Hochschule
Augsburg University of
Applied Sciences

Technische Projektarbeit

Technische Informatik

Wintersemester 2008/09

Der HiCoVec Prozessor in Aktion

Dokumentation

Jakob Golus	912417
David Lucinkiewicz	911671
Andreas Müller	912440
Florian Richter	912142
Sebastian Rigorth	911786
Soeren Rühm	912213

Hochschule für
angewandte Wissenschaften –
Fachhochschule Augsburg
University of Applied Sciences

An der Fachhochschule 1
D-86161 Augsburg

Telefon +49 821 55 86-0
Fax +49 821 55 86-3222
www.hs-augsburg.de
info@hs-augsburg.de

Datum: 11.01.2009

Inhaltsverzeichnis

1. Einleitung

Die Aufgabe des Projekts „Der HiCoVec Prozessor in Aktion“ war es, die Diplomarbeit von Harald Manske, ein einstellbarer Vektorprozessor, in eine Entwicklungsumgebung zu implementieren. Die Entwicklungsumgebung wurde durch ein FPGA Bord und der Viscy-Software zur Verfügung gestellt. Die Aufgabe war nun, durch Anpassung der Software und Einbindung der Hardware den HiCoVec als lauffähigen Demo Prozessor zu präsentieren. Idealerweise sollte dies durch eine kleine Demoapplikation, Bildschirmausgabe und individueller Eingabe realisiert werden.

Um dieser Aufgabe im gegebenen Zeitrahmen gerecht zu werden, haben wir uns dazu entschieden, unsere Gruppe in zwei Teams zu unterteilen. Je drei Leute gehörtem dem Team Software und dem Team Hardware an, welche untereinander nochmal individuell die Aufgaben aufteilten. Zur Leitung und Koordinierung wurde zusätzlich ein Gruppenmitglied zum Teamleiter ernannt.

Als unterstützung für die Durchführung des Projektes und als Informationsquelle, haben wir eine Projekthomepage mit WIKI und TRAC erstellt:

http://hicovec.informatik.fh-augsburg.de/trac

Die Aufgaben, Änderungen und Entwicklungen der einzelnen Teams werden folgend erklärt.

2. SCOTCH – Hardware

2.1 Einleitung

System to
Connect,
Operate,
Transfer data and
Compute with the
Hicovec

SCOTCH – diesen Namen haben wir für den Hardware-Teil der Projektarbeit gewählt, um einerseits unsere Arbeit von den bisherigen Arbeiten abzugrenzen und um andererseits dennoch darauf verweisen zu können.

Das SCOTCH-System setzt sich auf der einen Seite aus dem von Harald Manske entworfenen HiCoVec Prozessor, und dem Viscy-System auf der anderen Seite zusammen. Zusätzlich wurden Verbindungen zur VGA-Schnittstelle und zur LCD-Anzeige hergestellt. Um einen Betrieb zu ermöglichen wurde ein Blockram (Xilinx Standard) als Speicher entworfen welcher über das angepasste Memory Interface adressiert werden kann. Sämtliche I/O lässt sich über eigene Adressebereiche ansprechen. Die Adressbereiche sind in der cfg.vhd benannt. Als Debug-Hardware ist zwischen dem Prozessor und der I/O-Hardware eine Debug-Shell implementiert worden.

2.2 Der HiCoVec Prozessor

2.2.1 Die Architektur:

Der HiCoVec Prozessor wurde im Zuge einer Diplomarbeit an der Hochschule Augsburg von Harald Manske entwickelt. Er setzt sich aus einer Skalareinheit(SISD **=** Single Instruction Single Data) zusammen, die direkt mit einer Vektoreinheit, die im Stile einer SIMD(single instruction multiple data) Architektur arbeitet, verbunden ist. Die Größe der Vektoreinheit lässt sich je nach Bedarf und Platz auf dem FPGA-Board einstellen. Beide Einheiten sind parallel aufgebaut und können daher voneinander unabhängige Aufgaben gleichzeitig bearbeiten. Das parallel aufgebaute Operationswerk der Vektoreinheit wirkt, anstatt auf einzelne Operanden, auf einen kompletten Vektor von Daten. Jeder dieser Vektoren umfasst mehrere Daten-Wörter, deren Länge im Befehl kodierbar ist. So könnte z.B. ein 128-Bit-Vektorregister in zweit 64-Bit-, vier 32-Bit-, acht 16-Bit-, sechzehn 8-Bit-Wörter geteilt werden. Die Abarbeitung der Befehle erfolgt parallel, was zu einem Geschwindigkeitsvorteil im Vergleich zur skalaren Abarbeitung führt. Das folgende Beispiel (Abb. 2.2.1) zeigt die Addition von 4 Wörtern umfassenden Vektorregistern mit einem Befehl, der in etwa in einem Takt ausgeführt werden kann. Die Skalareinheit der CPU benötigt für diese Berechnung 4 Takte.

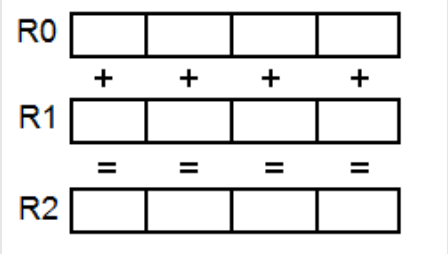


Bild 2.2.1: Beispielrechnung

Da die CPU als Soft-Core für unterschiedliche Anwendungen konzipiert ist, lassen sich folgende Parameter je nach Bedarf konfigurieren: Anzahl der Vektorregister, Anzahl der Wörter pro Vektorregister, Ein- bzw. Ausschalten der Multiplizierfunktion in der Skalar- und in der Vektoreinheit

getrennt, Zuschalten der Shuffle-Funktion(Mischen von Daten), Rotieren von Daten innerhalb des gesamten Vektorregisters. Näheres hierzu ist im Abschnitt 2.3.4 zu finden.

2.2.2 Der Befehlssatz

Der Befehlssatz gliedert sich in Befehle für die Skalareinheit, Befehle für die Vektoreinheit und in Befehle, welche in Kooperation beider Einheiten verarbeitet werden. Um den Befehlssatz verstehen zu können, sollten die Register des Prozessors bekannt sein. Die Register der Skalareinheit(siehe Abb. 2.2.2) sind als A, X und Y benannt. Sie haben 32 Bit Länge und nur der Inhalt des Registers A kann in den Speicher geschrieben werden. Je nach Befehl ist es zusätzlich für eine skalare oder kooperative Operation möglich, als skalare Quelle oder Ziel „0“ anzugeben. Dadurch wird entweder als Operand der Wert Null verwendet, oder das Ergebnis der Operation wird nicht in ein Register gespeichert. Außerdem kann für viele Befehle ein Direktwert als Operand aus dem Befehlswort genutzt werden.

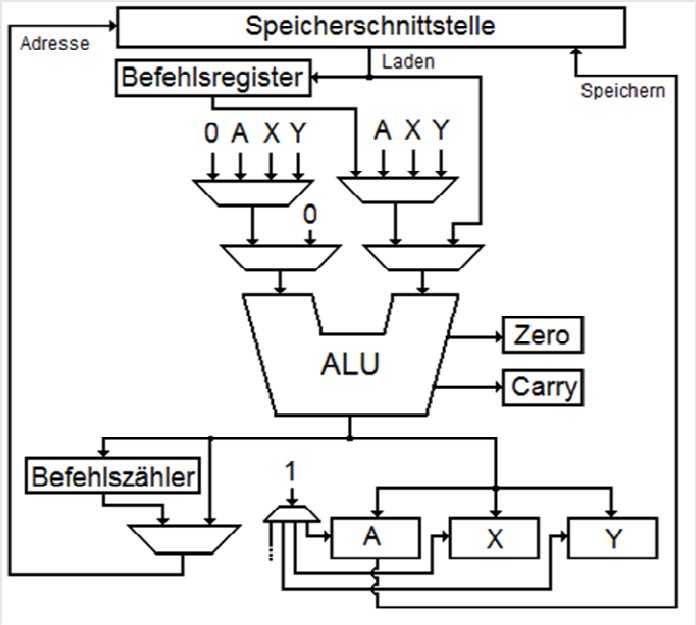


Bild 2.2.2: Die Skalareinheit

Die konfigurierbaren Vektorregister sind sowohl als Quell- als auch als Zielregister nutzbar. Daraus ergeben sich folgende Befehlsgruppen:

```
reg_axy  = A, X oder Y
reg_axy0 = A, X, Y oder 0
reg_axyi = A, X, Y oder Direktwert (16 Bit)
vregl = R0 … R15
vregh = R<0> … R<N>
breite_bw = B (8 Bit) oder W (16 Bit)
breite_voll = B (8 Bit), W (16 Bit), DW (32 Bit), QW (64 Bit)
```

Die vollständige Befehlstabelle wird unter 5.7.1 erklärt.

2.2.3 Befehlscodierung

Da die Wortlänge der Befehle 32 Bit beträgt wird ein Befehl sowohl zum Steuern der Skalar- als auch der Vektoreinheit verwendet. Die Aufteilung der Bereiche ist in Abb. 2.2.3 ersichtlich.

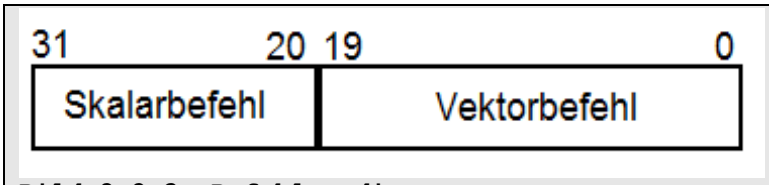


Bild 2.2.3: Befehlscodierung

12 Bit werden für die Befehle der Skalareinheit und übrigen 20 Bit für die Befehle der Vektoreinheit verwendet. Hieran sieht man, dass beide Einheiten mit einem Befehlswort angesprochen werden können. Man muss aber nicht immer beide Einheiten verwenden. Daher ist es möglich Direktwerte in die jeweils andere Einheit zu laden. Dazu müssen nur die ersten drei Bit des jeweils anderen Befehls mit 0 belegt sein. Dies hat zu Folge, dass diese Einheit einen NOP(No Operation)-Befehl durchführt. In den restlichen 9 bzw. 17 Bit kann dann der benötigte Direktwert stehen. (Beispiel siehe Abb. 2.2.4)

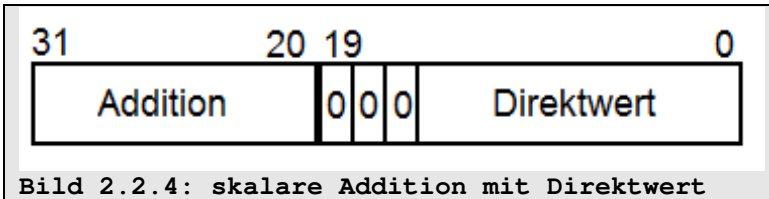


Bild 2.2.4: skalare Addition mit Direktwert

Die Hardwareseitige Codierung der Befehle wird in der folgenden Tabelle ersichtlich und ist weitestgehend selbsterklärend. Eine genaue Erklärung der Befehle ist im Kapitel 5.7.1 zu finden.

Befehl	Kodierung	Kommentar
Allgemein	d ➔ Zielregister s ➔ 1.Quellregister bzw. Adresse t ➔ 2. Quellregister bzw. Adresse r ➔ Vektor-Zielregister v ➔ 1. Vektor-Quellregister w ➔ 2. Vektor-Quellregister n ➔ Direktwert	
LD	`1000--ddsstt ----------` `1000--ddss00 00-nnnnnnnnnnnnnnnn`	
ST	`1010----sstt ----------` `1010----ss00 00-nnnnnnnnnnnnnnnn`	
Alu-Befehle	`01xxxxddsstt ----------` `01xxxxddss00 000-nnnnnnnnnnnnnnnn`	x ➔ Operation ADD: 0000 ADC: 0001 SUB: 0010 ABC: 0011 INC: 0100 DEC: 0110 AND: 1000 OR: 1001 XOR: 1010 MUL: 1011 LSL: 1100 LSR: 1110 ROL: 1101 ROR: 1111
Sprungbefehle	`00xxxxddsstt ----------` `00xxxxddss00 000-nnnnnnnnnnnnnnnn`	x ➔Sprungart JMP: 0001 JAL: 0001 JNC: 1100 JC: 1101 JNZ: 1110 JZ: 1111
Flag-Befehle	`110000--eeff ----------`	e ➔Flagauswahl f ➔neuer Wert CLC: 0100 SEC: 0101 CLZ: 1000 SEZ: 1010

HALT	00101------- --------------------	
NOP	000--------- --------------------	
VNOP	---------- 000----------------	
MOV	101110--sstt 10--0100rrrr-------- 101111dd--tt 10--0010rrrr--------	
MOVA	101101--ss-- 10--0110rrrr--------	
VLD	1001----sstt 10--0010rrrr--------	
VST	101100--sstt 10--0011----vvvv----	
VMOV	---------- 001-0001rrrrvvvv----	VMOV r,v
	000-nnnnnnnn 001-0010rrrr--------	VMOV r,v(n)
	000-nnnnnnnn 001-0011----vvvv----	VMOV r(n), v
VMOL/VMOR	---------- 001-1000rrrrvvvv----	VMOL r,v
	---------- 001-1100rrrrvvvv----	VMOR r,v
VALU-Befehle	---------- 01bbxxxxrrrrvvvvwwww	x ➔Operation VADD: 0000 VSUB: 0010 VAND: 1000 VOR: 1001 VXOR: 1010 VMUL: 1011 VLSL: 1100 VLST: 1110 b ➔Wortbreite 8 Bit: 00 16 Bit: 01 32 Bit: 10 64 Bit: 11
VSHUF	000-pppppppp 11bbssssrrrrvvvvwwww	x ➔ Wortbreite s ➔ Register-Auswahl p ➔ Permutation

2.2.4 Mikroarchitektur

Die Skalar- und die Vektoreinheiten können weitestgehend autonom agieren und stehen nur über wenige Daten und Steuerleitungen miteinander in Kontakt (Abb.2.2.6).

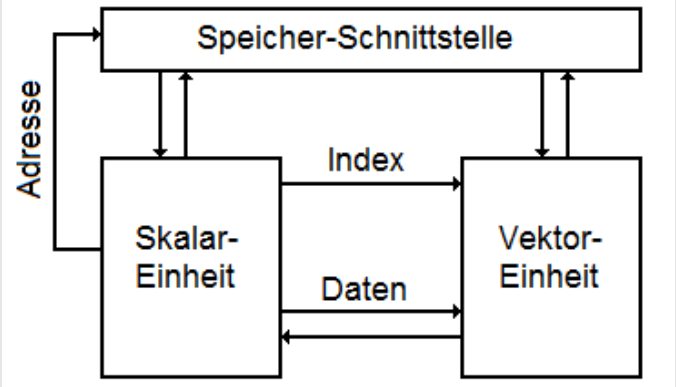


Bild 2.2.6: Mikroarchitektur des HiCoVec Prozessors

Die Speicherschnittstelle ist zwar kein direkter Bestandteil des Prozessors, aber ohne diese ist der Betrieb nicht möglich. Nähere Details zur Speicherschnittstelle sind im Kapitel 2.3.2 Memory Interface zu finden, weshalb hier nicht näher darauf eingegangen wird.

Die Struktur der Skalareinheit ist schon in Abb. 2.2.2 zu sehen gewesen. Für genauere Details der Funktionsweise empfiehlt sich ein Blick in die Diplomarbeit von Harald Manske auf die Seiten 36 bis 40 für die Skalareinheit und Seiten 40 bis 42 für die Vektoreinheit. In Abb. 2.2.7 ist der grundsätzliche Aufbau der Vektoreinheit skizziert.

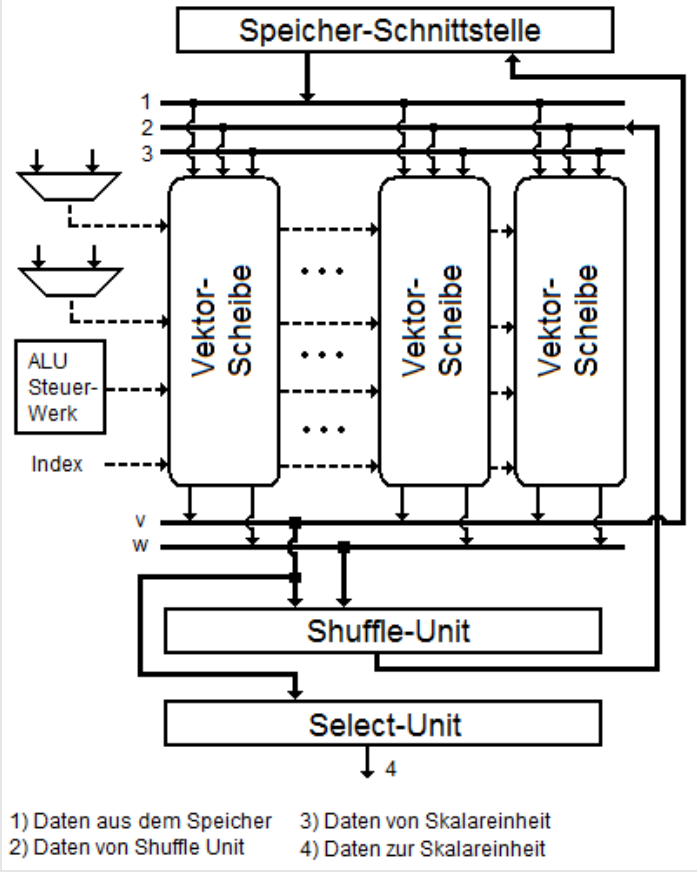


Bild 2.2.7: Struktur der Vektoreinheit

2.2.5 Schnittstelle zwischen Skalar- und Vektoreinheit

Wie in Abb. 2.2.6 zu sehen ist, sind die Skalar- und Vektoreinheit nur über wenige Leitungen miteinander verbunden.

2.3 Weiter Teile des SCOTCH-SYSTEM

2.3.2 Das Memory Interface

Das Memory Interface ist eine Logikebene die zwischen dem eigentlichen Prozessor und seiner Peripherie liegt und steuert bei allen komplexeren Prozessoren den externen Zugriff. Beim HiCoVec hat Harald Manske schon ein dafür entsprechendes Memory Interface geschrieben. Es regelt den Zugriff anhand der Steuersignale des Prozessors und der Peripherie.

Dieses konnte allerdings nicht ohne Änderungen übernommen werden, da Harald Manske einfacher halber von einem idealen Speicher ausgegangen ist. In diesem idealen Fall lagen die Daten der IO schon nach einem Takt an den Leitungen an. In einer realen Testumgebung ist dies allerdings nicht der Fall. Eine, für die Steuerung wichtige, Ready-Leitung zwischen Prozessor und Peripherie war ebenfalls noch nicht Bestandteil seines Memory Interface.

Um den Prozessor in einem FPGA in Betrieb nehmen zu können musste deshalb das Memory Interface für unsere Zwecke angepasst werden. Es wurde um das Ready-Signal erweitert und die einzelnen States wurden an die reale Umgebung angepasst.

2.3.3 Die DBG-Shell

Die DBG-Shell (sprich Debug-Shell) ist zwischen dem Prozessor und dem restlichen System angesiedelt. Ein klares Bild der geographischen Lage davon kann man sich beim Betrachten der folgenden Grafik machen:

./Zeichnungen/SCOTCH_Schematische_darstellung.pdf

Die DBG-Shell erlaubt somit die Kontrolle über das gesamte System von außen per seriellen Anschluss. Für die Einordnung und zum generellen Verständnis des Gesamtsystems reichen diese Informationen aus. Eine genaue Erläuterung der Funktionsweise wird im Kapitel 4.1 dieser Dokumentation gegeben.

2.3.4 Konfiguration mit cfg.vhd

Die Konfigurationsdatei cfg.vhd beinhaltet sämtliche Parameter welche eine variable Skalierung des HiCoVec Prozessors ermöglichen. Diese Datei ist in allen VHDL-Dateien des Systems integrierbar, was es auch ermöglicht spezielle Parameter der I/O zu integrieren.

Der nachfolgende Abschnitt enthält alle wichtigen Parameter welche in der Konfigurationsdatei eingestellt werden können.

HiCoVec Konfiguration:

Anzahl der Vektorregister:

```
constant n: integer := 16;
```

Anzahl der Vektorscheiben (nur gerade Zahlen):

```
constant k: integer := 16;
```

Skalare Multiplikation aktivieren:

```
constant use_scalar_mult : boolean := true;
```

Vektormultiplikation aktivieren:

```
constant use_vector_mult : boolean := true;
```

Integrieren der Shuffleeinheit:

```
constant use_shuffle : boolean := false;
```

Maximale Shuffle-breite (muss durch 4 teilbar sein):

```
constant max_shuffle_width : integer := 0;
```

Vektorshift aktivieren:

```
constant use_vectorshift : boolean := true;
```

Breite der Vektorverschiebung in Bit:

```
constant vectorshift_width : integer := 4;
```

I/O Konfiguration:

Anzahl der angeschlossenen I/O Elemente:

```
constant slaves: integer := 6;
```

Basisadressen der angeschlossenen I/O:

```
--Slave 0: BLOCKRAM 0x00000000 - 0x00001FFF
constant blockram_base_adr: std_logic_vector(31 downto 0) :=
"00000000000000000000000000000000";
--Slave 1: VGA 0x00002000 - 0x00003FFF
constant vga_base_adr: std_logic_vector(31 downto 0) :=
"00000000000000000010000000000000";
--Slave 2: DEBUG_IO 0xFFFF0010 - 0xFFFF0017
constant debug_io_base_adr: std_logic_vector(31 downto 0) :=
"11111111111111110000000000010000";
--Slave 3: IOMISC 0x0000FFFF
constant iomisc_base_adr: std_logic_vector(31 downto 0) :=
"00000000000000001111111111111111";
--Slave 4: LCD 0x0000FFF0
constant lcd_base_adr: std_logic_vector(31 downto 0) :=
"00000000000000001111111111110000";
--Slave 5: ASCII 0x00004000 - 0x00004400
constant ascii_base_adr: std_logic_vector(31 downto 0) :=
"00000000000000000100000000000000";
```

Weitere Einstellmöglichkeiten welche in die cfg.chd integriert werden könnte, wäre zum Beispiel die Größe des BLOCKRAMS oder die Farbtabelle der VGA-Schnittstelle.

2.4 Angeschlossene I/O

Für eine Demoapplikation, die den Prozessor in Betrieb zeigen soll, wird nebst lauffähigem Prozessor natürlich auch einiges an angeschlossenen I/O Geräten benötigt. Das wichtigste hierbei ist ein Speicher, eine Aus- und Eingabe und eine dazugehörige Debug Umgebung.

Die von uns genutzten I/O Schnittstellen, sowie die verwendete Bus-Logic wird in den nächsten Punkten näher erklärt.

2.4.1 Bus-Logic

In der Bus-Logic wird mittels eines Daisy-Chain Verfahrens die aktuell adressierte Hardware für den Prozessor freigeschalten.

Von der CPU kommend hat die Bus-Logic als Eingänge die read- und write-Leitungen. Die Rückmeldung erfolgt durch das ready-Signal. Als Verbundung zur IO verfügt die Bus-Logic über 4 Busse. In Richtung IO gehen der read- und der write-Bus. Als Eingänge von der IO kommend sind der

ready- und der adressed-Bus vorhanden. Die Breite der Busse ist entsprechend der in der cfg.vhd eingestellten Anzahl der IO-Slaves.

Die IO wird nun folgendermaßen mit der Bus-Logic Verbunden: Jede IO besitzt ebenfalls genau diese vier Leitungen (read, write, ready, adressed). Die Leitungen werden entsprechend ihrer Priorität in der Daisy-Chain an den entsprechenden Bus angeschlossen. Das Gerät mit der höchsten Priorität wird also an Leitung 0 der einzelnen Busse angeschlossen.

```
RAM_adressed      => IO_adressed(0),
RAM_rd            => IO_rd(0),
RAM_wr            => IO_wr(0),
RAM_ready   => IO_ready(0)
```

Auf dieselbe Weise werden nun alle weiteren Geräte, entsprechend der gewünschten Priorität, mit der Bus-Logic verbunden.

Der Prozessablauf innerhalb der Bus-Logic geht nun folgendermaßen vonstatten: Über einen interner Zähler nun, von 0 bis k (k = Anzahl der Slaves), überprüft nun die Bus-Logic ständig alle Eingänge des adressed-Bus ob eine Hardware meldet, dass sie adressiert wurde. Ist dies der Fall, so werden die read-, write-, und ready-Leitungen des entsprechenden Geräts zur CPU durchgeschaltet. Dies wird erst wieder zurückgenommen, wenn sowohl die adressed- als auch die ready-Leitung wieder auf 0 gesetzt werden. Da dieser Vorgang immer bei Slave 0 beginnt, hat dieser die höchste Priorität der IO-Geräte.

2.4.2 Block-RAM

Nebst externen Speicher welcher sich, unter anderem, auch auf dem Spartan-3A Bord befindet besitzt der FPGA Chip selber einen fest integrierten Block Ram. Dieser On Chip Speicher lässt sich um einiges leichter als Programmspeicher verwenden als die Chip externen Speicher. Da sich der Block RAM direkt auf dem Chip und nebst den eigentlichen Logikeinheiten des FPGAs befindet, verfügt er über eine sehr gute Reaktionszeit.

Die Einbindung erfolgt über eine von Xilinx vorgegebene Architektur. Sie gibt an, das für den verwendeten Speicher der Block Ram verwendet werden soll.

```
architecture RTL of blockram is
      constant words: integer := (2 ** width);

      type t_memory is array(0 to words-1)of STD_LOGIC_VECTOR(31 downto 0);
      signal ram: t_memory;
      attribute ram_style: string;
      attribute ram_style of ram: signal is "block";
      signal br_adr: std_logic_vector (width-1 downto 0);
      signal br_di, br_do: std_logic_vector (31 downto 0);
      signal br_en, br_we: std_logic;
```

Der Nachteil des Block RAMs ist die geringe Speichergröße. Auf dem Chip begrenzt beträgt der reine Speicher 360k Bits. Diese können, auf Kosten der Logikzellen des FPGAs, noch um weitere 92k Bits vergrößert werden. Für unser gewähltes Demoprogramm reicht der verfügbare Speicher aus, doch ein komplexeres Programm oder eine größere Grafikapplikation würde einen zusätzlichen, externen Programmspeicher benötigen.

2.4.3 VGA

Um eine visuelle Darstellung zu realisieren, ist auf dem Spartan3a Starter Kit eine VGA-Schnittstelle integriert. Diese Schnittstelle verfügt über einen 14-Bit A/D-Wandler. Dieser erstellt aus den vom FPGA kommenden digitalen Signalen die entsprechenden analogen Werte, welche von einem VGA-Monitor oder einem TFT-Bildschirm dargestellt werden können.

Da das Ansprechen eines VGA-Bildschirmes ein komplexes Timing-Verfahren benötigt, ist die Funktionsweise der VGA-Schnittstelle in Kapitel 3 genauer beschrieben.

2.4.4 IO-MISC

Die IO-Misc erweitert das System um einige verschiedene Ansprech und Steuermöglichkeiten. So werden über das Modul die LED Zeile des Spartan-3A verwaltet. Durch Ausgabe entsprechender Bitmuster können sie wahlweise An und Aus geschaltet werden.

Als zusätzliche Eingabe Möglichkeit dient hier der festverdrahtete Rotary Button des Bords. Durch drehen oder Drücken des Buttons werden die Signale über das IO-Misc Modul an den Prozessor zur Verarbeitung weitergeleitet.

Zusätzlich zu den bereits integrierten Schnittstellen, können über die zwei Pin-Leisten J18 und J20 benutzerdefinierte Eingaben gemacht werden. Wie weiter unten beschrieben, greifen die zwei Leisten Bitmuster ab und geben sie an das System weiter.

2.4.5 Debug-Hardware

Wie unter Kapitel 2.2.1 beschrieben, verfügt der Prozessor über Debug-Leitungen, über welche direkt die Register der Skalareinheit, sowie bestimmte Flags ausgelesen werden können. Um von extern auf diese Debug Leitungen zugreifen zu können, wurde eine Hardware erstellt, welche es ermöglicht, die Debug-Leitungen auf den Datenbus des Gesamtsystems zu legen.

Der wesentliche Bestandteil der Debug-IO ist ein Multiplexer, welcher abhängig von der angelegten Adresse, einen der 6 Registereingänge auf den Datenbus legt. Natürlich muss auch hier durch einen Tristate sichergestellt werden, dass sich die unterschiedlichen Busgeräte nicht gegenseitig stören.

In der nachfolgenden Tabelle sind die Adressen aufgelistet, welche angelegt werden müssen, um das jeweilige Register an den Datenbus zu legen. Zu dieser Adresse muss noch die Debug-IO-Basisadresse (Standard: 0xFFFF0010) addiert werden.

Adresse	**Register / Flags**
0x0	Register A (32 Bit)
0x1	Register X (32 Bit)
0x2	Register Y (32 Bit)
0x3	Instruction Register (IR) (32 Bit)
0x4	Instruction Counter (IC) (32 Bit)
0x5	Bit 0: Carry-Flag Bit 1: Zero-Flag Bit 2: IR-Ready-Flag Bit 3: CPU-Halted-Flag Bits 4 - 31: Immer 0
0x6	unbelegt
0x7	unbelegt

Diese Hardware ist besonders sinnvoll in einer Debug Umgebung einzusetzen, da hier im schrittweisen Programmablauf jederzeit, durch Anlegen der jeweiligen Adresse, die Registerbelegung ausgelesen werden können. Details dazu finden sich in Kapitel 4 wieder.

2.4.6 LC-Display

Auf dem Spartan3a Starter Kit befindet sich ein 2x16 Zeichen LC-Display, welches über einen HD44780 Standard-Industrie-Controller verfügt.

Dieser Controller ermöglicht eine einfache Ansteuerung des Displays über einen 8-Bit Datenbus und 3 zusätzliche Steuersignale.

Steuersignal	**Funktion**
Register Select (RS)	0: Instruction Register ausgewählt 1: Datenregister ausgewählt
Read/Write (RW)	0: Schreibzugriff auf die Register 1: Lesezugriff auf die Register
Enable (E)	0: Disable 1: Enable

Desweiteren verfügt das Display über ein Instruction Register, welches über RS = 0 angewählt werden kann. In diesem Register werden die genaue Funktionsweise des Displays sowie die Cursorposition festgelegt. Der Controller kann auch über einen, nur 4-Bit breiten Datenbus angesprochen werden, um Ressourcen zu sparen.

Da das Display Hardwareseitig zurzeit nur rudimentär integriert ist, muss die Initialisierung des Displays rein Softwaretechnisch erfolgen.

Das Display wird über eine Basisadresse angesprochen (Standard: 0x0000FFF0). Dabei entsprechen die niederwertigsten 8 Bit des Datenbusses den 8 Datenbits des Controllers. Bit 8 entspricht der Negation des RS Signales. Das RW Signal wird immer auf 0 (Schreibzugriff) gesetzt. Das EN Signal liegt so lange auf 1, wie das write-Signal des Datenbusses gesetzt ist.

Um das Display zu initialisieren, muss eine bestimmte Codefolge an den Controller gesendet werden:

- 0x30 bei gesetzem Bit 8 (Instruction Register ausgewählt) für min. 12 Takte
- Min. 5ms warten
- 0x30 bei gesetzem Bit 8 (Instruction Register ausgewählt) für min. 12 Takte
- Min. 0,1ms warten
- 0x30 bei gesetzem Bit 8 (Instruction Register ausgewählt) für min. 12 Takte
- Min. 0,05ms warten
- 0x20 bei gesetzem Bit 8 (Instruction Register ausgewählt) für min. 12 Takte
- Min. 0,05ms warten

Um nun Daten an das Display schreiben zu können, müssen noch folgende Befehle an das Display gesendet werden:

- 0x28: Display bereit setzten
- 0x06: Cursor auf 1. Stelle, automatisch inkrementieren
- 0x0C: Display einschalten, Cursor blinkt

Nun können, wenn das Bit 8 auf 0 gesetzt wird, dem ASCII-Zeichensatz entsprechende Zeichen an das Display gesendet werden.

Eine Verbesserungsmöglichkeit an dieser Stelle ist, sowohl die Initialisierung als auch die Auswahl der entsprechenden Register rein über VHDL-Code zu realisieren, da das Timing der Initialisierung softwareseitig eher schwer realisierbar ist.

2.4.7 Erweiterungsmöglichkeiten

Das gesamte IO-System der SCOTCH-Hardware ist auf eine einfache Erweiterbarkeit durch weitere Hardware ausgelegt. So muss an dem SCOTCH-System keine Änderung vorgenommen werden. Es muss lediglich die Datei cfg.vhd angepasst werden, sowie die entsprechende Hardware als Entity in der scotch.vhd integriert und verkabelt werden.

Jede Hardware welche an das SCOTCH-System angeschlossen werden soll, muss mindestens folgende Eigenschaften aufweisen:

- Ein Bidirektionaler, 32-Bit Breiter Datenbus welcher über Trisates an den Datenbus des Systems angschlossen wird.
- Ein 32 Bit breiter Adressbus (IN) von dem alle 32 Leitungen für die Adressierungsberechnung benötigt werden.
- Eine Adressed-Leitung (OUT), welche angibt ob die Hardware angesprochen wird.
- Eine read-Leitung (IN), welche angibt dass von der Hardware gelesen wird.
- Eine write-Leitung (IN), welche angibt dass an die Hardware geschrieben wird.
- Eine ready-Leitung (OUT), welche dem Prozessor signalisiert, dass die Daten auf dem Datenbus liegen, bzw. die Daten fertig geschrieben wurden.

Es müssen folgende Librarys verwendet werden. Auch die Portmap sollte mindestens über folgende Elemente verfügen:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;
use WORK.CFG.ALL;

entity hardware is
      port (
            clk: std_logic;
            adr: in std_logic_vector (31 downto 0);
            data: inout std_logic_vector (31 downto 0);
            adressed: out std_logic;
            rd, wr: in std_logic;
            ready: out std_logic;
      );
end hardware;
```

Der Bidirektionale Datenbus wird in VHDL folgendermaßen realisiert:

```
data <= zu_sendende_daten(31 downto 0)
when rd = '1'
else "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
```

Wobei „data“ die INOUT Leitung zum SCOTCH-System darstellt.

Die „Adressed“-Leitung sagt der BUS-Logic, dass die am Adressbus angelegte Adresse der der Hardware entspricht. Dazu wird die in der cfg.vhd befindliche Basisadresse mit der angelegten Adresse verglichen. Dabei werden die unteren Bits, entsprechend des benötigten Adressraumes der Hardware, maskiert.

```
adressed <= '1'
when hardware_base_adr (31 downto width) = adr (31 downto width)
else '0';
```

Für das setzen des ready-Signales wird am besten ein Prozess verwendet. Dieser wählt entsprechend der gesetzten read- oder write-Leitung die entsprechende Funktion der Hardware aus und setzt auch das ready-Signal.

```
process (clk)
begin
      if clk'event and clk = '1' then
            ready <= '0';
            if rd = '1' then
                  ready <= '1';
            end if;
            if wr = '1' then
                  interner_data_buffer (31 downto 0)<= data;
                  ready <= '1';
            end if;
      end if;
end process;
```

Diese Hardware kann nun problemlos an den Daten- und Adressbus des SCOTCH-Systems, sowie an die read-, write-, ready- und addressed-Leitungen der BUS-Logic angeschlossen werden. Desweiteren muss noch die Basiadresse der Hardware in die cfg.vhd geschrieben werden (siehe Kapitel 2.2.5).

2.5 Verwenden der Hardware

In diesem Kapitel wird kurz beschrieben wie man aus den vorhandenen VHDL-Dateien ein Lauffähiges System erstellt.

Prinzipiell kann der Prozessor auf jedem beliebigen FPGA, welcher über die benötigten Ressourcen verfügt, geladen werden. Inwiefern die angeschlossene Hardware (VGA ect.) zu konfigurieren ist, muss dann allerdings den entsprechenden Dokumentationen entnommen werden.

Der von uns erstellte VHDL-Code ist für ein SPARTAN3A-StarterKit Reference-Board mit einem Xilinx XC3S700A FPGA und der FG484 Hardware optimiert. Für andere Hardware muss mindestens die .ucf-Datei angepasst werden. In dieser Datei sind zum einen die Takte festgelegt, zum anderen werden die Pins festgelegt, an welche die Signale nach außen gelegt werden.

2.5.1 Erstellen und Laden des .bit-Files

Um ein .bit File zu erstellen, müssen alle VHDL-Dateien zunächst synthetisiert werden. Dies kann zum Beispiel mit den Synthese-Tools von Xilinx geschehen. Alle Dateien sind im Anhang 8.3.1 aufgelistet.

Nun kann das System implementiert werden. Dazu ist noch die Datei scotch.ucf nötig.

Der nächste Schritt ist nun ein Programming-File (scotch.bit) zu erstellen. Auch dieser Schritt kann, wie die beiden zuvor, auch mit den Xilinx Tools durchgeführt werden.

Mit dem Xilinx-Tool „Impact“ kann nun die Datei scotch.bit über ein USB-Kabel oder das JTAG Interface auf den FPGA übertragen werden.

2.5.2 Laden auf den internen Flash des Spartan3a-Boardes

Um SCOTCH dauerhaft auf das Board zu integrieren, kann das System auch auf den auf dem Board integrierten Flash übertragen werden. Wird das Board dann eingeschaltet, so wird der Inhalt des Flash-Speichers in den FPGA geschrieben und das System ist sofort einsatzbereit.

Um dies zu realisieren, erstellt man zunächst eine Datei, welche für den blockweisen Zugriff des Flash-Speichers optimiert wurde. Dies kann mit dem, in der grafischen Oberfläche von Impact integrierten Tool „PROM File Formatter“, geschehen. Die so erstellte scotch.mcs Datei kann nun auf auf den XCF04S-Flash des Boards übertragen werden.

Damit nun der Flash-Speicher beim starten in den FPGA geladen wird, müssen noch folgende Jumper gesetzt werden:

```
J26: „Master Serial“ (Alle 3 Jumper aktiviert)
J46: „DONE“ und „CE PROM“ verbinden
```

2.5.3 Anschließen externer Peripherie

Um mit dem System arbeiten zu können, muss zumindest die RS232-Schnittstelle J36 (Farbe: schwarz) mit einem Debug-PC verbunden sein. Für eine eventuelle Programmierung des FPGAs, muss das USB-Kabel, bzw. das JTAG Interface, angeschlossen werden.

Als weitere Peripherie kann an der VGA-Schnittstelle ein handelsüblicher CRT-Monitor oder TFT-Bildschirm angeschlossen werden. Bei gestartetem System ist hier nun der Schriftzug „SCOTCH“ in grüner Farbe zu sehen.

Die Bit 12 bis 15 der IOMISC-Hardware können an dem Erweiterungsstecker J18, die Bit 16 bis 19 am Erweiterungsstecker J20 des Boardes angesprochen werden.

Auf dem Reference-Board sind bereits die 8 LEDs, sowie die Schalter und Taster der IOMISC fest integriert. Wenn das System bereit ist, so zeigen die LEDs das Muster „10011001“ an. Außerdem sind das TFT-Display und der Rotary-Button bereits aufgelötet.

3. VGA

3.1 Einleitende Gedanken

Eigentlich verfügt die SCOTCH Hardware mit den Tastern und den LEDs bzw. dem LC-Display bereits über alle benötigten I/O um einen funktionierenden Betrieb zu ermöglichen. Allerdings kann man mit den Ausgabegeräten zwar viele Informationen anzeigen, jedoch ist die Darstellung auf Bitmuster oder ASCII-Zeichen beschränkt. Der überlegene Vorteil eines Vektorprozessors kann mit einem Vollgrafikdisplay wesentlich besser ausgespielt werden.

Aus diesem Grund haben wir uns dazu entschlossen, die auf dem Reference-Board integrierte VGA-Schnittstelle in die SCOTCH Hardware zu integrieren.

3.2 Die VGA Schnittstelle

Auf dem Reference Board befindet sich eine 15-polige SUB-D Buchse, an welche ein VGA-kompatibles Display angeschlossen werden kann.

Die VGA Schnittstelle wird über einen 14-Bit breiten Bus direkt vom FPGA aus angesprochen. Die 14 Bit unterteilen sich in je ein Bit für die horizontale und ein Bit für die vertikale Synchronisation des Elektronenstrahls. Die 12 weiteren Bits sind für die Farbgebung zuständig. Jede der 3 RGB-Anschlüsse (**R**ot **G**rün **B**lau) verfügt über einen 4-Bit D/A-Wandler. Dieser erstellt aus den codierten Signalen ein analoges Signal, welches an den Monitor übertragen wird. So kann das Display je Farbe 16 verschiedene Helligkeitsstufen darstellen, was ein maximales Farbspektrum von 4096 Farben zulässt.

Aufgrund des auf dem Board integrierten 50MHz Systemtaktes, ist eine Auflösung von bis zu 800x600 Pixel bei 60Hz Bildwiederholfrequenz möglich. Größere Auflösungen sind bei entsprechend niedrigerer Bildwiederholrate möglich.

Für unser SCOTCH haben wir uns, vor allem aus Mangel an freiem Block-RAM, für eine Auflösung von 640x480 bei 60Hz und 4Bit Farbtiefe (16 Verschiedene Farben) entschieden. Außerdem wird durch Beschneidung oben und unten sowie einer Pixelverdopplung eine tatsächliche Auflösung von 320x200 erzeugt. Diese Auflösung und Farbtiefe entspricht, eher zufällig, genau der des legendären Heimcomputers Commodore 64.

3.3 Der VGA-Core

Der VGA-Core ist für das relativ komplexe Timing, für die Adressberechnung des VGA-Speichers und für die Berechnung der Farbwerte zuständig.

3.3.1 Das VGA-Timing

Das Timing einer VGA-Schnittstelle orientiert sich vor allem an der Tatsache, dass ein CRT-Monitor das Bild mit Hilfe eines Elektronenstrahls erzeugt.

So tastet der Elektronenstrahl in unserem Fall 60-mal pro Sekunde den gesamten Schirm ab. Dabei beginnt der Elektronenstrahl in der linken oberen Ecke des Schirmes. Er „wandert“ von links nach rechts. Dabei ändert er 640-mal seine Intensität, wodurch die Helligkeitswerte der einzelnen Pixel in jeder Zeile wiedergegeben werden. Nun muss der Strahl wieder ganz nach links, sowie eine

Zeile weiter nach unten. Dafür benötigt er eine gewisse Zeit, in welcher er seine Intensität auf Null setzen muss um das darzustellende Bild nicht „durchzustreichen“. Dieser Vorgang wird nun für alle 480 Zeilen wiederholt und am Ende muss der Elektronenstrahl wieder in die linke obere Ecke gelenkt werden. Siehe auch Grafik 3.3.1.

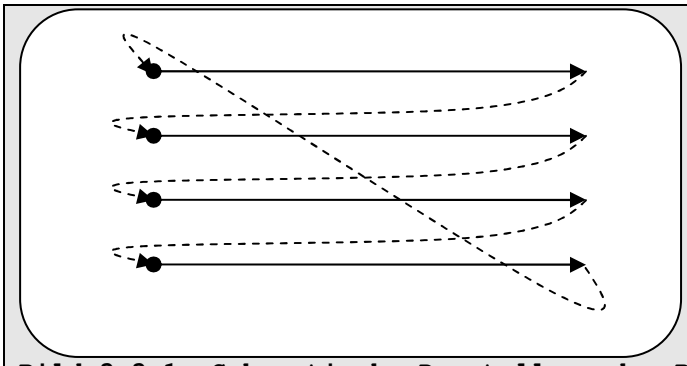


Bild 3.3.1: Schematische Darstellung der Bilderzeugung

Dieses Timing wird in unserer Hardware über 2 Zähler (horizontal und vertikal) realisiert. Zunächst muss dazu der 50MHz Takt auf 25MHz halbiert werden, um eine Auflösung von 640x480 zu ermöglichen.

Der horizontale Zähler hat einen Wert von 0 bis 800, der vertikale eine Wert von 0 bis 521. Daraus ergibt sich bei einem Takt von 25MHz die Bildwiderholrate von 60Hz (600x521x60 = 25*10^6). Beide Zähler stehen zu Beginn auf 0.

Der horizontale Zähler beginnt nun bei jeder steigenden Taktflanke um 1 nach oben zu zählen. Im Bereich von 0 bis 97 wird das HSYNC Bit auf 0 gesetzt, was den Elektronenstrahl dazu veranlasst nach links zu wandern. Bei einem Wert von 144 wird die Bilddarstellung für die nächsten 640 Takte aktiviert. Dies ist der Bereich in dem der Elektronenstrahl über den Bildschirm wandert. Erreicht der horizontale Zähler seinen Endwert von 800, so wird er zum einen auf 0 gesetzt, zum anderen wird der vertikale Zähler inkrementiert.

Befindet sich der vertikale Zähler im Bereich von 0 bis 3, so wird das VSYNC Bit auf 0 gesetzt, was den Elektronenstrahl nach oben wandern lässt. Die Bilddarstellung wird aktiviert, sobald der Zähler auf einem Wert von 71, sowie den nachfolgenden 400 Werten steht. Wenn nicht, wie in unserem Fall eine Darstellung von 640x400, sondern die volle Bildhöhe von 480 gewünscht wird, so muss die Bilddarstellung bereits bei 31 beginnen und für 480 Zählvorgänge erfolgen. Erreicht dieser Zähler seinen Endwert von 521, so wird auch dieser wieder auf 0 gesetzt und die Bilddarstellung beginnt von vorne.

3.3.2 Die VGA-Adressberechnung

Da unser System über eine 32-Bit breiten Datenbus verfügt, für die Darstellung eines Pixels jedoch nur 4 Bit benötigt werden, werden um Speicher nicht zu verschwenden, in einem Datenwort 8 Pixel gespeichert. Aus diesem Grund muss die Speicherzugriffsadresse nur alle 8 Pixel inkrementiert werden.

Wir verwenden jedoch eine Pixelverdopplung. Das heißt jeder Pixel wird zweimal abgebildet und jede Zeile wird ebenfalls doppelt wiedergegeben um effektive Auflösung von 320x200 zu realisieren. Diese Tatsache macht jedoch die Adressberechnung komplizierter.

Die Adresse des VGA-Speichers wird nun, so lange die Bilddarstellung läuft, alle 16 Takte um eins erhöht. Also bei 640 Zeichen in jeder Zeile 40-mal. Innerhalb dieser 16 Takte werden nun, mit Hilfe einer CASE-SELECT Anweisung, die 4 benötigten Datenbits jeden zweiten Takt an die entsprechende Stelle des 32-Bit Datenbusses gesetzt. Wird die nächste Zeile erreicht, so wird zunächst wieder dieselbe Adresse wie zu Beginn der vorherigen Zeile angelegt, um die Zeile zu

verdoppeln. Um sicherzustellen, dass die Zeile nicht bereits doppelt dargestellt wurde, wird ein eigener Zähler „linedoppler“, welcher von 0 bis 1 zählt, inkrementiert.

Hat der Elektronenstrahl dann das Ende des Bildschirmes erreicht, so wird der Adresszähler wieder auf 0 gesetzt und das Darstellen des VGA-Speicherabbildes wird wiederholt.

3.3.3 CLUT (Color LookUp Table)

Die D/A-Wandler der VGA-Schnittstelle erwarten je 4 Bit breite Daten, also insgesamt 12 Bit. Jedoch haben wir in unserem Fall nur eine Farbtiefe von 4 Bit, also müssen für jede der 16 möglichen Bitkombinationen ein entsprechender 12-Bit-Wert an die Schnittstelle übertragen werden. Für diesen Zweck haben wir eine Color-Lookup-Table (kurz: CLUT) integriert. In dieser Funktion wird über eine CASE/WHEN Abfrage zu jedem der 16 Werte der entsprechende 12-Bit Wert ausgegeben. In dieser Tabelle kann also aus den 4096 möglichen Farben zu den 16 Werten die gewünschte Farbe gesetzt werden.

```
function clut(input : in std_logic_vector(3 downto 0)) return
std_logic_vector is
begin
      case (conv_integer(input)) is
                           --"RRRR"&"GGGG"&"BBBB"
            when 15 => return "0000"&"0000"&"0000";   --schwarz
            when 14 => return "0100"&"0000"&"0000";   --dunkelrot
            when 13 => return "0000"&"0100"&"0000";   --dunkelgruen
            when 12 => return "0100"&"0100"&"0000";   --dunkelgelb
            when 11 => return "0000"&"0000"&"0100";   --dunkelblau
            when 10 => return "0100"&"0000"&"0100";   --dunkelmagenta
            when  9 => return "0000"&"0100"&"0100";   --dunkelcyan
            when  8 => return "0011"&"0011"&"0011";   --dunkelgrau
            when  7 => return "0111"&"0111"&"0111";   --hellgrau
            when  6 => return "1111"&"0000"&"0000";   --rot
            when  5 => return "0000"&"1111"&"0000";   --gruen
            when  4 => return "1111"&"1111"&"0000";   --gelb
            when  3 => return "0000"&"0000"&"1111";   --blau
            when  2 => return "1111"&"0000"&"1111";   --magenta
            when  1 => return "0000"&"1111"&"1111";   --cyan
            when  0 => return "1111"&"1111"&"1111";   --weiss
            when others => return "000000000000";
      end case;
end function clut;
```

Über die nachfolgende VHDL-Zeile kann auf die Funktion zugegriffen werden:

```
RGB <= clut(pixel);
```

Wobei RGB ein 12 Bit breiter Vektor und Pixel ein 4 Bit breiter Vektor ist.

3.4 Der VGA-RAM

Das Problematische an einem VGA-Speicher ist die Tatsache, dass nicht nur der Prozessor au ihn zugreifen muss, sondern auch die VGA-Schnittstelle.

Zunächst bestand die Idee über einen Multiplexer, welcher je nach gewünschter Zugriffsart entweder der CPU oder dem VGA-Core, Zugriff auf den Speicher ermöglicht. Das Problem an diesem

System war jedoch, dass die VGA-Schnittstelle ein sehr präzises Timing benötigt. So würde der Zugriff auf den Speicher während der Bilddarstellung immer auf den VGA-Core liegen. Also könnte der Prozessor nur in den kurzen Zeitspannen, während der Elektronenstrahl in die nächste Zeile wandert auf den Speicher zugreifen. Hierbei besteht jedoch auch das Problem, wie man mit der Situation umgehen soll, wenn der Schreibzugriff auf den Speicher noch nicht abgeschlossen ist, jedoch der VGA-Core seinen Zugriff bereits benötigt.

Die Lösung dieses Problems fand sich im Xilinx-ISE Handbuch wieder. So kann auf den internen Block-RAMs des FPGA über einen Dualport zugegriffen werden. Das heißt, dass auf diesen Speicher gleichzeitig über zwei getrennte Daten- und Adressleitungen zugegriffen werden kann. Zwar kann nur über einen Port der Speicher beschrieben werden, jedoch benötigt der VGA-Core nur einen Lesezugriff, wodurch dies kein Problem darstellt. Desweiteren können die beiden Ports sogar unterschiedlich getaktet werden. So kann auf den Leseport des VGA-Cores immer mit dem vollen 50MHz zugegriffen werden und der Lese/Schreib-Port des Prozessors kann mit dem normalen Systemtakt laufen.

```
-- Prozess für den normalen Speicherzugriff
process (clk)
begin
      if (clk'event and clk = '1') then
            ready <= '0';
            if (br_en = '1') then
                  if (br_we = '1') then
                        RAM(conv_integer(addra)) <= br_di;
                  end if;
                  read_addra <= addra;
                  ready <= '1';
            end if;
      end if;
end process;

-- Prozess für den Lesezugriffes des VGA-Cores
process (clk_50m)
begin
      if (clk_50m'event and clk_50m = '1') then
            read_addrb <= addrb;
      end if;
end process;
```

Die Größe des VGA-Speichers errechnet sich aus der Pixelanzahl multipliziert mit der Farbtiefe. Da der Speicher nur in 2er-Potenzen dimensioniert werden kann, ergibt sich eine Größe von 8192*32Bit. Benötigt werden eigentlich nur 8000*32Bit um eine Auflösung von 320x200x4 zu ermöglichen. Die restlichen 192*32-Bit ergeben 4,8 „blinde“ Zeilen, welche als Zwischenspeicher für noch nicht darzustellende Zeilen oder andere Daten verwendet werden können.

Wird die VGA Schnittstelle nicht verwendet, so kann der gesamte Speicher auch problemlos als normaler Arbeits- oder Programmspeicher verwendet werden.

Eine Tabelle der Adressen der jeweiligen VGA-Zeilen befindet sich im **Anhang 8.3.1.**

3.5 Der ASCII-Interpreter

3.5.1 Die Idee

Das Problem war, dass der Prozessor nur direkt auf die einzelnen Speicherzellen des Grafikspeichers zugreifen konnte. Man hatte also nur einen reinen Grafikmodus zur Verfügung. Möchte man nur einen Text oder Zahlen auf dem Bildschirm darstellen, zum Beispiel um das Ergebnis

einer Berechnung anzuzeigen, musste man nun jeden einzelnen Pixel im Grafikspeicher so setzen, dass sich daraus ein bestimmtes Zeichen ergibt. Dies benötigt jedoch immensen, unnötigen Rechenaufwand sowie einen nicht unbeträchtlichen Speicherbedarf.

Daraus ergab sich die Idee, dafür ein eigenes Hardwaremodul zu entwerfen. Dies soll, wenn man an eine bestimmte Adresse des HEX-Codes eines ASCII-Zeichens sendet, genau die Pixelfolge dieses Zeichen automatisch in den VGA-Speicher schreiben. Bei einer Zeichengröße von 8x8 Pixel lassen sich so 40x25 Zeichen auf dem Bildschirm darstellen. Dieser Modus kann problemlos mit dem Grafikmodus gemischt werden, da die ASCII-Zeichen einfach nur den Speicherinhalt des VGA-RAMs überschreibt.

Dazu werden im wesentlich 2 Elemente benötigt: Eine Tabelle welche den 8-Bit HEX-Code des ASCII-Zeichens in Pixelwerte übersetz und einen Prozess, welcher den Zugriff auf den VGA-Speicher ermöglicht.

3.5.2 ALUT (Ascii LookUp Table)

Zunächst wurde ein neuer Datentyp erstellt (ascii) welcher ein 8x8 Bit großes Array darstellt. In diesem Array kann das 8x8 Pixel Große Zeichen dann abgebildet werden.

Nun haben wir wie bereits in der Color-LookUp-Table eine CASE/WHEN-Funktion erstellt. Als Eingabewert besitzt es den 8-Bit ASCII Code und als Rückgabewert das 8x8-Bit-Array.

```
function alut(input : in std_logic_vector(7 downto 0)) return ascii is
begin
      case (conv_integer(input)) is
            --------- A ----------
            when 65 => return(      ("00011000"),
                                    ("00111100"),
                                    ("01100110"),
                                    ("01111110"),
                                    ("01100110"),
                                    ("01100110"),
                                    ("01100110"),
                                    ("00000000"));
      end case;
end function alut;
```

An diesem Beispiel des Buchstaben “A” kann man nun sehen, dass eine logische „1“ den Vordergrund und eine Logische „0“ den Hintergrund ergibt (zur besseren Darstellung wurden die 1er hier rot eingefärbt). Zu beachten ist hier noch, dass die Zeichen Spiegelverkehrt in die LookUp-Table geschrieben werden müssen.

In dieser Tabelle sind nun alle Großbuchstaben (A-Z) sowie die Zahlen (0-9) und die Zeichen „Punkt“ und „Doppelpunkt“ integriert. Die Tabelle kann jederzeit um eigene Zeichen erweitert werden. So wurden für das Demoprogramm „SCOTCH-RACE“ ein Bomben- sowie ein Kleeblattsymbol integriert.

3.5.3 Zugriff auf den VGA-RAM

Hier besteht wieder das Problem, dass sowohl die CPU als auch die ASCII-Hardware auf den VGA-Speicher Schreibzugriff benötigt. Da hier jedoch kein so exaktes Timing nötig ist wie bei dem Zugriff des VGA-Cores, kann dies problemlos über einen Multiplexer erfolgen, welcher wenn die ASCII-Hardware aktiv ist, den Daten- und Adressbus des VGA-RAMs auf die ASCII-Hardware richtet.

Da ein Pixel der VGA-Schnittstelle aus 4 Bit besteht, muss zunächst mit einer Funktion aus den Nullen und Einsen welche die ALUT liefert, eine Vervierfachung erfolgen. So wird aus jeder 1 die Folge „1111“ und aus jeder 0 die Folge „0000“. Nun können die nun insgesamt 32 Bit der ersten ASCII-Zeichen-Zeile an den VGA-Speicher geschrieben werden.

Dazu muss jedoch auch zuerst die entsprechende Adresse berechnet werden. Soll ein Zeichen an die zweite Stelle in der zweiten ASCII-Zeile geschrieben werden, so betrifft dies je die zweiten Speicherzellen in den VGA-Zeilen 9 bis 16.

Wurde nun die erste Zeile des Zeichens geschrieben, und hat der VGA-RAM das ready-Signal gesendet, so kann mit dem Schreiben der zweiten Zeile begonnen werde. Dieser Vorgang wird nun so oft wiederholt bis alle 8 Zeilen des Zeichens geschrieben wurden.

Eine Auflistung der Adressen jeder ASCII-Position befindet sich im **Anhang 8.3.2.**

3.5.4 Vorder- und Hintergrundfarbe

Als letzter Schritt kam nun noch eine Erweiterung der ASCII-Hardware für eine Bestimmung der Vorder- und Hintergrundfarbe hinzu. Dazu werden einfach die Bits 8-11 des Datenwortes für die Vordergrundfarbe verwendet, die Bits 12-15 stellen die Hintergrundfarbe dar.

Diese Bits werden nun anstelle der „1“ bzw. „0“ aus der ALUT gesetzt. So können die selben 16 Farben wie auch im Grafikmodus verwendet werden.

Insgesamt muss also nun ein ASCII-Zeichen folgendermaßen an den Datenbus der ASCII-Hardware gelegt werden:

```
Datenwort: 0x0000HVAA

H:  HEX-Wert der Hintergrundfarbe
V:  HEX-Wert der Vordergrundfarbe
AA: HEX-Wert des ASCII-Zeichens
```

Die Vorder- und Hintergrundfarbe muss immer gesetzt werden, da dies sonst als 2x „0“, also als „weiß auf weiß“ interpretiert wird.

Im Assembler wurde auch eine Erweiterung des .dc Befehls integriert um Strings eingeben zu können

```
.dc “HALLO SCOTCH” (0xV,0xH)

H:  HEX-Wert der Hintergrundfarbe
V:  HEX-Wert der Vordergrundfarbe
```

4. Debug Schnittstelle

Um das System vernünftig testen zu können wird eine Debug Schnittstelle benötigt. An dieser Schnittstelle werden interne Signale abgefragt, ausgewertet und bei Bedarf verändert. Zur Realisierung dieser Funktionen wird ein angeschlossener PC benötigt, welcher über die RS-232 Schnittstelle angeschlossen wird. Über den Debugger läuft ebenfalls die externe 50 MHz-clock, welche das System und den FPGA Chip speist.

Die Schnittstelle gliedert sich in zwei Module, welche auch unten noch erklärt werden:

Zum einen die DBG-Shell, die zwischen dem HiCoVec und der IO die Signale liest und verarbeitet und dem Debugger der diese Signale dann umwandelt und als Debug-Chain an den Entwicklungs-PC weitergibt, oder Daten von ihm empfängt und für das SCOTCH System aufbereitet.

Durch die DBG-Shell ist der HiCoVec Prozessor praktisch vom ganzen restlichen System getrennt. Seine sämtlichen Signale führen über die Debug-Shell, womit er theoretisch vollständig extern steuerbar ist.

4.1 Die DBG-Shell

Wie bereits im Kapitel 2.3.3 erwähnt, ist die DBG-Shell zwischen dem Prozessor und dem restlichen System angesiedelt; in unserem Fall den I/Os, die im Kapitel 2.4 aufgelistet sind. Sie erlaubt somit die Kontrolle über das gesamte System von außen, per seriellen Anschluss. Im Detail handelt es sich um folgende Funktionen:

- Betreiben des Prozessors im Single Step(Clock) Modus
- Auslesen oder Neubeschreiben aller CPU-Leitungen sowie die Möglichkeit der Emulation einer I/O Umgebung unabhängig vom aktuell angeschlossenem System
- Betreiben des I/O-Systems im Single Step(Clock) Modus, auch unabhängig von der CPU
- Zugriff und Ausführen von Lese- und Schreiboperationen auf das I/O-System, auch unabhängig von der CPU

Daher ist das System in 3 Taktdomänen unterteilt:

- CPU clock
- Shell clock
- I/O clock

Da es keine direkte Verbindung zwischen der CPU und der I/O Domäne gibt, da alle Signale zuerst durch die DBG-Shell müssen, muss man besonders auf die Signale acht geben, die die „I/O / Shell“ und die „Shell / CPU“ Grenze passieren. Daher sind die Kotrollsignale (‚rd‘, ‚wr‘ und ‚ready‘), die in die Shell hinein und aus der Shell heraus führen, jeweils um einen Takt verzögert. Hier eine Übersicht aller Signale:

```
      port(
            clk,                        -- shell clock
            se, sdi: in std_logic;      -- scan enable, shift data in
            sdo: out std_logic;         -- shift data out

            -- CPU signals...
            cpu_reset: out std_logic;
            cpu_rd, cpu_wr: in std_logic;
            cpu_ready: out std_logic;
            cpu_adr: in std_logic_vector (31 downto 0);
            cpu_data: inout std_logic_vector (31 downto 0);
```

```
        -- BUS signals...
        bus_rd, bus_wr: out std_logic;
        bus_ready: in std_logic;
        bus_adr: out std_logic_vector (31 downto 0);
        bus_data: inout std_logic_vector (31 downto 0)
    );
```

Die Signale `clk`, `se` und `sdi` sind Eingangssignale und kommen direkt vom Debugger per seriellen Anschluss. Als Ausgangssignal zur Kommunikation mit dem Debugger dient `sdo`. Hier werden alle benötigten Signale, die in der Debug-Chain festgelegt sind nach außen geshiftet.

Die Signale `cpu_rd` und `cpu_wr` sind Eingangssignale die vom Prozessor kommen. Sie legen die benötigte Art des Zugriffs auf den I/O-Block der CPU fest. Sie werden nach Bearbeiten durch die DBG-Shell als `bus_rd` und `bus_wr` an das I/O-System weitergeleitet. Bei `cpu_adr` handelt es sich um den Adressbus der CPU. Dieser wird bei Lese oder Schreibvorgängen benötigt, um den Ort des Zugriffs im I/O-Block zu deklarieren. Als `bus_adr` wird er nach Bearbeitung an das I/O System weitergeleitet. Der 32-bit breite Bus `cpu_data` dient zur Datenbeförderung in/aus der CPU aus/in den I/O Block, dessen Bus dann `bus_data` heißt. Das aus dem I/O-Block als `bus_ready` deklarierte Signal wird nach Bearbeitung durch die DBG-Shell als `cpu_ready` an den Prozessor weitergeleitet. Sonderstatus genießt das Signal `cpu_reset`, welches dafür zuständig ist, dass der Prozessor unabhängig vom I/O-Block zurückgesetzt werden kann.

4.2 Der Debugger

Der Debugger ist das Modul, das zwischen dem externen Entwicklungs-PC und dem darauf befindlichem VOHDCA auf der einen Seite, und der DBG-Shell auf der anderen Seite liegt. Es verwaltet die Clock Signale und empfängt und sendet die Daten des VOHDCA aufbereitet an die jeweilig entsprechenden Stellen.

Der Debugger selber handelt die einzelnen Signale in einer definierten Debug-Chain, welche auch vom VOHDCA gelesen und geschrieben werden kann.

In seiner Grundform wurde der Debugger von Michael Schäferling für das Viscy Projekt und einem dazugehörigen 16 Bit Prozessor geschrieben. Für unsere Zwecke wurde der Debugger dahingehend verändert, dass er nun einen 32 Bit Prozessor, wie der HiCoVec einer ist, unterstützt. Dazu wurden die Adressen und Daten Vektoren in der Debug Chain erweitert.

Aufbau der Debug-Chain:

```
CaptureIO->CaptureCPU->reset->rd->wr->ready->adr[31..0]->data[31..0]
```

4.3 VOHDCA

VOHDCA (**V**iscy2l **O**ptimized for **H**icovec to **D**ebug, **C**ompute and **A**ssemble), ist ein in Python entwickeltes Konsolenprogramm für Linux. Das Original Viscy2l wurde von unserem Betreuer Herrn Gundolf Kiefer entwickelt und von uns für den Betrieb mit dem HiCoVec angepasst. Um das Programm zu benützen, ist der Python Interpreter zu installieren, vorzugsweise mit einem Paketmanager.

4.3.1 Was macht VOHDCA

VOHDCA wurde entwickelt um den Binärcode den der Compiler produziert auf das SCOTCH-System zu übertragen. Der Binärcode befindet sich in einer Objektdatei, die einen besonderen Aufbau hat. Desweiteren werden verschiedene Debug-Befehle unterstützt. Die Übertragung geschieht mit der RS232 Schnittstelle. Für die Kommunikation mit der RS232 Schnittstelle, wird auf die chipdebug Library zurückgegriffen, da hier alle relevanten Funktionen zum korrekten Transport der Daten realisiert sind. Die Library ist in C geschrieben, kann aber ohne weiteres in Python Programme eingebunden werden.

4.3.2 Unsere Änderungen

Das für unsere Zwecke optimierte Tool kann auch Programme, welche für die Viscy CPU kompiliert wurden, einlesen. Für HiCoVec-Programme haben wir das Sektionenverzeichnis angepasst. Jedoch ist es zurzeit noch nicht möglich mit einem Tool für beide Systeme Programme zu übertragen, dies erfordert noch eine Anpassung, die wir aber in unserem Zeitrahmen nicht mehr realisieren konnten.

4.3.3 Das Einlesen der Objektdatei

Der Aufbau der Objektdatei wird unter Punkt 5.6 genau beschrieben. Ich will hier im Auszug Quellcode zeigen und diesen erklären, somit wird schnell klar, wie die Datei ausgelesen und für das Übertragen vorbereitet wird.

Die dafür verantwortliche Funktion ist: *def readObjAndSrcFile (objFileName)*: hier im Ausschnitt

```
elif secType == 0x11: # 'Code 32 BIT HICOVEC'
      adr = objData[secStart]
      adr = adr << 16
      adr += objData[secStart+1]
      n = secEnd - (secStart +1)
      print "-> Code: 0x%04x .. 0x%04x" % (adr, adr + n - 1)

      z = 0
      for i in range(1,n,2):
      outMemory[adr + z] = objData[secStart + 1 + i]
      outMemory[adr + z] = (outMemory[adr + z]) << 16
      outMemory[adr + z] += objData[secStart + 2 + i]
      z = z+1
```

Der Binärcode wird zuerst, so wie er ist, in das objData[] array geschrieben. Danach wird die Startadresse der ersten Codesektion ermittelt, damit diese dann an den richtigen Index des outMemory[] arrays geschrieben wird. Das outMemory[] array stellt das Speicherabbild dar. Wichtig hier zu erwähnen ist, dass in objData[] 16 Bit weise (also hinter jedem Index stehen 16 Bits) eingelesen wird. Also müssen wir für unsere 32 Bit Befehle immer 2 Zeilen einlesen. D.h. zuerst die erste Zeile einlesen, dann die Bits 16-mal nach links schieben, dann die 2te Zeile.

4.3.4 Das Arbeiten mit VOHDCA

Vorab ist zu überprüfen, ob die richtige serielle Schnittstelle eingestellt wurde. Dies wird gleich ganz am Anfang vom Code eingestellt.

```
optSerialDevice="/dev/ttyS0"
```

Die Klasse MyCmd stellt eine sehr komfortable Möglichkeit dar, die Benutzereingaben abzufragen. Wird in einer Funktion der Klasse MyCmd ein do_ vor den Funktionsnamen gestellt, so wird die Funktion direkt aufgerufen, wenn der Name der Funktion in der Konsole eingegeben wird. Außerdem ist auch die von Linux gewohnte Autovervollständigung via TAB Taste implementiert.

Der VOHDCA Prompt:

```
VOHDCA - Viscy2l Optimized for Hicovec to Debug, Compute and Assemble - v.
1.0

(C) 2008 - 2009
Jakob Golus
David Lucinkiewicz
Andreas Mueller
Florian Richter
Sebastian Rigorth
Soeren Ruehm
University of Applied Sciences Augsburg

Original viscy2l: (c) 2007-2008 Gundolf Kiefer, University of Applied
Sciences Augsburg

(VOHDCA)
```

Die Befehle:

```
dump              - dump the memory contents (VHDL syntax)
exit              - exit the tool
get <adr> [<n>]   - display <n> memory words starting from <adr>
open <file name>  - open an object file
reset             - reset the CPU
run_fullspeed     - run system at full speed
run_monitored     - run system slowly and print information on memory
accesses
set <adr> <val>   - write <val> to memory adress <adr>
step              - run system for one clock cycle and display CPU pins
upload            - write the loaded object file to the board memory
```

Das Laden eines Testfiles: (open test.o)

```
    0. 0ef0
    1. 0000
    2. 0800
    3. 1200
    4. 0000
    5. 2b00
    6. 0000
    7. 3000
    8. 1300
    9. 0000
 .....
  164. 7065
  165. 0000
  166. 00a0
  167. 0001
  168. 0028
  169. 0000
Sections = 8
Type = 18, Start = 43, End = 48  Type = 19, Start = 49, End = 105  -> Line
info: 28 source lines
Type = 5, Start = 106, End = 136  Type = 6, Start = 137, End = 139  Type =
23, Start = 140, End = 147  Type = 17, Start = 148, End = 151  -> Code:
0x0050 .. 0x0051
Type = 17, Start = 152, End = 157  -> Code: 0x0100 .. 0x0103
Type = 17, Start = 158, End = 169  -> Code: 0x0000 .. 0x0009
```

Die Ausgabe von VOHDCA:

Das Tool sucht im Sektionenverzeichnis nach der Code Sektion, springt dann dort hin und liest alle Daten in ein Speicherabbildarray ein. Wichtig bei der Ausgabe sind die letzen beiden Zeilen. Hier wird die Code-Sektion eingelesen. Das erzeugt Speicherabbildarray kann dann direkt auf das System übertragen werden.

Typ	gibt die Art der Sektion an 17 (dec) = 0x11 (hex) = Code-Sektion.
Start , Ende	ist der Index der durchnummerierten Debug-Ausgabe
Code	gibt die tatsächlichen Adressen im Speicher an.

5. Der Assembler

5.1 Einleitung

5.1.1 Einleitende Gedanken

Der SCOTCH Assembler ist eine abgeleitete und überarbeitet Version des Compilers für die VISCY CPU, daher empfehlen wir das Benutzerhandbuch SEPP, sowie die Diplomarbeit von Harald Manske, der für seine HICOVEC CPU auch einen Assembler entwickelt hat.

Der komplette Befehlssatz der CPU ist in der Diplomarbeit beschrieben, sowie die Grammatik u. Syntax seines in Python entwickelten Compilers. Die Syntax des Python Compilers unterscheidet sich nicht wesentlich von der des SCOTCH Compilers. Wir werden daher besonders auf die veränderte und neu definierte Grammatik eingehen.

Wir haben uns dazu entschieden den VISCY Compiler als Grundlage für unsere Entwicklungen zu nehmen, da dieser mit den professionellen Tools Flex und Bison erstellt wurde. Im Laufe unserer Entwicklung wurde der Compiler jedoch völlig überarbeitet und abgeändert, so dass eine Neuentwicklung nicht zeitaufwändiger gewesen wäre.

Das Projekt stellte uns vor eine große Herausforderung da niemand im Software Team Erfahrung mit dem Thema Compilerbau, noch in der Programmiersprache Python hatte.

Die vorhandenen Funktionen im Quellcode waren sehr dürftig oder gar nicht beschrieben, was uns das Einarbeiten sehr erschwerte. Deswegen ist es uns ein großes Bedürfnis, wichtige und komplizierte Funktionen, hier in der Dokumentation zu beschreiben. Damit das Projekt schnell von anderen Entwicklern erweitert und vor allem verstanden werden kann.

5.1.2 Empfohlene Literatur

Wenn man sich bisher noch nie mit dem Thema Compilerbau auseinandergesetzt hat können wir folgendes Buch wärmstens empfehlen:

Schmitt, F.J.
Praxis des Compilerbaus
C. Hansen, 1992

Unter folgendem link befindet sich ein Tutorial das bestens für den Einstieg in Flex und Bison geeignet ist. Es war auch unsere erste Anlaufstelle zu diesem Thema.

http://www4.tu-ilmenau.de/ate/TET/nlnet/flex_bison.html

5.2 Entwicklungsumgebung

Als Entwicklungsumgebung für den SCOTCH - Assembler wurde auf das Betriebssystem Linux zurückgegriffen. Da es eine Vielzahl von Bibliotheken bereitstellt, war es möglich professionelle und im Assembler- bzw. Compilerbau übliche Werkzeuge, wie Flex und Bison (yacc) zu verwenden. Um für spätere Anwendungen eventuelle Portabilität auf andere Betriebssystem zu bieten, haben wir uns für die Programmiersprache C entschieden. Ein weiterer Grund für dieses Vorgehen war, dass wir bereits einen Assembler, auch in der Programmiersprache C, einer anderen Projektgruppe als

Vorlage hatten. Als Compiler kam der GNU C-Compiler (GCC) zum Einsatz, somit konnte der Assembler durch eine einfache Makefile-Prozedur kompiliert, bzw getestet werden. Bei schwerwiegenderen Problemen wurde der GNU Debugger zu Rate gezogen. Der Einsatz dieses Assemlbertools ist wegen oben genannten Gründen auch für das Linux-Betriebssystem vorgesehen.

5.2.1 Hardwarevoraussetzungen

Als Hardwarevoraussetzung, um den Assembler betreiben bzw erweitern zu können, sollte ein moderner Computer mit der x86- oder x64-Architektur vorliegen. Es gibt für beide Architekturen bereits vorgefertigte Debian-Packete, daher sollte man vorher herausfinden welches von beiden Systemen man besitzt.

5.2.2 Benötigte Pakete

Wenn man den Assembler nur verwenden möchte, also nicht weiterentwickeln, und gleichzeitig Debian, Ubuntu oder ein ähnliches Debian-Derivat besitzt, kann eins von beiden vorgefertigten Debian-Packeten verwendet werden, ohne irgendwelchen zusätzlichen Pakete nachinstallieren zu müssen. Allerdings muss für die Präprozessor-Funktion der GCC installiert sein. Möchte man allerdings am bestehenden Assembler Änderungen vornehmen, so müssen folgende Pakete bzw. Tools/Bibliotheken nachinstalliert werden. Die Namen richten sich nach der Debianpaketbezeichnung und können daher bei anderen Distributionen unterschiedlich lauten, sollten aber auch in diesen ohne Probleme verfügbar sein:

Paketname	**Bezeichnung**
Flex	Scannergeneratortool
Parser	Parsergeneratortool
Automake	Zur Unterstützung von makefiles
Gcc-4.2-base (oder neuer)	GNU Compiler Collection (Basispaket)
Gcc-4.2 (oder neuer)	C-Compiler
Libc6 (oder libc6-amd64, je nach CPU-Architektur)	Standard C-Biblothek

5.2.3 Installationshinweise

Es gibt zwei Möglichkeiten den Assembler zu installieren. Die erste Möglichkeit besteht darin, sich alle oben genannten Pakete zu besorgen um den Assembler dann direkt neu zu übersetzen. Mit folgendem Befehl wird der Assembler in eine ausführbare Datei namens *scotchas* übersetzt.

```
make
```

Mit folgendem Aufruf wird der Assembler kompiliert und in das Verzeichnis /usr/bin kopiert, somit ist der Assembler in jeden Verzeichnis verfügbar.

```
make install
```

Die andere Möglichkeit den Assembler zu installieren, besteht darin das vorgefertigte Debian-Paket mit dem Paketmanager zu installieren, vorausgesetzt man besitzt Debian oder ein Debian-Derivat.

Installation der x86-Version:

```
dpkg -i scotchas_1.0.0_i386.deb
```

Installation der x64-Version:

```
dpkg -i scotchas_1.0.0_amd64.deb
```

5.3 Die Verwendung des Assemblers

Ist der Assembler installiert kann er ganz einfach mit folgendem Befehl aufgerufen werden:

```
scotchas [options]... sourcefile
```

5.3.1 Programmaufruf und Optionen

Ruft man den Assembler ohne Parameter in der Konsole auf, so erhält man folgende Hilfestellung:

```
This Assembler belongs to the Scotch Project 2009 of the FH augsburg and is
adapted to the HiCoVec processor.

programm call:   scotchas [options]... sourcefile

options:
-p
--preprocessor  switches the preprocessor for #define-constants on
-n
--nosection     minimal objectfile
-i
--info          minimal objectfile + info section
-c
--comments      minimal objectfile + comment section
-m
--labels        minimal objectfile + label section
-l
--lines         minimal objectfile + line section
-h
--help  print options
-s
--start         ignore if .start is missed
-v
--verbose       verbose mode
-o objectfilename
--outfile objectfilename        defines the name of the objectfile
```

Alle Optionen, bis auf den Quelldateinamen, sind optional und müssen dem Assembler daher nicht mitgegeben werden. Wie schon erwähnt muss der Quelldateiname beim Aufruf mitberücksichtig werden. Wird kein Zieldateiname bestimmt, so landet der assemblierte Maschinencode in der Datei

asm.o. Für den normalen Arbeitsbetrieb empfehlen wir doch folgenden Aufruf, um nicht versehentlich eine vorher übersetzte Datei zu bearbeiten:

```
scotchas –o output.o source.asm
```

5.3.2 Der Präprozessor

Um mehr Bedienerfreundlichkeit und Komfort zu bieten, haben wir uns entschieden einen Präprozessor mit in den Assembler einzubauen. Mit diesem ist es möglich #define-Konstanten (bekannt aus der Sprache C) zu definieren, um diese im jeweiligen Programm zu verwenden. Da bereits sehr gute Implementierungen von Präprozessoren vorhanden sind, haben wir auch auf eine externe Lösung zurückgegriffen. Wie bereits in 5.2.2 erwähnt, verwenden wir den GCC als Präprozessor, somit muss das entsprechende Paket für den GCC auf dem Rechner installiert sein, um diesen verwenden zu können. Befindet sich der GCC nicht im /usr/bin-Verzeichnis, so muss die Datei config.h entsprechend angepasst werden und der SCOTCHas neu übersetzt werden:

Auszug aus config.h:

```
#define GCC "/usr/bin/gcc "
```

5.3.3 Beispiel zur Verwendung des Präprozessors

Testfile.asm:

```
#define k 8
#define n 9

        .org 0x50

        LD X, [0 + k]
        LD Y, [0 + n]

            HALT
            .end
```

Aufruf des Assemblers mit Präprozessor:

```
scotchas –p –o output.o Testfile.asm
```

5.4 Syntax

5.4.1 Befehlssatz

Der Befehlssatz des SCOTCHas Assemblers wurde, bis auf ein paar Modifikationen, weitestgehend aus der Diplomarbeit von Harald Manske übernommen. Wir haben bei dem Assembler auch darauf geachtet, dass er im Hinblick auf Erweiterbarkeit einfach bleibt. Es wurde zudem darauf geachtet, dass die Befehle auch leicht verständlich bleiben, um eine intuitive Entwicklungsumgebung zu gewährleisten.

Die Datenregister:

AXY ➔ Es können entweder das Datenregister A, X oder Y ausgewählt werden.

AXY0 ➔ Wie oben, nur dass zudem noch als Quelle oder Ziel die "0" angegeben werden kann. Das hat zur Folge, dass das Ergebnis in kein Register geschrieben wird oder die "0" als Operand benutzt wird.

AXYi ➔ Hier kann statt der "0" ein Direktwert als Operand gewählt werden.

Die Vektorregister:

Vregl ➔ R0…R15

Vregh ➔ R<0>…R<N>: konfigurierbare Anzahl an Vektorregistern, diese können sowohl als Quell wie auch als Zielregister dienen.

Die Wortbreite:

Für Vektoroperationen ist es erforderlich die Wortbreite festzulegen.

Breitebw ➔ B(8Bit) oder W(16Bit)

Breitevoll ➔ B(8Bit), W(16Bit), DW (32Bit), QW(64Bit)

Nachfolgend eine Übersicht über den Befehlssatz:

Ladeanweisung

LD AXY0, [AXY0 + AXYi]

Erklärung:

Die LD-Anweisung holt ein Datum aus dem Speicher und legt es in einem der drei Datenregister ab.

Beispiel:

```
LD X, [0 + 5] ➔ lade Register X von Adresse 5
LD Y, [0 + A] ➔ lade Register Y von Adresse A
LD A, [X + Y] ➔ lade Register A von Adresse X + Y
```

Speicheranweisung

ST [AXY0 + AXYi], A

Erklärung:

Die ST-Anweisung schreibt den Inhalt eines der drei Register in den Speicher.

Beispiel:

```
ST A, [0 + 5] ➔ speichere Register A an Adresse 5
ST A, [0 + Y] ➔ speichere Register A an Adresse Y
ST A, [X + Y] ➔ speichere Register A an Adresse X + Y
```

Addition

ADD AXY0 , AXY0 , AXYi

Erklärung:

Dieser Befehl wird dazu verwendet, eine Additionsoperation durchzuführen.

Beispiel:

```
ADD A, B, 0 + 5 ➔ A = B + 5
```

Addition mit Carry

ADC AXY0 , AXY0 , AXYi

Erklärung:

Der Befehl dient dazu eine Additionsoperation mit einem Übertrag durchzuführen.

Beispiel:

```
ADC A, X, Y
```

Inkrementierung

INC AXY0, AXY0

Erklärung:

Mit INC kann ein Register inkrementiert werden.

Beispiel:

```
INC Y, Y ➔ entspricht i++ in C
```

Subtraktion

SUB AXY0, AXY0, AXYi

Erklärung:

Dieser Befehl wird dazu verwendet, eine Subtraktionsoperation durchzuführen und das Ergebnis in ein Datenregister abzulegen. Zudem können mit diesem Befehl auch zwei Register verglichen werden.

Beispiel:

```
SUB X, Y, 5  → X = Y – 5
```

Subtraktion mit Carry

SBC AXY0, AXY0, AXY

Erklärung:

Dieser Befehl wird verwendet, um eine Subtraktionsoperation mit einem Übertrag durchzuführen.

Beispiel:

```
SBC A, Y, X
```

Dekrementierung

DEC AXY0, AXY0

Erklärung:

Mit DEC kann ein Register dekrementiert werden.

Beispiel:

```
DEC X, X → entspricht i-- in C
```

UND-Verknüpfung

AND AXY0, AXY0, AXYi

Erklärung:

Mit dem AND Befehl können 2 Register miteinander verknüpft werden.

Beispiel:

```
AND A, A, Y
```

ODER-Verknüpfung

OR AXY0, AXY0, AXYi

Erklärung:

Der Befehl dient dazu, das logische ODER auf zwei Register anzuwenden. Zusätzlich können mit dem Befehl auch Register kopiert oder mit einem Direktwert belegt werden.

Beispiel:

```
OR Y, A, 0 ➔ kopiert Register A in Register Y
OR X, 0, $1234 ➔ belegt Register X mit 0x1234
```

Exklusiv-Oder-Verknüpfung

XOR AXY0, AXY0, AXYi

Erklärung:

XOR Verknüpfung zweier Register.

Beispiel:

```
XOR A, A, 0 + 5
```

Schiebeoperation nach Links

LSL AXY0, AXY0

Erklärung:

Dieser Befehl dient zum Verschieben eines Registers nach links.

Beispiel:

```
LSL X, X ➔ Schiebe Register X nach links.
```

Schiebeoperation nach Rechts

LSR AXY0, AXY0

Erklärung:

Dieser Befehl dient zum Verschieben eines Registers nach rechts.

Beispiel:

```
LSR X, X ➔ Schiebe Register X nach rechts.
```

Schiebeoperation nach Links (Carry einfügen)

ROL AXY0, AXY0

Erklärung:

Befehl um das Register nach links zu verschieben und das Carry-Flag rechts einzufügen.

Beispiel:

```
ROL A,X
```

Schiebeoperation nach Rechts (Carry einfügen)

ROR AXY0, AXY0

Erklärung:

Befehl um das Register nach rechts zu verschieben und das Carry-Flag links einzufügen.

Beispiel:

```
ROR A, X
```

Multiplikation (optional)

MUL AXY0, AXY0, AXYi

Erklärung:

Befehl um eine Multiplikationsoperation durchzuführen. Muss Hardwareseitig aktiviert sein.

Beispiel:

```
MUL A, X, 0+5 ➔ A = X * 5
```

Unbedingter Sprungbefehl

JMP [AXY0 + AXYi]

Erklärung:

Befehl um an eine gespeicherte Adresse im Speicher zu springen.

Beispiel:

```
JMP [0 + LABEL] ➔ Springe zu LABEL
```

„Jump-And-Link“ (für Unterprogramme)

JAL AXY0, [AXY0 + AXYi]

Erklärung:

Durch diesen Sprungbefehl wird beim Ausführen der Befehlszähler hinterlegt.

Beispiel:

```
JAL A, [0 + X] ➔ Springe zu Adresse in X und hinterlege Befehlszähler in A
```

Springe wenn Zero-Flag gesetzt

JZ [AXY0 + AXYi]

Erklärung:

Der Befehl ermöglicht es, nur dann einen Sprung im Programm zu machen, wenn auch das Zero-Flag vorher gesetzt wurde.

Beispiel:

```
JZ [0 + A] ➔ Springe zu Adress in A wenn Zero-Flag = 1.
```

Springe wenn Zero-Flag nicht gesetzt

JNZ [AXY0 + AXYi]

Erklärung:

Durch den Befehl JNZ wird nur ein Sprung gemacht, wenn das Zero-Flag nicht gesetzt wurde

Beispiel:

```
JNZ [X + LABEL] ➔ Springe zur Adresse X+LABEL wenn Zero-Flag = 0.
```

Springe wenn Carry-Flag gesetzt

JC [AXY0 + AXYi]

Erklärung:

Der Sprungbefehl wird nur ausgeführt, wenn das Carry-Flag vorher gesetzt wurde.

Beispiel:

```
JC [A + 5] ➔ Springe zu Adresse in A+5 wenn Carry-Flag = 1.
```

Springe wenn Carry-Flag nicht gesetzt

JNC [AXY0 + AXYi]

Erklärung:

Der Sprungbefehl wird nur ausgeführt, wenn das Carry-Flag nicht gesetzt wurde.

Beispiel:

```
JC [A + Y] ➔ Springe zu Adresse A+Y wenn das Carry-Flag = 0.
```

Lösche Zero-Flag

CLZ

Erklärung:

Durch CLZ wird das Zero-Flag gelöscht, falls eines gesetzt wurde.

Beispiel:

```
CLZ
```

Setze Zero-Flag

SEZ

Erklärung:

Das Zero-Flag wird gesetzt.

Beispiel:

```
SEZ
```

Lösche Carry-Flag

CLC

Erklärung:

Das Carry-Flag wird gelöscht.

Beispiel:

```
CLC
```

Setzte Carry-Flag

SEC

Erklärung:

Carry-Flag wird gesetzt.

Beispiel:

```
SEC
```

Prozessor anhalten

HALT

Erklärung:

Durch den Befehl wird der Prozessor angehalten.

Beispiel:

```
HALT
```

Keine Skalar-Operation ausführen

NOP

Erklärung:

Es wird keine Skalar-Operation ausgeführt.

Beispiel:

```
NOP
```

Keine Vektor-Operation durchführen

VNOP

Erklärung:

Es wird kein Vektor-Befehl ausgeführt.

Beispiel:

```
VNOP
```

Kopiere von Skalar- in Vektoreinheit oder umgekehrt

MOV vregl (AXY), AXY0 ➔ Kopiere aus Skalar- in Vektoreinheit

MOV AXY0, vregl (AXY) ➔ Kopiere aus Vektor- in Skalareinheit

Erklärung:

Durch den MOV-Befehl können Daten zwischen der Skalar- und der Vektoreinheit transferiert werden. Dies kann auch umgekehrt erfolgen. Dazu muss ein Wort aus einem Skalar-/Vektorregister selektiert werden. Es wird jedoch bei der Richtung des Transfers unterschieden, ob das Wort neu geschrieben oder übertragen wird. Die Auswahl des zu übertragenden Wortes wird durch die Skalareinheit bestimmt, der Index muss sich jedoch in einem der drei Datenregister befinden.

Beispiel:

```
MOV R0(A), 0      ➔ Y nach Wort A in R0 kopieren
MOV 0, R1(A)      ➔ Wort A in R1 nach A kopieren
```

k-maliges kopieren in Vektorregister

MOVA vregl, AXY0

Erklärung:

Der Befehl ermöglicht es in alle Wörter des Vektorregisters das skalare Quellregister zu kopieren.

Beispiel:

```
MOVA R3, X ➔ X in jedes Wort von R3 kopieren
```

Vektor-Ladeanweisung

VLD vregl, [AXY0 + AXY]

Erklärung:

Wird dazu benutzt Vektorregister ab einer bestimmten Adresse zu laden.

Beispiel:

```
VLD R0, [A+X] ➔ Register R0 mit Inhalt ab Adresse A+X laden
```

Vektor-Speicheranweisung

VST [AXY0 + AXY], vregl

Erklärung:

Befehl dient dazu Vektorregister in voller Breite ab einer bestimmten Adresse abzuspeichern.

Beispiel:

```
VST [X + Y], R1 ➔ Register R1 ab Adresse X+Y speichern
```

Kopieren von Vektorregistern

VMOV vregl, vregl

VMOV vregh, vregl

VMOV vregl, vregh

Erklärung:

Dieser Befehl ermöglicht es, nicht direkt adressierbare Vektorregister, auf direkt adressierbare Vektorregister und umgekehrt zu kopieren.

Beispiel:

```
VMOV R1, R0    ➔ R0 nach R1 kopieren
VMOV R1, R<42> ➔ R42 nach R1 kopieren
VMOV R<42>, R0    ➔ R0 nach R42 kopieren
```

Linksverschiebung aller Wörter des Vektorregisters

VMOL vregl, vregl

Erklärung:

Der Befehl dient dazu ein Wort um eine um eins verschobene Stelle ins Zielregister zu schreiben und nicht an die gleiche Stelle wie es im Quellregister stand.

Beispiel:

```
VMOL R0, R0 ➔ R0 nach links verschieben
```

Rechtsverschiebung aller Wörter des Vektorregisters

VMOR vregl, vregl

Erklärung:

Der Befehl dient dazu ein Wort um eine um eins verschobene Stelle ins Zielregister zu schreiben und nicht an die gleiche Stelle wie es im Quellregister stand.

Beispiel:

```
VMOL R1, R0 ➔ R0 nach rechts verschieben und in R1 abspeichern
```

Vektor-Addition

VADD.breitevoll vregl, vregl, vregl

Erklärung:

Der Befehl ermöglicht es Vektorregister zu mit einer bestimmten Wortbreite zu addieren.

Beispiel:

```
VADD.DW R2, R1, R0 ➔ Das Vektorregister R1 und R2 werden addiert und in
das Register R3 gespeichert bei einer Wortlänge von  32 bit.
```

Vektor-Subtraktion

VSUB.breitevoll vregl, vregl, vregl

Erklärung:

Der Befehl ermöglicht es Vektorregister zu mit einer bestimmten Wortbreite zu subtrahieren.

Beispiel:

```
VSUB.DW R2, R1, R0 ➔ Die Vektorregister R1 und R2 werden subtrahiert und
in Register R3 gespeichert, mit einer  Wortlänge von 32 bit.
```

Vektor-Und-Verknüpfung

VAND.breitevoll vregl, vregl, vregl

Erklärung:

Mit dem Befehl werden die Vektorregister UND-Verknüpft.

Beispiel:

```
VAND.W R2, R3, R4 ➔ UND-Verknüpfung der Vektorregister R3 und R4,
Wortlänge hat dabei keine Auswirkungen.
```

Vektor-Oder-Verknüpfung

VOR.breitevoll vregl, vregl, vregl

Erklärung:

OR-Verknüpfung der Vektorregister.

Beispiel:

```
VOR.DW R1, R0, R3 ➔ OR-Verknüpfung der Vektorregister
```

Vektor-Exklusiv-Oder-Verknüpfung

VXOR.breitevoll vregl, vregl, vregl

Erklärung:

Exklusiv-Oder-Verknüpfung der Vektorregister

Beispiel:

```
VXOR.W R2, R1, R0 ➔ XOR-Verknüpfung der Vektorregister
```

Vektor-Schiebeoperation links

VLSL.breitevoll vregl, vregl

Erklärung:

Linksverschiebung des Vektorregisters.

Beispiel:

```
VLSL.B R1, R1 ➔ schiebe R1 nach links mit einer Wortlänge von 8 bit.
```

Vektor-Schiebeoperation rechts

VLSR.breitevoll vregl, vregl

Erklärung:

Rechtsverschiebung des Vektorregisters.

Beispiel:

```
VLSR.B R1, R1 ➔ schiebe R1 nach rechts mit einer Wortlänge von 8 bit.
```

Vektor-Multiplikation (optional)

VMUL.breitebw vregl, vregl, vregl

Erklärung:

Multiplikation von Vektorregistern. Muss Hardwareseitig aktiviert sein.

Beispiel:

```
VMUL.W R2, R1, R0 ➔ Multipliziere R1 und R0 und speichere in R2.
```

Mischen von Vektorregistern

VSHUF vregl, vregl, vregl, perm

Erklärung:

Der Shuffle Befehl erlaubt es Daten in den Vektorregistern zu mischen.

Beispiel:

```
VSHUF R0, R3, R2, 0b00110011001100
```

5.4.2 Direktiven

Direktiven sind Anweisungen im Assemblercode die nicht zum Programmcode gehören, sondern spezifische Anweisungen für den Präprozessor sind.

Bei dem SCOTCHas Assembler ist es notwendig vor die Direktivennamen einen Punkt zu setzen, wie es bei den nachfolgenden Direktiven in den Beispielen der Fall ist.

ORG (ORIGIN):

.org [Adresse]

Erklärung:

Mit der Ort-Direktive lässt sich der Assembler anweisen Befehle oder Werte ab einer bestimmten Adresse im Speicher abzulegen. Durch ORG können auch Markierungen für einen Sprungbefehl gesetzt werden.

Beispiel:

```
.org $70
.org 0x0070
```

DC (Define Constant)

.dc Wert (HEX, DEC, BIN, STRING)

Erklärung:

Durch den DC-Befehl können Werte direkt im Speicher abgelegt werden. Dies ist notwenig wenn ein Wert zu groß für einen Direktwerk in einem Befehlswort ist. Werte können entweder als hexadezimale und dezimale Zahlen oder als String angegeben werden. Die Konstante hat die maximale Grösse eines 32 Bit unsigned Integer Wertes und die Stringlänge beträgt maximal 20 Zeichen..

Bei einem String kann man noch die Hintergrund- und Schriftfarbe für die VGA-Ausgabe angegeben werden.

```
.dc “STRING“(0xV,0xH)
V: Hexwert der Vordergrundfarbe
H: Hexwert der Hintergrundfarbe
```

Wenn man mit DC einen String definiert wird der ASCII-Wert jedes in den unteren 8-Bit des 32-Bit Wortes abgelegt. Wenn zusätzlich noch die Hintergrund- und die Schriftfarbe des Strings definiert werden so sind deren Farbwerte dahinter abgelegt, also 4-Bit für die Vordergrundfarbe und 4-Bit für die Hintergrundfarbe.

Beispiel:

```
BSP.1: .dc 7                  → dezimal
BSP.2: .dc 0x007              → hexadezimal
BSP.3: .dc “Beispiel 3“       → string
BSP.4: .dc “Beispiel 4“(0xF,0x5)    → string mit Schrift- und
Hintergrundfarbe
```

EQU (Equals)

.equ Name Wert

Erklärung

Die Direktive EQU kann dazu verwendet werden eine Konstante zu definieren.

Beispiel:

```
.equ k 7
```

START

.start

Erklärung:

Bei START beginnt der Assembler seine Arbeit.

Beispiel:

```
.start
```

END

.end

Erklärung:

Bei END beendet der Assembler die Arbeit, alles Nachfolgende wird ignoriert.

Beispiel:

```
.end
```

5.4.3 Parallele Befehle

Da der Hicovec gleichzeitig einen Befehl von der Skalareinheit und von der Verktoreinheit ausführen kann wurde dieses Feature auch im Assembler integriert. Zur parallelen Ausführung zweier Befehle wird der #-Operator verwendet, jedoch muss darauf geachtet werden, dass keine Direktwerte in den zu parallelisierenden Befehlen vorkommen. Der Assembler würde diesen Umstand bemerken und den Assembliervorgang abbrechen.

Liste der parallelisierbaren Befehle:

Skalarbefehl	Vektorbefehl
ADD	VMOV
ADC	VMOR
SUB	VADD
SBC	VAND
INC	VXOR
AND	VLSL
XOR	VSUB
LSL	VOR
ROL	VMUL
DEC	VLSR
OR	
MUL	
LSR	
ROR	

Diese Befehle können beliebig mit dem #-Operator kombiniert werden, jedoch muss, wie oben schon erwähnt, darauf geachtet werden, dass bei parallelen Befehlen keine Direktwerte auftreten dürfen

Beispiel:

```
ADD A,X,Y # VADD R0,R1,R2 ; paralleler Befehl von ADD und VADD
```

5.4.4 Expressions

Durch die Verwendung von Direktwerte wurde zusätzlich noch die Expressionsfunktionalität hinzugefügt. Diese ermöglicht es sich beispielsweise einfache Rechenoperationen vom Assembler ausrechnen zu lassen. Jeder Direktwert kann daher durch eine Expression ersetzt werden.

Mögliche Expressions-Operatoren:

Operator	Beschreibung
+	Addition
-	Subtraktion
*	Multiplikation
/	Division
&	UND
|	ODER
<<	Bitshift Links
>>	Bitshift Rechts
^	Negation
()	Klammern

Beispiel zur Verwendung einer Expression:

```
ADD A,X,5*(20-5) ;Klammer hat vorrang
SUB A,X,12<<1 ; Schiebe die Zahl 12 um eins nach rechts, das ergibt 24
```

5.4.5 Label

Label sind für den Programmierer Hilfen, um häufig benutzte Konstanten durch einen einprägsamen Namen zu ersetzten und aufrufen zu können, oder auch um Sprungmarken für die Jump-Befehle zu setzten.

Beispiele:

```
Label1: .dc "Label kann durch Label1 aufgerufen werden".
Label2:     VST [0 + X], R10  Label2
            SUB X, X, 0x10
            SUB Y, Y, 0x1
            JNZ [0 + Label2] Label2 wird aufgerufen
```

5.4.6 Kommentare

Kommentare in diesem Assembler werden mit dem „;“ eingeleitet und gelten bis zum Zeilenende.

Beispiel:

```
Label1: .dc "Ein Text" ; Kommentar ist gültig bis zum Zeilenende
```

5.5 Der interne Aufbau des Assemblers

5.5.1 FLEX und BISON

Diese zwei Werkzeuge sind notwendig um einen Assembler-Code in Maschinencode zu übersetzen, der für den Prozessor verständlich ist. FLEX und BISON sind zwei Quellcode-Generatoren für den Compilerbau. FLEX ist dabei für die lexikalische Analyse des Quellcodes zuständig. Lexikalische Analyse bedeutet, dass die im Quellcode vorkommenden Eingaben zerlegt werden und in so genannte Tokens umgeformt werden. Die Zerlegung der Eingabe im Quellcode geschieht nach einer vorher festgelegten Grammatik. Diese Tokens werden dann an das Programm BISON weitergegeben das die syntaktische Analyse durchführt. Syntaxanalyse bedeutet, dass die von FLEX übergegebenen Tokens entsprechend der Struktur ihrer Eingabesprache hierarchisch zu einem Baum zusammengefügt werden. Danach erfolgt dann noch die semantische Analyse bei der überprüft wird, ob die Teile des Programms auch von Ihrer Bedeutung her zusammenpassen.

5.5.2 Die Dateistruktur des Assemblers

commit.h: In dieser Headerdatei werden die Datentypen die in den Dateien Scanner.l und Parser.y benutzt werden in den einzelnen Strukturen definiert.

config.h: Diese Datei beinhaltet Konstanten die oft in Funktionen benötigt werden.

- Magic_Number: Version des Compilers
- Debug: ob der Debug Modus standardmäßig an- oder ausgeschaltet sein soll
- GCC : der direkte Pfad zum Compiler
- GCC_OPTIONS: definiert die Optionen die beim Kompilieren der Datei möglich sind

- DefaultOutFile: Standardausgabename der der kompilierten Datei
- DESTINATION_FILE_END: Art des Dateityps der ausgegebenen Datei
- SOURCE_FILE_END : Art des Dateityps der Quelldatei
- INFO & HELPTEXT: Zusatzinformationen die in die Infosection geschrieben werden

createFile.c: Hier sind alle Funktionen beinhaltet die zum Schreiben der Ausgabedatei "asm.o" benötigt werden. In den verschiedenen Funktionen werden die Anzahl und Länge der Sektionen und die einzelnen Sektionen an sich geschrieben. Im Folgenden ist eine Auflistung der einzelnen Sektionen die später in Kapitel 5.6.1 genauer behandelt werden:

- header
- register-section
- info-section
- comment-section
- label-section
- line-section
- code-section

debug.c: In der Datei sind alle Funktionen aufgeführt, die für das Debuggen nötige Informationen auf den Bildschirm ausgeben.

error.c: Die Funktionen überprüfen die Assemblerdatei auf Fehler. Die Funktionen überprüfen die Datei darauf, ob die ".start" Direktive gesetzt wurde oder ob die Codesections auf gleichen Speicheradressen abgelegt wurden.

errors_and_warnings.h: Deklaration der verschiedenen Fehler und Warnungen.

errOut.c: Hier werden die Fehler- und Warnmeldungen ausgegeben, die in der zugehörigen errors_and_warnings.h - Datei deklariert sind.

expression.c: In dieser Funktion werden die Expressions die beim Aufruf verwendet werden berechnet.

functions.h: In der Headerdatei werden die Funktionen der einzelnen Dateien deklariert die zum Parsen und Erstellen der Ausgabedatei benötigt werden. Dazu gehören die Funktionen aus den Dateien:

- lists.c
- wertReturn.c
- debug.c
- expression.c
- writeInFile.c
- createFile.c
- makeObjectFile.c
- error.c
- parseFunctions.c

lists.c: Enthält Funktionen zur Erzeugung der verketteten Listen für den Code und Labels

makeObjectFile.c: Die Funktionen in dieser Datei sind dafür zuständig den Inhalt in die Objektdatei zu schreiben, auf die im Abschnitt 5.6 eingegangen wird.

parseFunctions.c: In dieser Datei sind die Funktionen für den Parser.y abgelegt. Durch die Funktionen werden die Commands für die verschiedenen Befehle festgelegt. Diese sind notwendig, da die Befehle unterschiedlich in den Registern abgespeichert werden, je nachdem ob zum Beispiel ein Befehl mit einem Direktwert oder ein paralleler Vektor- und Skalarbefehl abgearbeitet werden.

Parser.y: Dies ist die Hauptdatei des SCOTCHas Assemblers.

Scanner.l: Im Scanner werden die erkannten Operationen, Commands und sonstige Zeichen erkannt und in Tokens für den Parser umgewandelt.

warOut.c: Hier sind Warnmeldungen definiert für den Fall, dass keine Startadresse oder eine Adresse im Speicher doppelt belegt wurden.

wertReturn.c: Enthält eine Funktion zur Erzeugung des Speicherabbilds

writeInFile.c: Die Funktionen der Datei definieren was in die Ausgabedatei geschrieben wird.

5.5.3 Die verschiedenen Datenstrukturen

Im folgendem werden die wichtigsten Datenstrukturen des Assemblers kurz aufgezeigt und erläutert. Die Datentypen bzw. Strukturen sind in der Datei commit.h definiert:

Die VALUE-Struktur

```
typedef struct
{     int line;
      int value;
      _Bool regex; // true when reg and false if expr.
      WERT_RETURN *wertret;
}VALUE;
```

Wie bereits in 5.5.1 erwähnt, übergibt der Scanner, bei erfolgreichem Erkennen eines Schlüsselwortes, einen Token. Zusätzlich wird eine Instanz des VALUE-Datentyps dem Parser übergeben. Dieser Datentyp enthält dann alle notwendigen Informationen über das erkannte Schlüsselwort. Dazu gehören die Zeile, der Wert (zum Beispiel ein Direktwert) und ein Zeiger auf `wertret`, falls es sich um eine Expression handelt.

Die COMMAND_RETURN-Struktur

```
typedef struct adress
{
      int line; // Zeile, in der der Befehl vorkommt
      int wortAnz; // Anzahl der Konstanten: für Befehl 0
      unsigned int code; // enthält den Opcode, Direktwerte oder Register
als Befehlswort
      int flag; // gibt an ob es sich um eine Expression handelt
      int isSkalar,isVektor;  // Vektor- oder Skalarbefehl?
      struct wertRet *expr; // Zeiger auf eine Expression, falls eine
vorkommen sollte
}COMMAND_RETURN;
```

Wurde vom Parser ein kompletter Befehl erkannt, so werden alle nötigen Informationen des Befehls in diese Struktur gepackt. Die obigen Kommentare dienen als Erklärung der einzelnen Variablen in der Struktur.

Die EXPRESSION_RETURN-Struktur

```
typedef struct exprRet
{
      int value; // Direktwert
      enum operations operation; // Operation, Variable, Label
      char label[TEXT_LENGTH]; // falls Label, steht hier der Labelname
      struct exprRet *left; // wenn vorhanden, Zeiger auf das linke Kind
```

```
      struct exprRet *right; // wenn vorhanden, Zeiger auf das rechte Kind
}EXPRESSION_RETURN;
```

Die Expression_Return-Struktur enthält alle nötigen Informationen zur Berechnung einer Expression. Die obigen Kommentare dienen als Erklärung der einzelnen Variablen in der Struktur. Falls man mehr über diese Struktur oder Implementierung wissen möchte, kann man sich im Abschnitt 5.7.3 darüber informieren

Die WERT_RETURN-Struktur

```
typedef struct wertRet
{     int load;
      int memory;
      EXPRESSION_RETURN *expression;
      struct wertRet *next;
}WERT_RETURN;
```

Diese Struktur kommt zum Einsatz wenn man mehrere Konstanten oder Expressions nacheinander mit der „.dc“-Direktive definiert. Aus dieser Wertefolge (Konstanten oder Expressions) wird eine verkettete Liste aufgebaut, um diese dann später in das Speicherabbild zu schreiben.

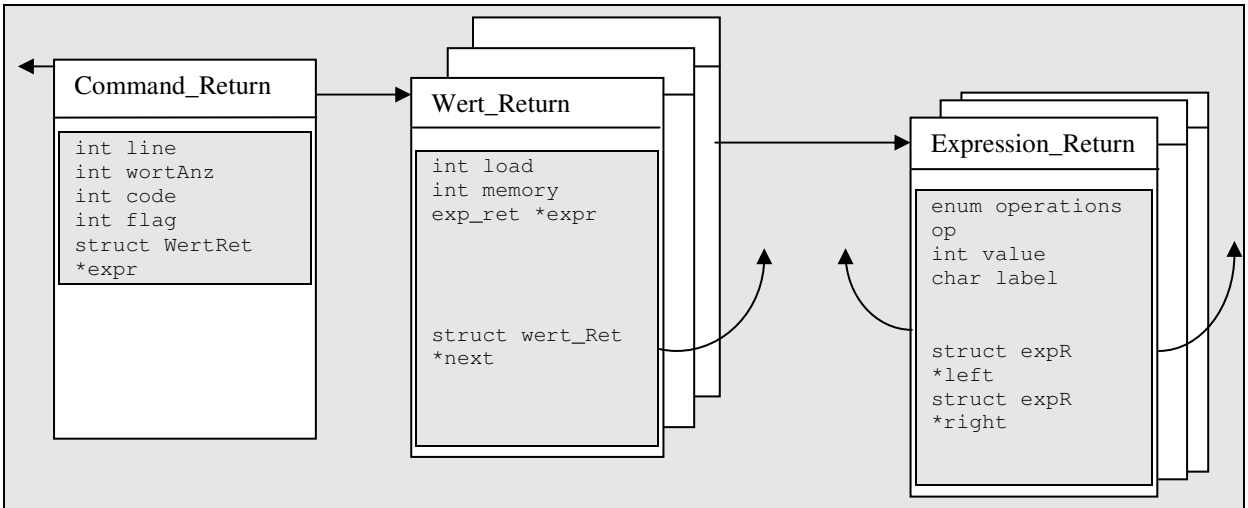


Bild 5.5.1: Illustration des internen Aufbaues der Datenstrukturen, z.B. .dc 80+10, 5+3*10, 5, 7

5.5.4 Die Parse Funktionen

Die Parse-Funktionen wurden implementiert, um den Maschinencode eines entsprechenden Befehls zu Erzeugen. Der Maschinencode eines bestimmten Befehls wird in eine Integervariable geschrieben. Da der Prozessor einen recht komplexen Befehlssatz besitzt, haben wir uns für mehrere Parse-Funktionen entschieden und somit erreicht, dass nahezu jeder Befehlstyp eine eigene Funktion besitzt. Insgesamt entspricht das einer Anzahl von sieben Parse-Funktionen, die wie folgt aussehen und sich in der parsefunctions.h befinden:

```
COMMAND_RETURN          NewCommand_ScalarAndReg
COMMAND_RETURN          NewCommand_VectorAndReg
COMMAND_RETURN          NewCommand_ScalarRegDirectValue
COMMAND_RETURN          NewCommand_MoveCmds
COMMAND_RETURN          NewCommand_VectorRegDirectValue
COMMAND_RETURN          NewCommand_VectorAlu
COMMAND_RETURN          NewCommand_shuffle
```

Achtung: Die Übergabeparameter wurden aus Platzgründen entfernt !!

Beispiel für die Verwendung einer Parse-Funktion:

Man möchte beispielsweise einen Maschinencode für einen ADD-Befehl erzeugen, dieser kann dann folgendermaßen erstellt werden.

Assemblerbefehl:

```
ADD A, X, Y
```

Zerlegung des Assemblerbefehls:

```
OPCode ADD: 0x20
Register A: 0x1
Register X: 0x2
Register Y: 0x3
```

Funktionsprototyp:

```
COMMAND_RETURN NewCommand_ScalarAndReg(int line,int cmd_6bit,int reg1_2bit,
int reg2_2bit,int reg3_2bit,WERT_RETURN *wert,int wortAnz, int low8);
```

Die Funktion zur Erzeugung des gewünschten Maschinencodes kann dann folgendermaßen aufgerufen werden:

```
cmdStruct = NewCommand_ScalarAndReg(1, 0x20, 0x1, 0x2,0x3, NULL, 1, 0);
```

5.5.4 Parallele Befehle

Da der Hicovec gleichzeitig einen Befehl von der Skalareinheit und von der Verktoreinheit ausführen kann mussten wir dieses Feature auch in unseren Assembler mit integrieren. Im Normalfall benötigen Befehle der Skalareinheit 12 Bit und Befehle der Vektoreinheit 20 Bit, die zusammen genau 32 Bit entsprechen. Aus diesem Grund wurde eine Funktion namens uniteCommands eingefügt, die jeweils einen Skalarbefehl und einen Vektorbefehl zu einem Befehlswort vereint.

Allerdings kann ein Befehl bei der Verwendung von Direktwerten, egal ob von der Vektor- oder Skalareinheit, auch die kompletten 32 Bit benötigen, somit muss vor dem Zusammenfügen zweier Befehle überprüft werden, dass keine Überschreibungen des Maschinencodes auftreten. Diese Entscheidung geschieht durch die Variablen isSkalar und isVektor in der CommandReturn Struktur. Handelt es sich beispielsweise um einen Skalarbefehl ohne Direktwert (12 Bit werden benötigt), so wird die flag isSkalar auf eins gesetzt. Analog geschieht das auch für Vektorbefehle. Kommen Direktwerte in einem Befehl vor, so werden die Variablen auf 0 gesetzt und somit ist kein Zusammenfügen der beiden Befehle möglich, da sonst Überschreibungen im Maschinencode auftreten würden.

Zur Verdeutlichung des Systems, hier ein kleines Schema:

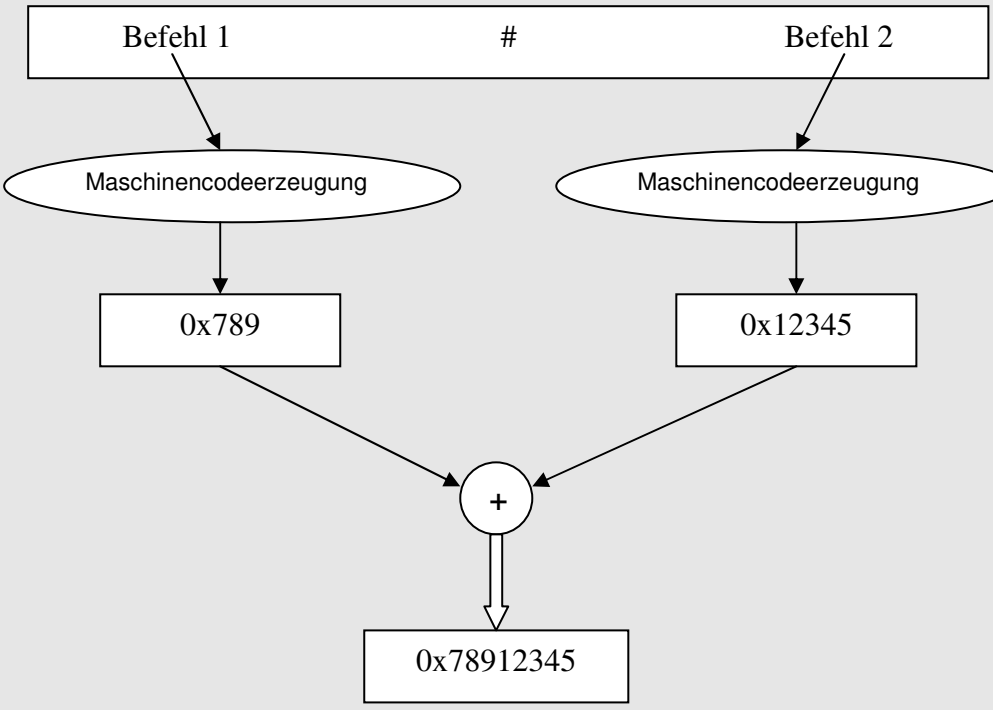


Bild 5.4.4: Schematische Darstellung zur Erzeugung von paralellen Befehlen

5.5.5 Aufbau der Expressions

Wird vom Scanner eine Expression erkannt, so werden entsprechend viele Instanzen der EXPRESSION_RETURN-Struktur erzeugt, bis die Expression komplett aufgelöst ist. Die Verwaltung einer Expression wird in einem Baum gehalten, wobei die Wurzel in den meisten Fällen die Operation enthält und die dazugehörigen Blätter die Zahlenwerte. Diese Wurzel-Struktur wird dann weiter gereicht und dem entsprechenden Befehl, indem die Expression auftritt, angehängt. Am Ende des Parse-Vorgangs werden dann alle Expressions rekursiv ausgerechnet und in die entsprechende Zieladresse des Speicherabbilds geschrieben.

Arten von Expressions:

- Zahlenwerte / Konstanten: Bei Zahlen oder Konstanten wird der entsprechende Wert in die Variable value geschrieben
- Operationen (Addition, Subtraktion, Multiplikation, etc.): Baumstruktur wird erstellt
- Label: Werden behandelt wie ein Wert, jedoch wird bei Berechnung die Adresse, an der das Label aufgetreten ist, ermittelt.

Beispiel einer Expression (15*7):

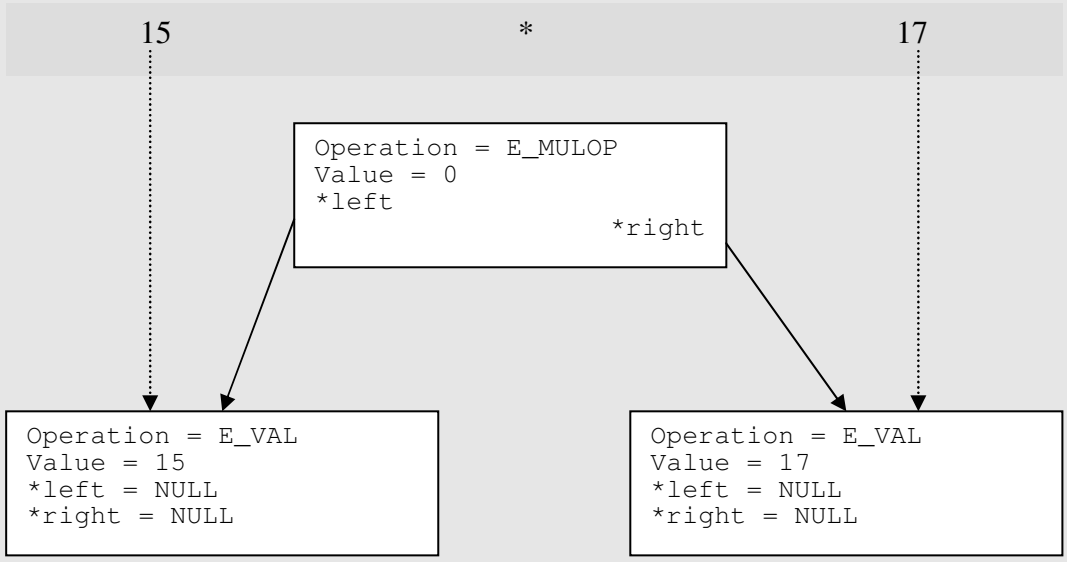


Bild 5.4.5: Expressionsstruktur eines Beispiels

5.5.6 Die Code-and-Expressions Funkion

Die Code-and-Expression() Funktion schreibt den Maschinencode (z.B. Opcode) in das Speicherabbild, lässt die Direktwerte, wenn vorhanden, aber aus. Das übernimmt die Funktion unter 5.5.7. Wenn Direktwerte vorkommen, werden sie in die Compilerstrukturen eingepflegt.

Die flag Variable zeigt an, um welchen Befehl es sich handelt:

- flag = 0 : Alle anderen Befehle ohne Expr.
- flag = 1 : Assembler Direktive kann auch Expr. enthalten z.B. .dc 80+30
- flag = 2 : Befehle mit Expr. z.B. LD ST ALU JMP VMOV

5.5.7 Die calculateExpressions Funktion

Die Funktion calculateExpression() schreibt das Ergebnis einer Expression in das Speicherabbild.Dies geschieht zum Schluss, da ja auch mit Labels gerechnet werden kann. Die load Variable zeigt die Art der Expression an, also wie sie letztendlich in den Speicher geschrieben werden muss.

Es sind letztendlich 3 Möglichkeiten zu unterscheiden:

- Load = 0: Bit 0 - Bit 15
- Load = 1: Bit 21 - Bit 28
- Load = 2: alle Bits, denn Konstante (.dc)

5.6 Die Objektdatei

Das Binärfile, dass der Assembler erstellt, hat eine ganz bestimmte Struktur. Diese wird nun hier erklärt. Es wird nicht nur der eigentliche Code, sondern viel mehr zusätzliche Information in das Objektfile gepackt. Die Maschine, welche den Code ausführt, interessiert natürlich nur die Code-Section. Also der Bereich der den Maschinencode enthält. Aber es können auch weitere Informationen ausgewertet werden, z.B. vom VOHDCA, dass den Code in den Speicher schreibt. Auch Simulatoren für die CPU können zusätzliche Informationen gebrauchen um komfortable Debug-Ausgaben zu tätigen.

Außerdem wird hier kein Speicherabbild erzeugt, welches dann direkt in der Hardware abgelegt wird, sondern es wird zu jedem neuen .org eine neue Code-Section erzeugt und die Anfangsadresse dieser Section mit in das Hexfile geschrieben, somit wird nicht unnötig Speicher belegt wenn größere Speicherbereiche unbenutzt bleiben. Das VOHDCA Tool muss natürlich auch mit dem Format konform sein.

Bsp:

```
.org 0x00
.dc 8
.dc 7
.org 0x100
.dc 5
```
Code 0.1

Wenn dieser Codeabschnitt in den Speicher direkt abgebildet werden soll, müssen alle Adressen von 0x02 bis 0x100 auch mit aufgeführt werden. Ein Hexdump des zu übertragenden Binär Files würde dann so aussehen (zur Demonstration hier 16 Bit Adressierung):

```
0000 0008 0007 0000 0000 0000 0000 0000 0000
0008 0000 0000 0000 0000 0000 0000 0000 0000
0010 0000 0000 0000 0000 0000 0000 0000 0000
0018 0000 0000 0000 0000 0000 0000 0000 0000
0020 0000 0000 0000 0000 0000 0000 0000 0000
0028 0000 0000 0000 0000 0000 0000 0000 0000
.
.
.
0080 0000 0000 0000 0000 0000 0000 0000 0000
0088 0000 0000 0000 0000 0000 0000 0000 0000
0090 0000 0000 0000 0000 0000 0000 0000 0000
0098 0000 0000 0000 0000 0000 0000 0000 0000
0100 0005 0000 0000 0000 0000 0000 0000 0000
```
Hexdump 0.1

Ein Hexdump mit Sektionen würde so aussehen (vereinfacht):

```
0000 0000 0008 0007 0100 0005 0000 0000 0000
```
Hexdump 0.2

Der Hexdump 0.1 könnte somit direkt vom Binärfile in den Speicher übertragen werden. Es sind keine weiteren Schritte notwendig, aber dafür wird jede Menge Platz verbraucht. Dieser Effekt würde noch verstärkt werden, wenn zum Beispiel das erste .org erst bei 0x5000 anfangen würde.

Dagegen ist das Binärfile mit Sektionen (Hexdump 0.2) platzsparend und übersichtlich. Zuerst steht die Adresse des physikalischen Speichers, dann die Daten, welche dorthin geschrieben werden sollen. Fängt ein neuer Abschnitt in einem anderen Adressbereich an, so wird wieder die Startadresse angegeben, dann die dort zu schreibenden Daten. Im Sektionenverzeichnis, welches vor den

eigentlichen Sektionen steht, ist genau vermerkt wo die einzelnen Sektionen starten und wo sie enden.

5.6.1 Das Sektionenverzeichnis

Header: Im Header steht die Magic-Number, welche angibt von welcher Assembler Version dieses Objectfile erstellt wurde. In diesem Falle steht die Zahl 0xF00E. Nach dieser Zahl stehen noch 2 Byte Padding (zum Auffüllen).

Sektionenverzeichnis: Zuerst stehen in der Section Map die Anzahl der Sections im Objectfile, danach werden nacheinander die vorkommenden Sections mit ihrer Nummer, Anfang und Ende im Objectfile in 16 Byte Schritten, aufgeführt.

Sektionen: Die Metadaten sind die Sections, welche in der Section Map aufgeführt sind. Hier werden die Informationen zu den einzelnen Sections hineingeschrieben.

Label: In dieser Section steht am Anfang die Anzahl der Sections, danach werden die Informationen zu jedem Label geschrieben. Die Informationen eines Labels enthalten Labellänge in Byte, Speicheradresse auf welche das Label verweist und den Namen in ASCII-Zeichen. Wenn die Länge ungerade ist, dann wird noch ein Nullbyte angehängt.

Zeilen: In dieser Section steht für jede Zeilennummer des Sourcefiles die zugehörige Speicheradresse in welcher der erste Befehl in dieser Zeile steht. Steht kein Befehl in der Zeile, so wird die Speicheradresse der nächsten Zeile verwendet.

Breakpoints: Hier stehen die Breakpoints mit ihrer zugehörigen Nummer und der Speicheradresse an denen sie stehen.

Comments: Die Comments-Section beinhaltet zuerst die Anzahl der Kommentare. Danach folgt eine Auflistung mit den Informationen Commentlänge in Byte und den Namen im ASCII Zeichenformat. Ist die Länge eines Kommentars ungerade wird ein Nullbyte angehängt.

Info: In dieser Section steht ein Informationstext, der vom Entwicklerteam des Assemblers definiert wurde. Ist die Länge wiederum ungerade wird ein Nullbyte angehängt.

Code: In der Code-Section steht die Anfangsadresse des jeweiligen Code-Teiles, die im Assembler bei .org festgelegt wurde. Nach dieser Adresse folgt der kompilierte Code in seiner binärform. Für jedes .org im Sourcefile wird eine Code Section im Objectfile erstellt.

Register: In der Register-Section steht der Inhalt des Programm Counters um dem Simulator mitzuteilen an welcher Adresse er starten soll. Außerdem stehen hier die Inhalte der Register A, X, Y.

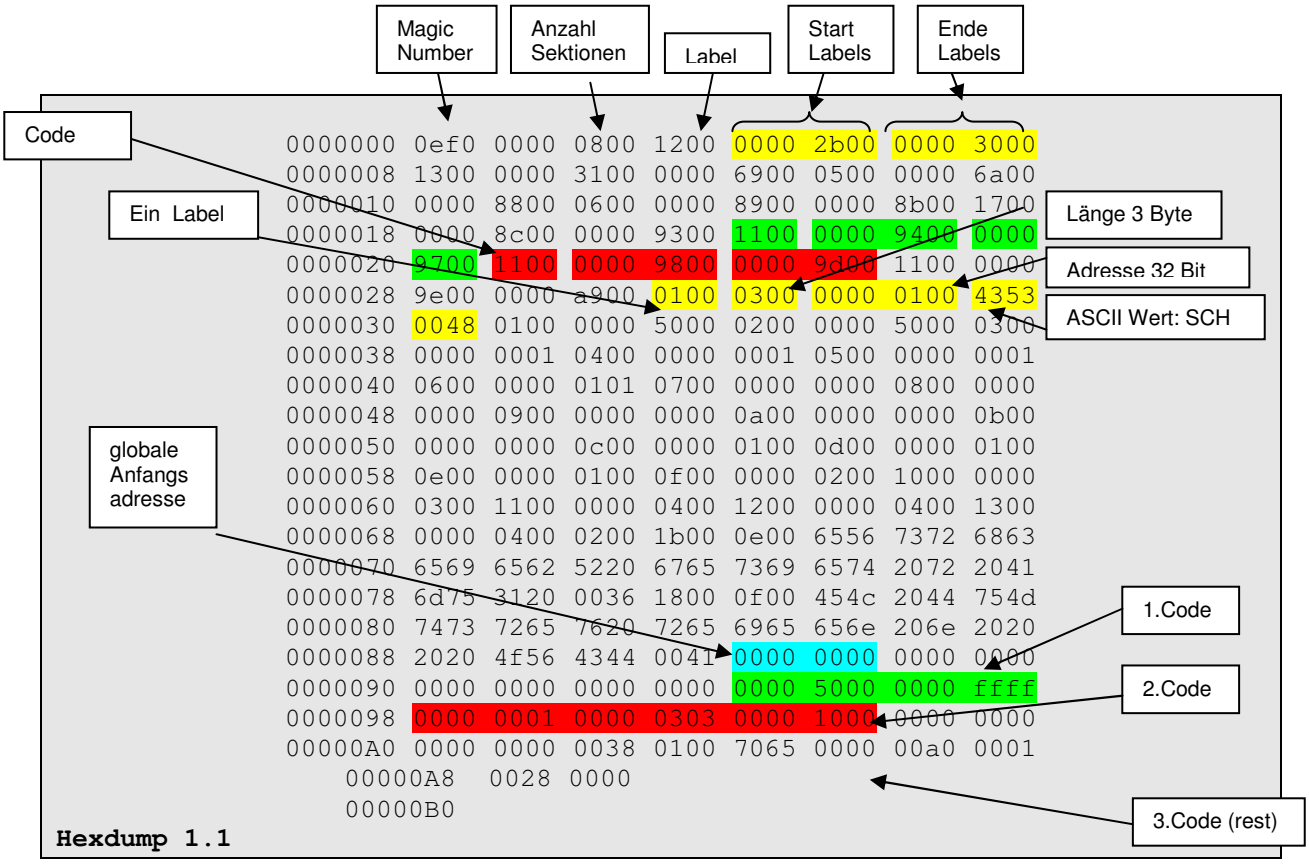
Aufbau:

Header:	Magic Number			16 Bit
	Padding			16 Bit
Sektionen-verzeichnis:	Anzahl der Sektionen N			16 Bit
	N mal:	Typ (Code: 0x11, Labels: 0x12, Zeilen: 0x13, Breakpoints: 0x4, Comments: 0x5, Info 0x6, Register: 0x17)		16 Bit
		Anfang in der Datei		32 Bit
		Ende in der Datei		32 Bit
Sektionen:	Label:	Labelanzahl X		16 Bit
		X mal:	Labellänge in Byte	16 Bit
			Speicheradresse	32 Bit
			ASCII-Name (8 Bit, wenn ungerade Zeichenanzahl ein Nullbyte anhängen)	N*16Bit
	Zeilen (Zeilenanzahl X):	X mal:	Zeilennummer	16 Bit
			Speicheradresse	32 Bit
	Breakpoints (Breakpointanzahl X):	X mal:	Breakpointnummer	16 Bit
			Speicheradresse	32 Bit
	Comments	Commentanzahl X		16 Bit
		X mal:	Zeilennummer	16 Bit
			Comment in ASCII (8 Bit, wenn ungerade Zeichenanzahl ein Nullbyte anhängen)	N*16 Bit
	Info:	Info in ASCII(8 Bit, wenn ungerade Zeichenanzahl ein Nullbyte anhängen)		N*16Bit
	Code:	Anfangsadresse		32 Bit
		Code		32 Bit
	Register:	Programm Counter		32 Bit
		Register A		32 Bit
		Register X		32 Bit
		Register Y		32 Bit

5.6.2 Beispiel-Programmausschnitt mit Hexcode

```
    .org 0x50
    .dc $FFFF

    .org 0x100
    .dc $0303
    .dc 16


    .org 0x00
    .start
     nop

SCH:                        // Label
     JNZ [0 + SCH]
     OR A,A,Y
     ST [0+$100], A


     HALT
     .end
Code 1.1
```

Der Compiler gibt die Binärdaten im BigEndian Format aus.



Zum Verständnis sind einige Bereiche des Binärfiles kommentiert worden. Das VOHDCA Tool liest die Daten ein und wandelt Sie direkt in das LittleEndian Format um, bevor die Abarbeitung weitergeht. Wir beziehen uns bei den weiteren Erklärungen auf den Hexdump 1.2.

```
00000000:  f0 0e 00 00 00 07 00 12  00 00 00 26 00 00 00 2b...........&...+
00000008:  00 13 00 00 00 2c 00 00  00 67 00 06 00 00 00 68.....,...g.....h
00000010:  00 00 00 6a 00 17 00 00  00 6b 00 00 00 72 00 11...j.....k...r..
00000018:  00 00 00 73 00 00 00 76  00 11 00 00 00 77 00 00...s...v.....w..
00000020:  00 7c 00 11 00 00 00 7d  00 00 00 88 00 01 00 03.|.....}........
00000028:  00 00 00 01 53 43 48 00  00 01 00 00 00 50 00 02....SCH......P..
00000030:  00 00 00 50 00 03 00 00  00 50 00 04 00 00 01 00...P.....P......
00000038:  00 05 00 00 01 00 00 06  00 00 01 00 00 07 00 00................
00000040:  01 01 00 08 00 00 00 00  00 09 00 00 00 00 00 0a................
00000048:  00 00 00 00 00 0b 00 00  00 00 00 0c 00 00 00 00................
00000050:  00 0d 00 00 00 01 00 0e  00 00 00 01 00 0f 00 00................
00000058:  00 01 00 10 00 00 00 02  00 11 00 00 00 03 00 12................
00000060:  00 00 00 04 00 13 00 00  00 04 00 14 00 00 00 04................
00000068:  56 4f 44 43 41 00 00 00  00 00 00 00 00 00 00 00VODCA...........
00000070:  00 00 00 00 00 00 00 00  00 50 00 00 ff ff 00 00.........P......
00000078:  01 00 00 00 03 03 00 00  00 10 00 00 00 00 00 00................
00000080:  00 00 38 00 00 01 65 70  00 00 a0 00 01 00 28 00..8...ep......(.
00000088:  00 00 -- -- -- -- -- --  -- -- -- -- -- -- -- --..--------------
Hexdump 1.2
```

Wir haben uns bei der Implementierung unseres Projektes für die 32 Bit Adressierung entschieden, **das Objekt-File jedoch ist im 16 Bit Modus** organisiert, daher ist beim Hexdump 1.2 die Adressierung links abgeändert worden. Das spielt aber keine Rolle, da VOHDCA mit dem 16 Bit adressierten Objekt-File umgehen kann und daraus ein 32 Bit adressiertes Speicherabbild anfertigt.

Achtung: Die unter Hexdump 1.2 dargestellte Binärdatei ist kein Speicherabbild. Die links dargestellten Adressen spiegeln also nicht den realen Adressraum wieder.

5.6.3 Funktionen zum erstellen der Hexdatei

Die Funktionen createFile und makeObjectFile sind für das Schreiben der Sektionen verantwortlich. Wenn man neue Sektionen einführen will, ist es nötig diese Dateien abzuändern. Alle Daten die beim Scan und Parse – Vorgang gespeichert werden, sind in mehreren Strukturen und Arrays abgebildet worden. Die Funktionen müssen also diese Daten einsammeln und strukturiert im Binärfile ablegen. Zuerst wird mit der getSections() Funktion, die Anzahl, Art und Länge der vorhandenen Sections aus den Compilerstrukturen ermittelt. Mit diesem Wissen kann das Sektionenverzeichnis mit der createSectionTab() Funktion erstellt werden. Zum Schluss werden dann mit der createSections() Funktion die einzelnen Sektionen geschrieben.

6. Demo Applikation SCOTCH-RACE

6.1 Einleitende Gedanken

Bei der Entwicklung der Demoapplikation haben wir vor allem darauf geachtet, die Funktionsweise des SCOTCH einem breiteren, nicht unbedingt fachkundigen Publikum zu zeigen. Zu diesem Zweck haben wir uns dazu entschieden, ein kleines Autorennspiel zu programmieren. In diesem Spiel kommen sowohl die Skalar als auch die Vektoreinheit zum tragen, so dass die Vorzüge des HiCoVec aufgezeigt werden können.

Als Eingabegerät wird die IOMISC verwendet. Hier können Eingaben entweder über die 4 Taster, oder über einen an J18 angeschlossen, digitalen Joystick erfolgen. Die Ausgabe erfolgt über die VGA-Schnittstelle, wobei sowohl der Grafik also auch der ASCII-Modus verwendet werden.

6.2 Spielerklärung

Gestartet wird das Spiel durch Drücken des Drehknopfes an dem Reference-Board. Nun sieht man eine Straße mit rot-weißen Begrenzungen. Innerhalb dieser Begrenzungen kann man das kleine Fahrzeug, welches sich am unteren Bildrand befindet, mit Hilfe des Joysticks oder der Taster nach links oder rechts bewegen. Die Vorwärtsbewegung des Autos erfolgt automatisch und wird durch die nach unten wandernde Straße dargestellt.

Nun liegen auf der Straße, in unterschiedlichen Abständen und Positionen, Kleeblätter und Bomben. Ziel ist es nun die Kleeblätter einzusammeln, in dem man frontal von unten auf sie zufährt. Für jedes Kleeblatt bekommt man einen Punkt. Fährt man jedoch frontal auf eine der Bomben, so wird eines der 5 Leben abgezogen, die zum Start gutgeschrieben werden.

Das Spiel ist gewonnen wenn man es schafft, 10 Kleeblätter zu sammeln ohne alle 5 Leben verloren zu haben.

Das Spiel kann, nachdem es gewonnen oder verloren wurde, durch Drücken des Drehknopfes neu gestartet werden.

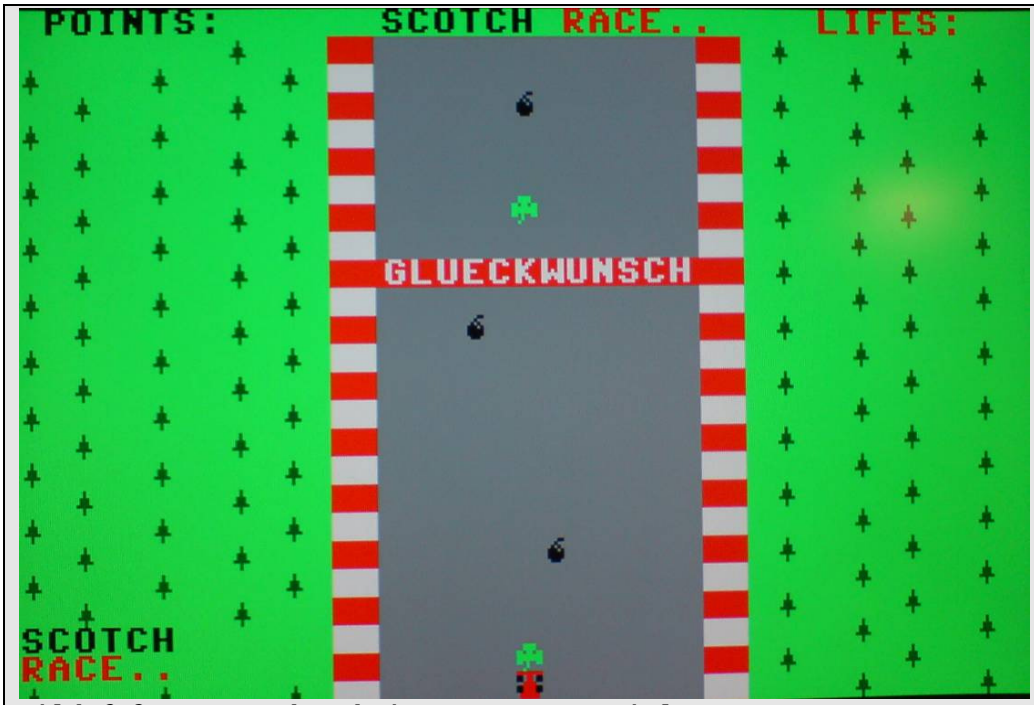


Bild 6.2: Screenshot bei gewonnenem Spiel

6.3 Funktionsweise

Aufbau der Straße:

Wird das Spiel in den Speicher geladen, so werden eine einzelne Zeile der Straße mit roter und eine mit weißer Begrenzung in den VGA Speicher auf eine „blinde Zeile“ geladen (siehe Kapitel 3.4). Wird das Spiel nun gestartet, so wird ein String, welcher aus 3 Leerzeichen und einem „t“ besteht, mit Hilfe der Vektoreinheit an den gesamten Adressbereich der ASCII-Schnittstelle gesendet. Die ASCII-Schnittstelle interpretiert nun das „t“ als einen kleinen Tannenbaum. Dadurch kommt der Hintergrund zustande.

Nun werden in Schleifen je 8x die Straßenzeile mit weißer Begrenzung und 8x die mit roter Begrenzung in den Grafikspeicher geschrieben. Dies geschieht mittels einer zweiten Schleife insgesamt 12x, so dass die Straße, wie unter Bild 6.2 ersichtlich, aufgebaut wurde. All diese Speicheroperationen werden mit dem Vektor-Store Befehl getätigt und werden dadurch besonders schnell ausgeführt.

Als letzten Schritt werden nun die Strings „SCOTCH“ und „RACE“ an 2 Stellen des Bildschirmes geschrieben.

Bewegen der Straße:

Um eine scheinbare Bewegung des Autos über die Straße zu simulieren, wird einfach die Straße nach unten „bewegt“. Dazu wird zunächst die letzte Zeile der Straße in einem Vektorregister gesichert. Dann inkrementiert sich ein Zähler um eine Zeile nach oben, sichert diese in ein anderes Vektorregister, dekrementiert sich wieder zurück und schreibt diese Zeile dort hin. Dieser Vorgang wird nun 191x wiederholt, so dass nun alle Zeilen um eins nach unten geschoben wurden. Als letztes wird die am Anfang gesicherte letzte Zeile über die erste geschrieben. Dann wird eine Warteschleife mit 10.000 Speicherzugriffen durchlaufen, da der Vorgang sonst durch die hohe Geschwindigkeit des Prozessors viel zu schnell ablaufen würde.

Platzieren des Autos:

Vor jedem Schleifendurchgang, welcher die Straße eine Zeile nach unten bewegt, wird zuerst die Tasterstellung (bzw. Joystickposition) überprüft. Je nachdem ob er nach links oder rechts bewegt wird, wird ein Zähler X dekrementiert bzw. inkrementiert. Dieser Zähler bestimmt die Position des Fahrzeuges. Ist dieser Zähler auf 0, so wird das Auto an den linken Straßenrand gezeichnet, bei 11 ganz rechts. Das Auto selbst befindet sich im normalen Programmspeicher und wird jedesmal in den Grafikspeicher an die jeweilige Position kopiert. Alle anderen möglichen Positionen werden mit der Straßenfarbe überzeichnet, um zum einen die vorherige Position zu überzeichnen, zum anderen um zu verhindern, dass das Auto oder die Objekte zusammen mit der Straße nach oben geshiftet werden.

Platzieren der Objekte:

Die Objekte (Kleeblatt und Bombe) werden auf dieselbe Weise wie das Auto platziert, jedoch am oberen Bildrand, wodurch sie zusammen mit der Straße nach unten „wandern“. Außerdem erfolgt die Darstellung nicht durch den Grafikmodus, sondern die Symbole für Bombe und Kleeblatt sind in die ASCII Hardware bei den Zeihen für „k“ und „b“ integriert.

Zählen der Punkte:

Hier kommen zwei Zähler zum Einsatz. Einer Zählt die Punkte von 0 bis 10, der andere die Leben von 5 bis 0. Um zu bestimmen ob einer der Zähler inkrementiert bzw. dekrementiert werden soll, wird bei jedem shiften der Straße überprüft, ob sich eine Zeile direkt über dem Auto eine Referenzzeile befindet. Entspricht diese Referenzzeile der untersten Zeile der Bombe, so werden die Leben dekrementiert, entspricht sie der des Kleeblattes, so wird der Punktezähler inkrementiert. Diese Zähler werden nun in der obersten Zeile zusammen mit dem Text „POINTS“ und „LIFES“ ausgegeben.

Ende des Spiels:

Erreicht der Punktezähler die 10, so wird das Spiel gestoppt und in der Mitte ein Banner mit „GLUECKWUNSCH“ eingeblendet (Siehe Bild 6.2). Erreicht der Lebenszähler die 0, so wird ein Banner mit der Aufschrift „GAME OVER“ eingeblendet. Das Spiel kann in beiden Fällen neu gestartet werden.

Erweiterungsmöglichkeiten:

Der Code des Spieles kann relativ einfach erweitert werden, um so zum Beispiel weitere Level oder Schwierigkeitsgrade einzuführen.

7. Aussicht

Als Erweiterung können bei späterem Bedarf noch zusätzliche Schnittstellen eingebunden und benutzt werden. Dies erlaubt einen noch größeren Anwendungsbereich und Funktionalität des HiCoVec Prozessor.

Nach erfolgreicher Implementierung des gewünschten, fertigen Systems in das FPGA Bord währe der nächste Schritt das Fertigen eines speziell für diesen Zweck Entwickelten ASICs, einem festverdrahteten, in Silizium gegossenem Chip. Durch den bereitstehenden Assembler und dem angepasstem VOHDCA wäre es dann möglich eigene Programme für den HiCoVec zu entwickeln und zu testen.

Eine softwareseitige Weiterentwicklung könnte noch ein Simulator für den HiCoVec-Prozessor sein. Dieser Simulator könnte dazu dienen, entwickelte Programme ohne die Hardware zu testen.

Auch der GCC-Compiler könnte an den Befehlssatz des HiCoVec-Prozessors angepasst werden. Dadurch wäre es möglich, in C geschriebene Programme auf dem SCOTCH-System ablaufen zu lassen.

Zu guter letzt könnte das Gesamtsystem aus Hard- und Software in der Lehre eingesetzt weren. So könnte mit der Zeit, sowohl in der Fakultät für Elektrotechnik, als auch in der Fakultät für Informatik, die bisherigen Assemblersysteme ersetzt werden. Der Vorteil hier wäre, dass der HiCoVec, im Gegensatz zu dem Hauptsächlich verwendeten Motorola 68000, hardwareseitig erweiterbar ist, und auch über eine moderne, parallele Architektur verfügt.

8. Anhang

8.1 Quellen

- Xilinx: www.xilinx.com
- HiCoVec: http://www.opencores.org/projects.cgi/web/hicovec/
- VISCY: http://www.hs-augsburg.de/~kiefer/

8.2 Verweise

8.2.1 Abbildungsverzeichnis:

8.3 Tabellen

8.3.1 Adressen der VGA-Zeilen

Zu diesen Adressen muss noch die Basisadresse (Standard: 0x2000) addiert werden.

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
0	28	50	78	A0	C8	F0	118	140	168	190	1B8	1E0	208	230	258	280	2A8	2D0	2F8
21	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**32**	**33**	**34**	**35**	**36**	**37**	**38**	**39**	**40**
320	348	370	398	3C0	3E8	410	438	460	488	4B0	4D8	500	528	550	578	5A0	5C8	5F0	618
41	**42**	**43**	**44**	**45**	**46**	**47**	**48**	**49**	**50**	**51**	**52**	**53**	**54**	**55**	**56**	**57**	**58**	**59**	**60**
640	668	690	6B8	6E0	708	730	758	780	7A8	7D0	7F8	820	848	870	898	8C0	8E8	910	938
61	**62**	**63**	**64**	**65**	**66**	**67**	**68**	**69**	**70**	**71**	**72**	**73**	**74**	**75**	**76**	**77**	**78**	**79**	**80**
960	988	9B0	9D8	A00	A28	A50	A78	AA0	AC8	AF0	B18	B40	B68	B90	BB8	BE0	C08	C30	C58
81	**82**	**83**	**84**	**85**	**86**	**87**	**88**	**89**	**90**	**91**	**92**	**93**	**94**	**95**	**96**	**97**	**98**	**99**	**100**
C80	CA8	CD0	CF8	D20	D48	D70	D98	DC0	DE8	E10	E38	E60	E88	EB0	ED8	F00	F28	F50	F78
101	**102**	**103**	**104**	**105**	**106**	**107**	**108**	**109**	**110**	**111**	**112**	**113**	**114**	**115**	**116**	**117**	**118**	**119**	**120**
FA0	FC8	FF0	1018	1040	1068	1090	10B8	10E0	1108	1130	1158	1180	11A8	11D0	11F8	1220	1248	1270	1298
121	**122**	**123**	**124**	**125**	**126**	**127**	**128**	**129**	**130**	**131**	**132**	**133**	**134**	**135**	**136**	**137**	**138**	**139**	**140**
12C0	12E8	1310	1338	1360	1388	13B0	13D8	1400	1428	1450	1478	14A0	14C8	14F0	1518	1540	1568	1590	15B8
141	**142**	**143**	**144**	**145**	**146**	**147**	**148**	**149**	**150**	**151**	**152**	**153**	**154**	**155**	**156**	**157**	**158**	**159**	**160**
15E0	1608	1630	1658	1680	16A8	16D0	16F8	1720	1748	1770	1798	17C0	17E8	1810	1838	1860	1888	18B0	18D8
161	**162**	**163**	**164**	**165**	**166**	**167**	**168**	**169**	**170**	**171**	**172**	**173**	**174**	**175**	**176**	**177**	**178**	**179**	**180**
1900	1928	1950	1978	19A0	19C8	19F0	1A18	1A40	1A68	1A90	1AB8	1AE0	1B08	1B30	1B58	1B80	1BA8	1BD0	1BF8
181	**182**	**183**	**184**	**185**	**186**	**187**	**188**	**189**	**190**	**191**	**192**	**193**	**194**	**195**	**196**	**197**	**198**	**199**	**200**
1C20	1C48	1C70	1C98	1CC0	1CE8	1D10	1D38	1D60	1D88	1DB0	1DD8	1E00	1E28	1E50	1E78	1EA0	1EC8	1EF0	1F18

8.3.2 Adressen der ASCII-Zeichen

Zu diesen Adressen muss noch die Basisadresse (Standard: 0x4000) addiert werden.

Zeile:																					
1	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13
1	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F	20	21	22	23	24	25	26	27
2	28	29	2A	2B	2C	2D	2E	2F	30	31	32	33	34	35	36	37	38	39	3A	3B
2	3C	3D	3E	3F	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F
3	50	51	52	53	54	55	56	57	58	59	5A	5B	5C	5D	5E	5F	60	61	62	63
3	64	65	66	67	68	69	6A	6B	6C	6D	6E	6F	70	71	72	73	74	75	76	77
4	78	79	7A	7B	7C	7D	7E	7F	80	81	82	83	84	85	86	87	88	89	8A	8B
4	8C	8D	8E	8F	90	91	92	93	94	95	96	97	98	99	9A	9B	9C	9D	9E	9F
5	A0	A1	A2	A3	A4	A5	A6	A7	A8	A9	AA	AB	AC	AD	AE	AF	B0	B1	B2	B3
5	B4	B5	B6	B7	B8	B9	BA	BB	BC	BD	BE	BF	C0	C1	C2	C3	C4	C5	C6	C7
6	C8	C9	CA	CB	CC	CD	CE	CF	D0	D1	D2	D3	D4	D5	D6	D7	D8	D9	DA	DB
6	DC	DD	DE	DF	E0	E1	E2	E3	E4	E5	E6	E7	E8	E9	EA	EB	EC	ED	EE	EF
7	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	FA	FB	FC	FD	FE	FF	100	101	102	103
7	104	105	106	107	108	109	10A	10B	10C	10D	10E	10F	110	111	112	113	114	115	116	117
8	118	119	11A	11B	11C	11D	11E	11F	120	121	122	123	124	125	126	127	128	129	12A	12B
8	12C	12D	12E	12F	130	131	132	133	134	135	136	137	138	139	13A	13B	13C	13D	13E	13F
9	140	141	142	143	144	145	146	147	148	149	14A	14B	14C	14D	14E	14F	150	151	152	153
9	154	155	156	157	158	159	15A	15B	15C	15D	15E	15F	160	161	162	163	164	165	166	167
10	168	169	16A	16B	16C	16D	16E	16F	170	171	172	173	174	175	176	177	178	179	17A	17B
10	17C	17D	17E	17F	180	181	182	183	184	185	186	187	188	189	18A	18B	18C	18D	18E	18F
11	190	191	192	193	194	195	196	197	198	199	19A	19B	19C	19D	19E	19F	1A0	1A1	1A2	1A3
11	1A4	1A5	1A6	1A7	1A8	1A9	1AA	1AB	1AC	1AD	1AE	1AF	1B0	1B1	1B2	1B3	1B4	1B5	1B6	1B7
12	1B8	1B9	1BA	1BB	1BC	1BD	1BE	1BF	1C0	1C1	1C2	1C3	1C4	1C5	1C6	1C7	1C8	1C9	1CA	1CB
12	1CC	1CD	1CE	1CF	1D0	1D1	1D2	1D3	1D4	1D5	1D6	1D7	1D8	1D9	1DA	1DB	1DC	1DD	1DE	1DF
13	1E0	1E1	1E2	1E3	1E4	1E5	1E6	1E7	1E8	1E9	1EA	1EB	1EC	1ED	1EE	1EF	1F0	1F1	1F2	1F3
13	1F4	1F5	1F6	1F7	1F8	1F9	1FA	1FB	1FC	1FD	1FE	1FF	200	201	202	203	204	205	206	207
14	208	209	20A	20B	20C	20D	20E	20F	210	211	212	213	214	215	216	217	218	219	21A	21B
14	21C	21D	21E	21F	220	221	222	223	224	225	226	227	228	229	22A	22B	22C	22D	22E	22F
15	230	231	232	233	234	235	236	237	238	239	23A	23B	23C	23D	23E	23F	240	241	242	243
15	244	245	246	247	248	249	24A	24B	24C	24D	24E	24F	250	251	252	253	254	255	256	257
16	258	259	25A	25B	25C	25D	25E	25F	260	261	262	263	264	265	266	267	268	269	26A	26B
16	26C	26D	26E	26F	270	271	272	273	274	275	276	277	278	279	27A	27B	27C	27D	27E	27F
17	280	281	282	283	284	285	286	287	288	289	28A	28B	28C	28D	28E	28F	290	291	292	293
17	294	295	296	297	298	299	29A	29B	29C	29D	29E	29F	2A0	2A1	2A2	2A3	2A4	2A5	2A6	2A7
18	2A8	2A9	2AA	2AB	2AC	2AD	2AE	2AF	2B0	2B1	2B2	2B3	2B4	2B5	2B6	2B7	2B8	2B9	2BA	2BB
18	2BC	2BD	2BE	2BF	2C0	2C1	2C2	2C3	2C4	2C5	2C6	2C7	2C8	2C9	2CA	2CB	2CC	2CD	2CE	2CF
19	2D0	2D1	2D2	2D3	2D4	2D5	2D6	2D7	2D8	2D9	2DA	2DB	2DC	2DD	2DE	2DF	2E0	2E1	2E2	2E3
19	2E4	2E5	2E6	2E7	2E8	2E9	2EA	2EB	2EC	2ED	2EE	2EF	2F0	2F1	2F2	2F3	2F4	2F5	2F6	2F7
20	2F8	2F9	2FA	2FB	2FC	2FD	2FE	2FF	300	301	302	303	304	305	306	307	308	309	30A	30B
20	30C	30D	30E	30F	310	311	312	313	314	315	316	317	318	319	31A	31B	31C	31D	31E	31F
21	320	321	322	323	324	325	326	327	328	329	32A	32B	32C	32D	32E	32F	330	331	332	333
21	334	335	336	337	338	339	33A	33B	33C	33D	33E	33F	340	341	342	343	344	345	346	347
22	348	349	34A	34B	34C	34D	34E	34F	350	351	352	353	354	355	356	357	358	359	35A	35B
22	35C	35D	35E	35F	360	361	362	363	364	365	366	367	368	369	36A	36B	36C	36D	36E	36F
23	370	371	372	373	374	375	376	377	378	379	37A	37B	37C	37D	37E	37F	380	381	382	383
23	384	385	386	387	388	389	38A	38B	38C	38D	38E	38F	390	391	392	393	394	395	396	397
24	398	399	39A	39B	39C	39D	39E	39F	3A0	3A1	3A2	3A3	3A4	3A5	3A6	3A7	3A8	3A9	3AA	3AB
24	3AC	3AD	3AE	3AF	3B0	3B1	3B2	3B3	3B4	3B5	3B6	3B7	3B8	3B9	3BA	3BB	3BC	3BD	3BE	3BF
25	3C0	3C1	3C2	3C3	3C4	3C5	3C6	3C7	3C8	3C9	3CA	3CB	3CC	3CD	3CE	3CF	3D0	3D1	3D2	3D3
25	3D4	3D5	3D6	3D7	3D8	3D9	3DA	3DB	3DC	3DD	3DE	3DF	3E0	3E1	3E2	3E3	3E4	3E5	3E6	3E7

8.4 Dateien

8.4.1 VHDL-Dateien

```
scotch.vhd
cfg.vhd
SCOTCH_SYSTEM/scotch_system.vhd
SCOTCH_SYSTEM/VISCY/buslogic.vhd
SCOTCH_SYSTEM/VISCY/dbgshell.vhd
SCOTCH_SYSTEM/HiCoVec/datatypes.vhd
```

```
SCOTCH_SYSTEM/HiCoVec/hicovec.vhd
SCOTCH_SYSTEM/HiCoVec/groups/addressgroup.vhd
SCOTCH_SYSTEM/HiCoVec/groups/aluinputgroup.vhd
SCOTCH_SYSTEM/HiCoVec/groups/cpu.vhd
SCOTCH_SYSTEM/HiCoVec/groups/flaggroup.vhd
SCOTCH_SYSTEM/HiCoVec/groups/registergroup.vhd
SCOTCH_SYSTEM/HiCoVec/groups/vector_executionunit.vhd
SCOTCH_SYSTEM/HiCoVec/groups/vector_slice.vhd
SCOTCH_SYSTEM/HiCoVec/units/alu.vhd
SCOTCH_SYSTEM/HiCoVec/units/controlunit.vhd
SCOTCH_SYSTEM/HiCoVec/units/dataregister.vhd
SCOTCH_SYSTEM/HiCoVec/units/db_multiplexer.vhd
SCOTCH_SYSTEM/HiCoVec/units/demultiplexer1x4.vhd
SCOTCH_SYSTEM/HiCoVec/units/flag.vhd
SCOTCH_SYSTEM/HiCoVec/units/instructioncounter.vhd
SCOTCH_SYSTEM/HiCoVec/units/memoryinterface.vhd
SCOTCH_SYSTEM/HiCoVec/units/multiplexer2.vhd
SCOTCH_SYSTEM/HiCoVec/units/multiplexer4.vhd
SCOTCH_SYSTEM/HiCoVec/units/selectunit.vhd
SCOTCH_SYSTEM/HiCoVec/units/shuffle.vhd
SCOTCH_SYSTEM/HiCoVec/units/valu_controlunit.vhd
SCOTCH_SYSTEM/HiCoVec/units/vector_alu_32.vhd
SCOTCH_SYSTEM/HiCoVec/units/vector_controlunit.vhd
SCOTCH_SYSTEM/HiCoVec/units/vector_register.vhd
Debugger/debugger.vhd
Debugger/rs323.vhd
IO/ascii.vhd
IO/blockram.vhd
IO/debug_io.vhd
IO/iomisc.vhd
IO/lcd.vhd
IO/VGA/ascii_mul.vhd
IO/VGA/vga.vhd
IO/VGA/vga_core.vhd
IO/VGA/vga_ram.vhd
```

8.4.2 SCOTCHas-Dateien

```
SCOTCHas/src/commit.h
SCOTCHas/src/config.h
SCOTCHas/src/createFile.c
SCOTCHas/src/debug.c
SCOTCHas/src/error.c
SCOTCHas/src/errors_and_warnings.h
SCOTCHas/src/errOut.c
SCOTCHas/src/expression.c
SCOTCHas/src/functions.h
SCOTCHas/src/lists.c
SCOTCHas/src/makeObjectFile.c
SCOTCHas/src/parseFunctions.c
SCOTCHas/src/Parser.y
SCOTCHas/src/Scanner.l
SCOTCHas/src/warOut.c
SCOTCHas/src/wertReturn.c
SCOTCHas/src/writeInFile.c
```

8.5 Befehlsreferenszkarte

Befehl		Beschreibung
LD	AXY0, [AXY0 + AXYi]	Ladeanweisung
ST	[AXY0 + AXYi], A	Speicheranweisung
ADD	AXY0, AXY0 , AXYi	Addition
ADC	AXY0, AXY0 , AXYi	Addition mit Carry
INC	AXY0, AXY0	Inkrementierung
SUB	AXY0, AXY0, AXYi	Subtraktion
SBC	AXY0, AXY0, AXYi	Subtraktion mit Carry
DEC	AXY0, AXY0	Dekrementierung
AND	AXY0, AXY0, AXYi	UND-Verknüpfung
OR	AXY0, AXY0, AXYi	ODER-Verknüpfung
XOR	AXY0, AXY0, AXYi	Exklusiv-Oder-Verknüpfung
LSL	AXY0, AXY0	Schiebeoperation nach Links
LSR	AXY0, AXY0	Schiebeoperation nach Rechts
ROL	AXY0, AXY0	Schiebeoperation nach Links (Carry einfügen)
ROR	AXY0, AXY0	Schiebeoperation nach Rechts (Carry einfügen)
MUL	AXY0, AXY0, AXYi	Multiplikation
JMP	[AXY0 + AXYi]	Unbedingter Sprungbefehl
JAL	AXY0, [AXY0 + AXYi]	„Jump-And-Link“ (für Unterprogramme)
JZ	[AXY0 + AXYi]	Springe wenn Zero-Flag gesetzt
JNZ	[AXY0 + AXYi]	Springe wenn Zero-Flag nicht gesetzt
JC	[AXY0 + AXYi]	Springe wenn Carry-Flag gesetzt
JNC	[AXY0+AXYi]	Springe wenn Carry-Flag nicht gesetzt
CLZ		Lösche Zero-Flag
SEZ		Setze Zero-Flag
CLC		Lösche Carry-Flag
SEC		Setzte Carry-Flag
HALT		Prozessor anhalten
NOP		Keine Skalar-Operation ausführen
VNOP		Keine Vektor-Operation durchführen
MOV	vregl (AXY), AXY0	Kopiere Skalareinheit ➔ Vektoreinheit
MOV	AXY0, vregl (AXY)	Kopiere Vektoreinheit ➔ Skalareinheit
MOVA	vregl, AXY0	K-maliges kopieren in Vektorregister
VLD	vregl, [AXY0 + AXY]	Vektor-Ladeanweisung
VST	[AXY0 + AXY], vregl	Vektor-Speicheranweisung
VMOV	vregl, vregl	Kopieren zwischen Vektorregistern
VMOV	vregh, vregl	Kopieren zwischen Vektorregistern
VMOV	vregl, vregh	Kopieren zwischen Vektorregistern
VMOL	vregl, vregl	Linksverschiebung aller Wörter des Vektorregisters
VMOR	vregl, vregl	Rechtsverschiebung aller Wörter des Vektorregisters
VADD	.breitevoll vregl, vregl, vregl	Vektor-Addition
VSUB	.breitevoll vregl, vregl, vregl	Vektor-Subtraktion
VAND	.breitevoll vregl, vregl, vregl	Vektor-Und-Verknüpfung
VOR	.breitevoll vregl, vregl, vregl	Vektor-Oder-Verknüpfung
VXOR	.breitevoll vregl, vregl, vregl	Vektor-Exklusiv-Oder-Verknüpfung
VLSL	.breitevoll vregl, vregl	Vektor-Schiebeoperation links
VLSR	.breitevoll vregl, vregl	Vektor-Schiebeoperation rechts
VMUL	.breitebw vregl, vregl, vregl	Vektor-Multiplikation
VSHUF	vregl, vregl, vregl, perm	Mischen von Vektorregistern