in Gl. (6.7) die rationale Funktion (6.1) ein, führt die Rücktransformation auf die *Differenzengleichung*:

$$y[n] = - \sum_{i=0}^M a_i y[n-i] + \sum_{i=0}^N b_i x[n-i] \quad (6.9)$$

Betrachtet man die Differenzengleichung genauer, so kann man leicht erkennen, dass diese mit Hilfe einfacher Addierer, Multiplizierer und Verzögerungsglieder in Hardware realisierbar ist.

Letztendlich kann man die Übertragungsfunktion noch in einer *Pol-Nullstellen-Darstellung* angeben:

$$H_z(z) = b_0 z^{M-N} \frac{(z - z_1)(z - z_2) \dots (z - z_N)}{(z - p_1)(z - p_2) \dots (z - p_M)} \quad (6.10)$$

Die Parameter p_i werden als *Pole* bezeichnet, die z_i als *Nullstellen*.

Zusammenfassung: Ein lineares Digitalfilter beschreibt man gewöhnlich mit Hilfe seiner Übertragungsfunktion. Aus ihr lassen sich die wichtigsten Funktionen und Gesetzmäßigkeiten einfach ableiten.

Aufgabe des Filterentwurfs ist es, die Ordnung und die Koeffizienten der Filterübertragungsfunktion zu einem vorgegebenen Zielfrequenzgang zu bestimmen. Wir werden in der Folge geeignete Methoden dazu diskutieren.

6.2.2 Eigenschaften und Strukturen von FIR-Filtern

Wie oben schon angesprochen teilt man die digitalen Filter in FIR- und IIR-Filter ein. Beide Klassen haben interessante Eigenschaften, wobei wir uns in diesem Praktikum auf die Verwendung von FIR-Filtern beschränken werden.

Die Struktur der Filter beschreibt man häufig in Form eines Signalflussdiagramms oder auch eines Blockdiagramms. Kennen wir diese, so sind wir in der Lage ein Programm zu schreiben und digitale Filter auf einem Rechner zu implementieren.

6.2.2.1 Grundlagen

Die Differenzengleichung eines FIR-Filters lautet:

$$y[n] = \sum_{i=0}^N b_i x[n-i] \quad (6.11)$$

Damit könnte ein nichtrekursives FIR-System 3. Ordnung folgendermaßen aussehen:

$$y[n] = b_0 x[n] + b_1 x[n-1] + b_2 x[n-2] + b_3 x[n-3] \quad (6.12)$$

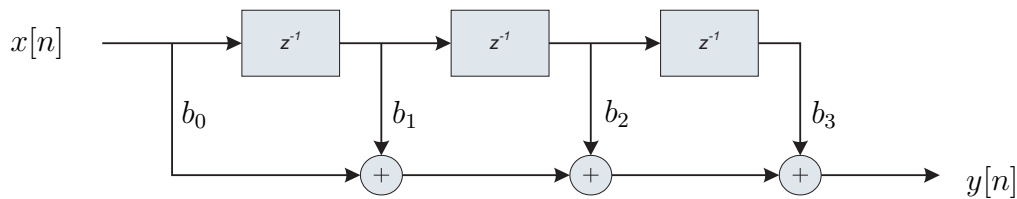


Abbildung 6.9: FIR-Filter 3. Ordnung

Anhand dieses einfachen Beispiels kann man erkennen, dass die Impulsantwort eines nichtrekursiven LTI-Systems N -ter Ordnung gleich den Koeffizienten der Differentialgleichung ist. Die Länge der Impulsantwort ist dann $N + 1$:

$$h[n] = h_0, h_1, \dots, h_N = b_0, b_1, \dots, b_N \quad (6.13)$$

Diese endliche Länge führte zur Bezeichnung FIR-Filter (Finite-Impulse-Response-Filter).

Die Übertragungsfunktion lautet allgemein:

$$H(z) = b_0 + b_1 z^{-1} + \dots + b_N z^{-N} \quad (6.14)$$

Durch Erweiterung mit z^N kann die Übertragungsfunktion auch in der Form

$$H(z) = \frac{b_0 z^N + b_1 z^{N-1} + \dots + b_N}{z^N} \quad (6.15)$$

geschrieben werden. Daraus geht hervor, dass alle Pole im Ursprung liegen und dass das FIR-Filter somit immer stabil ist.

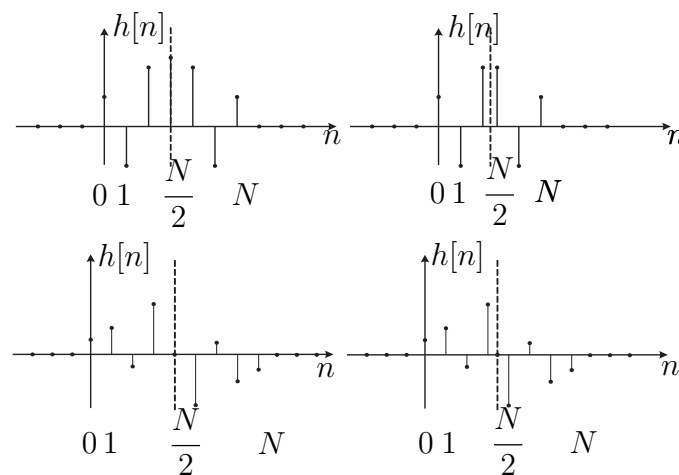


Abbildung 6.10: Symmetrieeigenschaften von FIR Filter

6.2.2.2 Eigenschaften von FIR-Filtern

Alle Entwurfsverfahren liefern symmetrische FIR-Filter, deren Impulsantwort immer spiegel- oder punktsymmetrisch ist (vgl. Abb. 6.10).

Symmetrische FIR-Filter haben, abgesehen von 180° -Phasensprüngen, einen linearen Phasengang. Daher werden sie auch als linearphasige Filter bezeichnet. Die Gruppenlaufzeit $\tau_g(f)$ symmetrischer Filter ist konstant und ihr Wert gleich $NT/2$. Es kommt also zu keinen Gruppenlaufzeitverzerrungen!

Filter mit konstanter Gruppenlaufzeit haben die angenehme Eigenschaft, dass sie Signale im Durchlassbereich nur verzögern, aber nicht verzerren. Zudem bleibt die Symmetrie symmetrischer Pulse erhalten, was für viele Anwendungen vorteilhaft ist.

Beispiel: Hier als Beispiel ein linearphasiger FIR-Tiefpass (vgl. Abbildungen 6.11 bis 6.15):

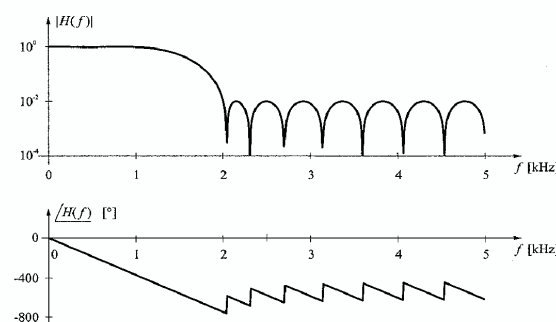


Abbildung 6.11: Frequenzgang eines linearphasigen TP-Filters

Es handelt sich um ein linearphasiges Tiefpassfilter. Abgesehen von 180° -Phasensprüngen (die an den Nullstellen des Amplitudengangs auftreten und daher keinen Einfluss auf das Übertragungsverhalten haben) ist der Phasengang linear. Die Länge des Filters ist

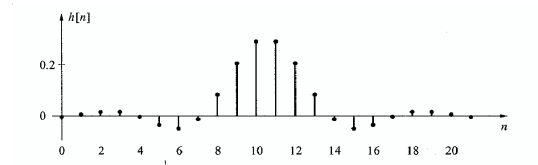


Abbildung 6.12: Impulsantwort eines linearphasigen TP-Filters

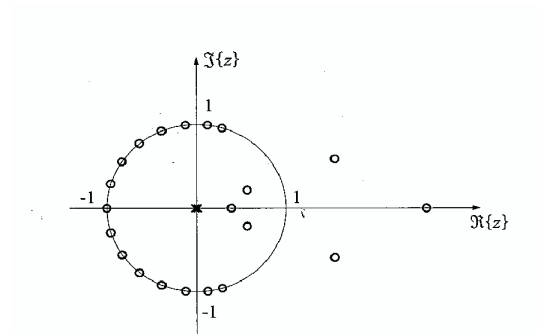


Abbildung 6.13: Pol-Nullstellendiagramm eines linearphasigen TP-Filters

22, demnach muss die Übertragungsfunktion 21 Nullstellen und 21 im Ursprung liegende Pole haben.

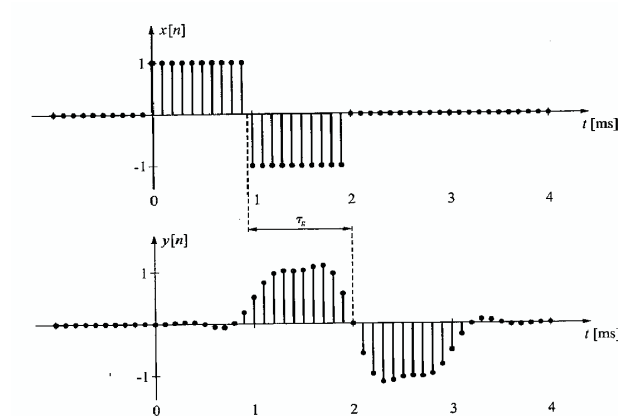


Abbildung 6.14: Ein- und Ausgangsverhalten bei Anregung mit einem punktsymmetrischen Doppelrechteckpuls: Zeitbereich

Der tiefpassgefilterte Rechteckimpuls ist erwartungsgemäß an seinen Ecken abgerundet, aber immer noch punktsymmetrisch (vgl. Abb. 6.14). Der Schwerpunkt des Ausgangspulses ist gegenüber dem Schwerpunkt des Eingangs-Pulses um die Gruppenlaufzeit $\tau_g = 1.05 \text{ ms}$ verzögert.

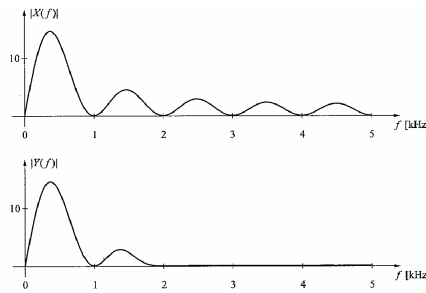


Abbildung 6.15: Ein- und Ausgangsverhalten im Frequenzbereich

6.2.2.3 Strukturen von FIR-Filtern

Am einfachsten lässt sich das Filter in einer Straight-Forward-Implementierung realisieren, indem man die Differenzengleichung 1:1 umsetzt. Diese Form nennt sich Direktform- oder Transversalfilterstruktur (vgl. Abb. 6.16).

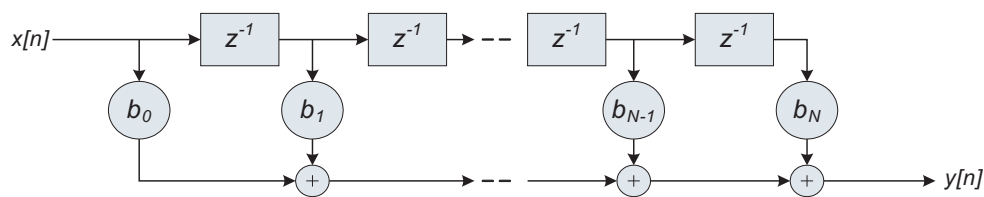


Abbildung 6.16: Direktform- oder Transversalfilterstruktur bei FIR-Filtern

Durch Ausnutzung der Symmetrie kann man etwa die Hälfte der Multiplikationen einsparen (vgl. Abb. 6.17).

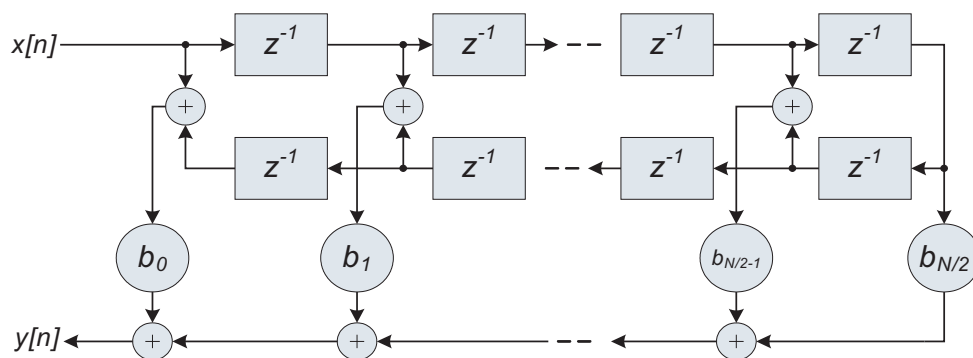


Abbildung 6.17: Linearphasenstruktur eines FIR-Filters

Die Differenzengleichung lautet dann:

$$y[n] = b_0(x[n] + x[n - N]) + b_1(x[n - 1] + x[n - N + 1]) + \dots + b_{N/2}x[n - N/2] \quad (6.16)$$

6.2.3 Entwurf von FIR-Filtern

Man beginnt das Filterdesign mit der Spezifikation der einzelnen Filtergrenzfrequenzen. Da Filter mit rechteckförmigen Amplitudengängen eine unendlich hohe Ordnung besitzen würden, sind sie in der Praxis nicht realisierbar. Man legt daher, wie beim Bandpass in Abb. 6.18 dargestellt, einen Toleranzbereich fest, in dem sich der Amplitudengang des Filters befinden darf.

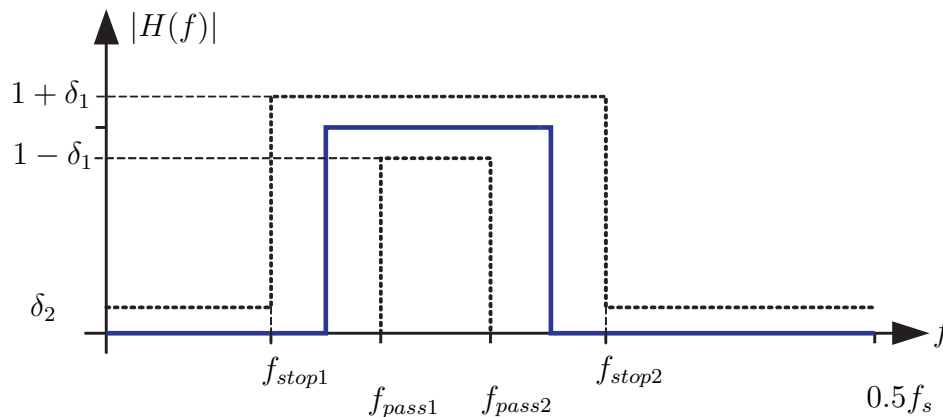


Abbildung 6.18: Toleranzschema des Amplitudengangs

Die maximal zulässigen Abweichungen δ_1 und δ_2 vom idealen Amplitudengang heißen *Ripple* im Durchlass- bzw. Sperrbereich. Die Bereiche, die durch das Festlegen der Grenzfrequenzen entstehen, nennt man Durchlass-, Übergangs- und Sperrbereich. Die zugehörigen Grenzfrequenzen heißen Durchlass- und Sperrfrequenzen.

Üblicherweise wird nur der Frequenzbereich von 0 bis $f_s/2$ bzw. von 0 bis π dargestellt, da sich danach der Frequenzgang wiederholt.

Die Wahl eines geeigneten Toleranzschemas ist eine typische Ingenieursaufgabe und hängt von der Anwendung des Filters ab.

Allgemein gilt, dass die Ordnung, und damit die Komplexität des Filters, steigt, je enger das Toleranzschema gewählt wird. Für eine gegebene Anwendung wird der Toleranzbereich daher so groß wie möglich gewählt, um die Filterordnung und somit auch die Zahl der nötigen Multiplikationen klein zu halten.

Die beiden wichtigsten Entwurfsmethoden für FIR-Filter sind die

- Fenstermethode und die
- Optimalmethode (auch Equiripple-Verfahren, Remez-Entwurf oder Chebychev-Approximation genannt).

6.2.3.1 Filterentwurf mit der Fenstermethode

Entwurfsschritte der Fenstermethode

1. Festlegen der Eckfrequenz(en) des idealen rechteckförmigen Filterfrequenzgangs $H_{ideal}(\Omega)$.

2. Berechnen (oder nachlesen in Tabellen) der entsprechenden Filterimpulsantwort $h_{ideal}(n)$ (diese ist unendlich lang und nicht kausal und daher nicht realisierbar)
3. Multiplizieren der idealen Impulsantwort $h_{ideal}(n)$ mit einer geeigneten Fensterfunktion $w[n]$

$$h[n] = h_{ideal}[n] \cdot w[n] \quad (6.17)$$

4. Verzögern der resultierenden Impulsantwort $h[n]$, so dass ein kausales (realisierbares) System entsteht.

Ideale Impulsantworten der grundlegenden Filterfunktionen

- Idealer Tiefpass mit der Eckfrequenz Ω_c

$$h_{ideal}[n] = \frac{\Omega_c}{\pi} \text{sinc}(n\Omega_c) \quad (6.18)$$

- Idealer Bandpass mit den Eckfrequenzen Ω_1 und Ω_2

$$h_{ideal}[n] = \frac{\Omega_2}{\pi} \text{sinc}(n\Omega_2) - \frac{\Omega_1}{\pi} \text{sinc}(n\Omega_1) \quad (6.19)$$

In den Abb. (6.19) und (6.20) ist der Entwurfsablauf mit der Fenstermethode im Zeit- und Frequenzbereich dargestellt.

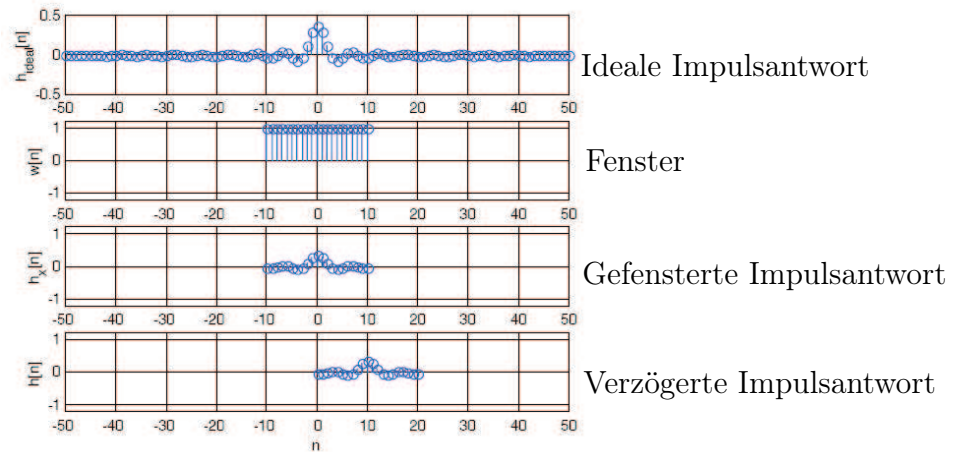


Abbildung 6.19: Fenstermethode im Zeitbereich

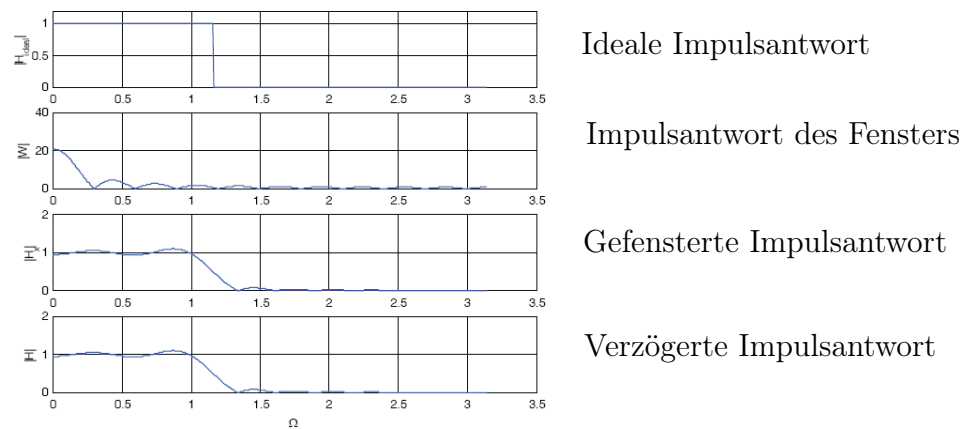


Abbildung 6.20: Fenstermethode im Frequenzbereich

Zur Wahl des Fensters: Für die Fenstermethode stehen unterschiedliche Fenstertypen zur Verfügung. Hier sind drei der Gebräuchlichsten aufgeführt. Informationen zu weiteren Fenstertypen finden sie unter anderem in der Online-Hilfe von MATLAB.

- Rechteckfenster:
 - hat große Rippel im Amplitudengang zur Folge
 - Das Toleranzschema resultiert nicht aus dem Input des Entwurfs, sondern ist das mehr oder weniger zufällige Resultat des Entwurfs
- Hanningfenster:
 - Die Rippel im Durchlassbereich sind deutlich kleiner als beim Entwurf mit einem Rechteckfenster
 - Wie schon beim Entwurf mit dem Rechteckfenster ist das Toleranzschema nicht der Input des Entwurfs, sondern ein mehr oder weniger zufälliges Resultat
- Kaiserfenster:
 - Zusätzlich zur Fensterlänge kann ein zusätzlicher Parameter (β) frei gewählt werden
 - Dies ermöglicht, dass mit dem Kaiserfenster tatsächlich das Toleranzschema vorgegeben werden kann. Die beiden Parameter $N + 1$ (Fensterlänge) und β können in der Folge bestimmt werden, das Endresultat des Entwurfs sind wiederum die Ordnung N des Filters sowie die Filterkoeffizienten b_0, \dots, b_n
 - Die Filterordnung ist üblicherweise relativ hoch, da das Toleranzschema nicht optimal ausgenutzt wird

Eine graphische Darstellung hierzu finden sie in Abb. 6.21.

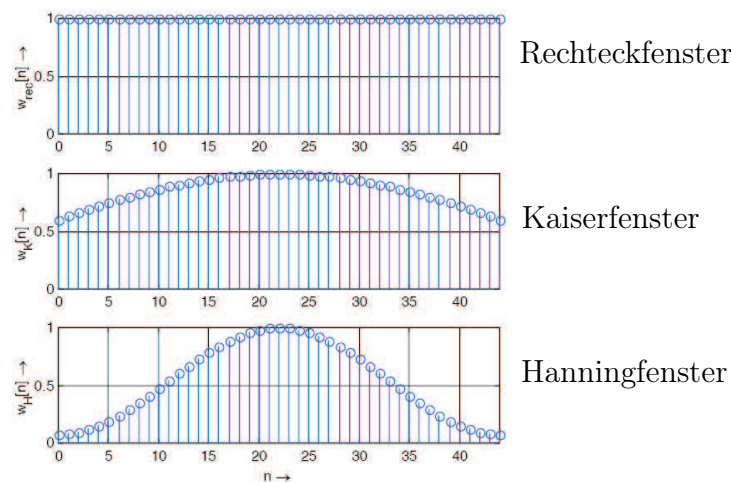


Abbildung 6.21: Fenstertypen

6.2.3.2 Filterentwurf mit der Optimalmethode

Mit dieser Methode lassen sich Filter entwerfen, die eine gleichmäßige Welligkeit im Durchlass- und Sperrbereich aufweisen. Dabei nutzen sie den gesamten Toleranzbereich

aus. Die daraus resultierende Filterordnung N ist im Allgemeinen wesentlich kleiner, als diejenige des Fensterentwurfs. Die Theorie hinter der Optimalmethode ist sehr umfangreich und würde den Rahmen des Praktikums sprengen. Allerdings wird die Optimalmethode als Standardverfahren von vielen Programmpaketen zur Signalverarbeitung unterstützt.

Die Optimalmethode ermöglicht nicht nur den Entwurf der vier Filtergrundarten, sondern auch den Entwurf von Multibandfiltern, Differentiatoren u.a.

6.2.4 Nichtideale Effekte bei digitalen Filtern

6.2.4.1 Quantisierungsrauschen durch AD-Wandlung

Bei der AD-Wandlung wird das analoge Signal zunächst abgetastet und in ein zeitdiskretes Signal überführt. Dieses wird in einem zweiten Schritt mit Hilfe einer Quantisierungskennlinie bewertet. Das resultierende digitale Signal ist jetzt zeit- und wertediskret.

Da digitale Signale, je nach Wortlänge, nur bestimmte Werte annehmen können, muss das Analogsignal entsprechend abgeschnitten oder gerundet werden. In jedem Fall geht bei diesem Schritt ein geringer Teil der Information verloren, weshalb das ursprüngliche Signal nicht mehr fehlerfrei rekonstruiert werden kann. Man spricht von einem **Quantisierungsfehler**. Je größer die Wortlänge des AD-Wandlers ist, desto geringer fällt der resultierende Fehler aus.

Quantisierungsrauschen: Man kann den Quantisierungsfehler als ein Zufallssignal $e[n]$ modellieren, das durch die Quantisierung entsteht und deshalb als Quantisierungsrauschen bezeichnet wird:

$$x_Q[n] = x[n] + e[n] \quad (6.20)$$

Das quantisierte Signal $x_Q[n]$ kann als Überlagerung des amplitudenkontinuierlichen Signals $x[n]$ und des Fehlersignals $e[n]$ aufgefasst werden.

Ein Qualitätsmaß in der Signalverarbeitung ist der sogenannte Signal-Rauschabstand SNR (Signal to Noise Ratio):

$$SNR_{dB} = 10 \cdot \log \left(\frac{P_S}{P_N} \right) \quad (6.21)$$

mit: $P_S \rightarrow$ Mittlere Leistung des Nutzsignals $x[n]$
 $P_N \rightarrow$ Mittlere Leistung des Fehlersignals $e[n]$

Für den Sonderfall, dass das Eingangssignal symmetrisch gleichförmig mit der Auflösung B (Anzahl und Bits) quantisiert wird und im gesamten Aussteuerbereich gleichverteilt ist, gilt:

$$SNR_{dB} \approx B \cdot 6dB \quad (6.22)$$

→ Jede Erhöhung der Auflösung um 1 Bit verbessert also den Abstand zwischen Nutzanteil und Quantisierungsrauschen um 6dB!

6.2.4.2 Quantisierung der Filterkoeffizienten

Die Entwurfsverfahren für digitale Filter liefern die b - und a -Koeffizienten der Übertragungsfunktion

$$H_z(z) = \frac{b_0 + b_1 z^{-1} + \dots + b_N z^{-N}}{1 + a_1 z^{-1} + \dots + a_M z^{-M}} \quad (6.23)$$

mit großer Genauigkeit (meist in Double-Precision-Floating-Point). Bei der Implementierung in Hard- oder Software können die Koeffizienten aber nur mit einer begrenzten Anzahl von Bits dargestellt werden, und aufgrund der günstigeren Komponenten meist auch nur in Fixed-Point-Darstellung. Die Folgen dieser reduzierten Genauigkeit können zum Beispiel

- eine Verletzung der Entwurfsspezifikation nach der Quantisierung oder
- Instabilitäten ehemals stabiler Filter bei IIR-Systemen

sein.

FIR-Filter: Bei FIR-Filtern der Länge N entsprechen die Filterkoeffizienten der Impulsantwort:

$$h[n] = \begin{cases} b_n & \text{für } n = 0, 1, \dots, N \\ 0 & \text{sonst} \end{cases} \quad (6.24)$$

Die Quantisierung der Filterkoeffizienten wirkt sich also direkt auf die Impulsantwort des Systems aus und damit auf die Übertragungsfunktion und den Frequenzgang.

Fasst man die Koeffizienten-Quantisierungsfehler als additive Störgröße $\Delta h[n]$ auf, so resultiert das Modell einer Parallelschaltung aus dem unquantisierten, also dem idealen System, und der Störung durch die Koeffizientenquantisierung (vgl. Abb. 6.22)

$$h_Q[n] = h[n] + \Delta h[n] \quad (6.25)$$

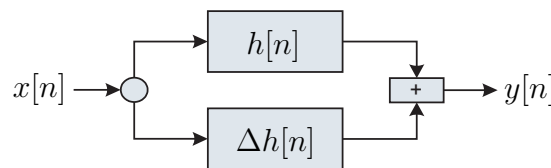


Abbildung 6.22: Fehlermodell für quantisierte Koeffizienten

Damit gilt für die obere Schranke des Fehlers im Frequenzgang:

$$\max_{\Omega} |\Delta H(\Omega)| \leq (N + 1) \frac{LSB}{2} \quad (6.26)$$

Für diese Abschätzung wurde die sehr pessimistische Annahme unterstellt, dass sich alle Quantisierungsfehler der einzelnen Koeffizienten ungünstig überlagern. Insbesondere bei langen Impulsantworten kompensieren sich die Fehler zum Teil. Daher arbeiten realistische Abschätzungen meist mit stochastischen Modellen. In der Praxis sollte man in JEDEM Fall vor der Implementierung das Filter simulieren!

IIR-Filter: Auf IIR-Filter und die bei der Koeffizientenquantisierung entstehenden Probleme soll hier nicht weiter eingegangen werden, da wir aufgrund der deutlich besseren Stabilität ausschließlich mit FIR-Strukturen arbeiten werden. Näheres hierzu kann man allerdings in [11] oder in [5] nachlesen.

6.2.4.3 Quantisierte Arithmetik

Die Quantisierung der Filterkoeffizienten ändert “nur” die Übertragungsfunktion und kann prinzipiell schon im Filterentwurf berücksichtigt werden. Wortlängeneffekte bei den arithmetischen Operationen treten dagegen während des Betriebs auf und können sehr unschöne Effekte verursachen.

Lineare Digitalfilter bestehen aus Addierern, Verzögerungsgliedern und Multiplizieren. Sowohl die Additionen als auch die Multiplikationen sind von der Quantisierung direkt betroffen.

Realisiert man ein Digitalfilter in Gleitkommadarstellung, so können Quantisierungsfehler in der Regel ausgeschlossen werden und auch Überläufe brauchen nicht berücksichtigt zu werden, da sie vernachlässigbar klein sind.

Bei der Realisierung in Festkommadarstellung dagegen können bei der Addition *Überläufe* und in der Folge *große Grenzzyklen* entstehen.

Bei der Multiplikation kann es zu *Rundungsfehlern* kommen, die *kleine Grenzzyklen* nach sich ziehen. Da man bei Festkommaimplementierungen allerdings in der Regel Abtastwerte und Koeffizienten als Fractional-Zahlen darstellt liegen die Ergebnisse von Multiplikationen im Bereich von $[-1,1[$, wodurch Überläufe vermieden werden (Ausnahme: $(-1) \cdot (-1) = 1$).

Grenzzyklen: Bei der Implementierung digitaler Systeme mit endlicher Wortlänge können parasitäre Schwingungen beobachtet werden, die trotz Stabilität der Filter auftreten. Diese Oszillationen werden Grenzzyklen oder Limit-Cycles genannt und entstehen durch Nichtlinearitäten in den Rückkoppelschleifen der Filter. Man unterscheidet

Kleine Grenzzyklen (auch Granulargrenzzyklen) mit kleinen Amplituden, deren Ursache die Quantisierung bei Multiplikationen im Rückkoppelzweig ist und

Große Grenzzyklen (auch Überlaufgrenzzyklen) mit großen Amplituden, die durch eine Überlaufbehandlung bei Additionen in den Rückkopplungen hervorgerufen werden.

Da Grenzzyklen nur bei Filterstrukturen mit Rückkopplung entstehen können sollen sie hier nur kurz angerissen werden. Für eine genauere Betrachtung sei auf die einschlägige Literatur verwiesen.

Das Auftreten dieser Effekte hängt von verschiedenen Parametern ab:

- Filterkoeffizienten (steile und schmalbandige Filter neigen mehr dazu)
- Inhalt der Verzögerungselemente vor dem Einschalten
- Eingangssignal (z.B. Zero-Input-Limit-Cycles)
- Filterstruktur

- Skalierung
- Wortlänge
- Quantisierungsart bzw. Überlaufkennlinie

Kleine Grenzzyklen: Sowohl beim Runden, als auch beim Abschneiden entstehen Fehler (Rundungsfehler oder Quantisierungsfehler). Während beim Betragsabschneiden die Wortlängenverkürzung stets zur Null hin vorgenommen wird, der Betrag der Zahl durch die Quantisierung also nicht zunehmen kann, wirkt eine Quantisierung mit Runden beim Aufrunden wie eine zusätzliche Signalquelle, die dem System zusätzliche Energie zuführt und zu kleinen Grenzzyklen führen kann. Bei Quantisierung mit Betragsabschneiden wird dem System keine zusätzliche Energie zugeführt, kleine Grenzzyklen werden auf Kosten einer größeren Ungenauigkeit vermieden.

Da kleine Grenzzyklen nur im Zusammenhang mit einer Rückkopplung auftreten können sind sie für uns uninteressant. Lediglich bei Verwendung von IIR-Filtern sollte man diesen Effekt im Hinterkopf behalten.

Große Grenzzyklen: Große Grenzzyklen sind weitaus unangenehmer, da ihre Amplitude den gesamten Aussteuerbereich umfassen kann. Das Auftreten großer Grenzzyklen macht daher die Ergebnisse der Filterung komplett unbrauchbar. Mögliche Gegenmaßnahmen sind:

- **Sättigungskennlinien:** In diesem Fall wird bei einem positiven oder negativen Überlauf die größte bzw. die kleinste darstellbare Maschinenzahl gesetzt.
- Geeignete Skalierung der Filterkoeffizienten.

Skalierung von FIR-Filtern (zur Vermeidung von Überläufen): Da eine Skalierung der Filterkoeffizienten nur bei Filtern in Direktform-Struktur einen Effekt erzielen würde soll hier nur kurz auf diese Möglichkeit eingegangen werden. In den späteren Versuchen wird das Filter in einer Linearphasenstruktur implementiert, die durch diese Entwurfsmodifikation nicht zu beeinflussen ist.

Betrachtet man ein FIR-Filter in Direktform- oder Transversalstruktur (vgl. Abb. 6.16), so sieht es auf den ersten Blick so aus, als müsste für jeden einzelnen Summenknoten dafür gesorgt werden, dass an dessen Ausgang kein Überlauf auftreten kann. Da Teilüberläufe bei der Summenbildung in Zweierkomplement-Arithmetik keine Rolle spielen, solange das Endergebnis im Bereich $[-1,1[$ liegt, ist dies glücklicherweise nicht der Fall. Bei der Zweierkomplement-Arithmetik entsteht dann unabhängig von der Reihenfolge der einzelnen Additionen die richtige Gesamtsumme. Aus diesem Grund können für unsere Überlegungen alle Einzelsummen der FIR-Struktur wie im Bild dargestellt, zu einem Summenknoten zusammengefasst werden und es reicht sicherzustellen, dass das Ergebnis der Gesamtsumme (das Ausgangssignal) nicht überläuft.

Mit der Einführung der l_1 -Norm

$$l_1 = \sum_{i=0}^N |b_i| \quad (6.27)$$

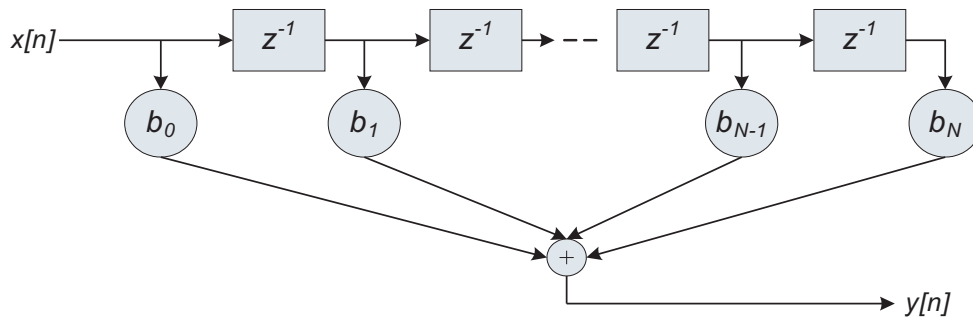


Abbildung 6.23: Überlaufbehandlung bei FIR-Filtern in Direktformstruktur

und der Skalierung der b-Koeffizienten

$$b_i = s_1 b_i \text{ mit } s_1 = \frac{1}{l_1} \quad (6.28)$$

kann somit Überlauffreiheit (bei der Direktform) garantiert werden. Allerdings wird durch die Skalierung das Ausgangssignal um den Faktor s_1 abgeschwächt (wenn $s_1 < 1$), während das Rundungsrauschen unverändert bleibt, wodurch sich das SNR mitunter deutlich verschlechtert. Man ist aus diesem Grund daran interessiert, den Skalierungsfaktor so groß wie möglich zu halten!

Das Ziel der richtigen Wahl von Filterstruktur und Skalierung ist es also, die Wahrscheinlichkeit von Überläufen zu minimieren und gleichzeitig das SNR am Ausgang zu maximieren!

Anmerkung:

Neben der l_1 -Norm gibt es noch weitere alternative Skalierungsfaktoren wie die Tschebyscheff-Norm oder die l_2 -Norm. Für detailliertere Informationen sei auch hier wieder auf [5] verwiesen.

Rundungsrauschen: Ein Maß für die Größe des Rundungsrauschens ist das vom Rundungsrauschen verursachte SNR am Filterausgang. Dieses kann man einfach durch ein möglichst starkes Anheben der Signalpegel im Inneren des Filters optimieren. Da das Rundungsrauschen bei gegebener Filterstruktur und Wortlänge unverändert bleibt wird das SNR maximiert. Allerdings darf der Dynamikbereich der Festkomma-Arithmetik nicht überschritten werden, da es sonst zu Überläufen kommen würde. Um die Maximierung des SNR zu erreichen, und gleichzeitig die Wahrscheinlichkeit von Überläufen zu minimieren kann man die Filterkoeffizienten skalieren.

Quantisierungsrauschen am Ausgang eines Filters kann durch eine additive Überlagerung eines Fehlersignals am Ausgang eines idealen Filters modelliert werden:

$$y_Q[n] = y[n] + e[n] \quad (6.29)$$

Das Fehlersignal $e[n]$ setzt sich dabei aus mehreren Fehlerquellen zusammen:

1. *Quantisierungsfehler des AD-Wandlers am Filtereingang:* Dieses Fehlersignal wird wie das Nutzsignal über das Filter übertragen und so auf den Filterausgang übersetzt.
2. *Rundungsrauschen des Filters:* Setzt sich zusammen aus Fehlern durch Runden oder Abschneiden nach Multiplikationen. Diese Fehler, die an unterschiedlichen Stellen innerhalb der Filterstruktur auftreten, werden jeweils (über unterschiedliche Teilübertragungsfunktionen) auf den Filterausgang übersetzt, wo sie als zusammengesetzter Rundungsfehler betrachtet werden können.
3. *Quantisierung des Ausgangssignal auf weniger Bits,* falls (wie es in Anwendungen häufig der Fall ist) der DA-Wandler oder ein Folgesystem mit kleineren Wortbreiten arbeiten.

Die letzte Fehlerquelle wird oft übersehen, da aufgrund der Akkumulierung der Rundungsrauschquellen normalerweise für die interne Arithmetik mehr Bits verwendet werden, als am Ausgang notwendig sind um dieses zu minimieren.

Wird nach den Multiplikationen gerundet, so kann der Fehler als gleichverteilt im Intervall $-q/2 < e < q/2$ ($q = \text{LSB}$) angesehen werden. Der Mittelwert (der eigentliche Erwartungswert) ist 0, die Varianz oder mittlere Leistung des Fehlers ist $\sigma_0^2 = q^2/12$. Beim Abschneiden liegt der Fehler im Bereich $-q < e < 0$, der Mittelwert ist dann $-q/2$. Der Mittelwert aller Abschneidefehleroperationen pflanzt sich durch das Filter bis zum Ausgang der Schaltung fort und überlagert sich dort. Der resultierende DC-Wert am Ausgang ist einfach zu berechnen, meist aber vernachlässigbar. Die Varianz der Abschneidefehler ist gleich der des Runden. Aus diesem Grund werden Abschneiden und Runden in der analytischen Betrachtung in der Regel gleich behandelt. Die durch die Rauschquelle verursachte mittlere Rauschleistung σ_y^2 ergibt sich, indem σ_0^2 mit der Rauschzahl NG multipliziert wird.

$$NG = \frac{1}{2\pi} \int_{-\pi}^{\pi} |\hat{H}(\Omega)| d\Omega \quad (6.30)$$

($\hat{H}(\Omega)$ gibt die Übertragungsfunktion vom Rauschknoten zum Systemausgang an) Hat das System mehrere unkorrelierte Rauschquellen, dann überlagern sich am Ausgang des Systems die übertragenen mittleren Rauschleistungen.

Zusammenfassung: Unerwünschte Effekte aufgrund von Koeffizientenquantisierung und quantisierter Arithmetik sind:

- Die realisierte Übertragungsfunktion weicht von der idealen Übertragungsfunktion ab. Bei großen Abweichungen können das Toleranzschema und sogar die Stabilitätsbedingung verletzt werden. Ursache ist die Quantisierung.
- Es können Überläufe auftreten, die im Ausgangssignal zu Verzerrungen führen. Bei einem IIR-Filter können diese Überläufe Oszillationen mit großer Amplitude (große Grenzzyklen) zur Folge haben, die das Filter unbrauchbar machen.
- Runden und Abschneiden von Zwischenergebnissen führt zu Rundungsrauschen. Bei einem IIR-Filter können daraus kleine Grenzzyklen, d.h. Oszillationen mit kleinen Amplituden am Ausgang entstehen.

Maßnahmen zur Eliminierung oder Verminderung der genannten Effekte:

- *FIR-Filter statt IIR-Filter verwenden:* FIR-Filter sind weniger sensitiv bezüglich ungenauer Filterkoeffizienten, haben ein kleineres Rundungsrauschen und sind immer stabil und frei von Grenzyklen.
- *Abtastfrequenz verkleinern:* Bei Verkleinerung der Abtastfrequenz werden Digitalfilter weniger empfindlich bezüglich ungenauer Filterkoeffizienten. Eine kleinere Abtastfrequenz bei IIR-Filtern vermindert zudem das Quantisierungsrauschen und die Tendenz zu Grenzyklen.
- *Wortlänge vergrößern:* Eine größere Wortlänge vermindert das Rundungsrauschen und erhöht die Genauigkeit der Übertragungsfunktion.
- *Fließkommaechnen verwenden:* Diese Maßnahme führt zu Digitalfiltern mit genauen Filterkoeffizienten, geringem Quantisierungsrauschen und überlauffreiem Ausgangssignal, erfordert aber kompliziertere Hardware.

6.2.5 Entwurf digitaler Filter mit MATLAB

Unter MATLAB stehen mehrere Möglichkeiten zur Verfügung digitale Filter zu designen. Ein sehr schönes und praktisches Tool ist das **Filter Design and Analysis Tool (FDA-Tool)**. Man kann es über die Eingabe

```
>> fdatool
```

im MATLAB-Command-Window starten.

6.2.5.1 Das Filterdesigntool FDA

Die Oberfläche des Tools ist in Abbildung 6.24 dargestellt.

Beim FDA-Tool handelt es sich um ein graphisches Design-Tool, mit dessen Hilfe man sich den Entwurfsablauf deutlich vereinfachen kann.

Vorgehen beim Filterentwurf mit FDA: Die gewünschten Spezifikationen des Filters lassen sich in der unteren Hälfte des Fensters festlegen:

- Im **Response Type** Fenster stellt man den gewünschten Filter und die Entwurfsmethode ein
- Im Fenster **Filter Order** kann man dem Tool eine feste Ordnung vorgeben, oder den Entwurf mit kleinstmöglicher Ordnung berechnen lassen.
- Das Toleranzschema für den Amplitudengang legt man in den **Frequency Specifications** fest.
- Mit den **Magnitude Specifications** gibt man ein Maß für die maximalen Rippel im Durchlass- und Sperrbereich an, wobei A_{pass} den Durchlassbereich und A_{stop} den Sperrbereich charakterisiert.

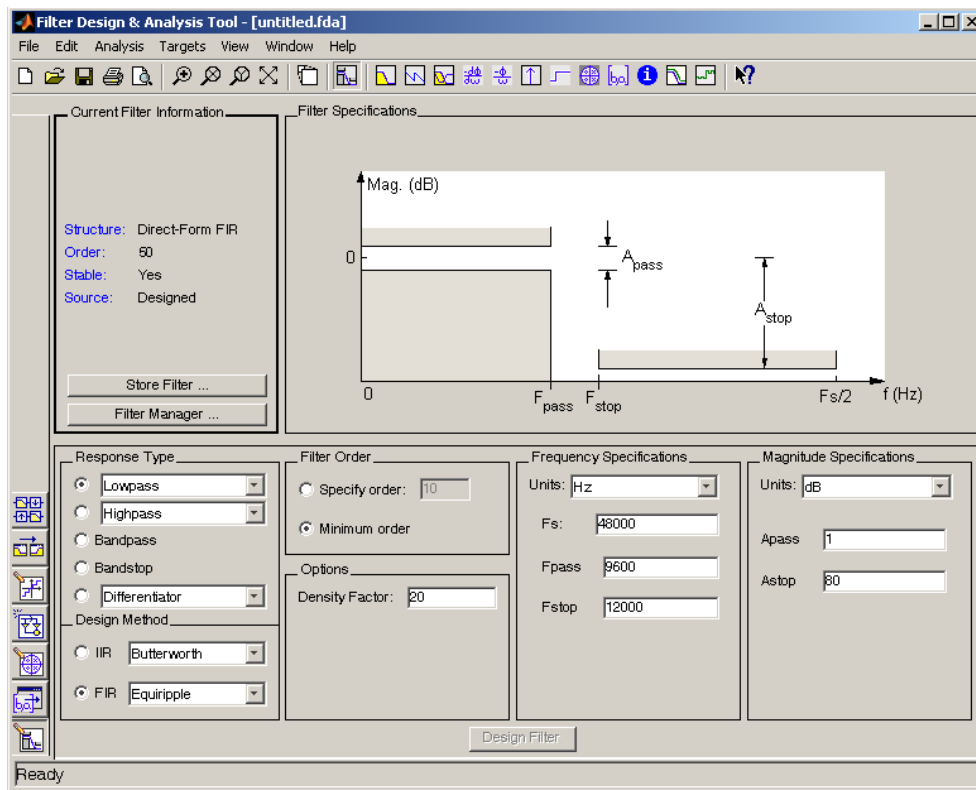


Abbildung 6.24: Graphische Oberfl ches des FDA-Tools

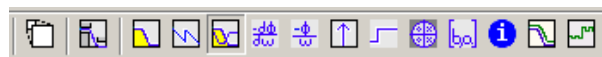


Abbildung 6.25: Iconleiste im FDA-Tool

Sind alle gew nschten Parameter festgelegt, so braucht man nur noch auf die Schaltfl che **Design Filter** am unteren Rand des Fensters klicken und FDA berechnet alle n tigen Koeffizienten.

Mit den in Abb. 6.25 dargestellten Icons in der Men leiste kann man sich jetzt genauere Informationen (wie Phasenverl ufe, Pol-Nullstellen-Diagramme, Filterkoeffizienten, etc.) zum berechneten Filter anzeigen lassen.



Die berechneten Filterkoeffizienten sind im Double-Precision-Floating-Point-Format berechnet. Ben tigt man Filterkoeffizienten in Fixed-Point-Arithmetik, so muss man dies IMMER extra einstellen!

Filterkoeffizienten in Fixed-Point-Darstellung

 ber das in Abb. 6.26 markierte Icon in der linken Symbolleiste l sst sich das Format der zu berechnenden Filterkoeffizienten festlegen.

Portierung der Filterkoeffizienten nach MATLAB: In der Men leiste  ffnet man  ber den Men punkt File → Export die in Abb. 6.27 dargestellte Exportfunktion von

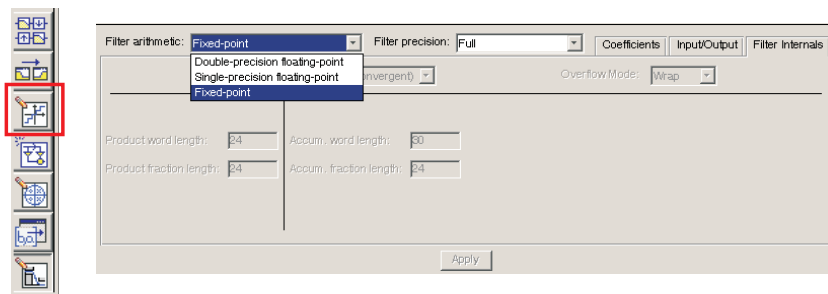


Abbildung 6.26: Fixed-Point Koeffizienten mit FDA-Tool

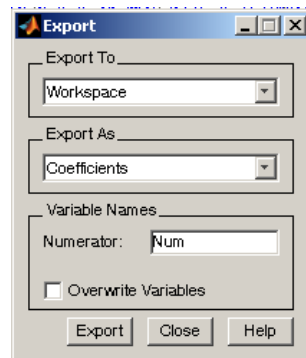


Abbildung 6.27: Export von Filterkoeffizienten

FDA. Hier hat man die Möglichkeit die Koeffizienten entweder direkt im Workspace oder in einem mat-File abzulegen, um später in MATLAB mit ihnen arbeiten zu können.

Filtern eines digitalen Signals: Um die entworfenen Filter in der MATLAB Simulation verwenden zu können, steht ihnen in ihrem Praktikumsverzeichnis die MATLAB-Funktion *filtfir_symm_qa.m* zur Verfügung. Diese Funktion rechnet bereits mit quantisierten Koeffizienten. Nach jedem Rechenschritt werden die Ergebnisse ebenfalls quantisiert, so dass die resultierenden Ergebnisse später zum Vergleich mit der Hardware herangezogen werden können.

6.2.6 MATLAB: Filterdesign

Aufgabe 1: Entwurf von FIR-Filtern mit der Fenstermethode

1. Generieren sie einen Bandpass mit beliebigem Toleranzschema mit Hilfe der Fenstermethode. Welche Unterschiede fallen ihnen bei Verwendung unterschiedlicher Fenster auf. Dokumentieren sie die Ergebnisse, indem sie von jedem Filter einen Ausdruck anfertigen. Welche Grenzfrequenzen verwenden sie für ihr Filter? Speichern sie ihre erzeugten Fenster und Filter im folgenden Verzeichnis:

MATLAB_src\Empfänger\MATLAB_Filterdesign\Aufgabe_01

2. Wie unterscheiden sich die einzelnen Amplitudenfrequenzgänge voneinander? Welchen würden sie bevorzugen?

Aufgabe 2: Optimalmethode

1. Entwerfen sie nun einen Bandpass mit der Optimalmethode. Wie unterscheidet sich das Ergebnis von den Ergebnissen der Fenstermethode? Speichern sie ihren entworfenen Bandpass im Verzeichnis

MATLAB_src\Empfänger\MATLAB_Filterdesign\Aufgabe_02

ab.

2. Gehen sie zu quantisierten Filterkoeffizienten mit 8 Bit Wortlänge über. Vergleichen sie die Amplitudenfrequenzgänge der Filter.

Aufgabe 3: Portierung nach MATLAB und Testen des Filters

1. Exportieren sie ihren Filter nun in den MATLAB-Workspace.
2. Machen sie sich mit der Filterfunktion *filtfir_symm_qa.m* im Verzeichnis

MATLAB_src\Empfänger\MATLAB_Filterdesign\Aufgabe_03

vertraut.

3. Erzeugen sie mit ihrem in Versuch 5 aufgebauten Modulator ein beliebiges FSK Signal und wenden sie den entworfenen Filter darauf an. Funktioniert ihr Design?

Was sehen sie?

Ein Tipp: um etwas sehen zu können sollte eine der beiden FSK-Frequenzen im Durchlassbereich des Bandpasses liegen!

6.3 Versuch 6: Simulation der Empfängerhardware in MATLAB

Aufgabe 1: Bandpassdesign

1. Erzeugen sie zwei Bandpässe, deren Durchlassbereich jeweils im Bereich ihrer beiden (vom Betreuer festgelegten) Signalfrequenzen liegt und legen sie ihre Entwürfe im Verzeichnis

MATLAB_src\Empfänger\EmpfängerhardwareMATLAB
\Aufgabe_01_Bandpassdesign

ab. Die Bandbreite sollte mindestens ein kHz betragen. Notieren sie die festgelegten Eckfrequenzen:

2. Welche Ordnung haben ihre Bandpässe? Bestimmen sie die Anzahl der Koeffizienten und geben sie die Zahl der notwendigen Multiplikationen an.

3. Ist die Ordnung der Filter im Hinblick auf die Hardware vertretbar? Sollten sie zu dem Ergebnis kommen, dass dies nicht der Fall ist, so passen sie ihre Filter den Gegebenheiten an.

Aufgabe 2: Simulation der Filter

1. Filtern sie ihr Sendesignal nun mit ihren generierten Bandpässen.
2. Stellen sie das Ergebnis graphisch dar und legen sie die Ergebnisse im folgenden Verzeichnis ab:

MATLAB_src\Empfänger\EmpfängerhardwareMATLAB
\Aufgabe_02_Filtersimulation

Wenn alles passt sollten ihre Signale ähnlich den auf Seite 60 Abb. 6.3 dargestellten sein.

3. Dokumentieren sie ihre Ergebnisse.

Aufgabe 3: Quadrierung der Einzelsignale

1. Quadrieren sie nun ihre beiden Bandpassgefilterten Signale
2. Stellen sie das Ergebnis wieder graphisch dar.

Aufgabe 4: Tiefpass-Design

1. Erzeugen sie ein Tiefpassfilter, das die gewünschte Funktion erfüllt. Wie würden sie die Durchlassfrequenz einstellen?

2. Welche Ordnung hat ihr Tiefpass? Bestimmen sie die Anzahl der Koeffizienten und geben sie die Zahl der notwendigen Multiplikationen an.

3. Ist die Ordnung des Filters im Hinblick auf die Hardware vertretbar? Sollten sie zu dem Ergebnis kommen, das dies nicht der Fall ist, so passen sie ihre Filter den Gegebenheiten an.

Speichern sie ihr Filter im Ordner

MATLAB_src\Empfänger\EmpfängerhardwareMATLAB
\Aufgabe_04_Tiefpass-Design

Aufgabe 5: Tiefpass-Simulation

1. Bauen sie ihr Tiefpassfilter, wie in Abb. 6.1 auf Seite 59 dargestellt, in ihr bestehendes System ein.
2. Simulieren sie nun die Funktion und stellen sie die Ausgangssignale graphisch dar. Stimmt ihr Ergebnis?

Aufgabe 6: Signalrückgewinnung

1. Implementieren sie eine Funktion, die aus den beiden tiefpassgefilterten Signalen das ursprüngliche Sendesignal zurückgewinnen kann. Wie würden sie diese aufbauen?

2. Erweitern sie ihr Programm dahingehend, dass es die auftretenden Bitfehler erfasst und im Command-Window ausgibt. Wie würden sie diese Funktion implementieren?

Aufgabe 7: Vereinfachung der Schaltung

1. Ist es möglich, die Schaltung mit einem einzigen Tiefpass aufzubauen und wenn ja, wo würden sie diesen in ihr System einbinden? Begründen sie ihre Antwort

2. Verifizieren sie ihre Behauptung in der Simulation und legen sie das Skript im Verzeichnis

MATLAB_src\Empfänger\EmpfängerhardwareMATLAB
\Aufgabe_07_Vereinfachung

ab.

6.4 Versuch 7: VHDL: Bandpass

Das eingesetzte FPGA enthält laut Datenblatt [6] 7 so genannte sysDSP-Blöcke, die in [7] (Kapitel 4: sysDSP Usage Guide) genauer beschrieben werden. Jeder dieser Blöcke kann verschiedene Operationen durchführen. Machen Sie sich mit diesen speziellen Blöcken vertraut.

Das Filter soll der bereits bekannten Linearphasenstruktur (Abb. 6.28) implementiert werden, da durch die Zusammenfassung gleicher Koeffizienten bereits sehr viel Hardware eingespart werden kann.

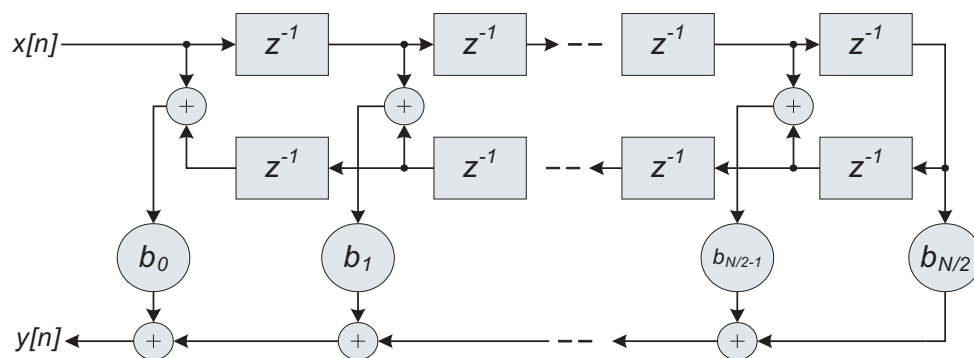


Abbildung 6.28: Linearphasenstruktur eines FIR-Filters

Hier wird auch deutlich, warum die Multiplizierer eine größere Wortbreite aufweisen, als die üblicherweise verwendeten Wortbreiten (8,16,32). Durch die Addition zweier Zahlen vor dem Multiplizierer, erhöht sich die Wortbreite dieser Zahl um ein Bit, wenn man gleiche Genauigkeit voraussetzt (vgl. 5.3.4). Bei der Addition zweier Zahlen beispielsweise die nahe an den Grenzen des darstellbaren Bereichs liegen (z.B. $0,9 + 0,8 = 1,7$) wird der Wertebereich der Eingangsvariablen $[-1; 1[$ verlassen. Indem ein weiteres Bit vor dem Komma zur Verfügung gestellt wird, das bei Addieren „Carry-Bit“ genannt wird, umgeht man diese Einschränkung und kann dem Multiplizierer eine Zahl mit möglichst hoher Genauigkeit zuführen. Anschließend muss die Genauigkeit des Ergebnisses beschnitten werden, um eine Weiterverarbeitung zu ermöglichen.

Aufgabe 1: Programmieren sie das Filter mit der Ordnung N mit Hilfe von $\frac{N}{2}$ Multiplizierern aus dem sysDSP-Block in `07_Bandpass\bandpass.vhd` und testen sie dies mit dem verschiedenen Frequenzen in und ausserhalb des Durchlassbereichs.

6.5 Versuch 8: VHDL: Optimierung der Filterhardware

Nach der Hardwaremäßig aufwändigen Implementierung des Filters in Linearphasenstruktur sollen im Weiteren Konzepte für ein effizienteres Design gefunden werden. Einer der üblichen Ansätze besteht in der mehrfachen Verwendung von Hardware, hier speziell der Multiplizierer, da es sich um die am stärksten begrenzte Ressource handelt.

Wie erreichen sie eine mehrfache Verwendung der Hardware, beachten sie speziell die Möglichkeiten der sysDSP-Blöcke?

Zeichnen sie nun schematisch den Aufbau ihres optimierten Filters, speziell die Komponenten rings um die gewählte sysDSP-Struktur. Lassen sie ihre Lösung von einem Betreuer verifizieren.

Welches zusätzliche Signal benötigen sie, um diese Funktionsweise implementieren zu können? Welche Eigenschaften muss dieses besitzen?

Aufgabe 1: Implementieren sie ihre neue Filterstruktur in `08_FilterOpt\bandpass.vhd`. Diese soll sich nach außen ähnlich verhalten wie die vorherige Struktur, also mit nur geringen Änderungen der äußeren Beschaltung und der Ports einsetzbar sein.

Welche Auswirkungen wird diese Änderungen nach sich ziehen, was sie bei größeren Designs nie aus dem Auge verlieren sollten? Hinweis: Bei der Technologie handelt es sich um CMOS.

6.6 Versuch 9: VHDL: Demodulator

Im Folgenden soll wie auf Seite 60 beschrieben, die Hüllkurvendemodulation in VHDL durchgeführt werden. Hierzu müssen sie die Ausgangssignale der beiden Bandpässe quadrieren.

Aufgabe 1: Implementieren sie die Quadrierung so, dass die Hardware wieder mehrfach genutzt wird, aber achten Sie darauf, dass nur die benötigten Berechnungen durchgeführt werden und in den übrigen Zyklen keine unnötigen Schaltvorgänge auftreten, da dies nur zu einer erhöhten Stromaufnahme und somit zu unnötiger Verlustleistung führt.

Verwenden sie die Struktur aus Abbildung 6.29

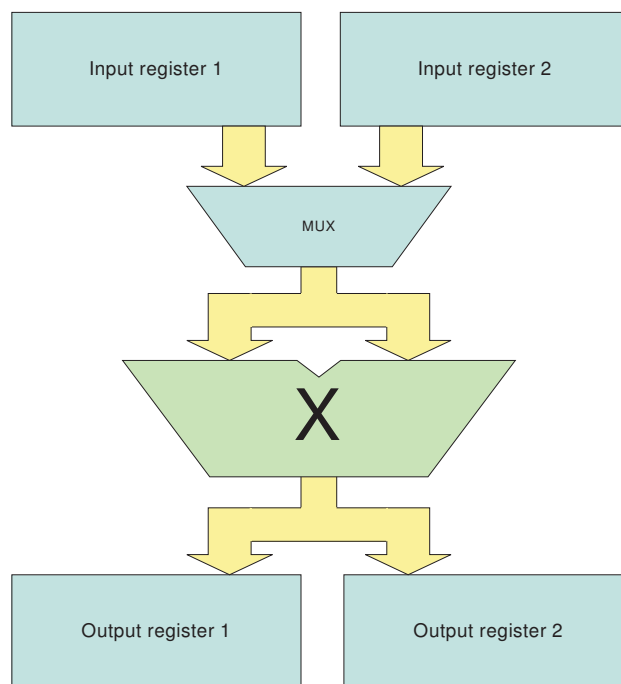


Abbildung 6.29: Quadrierungsschema

6.7 Versuch 10: VHDL: Tiefpass

Aufgabe 1: Filtern sie das quadrierte Signal mit einem Tiefpass. Dazu verwenden sie die Koeffizienten, die sie in der vorherigen MATLAB-Übung berechnet haben. Falls sie bei der Programmierung auf eine Effizienz bedacht waren, haben sie jetzt weniger Arbeit, falls nicht, versuchen sie dies jetzt. Hinweis: Es ist ihnen natürlich jederzeit erlaubt, eigene Module zu erstellen.

6.8 Versuch 11: VHDL: Signalregeneration

Um das übertragene Bit rekonstruieren zu können, muss aufgrund der vorliegenden Information eine Entscheidung getroffen werden. Hierzu ist es nötig, beide Signale miteinander zu vergleichen. In unserem Fall geschieht dies über eine Subtraktion der beiden nachbearbeiteten Signale. Ist das Ergebnis negativ, so handelt es sich um eine 0, ist es positiv, um eine 1. Zur Verbesserung der Robustheit sollte eine kleine Hysterese eingebaut werden.

Aufgabe 1: Implementieren Sie die beschriebene Funktion und testen sie diese in der Simulation und der Hardware. Ist dies erfolgreich, haben sie bereits alle Komponenten.

7 Gesamtsystem

7.1 Versuch 12: Simulation einer nichtidealen Übertragung

Nachdem die komplette Hardware bereits im Versuch 6 fertiggestellt wurde, werden wir nun noch ein paar Besonderheiten betrachten. In den Kapiteln 5.3.4 und 6.2.4 sind wir auf die Zahlendarstellung in digitalen Systemen und die hierdurch entstehenden Effekte eingegangen. Dies soll nun noch in der Simulationsstrecke berücksichtigt werden.

Aufgabe 1: Fixed-Point-Format

1. Wie bereits an anderer Stelle dargestellt wurde, arbeitet MATLAB standartmäßig mit Zahlen im Double-Precision-Floating-Point Format. Die Implementierung auf dem FPGA der Testplatine allerdings werden sie mit einer Fixed-Point-Darstellung durchführen müssen. An welchen Stellen werden voraussichtlich ungewünschte Effekte auftreten?

2. Um den Arbeitsaufwand nicht unnötig ausufern zu lassen werden wir im weiteren Verlauf mit einer Quasi-Quantisierung arbeiten. Die Funktion *quant2c.m* im Verzeichnis

MATLAB_src\Gesamtsystem

quantisiert eine beliebige Zahl auf eine gewünschte Wortlänge. Stellen sie mit ihrer Hilfe ihre Simulation so um, dass nach jeder Berechnung die Ergebnisse neu quantisiert werden.

Aufgabe 2: Kanalmodellierung

1. Der bisher angenommene ideale Übertragungs-Kanal wird in Praxis nicht realisierbar sein. Simulieren sie daher mit Hilfe der Funktion *AWGNchannel.m* die Arbeitsweise ihres Systems unter realistischeren Bedingungen.
2. Da es während des Praktikums nicht vorkommen wird, dass sie als einzige Gruppe das Übertragungsmedium (die Luft) nutzen werden, sollen nun noch die parallelen Übertragungen weiterer Gruppen simuliert werden. Machen sie sich hierzu mit der Funktion *Nachbarkanaele.m* vertraut und bauen sie diese an passender Stelle in ihre Simulation mit ein. Wie stark wirkt sich diese Übertragung auf ihre gemessenen Bit-Fehler aus?

7.2 Versuch 13: Loop-Test

7.2.1 Konzept

Um die Zusammenarbeit von Modulator und Demodulator zu testen, wird das gesamte System im FPGA implementieren und der Ausgang des SPATE mit dessen Eingang verbunden. Auf diese Weise wird eine Selbsttest der Hardware und des Programmes ermöglicht.

7.2.2 VHDL: Beschreibung der Hardware

Alle Module werden in geeigneter Weise in das Toplevel integriert. Entscheiden sie selbst, welche Möglichkeiten der Signalerzeugung und Signalisierung sie für die Verifikation der Schaltung einsetzen wollen. Der Anschluss des Ports RS232_RX als Eingang und RS232_TX als Ausgang ist für den nachfolgenden Versuch nötig.

7.2.3 TEST: Praxis

Aufgabe 1: Funktionstest Verbinden sie den Line-In-Eingang und den Line-Out-Ausgang der Schaltung mit Hilfe einer Leitung, die sie von ihrem Betreuer erhalten und überprüfen sie die Funktion der Hardware.

Aufgabe 2: Serielles Echo Anschließend verbinden sie das serielle Kabel mit dem PC und dem SPATES und starten HyperTerm mit dem Profil *ADSP-Loop-Settings.ht*, das in Ihrem Praktikumsverzeichnis liegt, indem sie das Profil im Explorer mittels Doppelklick starten. Das Hyperterm-Programm wird dabei automatisch gestartet. Anschließend können sie die Hardware auf die serielle Schnittstelle umschalten. Sobald sie im Programmfenster von Hyperterm eine Eingabe tätigen, sollten die Zeichen auf dem Bildschirm erscheinen. Ist dies nicht der Fall, so überprüfen Sie zuerst die Kabelverbindungen, bevor Sie sich an die Fehlersuche im Code machen. Hierzu kann Ihnen Ihr Betreuer auch einen Loop-Stecker für die serielle Schnittstelle des PC geben, der gleiches bewirken sollte wie die Hardware in diesem Versuch. Zumindest kann eine Fehlkonfiguration des Terminal-Programms und der Schnittstelle schnell ausgeschlossen werden.

7.3 Versuch 14: Chat-Session

7.3.1 Konzept

Es soll eine bidirektionale Verbindung zwischen 2 entfernten PCs mit Hilfe des SPATES hergestellt werden. Das SPATES arbeitet als MODEM¹ und Zugriffspunkt auf den physikalischen Übertragungskanal. Da ein akustischer Kanal wie im Praktikumsraum mit den vielen auftretenden Reflexionen und Störungen sich negativ auf die Kanalkapazität auswirkt, können keine hohen Datenübertragungsraten erwartet werden.

7.3.2 VHDL: Beschreibung der Hardware

Hat das vorherige Experiment gut funktioniert, können wir durch einfache Anpassung der Frequenzen von Sender und Empfänger die Veränderungen am Quellcode abschließen und erneut implementieren. Auch eine Simulation dürfte hier überflüssig sein.

7.3.3 TEST: Praxis

Verbinden Sie zunächst Ihr SPATES über gekreuzte Signalwege, jeweils Eingang auf Ausgang, mit dem SPATES der Nachbargruppe über die Line-In- und Line-Out-Buchsen.






Bitte nicht die Speaker- oder MIC-Buchsen verwenden, da sonst der Eingang zerstört werden könnte!

Aufgabe 1 Starten Sie nun eine Terminal-Session mit Ihrer Nachbargruppe und beobachten Sie, ob bei der Übertragung Fehler auftreten.

Aufgabe 2 Verbinden Sie nun den Lausprecher mit dem SPATES und testen Sie die Übertragung durch die Luft. Beobachten Sie auch hier die Häufigkeit von Fehlern.

Aufgabe 3 Erhöhen Sie nun schrittweise die Übertragungsgeschwindigkeit in den Terminal-Einstellungen. Dafür sind folgende Schritte auf beiden PCs nötig:

1. Verbindung beenden 
2. Eigenschaften 
3. Konfigurieren der seriellen Schnittstelle s. Abb. 7.1
4. Übertragungsgeschwindigkeit einstellen s. Abb. 7.2
5. Die Dialoge verlassen und Verbindung herstellen 

¹MODEM: Modulator/Demodulator



Abbildung 7.1: HyperTerm-Eigenschaften

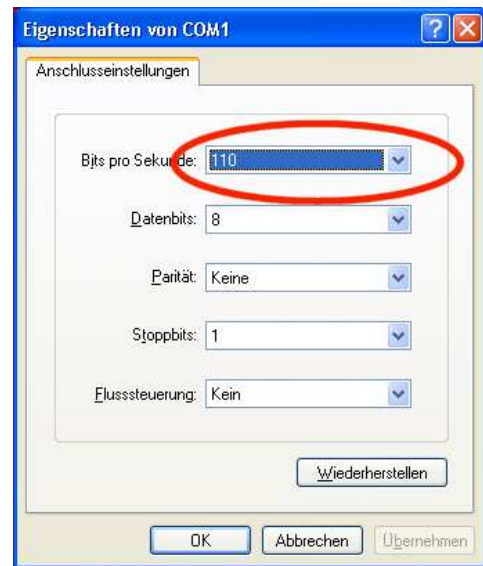


Abbildung 7.2: Einstellen der Übertragungsrate

Bis zu welcher maximalen Übertragungsgeschwindigkeit ist eine Datenübertragung mit tolerierbarer Fehlerrate möglich? Welche 4 Modulationsfrequenzen wurden bei Ihnen zugeteilt?

Welche einfachen Änderungen können Sie durchführen, um die Übertragungsgeschwindigkeit zu steigern?

Welche grundlegenden Änderungen wären nötig, um im gleichen Frequenzbereich und der gleichen Übertragungsstrecke wesentlich höhere Übertragungsraten zu realisieren?

Sollte noch etwas Zeit sein, können sie versuchen, eine gemeinsame Modulationsfrequenz für alle SPATES zu verwenden.

Teil III

Anhang

A MATLAB Tutorial

A.1 Bedeutung von MATLAB

“It is probably fair to say that one of the three or four most important developments in numerical computation in the past decade has been the emergence of MATLAB as the preferred language of tens of thousands of leading scientists and engineers.”¹

Bei MATLAB handelt es sich um ein leistungsfähiges Softwaresystem, mit dem sich alle Arten von Berechnungen durchführen lassen. Der Name MATLAB kommt von MATrix-LABoratory. Bei der Namensgebung spielten zwei Überlegungen eine Rolle:

1. Grundelemente der Berechnungen unter MATLAB sind Matrizen und deren Manipulation, die in numerischen Verfahren optimal eingesetzt werden können.
2. Laboratory bezieht sich auf den Gedanken der möglichen Entwicklung und Erweiterung der Softwareumgebung.

Mit MATLAB wird dem Ingenieur ein interaktives, matrixorientiertes Softwaresystem an die Hand gegeben, mit dem Probleme und Lösungen in der vertrauten mathematischen Schreibweise dargestellt werden können.

Typische Anwendungen von MATLAB sind:

- Numerische Berechnungen aller Art,
- Die Entwicklung von Algorithmen (zum Beispiel in der Signalverarbeitung),
- Die Modellierung, Simulation und Entwicklung von Prototypen technischer und wirtschaftlicher Probleme,
- Die Analyse, Auswertung sowie graphische Darstellung von Datenmengen,
- Visualisierungen,
- Wissenschaftliche und technische Darstellungen und
- Applikationsentwicklung mit Aufbau einer graphischen Benutzerschnittstelle.

In den siebziger Jahren wurde in den USA eine intensive Aktivität zur Entwicklung qualitativ hochwertiger Software gestartet: das NATS²-Projekt. 1976 lag als Ergebnis dieser Bemühungen das Softwarepaket Eispack zur Lösung algebraischer Eigenwertprobleme vor (Genauerer siehe [8]). Im Jahr 1975 begannen die Arbeiten an einem effizienten und portablen Softwarepaket zur Lösung linearer Gleichungssysteme. Das Ergebnis war das

¹Lloyd N.Trefethen, Professor of Numerical Analysis, Oxford University, 1997

²was heisst NATS denn ausgeschrieben?

Softwarepaket Linpack (vgl. [3]). Linpack und Eispack gewährleisteten lange Zeit das zuverlässige und portable Lösen von Problemen der Linearen Algebra. Um diese beiden Pakete leichter handhabbar zu machen, wurde MATLAB entwickelt. Damit bestand auch die Möglichkeit, ausgereifte Software effizient in der Lehre - zunächst in der (Numerischen) Linearen Algebra, später auch in vielen anderen Bereichen - einzusetzen.

MATLAB vereinfacht dem Ingenieur und Naturwissenschaftler die Lösung von Problemen aus Technik und Natur erheblich. Der Umfang von MATLAB ist in den letzten Jahren stark angestiegen. Hierzu trägt vor allem der Umstand bei, dass relativ einfach Toolboxes³ nachgerüstet werden können. Informationen zu den aktuellen Versionen von MATLAB können auf der Herstellerhomepage⁴ eingeholt werden.

Die drei wichtigsten Funktionskomponenten von MATLAB sind:

- Berechnung
- Programmierung und
- Visualisierung

Berechnung: Durch die Programmsammlung von MATLAB bleibt es dem Nutzer in vielen Fällen erspart, Standardalgorithmen neu programmieren zu müssen. Er kann auf fertige und getestete Programme zurückgreifen und darauf aufbauend eigene Algorithmen realisieren.

Visualisierung: Mittels der vielfältigen Visualisierungsmöglichkeiten kann der Benutzer berechnete Ergebnisse anschaulich darstellen.

Programmierung: MATLAB bietet dem Anwender Möglichkeiten, seine Funktionalität durch eigene Programme beliebig zu erweitern. Dies kann mittels eigener MATLAB-Programme (m-files), oder durch das Einbinden von Codes einer anderen höheren Programmiersprache erreicht werden.

Auf Grund seiner vielfältigen Anwendungsmöglichkeiten und seiner intuitiven Syntax erfreut sich MATLAB heute in der Industrie großer Beliebtheit in den Bereichen Forschung, Entwicklung, Datenauswertung und Visualisierung.

Das folgende Tutorial soll lediglich einen Überblick über die Möglichkeiten von MATLAB liefern. Detailliertere Beschreibungen finden sie in der einschlägigen Literatur (zum Beispiel [4]) sowie in der Online-Hilfe (vgl. Kapitel A.3.1) von MATLAB selbst.

³Funktionssammlungen zu bestimmten Anwendungsgebieten

⁴<http://www.mathworks.de>

A.2 Die MATLAB-Umgebung

Nach dem Starten des Programms durch einen Doppelklick auf das MATLAB Desktop-Icon



Abbildung A.1: MATLAB-Desktop-ICON

wird die MATLAB-Umgebung (vgl. Abb. A.2) geöffnet.

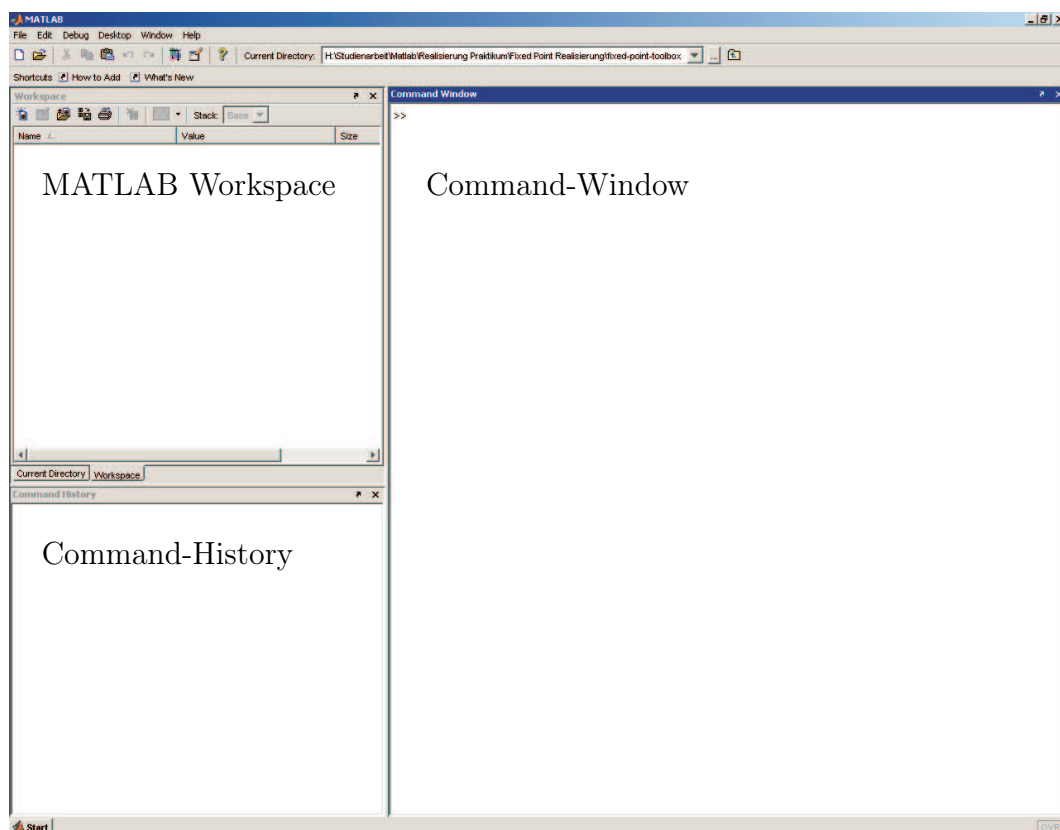


Abbildung A.2: Eine jungfräuliche MATLAB-Benutzeroberfläche

Im linken oberen Fenster, dem *Workspace*, sind alle zur Zeit im Speicher liegenden Variablen zu sehen.

Das große Fenster rechts ist das *Command Window*. Hier erfolgt die direkte Befehls-Eingabe sowie die Ausgabe der Ergebnisse durch MATLAB.

Das kleine Fenster unten links enthält die *Command History*. In dieser werden alle eingegebenen Kommandos gespeichert und können bei Bedarf (Mittels Befehl oder Mausklick) wieder aufgerufen werden. Alle genannten Fenster sollten zum aktuellen Zeitpunkt leer sein! An einem einfachen Beispiel soll die Funktion der drei Hauptfenster noch einmal verdeutlicht werden:

Bewegen sie den Mauszeiger über das Command Window und klicken sie einmal hinein. Jetzt sollte ein blinkender Cursor in der ersten Zeile des Fensters zu sehen sein. Geben sie nun folgenden einfachen MATLAB-Code ein:

```
>> a = 1;  
>> b = 2;  
>> c = a / b;  
>> c
```

Jetzt sollte MATLAB das Ergebnis `c = 0.5` liefern!

Hier kann man gleich eine weitere Eigenschaft der MATLAB-Eingabe erkennen: Wird ein Befehl mit einem Strichpunkt abgeschossen, so liefert MATLAB keine Ausgabe im Command-Window. Daher wurde auch erst nach der Eingabe von `c` das Ergebnis der Berechnung ausgegeben. Außerdem kann man jetzt die Variablen `a`, `b` und `c` im Workspace-Bereich finden. In der Command-History können alle bisher vorgenommenen Eingaben nachvollzogen und auch erneut aufgerufen werden. Den hier beschriebenen Eingabemodus bezeichnet man auch als *Direkten Modus*.

A.3 MATLAB im direkten Modus

A.3.1 Die MATLAB-Hilfe

Zunächst soll Ihnen die ausgezeichnete Online-Hilfe der MATLAB-Umgebung ans Herz gelegt werden. Bei einem Mausklick auf das gelbe Fragezeichen in der Symbolleiste öffnet sich das MATLAB-Hilfefenster (vgl. Abb.A.3 auf Seite 103). Hier haben sie die Möglichkeit sich über grundlegende Informationen zu MATLAB und seinen Toolboxes zu informieren oder sich mittels der Suchfunktion bzw. des Indexverzeichnisses gezielt Informationen zu bestimmten Themengebieten anzeigen zu lassen.

Eine weitere Möglichkeit schnell Dokumentationen zu Funktionen aufzurufen ist der `help` Befehl. Wenn sie zum Beispiel

```
>> help sin
```

im MATLAB-Command-Window eingeben wird ihnen MATLAB eine Kurzbeschreibung der Funktion `sin.m` und ihrer möglichen Parameter ausgeben! Zusätzlich bekommen sie am Ende der Hilfe noch eine Link auf die entsprechende Seite der MATLAB-Dokumentation, in der noch einmal genauer, meist mit Beispielen, auf den Befehl eingegangen wird.

Ist der genaue Name der gesuchten Funktion nicht bekannt, so kann man sich mit dem Befehl `lookfor` *Zeichenkette* helfen. MATLAB listet nun alle Funktionen die die gegebene Zeichenkette enthalten im Command-Window auf. Die Suche kann mit Strg+C abgebrochen werden.

A.3.2 Elementare Funktionen

Die wichtigsten Funktionen erhält man mit

```
>> help elfun
```

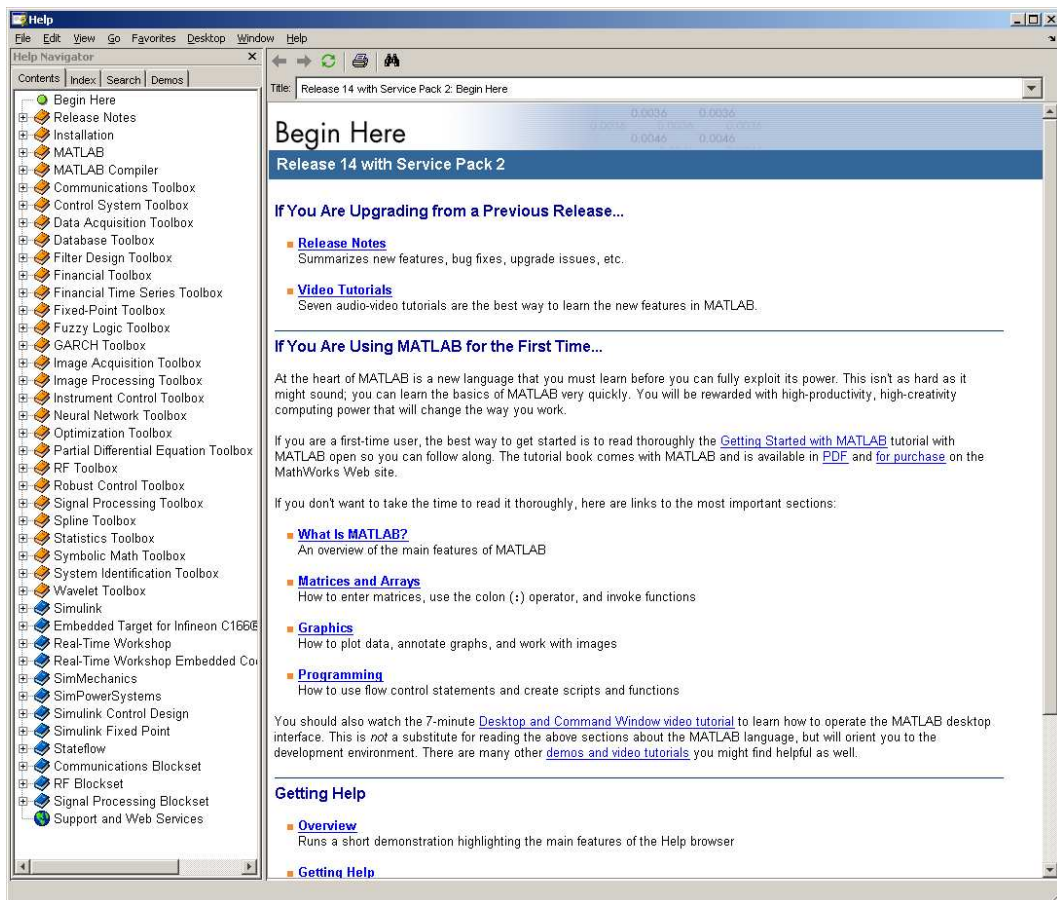


Abbildung A.3: Startseite der MATLAB Hilfe

aufgelistet. Wie schon am Beispiel der Sinusfunktion (vgl. Kapitel A.3.1) erläutert, kann man sich auch hier detailliertere Angaben zu den einzelnen Funktionen über die blau unterlegten Links anzeigen lassen.

Die Syntax einer MATLAB-Funktion lautet:

$$\text{NameDerAusgabe} = \text{MatlabFunktionsname}(\text{Eingabewert})$$

Beispiel:

Die Eulersche Zahl e lässt sich mit `x = exp(1)` berechnen. Lässt man den Namen der Ausgabevariable (und das Gleichheitszeichen) weg, so legt MATLAB das Ergebnis in der Variable `ans` ab.

A.3.3 Umgang mit dem MATLAB-Workspace

Nachdem wir nun die ersten Ergebnisse berechnet haben noch eine kurze Einführung zu den Möglichkeiten des *Workspace*:

Mittels der Befehle `save` und `load` kann man den Inhalt des Workspace entweder abschnittsweise oder komplett in eine Datei sichern bzw. auch wieder einlesen. Die Eingabe

```
>> save variables.mat
```

speichert den kompletten Workspaceinhalt in der Datei `variables.mat`. Will man nur einzelne Variablen speichern, erreicht man dies, indem man die entsprechenden Variablen mit Leerzeichen separiert an den Befehl anhängt:

```
>> x=1, y=2, z=x+y;  
>> save variables.mat x y
```

Hier werden nur die Variablen `x` und `y` im File abgelegt. Löschen lässt sich der Inhalt des Workspace mittels des Befehls `clear`, wobei man auch hier wieder das Löschen einzelner Variablen durch einfaches Anhängen der Variablenamen an den Befehl erreichen kann.

Zur Wiederherstellung von in mat-Files gesicherten Variablen dient der `load`-Befehl. Dieser lädt die Inhalte der Dateien wieder in den Workspace.



Bereits vorhandene Variablen mit gleichen Namen werden ohne Warnung überschrieben!

A.3.4 Matrizen und Vektoren

Die wohl wichtigste Fähigkeit der MATLAB-Tools ist die Möglichkeit direkt mit Matrizen und Vektoren zu arbeiten. Auf diese Weise lassen sich Ausdrücke, die in höheren Programmiersprachen lediglich mittels aufwändiger Schleifenkonstrukte zu verwirklichen sind relativ einfach beschreiben.

A.3.4.1 Matrizen

Eingabe und Handhabung von Matrizen Matricelemente werden in eckigen Klammern zeilenweise eingegeben. Elemente einer Zeile werden durch Leerzeichen oder Komma getrennt, das Ende einer Zeile wird durch einen Strichpunkt abgeschlossen. So erzeugt die Eingabe

```
>> A = [1 2 3;4,5,6]
```

die Matrix

```
A=  
 1 2 3  
 4 5 6
```

Einzelne Elemente können durch Angabe ihrer Zeilen- und Spaltenposition direkt angesprochen werden



In MATLAB beginnt die Indizierung im Gegensatz zu den meisten anderen Programmiersprachen mit Eins anstatt mit Null!!!

```
>> x = A(1,2)  
x =  
 2
```


In Tabelle A.1 sind weitere Möglichkeiten zur geschickten Handhabung von Matrizen dargestellt. Der Doppelpunkt dient als Wildcard und kann gelesen werden wie *alle*.

Operation	Befehl	Beispiel	Ergebnis
Zeile r herausgreifen	<code>A(r, :)</code>	<code>A(1, :)</code>	$\begin{bmatrix} 1 & 0 & 3 \end{bmatrix}$
Spalte s herausgreifen	<code>A(:, s)</code>	<code>A(:, 2)</code>	$\begin{bmatrix} 2 \\ 5 \end{bmatrix}$
C neben A	<code>[A,C]</code>	<code>[A,C]</code>	$\begin{bmatrix} 1 & 0 & 3 & 0 & 1 \\ 4 & 5 & 6 & 2 & 3 \end{bmatrix}$
1. Zeile von A unter A	<code>[A;A(1,:)]</code>	<code>[A;A(1,:)]</code>	$\begin{bmatrix} 1 & 0 & 3 \\ 4 & 5 & 6 \\ 1 & 0 & 3 \end{bmatrix}$

Tabelle A.1: Weitere Matrizenoperationen

Standard-Matrizen lassen sich mit den in Tabelle A.2 angeführten Befehlen erzeugen.

Name der Matrix	Befehl	Beispiel	Ergebnis
Nullmatrix	<code>zeros(m,n)</code>	<code>zeros(2,3)</code>	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$
Einsmatrix	<code>ones(m,n)</code>	<code>ones(3,2)</code>	$\begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix}$
Einheitsmatrix	<code>eye(n)</code>	<code>eye(2)</code>	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
Zufallsmatrix	<code>B=rand(m,n)</code>	<code>B=rand(2,3)</code>	Matrix vom Typ(2,3) mit Zufallszahlen aus $[0,1]$

Tabelle A.2: Standardmatrizen

Rechnen mit Matrizen Matrizen können addiert, miteinander multipliziert, potenziert und mit einem Skalar multipliziert werden. Bei der Addition müssen die Typen der beteiligten Matrizen übereinstimmen, bei der Multiplikation muss die Spaltenanzahl der ersten Matrix mit der Zeilenanzahl der zweiten Matrix übereinstimmen und beim Potenzieren mit einer natürlichen Zahl muss die Matrix quadratisch sein. Werden diese Bedingungen nicht erfüllt meldet sich MATLAB wie im folgenden Beispiel recht schnell mit einer Fehlermeldung:

```
>> A = [1 2 3;4 5 6], C = [1 2;3 4];
>> addAC = A + C
```

```
??? Error using --> plus
Matrix dimensions must agree.
```

Auf eine Besonderheit bezüglich der Addition soll noch hingewiesen werden:

Der Befehl

```
>> Aplus2 = A + 2
```

liefert das Ergebnis

```
Aplus2 =  
  3 4 5  
  6 7 8
```

obwohl die Matrix A vom Typ (2,3) und die Zahl 2 (in MATLAB als Matrix vom Typ (1,1) aufgefasst) eigentlich nicht addiert werden können. Intern wird die Matrix (2) aber aufgebläht zur Matrix $\begin{pmatrix} 2 & 2 & 2 \\ 2 & 2 & 2 \end{pmatrix}$ vom Typ (2,3), die nun zu A addiert werden kann. Diese Anpassung des Typs wird immer dann vorgenommen, wenn eine Zahl zu einer Matrix addiert werden soll.

Punktoperationen In vielen Fällen wären komponentenweise Operationen mit Matrizen (wie sie ja bei der Addition üblich ist) wünschenswert. Dies kann bei MATLAB über Punktoperationen erreicht werden. Am Beispiel der Multiplikation soll dies kurz erläutert werden:

```
>> A = [1 2 3;4 5 6], B = [2 1 0;1 0 2];  
>> G = A .* B  
G=  
  2 2  0  
  4 0 12
```

A.3.4.2 Vektoren

Vektoren werden in MATLAB als einzeilige Matrizen gehandhabt. Auf diese Weise wird ein Zeilenvektor mit 3 Elementen als Matrix vom Typ (3,1) dargestellt



MATLAB unterscheidet streng zwischen Zeilen- und Spaltenvektoren. Daher immer die Rechenregeln für Matrizen beachten!

Ein kurzes Beispiel soll den Umgang mit Vektoren erläutern:

```
>> ZeilenV = [1 3 3]  
ZeilenV =  
  1 3 3  
>> SpaltenV = [3;2;0]  
SpaltenV =  
  3  
  2  
  0
```

Das Skalarprodukt kann mittels

```
>> SkalProd = ZeilenV * SpaltenV  
SkalProd =  
  9
```

berechnet werden, wohingegen

```
>> Matrix33 = SpaltenV * ZeilenV
```

eine Matrix vom Typ (3,3) ergibt.

Relativ einfach lassen sich auch Vektoren mit vielen (äquidistant verteilten) Elementen erstellen. Diese können mit der Befehlsstruktur

» *Bezeichner = Startwert : Schrittweite : Endwert;*

erzeugt werden. Derartige Zeilenvektoren können die mathematische Berechnung von Funktionen besonders vereinfachen: Indem man die entsprechende Funktion mittels einer Punktoperation auf den Zeilenvektor anwendet, erhält man für jedes einzelne Vektorelement ein Ergebnis:

```
>> x = -5 : 1 : 5;  
>> y = x.^2;  
>> plot(x,y)
```

Mittels der `plot`-Funktion⁵ kann man sich nun den dargestellten Ausschnitt der hier berechneten Normalparabel darstellen lassen.

⁵vgl. Kapitel A.4.2

A.4 Programmieren in MATLAB

In den bisherigen Kapiteln wurden meist mehrere Kommandos nacheinander direkt im Command-Window ausgeführt. Beendet man eine MATLAB-Sitzung, sind alle Kommandos verloren. Es wäre also wünschenswert, alle Kommandos in eine Datei schreiben zu können, und die Befehlseingabe auf diese Weise zu automatisieren. Wie man solche Dateien - sogenannten *m-Files* - nutzen kann wird im folgenden Abschnitt behandelt.

A.4.1 MATLAB-Skripte

Vorteile der Programmierung mit MATLAB-Skripten:

Automatisierung Mittels M-Files können MATLAB-Algorithmen ziemlich einfach automatisiert werden. Standardabläufe können nach einmaliger Programmierung schnell wieder aufgerufen werden.

Debugging Durch einfache Änderungen der Inhalte des m-Files kann man schnell ein anderes Verhalten erreichen, oder auch eventuelle Fehlerquellen “ausklammern”. Bei der Eingabe vieler einzelner Befehle im direkten Modus wäre es nötig die komplette Eingabe noch einmal durchzuführen.

A.4.1.1 Erstellen von MATLAB-Skripten

An einem einfachen Beispiel werden die grundlegenden Eigenschaften eines Skripts verdeutlicht:

Der von MATLAB genutzte Editor (vgl. Abbildung A.4) öffnet sich automatisch über den Befehl

File → *New* → *M-File*

in der Menüleiste des MATLAB-Fensters⁶.

⁶Alternativ kann man auch im Command-Window den Befehl `edit` verwenden, oder auf das Icon in der Symbolleiste klicken

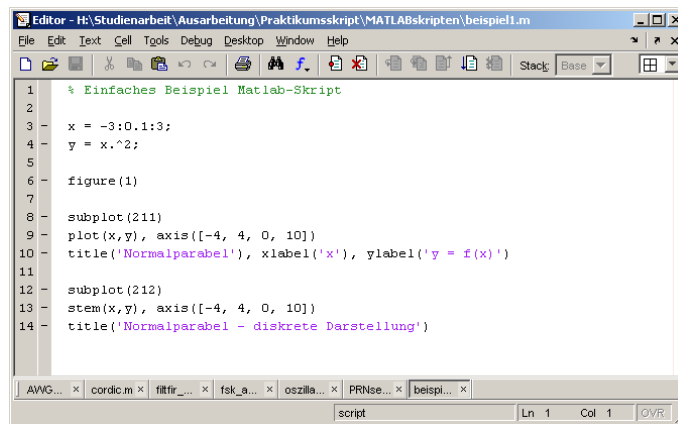


Abbildung A.4: MATLAB-Editor

Das Schreiben von MATLAB-Skripten erfolgt ähnlich wie im Command-Window⁷, mit dem Unterschied, dass nicht jeder Befehl sofort ausgeführt wird, sondern erst beim Starten des M-Files eine Befehlsabarbeitung erfolgt. Die wichtigsten Eigenschaften sollen am Beispiel des folgenden Quelltextes besprochen werden:

Listing A.1: Erstes MATLAB Skript

```

1  % Einfaches Matlab-Skript
2
3  x = -3:0.1:3;
4  y = x.^2;
5
6  figure(1)
7
8  subplot(211)
9  plot(x,y), axis([-4, 4, 0, 10])
10 title('Normalparabel'), xlabel('x'), ylabel('y=f(x)')
11
12 subplot(212)
13 stem(x,y), axis([-4, 4, 0, 10])
14 title('Normalparabel - diskrete Darstellung')

```

Kommentare werden in MATLAB immer mit % eingeleitet (vgl. Zeile 1 im Listing A.1). Hierzu muss am Anfang jeder Zeile ein %-Zeichen eingefügt werden⁸. Sie sollten von dieser Möglichkeit ausgiebig Gebrauch machen, um den geschriebenen Code lesbar und nachvollziehbar zu halten.

Ebenso wie im Command-Window werden nur diejenigen Werte auch am Bildschirm ausgegeben, die nicht mit einem Strichpunkt abgeschlossen sind.

⁷Allerdings ohne „»“ zu Beginn jeder einzelnen Zeile

⁸Alternativ kann man auch komplette Abschnitte markieren und über **Strg**+**R** auskommentieren lassen

Um ein fertiges Skript zu starten hat man mehrere Möglichkeiten:

- Direkt, per Eingabe des Skriptnamens im Command-Window
- Per Klick auf das *RUN* Symbol in der Symbolleiste des Editors
- Per Tastaturbefehl *F5*

Fehler im Code (vgl. Listing A.1 Zeile 13) werden direkt im Command-Window unter Angabe der Zeilennummer aufgelistet (vgl. Abb A.5).

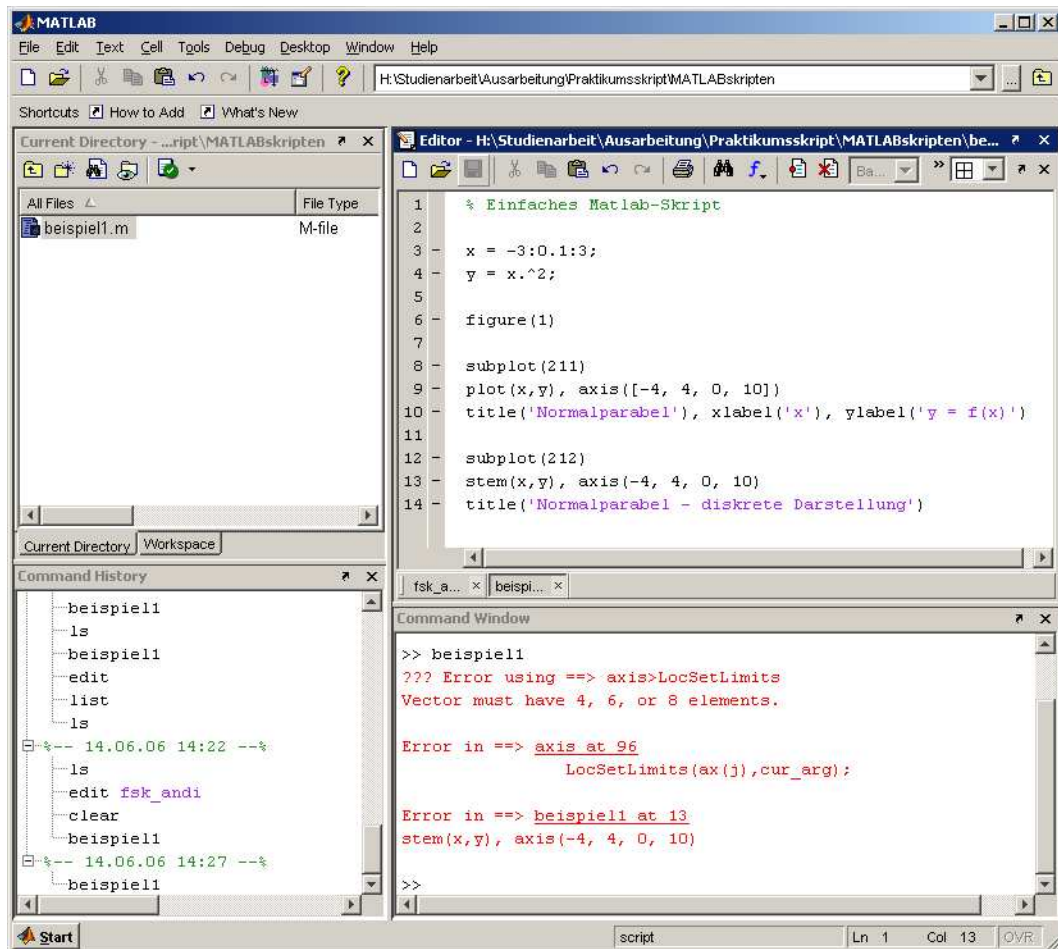


Abbildung A.5: Fehlerausgabe der MATLAB-Umgebung

A.4.1.2 Debugging

Im Debug-Mode ist es möglich die Abarbeitung eines Skripts anzuhalten, um sich Zwischenergebnisse oder Variablen-Inhalte während der Abarbeitung anzeigen zu lassen. Hierzu stehen in der Symbolleiste des Editorfensters verschiedene Befehle zur Verfügung, mittels derer man Haltepunkte, sogenannte *Breakpoints*, einfügen und handhaben kann (Vgl. Abb A.6). Die Befehlsabarbeitung stoppt dann am jeweiligen Breakpoint und wird erst nach manueller Bestätigung wieder aufgenommen.

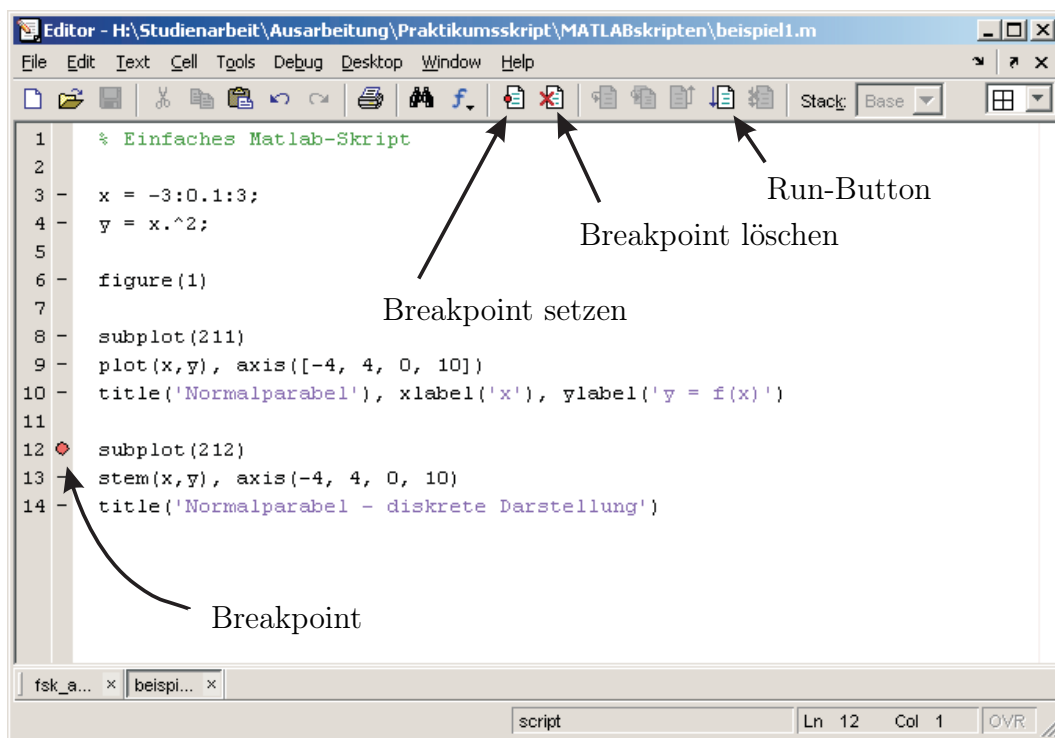


Abbildung A.6: Einfügen von Breakpoints in ein MATLAB-Skript

Jetzt besteht auch die Möglichkeit das Programm mit Hilfe des Step-Buttons Schritt für Schritt weiterlaufen zu lassen. Außerdem sind aktuelle Zwischenergebnisse der laufenden Berechnungen über das Command-Window abrufbar.

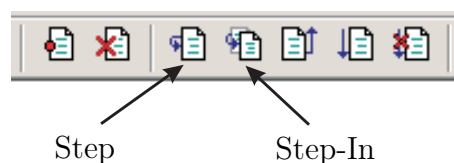


Abbildung A.7: Debugging-Symbolleiste

Mittels des Step-In-Buttons kann man in Unterprogramme einsteigen und deren Funktionen ebenso schrittweise ablaufen lassen.

A.4.2 Graphische Ausgabe

Wie bereits im Beispiel auf Seite 107 dargestellt kann der berechnete Vektor auch graphisch wiedergegeben werden. Im einfachsten Fall kann man mit dem Kommando `plot(x,y)` den Vektor `y` über seiner `x`-Koordinate darstellen lassen (vgl. Abb. A.8, oben).

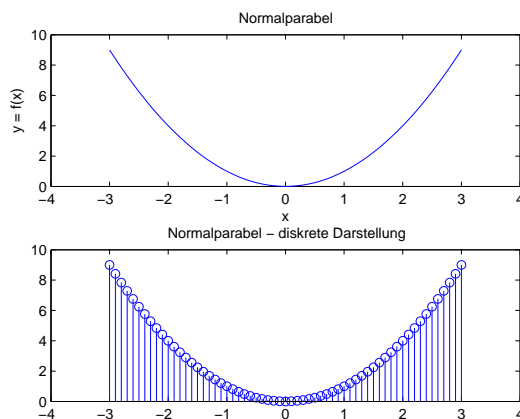


Abbildung A.8: Plot der Funktion $y = x^2$

Zur Darstellung diskreter Werte wurde im Beispiel (Listing A.1) schon der Befehl `stem(x,y)` eingeführt. Dieser stellt wirklich nur die berechneten Punkte dar (vgl. Abb. A.8, unten), während der `plot`-Befehl zwischen den einzelnen Werten linear interpoliert und somit einen kontinuierlichen, “eckigen”, Graphen erzeugt. Sie können dies einfach nachvollziehen, indem sie als `x`-Vektor weniger fein gerasterte Stützstellen für die Berechnung verwenden. Für weitere Grafikfunktionen sei auf die MATLAB-Online-Hilfe verwiesen.

Die im Beispiel verwendeten Formatierungsbefehle `axis`, `title`, `xlabel` und `ylabel` beschreiben die verwendete Achsenskalierung und das Aussehen der Achsen.

Über den Befehl `subplot(yxn)` können mehrere Graphen im gleichen Fenster dargestellt werden. Die `x`- und `y`-Parameter legen die Anordnung der einzelnen Graphen im Fenster fest, der Parameter `n` beschreibt die Position des aktuellen Plots.

Genauere Informationen zu den einzelnen Befehlen können der MATLAB-Hilfe entnommen werden.

A.4.3 MATLAB-Funktionen

Während MATLAB-Skripte in sich geschlossene Befehlsabläufe enthalten, können Funktionen auch Parameter von übergeordneten Files übergeben bekommen. Dies ist besonders dann sinnvoll, wenn die gleiche Funktionalität bei unterschiedlichen Eingangs-Parametern benötigt wird. Ein erstes Funktionsbeispiel ist im Listing A.2 dargestellt.

Listing A.2: Erste MATLAB Funktion

```

1 function [y]=parabel(n, s)
2 %% Syntax: [y]=parabel(n, s)
3 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
4 %% Berechnung einer Normalparabel mittels der Parameter
5 %% n:      Bereich +/-n
6 %% s:      Schrittweite der Stützstellen
7
8 x = -n:s:n;
9 y = x.^2;
10
11 figure(1)
12
13 subplot(211)
14 plot(x,y), axis([-n, n, 0, n^2])
15 title ('Normalparabel'), xlabel('x'), ylabel('y= $u^2$ (x)')
16
17 subplot(212)
18 stem(x,y), axis([-n, n, 0, n^2])
19 title ('Normalparabel—diskrete Darstellung')

```

Jede Funktion beginnt mit der Funktionsdeklaration wie in Zeile 1:

function [Ausgabeparameter] = Funktionsname(Übergabeparameter)

Es können der Funktion mehrere Parameter übergeben, sowie auch mehrere Ausgabeparameter definiert werden. Alles was direkt nach der Funktionsdeklaration im Kommentar enthalten ist wird bei Eingabe des Kommandos

help Funktionsname

als Ausgabe im Command-Window angezeigt. Dies funktioniert auch bei eigenen Funktionen, vgl. Kapitel A.4.3.

A.4.4 Kontrollstrukturen

Kontrollstrukturen werden für die eingabe- oder berechnungsabhängige Befehlsausführung, sowie für das Beschreibungen von Schleifen (z.B. bei rekursiven Strukturen) benötigt. Die wichtigsten Typen sind konditionale Verzweigungen und Schleifenkonstrukte.

A.4.4.1 Konditionale Verzweigung

Der einfachste Fall ist eine bedingte Befehlsausführung:

```

if Bedingung
    Anweisung
end

```

Wenn die Bedingung “true” ist, wird der Anweisungsteil ausgeführt. Dieser kann auch mehrere aufeinanderfolgende Einzelanweisungen enthalten.

Die konditionale Verzweigung

```
if Bedingung
    Anweisung
elseif Bedingung
    Anweisung
...
else Bedingung
    Anweisung
end
```

dient der Verzweigung bei mehreren möglichen Programmvarianten.

Ist die Bedingung wahr, wird der entsprechende Anweisungsteil ausgeführt, wobei - bei sich überschneidenden Bedingungen - immer der erste Zweig, der die Bedingung erfüllt, ausgeführt wird und alle anderen verworfen werden. Zur Formulierung von Bedingungen stehen die Vergleichsoperationen aus Tabelle A.6 sowie die logischen Operatoren aus Tabelle A.7 (Kapitel A.5) zur Verfügung.

In MATLAB hat eine richtige Bedingung den Wert 1 (true), während eine falsche Bedingung den Wert 0 (false) besitzt.

Konditionale Verzweigungen können auch bei Matrizen angewandt werden. Hier wird der `if`-Zweig allerdings nur durchlaufen, wenn alle Elemente der Vergleichsmatrix 1 sind. Der `else`-Zweig wird durchlaufen, wenn wenigstens ein Element 0 ist. Aus diesem Grund sollten in diesem Fall die Programme gut durchdacht werden, um eventuelle Fehler von vornherein auszuschließen.

Für detailliertere Informationen zum `if...elseif...else`-Konstrukt sowie zu weiteren konditionalen Verzweigungen (z.B. `switch...case`) sei auf die Fragezeichenhilfe von MATLAB verwiesen.

A.4.4.2 Schleifen

Schleifen lassen sich auf zwei mögliche Arten realisieren:

Ist die Anzahl der Schleifendurchläufe bekannt, so kann man mit

```
for Laufanweisung Anweisungsteil end
```

arbeiten. Der Anweisungsteil ist äquivalent zur bedingten Verzweigung aufgebaut. Eine *Laufanweisung* hat die Struktur

Laufvariable = *Anfangswert* [:*Schrittweite*] : *Endwert*

Der in eckigen Klammern beschriebene Teil ist optional, kann also weggelassen werden⁹. Nach Beenden der Schleife behält die Laufvariable den zuletzt angenommenen Wert - sie steht also auch außerhalb der Schleife zur Verfügung!

Häufig kann in MATLAB auf die Verwendung von `for`-Schleifen verzichtet werden. Dies soll folgendes Beispiel verdeutlichen:

Mit

⁹In diesem Fall ist die Schrittweite 1

```
>> n = 1000;  
>> a = rand(1,n);
```

wird ein Vektor mit n Zufallszahlen erzeugt. Sollen die Zahlen nun addiert werden kann man dies mit einer `for`-Schleife verwirklichen:

```
>> s = 0;  
>> for k = 1 : n, s = s + a; end  
>> s
```

Schneller und deutlich übersichtlicher ist aber die Matlabfunktion `sum`:

```
>> s = sum(a)
```

Die Programmierung mit `sum` bietet einen weiteren Vorteil: Misst man die Zeit, die zur Berechnung obiger Additionen notwendig ist, so stellt man fest, dass das Programm mit `sum` deutlich schneller ist als das Programm mit der `for`-loop! Hier stellt MATLAB die Funktionen `tic` und `toc` zur Verfügung:

```
>> tic; s = sum(a);toc
```

`tic` stellt eine Stoppuhr auf Null und startet, `toc` hält die Uhr an und gibt die Zeit in Sekunden aus.

Wird die Anzahl der Schleifendurchläufe erst innerhalb des Programms bestimmt, so kann man mit

```
while Bedingung, Anweisungsteil end
```

arbeiten. Solange die Bedingung, die in der Regel im Anweisungsteil neu bestimmt wird, erfüllt ist, wird der Anweisungsteil ausgeführt. Ist die Bedingung falsch, wird der anschließende Anweisungsteil übergangen und das Programm unterhalb der `end`-Anweisung fortgesetzt.

Beispiel:

```
a = 0  
while a < 11  
    a = a + 1;  
end
```

A.5 Zusammenfassung

Zusammenfassung oft verwendeter MATLAB-Befehle nach Themen¹⁰

Befehl	Klartext
who	listet verwendete Variablen auf
whos	listet verwendete Variable auf, ausführlich
workspace	aktiviert den Workspace
clear	löscht Variable
load	lädt gespeicherte Daten
save	speichert Daten
quit	beendet die MATLAB-Sitzung

Tabelle A.3: MATLAB Arbeitsspeicher

Befehl	Klartext
diary	speichert Eingaben einer MATLAB-Sitzung
format	legt das Ausgabeformat fest
beep	erzeugt einen Piepton

Tabelle A.4: Command-Window Kontrolle

Zeichen	Klartext
+	Matrizenaddition / Polynomaddition
*	(Matrizen) multiplizieren
.*	Matrizen komponentenweise multiplizieren
\	Linksdivision von Matrizen
.\	komponentenweise Linksdivision (bei Matrizen)
/	Rechtsdivision von Matrizen
./	komponentenweise Rechtsdivision (bei Matrizen)
^	(Matrizen) potenzieren
.^	Matrizen komponentenweise potenzieren

Tabelle A.5: Arithmetische Operationen

¹⁰Eine vollständige Darstellung findet man mit »help, »help general, » help ops...

Zeichen	Klartext
<	kleiner als
>	größer als
<=	kleiner gleich
>=	größer gleich
==	genau gleich
~=	ungleich

Tabelle A.6: Vergleichsoperatoren

Operator	Klartext
&	und
	oder
~	nicht
xor	exklusives oder

Tabelle A.7: Logische Operatoren

Befehl	Klartext
error	Fehlermeldung und Beenden der Funktion
warning	Warnung, ohne die Funktion zu beenden
disp	Meldung
fprintf	schreibt Daten in ein File
sprintf	schreibt Daten in einen Stream

Tabelle A.8: Meldungen im Command Window erzeugen

Befehl	Klartext
zeros	Nullmatrix
ones	Einsmatrix
eye	Einheitsmatrix
rand	Zufallsmatrix, gleichverteilte Zahlen
randn	Zufallszahlen, normalverteilte Zahlen
linspace	Vektor mit äquidistant verteilten Elementen
logspace	Vektor mit logarithmisch verteilten Elementen

Tabelle A.9: Elementare Matrizen

Befehl	Klartext
size	Typ einer Matrix
length	Anzahl der Elemente eines Vektors

Tabelle A.10: Grundlegende Informationen über Matrizen

Befehl	Klartext
ans	zuletzt gegebene Antwort
eps	Abstand von 1 zur nächstgrößeren darstellbaren Zahl
realmax	größtmögliche darstellbare Zahl
realmin	kleinstmögliche positive darstellbare Zahl
pi	3.1415926535897
i,j	imaginäre Einheit
Inf	unendlich
NaN	keine Zahl

Tabelle A.11: Spezielle Variable und Konstanten

Befehl	Klartext
sin	Sinus
sinh	Sinushyperbolicus
asin	Arcusinus
asinh	Areasinus
cos	Cosinus
cosh	Cosinushyperbolicus
acos	Arcuscosinus
acosh	Areascosinus
tan	ATangens
tanh	Tangenshyperbolicus
atan	Arcustangens
atanh	Areastangens
cot	Cotangens
coth	Cotangenshyperbolicus
acot	Arcuscotangens
acoth	Areacotangens

Tabelle A.12: Trigonometrische Funktionen

Befehl	Klartext
exp	Exponentialfunktion mit Basis e
log	Logarithmus naturalis (ln)
log10	Logarithmus zur Basis 10
log2	Logarithmus zur Basis 2

Tabelle A.13: Exponentialfunktionen und Logarithmen

Befehl	Klartext
abs	Betragsfunktion
angle	Winkelberechnung
conj	konjugiert komplexe Zahl
imag	Imaginärteil
real	Realteil
unwrap	beseitigt Phasensprünge

Tabelle A.14: Funktionen bei komplexen Zahlen

Befehl	Klartext
fix	rundet zu Null hin
floor	rundet nach unten
ceil	rundet nach oben
round	rundet zur nächsten ganzen Zahl
mod	vorzeichenbehafteter Rest nach Division
rem	Rest nach Division
sign	Vorzeichenfunktion

Tabelle A.15: Runden und Reste

Befehl	Klartext
norm	Norm
rank	Rang
det	Determinante
trace	Spur
rref	Gauss-Algorithmus
\	Linksdivision
/	Rechtsdivision
inv	Inverse
cond	Kondition
pinv	Pseudoinverse
eig	Eigenwerte und Eigenvektoren
poly	Charakteristisches Polynom

Tabelle A.16: Funktionen bei Matrizen

Befehl	Klartext
max	größtes Element
min	kleinstes Element
mean	Mittelwert
median	Median
std	Standartabweichung
var	Varianz
sort	sortieren (aufsteigend)
sum	Summe (spaltenweise)
prod	Produkt (spaltenweise)
cumsum	kumulative Summe
cumprod	kumulatives Produkt
diff	Differenzbildung

Tabelle A.17: Datenanalyse

Befehl	Klartext
roots	Nullstellenbestimmung
poly	Polynom berechnen aus Nullstellen
polyval	Polynom auswerten
residue	Partialbruchzerlegung
polyfit	Interpolation oder Regression
polyder	Ableitung eines Polynoms
conv	Multiplikation
deconv	Division
spline	Spline

Tabelle A.18: Polynome und Interpolation

Befehl	Klartext
quad	numerische Integration
quadl	numerische Integration
ode45	numerische Lösung von Differentialgleichungen

Tabelle A.19: Funktionen zur numerischen Integration

Befehl	Klartext
plot	Linearer Plot
loglog	logarithmischer Plot
semilogx	halblogarithmischer Plot
semilogy	halblogarithmischer Plot
polar	Plot in Polarkoordinaten
axis	Kontrolle der Achsen
grid	Gitterhilfslinien
subplot	weitere Plots in einem Fenster
hold on	weiter ins aktuelle Fenster
hold off	Fenster schließen
legend	Legende
title	Titel
xlabel	Beschriftung der x-Achse
ylabel	Beschriftung der y-Achse
text	Text positionieren
gtext	Text mit der Maus positionieren

Tabelle A.20: Graphiken2D

B ispLEVER

B.1 Die Oberfläche

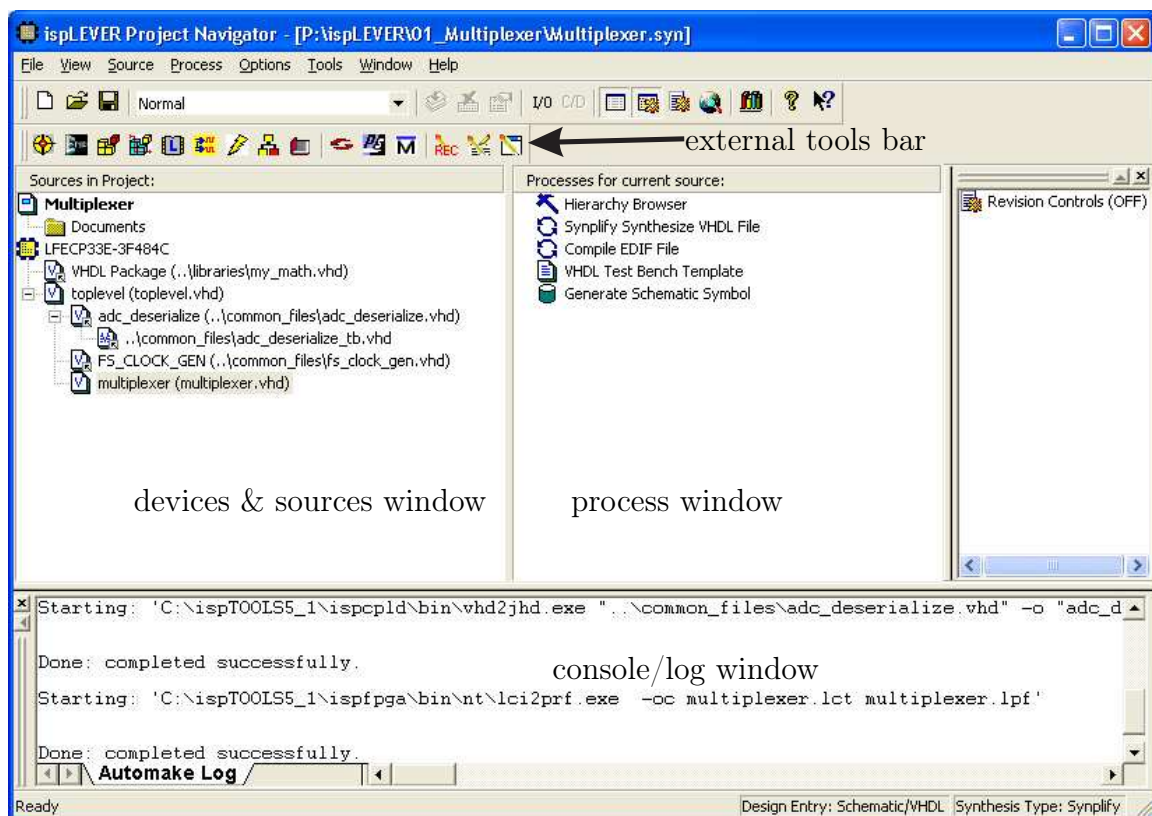


Abbildung B.1: Die ispLEVER-Oberfläche

ispLEVER ist das von Lattice¹ zur Verfügung gestellte Tool zur Programmierung derer isp²-Bausteine. Der Aufbau der Oberfläche (vgl. Abb. B.1) gestaltet sich übersichtlich.

Aus Zeitgründen können wir im Praktikum leider nicht genauer darauf eingehen, wie die Hardware zu konfigurieren ist. Diese Konfiguration beinhaltet die Zuweisung der Pins zu den Ports im Toplevel, die IO-Standards³ und vieles nützliche mehr.

Auch die einzelnen Tools⁴, die für die Erstellung des Praktikums Verwendung fanden, können allenfalls beiläufig erwähnt werden. Ein kurzer Blick hinter die Kulissen schadet

¹<http://www.latticesemi.com/>

²isp Abk. In-System Programmable

³z.B. LVCMOS 3.3V, 2.5V; LVTTL,...

⁴z.B. Floorplanner, IPexpress, ...

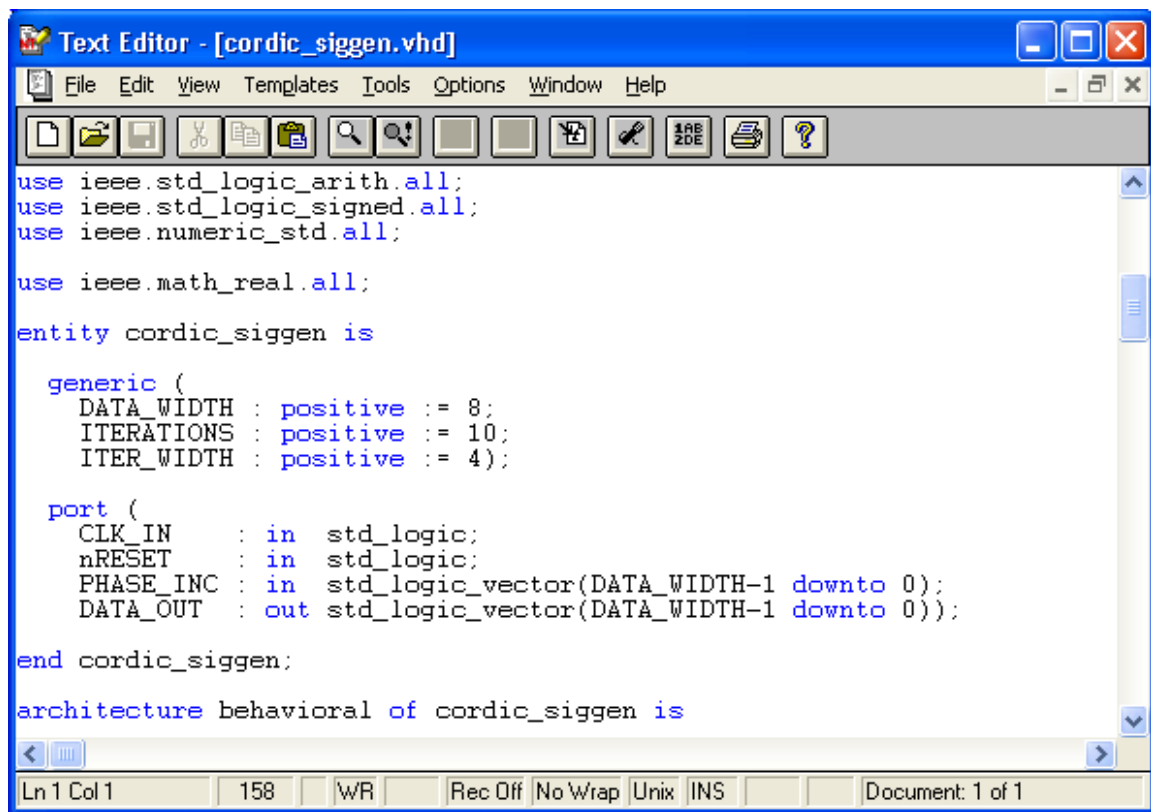


Abbildung B.2: Der integrierte Text Editor von ispLever

jedoch nicht, solange man nach den Veränderungen nicht abspeichert.

Die grundlegenden Funktionen dürften wohl jedem bekannt sein, der schon mit Windows gearbeitet hat. Im Grunde ist jedoch alles bereits voreingestellt, um direkt mit dem Schreiben des Sourcecodes anfangen zu können. Zum Editieren eines bestimmten Codes sollte man einfach auf die entsprechende Datei im source window doppelklicken und die Datei öffnet sich im eingestellten Editor.

B.2 Der VHDL-Editor

Derin ispLever integrierte Text Editor (vgl. Abb. B.2) stellt sich sehr übersichtlich dar. Er bietet Syntax Highlighting, Templates und Macrofunktionen für wiederkehrende Aufgaben. Eine genaue Ausführung dieser speziellen Funktionen soll im Rahmen des Praktikums nicht erfolgen.

B.3 Wie geht es weiter?

In der ersten Übung wird genauer anhand eines Beispiels darauf eingegangen, wie der VHDL-Code zu simulieren ist und wie man diesen dann in die Hardware überträgt.

C VHDL-Referenz

C.1 Einführung

Da auf die einzelnen Konstrukte von VHDL, die in diesem Praktikum verwendet werden, in den jeweiligen Kapiteln genauer eingegangen wird, soll an dieser Stelle lediglich eine Kurzreferenz zur Verfügung gestellt werden, die ein schnelles Nachschlagen der Konstrukte erlaubt. Eine genaue Beschreibung finden sie in [10] bzw. [1] oder den entsprechenden Kapiteln dieses Skripts. Eine weitere gute Referenz ist [2].

C.2 Basiskonstrukte

C.2.1 Libraries

```
library <LIB>;  
use <LIB>.<PACKAGE>.<FUNCTION|all>;  
use ...
```

vgl.: 5.1.3.1 S.12

C.2.2 Entity

```
entity <NAME> is  
  generic (  
    <GENERIC>  
  );  
  port (  
    <PORTS>  
  );  
end <NAME>;
```

vgl.: 5.1.3.2 S.14

C.2.3 Architecture

```
architecture <NAME> of <ENTITY> is  
  <DECLARATIONS>  
begin
```

```
<INSTANTIATIONS>  
end <NAME>;
```

vgl.: 5.1.3.3 S.14

C.2.4 Signale und Konstanten

```
signal <NAME>: <TYPE> [range <RANGE>] [:= <INITIALIZATION>];  
constant <NAME>: <TYPE> [range <RANGE>] [:= <INITIALIZATION>];
```

vgl.: 5.2.4.1 S.20 und 5.3.5.2 S. 44

C.2.5 Typen

```
type <NAME> is array of (<RANGE>) <TYPE>;  
type <NAME> is record  
    <ELEMENT(S)>  
end record;  
subtype <NAME> is <TYPE>;
```

vgl.: 5.3.5.5 S.45

C.2.6 Typenkonvertierung

```
conv_integer(<ELEMENT>);  
conv_unsigned(<ELEMENT>, <LENGTH>);  
conv_signed(<ELEMENT>, <LENGTH>);  
conv_std_logic_vector(<ELEMENT>, <LENGTH>);
```

vgl.: 5.3.5.3 S.44

Verwandte Datentypen können auch folgendermaßen umgewandelt werden:

```
integer(<ELEMENT>);  
natural(<ELEMENT>);  
unsigned(<ELEMENT>);  
signed(<ELEMENT>);  
std_logic_vector(<ELEMENT>);
```

C.2.7 Prozesse

```
[<name> :] process [(sensitivity list)]  
  [variable <NAME>: <TYPE> [:= <INITIALIZATION>]];  
begin  
...  
  -- CODE  
...  
end;
```

vgl.: 5.2.4.2 S.21

C.2.8 Kontrollstrukturen

C.2.8.1 if-then-else

```
if <BEDINGUNG> then  
  -- CODE  
[elsif <BEDINGUNG> then  
  -- CODE ]  
[else  
  -- CODE ]  
end if;
```

vgl.: 5.2.4.3 S.22

C.2.8.2 when-else

```
<SIGNAL> <= <SIGNAL> when <CONDITION> [else  
  <SIGNAL> when <CONDITION>];
```

vgl.: 5.1.3.4 S.15

C.2.8.3 case-when

```
case (<EXPRESSION>) is  
  when <CHOICE> => <SEQUENTIAL STATEMENTS>;  
  [when <CHOICE> => <SEQUENTIAL STATEMENTS>;]  
end case;
```


Abbildungsverzeichnis

4.1	ADSP-SPATES	6
4.2	Netzteil-Brücke	6
4.3	Power cable	6
5.1	Sender Blockschaltbild	11
5.2	8-bit 4-zu-1-Multiplexer	12
5.3	Synchrone Schaltung	13
5.4	Der Toplevel des Multiplexers	16
5.5	Das ispVM-Fenster	17
5.6	PRN-Schieberegister	18
5.7	PRN-Schieberegisterdurchlauf	19
5.8	16-stufiges Schieberegister mit XOR-Rückkopplung	23
5.9	Testbench simulieren	25
5.10	Modelsim nach dem Start mittels ispLEVER	26
5.11	Erneutes Compilieren einer VHDL-Datei	27
5.12	Direkte Digitale Synthese (DDS)	29
5.13	Erweiterte DDS	30
5.14	Drehung eines Vektors um den Winkel θ	33
5.15	Schrittweite der Variable z_i	34
5.16	Quantisierungskennlinie mit Zweierkomplementüberlauf	40
5.17	Quantisierungskennlinie mit Sättigungsverhalten	41
5.18	Sender Blockschaltbild	55
5.19	VHDL-Datei importieren	56
5.20	Importdialog	57
6.1	Blockschaltbild der Empfängerstufe	59
6.2	Bitstrom und Sendesignal	59
6.3	Bandpassgefilterte Signale	60
6.4	Bandpassgefiltertes und quadriertes Signal	60
6.5	Tiefpassgefiltertes Signal	61
6.6	Rekonstruiertes Empfangssignal	61
6.7	Echtzeitsystem zur digitalen Filterung	62
6.8	Filterfrequenzgänge	63
6.9	FIR-Filter 3. Ordnung	65
6.10	Symmetrieeigenschaften von FIR Filter	66
6.11	Frequenzgang eines linearphasigen TP-Filters	66
6.12	Impulsantwort eines linearphasigen TP-Filters	67
6.13	Pol-Nullstellendiagramm eines linearphasigen TP-Filters	67

6.14	Ein- und Ausgangsverhalten bei Anregung mit einem punktsymmetrischen Doppelrechteckpul	
6.15	Ein- und Ausgangsverhalten im Frequenzbereich	68
6.16	Direktform- oder Transversalfilterstruktur bei FIR-Filtern	68
6.17	Linearphasenstruktur eines FIR-Filters	68
6.18	Toleranzschema des Amplitudengangs	69
6.19	Fenstermethode im Zeitbereich	71
6.20	Fenstermethode im Frequenzbereich	71
6.21	Fenstertypen	72
6.22	Fehlermodell für quantisierte Koeffizienten	74
6.23	Überlaufbehandlung bei FIR-Filtern in Direktformstruktur	77
6.24	Graphische Oberflächen des FDA-Tools	80
6.25	Iconleiste im FDA-Tool	80
6.26	Fixed-Point Koeffizienten mit FDA-Tool	81
6.27	Export von Filterkoeffizienten	81
6.28	Linearphasenstruktur eines FIR-Filters	87
6.29	Quadrierungsschema	90
7.1	HyperTerm-Eigenschaften	97
7.2	Einstellen der Übertragungsrate	97
A.1	MATLAB-Desktop-ICON	101
A.2	Eine jungfräuliche MATLAB-Benutzeroberfläche	101
A.3	Startseite der MATLAB Hilfe	103
A.4	MATLAB-Editor	109
A.5	Fehlerausgabe der MATLAB-Umgebung	110
A.6	Einfügen von Breakpoints in ein MATLAB-Skript	111
A.7	Debugging-Symbolleiste	111
A.8	Plot der Funktion $y = x^2$	112
B.1	Die ispLEVER-Oberfläche	123
B.2	Der integrierte Text Editor von ispLever	124

Tabellenverzeichnis

4.1	Zuordnungen der Taster und Schalter	7
5.1	Der Typ <code>std_logic</code>	13
5.2	Vorgeschlagene Präfixe der bisher bekannten Typen	20
5.3	DDS Look-Up-Table	32
5.4	Look-Up-Table für α_i und Skalierungsfaktor k	34
5.5	Beispiel zur Auswirkung von Teilüberläufen	40
5.6	Port des LUT-Moduls	48
A.1	Weitere Matrizenoperationen	105
A.2	Standartmatrizen	105
A.3	MATLAB Arbeitsspeicher	116
A.4	Command-Window Kontrolle	116
A.5	Arithmetische Operationen	116
A.6	Vergleichsoperatoren	117
A.7	Logische Operatoren	117
A.8	Meldungen im Command Window erzeugen	117
A.9	Elementare Matrizen	117
A.10	Grundlegende Informationen über Matrizen	117
A.11	Spezielle Variable und Konstanten	118
A.12	Trigonometrische Funktionen	118
A.13	Exponentialfunktionen und Logarithmen	118
A.14	Funktionen bei komplexen Zahlen	119
A.15	Runden und Reste	119
A.16	Funktionen bei Matrizen	119
A.17	Datenanalysis	120
A.18	Polynome und Interpolation	120
A.19	Funktionen zur numerischen Integration	120
A.20	Graphiken2D	121

Literaturverzeichnis

- [1] P. Ashenden, *The Designers Guide to VHDL (Systems on Silicon)*. Morgan Kaufmann Publishers Inc,US, 2001.
- [2] A. Dennis, E. Quighey, P. Findlay, and S. Tippins, “Vhdl: Reference guide,” 1995.
- [3] J. Dongarra, J. Bunch, C. Moler, and G. Stewart, *LINPACK User’s Guide*. Society for Industrial and Applied Mathematics, 1993.
- [4] F. Grupp and F. Grupp, *MATLAB 7 für Ingenieure - Grundlagen und Programmierbeispiele*. Oldenbourg Verlag München Wien, 2004.
- [5] M. Huemer, *Skriptum zur Vorlesung “Architekturen der Digitalen Signalverarbeitung”*. Lehrstuhl für Technische Elektronik, Universität Erlangen-Nürnberg, 2006.
- [6] *LatticeECP/EC Family Data Sheet*, 02nd ed., Lattice semiconductor corporation, March 2006. [Online]. Available: http://www.latticesemi.com/dynamic/view_document.cfm?document_id=8517
- [7] *LatticeECP/EC Family Handbook*, 02nd ed., Lattice semiconductor corporation, April 2006. [Online]. Available: http://www.latticesemi.com/dynamic/view_document.cfm?document_id=8518
- [8] B. Smith, J. Boyle, and J. Dongarra, *Matrix Eigensystem Routines - EISPACK Guide*. Springer-Verlag GmbH, 1988.
- [9] P. D.-I. J. Thielecke, *Skriptum zur Vorlesung “Satellitengestützte Ortsbestimmung”*. Lehrstuhl für Informationstechnik mit dem Schwerpunkt Kommunikationselektronik, Universität Erlangen-Nürnberg, 2006.
- [10] “Vhdl-online,” Universität Erlangen-Nürnberg, LRS. [Online]. Available: <http://www.vhdl-online.de>
- [11] D. von Grünigen, *Digitale Signalverarbeitung mit einer Einführung in die kontinuierlichen Signale und Systeme*. Fachbuch Verlag Leipzig, 2004.