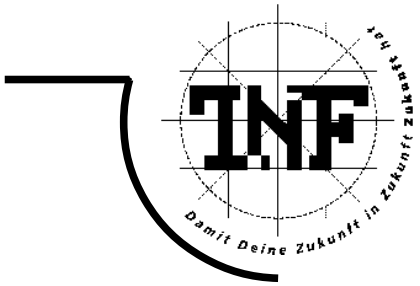
JOHANNES KEPLER
UNIVERSITÄT LINZ
Netzwerk für Forschung, Lehre und Praxis



Eine Hardwareimplementierung eines FORTH-Systems mit einem Java Compiler

DIPLOMARBEIT

zur Erlangung des akademischen Grades

DIPLOMINGENIEUR

in der Studienrichtung

INFORMATIK

Angefertigt am *Institut für integrierte Schaltungen*

Betreuung:

O. Univ.-Prof. DI Dr. Richard Hagelauer

Eingereicht von:

Gerhard Hohner

Linz, August 2008

Inhaltsverzeichnis

Abbildungsverzeichnis

Einleitung

1 Was ist FORTH?

FORTH ist vielseitig. Man versteht darunter ein Echtzeitbetriebssystem, einen Interpreter oder einen interaktiven Compiler, eine erweiterbare Datenstruktur, ein Softwarekonzept. In erster Linie ist FORTH aber als strukturierte Sprache zu sehen, die von ihren Anhängern der vierten Generation zugerechnet wird, und die mit sehr wenigen Sprachelementen auskommt, deren Sprachumfang aber, was Befehle und Datenstrukturen anlangt, erweiterbar ist. Jedes Programm, jede Variable wird als Spracherweiterung aufgefasst, und vom Laufzeitsystem in einem Dictionary als so genanntes Token verwaltet. Das bedeutet, dass auch ein geladenes Programmsystem jederzeit erweitert und ergänzt werden kann, ohne die gesamte Anwendung neu kompilieren und laden zu müssen. Für die Programmentwicklung bedeutet es weiter, dass ein schrittweises, aufbauendes Testen, beginnend mit den Grundfunktionen der Applikation, auf dem Zielsystem, möglich ist.

Der Kern eines FORTH-Systems ist eine Stapelmaschine mit einem Datenstapel zur Berechnung arithmetischer Ausdrücke, Parameterübergabe und Ergebnisrückgabe, und einem eigenen Stapel der Rücksprungadressen. Die Maschine erwartet keine getrennten Speicher für Daten und Programme. Außerdem muss es ein Laufzeitsystem geben, das eine Benutzerschnittstelle bietet und das Dictionary verwaltet.

2 Zu realisierendes FORTH-System

Der Kern soll durch einen eigens zu entwickelnden Prozessor realisiert werden, dessen Befehlssatz aus den elementaren Befehlen von FORTH besteht. Als Architektur ist eine Stapelmaschine mit zwei Stapeln vorgegeben, die in einem FPGA zu implementieren ist. Das Laufzeitsystem (Server) ist ebenfalls auf diesem Baustein unterzubringen, und enthält eine Bibliothek mit Standardfunktionen und Diensten. Das System soll minimal sein, also bloß aus dem Prozessor, einem Speicher und einer seriellen Schnittstelle zur Kommunikation mit einem Client bestehen. Die Logik des UART ist auch auf dem FPGA zu implementieren, verwaltet wird diese integrierte Komponente vom Laufzeitsystem. Der Client läuft auf einem normalen PC, und bietet die Benutzerschnittstelle, sowie Dienstprogramme (Assembler, etc.) an. Letztendlich soll es möglich sein mit dem Client FORTH-Programme zu übersetzen, die daraus erstellte Objektdatei an den Server zu senden, der diese ladet, bindet, initialisiert, im Dictionary einträgt und, wenn vorgegeben, startet. Zu einem Zeitpunkt kann nur eine Anwendung auf dem Minimalsystem laufen, die aber jederzeit abgebrochen werden kann. Es können aber mehrere Anwendungen geladen sein, die auch in Wechselwirkung stehen können.

3 Java

Es ist wünschenswert eine höher entwickelte Sprache zur Verfügung zu haben, da FORTH doch sehr einfach ist und genau genommen bloß eine Sprache der dritten Generation ist. Java bietet sich an. Zwei Möglichkeiten sind denkbar, zum Ersten eine Java Virtual Machine, kurz eine JVM, die auf dem Minimalsystem läuft und den Bytecode

der Java-Klassen interpretiert, zum Zweiten ein Compiler, der als Dienstprogramm des Clients bereit steht und Java nach FORTH übersetzt.

3.1 Java Virtual Machine

Argumente für eine JVM:

- Die Klassen einer Anwendung sind immer aktuell. Es genügt den aktuellen JDK herunterzuladen und die Anwendung an das Minimalsystem neu zu übertragen. Bloß die JVM muss dem aktuellen Bytecode von Java angepasst werden.

Argumente dagegen:

- Ein Emulator für den Prozessor der JDK-Plattform muss implementiert werden, um in Maschinencode geschriebene Methoden, gekennzeichnet durch das Attribut *native,* ausführen zu können.
- Eine völlig transparente, von der JVM und den Applikationen gänzlich unabhängige, dynamische Speicherverwaltung und Speicherbereinigung muss implementiert werden, etwa *Mark and Sweep*. Das Unangenehme daran ist, dass kein Einfluss auf die Speicherbereinigung genommen werden kann. Während dieser Prozess, zu einem unvorhersehbaren Zeitpunkt mit unbekannter Dauer läuft, kann eine Anwendung ihre Aufgaben nicht wahrnehmen.
- Es müssen alle benötigten Klassen einer Anwendung sofort geladen werden, was einen sehr großen Speicher des Minimalsystems voraussetzt. Abschätzungen der typischen Speichergröße können nicht gemacht werden.

Da die Nachteile gewichtig sind, wird auf eine JVM verzichtet.

3.2 Compiler

Vorteile:

- Die Anwendungen können im Maschinencode ausgeführt werden.
- Die Klassen aus dem JDK, deren Quellcode Sun kostenlos bereitstellt, können den Bedürfnissen des Minimalsystems angepasst werden; der Speicherbedarf kann klein gehalten werden.
- *Reference Counting* kann als dynamische Speicherverwaltung eingesetzt werden. Die Anwendungen haben dabei volle Kontrolle über die Speicherbereinigung. Speicher wird genau dann freigegeben, wenn er nicht mehr benötigt wird und steht sofort wieder zur Verfügung.
- Das Minimalsystem muss nicht für Java adaptiert werden.

Nachteile:

- Die Quellklassen müssen angepasst werden.
- *Reflection* ist wahrscheinlich nicht möglich.
- Man muss sich auf eine Version von Java festlegen, die Adaptierung auf den neuesten Stand ist kein unerheblicher Aufwand.

Der Compiler ist die attraktivere Option, zumal Java wie eine Schale über das FORTH-

System gelegt wird und für das System selbst nicht erkennbar ist.

4 Die Geschichte von FORTH

Folgende Chronologie repräsentiert eine Kurzfassung von [17].

1968	Charles Moore beginnt mit der Entwicklung von FORTH auf einem IBM-System
1970	Charles Moore publiziert das erste Paper über FORTH
1971 zur	Charles Moore implementiert FORTH auf einem Minicomputer für die NRAO Steuerung des Radioteleskops auf dem Kitt Peak
1973 FORTH	Charles Moore, Elizabeth D. Rather und Edward K. Conklin gründen die Inc.
1976 6800	Entwicklung von microFORTH für RCA für die Mikroprozessoren 8080, Z80,
1978	Selzer, Ragsdale entwickeln ein FORTH-System für den 6502
1978	FIG (FORTH Interest Group) wird gegründet
1980	FORTH läuft auch unter anderen Betriebssystemen (Apple II, CP/M)
1982	IBM PC FORTH läuft unter PC-DOS auf dem PC von IBM
1985	der erste FORTH-Prozessor wird von Harris angeboten
1987	Bildung eines Komitees zur Standardisierung von FORTH
1994	Veröffentlichung des Standards ANSFORTH in DPANS'94 durch ANSII
1998	Das Komitee stellt seine Arbeit ein

Hardware

1 Entwurf des Prozessors

Hier werden nur Festlegungen bezüglich des Prozessors getroffen, so detailliert wie nötig und so allgemein wie möglich. Es werden keinerlei Angaben bezüglich der Software gemacht.

1.1 Die elementaren FORTH-Befehle

Alle elementare Befehle [2, Kapitel 6, Seite 21 - 46], gruppiert nach Aufgabenfeldern, mit Mnemonik und detaillierter Beschreibung, werden angeführt. Darüber hinaus gibt es noch komplexe Befehle, die im Standard angeführt werden, diese lassen sich aber durch eine Kombination der elementaren Befehle (Makros) nachbilden.

Einige Bemerkungen zur Darstellung der Stapel in der Spalte *Aktion* in den folgenden Tabellen:

- Stapel werden im Zustand Vorher und Nachher gezeigt, und sind Durch „--“ getrennt. Sind von einem Befehl mehrere Stapel betroffen, erhält der zweite Betroffene ein Präfix. In den Tabellen ist dies bloß der Rücksprungadressenstapel, mit Präfix *R:*.
- Das am weitesten rechts stehende Element im Stapel ist dessen oberstes Element. Alle vorher angeführten liegen tiefer.
- addr steht für eine beliebige Adresse.
- n steht für ein vorzeichenbehaftetes Wort.
- u steht für ein vorzeichenloses Wort.
- x steht für ein Wort, dessen tatsächlicher Typ unbedeutend ist.
- half (offset) steht für ein vorzeichenbehaftetes Halbwort.
- char steht für ein einzelnes Zeichen, ein vorzeichenbehaftetes 8-Bit Zeichen.
- *Kursive* Mnemonik kennzeichnet Befehle, die im Standard nicht definiert sind.

1.1.1 Speicherbefehle

Da der Standard eine Wortbreite von 16 Bit vorsieht, dies aber in Hinblick auf Java unzulänglich ist, habe ich eine 32-Bit Architektur realisiert. Dies erfordert aber zusätzliche Befehle für Halbworte und Zeiger, für Doppelworte sind keine Befehle nötig, diese können nachgebildet werden. Die Speicherbefehle sind in Tabelle 1 angeführt.

1.1.1 Speicherbefehle

Mnemonik	***Aktion***
!	n addr -- speichert das zweite Wort n unter Speicheradresse addr und eliminiert die beiden Worte vom Stapel
@	addr – x ersetze addr durch den unter addr gespeicherte Inhalt x
C!	char addr -- speichert das zweite Wort char als einzelnes Zeichen unter Speicheradresse addr und eliminiert die beiden Worte vom Stapel
C@	addr – char ersetze addr durch das unter addr gespeicherte Zeichen
H!	half addr -- speichert das zweite Wort half als einzelnes Doppelbyte unter Speicher-adresse addr und eliminiert die beiden Worte vom Stapel
H@	addr – half ersetze addr durch das Doppelbyte auf Adresse addr
SP!	u addr -- initialisiert den Stapelzeiger mit Adresse addr und setzt die momentane Länge des Überlaufbereichs auf u Worte.
SP@	-- addr kopiert den Stapelzeiger auf den Stapel
B!	u -- der Übertrag wird gesetzt, wenn u <> 0, sonst gelöscht
B@	-- u u hat den Wert -1, wenn der Übertrag gesetzt ist, sonst 0
32@	– u liest das nächste Wort – beginnend mit dem höchstwertigen Wort – des Produkts des 64-bit-Multiplizierers.
64!	u u – lädt den Doppelwortoperanden in den 64-bit-Multiplizierer

Tabelle 1: Die Speicherbefehle

1.1.2 Stapelmanipulationsbefehle

Zweck dieser Befehle ist ausschließlich die Manipulation des Daten- oder Rücksprungadressenstapels. Es gibt aber einige Standardbefehle, die mit Doppelworten operieren, die nicht angeführt sind, sie können nachgebildet werden. Die Manipulationsbefehle sind in Tabelle 2 angeführt.

1.1.2 Stapelmanipulationsbefehle

Mnemonik	***Aktion***
DEPTH	-- u legt die Länge des Stapels (in Worten) auf den Stapel
DROP	x -- löscht das oberste Wort vom Stapel
2DROP	x1 x2 -- löscht die zwei obersten Worte vom Stapel
NIP	x1 x2 – x2 eliminiere das zweite Wort im Stapel
PICK	xu ... x0 u – xu ... x0 xu kopiere das u-te Element im Stapel an die Spitze des Stapels
PUT	xu xu-1 ... x0 n u – n xu-1... x0 das zweite Wort n ersetzt im Stapel das Wort an der relativen Position u im Stapel und eliminiert die beiden Operanden vom Stapel
[+|-] n *[VAL]*	-- n legt n auf den Stapel
[+|-] half *[VALH]*	-- half wandelt half in ein vorzeichenbehaftetes Wort und legt es auf den Stapel
[CONST] [+|-]x x ϵ [0, 15]	-- x wandelt x in ein vorzeichenbehaftetes Wort und legt es auf den Stapel
DUP	n1 – n1 n1 dupliziert das oberste Wort des Datenstapels
OVER	n1 n2 – n1 n2 n1 kopiert das zweite Wort des Datenstapels auf den Stapel
R@	-- n1 R: n1 -- n1 kopiert das oberste Wort des alternativen Stapels auf den Datenstapel
R1@	-- n1 R: n1 n2 – n1 n2 kopiert das zweite Wort des alternativen Stapels auf den Datenstapel

Mnemonik	***Aktion***
SAVE	... x -- Sichert den Cache des Stapels vollständig in den Überlaufbereich

Tabelle 2: Die Stapelmanipulationsbefehle

Die Unterteilung der Zahlen in 32-Bit, 16-Bit und 5-Bit Werte erhöht die Zahl der Befehle zwar, reduziert die Größe des compilierten Codes aber erheblich.

1.1.3 Arithmetische und logische Operationen

Die Befehle dieser Gruppe sind in Tabelle 3 angeführt.

Mnemonik	***Aktion***
+	n1|u1 n2|u2 – n3|u3 addiert die beiden obersten Worte und ersetzt sie durch deren Summe
-	n1|u1 n2|u2 – n3|u3 ersetzt die beiden obersten Worte durch die Differenz n1 - n2
CMP	n1|u1 n2|u2 – n1|u1 n2|u2 n3|u3 legt die Differenz n1 – n2 als n3 auf den Stapel
*H**	uhalf1 uhalf2 – u das vorzeichenlose Produkt der beiden Halbwort ersetzt diese am Stapel
1+ --	n1|u1 – n2|u2 inkrementiert das oberste Wort um 1
1- --	n1|u1 – n2|u2 dekrementiert das oberste Wort um 1
2*	x1 – x2 das oberste Wort wird mit 2 multipliziert
2/ --	x1 – x2 das oberste Wort wird durch 2 dividiert
AND	x1 x2 – x3 ersetze die beiden obersten Worte durch ihr logisches UND

1.1.3 Arithmetische und logische Operationen

Mnemonik	***Aktion***
CELL+	addr1 – addr2 addiert zu addr1 die Länge eines Wortes. Das Ergebnis ersetzt addr1
HALF+	addr1 – addr2 addiert zu addr1 die Länge eines Halbwortes. Das Ergebnis ersetzt addr1
INVERT	x1 – x2 invertiert das oberste Wort
INVERT	x1 – x2 invertiert das oberste Wort
LSHIFT	x1 u – x2 schiebe x1 um u Stellen nach links. Das Ergebnis ersetzt die beiden obersten Worte
NEGATE	x1 – x2 negiert das oberste Wort
OR	x1 x2 – x3 ersetze die beiden obersten Worte durch ihr logisches ODER
RSHIFT	x1 u – x2 schiebe x1 um u Stellen nach rechts, ohne Berücksichtigung des Vorzeichens. Das Ergebnis ersetzt die beiden obersten Worte
XOR	x1 x2 – x3 ersetze die beiden obersten Worte durch ihr EXKLUSIVES ODER
+B	n1|u1 n2|u2 – n3|u3 addiert die beiden obersten Worte mit dem Übertrag und ersetzt beide Worte durch diese Summe
-B	n1|u1 n2|u2 – n3|u3 ersetzt die beiden obersten Worte durch die Differenz n1 – n2 - Übertrag
LSHIFTC	x1 u – x2 schiebe x1 um u Stellen, mit Einbeziehung des Übertrags, nach links. Das Ergebnis ersetzt die beiden obersten Worte

Mnemonik	***Aktion***
RSHIFTC	x1 u – x2 schiebe x1 um u Stellen, mit Einbeziehung des Übertrags, nach rechts, ohne Berücksichtigung des Vorzeichens. Das Ergebnis ersetzt die beiden obersten Worte

Tabelle 3: Die arithmetisch logischen Befehle

Zur Unterstützung von Doppelwort- und Halbwortverarbeitung wurden Erweiterungen vorgenommen, die es im Standard nicht gibt.

1.1.4 Vergleichsoperationen

Die Befehle dieser Gruppe sind in Tabelle 4 angeführt.

Mnemonik	***Aktion***
0<!	n – n flag flag wird gesetzt, wenn n < 0, sonst gelöscht. Das flag wird auf den Stapel gelegt
0<	n – flag flag wird gesetzt, wenn n < 0, sonst gelöscht. Das flag ersetzt n auf dem Stapel
U0<	u – flag flag wird gesetzt, wenn der Übertrag gesetzt ist, sonst gelöscht. Das flag ersetzt u auf dem Stapel
U0>	u – flag flag wird gesetzt, wenn das vorzeichenlose u <> 0 und der Übertrag gelöscht ist, sonst gelöscht. Das flag ersetzt u auf dem Stapel
0>!	n – n flag flag wird gesetzt, wenn n > 0, sonst gelöscht. Das flag wird auf den Stapel gelegt
0>	n – flag flag wird gesetzt, wenn n > 0, sonst gelöscht. Das flag ersetzt n auf dem Stapel

1.1.4 Vergleichsoperationen

Mnemonik	***Aktion***
0=!	n – n flag flag wird gesetzt, wenn n == 0, sonst gelöscht. Das flag wird auf den Stapel gelegt
0=	n – flag flag wird gesetzt, wenn n == 0, sonst gelöscht. Das flag ersetzt n auf dem Stapel
0<>!	n – n flag flag wird gesetzt, wenn n != 0, sonst gelöscht. Das flag wird auf den Stapel gelegt
0<>	n – flag flag wird gesetzt, wenn n != 0, sonst gelöscht. Das flag ersetzt n auf dem Stapel

Tabelle 4: Die Vergleichsoperationen

Zur Effizienzsteigerung wurden Erweiterungen vorgenommen, die es im Standard nicht gibt. Grundsätzlich werden Relationen, die beide oberen Worte verwenden, mit zwei Befehlen realisiert. Als Erster „-“, wenn die Operanden nicht mehr benötigt werden, bzw. *CMP*, wenn die Operanden erhalten bleiben sollen, zur Bildung der Differenz. Es folgt *0<*, *0>*, *0=*, *0<>*, *U0<* oder *U0>* zur Auswertung der Differenz. Die FORTH-Befehle <, >, =, <>, *U>*, *U<* werden als Makros realisiert. Zusätzlich, um die Operanden zu erhalten, noch die Makros *<!*, *>!*, *=!*, *<>!*, *U>!*, *U<!*.

1.1.5 Programmflusssteuerungsbefehle

Mnemonik	***Aktion***
EXIT	R: addr -- ersetze den Befehlszähler durch die Rücksprungadresse addr, und nimmt sie vom Stapel
offset *CALL*	R: -- PC Ein Programm wird aufgerufen. Der momentane Wert des Befehlszählers PC auf dem Stack abgelegt, und zu PC offset addiert

Mnemonik	***Aktion***
addr *TRAP*	R: -- PC Ein Programm wird indirekt aufgerufen. Der momentane Wert des Befehlszählers PC auf dem Stack abgelegt, und PC durch den Inhalt des Wortes in addr ersetzt
offset *BRANCH*	Unbedingter Sprung, zum PC wird offset addiert
offset *0BRANCH!*	flag – flag hat flag den Wert 0, wird zum PC offset addiert
offset *0BRANCH*	flag – hat flag den Wert 0, wird zum PC offset addiert
core *SETPC*	PC -- initialisiert den PC des Kerns core. Dieser Befehl muss vor SWITCH ausgeführt werden
core *SWITCH*	-- Wechselt zum Kern core, was bedeutet, daß das aktuelle Stapelpaar sowie PC gewechselt werden.
GETCOREID	-- coreno Legt die ID des aktiven Kern auf den Stapel

Tabelle 5: Die Programmflusssteuerungsbefehle

Keiner der Befehle aus Tabelle 5 ist im Standard definiert, ihre Existenz wird aber vorausgesetzt. Nicht existentiell, aber modern, sind mehrere Kerne in einer CPU. Auch mehrere Kerne sollen unterstützt werden.

1.1.6 Sonstige Befehle

Mnemonik	***Aktion***
NOP	Produziert einen Verlängerungsschritt (stall)
HALT	Stoppt die Befehlsausführung; kann nur durch einen Interrupt aufgehoben werden
BREAK	Produziert einen Verlängerungsschritt (stall); nur für den Test der BIOS-Software vorgesehen

Tabelle 6: Sonstige Befehle

1.1.6 Sonstige Befehle

Die in Tabelle 6 angeführten Befehle gibt es im Standard nicht, sind aber nützlich.

Insgesamt sind es 99 Befehle! 33 davon sind aber redundant und dienen nur der Effizienzsteigerung, was die Länge des compilierten Codes um etwa 25% verkürzt.

1.1.7 Feststellungen

- Kein Befehl benötigt mehr als zwei Operanden
- Ein Befehl produziert genau ein oder gar kein Ergebnis
- Keine Angabe über die Wortbreite der Operanden
- Nur unmittelbare (immediate) Adressierung wird verwendet
- Eine *Load and Store*-Architektur ist impliziert

1.1.8 Festlegungen

- Unmittelbare Operanden sollen im Intel-Format (Little Endian – höherwertiges Byte auf der hohen Adresse, niederwertiges Byte auf der niederen Adresse) vorliegen
- Ein Bytecode ist für die Codierung der Befehle ausreichend
- Die Befehle sollen *general purpose*, also auf jeden Stapel anwendbar, sein
- Die Wortbreite soll 32 Bit betragen
- Der Befehlssatz soll problemlos erweiterbar sein

1.2 Eine Charakterisierung der Stapelmaschinen

Stapelmaschine ist nicht gleich Stapelmaschine. Zur genaueren Beschreibung wird auf markante Eigenschaften verwiesen, die eine Klassifizierung erlauben. Gebräuchlich sind folgende Merkmale [1, Kapitel 2.1, Seite 25 - 26]:

- Zahl der Stapel
- Größe des Stapelpuffers
- Zahl der Operanden einer Instruktion

Eine Maschine kann einen oder mehrere Stapel verwenden. Zu beachten ist, dass dieses Merkmal keine Angabe über die Realisierung der Stapel macht. Im einfachsten Fall handelt es sich bloß um spezielle Register, die die einzelnen Stapel im Adressraum lokalisieren. Eigene Maschinenbefehle unterstützen die Verwendung als Stapel. Es sei nur kurz festgehalten, dass es auch Computer gab oder gibt, die ohne Stapel auskommen, etwa VAX oder IBM 360/370.

Um den Durchsatz zu erhöhen, empfiehlt es sich die obersten Stapeleinträge in der Maschine zu puffern. Dies sollte so sein, muss aber nicht sein, ein Vertreter ungepufferter Stapel wäre der MC68000. Der Vorteil in der Pufferung ist darin zu sehen, dass die gepufferten Einträge ohne zusätzliches Zugreifen auf den Stapel verarbeitet werden können. Lediglich das Rückschreiben modifizierter Einträge muss nicht nur im Puffer, sondern auch am Stapel erfolgen. Eine weitere Verbesserung kann durch Verwendung eines Cache zwischen Stapel und Puffer erzielt werden, die Ersetzungsstrategie muss aber *Write Through* sein – also unmittelbare Ersetzung nicht nur im Cache, sondern auch

im Stapel. Alternativ zum Cache ist auch die Verwendung von lokalem Speicher möglich, der die Spitze des Stapels beherbergt. Erst wenn die Größe des Stapels über die Größe dieses Speichers hinaus wächst, wird Arbeitsspeicher vom Stapel in Anspruch genommen. Dieser lokale Speicher muss der CPU direkt zugänglich sein, ohne die Systembusse in Anspruch zu nehmen, um eine Verbesserung darzustellen. Cache und lokaler Speicher sind aber nur zusätzliche Merkmale, die bei einer Klassifizierung einer Architektur nicht außer Acht gelassen werden sollten.

Es macht einen Unterschied, ob die Maschine eine 0-, 1- oder 2-Adressmaschine repräsentiert. Während bei einer 0-Adressmaschine alle Operanden implizit am Stapel liegen, und nicht explizit angegeben werden müssen, können bei 1- und 2-Adressmaschinen zusätzliche Operanden, etwa Register oder Speicherinhalte, in eine Operation eingebracht werden. Zusätzliche Operanden müssen bereitgestellt werden, was nicht zeitlos erfolgen kann, während 0-Adressmaschinen sofort beginnen können den Befehl auszuführen. Allen Maschinen ist gemeinsam, das Ergebnis auf den Stapel zu legen. Bei 1-Adressmaschinen wird implizit als zweiter Operand das oberste Element des Stapels verwendet, sofern der Befehl einen weiteren Operanden benötigt.

Es sei bemerkt, dass Literale und Konstante, die dem Operationscode von Ladebefehlen oder Unterprogrammaufrufen angehängt sind, auch bei 0-Adressmaschinen möglich sind.

1- und 2-Adressmaschinen benötigen im allgemeinen weniger Befehle für eine Berechnung als eine 0-Adressmaschine, die Codierung der Instruktionen ist aber länger, eine 0-Adressmaschine kommt mit einem Bytecode aus.

1.3 Festlegung auf den Typ (2, 2, 0)

Als Maschine der Wahl genügt eine 0-Adressmaschine. Per Definition erwartet FORTH zwei Stapel – den Datenstapel für Berechnungen und Parameterübergabe, und einen Stapel für die Rücksprungadressen der aufgerufenen Unterprogramme.

Befehle verwenden höchstens die beiden obersten Elemente, somit genügt ein Stapelpuffer der Länge zwei für jeden der beiden Stapel.

Ein Stapel selbst soll aus einer Spitze bestehen, die in einem lokalen Speicher liegen soll. Zusätzlich einem Adressregister, das auf einen Speicherbereich verweist, die den Überlauf der Spitze aufnimmt, oder bei Unterlauf daraus die Spitze nachladet. Diese Operationen modifizieren das Adressregister selbstständig. Alternativ einen Cache zu verwenden, wird ausgeschlossen, da die Ersetzungsstrategie *Write Through* sein müsste, und somit auf jeden Fall einen Speicherzugriff über den Systembus nötig machen würde. Lokaler Speicher benötigt einen Solchen nicht.

Da zu einem Zeitpunkt immer nur mit einem Stapel gearbeitet werden kann – FORTH kennt keinen Befehl der gleichzeitig auf beiden Stapel arbeitet, genügt eine Stapelverwaltung um beide Stapel zu handhaben. Die Information, auf welchem Stapel gearbeitet wird, muss vom Befehlsdecoder kommen.

1.4 Die Stapelorganisation

Es wird ein lokaler Speicher verwendet, der als Ringpuffer organisiert ist, der einen Stapel beherbergt [1, Kapitel 3.2.1.2, Seite 33 - 34]. Kenngrössen des Stapels sind die Position des ersten und des letzten Elements im Speicher. Sollte dieser Stapel überlaufen, ist das

letzte Element in einen Rettungsbereich im Hauptspeicher auszulagern, umgekehrt bei einem Unterlauf aus dem Rettungsbereich einzulagern. Kenngröße für den Rettungsbereich ist ein Zeiger, der eine definierte Hauptspeicheradresse enthält. Dieser Zeiger muss bei jeder Lese- oder Schreiboperation modifiziert werden, sodass er immer auf das zuletzt gerettete Datum zeigt. Der Rettungsbereich ist somit selbst wieder ein Stapel.

1.4.1 Der lokale Speicher

Vorzugsweise ist seine Größe eine Potenz von 2 – man erspart sich eine Modulooperation bei der Modifikation der Stapelkenngrössen. Mit Simulationen [1, Kapitel 6.4.1, Seite 139 - 143] wurde ermittelt, dass eine Größe von 64 für die Abarbeitung von arithmetischen Ausdrücken völlig ausreichend ist. Da der Datenstapel nicht nur zur Auswertung von Ausdrücken, sondern auch zur Parameterübergabe verwendet wird, schlage ich als minimale Größe 64 vor, damit die Zahl der Über- und Unterläufe auf jeden Fall niedrig gehalten werden kann.

Indexf zeigt auf das oberste Element im Stapel. Zahlreiche Befehle benötigen auch das zweite Element, das über *Indexs* erreicht wird. Zwecks rascher Rettung bei einem Überlauf, muss auch das unterste Element stets verfügbar sein – indiziert durch *Indexl*. Mit *Indexr* soll es auch möglich sein wahlfrei auf jedes Element im Stapel lesend zuzugreifen. *Indexi* identifiziert eine beliebig wählbare Position im Stapel für Schreiboperationen. Es sei nur erwähnt, dass *Indexi* ein Freigabesignal benötigt, zwecks Übersichtlichkeit wurde es weggelassen. Wahlfreies Lesen erfolgt über den Befehl *PICK*, der als Adresse den Inhalt des obersten Stapelelements verwendet. Der Inhalt des adressierten Elements ersetzt das oberste Stapelelement. Die Größe des Stapels ändert sich also nicht, es kann kein Überlauf stattfinden. Daher werden die Indizes *Indexl* und *Indexr* zu einer Kenngröße zusammengefasst, es ist aber ein Steuersignal nötig um den aktuell gewünschten Index auszuwählen. *Indexf*, *Indexs* benötigen keine Steuersignale, sie sind stets aktiv.

Die ausgelesenen Daten *First*, *Second*, *Random* bilden Eingangswerte für den Stapelpuffer, *Last* wird bei Überlauf sofort in den Hauptspeicher geschrieben, ansonsten ignoriert.

Der lokale Speicher muss also über vier unabhängige Ports verfügen, drei zum gleichzeitigen Lesen, einen zum gleichzeitigen Schreiben. Da es sich um einen Speicher handelt, sind zum Lesen zwei Takte nötig, im Ersten werden die Adressen angelegt, erst im zweiten Takt sind die Daten bereitgestellt. Schreiben benötigt nur einen Takt, zu beachten ist, dass der neue Wert erst beim übernächsten Takt erfolgreich ausgelesen werden kann. Diese Randbedingungen sind beim Entwurf des Stapelpuffers zu beachten.

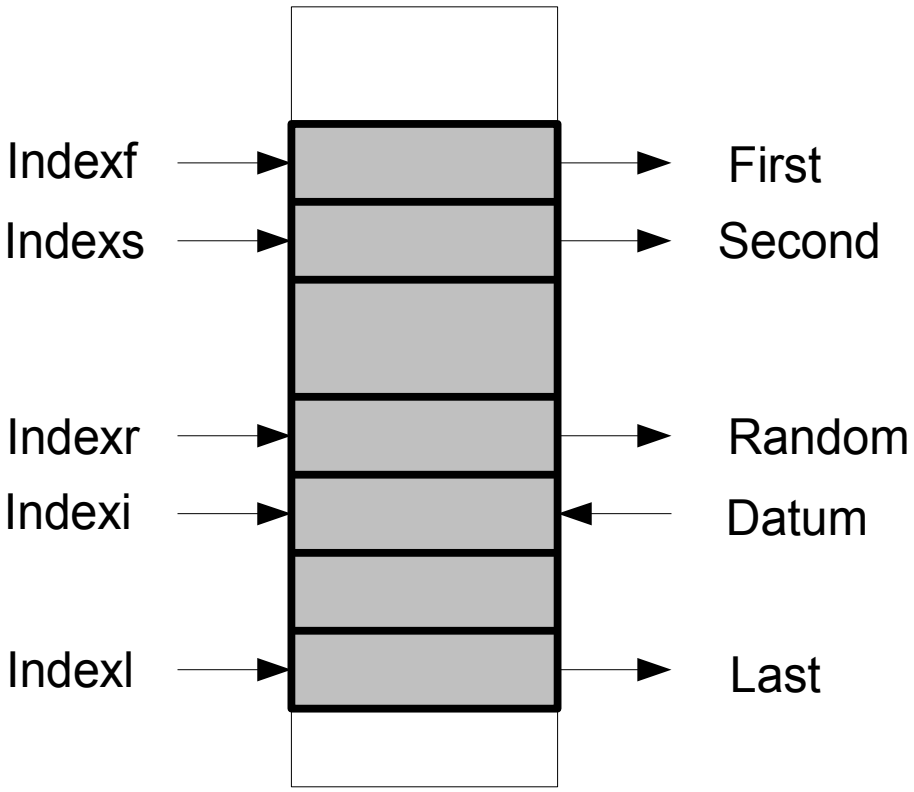


Abbildung 1: Der Stapel

1.4.2 Die Stapelpuffer

Folgende Vereinbarungen gelten:

- t der Zeitpunkt, zu dem die Kenngrössen aktualisiert werden
- t+1 der Zeitpunkt, zu dem der Stapel indiziert wird
- t+2 der Zeitpunkt, zu dem die Werte gepuffert und verarbeitet werden
- t+3 der Zeitpunkt, zu dem neue Werte geschrieben werden, und der Stapel puffer aktualisiert wird
- top der aktive Stapelpuffer
- buffer(2,2) die Matrix der gepufferten Einträge
- p(2,2) der Schattenspeicher der Stapelpuffer
- select die Nummer des ausgewählten Stapels
- datum ein neuer Wert
- random wahlfrei gelesener Stapeleintrag
- first oberster Stapeleintrag
- second zweiter Stapeleintrag
- last der Wert des untersten Stapeleintrags
- fast(2) ist wahr, wenn ein neues Datum zur Übernahme bereit steht
- early(3) ist wahr, wenn das vorletzte gespeicherte Datum der aktuellste Wert ist
- cur(3) ist wahr, wenn das zuletzt gespeicherte Datum der aktuellste Wert ist

Weiter sei „|“ das Trennzeichen alternativer Werte, die linke Alternative hat immer höhere Priorität gegenüber der Rechten.

Dann gilt für den Stapelpuffer, der aktuell bearbeitet wird:

$$top_{t+1} \leftarrow buffer_{t+1}(select_{t+2})$$

Anders ausgedrückt, es wird auf dem Stapel gearbeitet, dessen Kenngrössen zuletzt modifiziert wurden, seine aktuellen Datenwerte liegen im Schattenspeicher *p* bereit.

Das letzte Element *last* repräsentiert entweder den soeben auf diese Stelle geschriebene Wert oder den dort bereits vorhandenen Wert.

$$last_{t+3} \leftarrow datum_{t+2,\ cur(2)\ =\ true} \mid datum_{t+1,\ early(2)\ =\ true} \mid random_{t+2}$$

Ähnlich verhält es sich mit den Stapelpuffern, sowie dem Schattenspeicher.

$$buffer_{t+3}(select_{t+3},\ 0) \leftarrow datum_{t+3,\ fast(0)\ =\ true} \mid datum_{t+2,\ cur(0)\ =\ true} \mid datum_{t,+1\ early(0)\ =\ true} \mid first_{t+1}$$

$$buffer_{t+3}(select_{t+3},\ 1) \leftarrow datum_{t+3,\ fast(1)\ =\ true} \mid datum_{t+2\ cur(1)\ =\ true} \mid datum_{t,+1\ early(1)\ =\ true} \mid second_{t+1}$$

$$buffer_{t+3}(select_{t+3} + 1) \leftarrow p_{t+2}(select_{t+3} + 1)$$

$p_{t+3}(select_{t+3}) \leftarrow buffer_{t+3}(select_{t+3})$

Obige Berechnungen nehmen von *fast* und *early* deren Werte zum Zeitpunkt t+2. Die Stapelpuffer sind damit spezifiziert, die Funktionen *fast* und *early* lauten:

$early_{t+1}(0) \leftarrow$ true, wenn $indexf_{t+1} = indexi_{t+1}$

$early_{t+1}(1) \leftarrow$ true, wenn $indexs_{t+1} = indexi_{t+1}$

$early_{t+1}(2) \leftarrow$ true, wenn $indexl_{t+1} = indexi_{t+1}$

$cur_{t+2}(0) \leftarrow$ true, wenn $indexf_{t+2} = indexi_{t+2}$

$cur_{t+2}(1) \leftarrow$ true, wenn $indexs_{t+2} = indexi_{t+2}$

$cur_{t+2}(2) \leftarrow$ true, wenn $indexl_{t+2} = indexi_{t+2}$

$fast_{t+3}(0) \leftarrow$ true, wenn $indexf_{t+2} = indexi_{t+3}$

$fast_{t+3}(1) \leftarrow$ true, wenn $indexs_{t+2} = indexi_{t+3}$

In der Praxis ergeben sich die Werte für *fast* ganz nebenbei. Jede Operation, die ein Ergebnis auf den Stapel legt, bedingt implizit „$indexf_{t+2} = indexi_{t+3}$“. Zur Erläuterung: Zum Zeitpunkt *t* wird die Position $indexf_{t+1}$ des obersten Stapeleintrags neu ermittelt. Das Ergebnis der Operation zum Zeitpunkt *t+2* wird der neue Wert $datum_{t+3}$, der am Stapel an der Stelle $indexi_{t+3}$ zum Zeitpunkt *t+3* geschrieben werden soll. Es muss also die Gleichheit„$indexf_{t+2} = indexi_{t+3}$“ gelten. Eine Restaurierungsoperation, die durch Einlagern aus dem Überlaufbereich das erste bzw. zweiten Element, von unten her, aktualisiert, setzt für das erste Element „fast(0) = true“ bzw. für das Zweite „fast(1) = true“.

Die physische Realisierung der Stapelpuffer ist der Schattenspeicher *p*, der Stapelpuffer selbst ist eine Prioritätsfunktion, die die aktuellsten Werte von Schattenspeicher, aktuellem Stapelinhalt und letzten Ergebnissen auswählt.

1.5 Festlegung des Prozessorkerns

Ein Befehl durchläuft folgende in Tabelle 7 angeführten Phasen.

Bezeichnung	***Einheit***	***Beschreibung***
laden	Prefetch queue	Entnimmt den nächsten Befehl und den eventuell angehängten unmittelbaren Wert
dekodieren	Befehlsdekoder	Stellt die für den Befehl notwendigen Steuerinformationen bereit
aktualisieren	Stapelverwaltung	Aktualisiert die Kenngrössen des gewählten Stapel
vorbereiten	Stapelpuffer	Aktualisiert den selektierten Stapelpuffer
verarbeiten	Stapelpuffer, ALU, Speicher	Wertet Daten und Steuerinformationen aus
schreiben	Stapelverwaltung	Legt ein eventuell erzeugtes Datum auf den Stapel und in den Stapelpuffer

Tabelle 7: Die Phasen eines Befehls

Jede dieser Phasen arbeitet überschneidungsfrei mit den anderen Phasen, jede Phase sollte mit einem Operationsschritt auskommen. Phasenpipelining ist daher möglich, das bedeutet, dass mit jedem Takt ein Befehl vollständig abgearbeitet ist, obwohl jeder Befehl zumindest sechs Operationsschritte benötigt. Jede Phase kann zusätzliche Operationsschritte erzwingen, muss aber alle anderen Phasen darüber informieren, dass diese möglicherweise einen Verlängerungsschritt (stall) ausführen müssen.

1.5.1 Prefetch queue

Diese Einheit liest immer ganze Worte aus dem Speicher in eine Warteschlange, aus welcher sie die Befehle, mit unmittelbarem Wert, sofern gefordert, sequentiell entnimmt und an den Decoder weiterleitet. Vorteil dieser Strategie ist die Minimierung der Speicherzugriffe, da die meisten Befehle mit einem einzigen Byte auskommen, und mit einem Zugriff bis zu vier Befehle gelesen werden. Kann die Warteschlange mehr als ein Wort aufnehmen, sind auch lange Befehle, TRAP etwa benötigt fünf Byte, meist ohne Verlängerungsschritt für das Laden der zusätzlichen Byte ausführbar, da diese Einheit die Warteschlange immer gefüllt hält. Wird der Befehlszähler durch einen Sprungbefehl oder Programmaufruf neu gesetzt, muss auch die Warteschlange neu gefüllt werden, was als Nachteil zu sehen ist, da Verlängerungsschritte nötig werden.

Diese Einheit soll einen eigenen Daten- und Adressbus haben.

1.5.2 Befehlsdecoder

Er ist nur ein Umsetzer, dessen Beschreibung erst nach der Implementierung der Stapelverwaltung möglich ist.

1.5.3 Stapelverwaltung

Aktualisieren und Schreiben wurden bereits festgelegt, bleiben noch Festlegungen zur Verarbeitung zu treffen. Die ALU steht allein der Stapelverwaltung zur Verfügung, sie wird von dieser voll ausgelastet. Weiters muss es einen eigenen Daten- und Adressbus für Speicherzugriffe der Verarbeitung geben, diese sollen eine höhere Priorität als die der *Prefetch queue* haben.

1.6 Der Prozessor

Er soll den Kern und die nachfolgend angeführten Funktionsgruppen mit einem Busmanager synchronisieren und integrieren, und den Anschluss externer Einheiten, sowie des Speichers, ermöglichen.

1.6.1 UART

Die Kommunikation mit einem anderen Rechnern soll über eine serielle Schnittstelle (RS232) möglich sein, der UART ist die Brücke zwischen Schnittstelle und dem Prozessor. Eigenschaften des UART:

- getrennte Sender und Empfänger, beide fähig ein Unterbrechungssignal zu erzeugen
- Datenwort 7 oder 8 Bit breit
- Stoppbits 1, 1.5 oder 2
- Parität none, even oder odd
- Baudrate frei wählbar

Die Ports sind *memory mapped*

1.6.2 Counter und Timer

Vier unabhängige Abwärtszähler, jeder 24 Bit breit, jeder mit einem externen Zähleingang, außerdem fähig eine Unterbrechung zu erzeugen, sollen verfügbar sein. Die Ports sind *memory mapped*.

1.6.3 Interrupt Controller

Er soll 16 maskierbare Unterbrechungssignale verwalten und gegebenenfalls einen Unterbrechungsvektor an den Kern senden. Die Vektoren, in Tabelle 8 angeführt, belegen die ersten 16 Worte des Adressraums (0 – 63), ein Vektor enthält die Adresse seiner Unterbrechungsroutine.

1.6.3 Interrupt Controller

Unterbrechung	***Belegung***
0	Undefinierter Befehl
1	Software Unterbrechung
2	Extern 0
3	Extern 1
4	Extern 2
5	Extern 3
6	Extern 4
7	Extern 5
8	Extern 6
9	Extern 7
10	Zähler 0
11	Zähler 1
12	Zähler 2
13	Empfänger UART
14	Sender UART
15	Zähler 3

Tabelle 8: Die Unterbrechungssignale

Die Ports sind ebenfalls *memory mapped*.

1.6.4 ROM

Es soll das BIOS des Prozessorsystems aufnehmen und eine Größe von 4096 Worten haben. Auf das ROM kann nur lesend, immer ganze Worte, zugegriffen werden. Seine Adresslage soll unmittelbar an den Adressbereich des RAM anschließen.

1.6.5 Schnittstelle zur Außenwelt

Der Prozessor soll ein asynchrones SRAM oder ein synchrones SDRAM ansteuern können.

1.6.6 Das Blockschaltbild des Prozessors

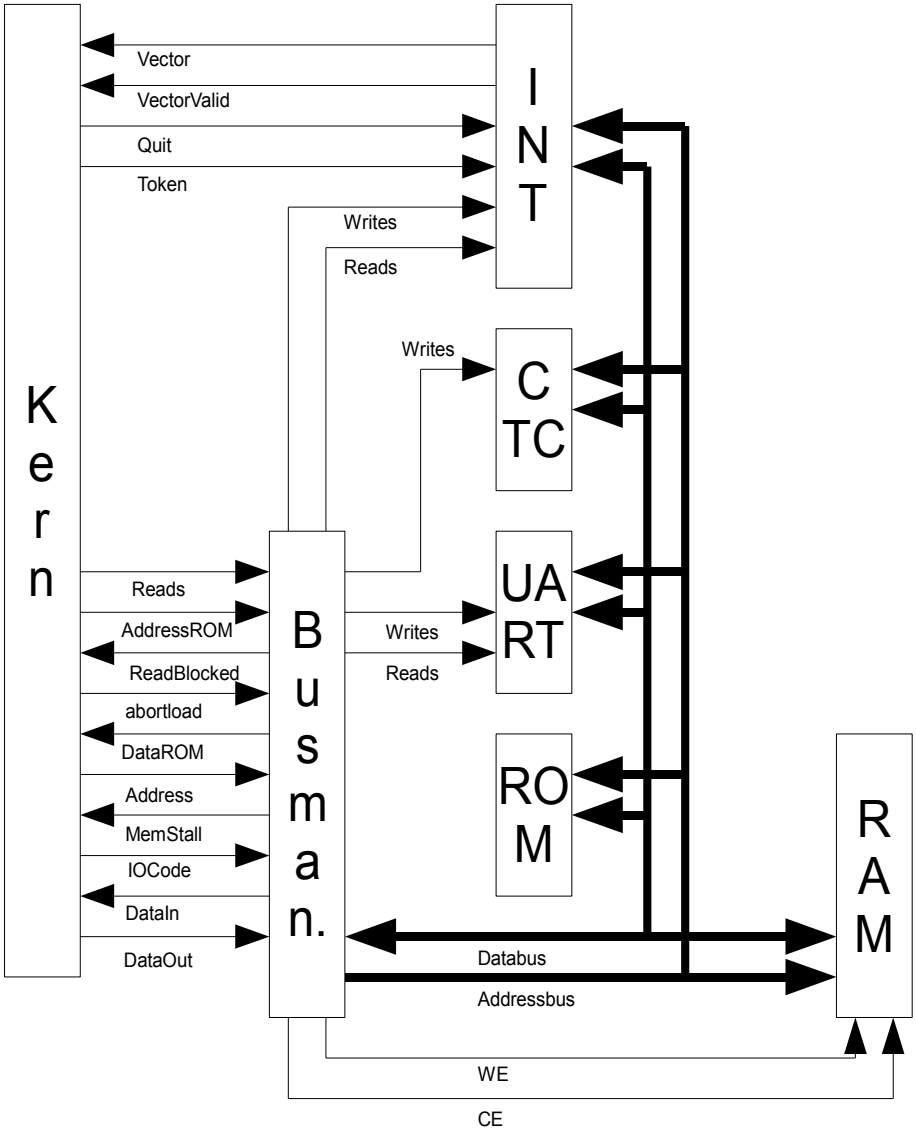


Abbildung 2: Das vereinfachte Blockschaltbild des Prozessors

Nicht berücksichtigt wurden die Unterbrechungssignale. Der Block RAM repräsentiert ein statisches SRAM. Das ROM dekodiert seinen zugeteilten Adressbereich selbständig und braucht daher kein Selektionssignal CE. Der Datenbus ist eine grobe Vereinfachung. In der Implementierung sind die Datenleitungen der Komponenten nicht zusammengeführt, sondern jede Komponente direkt an den Busmanager angeschlossen. Selbes gilt auch für den Addressbus.

1.6.7 Abschließende Bemerkungen

Der Prozessor soll ein Beweis dafür sein, dass RISC-Architekturen nicht auf

1.6.7 Abschließende Bemerkungen

Registermaschinen beschränkt sind, auch Stapelmaschinen kommen durchaus in Frage. Im Gegensatz zur klassischen Harvard-Architektur, wo Programm- und Datenspeicher streng getrennt sind, soll es nur einen Adressraum geben, wie bei von Neumann-Architekturen üblich. Zu bemerken ist auch die Befehlsabarbeitung, die einem von Neumann-Zyklus entspricht.

2 Implementierung des Prozessorkerns

Die Schnittstelle:

```
entity theCore is
  generic (constant CacheIndexBitWidth: integer := 10;-- ld(Cachesize)
           constant TableBitWidth : integer := 4);    -- ld(Interrupttable)
  port (nReset: in std_ulogic;                        -- system reset
        Clock: in std_ulogic;                         -- system clock
        abortload: out std_ulogic;                    -- abort memory cycle
        MemStall: in std_ulogic;                      -- memory stall
        DataIn: in DataVec;                           -- incoming data
        ReadBlocked: in std_ulogic;                   -- memory fetch blocked
        DataROM: in std_ulogic_vector(DataVec'range);-- incomming ROM code
        AddressROM: out std_ulogic_vector(RAMrange'high - 1 downto 0); -- ROM
                                                                        address
        Reads: out std_ulogic;                        -- read ROM
        Address: out std_ulogic_vector(RAMrange'high + 1 downto 0);    -- memory
                                                                        address
        DataOut: out DataVec;                         -- data to memory
        IOCode: out std_ulogic_vector(2 downto 0);    -- memory operation
        illegal: out std_ulogic;                      -- illegal opcode
        Token: out std_ulogic;                        -- service routine running
        Quit: out std_ulogic;                         -- vector processed
        Vector: in std_ulogic_vector(TableBitWidth - 1 downto 0);-- interrupt
                                                                   vector
        VectorValid: in std_ulogic);                  -- vector valid
end theCore;
```

Quelle	***Kommentar***
theCore.vhd	Integriert alle folgenden Quellen und implementiert den Befehls-dekoder
ALU.vhd	Implementiert die ALU
theStacks.vhd	Implementiert die Stapelverwaltung und integriert die ALU
ProgramCounter.vhd	Implementiert die *Prefetch Queue*
global.vhd	Globale Deklarationen

Tabelle 9: Die Quelldateien des Kerns

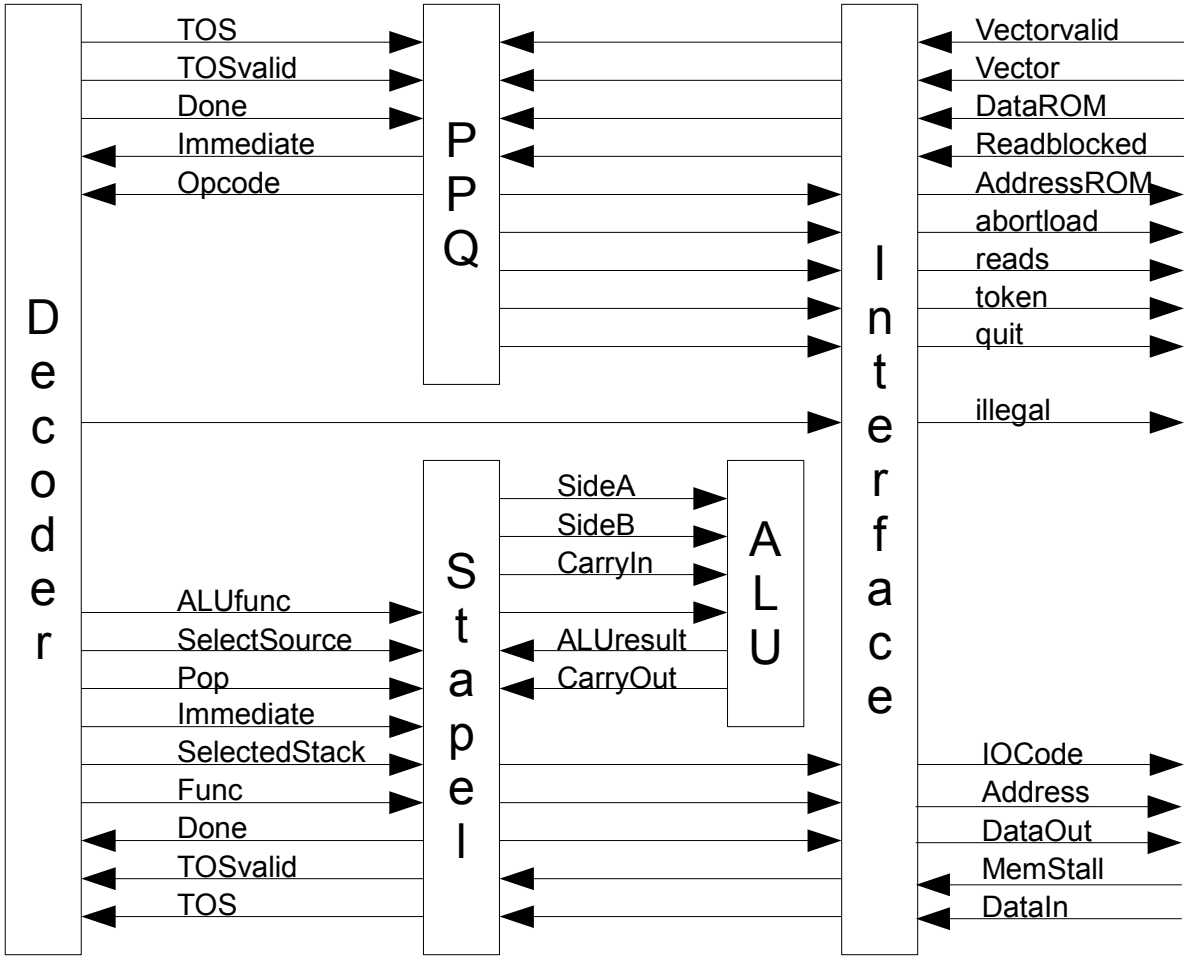


Abbildung 3: Das Blockschaltbild des Kerns

Nachfolgend werden die einzelnen Funktionsgruppen des Kerns vorgestellt.

2.1 Die ALU

Die Schnittstelle:

```
entity ALU is
      port (nReset: in std_ulogic;                    -- reset
            SideA: in DataVec;                        -- first Operand
            SideB: in DataVec;                        -- second Operand
            AluResult: out DataVec;                   -- result
            CarryIn: in std_ulogic;                   -- Carry in
            CarryOut: out std_ulogic;                 -- Carry out
            AluFunc: in AluFuncType);                 -- Opcode
```

```
end entity ALU;
```

Die eingehenden Operanden, sowie der eingehende Übertrag, und die Steuerinformation *Afunc* werden von vier Prozessen gleichzeitig ausgewertet. Die beiden höchstwertigen Bit der Steuerinformation selektieren aus den Teilergebnissen, Tabelle 10, das tatsächlich verlangte Ergebnis.

Prozess	***Aufgabe***
arith	Addition bzw. Subtraktion
relop	die Relation von *SideA*, *CarryIn* bezüglich 0
shifts	Verschiebt *SideA* um *SideB* Stellen nach links bzw. rechts
logic	Führt boolesche Operationen aus oder eine Multiplikation

Tabelle 10: Die Funktionsgruppen der ALU

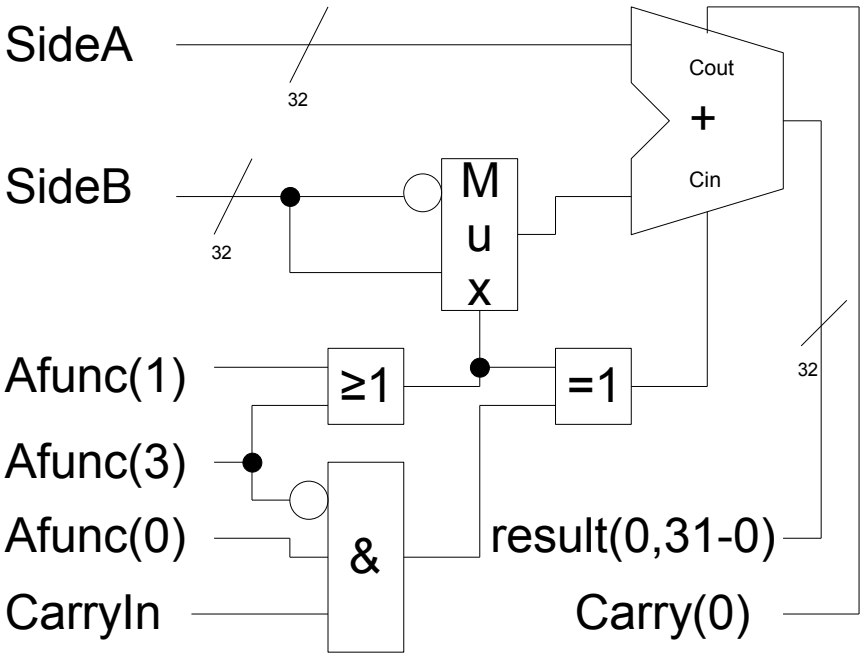


Abbildung 4: Der Addierer/Subtrahierer

Im Bild dargestellt ist der Addierer/Subtrahierer. *Afunc(1)* und *Afunc(3)* geben an, ob es sich um eine Addition oder Subtraktion handelt. Bei Subtraktionen wird der zweite Operand negiert addiert. *Afunc(3)* und *Afunc(0)* geben an ob *CarryIn* verwendet wird oder nicht. Der so ermittelte Eingangsübertrag für den Addierer wird bei einer Subtraktion invertiert angelegt. Die Ergebnisse sind *result(0)* und *Carry(0)*.

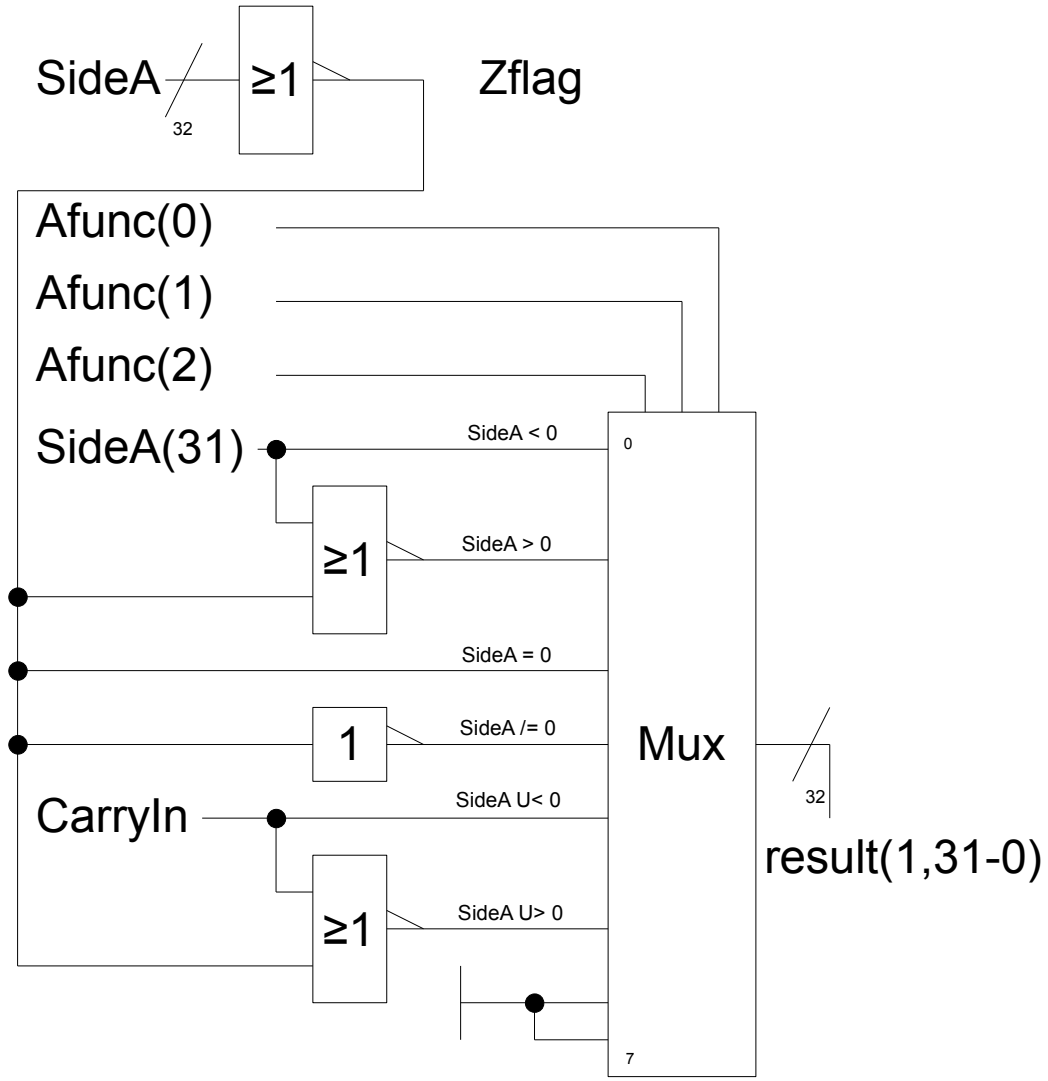


Abbildung 5: Vergleicher relop

Bei Vergleichsoperationen wird aus dem Msb von *SideA*, *CarryIn* und der Hilfsgröße *Zflag* (wahr, wenn *SideA* = 0), das Vergleichsergebnis ermittelt, -1 für wahr, 0 für falsch.

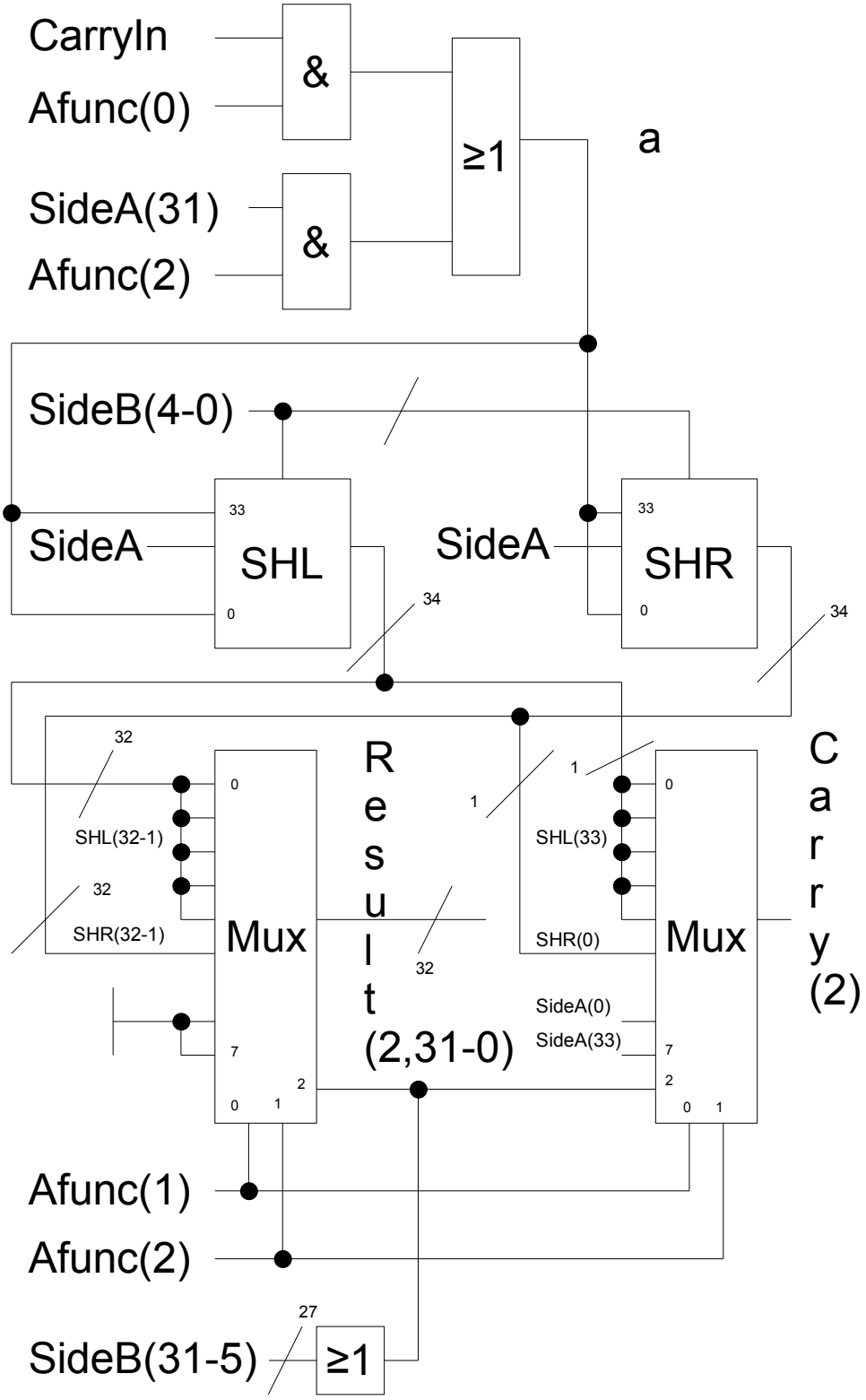


Abbildung 6: Die Schieberegister

Die Schiebeoperation verwendet ein rechtsseitiges und ein linksseitiges Schieberegister.

Als *CarryOut* ergibt sich jenes Bit, das zuletzt aus dem gewählten Schieberegister geschoben wurde.

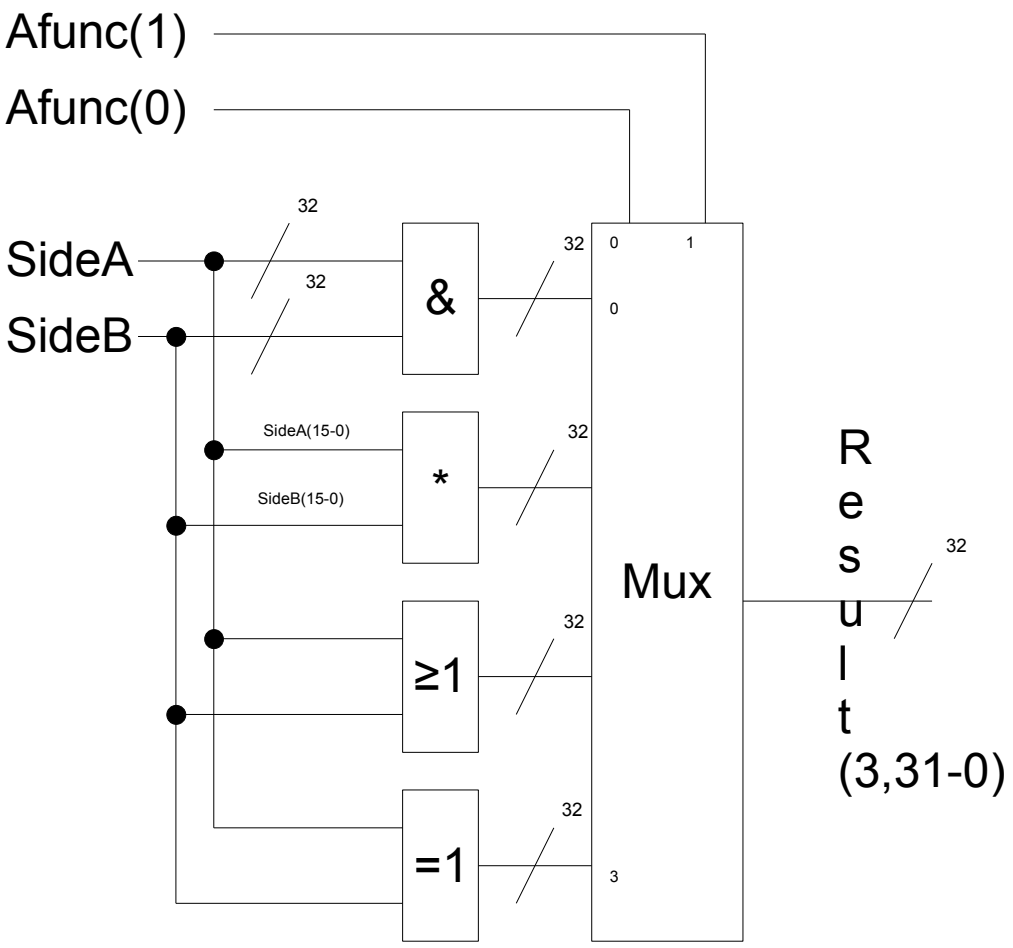


Abbildung 7: Logik und Multiplizierer

Die logischen Operationen sind selbsterklärend. Die Operation *not* wurde nicht extra realisiert, sie ist äquivalent zu

not SideA = SideA xor -1

Die vorzeichenlose Multiplikation verarbeitet 16-Bit Operanden und liefert ein 32-Bit Produkt. Sie benötigt bloß einen Takt.

Die bereitgestellten Operationen sind in den folgenden Tabellen 11 bis 14 angeführt.

Addierer/Subtrahierer	***Operation***	***Übertrag***	***Kommentar***
AddALU (00100)	SideA + SideB	berechnet	
SubALU (00110)	SideA - SideB	berechnet	
AddcALU (00101)	SideA + SideB + CarryIn	berechnet	
SubcALU (00111)	SideA - SideB - CarryIn	berechnet	
AddALUnC (00000)	SideA + SideB	CarryIn	Adressrechnung
SubALUnC (00010)	SideA - SideB	CarryIn	Adressrechnung

Tabelle 11: Addition/Subtraktion

Vergleicher	***Operation***	***Übertrag***	***Kommentar***
LtALU (01000)	SideA < 0	CarryIn	Msb
GtALU (01001)	SideA > 0	CarryIn	Msb nor Zflag
EquALU (01010)	SideA = 0	CarryIn	Zflag
NequALU (01011)	SideA /= 0	CarryIn	Not Zflag
UltALU (01100)	SideA < 0	CarryIn	Carry
UgtALU (01101)	SideA > 0	CarryIn	Carry nor Zflag

Tabelle 12: Relationen

Schiebeoperation	***Operation***	***Übertrag***	***Kommentar***
LshiftALU (10000)	SideA << SideB	berechnet	
LshiftCALU (10001)	SideA, CarryIn << SideB	berechnet	Mit Übertag
RshiftALU (10010)	SideA >>> SideB	berechnet	
RshiftCALU (10011)	CarryIn, SideA >>> SideB	berechnet	Mit Übertag
RshiftAALU (10100)	SideA >> SideB (:= 1)	berechnet	Arithmetisches Shift

Tabelle 13: Schiebeoperationen

Logische Operation	***Operation***	***Übertrag***	***Kommentar***
AndALU (11000)	SideA & SideB	gelöscht	
MultALU (11001)	SideA * Sieb	gelöscht	
OrALU (11010)	SideA | SideB	gelöscht	
XorALU (11011)	SideA ^ SideB	gelöscht	

Tabelle 14: Logische Operationen und Multiplikation

2.2 Die Stapelverwaltung

Die Schnittstelle:

```
entity theStacks is
   generic (constant CacheIndexBitWidth: integer := 10);    -- log. of depth of
                                                               cache
   port (nReset: in std_ulogic;                             -- Reset
         Clock: in std_ulogic;                              -- clock signal
         MemStall: in std_ulogic;                           -- memory stall
         Pop: in std_ulogic_vector(1 downto 0);             -- pop count
         SelectSource: in std_ulogic_vector(3 downto 0);    -- select a visible
                                                            element of the stacks
         SelectedStack: in BinRange;                        -- selects the
                                                               target stack
         DataIn: in DataVec;                                -- datum entering
         DataOut: out DataVec;                              -- datum leaving
         Immediate: in DataVec;                             -- immediate value
         Target: out std_ulogic_vector(RAMrange'high + 1 downto 0); -- memory
                                                                       address
         IOCode: out std_ulogic_vector(2 downto 0);         -- memory operation
         TOSValid: out std_ulogic;                          -- return address is
                                                               on top of stack
         Ready: out std_ulogic;                             -- stack ready for
                                                               next operation
         Func: in StackFuncType;                            -- Opcode Stack
```

```
        First: out DataVec;                              -- first operand
                                                            on stack
        AFunc: in AluFuncType;                           -- Opcode ALU
        SelectALU: in InputOrder_t);                     -- input selection
                                                            for ALU
end entity theStacks;
```

2.2.1 Benötigte Kenngrössen eines Stapels

Etliche Kenngrössen beschreiben einen Stapel, sind zu seiner Referenzierung nötig, bzw. erlauben einen Schluss auf seinen aktuellen Zustand.

- Top der Index des obersten Stapelelement (korrespondiert mit indexf)
- Tail der Index des untersten Stapelelement (korrespondiert mit indexl)
- Append der Index in den freien Bereich nach Tail
- cachelength die aktuelle Länge des Stapels im lokalen Speicher
- reloadstate aktueller Zustand des Überlaufautomaten
- stackptr zeigt auf das neueste Element im Überlaufbereich
- stacklength die aktuelle Länge des Überlaufbereichs

Diese Variable werden im ersten Operationsschritt (Aktualisierung des Stapels) neu berechnet, im zweiten Operationsschritt (Verarbeitung des Stapelpuffers) gegebenenfalls bloß gelesen. Weitere Operationsschritte benötigt der Stapel nicht. Beide Operationsschritte können aber unabhängig voneinander Verlängerungsschritte (stalls) erzwingen.

2.2.2 Aktualisierung des Stapels

Folgende Informationen werden vom Befehlsdecoder erwartet:

- $select_{t-1}$ die Nummer des gewählten Stapels
- $stackfunc_{t-1}$ die Operation, die auf den Stapel angewendet werden soll
- pop_{t-1} die Anzahl der obersten Einträge, die vom Stapel zu löschen sind
- $selectalu_{t-1}$ Information, welche Werte an welche Eingänge der ALU zu legen sind
- $afunc_{t-1}$ die Operation, die die ALU ausführen soll

Weitergegeben werden die modifizierten Kenngrössen, sowie die Information, ob ein Verlängerungsschritt (stall1) notwendig ist.

Die Aktualisierung wird nur durchgeführt, wenn die Verarbeitung keinen Verlängerungsschritt benötigt. Sollte die verlangte Stapeloperation *SetSPStack* sein, wird eine Initialisierung des Stapels erzwungen. Im obersten Stapelelement wird die Adresse des Überlaufbereichs erwartet, im zweiten Element die aktuelle Länge des Überlaufbereichs, der Stapel im lokalen Speicher wird auf leer gesetzt. Für alle anderen Stapeloperationen wird *cachelength* modifiziert und *Top* neu ermittelt. Das obere Ende des Stapels wäre damit auf neuestem Stand, fehlt noch das untere Ende.

Liegt ein Überlauf vor, oder die Stapeloperation *SaveStack*, die eine Rettung des Stapels in den Überlaufbereich erzwingt, wird *stackptr* vermindert und *stacklength* erhöht, das Schreiben von *last* angefordert. Die Kenngrössen *Tail*, *Append*, *cachelength* werden modifiziert und der Zustand von *reloadstate* auf *inhibit* gesetzt, im Fall von *SaveStack* können auch zusätzliche Verlängerungsschritte erzwungen werden.

Liegt kein Überlauf vor, so ist der Automat, Tabelle 15, auf eine Zustandsänderung zu prüfen.

Zustand	***Aktion***
check	wird gerade kein Verlängerungsschritt ausgeführt, ist der Überlauf nicht leer und sind im lokalen Speicher wenigstens zwei Einträge frei, erzwinge einen Verlängerungsschritt und wechsle in den Zustand *load*
load	leitet das Lesen des jüngsten Element des Überlaufbereichs ein und modifiziert die betroffenen Kenngrössen. Enthält der lokale Speicher wenigstens zwei Einträge, wird in den Zustand *check* gewechselt, ansonsten wird ein weiterer Verlängerungsschritt erzwungen. Ist der lokale Speicher nicht leer, wird in den Zustand *delay* gewechselt
delay	ein Verzögerungsschritt, der neue Zustand wird *check*
inhibit	ist der aktuelle Befehl DROP bzw. 2DROP, wird in den Zustand *check* gewechselt. Dieses Verhalten soll dem Programmierer ein problemloses wechseln von logischen Stapeln ermöglichen

Tabelle 15: Die Zustände des Überlaufautomaten

Als Letztes werden die neuen Steuerdaten an die ALU weitergegeben, aber nur wenn kein Verlängerungsschritt ausgeführt wird. Ein Verlängerungsschritt (stall1) wird erzwungen, wenn *stackfunc* die Werte *GetPut*, *LoadStore* oder *SetSPStack* hat.

2.2.3 Die Verarbeitung

Es werden alle Kenngrössen, die zum Zeitpunkt t ermittelt wurden, verwendet, weiters.

- $select_t$ die Nummer des gewählten Stapels
- $stackfunc_t$ die Operation, die auf den Stapel angewendet werden soll
- $source_t$ Angabe der Quelle, die den Wert für $datum_{t+1}$ liefert (ALU, $Immediate_t$, stackptr, stacklength + cachelength, buffer(source(1), source(0)))

Nur wenn die Verarbeitung keinen Verlängerungsschritt ausführt, wird *stackfunc,* wie in Tabelle 16 angeführt, ausgewertet.

2.2.3 Die Verarbeitung

Operation	***Aktion***
pushstack	der Wert, der durch *source* angegebenen Quelle wird $datum_{t+1}$
getput	wahlfreies Lesen eines Stapelelement, bzw. wahlfreies Überschreiben eines Stapelelements. Die Unterscheidung ermöglicht *source(2)*. Diese Operation erwartet die Adresse – relativ zum obersten Stapelelement – im obersten Stapelelement. Das zweite Stapelelement ersetzt den Wert des adressierten Elements beim Schreiben. Die Indizierung des Stapels beginnt beim ersten Element, das keinen Parameter für die Operation darstellt, dessen Index ist 0. Diese Operation erkennt selbstständig, ob die Quelle bzw. das Ziel im lokalen Speicher oder im Überlauf liegt. Es setzt entsprechend *indexr* bzw. *indexi* oder ein Adressregister *target* für einen Speicherzugriff. Ebenso werden die entsprechenden Steuersignale bereitgestellt
loadstore	analog zu *getput*, Quelle bzw. Ziel ist der Arbeitsspeicher
switchcore	wechseln zu einem anderen Kern
readpop	erzwungenes Lesen des neuesten Wertes vom Überlauf, die Adresse kommt vom Operationsschritt *Aktualisierung*
writepop	erzwungenes Schreiben von *last* in den Überlauf, die Adresse kommt ebenfalls vom Operationsschritt *Aktualisierung*

Tabelle 16: Alle Operationszyklen der Phase Verarbeitung

Die neuen Steuerwerte für den Speicherzugriff werden in *IOCode*, für den Stapelzugriff in *fetch* – wahlfreies Lesen des Stapels –, und *store* – für wahlfreies Schreiben –, abgelegt. Alle Operationen, die lesend auf den Stapel oder Arbeitsspeicher zugreifen – *stackfunc* hat den Wert *getput* oder *loadstore* –, benötigen wenigstens einen zusätzlichen Operationsschritt (stall2) zum Übernehmen des Wertes in den Stapel.

2.2.4 Weitere Aufgaben

Für bedingte Sprünge und die Rückkehr aus Unterprogrammen benötigt die *Program prefetch queue* den Wert von $top_t(0)$, sowie eine Gültigkeitsgarantie

$$TOSvalid_{t+1} \leftarrow source_t(3)$$

Der Befehlsdecoder muss informiert werden, ob er warten muss, weil ein Verlängerungsschritt ausgeführt wird, oder fortfahren kann

$$Ready_{t+1} \leftarrow stall1_t \text{ nor } stall2_t$$

Weiter sind an die Eingänge der ALU die gewünschten Datenquellen zu legen, die Information steht in $selectalu_t$, Tabelle 17.

selectalu$_t$	***Eingang SideA***	***Eingang SideB***
ImIm	Immediate$_t$	Immediate$_t$
FIm	top$_t$(0)	Immediate$_t$
ImF	Immediate$_t$	top$_t$(0)
SF	top$_t$(1)	top$_t$(0)

Tabelle 17: Die Operandenbelegungen der ALU-Eingänge

2.3 Der Befehlsdecoder

Der Decoder, Tabelle 18, übernimmt von *program prefetch queue* eine einzelne vollständige Instruktion – den Operationscode und einen eventuell angehängten unmittelbaren Wert – und übersetzt diese in einen fixen Satz von Steueranweisungen für ihre Verarbeitung. Dieser Satz und der (übernommene) unmittelbare Wert werden an die Stapelverwaltung weitergegeben. Der Decoder ist über *Ready* mit der Stapelverwaltung synchronisiert und liefert nur dann einen neuen Satz, wenn die Stapelverwaltung dazu bereit ist. Dieses Synchronisationssignal wird auch an die *program prefetch queue* weitergeleitet, weiters werden *TOSvalid$_t$* und *top$_t$(0)* - das oberste Stapelelement und sein Bestätigungssignal – unverändert durch geleitet.

2.3 Der Befehlsdecoder

Kennung		***Satz***						
Opcode	Mnemnonik	select	Pop	source	stackfunc	select-alu	Operand	afunc
0000000	NOP	Opcode(7)	0	0100	NopStack	ImIm	0	AddALUnC
0000001	!	Opcode(7)	2	0111	LoadStore	ImIm	0	AddALUnC
0000010	@	Opcode(7)	0	0011	LoadStore	ImIm	0	AddALUnC
0000011	C!	Opcode(7)	2	0101	LoadStore	ImIm	0	AddALUnC
0000100	C@	Opcode(7)	0	0001	LoadStore	ImIm	0	AddALUnC
0000101	H!	Opcode(7)	2	0110	LoadStore	ImIm	0	AddALUnC
0000110	H@	Opcode(7)	0	0010	LoadStore	ImIm	0	AddALUnC
0000111	DEPTH	Opcode(7)	0	0100	PushStack	ImIm	0	AddALUnC
0001000	DROP	Opcode(7)	1	0100	NopStack	ImIm	0	AddALUnC
0001001	2DROP	Opcode(7)	2	0100	NopStack	ImIm	0	AddALUnC

Kennung		***Satz***						
Opcode	Mnemnonik	select	Pop	source	stackfunc	select-alu	Operand	afunc
0001010	NIP	Opcode(7)	2	00 Opcode(7) 0	PushStack	ImIm	0	AddALUnC
0001011	PICK	Opcode(7)	0	0011	GetPut	ImIm	0	AddALUnC
0001100	PUT	Opcode(7)	2	0111	GetPut	ImIm	0	AddALUnC
0001101	VAL	Opcode(7)	0	0110	PushStack	ImIm	PPQ	AddALUnC
0001110	DUP	Opcode(7)		00 Opcode(7) 0	PushStack	ImIm	0	AddALUnC
0001111	OVER	Opcode(7)	0	00 Opcode(7) 1	PushStack	ImIm	0	AddALUnC
0010000	R@	Opcode(7)	0	00 not Opcode(7) 0	PushStack	ImIm	0	AddALUnC
0010001	R1@	Opcode(7)	0	00 not Opcode(7) 1	PushStack	ImIm	0	AddALUnC
0010010	SAVE	Opcode(7)	0	0100	SaveStack	ImIm	0	AddALUnC
0010011	HALT	Opcode(7)	0	0100	NopStack	ImIm	0	AddALUnC
0010100	SP!	Opcode(7)	0	0100	SetSPStack	ImIm	0	AddALUnC
0010101	SP@	Opcode(7)	0	0101	PushStack	ImIm	0	AddALUnC

Kennung		***Satz***						
0010110	+	Opcode(7)	2	0111	PushStack	SF	0	AddALU
0010111	-	Opcode(7)	2	0111	PushStack	SF	0	SubALU
0011000	1+	Opcode(7)	1	0111	PushStack	FIm	1	AddALUnC
0011001	1-	Opcode(7)	1	0111	PushStack	FIm	1	SubALUnC

Kennung		***Satz***						
Opcode	Mnemnonik	select	Pop	source	stackfunc	select-alu	Operand	afunc
0011010	2*	Opcode(7)	1	0111	PushStack	FIm	1	LshiftALU
0011011	2/	Opcode(7)	1	0111	PushStack	FIm	1	RshiftAALU
0011100	AND	Opcode(7)	2	0111	PushStack	SF	0	AndALU
0011101	CELL+	Opcode(7)	1	0111	PushStack	FIm	4	AddALUnC
0011110	HALF+	Opcode(7)	1	0111	PushStack	FIm	2	AddALUnC
0011111	INVERT	Opcode(7)	1	0111	PushStack	FIm	-1	XorALU
0100000	LSHIFT	Opcode(7)	2	0111	PushStack	SF	0	LshiftALU
0100001	NEGATE	Opcode(7)	1	0111	PushStack	ImF	0	SubALU
0100010	OR	Opcode(7)	2	0111	PushStack	SF	0	OrALU
0100011	RSHIFT	Opcode(7)	2	0111	PushStack	SF	0	RshiftALU
0100100	XOR	Opcode(7)	2	0111	PushStack	SF	0	XorALU
0100101	+B	Opcode(7)	2	0111	PushStack	SF	0	AddCALU
0100110	-B	Opcode(7)	2	0111	PushStack	SF	0	SubCALU
0100111	LSHIFTC	Opcode(7)	2	0111	PushStack	SF	0	LshiftCALU
0101000	RSHIFTC	Opcode(7)	2	0111	PushStack	SF	0	RshiftCALU
0101001	0<!	Opcode(7)	0	0111	PushStack	FIm	0	LtALU
0101010	0<	Opcode(7)	1	0111	PushStack	FIm	0	LtALU
0101011	0=!	Opcode(7)	0	0111	PushStack	FIm	0	EquALU
0101100	0=	Opcode(7)	1	0111	PushStack	FIm	0	EquALU
0101101	CMP	Opcode(7)	0	0111	PushStack	SF	0	SubALU

Kennung		***Satz***						
0101110	CONST-10	Opcode(7)	0	0100	PushStack	ImIm	0	AddALUnC

Kennung		***Satz***						
Opcode	Mnemnonik	select	Pop	source	stackfunc	select-alu	Operand	afunc
0101111	CONST-15	Opcode(7)	0	0100	PushStack	ImIm	0	AddALUnC
0110000	CONST-5	Opcode(7)	0	0100	PushStack	ImIm	0	AddALUnC
0110001	CONST-14	Opcode(7)	0	0100	PushStack	ImIm	0	AddALUnC
0110010	CONST-9	Opcode(7)	0	0100	PushStack	ImIm	0	AddALUnC
0110011	CONST-13	Opcode(7)	0	0100	PushStack	ImIm	0	AddALUnC
0110100	CONST-8	Opcode(7)	0	0100	PushStack	ImIm	0	AddALUnC
0110101	0<>!	Opcode(7)	0	0111	PushStack	FIm	0	nEquALU
0110110	0<>	Opcode(7)	2	0111	PushStack	SF	0	nEquALU
0110111	0>!	Opcode(7)	0	0111	PushStack	FIm	0	GtALU
0111000	0>	Opcode(7)	1	0111	PushStack	FIm	0	GtALU
0111001	CONST-12	Opcode(7)	0	0100	PushStack	ImIm	0	AddALUnC
0111010	CONST-7	Opcode(7)	0	0100	PushStack	ImIm	0	AddALUnC
0111011	CONST-11	Opcode(7)	0	0100	PushStack	ImIm	0	AddALUnC
0111100	CONST-6	Opcode(7)	0	0100	PushStack	ImIm	0	AddALUnC
0111101	EXIT	Not Opcode(7)	1	1000	NopStack	ImIm	0	AddALUnC
0111110	CALL	Not Opcode(7)	0	0110	PushStack	ImIm	PPQ	AddALUnC

2.3 Der Befehlsdecoder

Kennung		***Satz***						
0111111	TRAP	Not Opcode(7)	0	0110	PushStack	ImIm	PPQ	AddALUnC
1000000	BRANCH	Opcode(7)	0	0100	NopStack	ImIm	0	AddALUnC
1000001	0BRANCH!	Opcode(7)	0	1000	NopStack	ImIm	0	AddALUnC
1000010	0BRANCH	Opcode(7)	1	1000	NopStack	ImIm	0	AddALUnC

Kennung		***Satz***						
Opcode	Mnemnonik	select	Pop	source	stackfunc	select-alu	Operand	afunc
1000011	B@	Opcode(7)	0	0111	PushStack	ImIm	0	SubCALU
1000100	B!	Opcode(7)	1	0111	NopStack	ImF	0	SubALU
1000101	BREAK	Opcode(7)	0	0100	NopStack	ImIm	0	AddALUnC
1000110	VALH	Opcode(7)	0	0100	PushStack	ImIm	PPQ	AddALUnC
1000111	CONST15	Opcode(7)	0	0100	PushStack	ImIm	0	AddALUnC
1001000	CONST14	Opcode(7)	0	0100	PushStack	ImIm	0	AddALUnC
1001001	CONST13	Opcode(7)	0	0100	PushStack	ImIm	0	AddALUnC
1001010	CONST12	Opcode(7)	0	0100	PushStack	ImIm	0	AddALUnC
1001011	CONST11	Opcode(7)	0	0100	PushStack	ImIm	0	AddALUnC
1001100	CONST10	Opcode(7)	0	0100	PushStack	ImIm	0	AddALUnC
1001101	CONST9	Opcode(7)	0	0100	PushStack	ImIm	0	AddALUnC
1001110	CONST8	Opcode(7)	0	0100	PushStack	ImIm	0	AddALUnC
1001111	CONST7	Opcode(7)	0	0100	PushStack	ImIm	0	AddALUnC
1010000	CONST6	Opcode(7)	0	0100	PushStack	ImIm	0	AddALUnC
1010001	CONST5	Opcode(7)	0	0100	PushStack	ImIm	0	AddALUnC
1010010	CONST4	Opcode(7)	0	0100	PushStack	ImIm	0	AddALUn

Kennung		***Satz***						
								C
1010011	CONST3	Opcode(7)	0	0100	PushStack	ImIm	0	AddALUnC
1010100	CONST2	Opcode(7)	0	0100	PushStack	ImIm	0	AddALUnC
1010101	CONST1	Opcode(7)	0	0100	PushStack	ImIm	0	AddALUnC
1010110	CONST0	Opcode(7)	0	0100	PushStack	ImIm	0	AddALUnC
1010111	CONST-1	Opcode(7)	0	0100	PushStack	ImIm	0	AddALUnC
1011000	CONST-2	Opcode(7)	0	0100	PushStack	ImIm	0	AddALUnC

Kennung		***Satz***						
Opcode	Mnemnonik	select	Pop	source	stackfunc	select-alu	Operand	afunc
1011001	CONST-3	Opcode(7)	0	0100	PushStack	ImIm	0	AddALUnC
1011010	CONST-4	Opcode(7)	0	0100	PushStack	ImIm	0	AddALUnC
1011011	U0<	Opcode(7)	1	0111	PushStack	FIm	0	ULtALU
1011100	U0>	Opcode(7)	1	0111	PushStack	FIm	0	UGtALU
1011101	H*	Opcode(7)	2	0111	PushStack	SF	0	MultALU
1011110	SETPC	Opcode(7)	1	1000	NopStack	ImIm	0	AddALUnC
1011111	SWITCH	Opcode(7)	0	0100	SwitchCore	ImIm	PPQ	AddALUnC
1100000	GETCOREID	Opcode(7)	0	0110	PushStack	ImIm	PPQ	AddALUnC
sonstige		Opcode(7)	0	0100	NopStack	ImIm	0	AddALUnC

Tabelle 18: Alle Befehle mit den notwendigen Steuerinformationen

(PPQ ist der unmittelbare Operand der *program prefetch queue*)

97 Instruktionen sind bisher vergeben, 31 mögliche weitere noch undefiniert. Liegt eine undefinierte Instruktion vor, verhält sie sich wie ein NOP-Befehl, zusätzlich aktiviert sie das Steuersignal *illegal*, das als Unterbrechungssignal verwendet werden kann. Das achte Bit des Operationscode gibt an, ob – mit 0 – der für den Befehl definierte Stapel verwendet werden soll, oder – mit 1 – der alternative Stapel, dieser kann vom Programmierer durch das Präfix „A:“, das dem Befehl voran zustellen ist, erzwungen werden.

Die Bedeutung von *source* – Tabelle 19, Angaben zur Datenquelle – ist abhängig von *stackfunc*. *source(3)* ist aber immer eine Anforderung des obersten Stapelelements, die *program prefetch queue* benötigt dieses bei bedingten Sprüngen, Wechsel des Kerns und *EXIT*, alle anderen Befehle nicht.

stackfunc	*source(0)*	*source(1)*	*source(2)*	*Kommentar*
PushStack	1	1	1	ALU
PushStack	0	1	1	Unmittelbarer Operand
PushStack	1	0	1	stackptr
PushStack	0	0	1	Stacklength + cachelength
PushStack	Element	Stapel	0	Ein Element aus den Stapelpuffer
GetPut	--	--	0	Wahlfreies Stapelelement lesen
GetPut	--	--	1	Wahlfreies Stapelelement schreiben
LoadStore	--	--	0	Wahlfreie Speicherzelle lesen
LoadStore	--	--	1	Wahlfreie Speicherzelle schreiben

Tabelle 19: Die Bedeutungen der Steuerinformation source für Verarbeitungszyklen

Die Vergabe der Codes gehorcht keinerlei System, sondern erfolgte willkürlich.

2.4 Die program prefetch queue

Die Schnittstelle:

```
entity ProgramCounter is
  generic (constant TableBitWidth : integer := 4);
  port (nReset: in std_ulogic;                                -- system reset
        Clock: in std_ulogic;                                 -- system clock
        abortload: out std_ulogic;                            -- abort current
                                                                 fetch cycle
        ReadBlocked: in std_ulogic;                           -- fetch cycle
                                                                 delayed
        Opcode: out std_ulogic_vector(7 downto 0);            -- coded operation
        DataIn: in DataVec;                                   -- incoming data
        Address: out std_ulogic_vector(RAMrange'high - 1 downto 0);-- memory
                                                                      address
        Immediate: out DataVec;                               -- immediate operand
        Reads: out std_ulogic;                                -- memory operation
        Done: in std_ulogic;                                  -- core ready
```

```
        TOS: in DataVec;                                    -- return address
        TOSValid: in std_ulogic;                            -- return address
                                                               available
        Token: out std_ulogic;                              -- service routine
                                                               running
        Quit: out std_ulogic;                               -- vector processed
        Vector: in std_ulogic_vector(TableBitWidth - 1 downto 0);
                                                            -- interrupt vector
        VectorValid: in std_ulogic);                        -- interrupt vector
                                                               present
end ProgramCounter;
```

Diese Einheit liefert den nächsten Befehl für den Befehlsdecoder. Da die Operationscodes zwar eine fixe Länge von einem Byte haben, aber einigen ein Operand mit einer Länge von zwei bzw. vier Byte angehängt ist, können zwei Speicherzugriffe nötig sein um den Befehl vollständig zu lesen. Das bedeutet einen zusätzlichen Verlängerungsschritt, der vermieden werden kann - mit einer prefetch queue. Nicht nur das Wort, das den Operationscode des nächsten Befehls enthält, wird gelesen, sondern zumindest auch das unmittelbar darauf folgende Wort. Diese vorab gelesenen Worte bilden die prefetch queue, aus der der nächste Befehl entnommen wird. Die Zahl der Lesezugriffe wird mit dieser Methode zwar etwas größer als bei einer ungepufferten Methode, da z.B. im Fall einer Sprunganweisung der Pufferinhalt ungültig wird, daher zu viel gelesen wurde, andererseits benötigen lange Befehle keinen zusätzlichen Verlängerungsschritt mehr. Ein Puffer der Länge zwei bringt im Fall zwei aufeinander folgender langer Befehle noch keine Verbesserung, er muss also mindestens die Länge drei haben. Eine weitere Vergrößerung bringt keine Vorteile, sondern erhöht bloß die Zahl der Vorgriffe auf den Speicher. Um eine optimale Wirkung zu erzielen, wird vorausgesetzt, dass die Befehle lückenlos aufeinander folgen, also nicht auf Wortgrenzen ausgerichtet sind.

Die Einheit verfügt über mehrere Schnittstellen, unter anderem eine zum Speicher.

- DataIn — enthält ein gelesenes Wort (32 Bit)
- abortload — die zuletzt gestartete Leseanforderung wird sofort abgebrochen
- Reads — die Leseanforderung ist aktiv, wenn der Puffer nicht voll ist
- ReadBlocked — ist aktiv, wenn trotz Leseanforderung noch kein Wort gelesen werden konnte
- Address — die Adresse des zu lesenden Wortes ist ein sich selbständig, nach jeder Leseoperation, erhöhender Zeiger, der, sollte der Puffer leer sein, auf den Wert des Befehlszähler zu setzen ist.

Unterbrechungen können über folgende Schnittstelle angefordert werden.

- Vector — die Adresse eines Wortes, in welchem die Adresse der Unterbrechungsroutine gespeichert ist
- VectorValid — signalisiert einen gültigen Wert des Vektors
- Token — signalisiert dem Controller, dass die Serviceroutine gestartet ist

- Quit signalisiert dem Controller die Annahme des Vektors, sie kann VectorValid zurücksetzen

Bleibt noch die Schnittstelle zu Befehlsdecoder und Stapelverwaltung.

- Opcode ein 8-Bit-Befehlscode
- Immediate der 32-Bit Operand des Befehls
- Done der Decoder ist bereit zur Übernahme
- TOS das oberste Element des aktuellen Stapels
- TOSValid signalisiert einen gültigen Wert von TOS

Der prinzipielle Ablauf ist zuerst den Puffer zu aktualisieren, dann den nächsten Befehl extrahieren, den Befehlszähler erhöhen, gegebenenfalls ein weiteres Wort für den Puffer anfordern, und zuletzt den Operationscode und seinen Operanden an den Decoder weiterzugeben. Sollte ein Verlängerungsschritt notwendig werden – weil der Puffer leer ist oder auf den Stapel gewartet wird - wird der Befehl *NOP* an den Decoder gesendet, damit die folgenden Phasen ungehindert weiterarbeiten können.

Einige Befehle, Tabelle 20, werden zumindest teilweise in dieser Einheit verarbeitet bzw. nehmen gezielt Einfluss auf die *prefetch queue* bzw. den Befehlszähler, was ein Löschen des Puffers bewirkt.

Befehl	***Aktion***

EXIT	while(!TOSValid);
	PC = TOS;
	empty(Puffer);

Offset *CALL*	Immediate = PC; PC += Offset; empty(Puffer);
addr *TRAP*	Immediate = PC; PC = *addr; empty(Puffer);
vektor *Vektorvalid*	Immediate = PC; PC = *vektor; empty(Puffer);
offset *BRANCH*	PC += Offset; empty(Puffer);
offset *0BRANCH!*	while(!TOSValid); if (TOS == 0) { PC += Offset; empty(Puffer); }

2.4 Die program prefetch queue

Befehl	***Aktion***
offset *0BRANCH*	while(!TOSValid); if (TOS == 0) { PC += Offset; empty(Puffer); }
HALT	while(!Vektorvalid); PC++;

Tabelle 20: Einfluss nehmende Befehle

Tabelle 21 enthält Befehle, die mehr als fünf Operationsschritte brauchen.

Befehl	***stall***	***Kommentar***
EXIT	3	Die prefetch queue muss 3 Takte warten, bis die Rücksprungadresse in TOS erscheint
Offset *CALL*	1	Die prefetch queue benötigt einen zusätzlichen Takt um „PC + Offset“ zu berechnen
addr *TRAP*	1	Die *prefetch queue* benötigt einen zusätzlichen Takt um den Vektor in *addr* zu lesen
offset *BRANCH*	1	Die prefetch queue benötigt einen zusätzlichen Takt um „PC + Offset“ zu berechnen
offset *0BRANCH!*	3	Die prefetch queue muss 3 Takte warten, bis in TOS die Testgröße erscheint
offset *0BRANCH*	3	Die *prefetch queue* muss 3 Takte warten, bis in TOS die Testgröße erscheint
PICK	2	Aktualisierung und Verarbeitung benötigen einen zusätzlichen Takt für die Adressierung des Stapels und Übernahme des Datums
PUT	1	Die Aktualisierung benötigt einen zusätzlichen Takt
[C | H]@	>= 2	Die Aktualisierung benötigt einen zusätzlichen Takt und die Verarbeitung mindestens einen weiteren zur Übernahme des Datums
[C | H]!	>= 1	Die Aktualisierung benötigt einen zusätzlichen Takt, die Verarbeitung eventuell weitere

Befehl	***stall***	***Kommentar***
SAVE	*cachelength*	Die Auslagerung in den Speicher benötigt zusätzliche Takte
SP!	>= 1	Die Einlagerung aus dem Speicher braucht eventuell zusätzliche Takte um den Stapelpuffer zu füllen

Tabelle 21: Befehle, die stalls erzwingen

3 Der Prozessor

Quelle	***Kommentar***
mycpu.vhd	Der Prozessor
theCore.vhd	Der Prozessorkern
UART.vhd	Der UART
FiFo.vhd	Der FiFo des UART
counter.vhd	Zähler und Zeitgeber
IntVectors.vhd	Der Interrupt Controller
ROMcode.vhd	Das ROM
vhdl_syn_bl4.vhd	Das RAM-Interface
global.vhd	Globale Deklarationen

Tabelle 22: Die Quelldateien des Prozessors

Nachfolgend werden die integrierten Einheiten vorgestellt.

3.1 UART

Die Schnittstelle:

```
entity UART is
      port (nReset: in std_ulogic;                    -- reset
            Clk: in std_ulogic;                       -- system clock
            RxD: in std_ulogic;                       -- UART input
            TxD: out std_ulogic;                      -- UART output
```

```
            RTS: out std_ulogic;                          -- request to send
            CToS: in std_ulogic;                          -- clear to send
            ReadyR: out std_ulogic;                       -- receiver ready
            ReadyT: out std_ulogic;                       -- transmiter ready
            Address: in std_ulogic_vector(2 downto 0);       -- port address
            DataIn: in std_ulogic_vector(23 downto 0);       -- parallel input
            DataOut: out std_ulogic_vector(23 downto 0);     -- parallel output
            Writes: in std_ulogic;                        -- write to UART
            Reads: in std_ulogic);                        -- read from UART
end UART;
```

Der UART enthält einen Empfangs- und einen Sendeteil, die beide Teile arbeiten von einander unabhängig. Dem Empfänger ist ein Filter vorgelagert, der ein gesendetes Bit einwandfrei erkennt und den Empfangstakt mit dem Empfangssignal synchronisiert. Dem Sender ist ein Taktgenerator vorgelagert, dessen Takt die Zeitdauer eines gesendeten Bit vorgibt. Beide Einheiten verfügen über je eine parallele Schnittstelle, über die zu sendende bzw. empfangene Daten zur Weiterverarbeitung transportiert werden können.

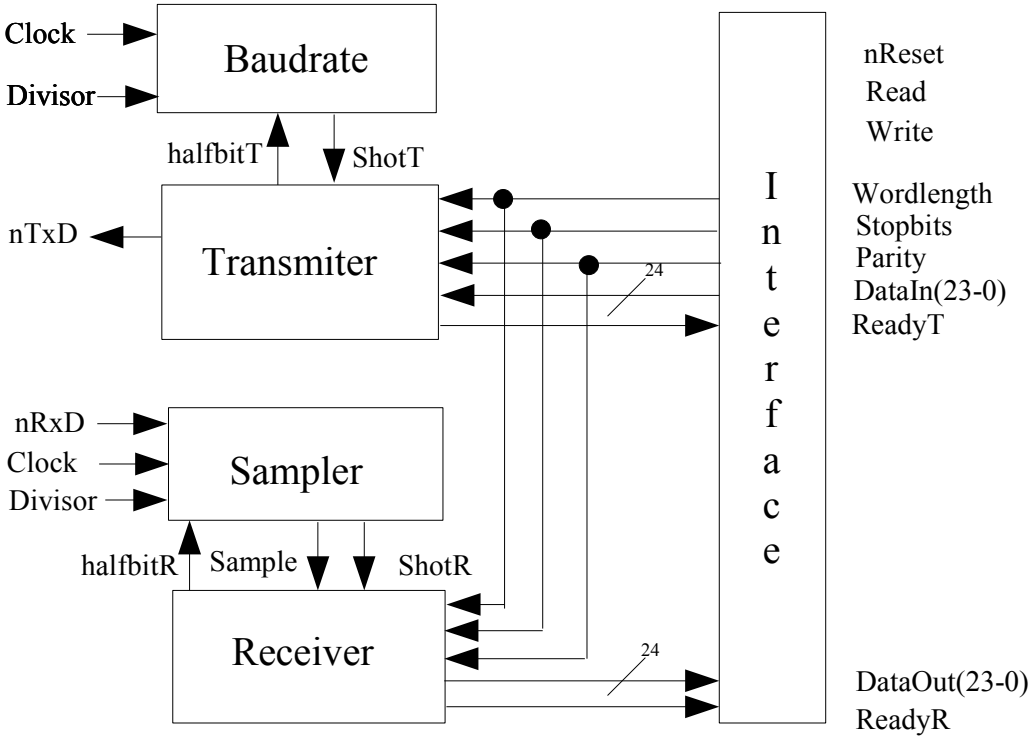


Abbildung 8: Blockschaltbild des UART

3.1.1 Das Interface

Das 9-Bit-Datum auf *DataIn* wird durch Aktivieren von *Writes* vom Transmiter übernommen. Das neunte Bit gibt an, ob ein Maskierungszeichen voraus gesendet werden soll, da die restlichen Bit ein Steuerzeichen repräsentieren. Sobald der Transmiter wieder bereit ist zu senden, aktiviert er *ReadyT*. Dieses Signal kann als Unterbrechungsanforderung verwendet werden. Es wird durch *Writes* automatisch deaktiviert.

Über *DataOut* kann das empfangene Datum übernommen werden, das 7 oder 8 Bit lang ist. Das neunte Bit muss rückgesetzt sein, andernfalls zeigt es einen fehlerhaften Empfang des Datums an. Ein Lesen von *DataOut* wird durch Aktivieren des Signal *Reads* eingeleitet, welches automatisch das Signal *ReadyR* rücksetzt, welches seinerseits ein vollständig empfangenes Wort im Receiver mitteilt. *ReadyR* kann als Unterbrechungsanforderung verwendet werden.

Die ab jetzt aufgezählten Signale, Tabelle 23, betreffen sowohl den Empfangs- als auch den Sendeteil.

Signal	***Bedeutung***
Wordlength	Die Länge eines zu sendenden bzw. empfangenen Datums. 1 8 Bit Datum 0 7 Bit Datum
Parity	Gibt an, ob ein Prüfbit nötig ist und seine Parität 00 kein Paritätsbit verwenden 10 Paritätsbit auf gerade Parität ergänzen 01 Paritätsbit auf ungerade Parität ergänzen 11 ohne Bedeutung
Stoppbits	Gibt an, wie viele Stoppbits ein Zeichen beenden 00 2 Stoppbits 01 1 Stoppbits 10 1,5 Stoppbits 11 ohne Bedeutung
Divisor	Eine frei wählbare 24 Bit Konstante, mit der der Systemtakt zu teilen ist, um die doppelte Baudrate zu erzeugen.
Clock	Der Systemtakt
nReset	Das allgemeine Rücksetzsignal, für alle Einheiten gültig

Tabelle 23: Die Parameter der Schnittstelle

3.1.2 Der Sendeteil

3.1.2.1 Der Baudratengenerator

Er ist nicht mehr als ein programmierbarer Teiler, der synchron zum Transmiter läuft. Solange der Transmiter auf einen Schreibbefehl (Signal *Writes*) wartet, ist er inaktiv. Der Generator wird gestartet, sobald das Datum zum Senden bereit ist, und läuft dann synchron, bis das Datum vollständig übertragen ist. Anschließend ist er wieder inaktiv.

Es ist möglich, im laufenden Betrieb die Zeitdauer für ein Bit zu halbieren. Dazu dient das Signal *halfbitT*, welches vom Transmiter bereitgestellt wird. Diese Möglichkeit ist notwendig um 1,5 Stoppbit senden zu können. Jedes mal wenn der Arbeitszähler neu zu laden ist, wird dieses Signal geprüft, und der Zähler entsprechend initialisiert, entweder mit dem zweifachen Wert von *Divisor* für ein ganzes Bit, oder mit *Divisor* für ein halbes Bit.

3.1.2.2 Der Transmiter

Er wird durch einen endlichen Automaten realisiert. Im inaktiven Initialzustand *Idle* wird geprüft, ob der Empfänger die Sendung von *XON* bzw. *XOFF* verlangt, und sendet diese gegebenenfalls. Sonst wird auf ein Datum gewartet. Liegt eines an, wird geprüft, ob ein Fluchtzeichen verlangt wird, und dieses gegebenenfalls gesendet, unmittelbar darauf das Datum.

Der Sendevorgang wird eingeleitet, in dem der 11-Bit breite Sendepuffer mit dem Datum, Start-, Parität- und erstem Stoppbit initialisiert wird, in den Zustand *Sending* gewechselt wird. Dieser Zustand wird erst verlassen, wenn der Sendepuffer leer ist, also nur mehr Nullen enthält. Folgezustand ist *SendingDone*, wenn kein weiteres Stoppbit nötig ist, andernfalls ist *Stopping* zwischengeschaltet, in dem ein weiteres halbes oder ganzes Stoppbit gesendet wird. In *SendingDone* wird lediglich das Ende der Übertragung abgewartet und in den inaktiven Initialzustand *Idle* gewechselt.

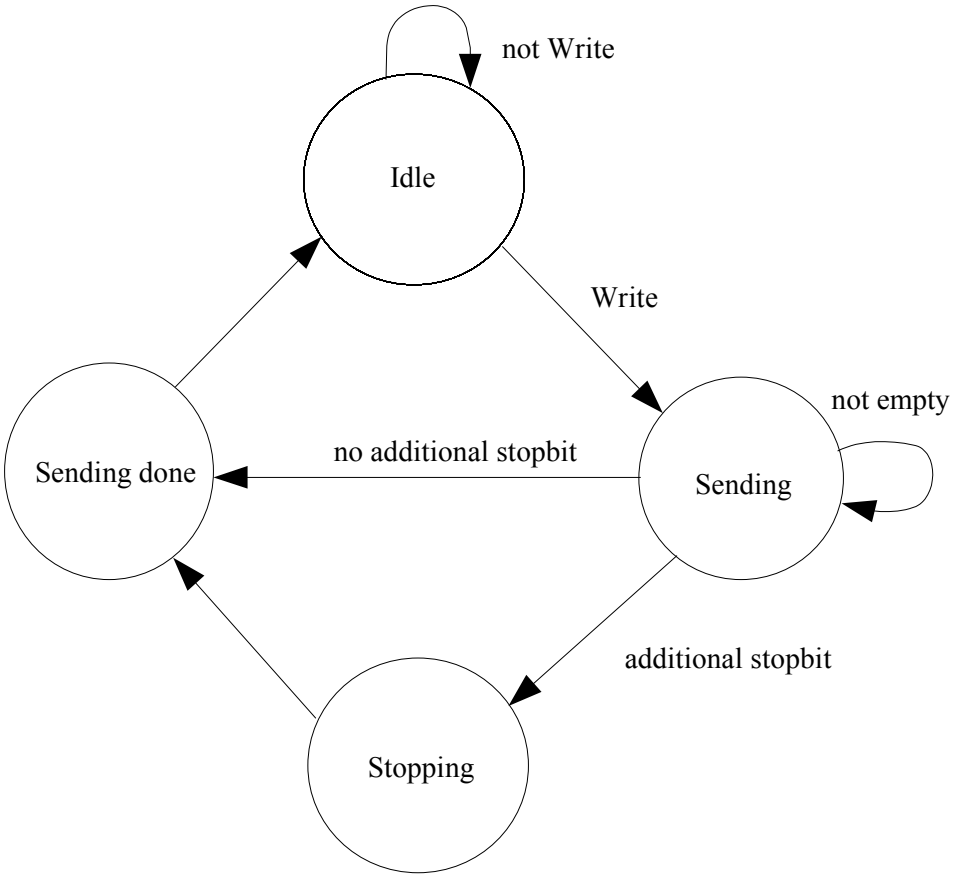


Abbildung 9: Automat des Transmiters

3.1.3 Der Empfangsteil

3.1.3.1 Der Filter

Der Filter startet die Erkennung und Takterzeugung, sobald am Empfängereingang eine logische 0 erscheint. Diese muss zumindest für die halbe Zeitdauer eines Taktes anliegen, um ein Startbit eindeutig zu erkennen. Ist dies nicht der Fall, wartet der Filter weiter auf ein Startbit, bei erkanntem Startbit erzeugt er einen stabilen Takt für den Receiver. Die Filterung des Eingangssignals wird mit Beginn eines jeden Taktes erneut gestartet. Wird während dieser Zeitdauer wenigstens für den halben Zeitraum eine logische 0 detektiert, wird eine 0, sonst eine 1, an den Receiver weitergeleitet. Die kontinuierliche Takterzeugung wird beendet, sobald der Receiver wieder im Zustand *Idle* ist.

3.1.3.2 Der Receiver

Im inaktiven Startzustand *Idle* prüft er, ob der Empfangspuffer frei ist oder nicht und erzwingt gegebenenfalls die Sendung von *XOFF* bzw. *XON*. Ansonsten wartet der

3.1.3 Der Empfangsteil

Automat auf die einwandfreie Erkennung des Startbits durch das Filter, und geht erst im Zustand *ReceiveData* auf Empfang des Datums. Dort wird geprüft, ob alle Datenbit empfangen wurden und entschieden, ob ein zusätzliches Paritätsbit oder ein oder zwei Stoppbits nachfolgen. Im Zustand *ReceiveParity* wird der korrekte Empfang durch eine logische 0 im Bit 8 des Empfangspuffers gekennzeichnet, im Fehlerfall wird es auf 1 gesetzt. Dann wird geprüft, ob ein oder zwei Stoppbits ausständig sind. Im Zustand *ReceiveStopbit* wird der Empfang des ersten Stoppbit abgewartet. Im Zustand *ReceiveStopbits* wird der Empfang des letzten Stoppbit abgewartet, der vollständige Empfang des Datums wird durch *ReadyR* signalisiert, und in den inaktiven Zustand *Idle* gewechselt.

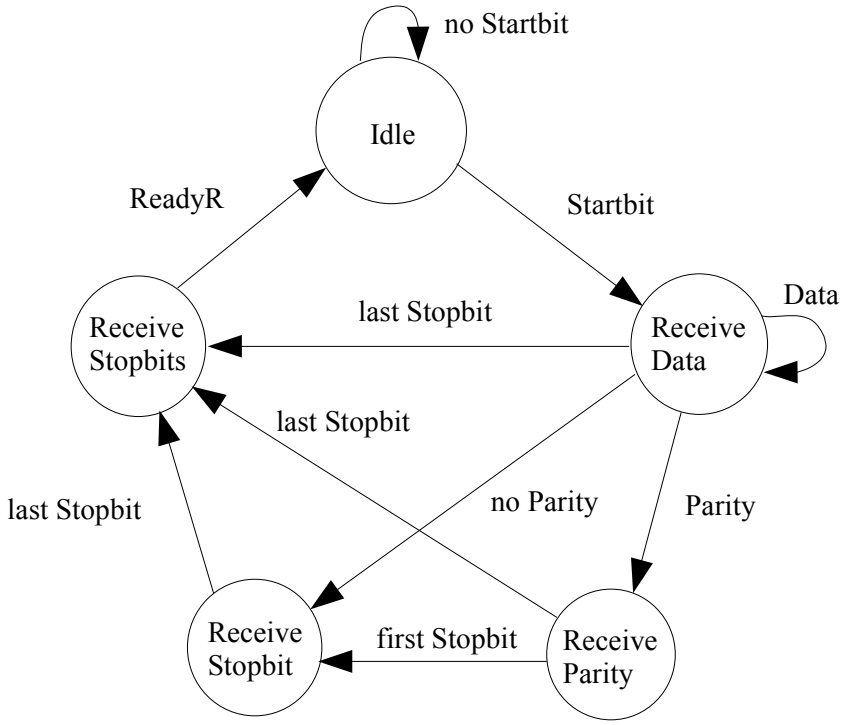


Abbildung 10: Automat des Receivers

Diese Einheit belegt folgende Adressen, angeführt in Tabelle 24.

Adresse	***Kommentar***
ffffffe0H	Datenport
ffffffe4H	Definitionsport
ffffffe8H	Vorteiler des Baudratengenerators, 24 Bit
fffffecH	Statusport
fffffff0H	Das Fluchtzeichen escape
fffffff4H	Das Zeichen XON
fffffff8H	Das Zeichen XOFF

Tabelle 24: Die Ports des UART

Tabelle 25 definiert die Belegung des Datenport.

Bit	***Bedeutung***
7-0	Das Datum
8	Ist es gesetzt, erzwingt es die Sendung eines Fluchtzeichen, bei einem Schreibzugriff Ist es gesetzt, zeigt es einen fehlerhaften Empfang an, bei einem Lesezugriff

Tabelle 25: Bedeutung der Bits des Schreib- bzw. Leseports

Tabelle 26 definiert die Belegung von Definition- bzw. Statusport.

3.1.3 Der Empfangsteil

Bit	***Werte***
1 – 0	Parität: 0 none 1 odd 2 even
3 – 2	Stoppbit: 1 ein Stoppbit 0 zwei Stoppbit 2 1.5 Stoppbit
4	Wortlänge: 0 7 Bit 1 8 Bit

Tabelle 26: Bedeutung der Bits des Definition- bzw. Statusports

Tabelle 27 definiert die Belegung der Ports Fluchtzeichen, XON, XOFF

Bit	***Bedeutung***
7-0	Bitmuster des Zeichens
8	Ist es gesetzt, sind die vorangestellten Bit ohne Bedeutung

Tabelle 27: Bedeutung der Bits des Ports der Fluchtzeichen

Ohne FiFo kam es gelegentlich zu Datenverlusten. Um diese zu vermeiden, wurden Sender und Empfänger mit je einem FiFo versehen.

3.1.4 Der FiFo

Die Schnittstelle:

```
entity FiFo is
  generic(constant WordLength: integer := 9;          -- size of word
          constant ldFiFoSize: integer := 3);         -- default FiFo size 8
  port (nReset: in std_ulogic;                        -- reset
        Clock: in std_ulogic;                         -- system clock
```

```
        put: in std_ulogic;                              -- write into FiFo
        get: in std_ulogic;                              -- read from FiFo
        DataIn: in std_ulogic_vector(WordLength - 1 downto 0);-- incoming data
        DataOut: out std_ulogic_vector(WordLength - 1 downto 0);-- outgoing data
        notFull: out std_ulogic;                         -- FiFo not full
        AlmostFull: out std_ulogic;                      -- only last half free
        Empty: out std_ulogic);                          -- FiFo empty
end FiFo;
```

Als Kenngrössen werden die Adressen des ersten Eintrags *first*, des unmittelbar auf den Letzten folgenden *last* und die aktuelle Zahl der freien Einträge *free*, verwendet. In einem einzigen Prozess werden die Eingangssignale ausgewertet und die Kenngrössen aktualisiert, sowie die Ausgangsdaten generiert.

3.2 Zähler und Zeitgeber

Die Schnittstelle:

```
entity counter is
  port (nReset: in std_ulogic;                           -- system reset
        Clock: in std_ulogic;                            -- system clock
        Address: in std_ulogic_vector(1 downto 0);       -- address bus
        source: in std_ulogic_vector(0 to 3);            -- counter input
        DataIn: in std_ulogic_vector(26 downto 0);       -- incomming data
        DataOut: out std_ulogic_vector(26 downto 0); -- outgoing data
        Writes: in std_ulogic;                  -- write operation (active high)
        Tint: out std_ulogic_vector(3 downto 0));    -- interrupt request
end counter;
```

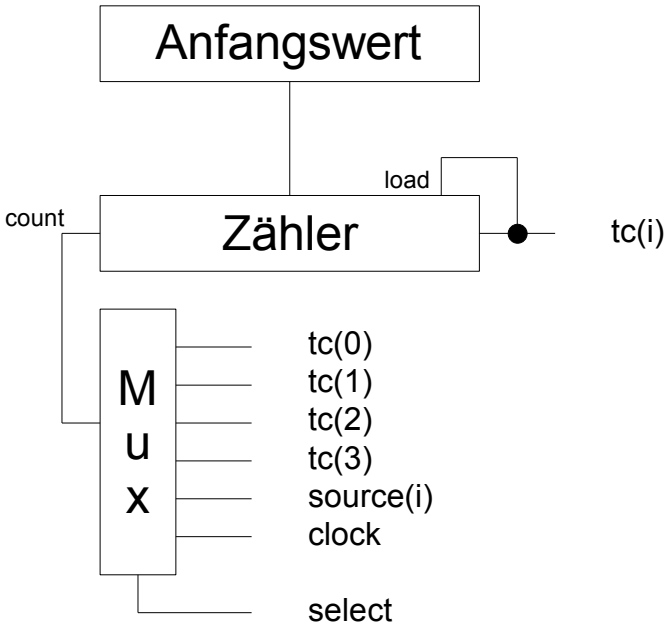


Abbildung 11: Ein einzelner Abwärtszähler

Im Bild ist der prinzipielle Aufbau eines der vier implementierten 24-Bit-Zähler dargestellt. Es handelt sich um einen Abwärtszähler, der sich automatisch neu mit dem Anfangswert initialisiert, sobald er auf Null herunter gezählt hat. Das dabei erzeugt Signal *tc,* ein Impuls von einem Systemtakt Dauer, wird auch als Unterbrechungssignal verwendet, weiter kann es Zählereignis eines anderen Zählers sein. Alternative Zählereignisse sind das externe Eingangssignal *source*, sowie der Systemtakt *clock*.

Die Zähler arbeiten parallel. Die Zählereignisse können mit maximal dem halben Systemtakt eintreffen, da der Eingang *count* nicht auf den Pegel, sondern auf den Wechsel des Pegels von „nicht aktiv“ auf „aktiv“ reagiert. Ist das Zählereignis der System-takt *clock*, geht es doppelt so schnell, da kein Wechsel ausgewertet werden muss.

Die Zähler werden durch Schreiben einer Definition auf ihre Basisadressen (ffffffc0H, ffffffc4H, ffffffc8H oder ffffffccH) definiert, der Zähler übernimmt die neue Definition, Tabelle 28, sofort, unterbricht den laufenden Zähler aber nicht. Erst wenn 0 erreicht ist, wird der neue Anfangswert in den Zähler geladen.

Bit	***Datum***
2 - 0	Select, dient zur Selektion des Zählereignis
26 - 3	Der Anfangswert

Tabelle 28: Belegung des Definitionsport

Die Selektion des Zählereignisses ist in Tabelle 29 definiert.

Select	***Eingang***	***Programmaufruf zur Erzeugung von Select***
000	Tc(0), Zähler 0 ist der Vorteiler	0 TRIGGER-PRESCALER
001	Tc(1), Zähler 1 ist der Vorteiler	1 TRIGGER-PRESCALER
010	Tc(2), Zähler 2 ist der Vorteiler	2 TRIGGER-PRESCALER
011	Tc(3), Zähler 3 ist der Vorteiler	3 TRIGGER-PRESCALER
100	Der externe Eingang *source*	TRIGGER-INPUT
101	Der Systemtakt *clock*	TRIGGER-SYSCLOCK
110	Der Systemtakt *clock*	TRIGGER-SYSCLOCK
111	Der Zähler ist gesperrt	LOCK-COUNTER

Tabelle 29: Die Bedeutung des Parameters Select

Beim Auslesen der Basisadresse erscheinen die Daten in folgendem Format, Tabelle 30.

Bit	***Datum***
2 - 0	Select
26 - 3	Der aktuelle Zählerstand

Tabelle 30: Belegung des Leseport

3.3 Der Interrupt Controller

Die Schnittstelle:

```
entity IntVectors is
  generic (constant TableBitWidth : integer := 4);
  port (nReset: in std_ulogic;                          -- system reset
        Clock: in std_ulogic;                           -- system clock
        Quit: in std_ulogic;                            -- interrupt quitted
        Token: in std_ulogic;                           -- update pending
        IntSignal: in std_ulogic_vector(2 ** TableBitWidth - 1 downto 0);
                                                        -- input
        Writes: in std_ulogic;                          -- write port
        Reads: in std_ulogic;                           -- read port
        Address: in std_ulogic_vector(1 downto 0);      -- port
        DataIn: in std_ulogic_vector(2 ** TableBitWidth - 1 downto 0);
                                                        -- incomming data
        DataOut: out std_ulogic_vector(2 ** TableBitWidth - 1 downto 0);
                                                        -- outgoing data
        Vector: out std_ulogic_vector(TableBitWidth - 1 downto 0);
                                                        -- vector number
        VectorValid: out std_ulogic);                   -- vector valid
end IntVectors;
```

Ein Prozessor sollte auch unvorhergesehenen externen Ereignissen Rechnung tragen können. Er benötigt dazu eine Einheit die derartige Ereignisse erkennt, bei mehreren gleichzeitigen Ereignissen das mit der höchsten Dringlichkeit auswählt, und abschließend dem Prozessorkern mitteilt, wie er mit diesem Ereignis zu verfahren hat. Derartige Ereignisse nennt man Unterbrechungen (Interrupts), und die verarbeitende Einheit *Interrupt Controller*.

Es gibt etliche Arten wie die Einheit auf eine Unterbrechung reagieren kann, die Flexibelste ist in Form eines Unterbrechungsvektors. Dabei sendet der Controller eine Adresse, unter der die Adresse einer Unterbrechungsroutine gespeichert ist, an den Prozessorkern, dieser unterbricht die laufende Verarbeitung und führt stattdessen die Unterbrechungsroutine aus, anschließend setzt er das unterbrochene Programm fort.

Bei dieser Implementierung wird die Nummer (0 – 15) des unterbrechenden Signals an die *program prefetch queue* gesendet, die diese mit 4 multipliziert und das Produkt als Adresse des Vektors interpretiert. Für Vektoren sind daher die ersten 64 Byte des Adressraums zu reservieren, in denen die Adressen der Unterbrechungsroutinen zu speichern sind. Sie können daher beliebig geändert werden, womit ein Höchstmaß an Flexibilität garantiert ist.

Die Synchronisation mit dem Kern erfolgt durch das Aufforderungssignal *VektorValid* – ein gültiger Vektor steht bereit –, das mit *Quit* quittiert wird, sobald der Vektor akzeptiert wird. Erst das Signal *Token* – die Serviceroutine wird vom Kern ausgeführt –, erlaubt dem Controller einen neuen Vektor zu erzeugen.

Diese Einheit besitzt 16 asynchrone, Impuls- oder Pegel gesteuerte, Eingänge, die bei jedem Takt gelesen werden – daher eine Impulsdauer von mindestens einem Takt. Der

Eingang 0 hat die höchste Priorität, der Eingang 15 die Niedrigste, höherwertige Unterbrechungen können Niederwertigere unterbrechen, umgekehrt nicht. Jeder Eingang ist maskierbar, kann also gezielt von einer Verarbeitung ausgeschlossen werden. Die Eingänge werden nicht direkt ausgewertet, sondern akkumuliert, der Akkumulator wird aber in jedem Takt ausgewertet, das heißt es wird die Unterbrechung mit der höchsten Priorität ermittelt und, wenn erlaubt, ihr Vektor generiert. Der Akku garantiert, dass nicht-priorisierte Unterbrechungen erhalten bleiben. Unterbrechungsroutinen müssen ihr akkumuliertes Signal, durch Lesen bzw. Schreiben auf eine reservierte Adresse, gezielt zurücksetzen. Vorzugsweise erfolgt das unmittelbar vor dem Befehl EXIT, damit ist garantiert, dass keine niederwertigen Unterbrechungen die Unterbrechungsroutine unterbrechen können.

Weiter existiert die Möglichkeit den Controller ein- bzw. auszuschalten, was für den Programmierer von Interesse sein kann, auch dafür stellt der Assembler Makros, Tabelle 31, bereit.

Makro	***Eingehend***	***Ausgehend***	***Adresse***	***Kommentar***
EI	0|1		ffffffd4H	mit Parameter 1 wird der Controller mit 4 Takten Verzögerung eingeschaltet, mit 0 sofort ausgeschaltet. Das Makro liefert keinen Wert zurück
DI		0|1	ffffffd4H	der Controller wird bedingungslos ausgeschaltet, gibt aber seinen bisherigen Zustand zurück
QI			ffffffd8H	Quittiert die Unterbrechung – setzt das akkumulierte Signal zurück - und ermöglicht damit eine neuerliche Auswertung seines Eingangs
PI		pending	ffffffd8H	Gibt einen 16-Bit-Vektor zurück, bei dem jedes gesetzte Bit angibt, ob sein korrespondierender Eingang eine Unterbrechung anfordert
SETIMASK	Maske		ffffffd0H	Der korrespondierende Eingang jedes gesetzten Bit wird gesperrt.
GETIMASK		Maske	ffffffd0H	Die Maske lesen

Tabelle 31: Makros zur Einbindung des Controllers in FORTH

3.4 Das ROM

Die Schnittstelle:

3.4 Das ROM

```
entity ROMcode is
  port (Clock: in std_ulogic;                          -- system clock
        Address: in std_ulogic_vector(ROMrange);       -- address bus
        Data: out DataVec);                            -- outgoing data
end ROMcode;
```

Die Kapazität ist durch *ROMrange*, Datei *global.vhd*, definiert. Zur Zeit besitzt eine Kapazität von 4096 Worten zu 32 Bit. Seine Größe kann in 2048 Byte Schritten erweitert werden, wenn in Datei *global.vhd*, *ROMrange'high* auf neu gesetzt wird und I*ndexBitWidth* gegebenenfalls entsprechend vermindert wird. Der lokale Speicher der Stapel hat dann eine kleinere Kapazität als 1024 Worte, also 2 Stapel mit je der halben Kapazität. Im ROM ist das BIOS gespeichert. Seine Startadresse liegt unmittelbar nach dem RAM. Die Größe des RAM ist in *RAMrange* (Datei *global.vhd*) definierten. Das BIOS selbst wird im Abschnitt *Software* im ersten Kapitel erklärt.

3.5 Die CPU

Die Schnittstelle:

```
entity CPU is
  generic (constant TableBitWidth : integer := 4;     -- ld(number of interrupts)
           constant IndexBitWidth: integer := IndexBitWidth);
                                          -- ld(size of dedicated stack memory)
  port (nReset: in std_ulogic;                        -- system reset
        SysClock: in std_ulogic;                      -- system clock
        RxD2: in std_ulogic;                          -- UART serial input
        TxD2: out std_ulogic;                         -- UART serial output
        RxD1: in std_ulogic;                          -- UART serial input
        TxD1: out std_ulogic;                         -- UART serial output
        source: in std_ulogic_vector(0 to 3);         -- CTC input signals
        SD_A: out std_logic_vector(12 downto 0);      -- memory address
        SD_BA: out std_logic_vector(1 downto 0);      -- bank address
        SD_DQ: inout std_logic_vector(15 downto 0);   -- databus
        SD_RAS: out std_logic;                        -- RAS of DDR2 memory
        SD_CAS: out std_logic;                        -- CAS of DDR2 memory
        SD_CK_N: out std_logic;
        SD_CK_P: out std_logic;
        SD_CKE: out std_logic;
        SD_ODT: out std_logic;
        SD_CS: out std_logic;
        SD_WE: out std_logic;                         -- WE of DDR2 memory
        SD_LDM: out std_logic;
        SD_LOOP_IN: in std_logic;                     -- SDRAM calibration loop
        SD_LOOP_OUT: out std_logic;
        SD_UDM: out std_logic;
        SD_LDQS_N: inout std_logic;
        SD_LDQS_P: inout std_logic;
        SD_UDQS_N: inout std_logic;
        SD_UDQS_P: inout std_logic;
        Interrupt: in std_ulogic_vector(0 to 7));  -- external interrupt sources
end CPU;
```

Die CPU implementiert den Busmanager. Hier erfolgt die Adressdekodierung der integrierten Einheiten gemäß Tabelle 32, die Unterbrechungssignale werden angeschlossen, die Ports werden ausschließlich der Stapelverwaltung zugänglich gemacht, die *program prefetch queue* kann nur auf ROM und RAM zugreifen. Stapelverwaltung und *program prefetch queue* können gleichzeitig auf ROM und RAM zugreifen, das heißt die eine Einheit auf das RAM, die Andere auf das ROM. Beide auf RAM bzw. ROM ist nicht möglich, in einem solchen Fall hat die Stapelverwaltung Vorrang.

Adressbereich	***Einheit***
0 – 3ffffffH	RAM
4000000H - 4003fffH	ROM
fffffc0H - fffffcfH	Zähler und Zeitgeber
fffffd0H - fffffdfH	Interrupt Controller
fffffe0H - ffffffbH	UART
ffffffcH - ffffffH	Software Unterbrechung (Schreibzugriff)
ffffffcH - ffffffH	Systemtakt in Hz (Lesezugriff)

Tabelle 32: Adressaufteilung des Prozessors

3.5.1 Der RAM-Controller

```
entity vhdl_syn_bl4  is
  port (
    cntrl0_DDR2_DQ: inout  std_logic_vector(15 downto 0);
    cntrl0_DDR2_A: out  std_logic_vector(12 downto 0);
    cntrl0_DDR2_BA: out  std_logic_vector(1 downto 0);
    cntrl0_DDR2_CK: out std_logic;
    cntrl0_DDR2_CK_N: out std_logic;
    cntrl0_DDR2_CKE: out std_logic;
    cntrl0_DDR2_CS_N: out std_logic;
    cntrl0_DDR2_RAS_N: out std_logic;
    cntrl0_DDR2_CAS_N: out std_logic;
    cntrl0_DDR2_WE_N: out std_logic;
    cntrl0_DDR2_ODT: out std_logic;
    cntrl0_DDR2_DM: out  std_logic_vector(1 downto 0);
    cntrl0_rst_dqs_div_in: in std_logic;
    cntrl0_rst_dqs_div_out: out std_logic;
    SYS_CLK: in std_logic;
    reset_in_n: in std_logic;
    cntrl0_burst_done: in std_logic;
    cntrl0_init_val: out std_logic;
```

```
    cntrl0_ar_done: out std_logic;
    cntrl0_user_data_valid: out std_logic;
    cntrl0_auto_ref_req: out std_logic;
    cntrl0_user_cmd_ack: out std_logic;
    cntrl0_user_command_register: in  std_logic_vector(3 downto 0);
    cntrl0_clk_tb: out std_logic;
    cntrl0_clk90_tb: out std_logic;
    cntrl0_sys_rst_tb: out std_logic;
    cntrl0_sys_rst90_tb: out std_logic;
    cntrl0_sys_rst180_tb: out std_logic;
    cntrl0_user_output_data: out  std_logic_vector(31 downto 0);
    cntrl0_user_input_data: in  std_logic_vector(31 downto 0);
    cntrl0_user_data_mask: in  std_logic_vector(3 downto 0);
    cntrl0_user_input_address: in  std_logic_vector(25 downto 0);
    cntrl0_DDR2_DQS: inout  std_logic_vector(1 downto 0);
    cntrl0_DDR2_DQS_N: inout  std_logic_vector(1 downto 0)
    );
end vhdl_syn_bl4;
```

Der Controller für DDR2-RAM, organisiert als 16x32MB Matrix, wurde vom Xilinx Spartan3A-Starter-Kit [16] übernommen und adaptiert. Die Daten können nur im Burst-mode (Länge vier 16-Bit Worte) gelesen bzw. geschrieben werden. Die Adaptierung unterstützt nur einzelne 16- oder 8-Bit Zugriffe, aber keine direkten 32-Bit Zugriffe. Grund ist ein fehlerhaftes Lesen bzw. Schreiben – hohes und niedriges Wort werden gelegentlich vertauscht, abhängig von der Adresslage -, deren Ursache ich nicht lokalisieren konnte. Die vorgenommenen Adaptierungen sind im Anhang, Kapitel 2 beschrieben.

3.5.2 Die Statemachine des RAM

Der Controller wird von der Statemachine kontrolliert und in den folgenden Diagrammen dargestellt.

3.5.2 Die Statemachine des RAM

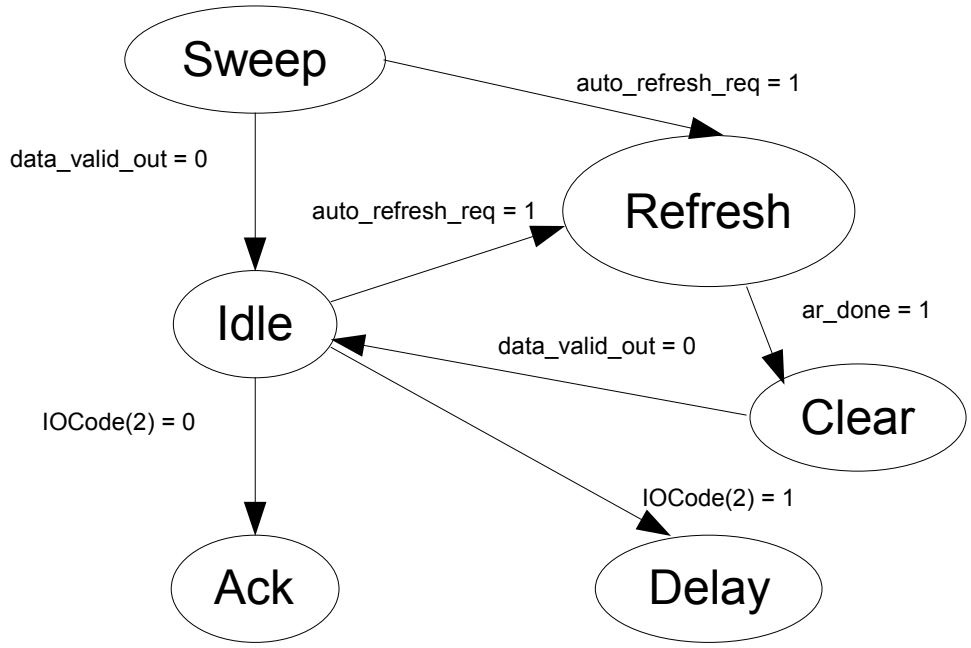


Abbildung 12: Startknoten Idle und Refresh-zyklus

Vom Zustand *Idle* geht es mit *IOCode(2) = 0* in den Zustand *Ack*, Lesen des RAM, mit *IOCode(2) = 1* in den Zustand *Delay*, Schreiben ins RAM, ist *auto_refresh_req* aktiv in den Zustand *Refresh*. *Refresh* hat Vorrang gegenüber IOCode(2). *IOCode(2)* wird nur ausgewertet, wenn *IOCode(1) /= 0* und *IOCode(0) /= 0* sind. Zustand *Sweep* schließt Lese- und Schreibzugriffe ab. Erst wenn *data_valid_out = 0* ist, kann ein neuer Speicherzugriff eingeleitet werden, wird vorher *auto_refresh_req* aktiv, geht es unmittelbar in den Zustand *Refresh*. Refresh wird erst verlassen, wenn der Controller sein Ende durch *ar_done = 1* signalisiert. Im Zustand *Clear* wird auf die Bereitschaft des Controllers gewartet und mit *data_valid_out = 0* nach *Idle* gewechselt.

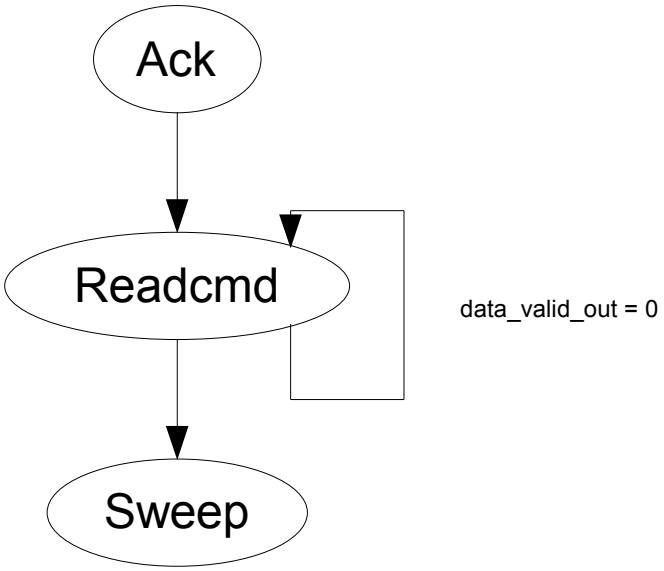


Abbildung 13: Der Lesezyklus

Vom Zustand *Ack* geht es unmittelbar in den Zustand *Readcmd*, der erst verlassen wird, wenn *data_valid_out = 1* ist, erst dann werden die Daten von *user_output_data* übernommen und das Signal *stall = 0* gesetzt. Dies zeigt der Stapelverwaltung bzw. der program prefetch queue an, dass die Daten im nächsten Takt übernommen werden können. Das Ende des Zugriffs wird in *Sweep* abgewartet.

3.5.2 Die Statemachine des RAM

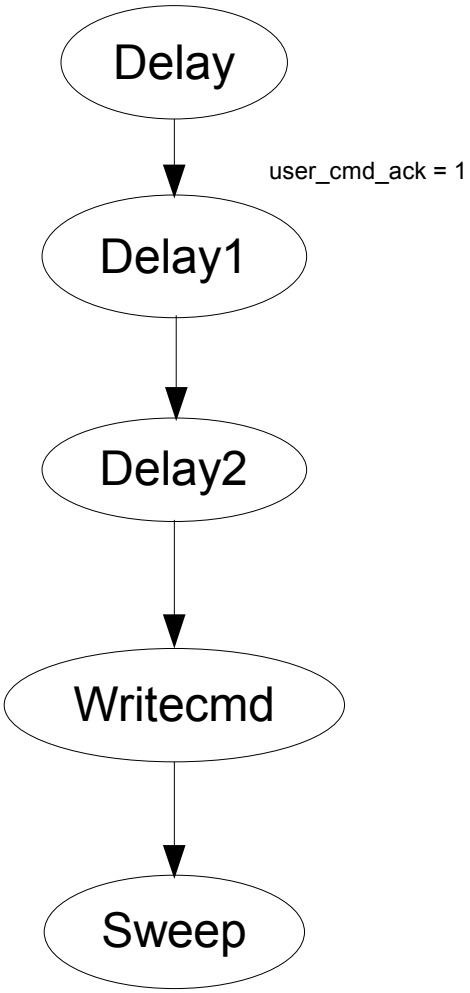


Abbildung 14: Der Schreibzyklus

Sobald *user_cmd_ack = 1*, der Zyklus also eingeleitet wurde, geht es in den Zustand *Delay1*, und darauf unmittelbar in den Zustand *Delay2*, wo *user_data_mask* mit lauter Einsen gefüllt wird, um ein Schreiben der letzten 2 Worte des Burstzugriffs zu verhindern. Im Zustand *Writecmd* wird das Quittierungssignal *stall = 0* gesetzt und damit der Stapelverwaltung das Ende der Operation signalisiert. In Sweep wird das Ende des Zugriffs abgewartet.

Die Statemachine wurde vereinfacht, nur für 16- bzw. 8-Bit Zugriffe wiedergegeben. Implementiert wurde auch der 32-Bit Zugriff, der durch zwei aufeinander folgende 16-Bit Zugriffe realisiert wurde. Für Speicheroperationen ergeben sich zusätzliche Operationsschritte, wie in Tabelle 33 festgehalten.

Einheit	*32- bit schreiben*	*8 bzw. 16-bit schreiben*	*32-bit lesen*	*8 bzw. 16-bit lesen*
RAM	14	7	14	7
ROM	-	-	1	1
Integrierte Einheiten	0	0	0	0

Tabelle 33: Die Zahl der zusätzlichen Takte für Zugriffe auf RAM bzw. ROM

4 Test und Verifikation

Quelle	*Kommentare*
Countertest.vhd	Testbench für den Zählerbaustein
Inttest.vhd	Testbench für den Interrupt Controller
Programcountertest.vhd	Testbench für die program prefetch queue
Uarttest.vhd	Testbench für den UART
cpu1sim.vhd	Testbench für den Prozessor (Verifikation)
Cpu1.vhd	Der Prozessor mit SRAM-Interface
TheCore1.vhd	Der Prozessorkern
SRAM.vhd	Ein statisches RAM
UART1.vhd	Ein zweiter UART
cpu2.vhd	Testbench für den Prozessor (Postsimulation)

Tabelle 34: Die Quelldateien für die Verikation und Test

Zur Verifikation bzw. Test wurden oben angeführte Programme erstellt, die sich in einem eigenen ISE-Projekt (FORTHS) befinden. Diese Version der CPU verwendet ein SRAM, welches sich einfacher simulieren lässt als ein DDR2-RAM.

4.1 Test einzelner Bausteine

4.1.1 Der Zählerbaustein

Der Zähler 0 wird mit 1 initialisiert und als Timer – Zeitgeber ist der Systemtakt –

betrieben. Sobald das zugeordnete Unterbrechungssignal aktiv wird, wird der Testlauf beendet. Das Signal muss genau einen Takt anliegen.

4.1.2 Der Interrupt Controller

An den Unterbrechungseingängen wird zuerst Leitung 4 aktiv, nach 200 ns Leitung 5. Auf *Vector* sollte 4 ausgeben werden mit aktivem *VectorValid*. Nach weiteren 200 ns werden beide Leitungen inaktiv und *Quit* sowie *Token* quittieren die Unterbrechung 4. Nun sollte *Vectorvalid* inaktiv sein. Nach weiteren 100 ns werden *Quit* und *Token* inaktiv und Leitung 3 aktiv. Auf dem *Vector* sollte 3 ausgegeben werden und *VectorValid* aktiv sein.

4.1.3 Die Program Prefetch Queue

Überprüft wird das Verhalten bei CALL, TRAP und EXIT. Die korrekte Funktion kann an den Signalverläufen von *Address*, *Opcode* und *Immediate* im WAVE-Fenster von ModelSim abgelesen werden. Das ausgeführte Programm ist in *myromtable* der Testbench zu finden.

4.1.4 Der UART

Sender und Empfänger sind kurzgeschlossen. Überprüft wird, ob einzelne Worte (lauter 0, lauter 1, 1 0 alternierend) einschließlich Parity korrekt empfangen werden. Bei fehlerhaftem Empfang wird eine Meldung ausgegeben.

4.1.5 Die Stapelverwaltung

Für die Stapelverwaltung wurde ebenfalls eine Testbench verwendet, die aber nicht mehr vorhanden ist. Sie war aber ebenso einfach gebaut, wie die noch vorhandenen Benches. Die korrekte Funktion konnte mit den Testbenches jedenfalls nachgewiesen werden.

4.2 Test des Prozessors

4.2.1 Behavioral Simulation des Codes

In *cpu1sim.vhd* wird ein Assemblerprogramm, welches in der Variable *object* angegeben ist, an den Server (Prozessor und BIOS) gesendet, dort gebunden und exekutiert. Die gesamte Kommunikation zwischen Server und Simulator wird als Dump auf der Console des Simulators angezeigt. Zweck dieser Testbench ist es die Verifikation des Prozessors, des BIOS und des Assemblers zu ermöglichen. Einige Dateien mussten dafür modifiziert werden.

- cpu1.vhd Enthält ein SRAM-Interface und einen zusätzlichen Ausgang Mnemnonic zur Anzeige des aktuellen Befehls im Befehlsdecoder.
- theCore1.vhd Stellt zusätzlich Mnemnonic bereit.
- SRAM.vhd Enthält ein 32x64k statischen RAM. Dumps sind möglich in dem man in die höchste Adresse des RAM die Startadresse des Dump schreibt.

- UART1.vhd Ein zusätzlicher UART zur Kommunikation mit dem Server.

Die UART arbeiten mit 4,5 Mbaud. In bios.fs ist dazu das Makro SIMULATION auf -1 zu setzen. Mit dieser Testbench wurde das BIOS erfolgreich verifiziert. Fehler im Prozessor traten nicht auf, wären aber erkannt worden – er hätte falsche Ergebnisse geliefert, was aufgefallen wäre. Getestet wurde mit folgenden Assemblerprogrammen.

- countt.fs Überprüfung und Test des Zählerbausteins
- juchu.txt Test der Consolenausgabe
- memorytest.fs Test des dynamischen Speichers
- mirror.fs Spiegeln der Bits des Wortes f0f0f0f0
- shift.fs Test der Schiebeoperationen
- test.fs Test der Operation /MOD
- testf.fs Test der Mathematik-Bibliothek

4.2.2 Post Map Simulation

cpu2.vhd funktioniert analog cpu1sim.vhd, es dient ausschließlich der Simulation auf RTL-Ebene.

- mycpu.vhd Enthält ein SRAM-Interface und den Prozessor.
- SRAM.vhd Enthält ein 32x64k statischen RAM. Dumps sind möglich in dem man in die höchste Adresse des RAM die Startadresse des Dump schreibt.
- UART1.vhd Ein zusätzlicher UART zur Kommunikation mit dem Server.

Dieser Test war eine Formsache. Alle Signalverläufe waren glatt, ohne Peaks und Spikes, eine Überarbeitung der Quellen daher unnötig.

4.2.3 Post Place and Route Simulation

Diese wurde ebenfalls mit *cpu2.vhd* durchgeführt und war der abschließende Test des Prozessors, bevor er auf dem Entwicklungsboard getestet wurde. Ein Simulationslauf ist sehr zeitaufwendig und dauerte mit Programm *juchu.obj* etwa 4 Tage, verlief aber sofort erfolgreich.

Nachfolgend sind noch vier Prints von Speicherzugriffen auf das SRAM gelistet. Im Ersten wird -2 als 16-Bit-Wort geschrieben, im Nächsten ausgelesen. Im Dritten wird -2 als 8-Bit-Wort geschrieben, und im Letzten ausgelesen.

4.2.3 Post Place and Route Simulation

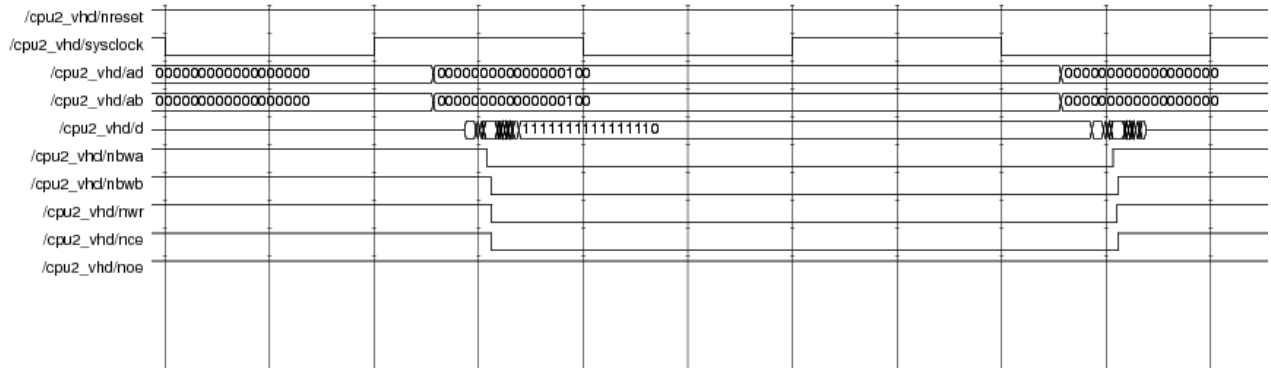


Abbildung 15: 16-Bit Schreibzugriff auf das SRAM

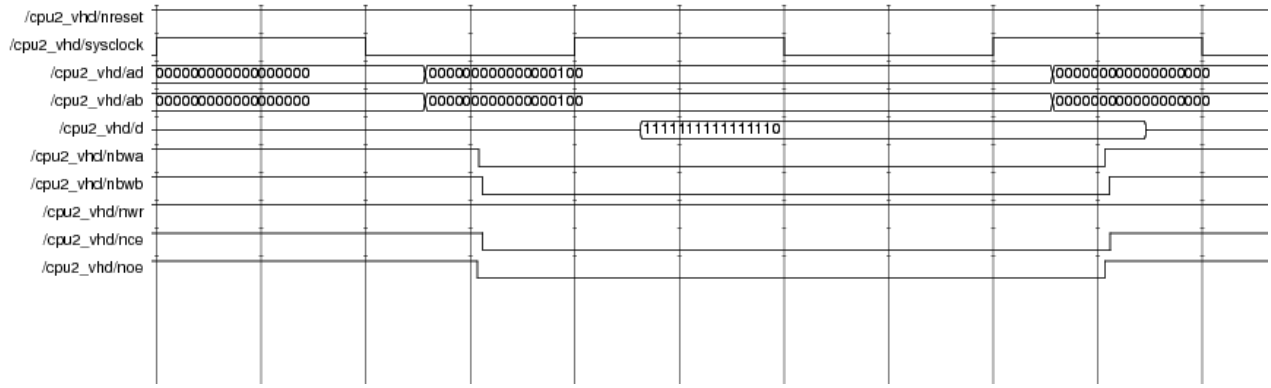


Abbildung 16: 16-Bit Lesezugriff auf das SRAM

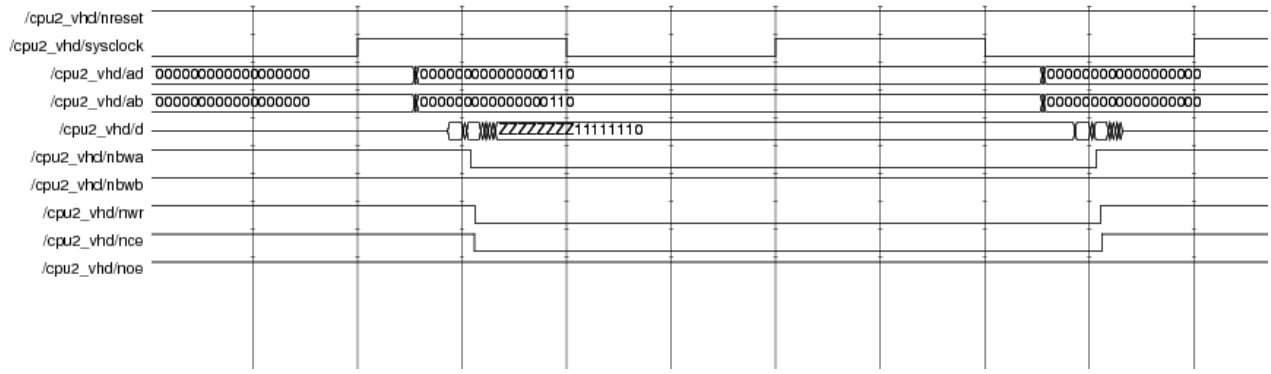


Abbildung 17: 8-Bit Schreibzugriff auf das SRAM

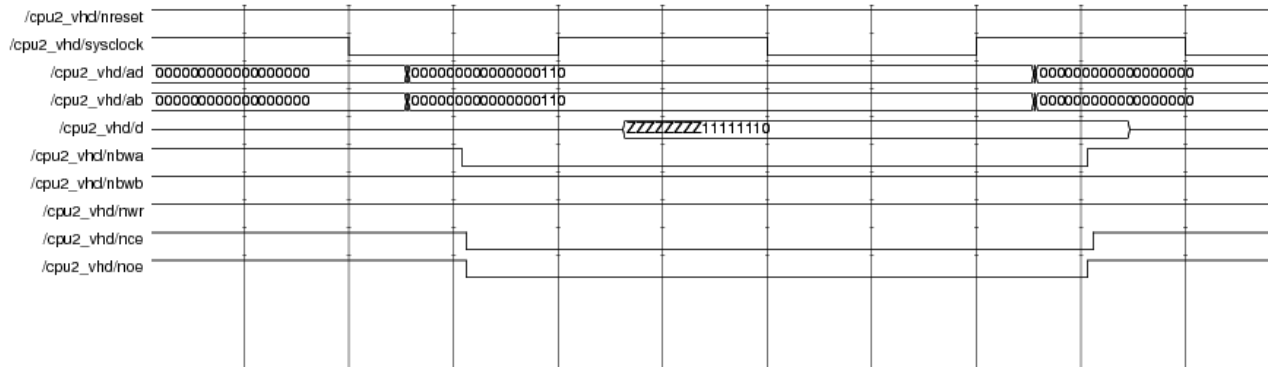


Abbildung 18: 8-Bit Lesezugriff auf das SRAM

Software

Nachfolgend werden Grammatiken in EBNF verwendet – die Notation wird als bekannt vorausgesetzt –, sowie für deren Nonterminalsymbole *regex*-Ausdrücke – nachzulesen in *JavaDoc* von *Sun,* Klasse *Pattern* [20].

1 Das BIOS

Es umfasst mehrere Quelldateien, Tabelle 35.

Quelle	***Kommentar***
bios.fs	Initialisiert und startet den Server
dictionary.fs	Schnittstelle zum FORTH-Dictionary
memorymanagement.fs	Die dynamische Speicherverwaltung
SIO.fs	Serviceroutinen der seriellen Schnittstelle
counter.fs	Unterstützung für Zähler und Zeitgeber
interruptservice.fs	Unterstützung für die Unterbrechungsverwaltung
consoleIn.fs	Die Standardeingabe
consoleOut.fs	Die Standardausgabe
fileIO.fs	Die Dateiverwaltung
java.txt	Unterstützung für Java
math.fs	Ganzzahlige Arithmetik und Logik
fmath.fs	Gleitkommazahlen
formatnumber.fs	Formatierung ganzzahliger Werte
linker.fs	Laden, binden und exekutieren von Anwendungen

Tabelle 35: Die Quelldateien des BIOS

Der Arbeitsspeicher wird, wie in Tabelle 36 festgelegt, aufgeteilt.

Beginn	***Ende***	***Kommentar***
0000000	000003F	Tabelle der Unterbrechungsroutinen
0000040	000083F	Schreib- bzw. Lesepuffer der Console
0000840	000093F	PAD, ein vordefinierter Scratchbuffer
0000940	0000A3F	16 Bufferhandle
0000A40	2FFFFFF	Heap
2FFFFFF	0000A40	Dictionary
3000000	3FFFFFF	Anwendungen
3FFF000		Stapel der Rücksprungadressen
4000000		Datenstapel

Tabelle 36: Die Aufteilung des RAM

Nach einem Reset wird das Dictionary aufgebaut, die Stapel und ihr Überlaufbereich gesetzt, die Tabelle der Unterbrechungsroutinen, gemäß Tabelle 37, initialisiert, und der Client über die Bereitschaft des Servers – Dienst 252 – informiert.

Unterbrechung	***Routine***
15	SERIALOUT – der Sender des UART
14	SERIALIN – der Empfänger des UART
13	TIMER3 – fort zählen der Systemzeit
0	ILLEGAL – undefinierter Befehlscode
1 - 12	maskiert

Tabelle 37: Verwendete Unterbrechungen

Anschließend prüft der Server immerfort, ob ein Dienst angefordert wird und führt ihn gegebenenfalls aus. Die implementierten Dienste sind in Tabelle 38 angeführt.

Code	***Dienst***
255	Software Reset – bewirkt einen Neustart des Servers
254	Laden, binden und exekutieren einer Anwendung
253	Das Dictionary komplett auslesen und an den Client senden
252	Die Länge des freien Empfangspuffers der Standardeingabe an den Client senden
251	Sofortiger Abbruch einer Anwendung
250	Die serielle Schnittstelle neu definieren
249	Die Größe von Heap und dynamischem Speicher abfragen
248	Den dynamischen Speicher anlegen und initialisieren
247	Den dynamischen Speicher freigegeben
246	Den dynamischen Speicher neu initialisieren
245	Ein Modul entfernen
244	Spätes Binden der zuletzt geladenen Module
243	Laden, binden und exekutieren mehrerer Module einer Anwendung
242-241	undefiniert

Tabelle 38: Die Dienste des BIOS

Neu definierte Dienste erfordern eine Modifikation von *bios.fs* und der Serviceroutine *SERIALIN* in *SIO.fs*.

Folgende Standardvariable [2] werden vom BIOS definiert: PAD, HLD, TIB, #TIB, >IN, SOURCE-ID, SPAN, BASE, PRECISION, SYSCLOCK, MILLISECONDS. Ihre Bedeutung ist, mit Ausnahme von SYSCLOCK, der Systemtakt in Hz, und MILLISECONDS, die Zeit, die seit dem letzten Hardwarereset verstrichen ist, im Standard [2] nachzulesen und wird hier nicht erläutert. Die Standardfunktionen sind nachfolgend aufgeführt und ebenfalls nicht näher erläutert, mit Ausnahme der Erweiterungen.

1.1 Das Protokoll

Verwendet wird der 8-Bit-ASCII Code. Die Zeichenbereiche 0 – 27 und 241 – 255 sind reservierte Steuerzeichen, die, sollen sie als Daten verwendet werden, mit einem voran-gestellten Zeichen 27 (escape) maskiert werden müssen.

Der Aufbau eines Satzes ist in Tabelle 39 festgelegt.

Byte	***Bedeutung***
0	Ein Handle, Wertebereich 0 – 22 0 ... Standardein- bzw. ausgabe 1 – 14 ... Dateihandle 15 ... für das Protokoll reserviert 16 .. für „freier Empfangspuffer“ der Standardeingabe 17 – 22 reserviert 23 ... reserviert für Xon 24 ... reserviert für Xoff
1	Spezifiziert die Aktion
Ab 2	Daten
zuletzt	Satzende: 26 für korrekte Daten, 25 für fehlerhafte Daten

Tabelle 39: Der Aufbau eines von den Diensten verwendeten Frame

Die Dienste arbeiten nach folgenden Protokollen der Tabelle 40. Sie ist von links nach rechts zu lesen, wobei jede neue Zeile einen Fortschritt in der Abarbeitung bedeutet. Ein Dienst kann nur vom Client durch Senden der Nummer des Dienstes eingeleitet werden, der Server antwortet mit dem Ergebnis bzw. fordert den Client zum Senden von Daten auf, usw.

1.1 Das Protokoll

Client	***Server***
255	16 freeinputbufferlength 26
254 bzw. 243 15 deck 26 -- senden des Deck	15 READ 240 -1 26 -- Leseanforderung 15 MODULPROCESSED 4-byte-errorcount (26 | 25) -- Ende des (erfolgreichen = 26) Binden mitteilen
253	15 LISTDICT dictionary 26 -- Dictionary senden
252	16 freeinputbufferlength 26
251	

Client	***Server***
250 15 7-Byte 26	15 READBUFFERED 240 7 26 -- Leseanforderung der Parameter
249	15 MEMORYUNUSED freeheapsize dynamicmemorysize freeprogramstorage 26
248 15 4-Byte 26	15 READBUFFERED 240 4 26 -- Anforderung der Größe
247	--
246	--
245 15 Modulname 26	15 READBUFFERED 240 -1 26 -- Leseanforderung des Namen
244 15 Liste 26	15 READBUFFERED 240 -1 26 -- Anforderung der Liste der nachträglich zu bindenden Referenzen

Tabelle 40: Protokolle der Dienste

1.2 Dateioperationen

Das Dateisystem [2, Kapitel 11.6.1, Seite 66 - 70] erlaubt dem Server Daten vom Client zu lesen bzw. auf ihm zu schreiben. Insgesamt können gleichzeitig 16 Dateien verwendet werden, die über Handle verwaltet werden. Reserviert sind Handle 0, die Standardeingabe bzw. -ausgabe, und Handle 15 für das Protokoll der Kommunikation zwischen Server und Client, die Handle 1 bis 14 sind frei verfügbar, werden aber vom Server beim Öffnen bzw. Anlegen einer Datei automatisch vergeben, und kann vom Anwender nicht beeinflusst werden. Ein Handle ist durch ein einzelnes Byte repräsentiert, definiert in Tabelle 41.

Bit	***Bedeutung***
7 - 4	Datei- bzw. Pufferhandle
3	Directory flag
2	Datei beschreibbar
1	Datei lesbar
0	Binäre Daten

Tabelle 41: Bedeutung der Bits eines Filehandles

Die Attribute eines Handle können mit DIR, BIN, W/O, RIO, R/W definiert werden, Handle werden von OPEN-FILE, CREATE-FILE vergeben und mit CLOSE freigegeben.

Ein Pufferhandle, definiert in Tabelle 42, ist ein Quadtupel von Worten mit folgender Bedeutung.

Wort	***Bedeutung***
0	Adresse des Ringpuffers
1	Länge des Ringpuffers
2	Startindex des logischen Puffers
3	Aktuelle Länge des logischen Puffers

Tabelle 42: Der Aufbau eines Pufferhandle

Die restlichen in *fileIO.fs* spezifizierten Befehle sind: DELETE-FILE, FILE-POSITION, FILE-SIZE, INCLUDE-FILE, INCLUDED, READ-FILE, READ-LINE, REPOSITION-FILE, RESIZE-FILE, SET-FILEDATE, WRITE-FILE, WRITE-LINE, FILE-STATUS, GET-FILEDATE, FLUSH-FILE, RENAME-FILE, *PAGE*, *AT-XY*. Zusätzlich definierte Operationen sind in Tabelle 43 angeführt.

Befehl	***Bedeutung***
ABSOLUTE-FILE	buffer bufferlength name namelength – buffer bufferlength success(=0) Den absoluten Dateinamen im Dateisystem des Client ermitteln. Retourniert wird eine Zeichenkette
ENTRIES-FILE	buffer bufferlength name namelength – buffer bufferlength success(=0) Die Einträge eines Verzeichnisses auslesen. Retourniert wird ein Vektor von Zeichenketten
CHANGECHARSET	flag -- Setzt den Zeichensatz der Ausgabekonsole am Client mit wahr auf ASCII, sonst auf Unicode. Startvorgabe ist ASCII.

Tabelle 43: Zusätzliche Operationen für Dateien

Grundsätzlich ist jedem Befehl, der die Hilfe des Clients benötigt, eine Code zugeordnet, der den Dienst des Client spezifiziert. Über Handle 15 werden Sätze folgender Struktur weitergegeben.

Action [handle] { parameter } 26

Action ← OPEN | CREATEF | READ | LINE | *LISTDICT* | *ATXY* |
GETFILEPOS | SETFILEPOS | GETFILESIZE | SETFILESIZE |
INCLUDED | *CLEAR* | STATUS | RENAME | DELETE | CLOSE |
ABSOLUTE | RELATIVE | SETDATE | GETDATE | READBUFFERED |
ASCII | ERRORREC

Die Parameter sind die, die der Befehl erwartet, und werden mit Code 26 abgeschlossen. Anschließend wartet der Server auf eine Antwort des Client. Der Client antwortet nach Ausführung der remote procedure über Handle 15. Ausnahme sind die *kursiv* geschriebenen Befehle, die keine Quittierung benötigen.

{ result } (26 | 25)

1.3 Das Dictionary

Ein einzelner Eintrag wird in FORTH Token [2, Kapitel 3.3, Seite 13] genannt, es wird durch ein Tripel von Worten definiert:

- Name — die Adresse einer FORTH-Zeichenkette – eine 2 Byte Längenangabe gefolgt von der Zeichenkette. Über diesen Namen kann das Token referenziert werden.
- Typ — der Typ des Token
- Datenfeld — ein Wert einer Konstante bzw. die Adresse einer Funktion oder

1.3 Das Dictionary

Variable

Folgende Typen sind, in Tabelle 44 angeführt, möglich:

Wert	***Bedeutung***
1	CONSTANT – ein konstanter Wert
2	VARIABLE – die Adresse einer Variable
3	PROCEDURE - die Adresse einer Funktion (kein Standard)
4	VALUE – ein modifizierbarer Wert
5	MODULE - Basisadresse einer Anwendung (kein Standard)
6	MARKER – speichert die momentane Startadresse des freien Heap für spätere Wiederherstellung
7	2VARIABLE – die Adresse eines Doppelwortes
sonst	undefiniert

Tabelle 44: Datentypen von FORTH

Folgende Befehle sind in der Quelldatei *dictionary.fs* zu finden, Standardbefehle sind: FIND, EXECUTE. Im Assembler sind zusätzliche Makros, Tabelle 45, definiert.

Befehl	***Bedeutung***
create	datafield type adr -- Die Parameter definieren ein Token, das einzutragen ist. Diese Funktion sollte nicht direkt, sondern nur über das Makro CREATE, aufgerufen werden.
forget	adr -- Parameter ist der Name des zu löschenden Eintrags, in Form einer FORTH-Zeichenkette. Diese Funktion sollte nicht direkt, sondern nur über das Makro FORGET, aufgerufen werden.

Tabelle 45: Makros für das Dictionary

Das Dictionary ist bloß eine unsortierte konsekutive Liste, an deren einem Ende Einträge angehängt werden. Zu löschende Einträge werden nicht notwendiger Weise sofort gelöscht, sondern als undefiniert markiert. Sollten alle Einträge, die auf den aktuell zu löschenden folgen, undefiniert sein, wird die Liste verkürzt.

Wird ein Token vom Type MODULE exekutiert, wird die Liste sofort verkürzt, das heißt, auch alle auf das Modul folgenden Einträge werden sofort gelöscht.

Das BIOS reserviert für den Heap und das Dictionary einen gemeinsamen Speicherblock. Der Heap beginnt an der niederwertigen Blockadresse und wächst Richtung höherwertige Blockadresse, das Dictionary beginnt an der höherwertigen Blockadresse und wächst Richtung niederwertige Blockadresse, wodurch eine bestmögliche Ausnutzung des reservierten Blocks möglich ist.

1.4 Laden, Binden und initialisieren von Anwendungen

Nach dem Empfang einer Objektdatei, das Laden, muss diese lauffähig gemacht werden. Das geschieht in 3 Stufen. Zuerst wird der Initialteil, üblicherweise Deklarationen von Variablen und Eintragung der neuen Befehle ins Dictionary, durch Auflösen der Referenzen lauffähig gemacht und exekutiert. Anschließend werden die Referenzen des residenten Teil des Decks aufgelöst, und Initialteil und Referenzliste freigegeben.

Grammatik eines Decks:

deck ← length resident length initial references flag

flag ... 0 nur exekutieren, -1 exekutieren und resident halten

length ... die Länge des folgenden Blocks, ein Integer

resident ← modulename [code]

initial ← code

modulename ← forthstring

code ← (any | unresolved) [code]

unresolved ← local | dictionary

local ← null

dictionary ← stack null zero

null ← zero zero zero zero

zero ← (byte)0

stack ← zero | (byte)0x80

any ... irgendein Bytewert

references ← { record }

record ← addr [forthstring align length] { offset }

offset ← positive | negative

addr ... die relative Adresse des Bezeichners forthstring im Deck bzw. -1, wenn
er einen Dictionaryeintrag bezeichnet

forthstring ... der Bezeichner – 2 byte Längenangabe gefolgt von der Zeichenkette

align ... Füllzeichen garantieren eine Wortgrenze als Position von *length*

positive ... die relative Adresse eines *unresolved* im Deck, Fall *local* liegt vor

negative ... die relative Adresse eines *unresolved* im Deck, Fall *dictionary* liegt vor

Ist *addr* nicht -1, liegt auf jeden Fall *local* vor und ist durch die absolute Adresse des referenzierten Bezeichners zu überschreiben. Liegt Fall *negative* für *offset* vor – nur dann folgt auf *addr* eine Namensangabe (forthstring) –, ist *dictionary* gemäß Tabelle 46 zu überschreiben.

Typ	***Ersetzung***
CONSTANT	(stack | VAL), Datenfeld
VARIABLE	(stack | VAL), Datenfeld
PROCEDURE	(stack | TRAP), Adresse des Datenfeldes
VALUE	(stack | VAL), Adresse des Datenfeldes, (stack | VAL)
MODULE	(stack | VAL), Adresse des Datenfeldes
MARKER	(stack | VAL), Adresse des Datenfeldes
2VARIABLE	(stack | VAL), Datenfeld

Tabelle 46: Ersetzungen für die einzelnen Datentypen

(„|“ ist ein Verkettungsoperator, *stack* (0 oder 1) gibt den Stapel an, für VAL bzw. TRAP ist der entsprechende Befehlscode zu verwenden.)

Im Fall *positive* ist *local* durch die Adresse des Datenfeldes zu überschreiben.

Für eine in sich geschlossene Anwendung, also ein einziges Deck, ist diese Strategie ausreichend und wird durch den Code 254 initiiert. Besteht eine Anwendung aus einer Serie von Decks, Code 243, können einzelne Referenzen möglicherweise nicht unmittelbar aufgelöst werden. Diese Referenzen werden, gemeinsam mit der Startadresse des Decks im Speicher, an den Client mit der Kennung *ERRORREC* zurückgesendet und dort in einer Liste gesammelt. Nachdem des letzte Deck der Serie gebunden wurde, wird das nachträgliche Binden dieser Liste durch Senden von Code 244 eingeleitet und versucht die noch offenen Referenzen aufzulösen.

1.5 Speicherverwaltung mit Reference Counting

Standard sind: UNUSED, modifizierte Standards sind ALLOCATE, FREE, RESIZE [2, Kapitel 14.6.1, Seite 87]. Die Modifikation besteht darin, dass die Speicherverwaltung dem Anwender einen Speicherblock nicht direkt zur Verfügung stellt, sondern über einen Handle. Der Aufbau eines Handle ist in Tabelle 47 definiert.

Wort	***Bedeutung***
0	Der Referenzenzähler
1	Startadresse des Datenbereiches des allozierten Speicher-blocks

Tabelle 47: Der Aufbau eines Speicherhandle

Diese Lösung ermöglicht erst eine Speicherverdichtung! Sollte eine Speicheranforderung nicht unmittelbar befriedigt werden können, wird die Verdichtung automatisch aufgerufen. Diese verdichtet die belegten Blöcke durch Verschieben ans Ende des dynamischen Speichers. So entsteht am Anfang des dynamischen Speichers ein einziger großer freier Block, der die Anforderung wahrscheinlich erfüllen kann.

Java erwartet eine Speicherverwaltung, die selbständig erkennt, wann ein belegter Block tatsächlich freigegeben werden kann. Eine sehr alte Methode, die dies unterstützt, ist das Zählen der Referenzen [8, Kapitel 2.1, Seite 19 - 25], eine Aufgabe, die von der Anwendung erledigt werden muss – die Speicherverwaltung muss das aber unterstützen. Kurz, ein Block wird nur dann in die Liste der freien Blöcke übernommen, wenn sein Referenzenzähler den Wert 0 erreicht hat, ansonsten wird der Zähler bloß erniedrigt. Vorteil dieser Methode ist die Aufteilung der Verwaltungsarbeit, sie wird nur dann ausgeführt, wenn sie tatsächlich notwendig wird. Andere Verfahren benötigen einen eigenen periodischen Prozess der Speicherbereinigung, der nicht unterbrochen werden darf, was nicht wünschenswert erscheint.

Reference Counting hat auch einen Nachteil. Ein Kreis in einem Graphen kann nicht selbständig erkannt und aufgelöst werden. Der Anwendungsprogrammierer benötigt bei Verwendung von Graphen, in den betreffenden Klassen je eine Methode *dispose*, die die Zeiger der Knoten des Graphen auf *null* setzt, wodurch eventuelle Kreise garantiert aufgelöst werden. Trotzdem halte ich diese Methode für die am besten Verträglichste, mit einem Minimum an Implementierungsaufwand, und am ökonomischsten hinsichtlich des Speicherbedarfs – wird ein Block nicht mehr gebraucht, wird er freigegeben und steht sofort wieder zur Verfügung.

Der dynamische Speicher ist ein geschlossener Speicherblock, der über die Dienste angelegt, freigegeben und initialisiert werden kann. Es ist nur ein dynamischer Speicher möglich. Im ersten Wort des Blocks steht die Zahl der angelegten Handle, im zweiten Wort ist der Kopf der Liste der freien Blöcke gespeichert. Ab dem dritten Wort steht die konsekutive Liste der Handle, die beliebig wachsen kann. Der Rest des Blocks zerfällt in belegte, nicht zusammenhängende Bereiche, die über die Handle zugänglich sind. Die unbelegten Bereiche sind in einer doppelt verketteten Liste organisiert. Ein freier Block hat folgenden Aufbau, Tabelle 48.

Wort	***Verwendung***
0	Adresse des nächsten freien Blockes
1	Länge des Blockes
2	Adresse des vorhergehenden freien Blockes
Ab 3	ungenutzt

Tabelle 48: Der Aufbau eines freien Speicherblocks

Ein belegter Block ist in Tabelle 49 festgelegt.

Wort	***Verwendung***
0	Adresse des Handle
1	Länge des Blockes
Ab 2	Datenbereich

Tabelle 49: Der Aufbau eines belegten Speicherblocks

1.5.1 Allozieren

Zuerst wird ein freier Handle gesucht bzw. angelegt und ein bestmöglich passender freier Block gesucht. Ist keiner vorhanden, wird nach einer Speicherbereinigung erneut gesucht, bei Misserfolg wird abgebrochen. Ansonsten wird der Block ausgekettet, angepasst, initialisiert, die Liste der freien Blöcke modifiziert und die Adresse des Handle zurückgegeben.

1.5.2 Freigeben

Der Zähler wird erniedrigt, und nur wenn er 0 wird, in die Liste der freien Blöcke, nach eventueller Verschmelzung mit den Nachbarn, eingetragen. Der Handle wird nicht freigegeben, kann aber wiederverwendet werden.

1.5.3 Größe ändern

Der Datenbereich des Blocks wird auf den Heap kopiert, und der Block frei gegeben. Ein neuer Block der gewünschten Größe wird alloziert, der Datenbereich des bisherigen Blocks vom Heap kopiert, und der alte Handle, der nun den neuen Datenbereich enthält, zurückgegeben.

Zusätzliche Befehle sind in Tabelle 50 festgehalten.

Befehl	***Bedeutung***
MALLOC	Size – handle flag diese Funktion ist ident mit ALLOCATE, aber speziell für Java adaptiert
ALLOCATED-SIZE	handle – size gibt die Größe des Datenbereichs, ohne Verwaltungsdaten, zurück
HANDLEVALID	handle – flag wenn es sich um eine gültige Adresse handelt, wird -1, sonst 0, zurückgegeben
INCREFERENCE	handle -- erhöht den Referenzenzähler
DECREFERENCE	handle -- erniedrigt den Referenzenzähler. Wird 0 erreicht, wird der Block freigegeben

Tabelle 50: Zusätzliche Funktionen für die Speicherverwaltung

1.5.4 Zeiten für INCREFERENCE, MALLOC und DECREFERENCE

Es gelten folgende Abkürzungen:

- N die Größe des dynamischen Speichers in Worten
- h die Zahl der angelegten Handle
- m die Zahl der belegten Blöcke
- b der Speicherbedarf aller belegter Blöcke
- f die Zahl der freien Blöcke

Außerdem gelte die Annahme, dass ein Befehl in einem Takt abgearbeitet ist.

Die Konstanten repräsentieren die ausgezählte Zahl der Befehle einer Sequenz, die Variablen repräsentieren die Zahl der Schleifendurchläufe einer Sequenz.

Die Zahl der Befehle für die verborgene Prozedur zur Speicherverdichtung:

COMPACTPOOL = 56 + 45 * f + 11 * b + 15 * m

Die Zahl der Befehle für die verborgene Funktion zur Handlesuche:

$GETFREEHANDLE_{usually} = 88 + 8 * h$

$GETFREEHANDLE_{worst} = 88 + 8 * h + COMPACTPOOL$

1.5.4 Zeiten für INCREFERENCE, MALLOC und DECREFERENCE

Die Zahl der Befehle für die verborgene Funktion zur Ermittlung des letzten besten freien Blocks:

SEARCHLAST = 13 + 23 * f

Die Zahl der Befehle für die verborgene Funktion zur Ermittlung des besten freien Blocks:

$BESTFREEBLOCK_{usually}$ = 53 + SEARCHLAST

$BESTFREEBLOCK_{worst}$ = 53 + SEARCHLAST + COMPACTPOOL + SEARCHLAST

Die Zahl der Befehle für die Funktion MALLOC:

MALLOC(n) = 70 + GETFREEHANDLE + BESTFREEBLOCK + (n + 3) / 4 * 11

Die Zahl der Befehle für die Funktion DECREFERENCE:

$DECREFERENCE_{decrement}$ = 48

$DECREFERENCE_{destroy}$ = 48 + 188 + 32 * f

Die Zahl der Befehle für die Prozedur INCREFERENCE:

INCREFERENCE = 38

1.5.5 Abschließende Bemerkung

Die implementierte Speicheraufteilung ist MS-DOS nachempfunden. Hinsichtlich der Speicherausnutzung ist die implementierte Strategie optimal! Sie kann aber problemlos durch eine andere Speicheraufteilung ersetzt werden. Die Implementierung startet mit MOVEBLOCKUP einschließlich und endet bei UNUSED ausschließlich und nimmt 1193 Byte des BIOS ein. Für eine andere Implementierung sind knapp 2 kB im ROM, mit einer Kapazität von 4096 Worten, verfügbar. Wird mehr Platz benötigt, ist die Kapazität, wie in Kapitel 3.4 beschrieben, zu erweitern..

1.6 Standardeingabe und Standardausgabe

In *consoleIn.fs* sind folgende Befehle: KEY?, KEY, WORD, PARSE, RESTORE-INPUT, SAVE-INPUT, QUERY, REFILL, ACCEPT, EXPECT.

In *consoleOut.fs* sind folgende Befehle: EMIT?, EMIT, TYPE, SPACES, SPACE, CR. Eine Erweiterung ist in Tabelle 51 festgehalten.

Befehl	***Bedeutung***
PRINT	addr length -- Die Bytefolge, beginnend bei addr, mit der Länge length, besteht aus Zeichen im Unicode, und wird auf der Console angezeigt.

Tabelle 51: Zusätzliche Funktionen für die Console

1.7 Serielle Schnittstelle

Die lokalen Serviceroutinen SERIALIN bzw. SERIALOUT bedienen die Unterbrechungen

des UART. Die Senderoutine entnimmt dem Sendepuffer bloß das nächste Zeichen, wenn eines vorhanden ist, und sendet es über den UART. Die Empfangsroutine erkennt die Anforderung von Diensten und wertet Steuerdaten aus, alle anderen Daten werden unmaskiert in den Ringpuffer des aktuellen Pufferhandle eingetragen – Standard ist 0, die Consoleneingabe. Die Einbindung des UART ist in Tabelle 52 dokumentiert.

Befehl	***Bedeutung***
SETSIO	Stoppbits parity wordlength baudrate -- definiert die Betriebsart des UART, wobei: baudrate die Baudrate wordlength 7 oder 8, die Wortlänge in Bit parity 0 für kein Paritätsbit, 1 für ungerade Parität, 2 für gerade Parität Stoppbits 4 für 2 Stoppbit, 3 für 1.5 Stoppbit, 2 für 1 Stoppbit
GETSIO	-- Stoppbits parity wordlength baudrate gibt die momentane Einstellung des UART zurück
REDEFINESIO	Stoppbits parity wordlength baudrate -- neu Initialisierung der Schnittstelle

Tabelle 52: Zusätzliche Funktionen zur Einbindung des UART in FORTH

1.8 Zähler und Zeitgeber

Die Prozeduren zur Einbindung der Zähler ist in Tabelle 52 dokumentiert.

Befehl	***Bedeutung***
TRIGGER-SYSCLOCK	-- mode Der Zähler soll den Systemtakt teilen
TRIGGER-PRESCALER	prescaler -- mode Der Zähler verwendet den Zähler *prescaler* als Vorteiler
TRIGGER-INPUT	-- mode Der Zähler verwendet den externen Zähleingang
LOCK-COUNTER	-- mode Der Zähler soll gesperrt werden

Befehl	***Bedeutung***
SETCOUNTER	counter mode reloadvalue -- Startet den Zähler counter in der Betriebsart mode und dem Anfangswert reloadvalue
READCOUNTER	counter – count Liest den aktuellen Zählerstand des Zählers *counter*
SET-COUNTER-SERVICE	serviceroutine counter -- Ordnet dem Zähler *counter* eine Serviceroutine zu und demaskiert sein Unterbrechungssignal

Tabelle 53: Zusätzliche Funktionen zur Einbindung des Zählerbausteins in FORTH

1.9 Unterbrechungen

Stellt die Standardbefehle ABORT und QUIT bereit, sowie zusätzlich gemäß Tabelle 54.

Befehl	***Bedeutung***
SETVECTOR	serviceroutine number -- Ordnet der Unterbrechung *number* die Serviceroutine zu
GETVECTOR	number – serviceroutine Gibt die Serviceroutine der Unterbrechung *number* zurück
SOFTINT	-- erzwingt einen Software-Interrupt (Serviceroutine 1).

Tabelle 54: Zusätzliche Funktionen zur Einbindung des Interrupt Controllers in FORTH

1.10 Java

Einige Routinen, die jede Java-Anwendung benötigt, sind in Tabelle 55 angeführt.

Befehl	***Bedeutung***
INSTANCEOF	object forthstring – castedobj Ein Objekt soll in seine, durch *forthstring* bezeichnete, Superklasse gewandelt werden und bei Erfolg das gewünschte Superobjekt zurückgeben, sonst 0.

Befehl	***Bedeutung***
CASTTO	object forthstring – castedobj Ein Objekt soll in seine, durch *forthstring* bezeichnet, Superklasse bzw. abgeleitete Klasse gewandelt werden. Bei Erfolg das gewünschte Objekt zurückgegeben, sonst 0.
EXECUTE-NEW	table hashcode – obj Der durch *hashcode* identifizierte Konstruktor der Methodentabelle einer Klasse soll aufgerufen werden und eine neue Instanz zurückgeben. Die Tabelle ist eine Liste von Tupeln der Gestalt (hashcode, routine).
EXECUTE-METHOD	object hashcode polymorph – Ergebnis der Methode Die durch *hashcode* identifizierte Methode des Objekts bzw. seiner Superobjekte soll ausgeführt werden, wenn gewünscht, was meist der Fall ist, soll der Polymorphie Rechnung getragen werden.

Tabelle 55: Zusätzliche Funktionen zur Unterstützung von Java

Für die obersten 30 Bit des Hashcode gibt es keine Vorgaben, was sie enthalten und wie sie bestimmt werden, ist Sache des Compilers. Für die beiden niedrigsten Bit ist die Vorgabe in Tabelle 56 festgelegt.

Bit	***Vorgabe***
0	1 für PRIVATE, sonst 0
1	1 für PROTECTED, sonst 0

Tabelle 56: Vordefinierte Bit eines Hashcodes für Java

1.11 Mathematik

1.11.1 Einfach ganzzahlig

*, /MOD, /, MOD, */MOD, */ [2] verarbeiten einfach ganzzahlige Operanden und liefern einfach ganzzahlige Ergebnisse, M* liefert ein doppelt ganzzahliges Ergebnis.

1.11.2 Doppelt ganzzahlig

>NUMBER, CONVERT konvertieren eine Zeichenkette in einen doppelt ganzzahligen Wert.

UM/MOD, SM/REM, FM/MOD liefern einfach ganzzahlige Ergebnisse. Zusätzliche Funktionen sind in Tabelle 57 dokumentiert.

Befehl	***Bedeutung***
UD/DMOD	uddividend uddivisor – udremainder udquotient dividiert vorzeichenlose doppelt ganzzahlige Operanden
UD/MOD	uddividend udivisor – uremainder udquotient dividiert einen vorzeichenlosen doppelt ganzzahligen Operanden durch einen einfach ganzzahligen Operanden
ASHIFT	d shiftfactor – d ein doppelt ganzzahliger Wert wird arithmetisch nach rechts verschoben, der Schiebewert ist vorzeichenbehaftet
SHIFTL	ud shiftfactor – ud ein doppelt ganzzahliger Wert wird nach links verschoben, der Schiebewert ist vorzeichenbehaftet
SHIFTR	ud shiftfactor – ud ein doppelt ganzzahliger Wert wird nach rechts verschoben, der Schiebewert ist vorzeichenbehaftet
D*	d1 d2 – d3 multipliziert zwei doppelt ganzzahlige Operanden
D/	ddividend ddivisor – dquotient dividiert zwei doppelt ganzzahlige Operanden
DMOD	ddividend ddivisor – dremainder modulo zweier doppelt ganzzahliger Operanden

Tabelle 57: Zusätzliche arithmetish logische Funktionen

1.11.3 Formatierung und Ausgabe

<#, HOLD, SIGN, #, #S, #>, U.R, U., .R, ., D., D.R sind Standardfunktionen. Zusätzliche Funktionen sind in Tabelle 58 dokumentiert.

Befehl	***Bedeutung***
..R	value width – addr length wie .R, nur wird hier die Länge und die Adresse der konvertierten Darstellung retourniert
U..R	uvalue width – addr length wie U.R, nur wird hier die Länge und die Adresse der konvertierten Darstellung retourniert
D..R	dvalue width – addr length wie D.R, nur wird hier die Länge und die Adresse der konvertierten Darstellung retourniert
UD..R	udvalue width – addr length wie U.R, nur wird hier die Länge und die Adresse der konvertierten Darstellung retourniert

Tabelle 58: Zusätzliche Funktionen zur Formatierung von Zahlen

1.11.4 Doppelt genaue Gleitkommafunktionen

Als Format wird der Standard ANSI 754 [12] verwendet. Die vier Grundrechenoperationen liefern Resultate mit einem Fehler von höchstens ½ Bit. Alle höheren Funktionen brechen spätestens nach 50 Iterationen ab, bzw. früher, wenn weitere Iterationen keine Veränderung des Ergebnisses bringen können.

Die Grundoperationen F*, F/ implementieren quadratische Algorithmen, alternativ kommt für die Multiplikation der Algorithmus von Schönhage-Strassen [14, Theorem 7.8, Seite 273] in Frage. Die am besten geeignete Methode ist auszuwählen. Es gilt:

n, die Problemgröße: n = 53, die Länge der Mantisse in Bits

a, die Zahl der 32-Bit-Worte für die Mantisse: a = ceil(n/32) = 2

Q(n), die Komplexität des quadratischen Algorithmus: Q(n) = 3 * a * n

S(n), die Komplexität von Schönhage-Strassen: S(n) = O(n * ld(n) * ld(ld(n))).

Gilt Q(53) < S(53), dann ist das quadratische Verfahren besser

3 * a * 53 < 53 * ld(53) * ld(ld(53))

318 < 53 * 5,72792 * 2,518

318 < 764,41738

Damit sind die quadratischen Algorithmen, bei einer Mantissenlänge von 53 Bit, effizienter als Schönhage-Strassen, der erst ab einer Mantissenlänge > 8 * 32 = 256 Bit besser sein kann.

Folgende, im Standard gelistete Funktionen, sind in der Quelldatei *fmath.fs* zu finden: F*,

F/, F+, F-, FNEGATE, FMAX, FMIN, FLOOR, FROUND, F**, FACOS [= π / 2 - FASIN(x)], FACOSH [13, Kapitel 4.6.21, Seite 87], FASIN [= FATAN(x / FSQRT(1 – x^2))], FASINH [13, Kapitel 4.6.20, Seite 87], FALOG, FATAN [13, Kapitel 4.4.42, Seite 81], FATAN2, FATANH [13, Kapitel 4.6.22, Seite 87], FCOS [13, Kapitel 4.3.66, Seite 74], FCOSH [13, Kapitel 4.5.2, Seite 83], FEXP [13, Kapitel 4.2.1, Seite 69], FEXPM1, FLN [13, Kapitel 4.1.25, 68], FLNP1, FLOG, FSIN [13, Kapitel 4.3.65, Seite 74], FSINH [13, Kapitel 4.5.1, Seite 83], FSINCOS, FSQRT [18, Kap 1.3, Seite 11 - 15], FTAN [13, Kapitel 4.3.67, Seite 75], FTANH, F~, F. FS., FE., >FLOAT, D>F, F>D. Zusätzliche Funktionen sind in Tabelle 59 dokumentiert.

Befehl	***Bedeutung***
RECIPROCAL	df – df berechnet den Kehrwert (1/x) [18, Kapitel 1.2, Seite 2 - 10] nach Newtons Methode und verwendet bloß Addition, Subtraktion und Multiplikation – keine Division
F..	df – addr length wie F., nur wird hier die Länge und die Adresse der konvertierten Darstellung retourniert
FS..	df – addr length wie FS., nur wird hier die Länge und die Adresse der konvertierten Darstellung retourniert
FE..	df – addr length wie FE., nur wird hier die Länge und die Adresse der konvertierten Darstellung retourniert

Tabelle 59: Zusätzliche Funktionen zur Behandlung von Gleitkommazahlen

1.11.5 Eine Verbesserung des quadratischen Divisionsalgorithmus

Es wird vorausgesetzt, daß eine n-Bit Addition bzw. Subtraktion in einem Schritt durchgeführt werden kann. Dann ist für den Multiplikationsalgorithmus die Zahl der Operationen ≤ 3 n, beim Divisionsalgorithmus für vorzeichenlose ganze Zahlen die Zahl der Operationen ≤ 4 n + ld(n), was merklich schlechter ist. Es ist aber eine Verbesserung auf ≤ 3 n + ld(n) möglich.

Seien dividend, divisor, m, n beliebige natürliche Zahlen, dann gilt:

$$divisor * 2^n > dividend \geq divisor * 2^m \wedge divisor * 2^{m+1} > dividend \Rightarrow$$
$$dividend - divisor * 2^n + \sum_{i=m}^{n-1} divisor * 2^i = dividend - divisor * 2^m$$

Das heißt, man subtrahiert den Divisor im n-ten Schritt, stellt fest, die Differenz ist kleiner 0, addiert in den folgenden Schritten den Divisor solange, bis die Summe größer gleich 0

ist, wodurch das begrenzende größte m ermittelt ist. Die Quotientenbits n bis m+1 sind dann 0, das Bit m 1. Es entfällt also die ansonsten notwendige, aber als unangenehm empfundene, Korrekturaddition bei einem misslungenen Subtraktionsversuch, dadurch verringert sich die Zahl der Operationen eines Iterationsschrittes von 4 auf 3. Man benötigt daher in einem Schritt entweder eine Subtraktion oder eine Addition, sowie je eine Schiebeoperation für den Divisor und den Quotienten.

Meine Recherchen, hinsichtlich dieser Modifikation, im Internet waren erfolglos. Ich schließe daraus, dass diese einfache Tatsache nicht allgemein bekanntes Wissen ist! Der quadratische Divisionsalgorithmus ist nicht deutlich schlechter als die Multiplikation.

2 Der Client

Er implementiert das Protokoll zur Kommunikation mit dem Server, ebenso die Remote Procedures, die dem Server eine Dateiverwaltung ermöglichen – alles in der Klasse *Serial* im Package *communicate*. Darüber hinaus einen Assembler, einen Java-Compiler und eine Benutzerschnittstelle.

2.1 Der Assembler

Der Assembler [2, Kapitel 6 - 14, Seite 21 – 87] generiert aus Quelldateien ein Deck – bedeutet eine oder mehrere Objekdateien – das den Anforderungen des BIOS genügt. Die Arbeitsfolge der Übersetzung

Quelldatei → Scanner → Parser → Codeerzeugung → Deck

Der Parser ist der steuernde Teil. Er fordert den Scanner auf die Token aus dem Quellstrom zu extrahieren, erkennt die Korrektheit bzw. meldet Fehler, veranlasst die Codeerzeugung, und schreibt am Ende der Übersetzung den Code in eine Zieldatei.

2.1.1 Der Scanner des Assemblers

Dem Scanner werden in einem Initialisierungsschritt die Quelldateien mitgeteilt, mit denen er den Stapel der Quelldateien initialisiert. Eine Quellzeile hat folgende Struktur.

sourceline ← { whitespace | include | comment | undefine | { "A:" } macro | { "A:" } token }

[define | endcomment] "\n"

wobei *whitespace* die Trennzeichen definiert. Zu beachten ist, dass Terminalsymbole durch *whitespace* separiert sein müssen, da sie sonst als Bezeichner *ident* erkannt werden. Terminalsymbole sind in FORTH Bezeichner mit einer speziellen Bedeutung.

whitespace ← [\0-] -- alle Steuerzeichen und das Leerzeichen

Mit *include* kann eine weitere Quelldatei unmittelbar in die aktuelle Quelldatei eingefügt werden. Diese Direktive legt eine neue Datei auf den Stapel der Quelldateien. Gelesen wird immer von der obersten Datei. Erst wenn diese vollständig gelesen wurde, wird sie vom Stapel genommen.

include ← "INCLUDE" ident

ident ← [!-~]+ -- jede Kette von druckbaren Zeichen ohne Leerzeichen

Kommentare können beliebig im Quelltext vorkommen, bewirken aber keinerlei Aktion.

Comment ← "(" charsequence ")"

endcomment ← "\" [^\n]* -- alles, bis zum Zeilenende

charsequence ← [-~\t\n]+ -- jede Kette von druckbaren Zeichen

-- Bem.: besitzt immer ein begrenzendes Zeichen, z.B. ")".

-- Kommt dieses Zeichen freistehend in der

2.1.1 Der Scanner des Assemblers

Sequenz vor,

-- ist es mit einem vorangestellten "\" zu maskieren.

Es ist möglich eigene Makros zu definieren und wieder zu löschen. Ein Makro ist eine benannte Textersetzung, die einmal definiert, beliebig in den Quelltext eingefügt werden kann und dabei vom Scanner durch den assoziierten Text ersetzt wird. Makros können auch andere Makros aufrufen, sind aber nicht rekursiv. Die Namensgebung ist nicht eingeschränkt, deshalb kann wirklich alles durch ein Makro überschrieben werden. Makros sind nicht im Standard definiert!

define ← "DEFINE" userdefined expansion

undefine ← "UNDEFINE" userdefined

userdefined ← ident -- der Bezeichner der Textersetzung

expansion ← [^\n]* -- der Rest der Quellzeile

Zusätzlich gibt es vordefinierte Makros, die aber nicht gelöscht werden können. Alle Makros, auch die selbst definierten, können den alternativen Stapel mit einer ungeraden Anzahl von vorangestellten Präfixen „A:“ wählen. Eine gerade Anzahl, einschließlich 0, wählt den Standardstapel.

macro ← "EI" | "DI" | "QI" | "PI" | "SETIMASK" | "GETIMASK" | "SP!" | "," | ">BODY" |
"+!" | "C+!" | "H+!" | "2!" | "2@" | "2DUP" | "2OVER" | "2SWAP" | ">R" | "?DUP" |
"ABS" | "ALIGN" | "ALIGNED" | "ALLOT" | "C," | "BL" | "CELLS" | "CHAR+" |
"CHARS" | "HALVES" | "COUNT" | "DECIMAL" | "ERASE" | "FILL" | "HEX" | "I" |
"J" | "MAX" | "MIN" | "MOVE" | "R>" | "ROLL" | "ROT" | "S>D" | "SWAP" | "2>R" |
"UNLOOP" | "2R>" | "2R@" | "FALSE" | "TRUE" | "TUCK" | "WITHIN" | "D+" |
"D-" | "D0<" | "D0=" | "D2*" | "D2/" | "D<" | "D>" | "D=" | "D>S" | "DABS" |
"DMAX" | "DMIN" | "2ROT" | "DU<" | "DU>" | "F!" | "F@" | "FALIGNED" |
"FDEPTH" | "FDROP" | "FDUP" | "FLOAT+" | "FLOATS" | "F0<" | "F<" | "F0=" |
"FOVER" | "FSWAP" | "DF!" | "DF@" | "DFALIGNED" | "SFALIGNED" | "FABS" |
"SFLOAT+" | "DFLOAT+" | "DFLOATS" | "SFLOATS" | "TIB" | "#TIB" | ">IN" |
"<" | ">" | "=" | "<>" | "<!" | ">!" | "=!" | "<>!" | "U<" | "U>" | "U<!" | "U>!" |
"SOURCE" |
"ABORT"" charsequence """ |
".(" charsequence ")" |
"[CHAR]" charsequence |
"[']" charsequence |
"CONSTANT" ident |
"CREATE" ident |

"VARIABLE" ident |
"2VARIABLE" ident |
"PROCEDURE" ident |
("LITERAL" | "VALUE") ident |
"MODULE" ident |
"MARKER" ident |
"FORGET" ident |
userdefined

Wird CONSTANT, VARIABLE, CREATE, 2VARIABLE, PROCEDURE, MODULE oder MARKER im Quelltext vorgefunden, kennzeichnet der Scanner den Quellcode als resident. Diese Kennzeichnung ist für das Deck verbindlich.

Die Ermittlung der Token ist der eigentliche Zweck des Scannen. Auch Token können den Stapel mit dem vorangestellten Präfixen „A:“ wählen.

token ← (executable | ident | number | floating | string | control | EOF)

Executable ← "NOP" | "!" | "@" | "C!" | "C@" | "H!" | "H@" | "DEPTH" | "DROP" | "2DROP" |
"NIP" | "PICK" | "PUT" | "VAL" | "DUP" | "OVER" | "R@" | "R1@" | "SAVE" |
"HALT" | "_SP!" | "SP@" | "+" | "-" | "1+" | "1-" | "2*" | "2/" | "AND" | "CELL+" |
"HALF+" | "INVERT" | "LSHIFT" | "NEGATE" | "OR" | "RSHIFT" | "XOR" |
"+B" | "-B" | "LSHIFTC" | "RSHIFTC" | "0<!" | "0<" | "0=!" | "0=" | "0<>!" | "0<>" |
"0>!" | "0>" |"H*" | "EXIT" | "CALL" | "TRAP" | "BRANCH" |
"0BRANCH!" | "0BRANCH" | "B@" | "B!" | "BREAK"

number ← [+-] ([0-9]+ | [0-9a-fA-F]+"H")

floating ← [+-][0-9]+\.[0-9]*([eE][+-][0-9]+)?

string ← "U"" charsequence """ | -- Zeichenkette aus Zeichen des Unicodes
"S"" charsequence """ |
"C"" charsequence """ |
""" charsequence """

control ← "IF" | "ELSE" | "THEN" | "ENDIF" |
"?DO" | "DO" | "+LOOP" | "LOOP" |
"BEGIN" | "WHILE" | "REPEAT" | "UNTIL" | "AGAIN" |
"LEAVE" |
"CASE" | "OF" | "ENDOF" | "ENDCASE" |

"TO" |
"LOCAL" | "2LOCAL" | "LOCALS" | "2LOCALS" | "|" | "PURGE" |
"THROW" | "CATCH" |
"$ORG" | -- absolute Startadresse des Codes
"MODULENAME" -- Name, unter dem das erzeugte Modul im Dictionary
-- eingetragen wird

EOF ← alle Quelldateien gelesen, der Stapel der Quelldateien ist leer

Die Daten eines Token.

```
public class Token
{
  String sourceFile;                  // filename
  public boolean alternate;           // marks alternative stack
  public int kind;                    // type of token
  public int line;                    // sourceline number
  public int col;                     // column in sourceline
  public long val;                    // number
  public String ident;                // identifier
  public String string;               // read string
  double d;                           // floating
}
```

Erweiterungen des Standards, die den Scanner betreffen, sind in Tabelle 60 festgehalten.

Sprachelemente	***Erklärung***
"U"" charsequence """"	Definiert eine Zeichenkette im Unicode. Wird von Java benötigt.
"DEFINE" userdefined expansion	Definiert ein Makro, eine reine Textersetzung.
"UNDEFINE" userdefined	Löscht die Definition eines Makro

Tabelle 60: Nicht in FORTH definierte Token

2.1.2 Der Parser des Assemblers

Die Regeln der Grammatik des Parsers.

parse ← ["MODULENAME" ident] { statements | colonDefinition } .

statements ← { { "A:" } ("NOP" | "!" | "@" | "C!" | "C@" | "H!" | "H@" | "DEPTH" | "DROP" |
"2DROP" | "NIP" | "PICK" | "PUT" | "DUP" | "OVER" |

```
                    "R@" | "R1@" | "SAVE" | "HALT" | "_SP!" | "SP@" | "+" | "-" |
"1+" |
                "1-" | "2*" | "2/" | "AND" | "CELL+" | "HALF+" | "INVERT" |
                "LSHIFT" | "NEGATE" | "OR" | "RSHIFT" | "XOR" | "H*" |
                "+B" | "-B" | "LSHIFTC" | "RSHIFTC" | "0<!" | "0<" | "0=!" | "0=" |
                "U0<" | "U0>" | "0<>!" | "0<>" | "0>!" | "0>" | "CMP" | "EXIT" |
                        ( ident | number | build ) ( "VAL" | "CALL" | "TRAP" |
"BRANCH" |
                                                    "0BRANCH!" | "0BRANCH" |
"LABEL" |
                                                "$ORG" ) |
                "B@" | "B!" | "BREAK" ) |
        "CATCH" | "THROW" |
        ( "LOCAL" | "2LOCAL" ) ident |
        ( "LOCALS" | "2LOCALS" ) ident { ident } "|" |
        "PURGE" number |
        {  "A:" } "TO" ( number | ident ) |
        {  "A:" } ( string | floating | ident | number | build | "LEAVE" |
                                        ifStatement | doStatement | caseStatement |
beginStatement ) } .

colonDefinition ← ( ":" | ":LOCAL" ) ident statements {  "A:" } ";" .
ifStatement ← "IF" statements [ "ELSE" statements ] ( "ENDIF" | "THEN" ) .
doStatement ← ( "DO" | "?DO" ) statements ( "LOOP" | "+LOOP" ) .
caseStatement ← "CASE" { statements "OF" block "ENDOF" } "ELSE" statements
"ENDCASE" .
beginStatement ← "BEGIN" statements
                ( "WHILE" statements "REPEAT" | "UNTIL" | "AGAIN" ) .
build ← "BUILD>" expr "DOES>" .
expr ← ( build | number | floating ) { "1+" | "1-" | "2*" | "2/" | "CELL+" | "HALF+" | "INVERT" |
                                        "ABS" | "FABS" | "D>F" | "F>D" | "FLOOR" |
"FROUND" }
        { expr ( "+" | "-" | "AND" | "OR" | "XOR" | "LSHIFT" | "RSHIFT" | "F+" | "F-" | "F*" |
                "F/" | "F**" | "*" | "/" | "MOD" | "FMAX" | "FMIN" | "MAX" | "MIN" )
            { "1+" | "1-" | "2*" | "2/" | "CELL+" | "HALF+" | "INVERT" | "ABS" | "FABS" |
                "D>F" | "F>D" | "FLOOR" | "FROUND" } } .
```

2.1.2 Der Parser des Assemblers

Einige Sprachelemente stehen nicht zur Verfügung, sie sind in Tabelle 61 festgehalten.

Sprachelemente	***Erklärung***
"[" statements "]"	Zweck: Befehle in diesem Block werden nicht compiliert, sondern sofort ausgeführt. Begründung: Ein Compiler kann diese Möglichkeit nicht bieten. Ersatz: solche Blöcke als eigenständige Prozedur definieren oder den Block in den Startcode eines Moduls verlagern
"IMMEDIATE"	Zweck: markiert unmittelbar ausführbare Worte Begründung: alle Worte im Dictionary, sowie Worte im erstellten Modul, sind unmittelbar ausführbar. Ersatz: nicht notwendig, da implizit gegeben.

Sprachelemente	***Erklärung***
"COMPILE" | "[COMPILE]" | "POSTPONE"	Zweck: erzwingt die Erzeugung eines Immediate Wortes Begründung: überflüssig, da alle Worte Immediate sind
"RECURSE"	Zweck: ermöglicht die Rekursion Begründung: alle Prozeduren sind rekursiv aufrufbar
":NONAME"	Zweck: definiert eine namenlose Prozedur Begründung: Worte im Dictionary müssen einen Namen haben Ersatz: benannte lokale Prozedur
"ENVIRONMENT?"	Zweck: Abfragen der Systemumgebung Begründung: Es gibt keine Systemumgebung
"EVALUATE"	Zweck: interpretiert den Inhalt einer Zeichenkette Begründung: Ein Compiler kann diese Möglichkeit nicht bieten. Ersatz: Ein Makro oder eine Prozedur definieren

Tabelle 61: Sprachelemente von FORTH, die nicht realisiert werden konnten

Zusätzlich wurden folgende, im Standard nicht definierte, Sprachelemente gemäß Tabelle 62 aufgenommen:

Sprachelemente	***Erklärung***
"MODULENAME" ident	Das ladbare Modul soll mit einem bestimmten Namen im Dictionary eingetragen werden. Es muss immer die erste Anweisung in der Quelldatei sein.
(ident | number | build) "$ORG"	Die absolute Startadresse des erzeugten Codes wird vorgegeben; hat nur eine Wirkung, wenn die Zieldatei „romable“ sein soll.
(ident | number | build) "LABEL"	Eine Marke, die als Sprungziel verwendet werden kann.
":LOCAL" ident	Ermöglicht die Definition einer lokalen Prozedur, die nicht im Dictionary eingetragen wird.
"PURGE" number	Damit werden die zuletzt deklarierten *number* Variablen unmittelbar gelöscht. Wird von Java benötigt.

Sprachelemente	***Erklärung***
number ["VAL"]	*number* kann auch ein Doppelwort – 64 Bit – repräsentieren. Es wird automatisch Code zum Laden des Doppelwortes erzeugt – eine Modifikation des Standards

Tabelle 62: Zusätzlich definierte Sprachelemente

Alle Regeln, die nach *statements* abgeleitet werden, überwachen die Integrität des Stapels der Rücksprungadressen – die Stapelhöhe darf sich durch die Ableitung nach *statements* nicht verändern –, und geben bei Verletzung dieser Bedingung eine Fehlermeldung aus. Für den Datenstapel ist diese Überwachung nicht möglich, man müsste zur Übersetzungszeit wissen, wie viele Elemente ein Prozeduraufruf vom Datenstapel nimmt und darauf zurück gibt. Wenn die Prozedur im Dictionary steht, ist dieses zur Compilezeit möglicherweise für den Client nicht verfügbar (Offline), und die Zahl der Parameter daher nicht bekannt.

Lokale Variablen können deklariert werden, werden am Stapel der Rücksprungadressen angelegt, und können, wenn einmal deklariert, über den Namen referenziert werden, der notwendige Code wird automatisch erzeugt. Mittels *PURGE* können die Variablen wieder gelöscht werden. Eine automatische Löschung wird auf jeden Fall am Ende einer Prozedur bzw. dem Initialcode vorgenommen, sofern sie nicht schon vorher mit *PURGE* freigegeben wurden.

2.1.2.1 build

Wertet zur Übersetzungszeit arithmetische Ausdrücke aus und kann daher nur Konstante verarbeiten. Das Ergebnis ist ein Token vom Typ *number* oder *floating*.

2.1.2.2 Flusssteuerungen

Setzen die Steueranweisungen in Befehlsfolgen um, die sie in den Quellstrom des Scanners, gleich einem Makro, einfügen.

2.1.2.3 Colon Definitionen

Damit wird eine Prozedur definiert. Wenn sie nicht lokal ist, wird automatisch Initialcode erzeugt, der sie ins Dictionary einträgt. Auf jeden Fall wird die Prozedur in die Referenzliste aufgenommen.

2.1.2.4 Statements

Hier werden die Maschinenbefehle erkannt, übersetzt und an die Codetabellen weitergeleitet. Eine wichtige Teilaufgabe ist die Verwaltung der Referenzen.

2.1.2.5 Parse

Nach dem Übersetzen der Quelldateien wird versucht die Referenzen aufzulösen. Nach diesem Schritt müssen alle Referenzen von Sprungbefehlen sowie CALL durch 16-Bit Offsetwerte ersetzt sein, andernfalls konnte eine lokale Referenz nicht aufgelöst werden. Die restlichen Referenzen müssen vom Binder aufgelöst und gebunden werden. Referenzierte Werte, die im Programm deklariert sind, müssen bloß gebunden werden, ihre Referenzen haben den Typ *local* (BIOS). Die Restlichen müssen im Dictionary zu finden sein. Wenn ihre Verwendung durch VAL bzw. TRAP vorgegeben ist, ist ihr Typ *positive*, ansonsten definiert das referenzierte Token den Typ, und damit den Ladecode, und ist vom Typ *negative*.

Abschließend wird das Deck geschrieben, wenn der Typ der Zieldatei *linkable* ist. Alternativ gibt es den Typ *romable*, der den Code als Vektor von 32-Bit Worten in VHDL-Syntax auf die Zieldatei schreibt. Dieser Code kann direkt in eine VHDL-Beschreibung übernommen werden.

2.1.3 Codeerzeugung

Sie verwaltet zwei Codevektoren, einen für den Initialcode des Programms, und einen für den residenten Teil – Prozeduren und Zeichenketten. Die Auswahl des gewünschten Codevektors nimmt der Parser vor. Es gibt etliche Funktionen, die auf einen Vektor angewendet werden können – anhängen eines Datums, überschreiben an einer bestimmten Position, wahlfreies lesen, etc. Eine wichtige Teilaufgabe ist die Protokollierung der Höhe des Stapels der Rücksprungadressen.

2.1.4 Schnittstelle

Die Klasse *Assembler* des Package *forthassembler* stellt die Methode in Tabelle 63 bereit.

Methode
public static boolean assemble(String option, String args, String editor, String report, boolean applet) assemble forth source to produce targets ending with „.obj“ or „.rom“ and listing ending „.lst“ option String, "linkable" or "romable" args String sourcename editor String name of preferred editor report String name of report file applet boolean true, if started from applet return boolean true, if successful

Tabelle 63: Methode Assembler.assemble zum Aufruf des Assemblers

2.2 Java

Der Compiler übersetzt nicht direkt in Maschinencode, sondern erzeugt aus den Quelldateien Assemblerprogramme, die mit dem Assembler zu übersetzen sind. Vorteilhaft ist, dass das Compilat lesbar und editierbar ist, was eine Überprüfung der Korrektheit der Übersetzung ermöglicht. Der Compiler arbeitet inkrementell. In jedem Schritt wird der Parser aufgerufen, der eine Liste der referenzierten Dateien zurück gibt, woraus ein Referenzdatei mit Endung „.ld“ erstellt wird. Jeder Listeneintrag hat einen Vermerk, ob die Quelldatei bereits übersetzt ist, oder separat übersetzt werden muss. Die Iteration endet, wenn alle benötigten Referenzen übersetzt sind. Die Reihenfolge der Schritte eines Durchlaufs.

Quelldatei → Scanner → Parser → Übersetzung → Zieldateien

Steuernder Teil ist der Parser, er liest die Quelldateien vollständig, erzeugt dabei Operatorbäume und Referenzlisten bzw. Scopes, letztere schreibt er in eine eigene Datei mit Endung „.h“, künftig Header genannt, um sich bei neuerlicher Übersetzung das Parsen ersparen zu können, und löst die Referenzlisten auf. Meist gelingt das nicht in einem Schritt, sondern zusätzliche Quellen aus dem Package und den Importdeklarationen müssen übersetzt werden bzw. deren Scopes geladen werden, falls sie bereits erfolgreich übersetzt wurden. Dies geschieht rekursiv, wobei die abschließende Übersetzung bei importierten Dateien entfällt. Erst wenn alle Referenzen der Quelldatei bzw. der Dateien aus dem Package aufgelöst sind, erfolgt, rekursiv aufsteigend, die Übersetzung – die Auswertung der Header und seiner Operatorbäume zur Erzeugung der Assemblerprogramme. Von jeder Quelldatei werden zwei Zieldateien erzeugt. Das residente Modul mit Endung „.fs“, und der flüchtige, initialisierende Teil – der Code der statischen Blöcke und

Initialisierungen – mit Endung „.start.fs“.

2.2.1 Der Scanner

Er liest die Quelldatei und extrahiert auf Anforderung das nächste Token aus dem Quellstrom. Eine Quellzeile gehorcht folgender Regel.

Sourceline ← { whitespace | token } [endcomment] "\n" .

whitspace ← [\0-]

token ← comment | endcomment | charConst | number | lnumber | stringLiteral | dnumber | ident |

keyword .

Comment ← /*.**/

endcomment ← //[^\n]*

charConst ← \b|\t|\n|\f|\r|\u[0-9a-fA-f][0-9a-fA-f]?[0-9a-fA-f]?[0-9a-fA-f]?|[0-3][0-7]?[0-7]?|

[0-7][0-7]?|.

number ← 0[0-7]*|0x[0-9a-fA-F]+|[0-9]+

lnumber ← (0[0-7]*|0x[0-9a-fA-F]+|[0-9]+)[lL]

stringLiteral ← "(\\.|[^"])*"

dnumber ← [0-]*\.[0-9]*([eE][+-][0-9]+)?[fF]

ident ← [a-zA-Z$_][a-zA-Z$_0-9]*

keyword ← "+" | "-" | "*" | "/" | "%" | "==" | "!=" | "<=" | ">=" | "<" | ">" | "&" | "|" | "^" | "=" |

"!<" | "!>" | "++" | "--" | ";" | ":" | "?" | "(" | ")" | "{" | "}" | "[" | "]" | "." | "!" | "~" | "," |

"break" | "else" | "if" | "new" | "return" | "#ass" | "while" | "for" | "do" | "continue" |

"try" | "catch" | "finally" | "switch" | "throw" | "case" | "default" | "instanceof" |

"this" | "super" | "true" | "false" | "null" | "+=" | "-=" | "*=" | "/=" | "%=" |

"&=" | "|=" | "^=" | ">>>=" | ">>=" | "<<=" | ">>>" | ">>" | "<<" | "package" |

"import" | "extends" | "implements" | "assert" | "throws" | "byte" | "short" | "char" |

"int" | "long" | "float" | "double" | "boolean" | "void" | "public" | "private" | "protected" |

"static" | "final" | "synchronized" | "volatile" | "transient" | "native" | "abstract" |

"strictfp" | "interface" | "class" | EOF .

Kommentare werden nicht überlesen, der Parser fügt diese in den Operatorbaum ein, und der Übersetzer fügt sie als FORTH-Kommentare in das erzeugte Assemblerprogramm ein.

Bei den Schlüsselwörtern wurden *!<* als Alternative zu >= und *!>* für <= zusätzlich definiert. Erfahrungsgemäß lassen sich logische Fehler, die durch Verwechseln von <= mit < bzw.

>= mit > entstehen, oft nicht auf Anhieb ermitteln und erweisen sich als sehr hartnäckig. In FORTH wird <= per Definition durch „> INVERT“ bzw. >= durch „< INVERT“ ausgedrückt, wodurch es, nach eigener Erfahrung, auch zu keinen Verwechslungen von < mit <= bzw. > mit >= kommt. Ich denke, dass die definierten Alternativen leichter zu lesen sind als der Standard und vermutlich – wären sie Standard – ganz allgemein zu weniger logischen Fehlern in Programmen führen würden. <= bzw. >= gänzlich zu streichen, wie in FORTH, halte ich für nicht machbar.

2.2.2 Der Parser

Die dem Parser zu Grunde liegende Grammatik ist eine korrigierte Version einer Grammatik des Informatikinstituts SSW[11] der Johannes Kepler Universität und ist wie die Vorlage LALR(1). Im Gegensatz zur Vorlage, erkennt die neue Grammatik die JDK-Quelldateien des Pakets *j2sdk-sec-1_4_2-src-scsl* von Sun anstandslos. Die Modifikationen sind:

- Label sind möglich in der Regel *statement*, *assert* wird erkannt
- die Regel *expression2Rest* wurde komplett überarbeitet und berücksichtigt nun die Hierarchie der Operatoren.
- *!>* wurde als alternativer Bezeichner für <= aufgenommen
- *!<* wurde als alternativer Bezeichner für >= aufgenommen

Die neue Grammatik lautet:

parse ← ["package" qualident ";"] { importDeclaration } { typeDeclaration } .

qualident ← ident { "." ident } .

importDeclaration ← "import" ident qualifiedImport ";" .

qualifiedImport ← { "." ident } [".*"] .

typeDeclaration ← ";" | classOrInterfaceDeclaration .

classOrInterfaceDeclaration ← [modifiers] (classDeclaration | interfaceDeclaration) .

type ← (qualident | basicType) bracketsOpt .

basicType ← "byte" | "short" | "char" | "int" | "long" | "float" | "double" | "boolean" .

BracketsOpt ← { "[" "]" } .

typeList ← type { "," type } .

formalParameter ← ["final"] type variableDeclarartorId .

qualidentList ← qualident { "," qualident } .

variableDeclarator ← ident variableDeclaratorRest .

variableDeclaratorId ← ident bracketsOpt .

variableDeclaratorRest ← bracketsOpt ["=" variableIntitializer] .

variableIntitializer ← arrayInitializer | expression .

classDeclaration ← "class" ident ["extend" type] ["implement" typeList] classBody .

classBody ← "{" classBodyDeclaration "}" .

classBodyDeclaration ← ";" | ["static"] (block | [modifiers] memberDecl) .

memberDecl ← constructorDeclaratorRest | "void" voidMethodDeclaratorRest |
classDeclaration | interfaceDeclaration | methodOrFieldDeclaration .

methodOrFieldDeclaration ← ident methodOrFieldRest .

methodOrFieldRest ← methodDeclaratorRest | variableDeclaratorsRest .

variableDeclaratorsRest ← variableDeclaratorRest { "," variableDeclarator } .

arrayInitializer ← "{" [variableIntitializer { "," variableIntitializer }] [","] "}" .

methodDeclaratorRest ← formalParameters bracketsOpt ["throws" qualidentList] (";" | block) .

voidMethodDeclaratorRest ← formalParameters ["throws" qualidentList] (";" | block) .

constructorDeclaratorRest ← formalParameters ["throws" qualidentList] block .

formalParameters ← "(" [formalParameter { "," formalParameter }] ")" .

interfaceDeclaration ← "interface" ident ["extends" typeList] interfaceBody .

interfaceBody ← "{" interfaceBodyDeclaration "}" .

interfaceBodyDeclaration ← ";" | [modifiers] interfaceMemberDecl .

interfaceMemberDecl ← "void" voidInterfaceMethodDeclaratorRest | classDeclaration |
interfaceDeclaration | interfaceMethodOrFieldDeclaration .

interfaceMethodOrFieldDeclaration ← ident interfaceMethodOrFieldRest .

interfaceMethodOrFieldRest ← constantDeclaratorsRest | interfaceMethodDeclaratorRest .

constantDeclaratorsRest ← constantDeclaratorRest { "," constantDeclarator } .

```
constantDeclaratorRest ← bracketsOpt "=" variableIntitializer .

constantDeclarator ← ident constantDeclaratorRest .

interfaceMethodDeclaratorRest ← formalParameters bracketsOpt [ "throws" qualidentList ]
";" .

voidInterfaceMethodDeclaratorRest ← [ "throws" qualidentList ] ";" .

statement ← { ident ":" } ( block
                                    | "assert" expression [  ":" expression ] ";"         -- wird nur
erkannt!
                                | "if" parExpression statement [ "else" statement ]
                                | "for" "(" [ forInit ] ";" [ expression ] ";" [ forUpdate ] ")" statement
                                | "while" parExpression statement
                                | "do" statement "while" parExpression ";"
                                | "try" block ( catches ["finally" block ] | "finally" block )
                                | "switch" parExpression "{" switchBlockStatementGroups "}"
                                | "synchronized" parExpression block
                                | "return" [ expression ] ";"
                                | "throw" expression ";"
                                | "break" [ident] ";"
                                | "continue" [ident] ";"
                                | ";"
                                | statementExpression ";"
                                | "#ass" stringLiteral { "+" stringLiteral } ";"  ) . -- inline Code

block ← "{" blockStatement "}" .

blockStatement ← localVariableDeclaration ";" | classOrInterfaceDeclaration | statement .

localVariableDeclaration ← [ "final" ] type variableDeclarators .

variableDeclarators ← variableDeclarator { "," variableDeclarator } .

forInit ← localVariableDeclaration | statementExpression moreStatementExpression .

forUpdate ← statementExpression moreStatementExpression .

statementExpression ← expression .

moreStatementExpression ← { "," statementExpression } .

catches ← catchClause { catchClause } .

catchClause ← "catch" "(" formalParameter ")" block .
```

2.2.2 Der Parser

```
switchBlockStatementGroups ← { switchBlockStatementGroup } .

switchBlockStatementGroup ← switchLabel { switchLabel } blockStatement .

switchLabel ← ( "default" | "case" type ) ":" .

expression ← expression1 [ assign expression ] .

expression1 ← expression2 [ conditionalExpr ] .

conditionalExpr ← "?" expression ":" expression1 .

expression2 ← expression3 [ expression2Rest ] .

expression2Rest ← logicalOr .

logicalOr ← [ logicalAnd ] { "||" expression3 [ logicalAnd ] } .

logicalAnd ← [ relop ] { "&&" expression3 [ relop ] } .

relop ← "instanceof" type [ ( "==" | "!=" ) expression3 [ instanceOf ] ]  |
          [ or ] [ ( "==" | "!=" ) expression3 [ instanceOf ]  |
                     ( "<=" | "<" | ">=" | ">" | "!<" | "!>" ) expression3 [ or ] ] .

instanceOf ← ( or | "instanceof" type ) .

or ← [ and ] { ( "|" | "^" ) expression3 [ and ] } .

and ← [ shift ] { "&" expression3 [ shift ] } .

shift ← [ lowArith ] { ( ">>" | "<<" | ">>>" ) expression3 [ lowArith ] } .

lowArith ← [ arith ] { ( "+" | "-" ) expression3 [ arith ] } .

arith ← { ( "*" | "/" | "%" ) expression3 } .

expression3 ← { prefix | "(" type ")" } primary { selector } postfix .

primary ← "(" expression ")"
            | "this" argumentsOpt
            | "super" superSuffix
            | literal
            | "new" creator
            | ident { "." ident } [ identifierSuffix ]
            | basicType bracketsOpt "." "class"
            | "void" "." "class" .

argumentsOpt ← [ arguments ] .
```

arguments ← "(" [expression { "," expression }] ")" .

superSuffix ← arguments | "." ident argumentsOpt .

literal ← charConst | number | lnumber | stringLiteral | dnumber | "false" | "true" | "null" .

creator ← basicType arrayCreatorRest | qualident (arrayCreatorRest | classCreatorRest) .

arrayCreatorRest ← "[" ("]" bracketsOpt arrayInitializer | expression "]" { "[" expression "]" } bracketsOpt) .

classCreatorRest ← arguments [classBody] .

identifierSuffix ← "[" "]" bracketsOpt "." "class" | argumentsOpt | "." ("class" | "this") .

selector ← "." (ident argumentsOpt | "super" arguments | "new" innerCreator) | "[" expression "]" .

innerCreator ← ident classCreatorRest .

castExpression ← expression .

parExpression ← "(" castExpression ")" .

assign ← "=" | "+=" | "-=" | "*=" | "/=" | "%=" .

prefix ← "++" | "--" | "!" | "~" | "+" | "-" .

postfix ← "++" | "--" .

modifiers ← "public" | "protected" | "private" |
"abstract" | "final" | "strictfp" | "static" | "transient" | "volatile" |
"synchronized" | "native" .

Der Parser legt einen Header an und füllt ihn, den Quellstrom rekursiv absteigend, analysierend, auf.

```
class Header
 {
  String name;                  // name of java sourcefile
  String [] imports;            // import declarations
  String myPackage;             // package
  Vector scopes;                // starting scopes of all declared classes
  Scope base;                   // root scope of scope tree
  final int depth;              // priority: 0 ... sourcefile
 }
```

Der Header ist eine integrierende Klasse, die Methoden zum Anlegen, Lesen und

2.2.2 Der Parser

Schreiben eines Headers bereitstellt. Die eigentliche Information liegt in den Scopes und den dort eingetragenen Deklarationen der Basistypen ClassType, VariableType und MethodType, alle abgeleitet von Basic.

```
class Scope
 {
  private static Vector table = new Vector();  // holds all scopes for indexing
  static final int automatic = 1,              // storage types
                   heap = 2,                   // static
                   classed = 3,                // class or interface object
                   BLOCK = 1,                  // scope types
                   MAIN = 2,
                   LOOP = 3,
                   TRY = 4,
                   FINALLY = 5,
                   CATCH = 6,
                   SWITCH = 7,
                   SEQUENCE = 8,
                   DUMMY = 9;
  final String prefix;                    // name of class, if starting scope
  int storageClass;                       // automatic or heap for static scope
  final String trailer;                   // trailing path of scope
  int offset = 0;                         // offset for class variables
  private TreeMap map;                    // entries of scope
  private Vector follower;                // scopes of inner classes
  Scope prev;                             // enclosing scope
  Vector label;                           // list of labels (for jumps)
  int block;                              // type of scope
  String exception = null;                // for exception handling
  protected static Scope cur = null;      // for loading of header
}
```

Diese Klasse bietet zahlreiche Methoden zur Verwaltung und Suche von Einträgen im Baum der Scopes. Die Klasse Basic implementiert den CRC-16, der als Basis zur Erzeugung eines Hashcodes dient.

```
public class Basic
 {
   int modify;                  // modifier
   Token name;                  // name
   int version;                 // holds the hashcode
 }
```

Die abgeleiteten Klassen überladen alle *hashCode*, der aus der Signatur des Objekts berechnet wird.

```
class ClassType extends Basic
 {
  ClassType extend;              // superclass
  ClassType [] implement;        // abstract classes and interfaces
  Scope scope;                   // scope of class
  Vector statics;                // list of operator trees of static blocks
  HashSet unresolved;            // list of names of unresolved references
  String comment;                // trailing comment
 }
class VariableType extends Basic
 {
   int offset;             // relative position in object, for class variables
   boolean referenced = false;   // needed in code generation
   Type type;                    // type of variable
 }
class MethodType extends Basic
 {
   Type type;                     // type of result
   Parameter [] parameter;        // list of parameter
   String [] throwing;            // list of exceptions
   Scope scope;                   // scope of method
   Vector operation;              // list of operator trees
   String comment;                // trailing comment
 }
```

Jede Anweisung im Quellcode wird in einen oder mehrere Operatorbäume übersetzt, die Bäume von Operationen in Postfixnotation sind. Die Wurzeln der Bäume werden in einer konsekutiven Liste *operation* bzw. *statics* eingetragen.

```
class Operation
 {
   Operation left;                // left son
   Operation right;               // right son
```

```
    Keyword operator;              // operator
    static int labelno = 0;
    Token name;                    // name, if identifier
    Type type;                     // resulting type
    Scope scope;                   // scope
    String code = "";              // resulting FORTH-Code
    boolean loaded = false;        // resulting value on stack
  }
```

Die Blätter des Baumes (Referenzen) sind durch den Operator *LEAFSY* gekennzeichnet. Ein Blatt enthält die Informationen von Regel *expression3*, diese werden von links nach rechts abgearbeitet.

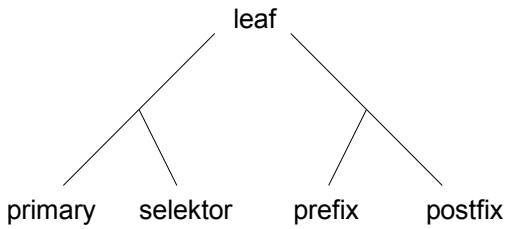


Abbildung 19: Aufbau eines Blattes eines Operatorbaumes

2.2.3 Die Übersetzung

Es wird jede Klasse separat übersetzt. Zuerst wird Code für die statischen Variablen erzeugt, wenn gewünscht ein Modulname deklariert, und Code für die statischen Blöcke generiert. Abschließend werden Methoden und Konstruktoren übersetzt.
Kernstück ist die Auswertung der Listen der Operatorbäume, gesteuert durch die Methode *code*. Aus jedem Baum ergibt sich Code folgender Struktur.

Code ← [locals] statement [exception] [free purge] [exceptionhandling] .
locals ← „LOCALS“ name { name } „|“ .
name ← [0-9]+§
statement ... der Code des Operatorbaums
exception ← "FALSE DUP IF " exceptionlabel " LABEL TRUE ENDIF " .
exceptionlabel ← std[0-9]+
free ... die Referenzen der Objekte in den lokalen Variablen werden erniedrigt
purge ← PURGE [0-9]+
exceptionhandling ... Code für die Einleitung einer Ausnahmebehandlung

Die Anweisung benötigt eventuell verborgene lokale Variablen für Zwischenergebnisse, die der dynamischen Speicherverwaltung, hier Referenzen zählen, Rechnung tragen

müssen, diese müssen zuerst angelegt werden. Vom Compiler erzeugte Hilfsvariablen sind durch ein vorangestelltes oder angehängtes Zeichen „§“ gekennzeichnet, hier heißen sie „0§“, „1§“, usw.. Ob sie benötigt werden, stellt die Auswertung des Operatorbaums durch die Methode *traverse* fest, ebenso die Notwendigkeit einer Ausnahmebehandlung. Sollte diese möglich sein, wird bei regulärem Ablauf FALSE auf den Stapel gelegt, sonst TRUE. Diese Testgröße gibt an, ob abschließend eine Ausnahmebehandlung eingeleitet werden muss.
Methode *traverse* führt teilweise Optimierungen, wie Auswerten konstanter Ausdrücke, durch. Bei Zuweisungen an Variable wird der Speicherverwaltung Rechnung getragen, ebenso werden, eventuell vorhandene, überflüssig gewordene Daten vom Datenstapel eliminiert. Besondere Operatoren, Tabelle 64, die nicht von Java vorgegeben sind.

Operator	***Aktion***
ALLOCATESY	outerclass superclass -- Alloziert den Speicher für eine Instanz einer Klasse. Fix vergeben sind folgende relativen Adresspositionen in einem Klassenobjekt. 0 Referenz auf die Superklasse 4 Referenz auf das Objekt der abgeleiteten Klassen 8 Referenz auf die äußere Klasse 12 Referenz auf die Tabelle der Tupel (Hashcode, Startadresse) der nicht statischen Methoden, einschließlich Konstruktoren 16 Referenz auf eine FORTH-Zeichenkette, die den Klassennamen enthält. Ab 20 stehen die Attribute, beginnend mit Offset 0 Bei den Referenzen Superklasse und äußere Klasse wird der Referenzenzähler erhöht, bei der abgeleiteten Klasse nicht – daher kreisfrei
PUSHSY	Ein neues Scope wird geöffnet und seine lokalen Variablen angelegt
POPSY	Das aktuelle Scope wird geschlossen und zum Umgebenden zurückgekehrt. Wobei die lokalen Variablen vom Stapel der Rücksprungadressen eliminiert werden, nachdem eventuell betroffene Referenzen der Speicherverwaltung Rechnung getragen haben.
LOCKSY	Sperrt ein Objekt für andere Prozesse.
UNLOCKSY	Gibt ein Objekt für andere Prozesse frei.

Tabelle 64: Zusätzliche Operatoren, die nicht von Java vorgegeben sind

2.2.3.1 Deferred Reference Counting

Der Referenzenzähler wird nur erhöht, wenn das Objekt am Stapel aus einer (indizierten)

Variable geladen wurde, und das Ergebnis der rechten Seite einer Zuweisung ist, oder der Rückgabewert einer Methode oder einer *throw*-Anweisung ist, oder ein Aktualparameter eines Methoden- oder Konstruktoraufruf, aber nicht die verborgene Objektreferenz „§this“ – ausgenommen „§this“ ist der Rückgabewert eines *return*, wo sehr wohl zu inkrementieren ist –, ist.

Der Referenzenzähler wird erniedrigt, wenn der verborgene Destruktor „~destructor“ eines Objekts aufgerufen wird. Erreicht er dabei den Wert 0, wird das Objekt gelöscht, nachdem die Destruktoren für die enthaltenen Objekte aufgerufen wurden. Wird eine lokale Variable gelöscht, wird der Destruktor, ausgenommen für „§this“, aufgerufen.

Auf diese Weise kann der Aufwand für die Speicherverwaltung minimal gehalten werden.

2.2.3.2 Klassen

Jede Klasse besitzt einen automatisch erzeugten Konstruktor und Destruktor, die aber vom Programmierer überschrieben werden können. Für alle Klassen ist ein Integer „_dynamicBlocking“ vordeklariert, es soll bei einer späteren Implementierung der Operatoren LOCK und UNLOCK Verwendung finden. Für jede Klasse ist die statische Integer Variable „_staticBlocking“ – für synchronisierte statische Methoden – vordeklariert, ebenfalls für das Sperrprotokoll vorgesehen.

2.2.3.3 Variable

Statische Variablen werden am Heap angelegt, alle anderen sind entweder Bestandteil einer Klasse und daher im dynamischen Speicher, oder lokal deklariert und daher temporär am Stapel der Rücksprungadressen im Prozessor.

2.2.3.4 Methoden

Wenn sie nicht statisch sind, ist ihr erster (verborgener) Parameter „§this“, die Referenz auf das Objekt der Methode. Kann die Methode, gilt auch für Konstruktoren, eine Ausnahme zurückgeben, ist der letzte (verborgene) Parameter eine alternative Rücksprungadresse – das oben spezifizierte *exceptionlabel*. Bei Eintritt in die Methode werden alle Aktualparameter als lokale Variable auf den Stapel der Rücksprungadressen verschoben und formal dem Scope der Methode bzw. Konstruktor zugeordnet. Da der Polymorphie Rechnung zu tragen ist, wird eine Methode indirekt über das BIOS, Prozedur EXECUTE-METHOD, gestartet.

2.2.3.5 Konstruktoren

Member und anonyme Klassen haben als ersten (verborgenen) Parameter „§outer“, die Referenz auf ihr äußeres Objekt. Die erste Aktion im Rumpf ist immer, wenn kein Aufruf „this(...)“ vorgegeben ist, die Erzeugung des Superobjekts, dann das Anlegen des eigentlichen Objekts und zuweisen der Referenz an „§this“. Unmittelbar darauf werden alle nicht statischen Initialisierungen und Blöcke der Klassendeklaration eingefügt, darauf folgen die Anweisungen des Konstruktors. Impliziter Rückgabewert ist „§this“. Tatsächlich gestartet wird der Konstruktor durch Aufruf des BIOS, Prozedur EXECUTE-NEW.

2.2.3.6 Strings

Verwenden als Zeichensatz den Unicode.

2.2.3.7 JavaArray

Eigens zur Handhabung von Feldern steht diese Klasse, die ins Package *java.lang* zu stellen ist, bereit.

```
public class JavaArray
 {
   public final int length;      // length of array
   public final int shift;       // ld(wordlength)
   public final int code;        // (crc16, dimension)
   public int array;             // reference to physical array
 }
```

Die Methoden der Klasse sind in Tabelle 65 dokumentiert.

Methode
JavaArray(int length, int code) constructor length number of elements code lower 16 Bit are the dimension, high part is the type return a new instance of class
public int getElem(int pos) fetch an element of the array pos index return a reference to the element

2.2.3 Die Übersetzung

Methode
public JavaArray clone(JavaArray a) clone an array a array to clone return a clone
public void ~destructor() decrement (and remove) object and array
public static String createString(int bytestring, int length) create an standard string length stringlength bytestring points to initial content return an initialized string object
public static String createUnicode(int forthstring, int length) create an unicode string length stringlength forthstring points to initial content return an initialized string object

Methode
public static void kill(Object obj, int polymorph) decrement (and remove) an object obj the object polymorph the complete object, if true, only this part otherwise
public static void handler(Exception e) default exception handler e Exception
public static void print(String s) print string s s String

Tabelle 65: Schnittstelle der Klasse JavaArray

2.2.4 Schnittstelle

Die Schnittstelle ist die Klasse *Compiler* des Package *MyJava*. Sie stellt eine einzige Methode, Tabelle 66, bereit.

2.2.4 Schnittstelle

Methode
public static boolean compile(String arg, String jdk, String editor, boolean applet, boolean force) compile source named in arg and produce loader list arg sourcename jdk path of jdk sources editor name of editor applet true, if called from an applet force true, force recompilation of all necessary sources return true, if compiled successfully

Tabelle 66: Methode MyJava.compile zum Aufruf des Compilers

2.3 Die Benutzerschnittstelle

So präsentiert sich der Client am Bildschirm.

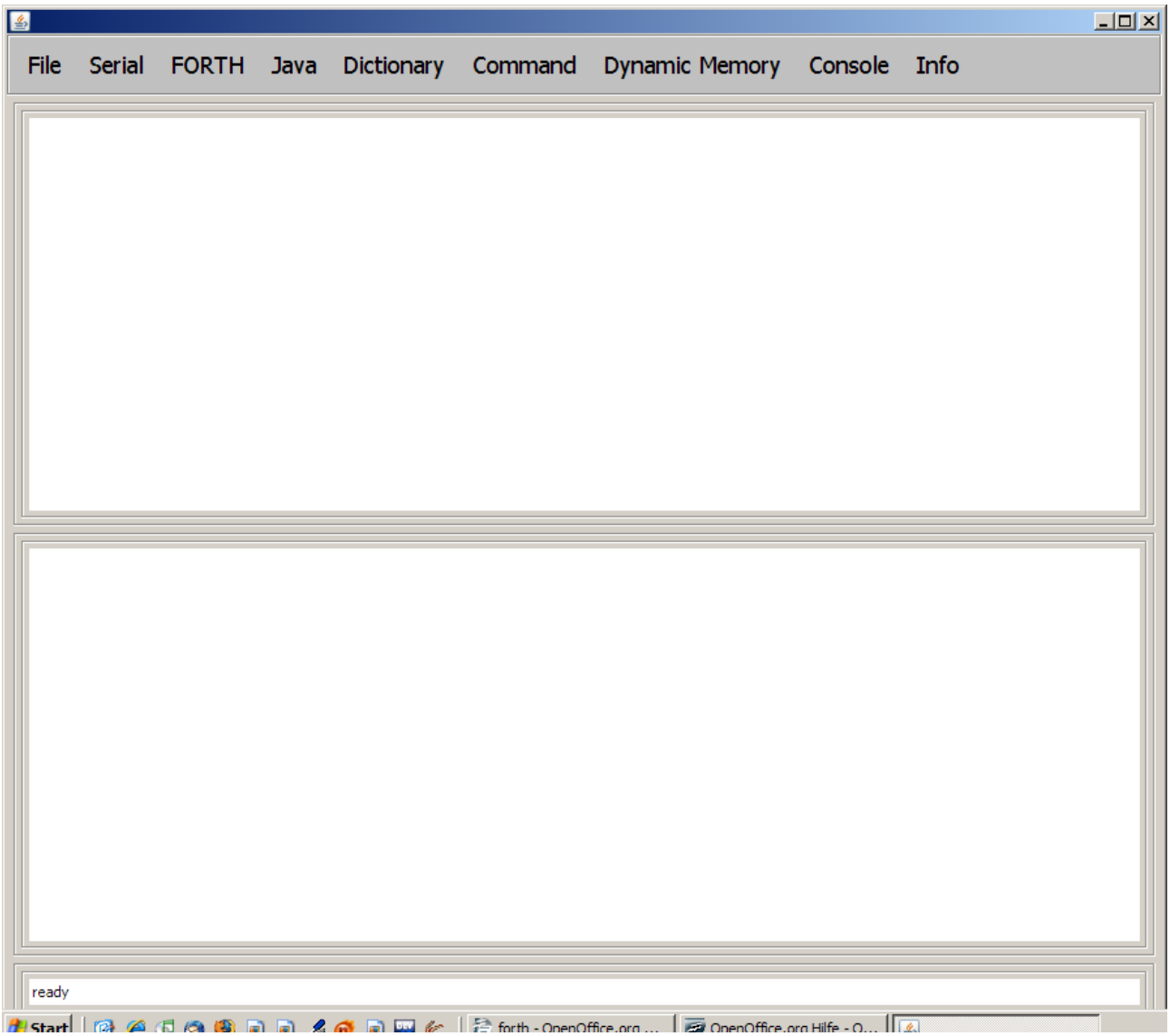


Abbildung 20: Ein Screenshot des Client

Die Anwendung bietet eine Menüleiste, gefolgt von einem Ausgabe-, einem Eingabefenster und einer Statuszeile. Einer kurzen Erklärung bedürfen lediglich die Menüs, Tabellen 67 bis 74.

Menue File	
Menue	***Aktion***
New project	Erlaubt eine neues Projekt im Stammverzeichnis des aktuellen Projekts durch Eingabe seines Namens zu erstellen. Das neue Projekt wird automatisch das aktuelle Projekt.
New file	Es kann eine neue Datei im Projektverzeichnis angelegt und mit dem Editor erstellt werden.
Open project	Öffnet ein bestehendes Projekt, welches zum aktuellen Projekt wird.
Open file	Öffnet eine Datei mit dem Editor.
Close project	Schließt das aktuelle Projekt. Das Stammverzeichnis wird das neue aktuelle Projekt.
Set editor	Der Editor kann festgelegt werden.
Exit	Schließt das Applet und schreibt die Properties in die Datei *myProperties*.

Tabelle 67: Das Dateimenue

Menue Serial	
Menue	***Aktion***
Port	Der verwendete serielle Port kann gewählt werden.
Stoppbits	Wahlweise 1 (default), 1.5 oder 2
Parity	Wahlweise even, odd oder none (default)
Baudrate	Einer der vorgegebenen Werte kann gewählt werden (9600 default).
Control	Flusssteuerung RTS/CTS verwenden, oder keine Steuerung
Force changes	Die vorgenommenen Einstellungen werden übernommen und die Schnittstelle von Server und Client neu initialisiert.
Default values	Die Standardwerte werden gewählt und können mit *Force changes* übernommen werden.

Tabelle 68: Das Menue serielle Schnittstelle

Menue FORTH	
Menue	***Aktion***
Option	Der Typ des Decks kann aus „romable“ (VHDL-Code) und „linkable“ (Objektdatei, für den Server bestimmt) gewählt werden.
Template	Mit diesem Template und dem assemblierten Code wird eine VHDL-Beschreibung eines ROM erzeugt, das direkt in das Projekt eingefügt werden kann.
Blocksize	Die Größe eines Blockram des verwendeten FPGA – bei Spartan3a 2048 Byte.
File	Die zu assemblierende Datei kann gewählt werden. Anschließend wird sie assembliert und die Liste der Quelldateien, sowie eventuelle Fehlermeldungen im Ausgabefenster angezeigt.
File list	Es werden die Dateien einer Ladeliste assembliert. Diese Liste musste zuvor durch Übersetzung einer Java-Datei erzeugt werden. Die Aktion unterstützt hauptsächlich die Fehlersuche in Java-Quellen.

Tabelle 69: Das Menue FORTH-Assembler

Menue Java	
Menue	***Aktion***
Option	Zwischen vollständiger neuerlicher Übersetzung aller Dateien, und ausschließlicher Übersetzung modifizierter Dateien, kann gewählt werden.
gc	Der, im BIOS implementierte, Garbage Collector ist bei der Codeerzeugung zu berücksichtigen. Es ist zwischen „tricolor marking“, „reference counting“ und „modified reference counting“ zu wählen.
To Forth	Die zu übersetzende Datei kann gewählt werden. Anschließend wird sie übersetzt. Die Liste der Quelldateien, sowie eventuelle Fehlermeldungen werden im Ausgabefenster angezeigt.
Sun sources	Das Verzeichnis mit den Quelldateien des JDK kann gewählt werden.
home	Das Verzeichnis der Quelldateien kann gewählt werden.

Tabelle 70: Das Menue Java-Compiler

Menue Dictionary	
Menue	***Aktion***
list	Das Dictionary wird vom Server angefordert und angezeigt.
Load Forth module	Ein ladbares Deck kann gewählt werden. Es wird an den Server übertragen, der es bindet und initialisiert.
Load Java module	Alle noch nicht geladenen Decks einer übersetzten, frei wählbaren Java-Applikation werden an den Server übertragen, der sie bindet und initialisiert.
Initialize Java module	Alle initialisierenden Startdecks einer übersetzten, frei wählbaren Java-Applikation werden an den Server übertragen und ausgeführt.
unload	Ein geladenes Modul kann gewählt werden. Anschließend wird der Server aufgefordert das Dictionary zu verkürzen. Das gewählte Modul und alles was später eingetragen wurde, wird gelöscht.

Tabelle 71: Das Menue Dictionary

Menue Command	
Menue	***Aktion***
Reset	Der Server wird zurückgesetzt und neu initialisiert.
Abort	Die vom Server gestartete Applikation wird abgebrochen und der dynamische Speicher zurückgesetzt.

Tabelle 72: Die direkten Kommandos

Menue Console	
Menue	***Aktion***
Clear output	Das Ausgabefenster wird gelöscht.
Save output as	Der Inhalt des Ausgabefensters wird unter einem wählbaren Namen als Datei gespeichert.
Clear input	Das Eingabefenster wird gelöscht.
Save input as	Der Inhalt des Eingabefensters wird unter einem wählbaren Namen als Datei gespeichert.
Load file	Eine frei wählbare Datei wird ins Eingabefenster geladen.
Execute input	Der Inhalt des Eingabefensters wird assembliert, vom Server geladen und initialisiert.

Tabelle 73: Das Menue Console

Menue Dynamic Memory	
Menue	***Aktion***
available	Die Größe des freien Heap und die Größe des eventuell vorhandenen dynamischen Speichers wird angezeigt.
allocate	Ein dynamischer Speicher wählbarer Größe wird am Heap angelegt und initialisiert.
free	Der dynamische Speicher wird freigegeben.
reset	Der dynamische Speicher wird neu initialisiert.
Default size	Die Standardgröße des dynamischen Speichers kann hier definiert werden.

Tabelle 74: Das Menue zum dynamischen Speicher

Ergebnisse

1 Hardware

Implementiert und getestet wurde auf einem „Spartan-3A FPGA Starter Kit“ von Xilinx. Kernstück diese Boards ist ein Spartan-3A-700k. Weiters enthält es ein 32MBx16 DDR2-RAM, sowie diverse Schnittstellen, unter anderem eine serielle Schnittstelle, über die ein Notebook angeschlossen werden kann. Für den Test der Rechenleistung wurde der Whetstone-benchmark [19] in FORTH implementiert.

Vier Varianten des Prozessors, Tabelle 75, wurden realisiert und getestet.

Variationen	
FORTHSP	Enthält bloß einen Addierer im Rechenwerk und realisiert nur einen Kern
FORTHSPM	Neben dem Addierer enthält das Rechenwerk noch einen 64x64-Bit Multiplizierer und realisiert nur einen Kern
FORTHSPC	Enthält bloß einen Addierer im Rechenwerk, realisiert aber 4 Kerne
FORTHSPMC	Neben dem Addierer enthält das Rechenwerk noch einen 64x64-Bit Multiplizierer und realisiert 4 Kerne

Tabelle 75: Die Varianten der CPU

Für sie wurden folgende Characteristika, Tabelle 76, ermittelt.

Charakteristika				
	FORTHSP	FORTHSPM	FORTHSPC	FORTHSPMC
Fläche (%)	63	72	88	93
Max. Takt (MHz)	73,6	67,5	62,5	57,5
Kritischer Pfad	Stapelverwaltung (Indizierung)	Stapelpuffer – Schieberegister – Stapelpuffer	Stapelverwalt-ung (Indizierung)	Stapelverwalt-ung (Indizierung)
Whetstone (KWIPS)	8,32	25,15	7,32	22,5

Tabelle 76: Die Characteristika der Varianten

1.1 Resourcen und kritische Pfade

Folgende Auflistungen sind Auszüge der Sysnthesis Reports der einzelnen Projekte.

1.1.1 FORTHSP

Selected Device : 3s700afg484-4

Number of Slices:	3418	out of 5888	58%
Number of Slice Flip Flops:	2706	out of 11776	22%
Number of 4 input LUTs:	6385	out of 11776	54%
Number used as logic:	6236		
Number used as Shift registers:	85		
Number used as RAMs:	64		
Number of IOs:	73		
Number of bonded IOBs:	73	out of 372	19%
IOB Flip Flops:	1		
Number of BRAMs:	12	out of 20	60%
Number of GCLKs:	4	out of 24	16%
Number of DCMs:	2	out of 8	25%

Data Path: myCore/SelectedStack_0 to myCore/myStacks/Stack<1>.Cached_8

Cell:in->out	Gate fanout	Net Delay	Delay	Logical Name (Net Name)
FDCE:C->Q	6	0.591	0.701	myCore/SelectedStack_0
LUT3_D:I2->LO	1	0.648	0.132	myCore/myStacks/Selected_0_mux00001_1
LUT3:I2->O	4	0.648	0.667	myCore/myStacks/newCached_mux0000<0>1
LUT2:I1->O	1	0.643	0.000	myCore/myStacks/Msub_newCached_sub0000_lut<0>
MUXCY:S->O	1	0.632	0.000	myCore/myStacks/Msub_newCached_sub0000_cy<0>
MUXCY:CI->O	1	0.065	0.000	myCore/myStacks/Msub_newCached_sub0000_cy<1>
MUXCY:CI->O	1	0.065	0.000	myCore/myStacks/Msub_newCached_sub0000_cy<2>
MUXCY:CI->O	1	0.065	0.000	

1.1.1 FORTHSP

myCore/myStacks/Msub_newCached_sub0000_cy<3>
MUXCY:CI->O 1 0.065 0.000
myCore/myStacks/Msub_newCached_sub0000_cy<4>
MUXCY:CI->O 1 0.065 0.000
myCore/myStacks/Msub_newCached_sub0000_cy<5>
MUXCY:CI->O 1 0.065 0.000
myCore/myStacks/Msub_newCached_sub0000_cy<6>
MUXCY:CI->O 0 0.065 0.000
myCore/myStacks/Msub_newCached_sub0000_cy<7>
XORCY:CI->O 16 0.844
1.066myCore/myStacks/Msub_newCached_sub0000_xor<8>
LUT3_D:I2->O 3 0.648 0.611 myCore/myStacks/newCached_mux0001<0>1
LUT4:I1->O 13 0.643 0.986 myCore/myStacks/incrCached_1_mux00031
LUT4:I3->O 1 0.648 0.500 myCore/myStacks/incrCached_0_mux0003
LUT2:I1->O 1 0.643
0.000myCore/myStacks/Madd_newCached_add0000_lut<0>
MUXCY:S->O 1 0.632 0.000
myCore/myStacks/Madd_newCached_add0000_cy<0>
MUXCY:CI->O 1 0.065 0.000
myCore/myStacks/Madd_newCached_add0000_cy<1>
MUXCY:CI->O 1 0.065 0.000
myCore/myStacks/Madd_newCached_add0000_cy<2>
MUXCY:CI->O 1 0.065 0.000
myCore/myStacks/Madd_newCached_add0000_cy<3>
MUXCY:CI->O 1 0.065 0.000
myCore/myStacks/Madd_newCached_add0000_cy<4>
MUXCY:CI->O 1 0.065 0.000
myCore/myStacks/Madd_newCached_add0000_cy<5>
MUXCY:CI->O 1 0.065 0.000
myCore/myStacks/Madd_newCached_add0000_cy<6>
MUXCY:CI->O 0 0.065 0.000
myCore/myStacks/Madd_newCached_add0000_cy<7>
XORCY:CI->O 2 0.844
0.000myCore/myStacks/Madd_newCached_add0000_xor<8>
FDCE:D 0.252 myCore/myStacks/Stack<1>.Cached_8

Total 13.889ns (9.226ns logic, 4.663ns route)
(66.4% logic, 33.6% route)

1.1.2 FORTHSPM

Selected Device : 3s700afg484-4

Number of Slices: 3975 out of 5888 67%

Number of Slice Flip Flops: 2871 out of 11776 24%

Number of 4 input LUTs: 7318 out of 11776 62%
Number used as logic: 7169
Number used as Shift registers: 85
Number used as RAMs: 64
Number of IOs: 73
Number of bonded IOBs: 73 out of 372 19%
IOB Flip Flops: 1
Number of BRAMs: 11 out of 20 55%
Number of MULT18X18SIOs: 16 out of 20 80%
Number of GCLKs: 5 out of 24 20%
Number of DCMs: 2 out of 8 25%

Data Path: myCore/myStacks/fast_1 to myCore/myStacks/BufferedInput_19

Cell:in->out	fanout	Gate Delay	Net Delay	Logical Name (Net Name)
FDCE:C->Q	13	0.591	1.015	myCore/myStacks/fast_1 (myCore/myStacks/fast_1)
LUT3_D:I2->O	26	0.648	1.292	myCore/myStacks/SideA<0>11_1 (myCore/myStacks/SideA<0>11)
LUT3:I2->O	12	0.648	1.041	myCore/myStacks/SideA<18>1 (myCore/myStacks/SideA<18>)
LUT4:I1->O	1	0.643	0.000	myCore/myStacks/myALU/res_and0000_wg_lut<1> (myCore/myStacks/myALU/res_and0000_wg_lut<1>)
MUXCY:S->O	1	0.632	0.000	myCore/myStacks/myALU/res_and0000_wg_cy<1> (myCore/myStacks/myALU/res_and0000_wg_cy<1>)
MUXCY:CI->O	1	0.065	0.000	myCore/myStacks/myALU/res_and0000_wg_cy<2> (myCore/myStacks/myALU/res_and0000_wg_cy<2>)
MUXCY:CI->O	1	0.065	0.000	myCore/myStacks/myALU/res_and0000_wg_cy<3> (myCore/myStacks/myALU/res_and0000_wg_cy<3>)
MUXCY:CI->O	1	0.065	0.000	myCore/myStacks/myALU/res_and0000_wg_cy<4> (myCore/myStacks/myALU/res_and0000_wg_cy<4>)
MUXCY:CI->O	1	0.065	0.000	myCore/myStacks/myALU/res_and0000_wg_cy<5> (myCore/myStacks/myALU/res_and0000_wg_cy<5>)
MUXCY:CI->O	1	0.065	0.000	myCore/myStacks/myALU/res_and0000_wg_cy<6> (myCore/myStacks/myALU/res_and0000_wg_cy<6>)
MUXCY:CI->O	2	0.269	0.450	myCore/myStacks/myALU/res_and0000_wg_cy<7> (myCore/myStacks/myALU/res_and0000)
LUT4:I3->O	1	0.648	0.423	myCore/myStacks/myALU/Mmux_res_mux0000_5_f5 (myCore/myStacks/myALU/Mmux_res_mux0000_5_f5)
LUT4:I3->O	32	0.648	1.265	myCore/myStacks/myALU/result<1><0>

1.1.2 FORTHSPM

(myCore/myStacks/myALU/result<1><0>)
LUT4:I3->O 1 0.648 0.000 myCore/myStacks/myALU/Mmux_AluResult_910
(myCore/myStacks/myALU/Mmux_AluResult_910)
MUXF5:I0->O 1 0.276 0.000 myCore/myStacks/myALU/Mmux_AluResult_7_f5_9
(myCore/myStacks/myALU/Mmux_AluResult_7_f510)
MUXF6:I0->O 1 0.291 0.563 myCore/myStacks/myALU/Mmux_AluResult_5_f6_9
(myCore/myStacks/myALU/Mmux_AluResult_5_f610)
LUT4:I0->O 1 0.648 0.000 myCore/myStacks/BufferedInput_mux0001<19>113_G
(N3678)
MUXF5:I1->O 1 0.276 0.423 myCore/myStacks/BufferedInput_mux0001<19>113
(myCore/myStacks/BufferedInput_mux0001<19>113)
LUT4:I3->O 1 0.648 0.000 myCore/myStacks/BufferedInput_mux0001<19>148
(myCore/myStacks/BufferedInput_mux0001<19>)
FDCE:D 0.252 myCore/myStacks/BufferedInput_19

Total 14.564ns (8.091ns logic, 6.473ns route)
(55.6% logic, 44.4% route)

1.1.3 FORTHSPC

Selected Device : 3s700afg484-4

Number of Slices:	4798	out of 5888	81%
Number of Slice Flip Flops:	3657	out of 11776	31%
Number of 4 input LUTs:	9094	out of 11776	77%
Number used as logic:	8945		
Number used as Shift registers:	85		
Number used as RAMs:	64		
Number of IOs:	73		
Number of bonded IOBs:	73	out of 372	19%
IOB Flip Flops:	1		
Number of BRAMs:	14	out of 20	70%
Number of GCLKs:	4	out of 24	16%
Number of DCMs:	2	out of 8	25%

Data Path: myCore/Func_2 to myCore/myStacks/newTarget_0

Cell:in->out	Gate fanout	Net Delay	Delay	Logical Name (Net Name)

```
    FDCE:C->Q             3   0.591   0.674  myCore/Func_2 (myCore/Func_2)
    LUT3:I0->O            1   0.648   0.000  myCore/myStacks/newcore_mux0001<0>
    MUXF5:I0->O           5   0.276   0.665  myCore/myStacks/newcore_mux0001<0>
    LUT3_D:I2->O         15   0.648   1.049  myCore/myStacks/newcore_mux0000<1>
    LUT3:I2->O            1   0.648   0.000  myCore/myStacks/newCached_mux0001<0>
    MUXF5:I0->O           3   0.276   0.563  myCore/myStacks/newCached_mux0001<0>
          LUT3:I2->O                    1        0.648        0.000
myCore/myStacks/Msub_newCached_sub0000_lut<0>
          MUXCY:S->O                    1        0.632        0.000
myCore/myStacks/Msub_newCached_sub0000_cy<0>
          MUXCY:CI->O                   1        0.065        0.000
myCore/myStacks/Msub_newCached_sub0000_cy<1>
          MUXCY:CI->O                   1        0.065        0.000
myCore/myStacks/Msub_newCached_sub0000_cy<2>
          MUXCY:CI->O                   1        0.065        0.000
myCore/myStacks/Msub_newCached_sub0000_cy<3>
          MUXCY:CI->O                   1        0.065        0.000
myCore/myStacks/Msub_newCached_sub0000_cy<4>
          MUXCY:CI->O                   1        0.065        0.000
myCore/myStacks/Msub_newCached_sub0000_cy<5>
          MUXCY:CI->O                   0        0.065        0.000
myCore/myStacks/Msub_newCached_sub0000_cy<6>
          XORCY:CI->O                  21        0.844        1.131
myCore/myStacks/Msub_newCached_sub0000_xor<7>
    LUT4_D:I3->O          3   0.648   0.534  myCore/myStacks/newCached_mux0002<7>1
    LUT4_L:I3->LO         1   0.648   0.132  myCore/myStacks/incrCached_1_mux00031_SW3
    LUT4:I2->O           94   0.648   1.285  myCore/myStacks/newSave_mux00021
    LUT4:I3->O           25   0.648   1.260  myCore/myStacks/newTarget_and00001
    FDE:CE                    0.312          myCore/myStacks/newTarget_0
    ----------------------------------------
    Total                     15.798ns (8.505ns logic, 7.293ns route)
                              (53.8% logic, 46.2% route)
```

1.1.3 FORTHSPMC

Selected Device : 3s700afg484-4

Number of Slices: 5261 out of 5888 89%

1.1.3 FORTHSPMC

Number of Slice Flip Flops: 4097 out of 11776 34%
Number of 4 input LUTs: 9415 out of 11776 79%
Number used as logic: 9010
Number used as Shift registers: 85
Number used as RAMs: 320
Number of IOs: 73
Number of bonded IOBs: 73 out of 372 19%
IOB Flip Flops: 1
Number of BRAMs: 14 out of 20 70%
Number of MULT18X18SIOs: 16 out of 20 80%
Number of GCLKs: 5 out of 24 20%
Number of DCMs: 2 out of 8 25%

Data Path: myCore/SelectedStack_0 to myCore/myStacks/Core<1>.Stack<0>.Top_6

Gate Net
Cell:in->out fanout Delay Delay Logical Name (Net Name)
---------- ------------
FDCE:C->Q 10 0.591 1.025 myCore/SelectedStack_0 (myCore/SelectedStack_0)
LUT4:I0->O 1 0.648 0.000 myCore/myStacks/ReloadState_and000121_F (N2251)
MUXF5:I0->O 6 0.276 0.701 myCore/myStacks/ReloadState_and000121
(myCore/myStacks/N169)
LUT4_D:I2->O 42 0.648 1.297 myCore/myStacks/ReloadState_and0007
(myCore/myStacks/ReloadState_and0007)
LUT4:I2->O 1 0.648 0.423 myCore/myStacks/newCached_mux0001<1>21
(myCore/myStacks/newCached_mux0001<1>21)
LUT4:I3->O 8 0.648 0.760 myCore/myStacks/newCached_mux0001<1>36
(myCore/myStacks/newCached_mux0001<1>)
LUT4:I3->O 1 0.648 0.452 myCore/myStacks/newCached_cmp_gt0000214
(myCore/myStacks/newCached_cmp_gt0000214)
LUT4:I2->O 19 0.648 1.088 myCore/myStacks/newCached_cmp_gt0000247
(myCore/myStacks/newCached_cmp_gt0000)
LUT4:I3->O 3 0.648 0.534 myCore/myStacks/newToPop_mux0001<1>1
(myCore/myStacks/N1111)
LUT4:I3->O 1 0.648 0.000 myCore/myStacks/newTop_and0000_f5_F (N2253)
MUXF5:I0->O 5 0.276 0.713 myCore/myStacks/newTop_and0000_f5
(myCore/myStacks/newTop_and0000)
LUT2:I1->O 1 0.643 0.000 myCore/myStacks/Msub_newTop_addsub0000_lut<2>
(myCore/myStacks/Msub_newTop_addsub0000_lut<2>)
MUXCY:S->O 1 0.632 0.000 myCore/myStacks/Msub_newTop_addsub0000_cy<2>
(myCore/myStacks/Msub_newTop_addsub0000_cy<2>)
MUXCY:CI->O 1 0.065 0.000 myCore/myStacks/Msub_newTop_addsub0000_cy<3>
(myCore/myStacks/Msub_newTop_addsub0000_cy<3>)
MUXCY:CI->O 1 0.065 0.000 myCore/myStacks/Msub_newTop_addsub0000_cy<4>
(myCore/myStacks/Msub_newTop_addsub0000_cy<4>)
MUXCY:CI->O 0 0.065 0.000 myCore/myStacks/Msub_newTop_addsub0000_cy<5>
(myCore/myStacks/Msub_newTop_addsub0000_cy<5>)
XORCY:CI->O 1 0.844 0.423 myCore/myStacks/Msub_newTop_addsub0000_xor<6>

(myCore/myStacks/newTop_addsub0000<6>)
LUT4:I3->O 8 0.648 0.000 myCore/myStacks/newTop_mux0004<6>51
(myCore/myStacks/newTop_mux0004<6>)
FDPE:D 0.252 myCore/myStacks/Core<1>.Stack<0>.Top_6

Total 16.957ns (9.541ns logic, 7.416ns route)
(56.3% logic, 43.7% route)

Der Flächenbedarf ist, bedingt durch die aufwändige Stapelverwaltung, hoch und variiert zwischen 441k-Gatter und 623k-Gatter. Die max. Frequenzen sind zufriedenstellend, zumal alle Bedingungen eingehalten werden – ziemlich gleich wie Microblaze. Steht eine harte Versorgungsspannung im Einsatzbereich zur Verfügung, kann auch übertaktet werden. Im Test – auch über mehrere Stunden – war das kein Problem und führte zu keinem Absturz..

Die kritischen Pfade gehen, wie erhofft, zumindest bei der Variante mit einem Kern und einem Multiplizierer, nicht durch die Stapelverwaltung, sondern, wie bei Register-maschinen, durch die ALU. Bemerkenswert ist, dass im kritischen Pfad nicht der Multiplizierer, sondern das Schieberegister liegt. Eine Verbesserung des kritischen Pfades, mit dem Ziel einer Geschwindigkeitssteigerung kann nur durch Optimierung der Stapelverwaltung (Indizierung) erreicht werden, zusätzliche Operationsschritte bringen keine weitere Verbesserung.

Die Zahl der Gleitkommaoperationen pro Sekunde ist, bedingt durch ihre Realisierung als Software, nicht berauschend – verglichen mit einem PC, wo mehrere MIPS erreicht werden –, aber für den Einsatz des Prozessors zum Messen, Auswerten, Steuern und Regeln völlig ausreichend. Der Einsatz eines Multiplizierers, sowie realisieren der Gleitkommadivision als Multiplikation mit dem Kehrwert, erhöht die Zahl der Whetstones deutlich.

2 Software

Zur Benutzerschnittstelle im Client werden keine Ergebnisse vorgestellt, sie wurde lediglich auf ihre korrekte Funktion getestet. Zwecks Demonstration des Systems wurden folgende Applikationen, Tabelle 77, erstellt.

Applikation	
Whetstone.fs	Whetstone benchmark in FORTH
Bubble.fs	Demo eines Bubblesort in FORTH
Hanoi.txt	Demo der Türme von Hanoi in FORTH
Switch.txt	Demo, Wechsel von Kern 0 zu Kern 1 und wieder zurück
Sort.java	Demo des Bubblesort bzw. Quicksort
Equation.java	Demo zum Lösen linearer Gleichungssysteme

Tabelle 77: Applikationen zum Test und zur Demonstration

2.1 BIOS

Alle Funktionen und Dienste wurden getestet und arbeiten einwandfrei. Folgende Tests, Tabelle 78, wurden durchgeführt.

Programme	
Testall.txt	Testet alle arithmetischen Funktionen und die Dateioperationen des BIOS
Memorytest.fs	Testet den dynamischen Speicher und seine Grundoperationen

Tabelle 78: Testprogramme für das BIOS

Die arithmetischen Funktionen im BIOS liefern die gleichen Werte wie die entsprechenden Funktionen der Klasse *Math* von Java auf einem PC. Verglichen wurden die ersten 16 tatsächlich signifikanten Stellen der Resultate, da ceil(53 * $\log_{10}(2)$) == 16 gilt. Die Ergebnisse unterscheiden sich höchstens im letzten Bit der Mantissen. Genaueres kann nicht gesagt werden, da der Fehler auch durch die unterschiedliche Konvertierung der Ausgabefunktionen vor der Bildschirmausgabe verursacht sein kann. Der Fehler beträgt höchstens 2 an der letzten Stelle der konvertierten Zeichenkette.

2.2 Assembler

Auch der Assembler arbeitet mit allen Optionen fehlerfrei. Zusätzliche Befehle können mit wenig Aufwand aufgenommen werden. Dazu ist im Scanner *Scanner.java* im Verzeichnis *Forthassembler* in den Deklarationen, direkt nach *getcoreid*, der Bezeichner des neuen Befehls einzufügen und mit seinem Operationscode zu initialisieren. Wenn es sich um einen Befehl mit unmittelbarem Operanden handelt, muss ein eigener Fall in der Fallunterscheidung in der Regel *statements* von *Parser.java* eingefügt werden, andernfalls

sind dort keine Modifikationen nötig. In *appendCommand* von *Code.java* muß je ein Fall in den zwei Fallunterscheidungen eingetragen werden. In der Ersten wird der Befehl einer Befehlgruppe zugeordnet – dient nur der Statistik –, in der Zweiten wird er gemäß der Differenz „Zahl der Ergebnisse (entweder 0 oder 1)“ – „Zahl der verbrauchten Stapelwerte (entweder 0, 1 oder 2)“ eingeordnet. Damit ist die Erweiterung abgeschlossen.

2.3 Java

Die Funktion des Compilers wurde bloß mit den Applikationen *Sort* und *Equation* überprüft, aber nicht umfassend getestet. Es scheinen aber alle Kontrollstrukturen (for, while, do-while, continue, break, switch, sowie try-catch), das Anlegen und Löschen von Scopes, sowie Reference counting problemlos zu funktionieren. Der Grund für diese kurze Prüfung liegt an der langen Ladezeit einer Applikation. Sie besteht nicht nur aus den Quelldateien allein, es müssen vielmehr alle Dateien der referenzierten Klassen geladen werden, was sehr viel Zeit benötigt. Die Ausführung der Applikation hingegen ist erfreulich schnell. Die lange Ladezeit ist hauptsächlich auf die serielle Schnittstelle zurück zuführen, eine USB-Schnittstelle würde eine deutliche Verbesserung bringen und den Test des Compilers erleichtern.

Eine Java-Applikation wird mit dem Menue *Load Java module* geladen und mit *initialize Java module* aufgerufen und nicht durch eine Befehlszeile im Eingabefenster! Dazu muss in der Hauptklasse ein abschließender statischer Block eingefügt werden, der den Aufruf durchführt sowie die Parameter bereitstellt. Bei der Initialisierung wird dann die Applikation ausgeführt. Für einen Aufruf mit anderen Parametern ist dieser Block zu modifizieren und die Applikation neu zu übersetzen. Es muss aber nicht mehr geladen werden, es kann sofort mit dem Menue *initialize Java module* neu gestartet werden.

Die Basisklassen (Object, String, etc.) einer Applikation können aus dem Quellpaket *j2sdk-1_4_2-src-scsl.zip* von Sun übernommen werden, sind aber meist anzupassen, insbesondere müssen alle *native*-Methoden ausprogrammiert werden, Reflektion (Class.java) wird nicht unterstützt und muss aus von Sun übernommenen Quellen entfernt werden. Einige Klassen – sie sind *Sort* bzw. *Equation* zu entnehmen – sind bereits teilweise adaptiert, Klasse *Math* vollständig.

3 Abschließender Ausblick

Zur Zeit ist das System als Prototyp zu betrachten, das als Grundlage für Mikrocontroller verschiedenster Anwendungen zu sehen ist. Es können über den Hirose-Stecker unterschiedlichste Geräte und Hardware angeschlossen werden, USB-, ADC- und DAC-Bausteine sind auf dem Xilinx Starter Kit bereits vorhanden. Sie müssen nur noch an den Prozessor angeschlossen werden. Für welche Aufgaben das System zum Einsatz kommt, wenn überhaupt, ist völlig offen.

3.1 Verbesserungsvorschläge

3.1.1 Der Prozessor

Die *program prefetch queue* könnte überarbeitet werden, da sie bei den Versionen mit 4 Kernen der Flaschenhals ist.

3.1.2 Die Speicherverwaltung im BIOS

Zum Ersten kann die Methode der Speicheraufteilung gewechselt werden, unter beibehalten der Verwaltung mit *Reference Counting*. Zum Zweiten kann *Reference Counting* zu Gunsten einer anderen Methode, etwa *Mark and Sweep*, geändert werden. Dazu könnten aber Modifikationen am Prozessor nötig sein, etwa ein zusätzliches Bit im Stapel, das eine Referenz in einer lokalen Variable markiert. Im Schritt *Mark*, beim Durchwandern der lokalen Variable, werden daran lebende Speicherblöcke erkannt und als lebend markiert. Eine weitere Variante wäre ein dritter Stapel, nur für lokale Variable, die Referenzen enthalten.

Das soll nicht heißen, dass eine spezielle Hardware notwendig ist, aber überlegen sollte man das schon.

3.1.3 Der Client

Eine USB-Schnittstelle wäre wünschenswert, da die serielle Schnittstelle doch recht langsam ist. Sun stellt leider keine Bibliothek dafür bereit, diese müsste zugekauft werden. Beim Server müsste natürlich auch der USB-Baustein angeschlossen werden.

3.1.4 Java

Wird *Reference Counting* zu Gunsten einer anderen Methode verworfen, so sind folgende Änderungen beim Compiler vorzunehmen:

In *Pass.java*, Methode *ClassBody* ist die Erzeugung von *~destructor* zu unterbinden, d.h. Der if-Block nach der Kommentarzeile „// default destruktor“ ist zu löschen. In *Code.java* sind die Methoden *freeVariable*, *incrementReference* und ihre Aufrufe zu löschen.

Bei Bedarf sollten – was sicher der Fall sein wird – Quellklassen von *Sun* für das System adaptiert werden.

Erweiterungen

1 Garbage collector tricolor marking

Alternativ zu Reference Counting kann diese Methode – Quelldatei „onthefly.fs“ - eingesetzt werden.

Die Methode ist eine Modifikation der Methode *Mark and Sweep*. Im Zustand *Mark* wird ein besuchter Block nicht sofort als essentiell (Farbe schwarz) markiert, sondern als essentiell, aber noch nicht ausgewertet (Farbe grau) markiert. Erst wenn alle seine verborgenen Referenzen besucht wurden und mit Farbe grau markiert sind, wird seine Farbe schwarz. Ferner gilt: Ein allozierter Block hat die Farbe weiß, ein freier Block hat keine Farbe. Am Ende von *Mark*, sind alle essentiellen belegten Blöcke nicht mehr weiß. Im Schritt *Sweep* werden die weißen Blöcke freigegen und farblos, und die nicht-weißen Blöcke weiß. Die Speicherbereinigung ist damit abgeschlossen. Am Beginn von Schritt *Mark* werden alle Einträge im Stapel der Referenzvariablen (*VBLOCK*) und alle Einträge im zyklischen Speicher der statischen Referenzvariablen (*SBLOCK*) grau gefärbt, sofern die Einträge nicht null sind. Dann folgt eine Schleife in der alle grauen Blöcke abgearbeitet werden. Sie wird erst verlassen, wenn kein grauer Block mehr vorhanden ist, dann sind alle essentiellen Blöcke abgearbeitet und schwarz markiert. *Tricolor marking* ist im Gegensatz zu *mark and sweep* unterbrechbar. Vom garbage collector wird eine Zustandsvariable *VBARRIER* auf *true* gesetzt, wenn er die Bereinigung beginnt und am Ende von Schritt *Sweep* auf *false* gesetzt. Die Verwaltungsroutinen, Tabelle 1, des Speichers der Referenzvariablen benötigen diese Information.

Funktion	Bedeutung
VALLOCATE	n – adr Ist n == 0, wird eine Addresse *adr* aus dem zyklischen Speicher zurückgegeben. Ist n > 0, werden n leere Einträge am Stapel der Referenzvariablen abgelegt und als *adr* die Adresse des ersten allozierten Eintrags zurückgegeben. Ist n größer als die freie Kapazität des Stapel, wird 0 retourniert
SETVTOP	adr -- Verweist adr in den Stapel der Referenzvariablen, wird *adr* die neue Spitze des Stapels
V!	handle adr -- Ist *adr* == 0, wird handle in der Refernzvariable mit dem Index 0 im Stapel der Referenzvariablen geschrieben. Sonst in die durch *adr* identifizierte Speicherposition. Ist VBARRIER gesetzt und *handle* <> 0, wird die Farbe des handle auf grau gesetzt (die starke Invariante von tricolor marking).
MARK&SWEEP	-- Jede Millisekunde wird geprüft, ob *MACCU* größer als ein Viertel der Größe des dynamischen Speichers ist. Wenn dem so ist, wird der garbage collector MARK&SWEEP aufgerufen.

Tabelle 79: Die Routinen der Speicherbereinigung

Von der Speicherverwaltung wird ein Akkumulator *MACCU* bereitgestellt, der beim Allozieren eines Blocks (*MALLOC*) die angeforderten Blockgrößen akkumuliert. Vom garbage collector wird diese Variable wieder zurückgesetzt.

1.1 Zeiten für Routinen

Es gelten folgende Abkürzungen:

- r die Größe des zyklischen Speichers und des Stapels der Referenzvaria
  blen in Worten
- h die Zahl der angelegten Handle
- n_i die Zahl der verborgenen Referenzen des Blocks i
- x die Zahl der Iterationen von Schritt Mark, hier unbekannt

Außerdem gelte die Annahme, dass ein Befehl in einem Takt abgearbeitet ist.

Die Konstanten repräsentieren die ausgezählte Zahl der Befehle einer Sequenz, die Variablen repräsentieren die Zahl der Schleifendurchläufe einer Sequenz.

1.1 Zeiten für Routinen

Die Zahl der Befehle für die Funktion VALLOC:

VALLOC(m) = 36 + m * 9

Die Zahl der Befehle für die Funktion SETVTOP:

SETVTOP = 20

Die Zahl der Befehle für die Prozedur V!:

V! = 32

Die Zahl der Befehle für den garbage collector MARK&SWEEP:

MARK&SWEEP = 36 + 25 * r + x * (12 + h * (138 + n_i * 20))

1.2 Dynamischer Speicher (Buddy-Methode)

Alternativ zur Speicherverwaltung mit der MS-DOS-Methode – Quelldatei „memorymanagement.fs“ - kann die Buddy-Methode – Quelldatei „buddy.fs“ - verwendet werden, unabhängig vom Garbage Collector.

Die Buddy-Methode setzt einen Speicher voraus, dessen Länge eine Potenz von 2 ist. Bei einer Anforderung wird ein Block mit der Länge pow(2, (int)ld(2 * n - 1)) zurückgegeben, also ein Block mit einer Länge der kleinsten Potenz von 2, die n einschließt. Jeder Block mit Potenz m hat einen gleich großen Buddy. Die relativen Adressen der Buddies stehen in folgender Beziehung zueinander Adresse(budy0) = Adresse(buddy1) ^ pow(2, m). Jeder Buddy enthält wiederum 2 Buddies mit einer Potenz von m-1, usw.

Standardfunktionen sind: UNUSED, modifizierte Standards sind ALLOCATE, FREE, RESIZE [2, Kapitel 14.6.1, Seite 87]. Die Modifikation besteht darin, dass die Speicherverwaltung dem Anwender einen Speicherblock nicht direkt zur Verfügung stellt, sondern über einen Handle. Der Aufbau eines Handle:

Wort	Bedeutung
0	Der Referenzenzähler
1	Startadresse des Datenbereiches des allozierten Speicherblocks

Diese Lösung ermöglicht erst eine Speicherbereinigung! Sollte eine Speicheranforderung nicht unmittelbar befriedigt werden können, wird die Bereinigung automatisch aufgerufen. Sind wenigstens 2 Böcke, die nur halb so groß sind wie der benötigte, frei, versucht die Bereinigung durch umkopieren des Buddies eines der freien Blöcke, einen doppelt so großen freien Block zu erzeugen. Gelingt dies, kann die Anforderung erfüllt werden.

Java erwartet eine Speicherverwaltung, die selbständig erkennt, wann ein belegter Block tatsächlich freigegeben werden kann. Eine sehr alte Methode, die dies unterstützt, ist das Zählen der Referenzen [8, Kapitel 2.1, Seite 19 - 25], eine Aufgabe, die von der Anwendung erledigt werden muss – die Speicherverwaltung muss das aber unterstützen. Kurz, ein Block wird nur dann in die Liste der freien Blöcke übernommen, wenn sein Referenzenzähler den Wert 0 erreicht hat, ansonsten wird der Zähler bloß erniedrigt. Vorteil dieser Methode ist die Aufteilung der Verwaltungsarbeit, sie wird nur dann

ausgeführt, wenn sie tatsächlich notwendig wird. Andere Verfahren benötigen einen eigenen periodischen Prozess der Speicherbereinigung, der nicht unterbrochen werden darf, was nicht wünschenswert erscheint.

Der dynamische Speicher ist ein geschlossener Speicherblock, der über die Dienste angelegt, freigegeben und initialisiert werden kann. Es ist nur ein dynamischer Speicher möglich. Im ersten Wort des Blocks steht die Zahl der angelegten Handle, im zweiten Wort ist der Kopf der Liste der freien Blöcke gespeichert. Ab dem dritten Wort steht die konsekutive Liste der Handle, die beliebig wachsen kann. Der Rest des Blocks zerfällt in belegte, nicht zusammenhängende Bereiche, die über die Handle zugänglich sind. Die unbelegten Bereiche sind in einer doppelt verketteten Liste organisiert, für jede Potenz von 2 eine Eigene. Ein freier Block hat folgenden Aufbau:

Wort	Verwendung
0	Adresse des nächsten freien Blockes
1	Adresse des vorhergehenden freien Blockes
2	Potenz der Länge des Blockes
Ab 3	ungenutzt

Ein belegter Block:

Wort	Verwendung
0	Potenz der Länge des Blockes
1	Adresse des Handle
Ab 2	Datenbereich

1.2.1 Allozieren

Zuerst wird ein freier Handle gesucht bzw. angelegt und ein bestmöglich passender freier Block gesucht. Ist keiner vorhanden, wird nach einer Speicherbereinigung erneut gesucht, bei Misserfolg wird abgebrochen. Ansonsten wird der Block ausgekettet, angepasst, initialisiert, die Liste der freien Blöcke modifiziert und die Adresse des Handle zurückgegeben.

1.2.2 Freigeben

Der Zähler wird erniedrigt, und nur wenn er 0 wird, in die Liste der freien Blöcke, nach eventueller Verschmelzung mit den Nachbarn, eingetragen. Der Handle wird nicht freigegeben, kann aber wiederverwendet werden.

1.2.3 Größe ändern

Der Datenbereich des Blocks wird auf den Heap kopiert, und der Block frei gegeben. Ein neuer Block der gewünschten Größe wird alloziert, der Datenbereich des bisherigen Blocks vom Heap kopiert, und der alte Handle, der nun den neuen Datenbereich enthält, zurückgegeben.

Zusätzliche Befehle:

Befehl	Bedeutung
ALLOCATED-SIZE	handle – size gibt die Größe des Datenbereichs, ohne Verwaltungsdaten, zurück
HANDLEVALID	handle – flag wenn es sich um eine gültige Adresse handelt, wird -1, sonst 0, zurückgegeben
INCREFERENCE	handle -- erhöht den Referenzenzähler

1.2.4 Zeiten für INCREFERENCE, ALLOCATE und DECREFERENCE

Es gelten folgende Abkürzungen:

- N die Größe des dynamischen Speichers in Worten
- h die Zahl der angelegten Handle
- n, i $3 < n, i < ld(N) + 1$

Außerdem gelte die Annahme, dass ein Befehl in einem Takt abgearbeitet ist.

Die Konstanten repräsentieren die ausgezählte Zahl der Befehle einer Sequenz, die Variablen repräsentieren die Zahl der Schleifendurchläufe einer Sequenz.

Die Zahl der Befehle für die verborgene Prozedur zum Ausketten eines freien Blocks:

UNCHAINFREEBLOCK = 24

Die Zahl der Befehle für die verborgene Prozedur zum Einketten eines freien Blocks:

CHAINFREEBLOCK = 37

Die Zahl der Befehle für die verborgene Prozedur zum Belegen eines freien Blocks:

OCCUPYBLOCK(n) = 41 + n * (24 + CHAINFREEBLOCK)

Die Zahl der Befehle für die verborgene Prozedur zum Verschieben eines Buddy:

MOVEBUDDY(n) = 13 + n * (51 + UNCHAINFREEBLOCK + 63 + SEARCHLAST(i) + OCCUPYBLOCK(i)) / 2

Die Zahl der Befehle für die verborgene Prozedur zur Speicherverdichtung:

COMPACTPOOL = 104 + UNCHAINFREEBLOCK + MOVEBUDDY + CHAINFREEBLOCK

Die Zahl der Befehle für die verborgene Funktion zur Handlesuche:

$GETFREEHANDLE_{usually} = 12 + 13 * h$

$GETFREEHANDLE_{worst} = 12 + 13 * h +$ MOVEBUDDY(ld(h)) + 19

Die Zahl der Befehle für die verborgene Funktion zur Ermittlung des letzten besten freien Blocks:

SEARCHLAST(n) = 24 + (ld(N) – n + 1) * 12

Die Zahl der Befehle für die verborgene Funktion zur Ermittlung des besten freien Blocks:

$BESTFREEBLOCK_{usually} = 26 +$ SEARCHLAST

$BESTFREEBLOCK_{worst} = 26 +$ SEARCHLAST + COMPACTPOOL + SEARCHLAST

Die Zahl der Befehle für die Funktion MALLOC:

MALLOC(n) = 59 + GETFREEHANDLE + BESTFREEBLOCK + (n + 3) / 4 * 12

Die Zahl der Befehle für die Funktion DECREFERENCE:

$DECREFERENCE_{decrement} = 48$

$DECREFERENCE_{destroy} = 48 + 53 + 48 * n$, mit n = ld(Größe des Blocks)

Die Zahl der Befehle für die Prozedur INCREFERENCE:

INCREFERENCE = 40

1.2.5 Abschließende Bemerkung

Die implementierte Speicheraufteilung ist sicher nicht optimal hinsichtlich der Speichernutzung, aber schnell. Die Buddy-Methode kann aber problemlos durch eine andere Speicheraufteilung ersetzt werden. Die Implementierung startet mit MOVEBLOCK-UP einschließlich und endet bei UNUSED ausschließlich und nimmt 1186 Byte des BIOS ein. Für eine andere Implementierung sind knapp 2 kB im ROM verfügbar, länger darf sie nicht sein.

Anhang

1 Verwendete Resourcen

Hardware:

- Xilinx, Spartan 3A Kit: zur Realisierung des Prozessors
- Digitus, USB 2.0 to RS232 Adapter: serielle Schnittstelle für ein Notebook
- Ein nichtausgekreutztes Kabel mit 9-poligem Stecker und Buchse: verbindet Notebook mit der Hardware
- Notebook mit Windows XP

Software:

- Xilinx ISE Webpack 9.2: Entwicklungsumgebung für den Prozessor, freie Software
- ModelSim Simulator 6.1e: zum Testen des VHDL-Codes
- Jbuilder 2005: zur Entwicklung des Client
- javacomm20-win32.zip: Java-Interface für die serielle Schnittstelle, freie Software von Sun
- Open Office: für die Dokumentation, freie Software
- j2sdk-1_4_2-src-scsl.zip: Quelldateien der Java-Basisklassen von Sun, freie Software

2 Beschreibung der Adaptierung des RAM-Controllers von Xilinx

Die Adaptierung in *vhdl_syn_bl4_parameters_0.*vhd:

Eifügen von

```
use work.global.all;
```

und Modfikation von

```
constant  reset_active_low: std_logic := '1';
constant rfc_count_value:std_logic_vector(5 downto 0) :=
std_logic_vector(to_unsigned(integer(60.0e-9 * real(theClock) + 0.49),6));
constant   max_ref_cnt: std_logic_vector(10 downto 0) :=
std_logic_vector(to_unsigned(integer(real(theClock)*7.6e-6),max_ref_width));
```

Die Adaptierung von *vhdl_syn_bl4_infrastructure.vhd* durch Modifikation (fett hervorgehoben) unmittelbar nach dem Schlüsselwort *begin* bis einschließlich dem ersten *process*-block:

```
  sys_clk_ibuf <= SYS_CLK;
  --lvds_clk_input : IBUFG port map(
  --  I  => SYS_CLK,
  --  O  => sys_clk_ibuf
  --  );
  clk_int_val           <= clk_int;
  clk90_int_val         <= clk90_int;
  sys_rst_val           <= sys_rst;
  sys_rst90_val         <= sys_rst90;
  sys_rst180_val        <= sys_rst180;
  delay_sel_val1_val    <= delay_sel_val1;



-- To remove delta delays in the clock signals observed during simulation
-- ,Following signals are used


  clk_int_val1   <= clk_int;
  clk90_int_val1 <= clk90_int;
  clk_int_val2   <= clk_int_val1;
  clk90_int_val2 <= clk90_int_val1;
  user_rst       <= not reset_in_n when reset_active_low = '1'  else reset_in_n;
  user_cal_rst   <= reset_in_n     when reset_active_low = '1'  else not
reset_in_n;



  process(clk_int_val2)
```

```
begin
 if clk_int_val2'event and clk_int_val2 = '1' then
  if user_rst = '1' or dcm_lock = '0' then
     wait_200us_i     <= '1';
     Counter200       <= (others => '0');
  else
      if( Counter200 < integer(real(theClock) * 200.0e-6 + 0.5)) then
         wait_200us_i <= '1';
         Counter200 <= Counter200 + 1;
      else
         Counter200 <= Counter200;
         wait_200us_i <= '0';
     end if;
  end if;
 end if;
end process;
```

Die Adaptierung von *vhdl_syn_bl4_controller_0.vhd* durch Einfügen von

```
use work.global.all;
```

Die Adaptierung von *vhdl_syn_bl4.vhd* durch Entfernen der Komponente *vhdl_syn_bl4_main_0*, dem Testmodul.

Referenzen

[1] Koopman, Ph. (1989) Stack Computers – the new wave, Ellis Horwood

[2] ANSI, 1994 DPANS'94 – Programming Languages – FORTH, ANSI, 24. März 1994

[3] Zech, R. (1985) Die Programmiersprache FORTH, Franzis Verlag

[4] FIG, http://www.forth.org/, FORTH interest group (präsent August 2008)

[5] Knuth, Donald E. (2005) The Art of Computer Programming Vol. 1 MMIX, Addison-Wesley

[6] RIIC, VLSI-Entwurf Vorlesungsskript, Institut für integrierte Schaltungen, Johannes Kepler

Univ.

[7] Lindholm, T., Yellin, F. The Java Virtual Machine Specification Second Edition, Addison-

Wesley

[8] Jones, R., Lins, R. (1996) Garbage Collection, John Wiley & Sons

[9] Dunn, D. (2002) Java Rules, Addison-Wesley

[10] Gosling, J., Joy, B., Steele, D., Bracha, G. (2005) The Java Language Specification Third

Edition, Addison-Wesley

[11] SSW, http://www.ssw.uni-linz.ac.at/Coco/Java/JavaGrammar.html, Institut für System

software, Johannes Kepler Univ. (präsent August 2008)

[12] Kahan, W. (1997) http://www.cs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF,

Berkley (präsent August 2008)

[13] Abramowitz, M., Stegun, I. (1965) Handbook of Mathematical Functions, Dover

Publications

[14] Aho, Hopcroft, Ullman, (1974) The Design and Analysis of Computer Algorithms, Addison-

Wesley

[15] Trenz, http://www.trenz-electronic.de/home/indexde.htm, FPGA-Module, Entwicklungs

boards (präsent August 2008)

[16] Xilinx, http://www.xilinx.com, ISE Webpack, Entwicklungsboards (present August 2008)

[17] Forth Inc., http://www.forth.com/resources/evolution/index.html, Geschichte von FORTH

(präsent August 2008)

Referenzen

[18] O'Connor, Derek, 2006, http://www.derekoconnor.net/NA/Notes/RecSqRoot.pdf, RECIPROCAL & SQUARE ROOT (präsent August 2008)

[19] Wikipedia, http://de.wikipedia.org/wiki/Whetstone, Whetstone benchmark (präsent August 2008)

[20] JavaDoc, http://java.sun.com/j2se/1.4.2/docs/api/, (präsent August 2008)

Lebenslauf

Gerhard Hohner
Dinghoferstraße 63
4020 Linz

Persönliche Daten:	geboren am 20. Dezember 1956 in Linz, Oberösterreich
	österreichischer Staatsbürger, röm.-kath.
Familie:	Herbert Hohner, Schlossermeister (verstorben)
	Paula Hohner, Hausfrau
	1 Bruder (50 Jahre)
Werdegang:	1963-1967 Volksschule in Linz
	1967-1975 Gymnasium in Linz
	Juni 1975 Abschluss mit Matura
	1975-1981 Studium der Informatik in Linz, ohne Abschluss
	1981-1986 freier Mitarbeiter am Institut für
	Experimentalphysik 2
	1987 Ableistung des Wehrdienstes
	1987-1990 Programmierer bei der Firma Plasser
	1990-1995 beschäftigungslos
	Ab 1996 Pensionist
	1996 Wiederaufnahme des Studiums
	2004 Beginn der Diplomarbeit
	2005 Abschluss des ersten Studienabschnitts
	2008 voraussichtlicher Abschluss des Studiums

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benutzt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Linz, am

_____

Name