Hochschule
Augsburg University of
Applied Sciences

HiCoSim

- Simulator für einen konfigurierbaren Vektorprozessor -

Projektarbeit im Sommersemester 2009

Luca Calchera
Hibetoullah Ftima
Sebastian Minning
Daniel Obermüller
Müge Özugur
Michael Wager
Andreas Weber

Betreuer: Prof. Dr. Gundolf Kiefer

Inhaltsverzeichnis

1) Einführung (Müge Özugur, Hibetoullah Ftima)

1.1) Motivation (Müge Özugur)

„Der HiCoVec-Prozessor ist ein an der Hochschule Augsburg entworfener und als Open-Source-Hardware freigegebener konfigurierbarer Vektorprozessor **[1]**“.

Das Ziel dieser Projektarbeit ist, einen erweiterbaren Simulator für den HiCoVec-Prozessor zu entwerfen.

Der Grund warum wir uns für das Projekt HiCoSim entschieden haben, war unter anderem die interessante und innovative Architektur des HiCoVec. Durch die integrierte konfigurierbare Vektoreinheit sind beachtliche Rechenleistungen möglich. Die Vorzüge des HiCoVec kommen unter anderem bei der Bild- und Grafikverarbeitung zum tragen, da hier besonders oft Vektorrechnungen ausgeführt werden müssen. Beispiel 2.1 verdeutlicht dieses Potential anhand einer einfachen Vektoraddition.

Der Software-Entwurf für den HiCoVec war bis jetzt wenig komfortabel:
Zum Ablauf wurde in jedem Fall ein FPGA-Board benötigt. Das Debuggen und Testen war nur in eingeschränkter Form möglich. Auch spätere Erweiterungen (wie z.B. ein Compiler) werden dadurch erschwert.

Diese Probleme sollen durch den „HiCoSim“ behoben werden.

HiCoSim - Projekt:

Als Versionsverwaltung kam Subversion (svn) zum Einsatz.

Das Projekt kann über

svn co https://io.informatik.fh-augsburg.de/svn/HiCoSim/

ausgecheckt werden.

1.2) Aufgabenstellung und Team (Hibetoullah Ftima)

Für den HiCoVec-Prozessor wird ein erweiterbarer Simulator benötigt, der in der Lage ist, beliebige HiCoVec-Programme auszuführen. Des weiteren muss er, um der flexiblen Architektur des HiCoVec gerecht zu werden, alle möglichen Konfigurationen des Prozessors unterstützen.

Der Simulator soll über eine grafische Oberfläche (GUI) verfügen, über die der Benutzer die Möglichkeit hat, die Simulation zu steuern sowie sich über den aktuellen Zustand des Prozessors zu informieren.
Außerdem muss ein skriptgesteuerter Programmlauf möglich sein – das bedeutet, der Simulator muss auch auf Konsolenebene eine Ein-/Ausgabemöglichkeit zur Verfügung stellen und unabhängig von der grafischen Benutzeroberfläche lauffähig sein.

Für die eigentliche Simulation müssen Methoden entwickelt werden, um die vom scotchAS **[2]** (Assembler) erstellten Objektdateien einzulesen, zu interpretieren und den Speicher zu initialisieren.
Die ausgelesenen Maschinenbefehle müssen decodiert und ausgeführt werden. Der aktuelle Prozessorzustand muss jederzeit überwacht werden können – das beinhaltet die Anzeige sämtlicher Register sowie des Speicherinhalts.

Um einen komfortablen Softwareentwurf zu ermöglichen, soll der HiCoSim umfassende Debugging-Möglichkeiten zur Verfügung stellen. Dazu gehört unter anderem das Setzen und Löschen von Break- und Watchpoints, die Unterbrechung und schrittweise Ausführung des Programms („Single Stepping“) sowie das Modifizieren des Speichers zur Laufzeit.

Um Leistungsabschätzungen hinsichtlich der ausgeführten Programme zu erlauben, sollen umfangreiche Statistiken gesammelt werden. Für an den HiCoVec angeschlossene Peripherie wird eine möglichst problemlose Integration von Plugins angestrebt.

Da weitere Entwicklungen des HiCoVec-Prozessor bereits in Vorbereitung sind (Compiler) oder zukünftig realisiert werden könnten (z.B. Gleitkommaeinheit), ist die Erweiterbarkeit des HiCoSim von Anfang an eine wichtige Anforderung. Diese Erweiterbarkeit bezieht sich dabei vor allem auf den Befehlssatz, die Plugins sowie die Statistiken. Details hierzu können dem Kapitel 5 entnommen werden.

Der Simulator soll einfach zu benutzen, sowie auf unterschiedlichen Systemen lauffähig, also plattformunabhängig, sein. Deshalb wurde die Programmiersprache Java gewählt.

Um eine bessere Auslastung und eine parallele Arbeit am Simulator (Backend) und der grafischen Benutzeroberfläche (Frontend) zu gewährleisten, verteilen sich die Mitglieder der Projektarbeit entsprechend auf diese 2 Bereiche.

Das Team besteht aus:

Frontend:
Daniel Obermüller, Sebastian Minning

Backend:
Luca Calchera, Hibetoullah Ftima, Müge Özugur, Michael Wager, Andreas Weber

1.3) Aufbau des Dokuments (Hibetoullah Ftima)

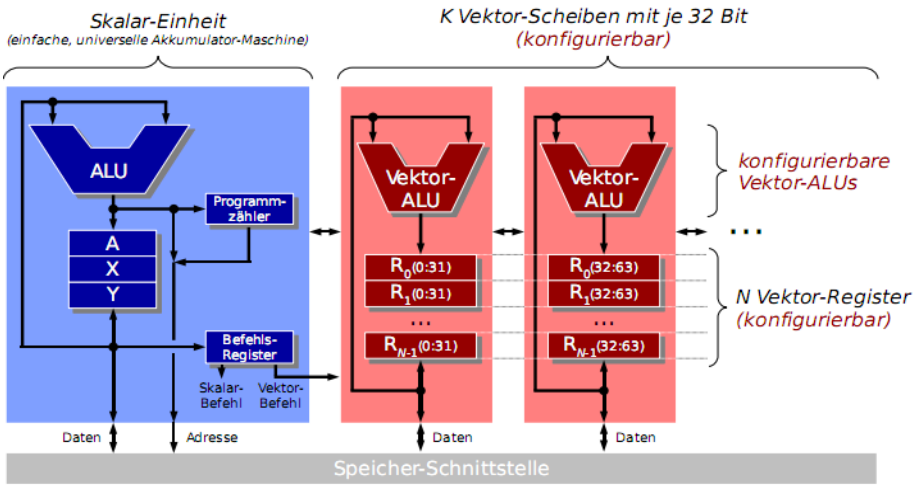
Diese Arbeit ist in sieben Kapitel unterteilt. Kapitel 1 gibt einen kleinen Einblick in das Projekt HiCoSim, schildert die Aufgabenstellung und stellt kurz das Team vor. Kapitel 2 behandelt den HiCoVec Prozessor, der die Grundlage des Simulators bildet. Die Installation und Benutzung des HiCoSim werden im 3. Kapitel erläutert. Kapitel 4 befasst sich mit der Realisierung dieses Projektes. Kapitel 5 behandelt mögliche Änderungen des Befehlssatz, der Statistiken und der Plugins. Die Zusammenfassung ist in Kapitel 6 zu finden. Abschließend bildet Kapitel 7 den Anhang.

2) Beschreibung HiCoVec (Müge Özugur, Hibetoullah Ftima, Luca Calchera)

2.1) Prozessor (Müge Özugur)

Der Prozessor besteht aus einer Skalar- und einer Vektoreinheit (SIMD – Single Instruction Multiple Data), die es ermöglicht, mit einem Befehlsaufruf mehrere Datensätze zu verarbeiten. Der HiCoVec wurde im Rahmen einer Diplomarbeit von Harald Manske **[1]** an der Hochschule Augsburg entwickelt und basiert auf der Idee von Prof. Dr. Kiefer.

Die folgende Abbildung zeigt den Aufbau der Skalar- und Vektoreinheit:



(Abb. 2.1 – HiCoVec [3])

Wie aus der Abbildung ersichtlich, ist die Anzahl der Vektorregister N und die Anzahl der Vektorscheiben K konfigurierbar (das heißt beliebig änderbar), die skalare Multiplikation und die Vektormultiplikation lassen sich getrennt voneinander ein- und ausschalten. Weitere Parameter, die konfigurierbar sind, werden inkl. der Syntax im folgenden Beispiel aufgelistet:

Anzahl der Vektorregister:

```
constant n : integer := 20;
```

Anzahl der Vektorscheiben (nur gerade Zahlen erlaubt):

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

Skalare Multiplikation aktivieren:

```
constant use_scalar_mult : boolean := false;
```

Vektormultiplikation aktivieren:

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

Integrieren der Shuffleeinheit:

constant use_shuffle : boolean := true;

Maximale Shufflebreite (muss durch 4 teilbar sein):

constant max_shuffle_width : integer := 32;

Vektorshift aktivieren:

constant use_vectorshift : boolean := true;

Breite der Vektorverschiebung in Bit:

constant vectorshift_width : integer := 32;

Wie anfangs schon erwähnt lässt sich durch den Einsatz des Vektorprozessors die Anzahl der Befehle reduzieren. Das folgende Beispiel soll dies demonstrieren, es aus der Diplomarbeit von Harald Mankse **[1]** entnommen und geringfügig modifiziert.
Ziel ist die Addition zweier Arrays.

Algorithmus unter Verwendung der Skalareinheit:

```
.org 0x00
.start
            OR Y, 0, 0  ;Zaehler auf 0
LOOP: LD A, [Y + VEK1]  ; VEK1[i] in A laden
            LD X, [Y + VEK2]  ; VEK2[i] in X laden
            ADD A, A, X       ;A + X in A speichern
            ST [Y + ERG], A   ;
            INC Y, Y                ; Zaehler inkrementieren
            SUB 0, Y, 16            ; vgl. Zaehler mit 16
            JNZ [0 + LOOP]          ; springe, wenn Zaehler < 16
            HALT

.org 0xff
VEK1:
      .dc 1
      .dc 2
      .dc 3
      .dc 4
      .dc 5
      .dc 6
      .dc 7
      .dc 8
      .dc 9
      .dc 10
      .dc 11
      .dc 12
      .dc 13
      .dc 14
      .dc 15
      .dc 16

VEK2:
      .dc 16
```

```
      .dc 15
      .dc 14
      .dc 12
      .dc 11
      .dc 10
      .dc 9
      .dc 8
      .dc 7
      .dc 6
      .dc 5
      .dc 4
      .dc 3
      .dc 2
      .dc 1

.org 0x00
ERG:
      .dc 0
.end
```

Algorithmus unter Verwendung der Vektoreinheit:

```
.org 0x00
.start
      OR A, 0, VEK1
      OR X, 0, VEK2
      OR Y, 0, ERG
      VLD R0, [0 + A]   ; VEK1 in R0 laden
      VLD R1, [0 + X]   ; VEK2 in R1 laden
      VADD.W R2, R0, R1 ; R0 + R1 in R2 addieren
      VST [0 + Y], R2   ; R2 in ERG speichern
      HALT

.org 0xff

 [… Registerdefinitionen analog zu Skalarquelltext …]
```

Die Vektoreinheit benötigt sieben Befehle um den Algorithmus auszuführen, während die Skalareinheit 113 Befehle benötigt. Dadurch wird ersichtlich, dass die Vektoreinheit für die selbe Aufgabe wesentlich weniger Rechenzeit in Anspruch nimmt.

Dieses Beispiel wurde auch im Rahmen unserer Projektpräsentation verwendet, um die Vorzüge der Vektoreinheit zu demonstrieren.

2.2) Befehlssatz (Luca Calchera, Hibetoullah Ftima)

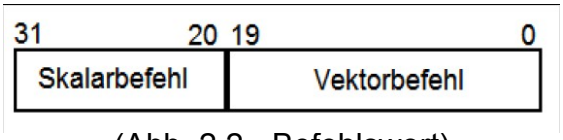
Der Befehlssatz gliedert sich in Befehle für die Skalareinheit, Befehle für die Vektoreinheit und in Befehle, welche in Kooperation beider Einheiten verarbeitet werden. Um den Befehlssatz verstehen zu können, sollte man die Register des Prozessors kennen. Die Register der Skalareinheit sind A, X und Y. Sie sind 32 Bit lang 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.
Die konfigurierbaren Vektorregister sind sowohl als Quell- als auch als Zielregister nutzbar.

Daraus ergeben sich folgende Befehlsgruppen:

regaxy = A, X oder Y
regaxy0 = A, X, Y oder 0
regaxyi = A, X, Y oder Direktwert (16 Bit)
vregl = R0 … R15
vregh = R<0> … R<N>
breitebw = B (8 Bit) oder W (16 Bit)
breitevoll = B (8 Bit), W (16 Bit), DW (32 Bit), QW (64 Bit)

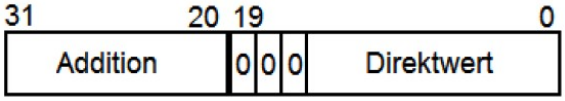
2.2.1) Befehlscodierung (Luca Calchera, Hibetoullah Ftima)

Da die Wortlänge der Befehle 32 Bit beträgt wird ein Befehl sowohl zum Steuern der Skalar- als auch der Vektoreinheit verwendet:



(Abb. 2.2 - Befehlswort)

12 Bit werden für die Befehle der Skalareinheit und die 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:



(Abb. 2.3 – Befehlswort bei Direktwert)

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)

2.2.2) Befehlsreferenz (Luca Calchera)

Der HiCoSim Prozessor kann die folgenden Befehle dekodieren und ausführen:

LOAD: 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 1000--ddss00 00-nnnnnnnnnnnnnnnn
LD Y, [0 + A] ; lade Register Y von Adresse A 1000--ddsstt ----------

LD A, [X + Y] ; lade Register A von Adresse X + Y

Beeinflussung der Flags:

Zeroflag wird gesetzt wenn 0 geladen wird.
Carryflag wird nicht beeinflusst.

STORE: ST [AXY0 + AXYi], A

Erklärung:

Die ST-Anweisung schreibt *nur* den Inhalt des Registers A (Akkumulatorregister) in den Speicher.

Beispiel:

ST A, [0 + 5] ; speichere Register A an Adresse 5 1010----ss00 00-nnnnnnnnnnnnnnnn
ST A, [0 + Y] ; speichere Register A an Adresse Y 1010----sstt ----------
ST A, [X + Y] ; speichere Register A an Adresse X + Y

Beeinflussung der Flags:

Keine Beeinflussung der Flags

ADDITION: ADD AXY0 , AXY0 , AXYi

Erklärung:

Dieser Befehl wird dazu verwendet um eine Additionsoperation durchzuführen.

Beispiel:

ADD A, B, 0 + 5 ; A = B + 5 010000ddss00 000-nnnnnnnnnnnnnnn

Beeinflussung der Flags:

Zeroflag wird nur gesetzt wenn das Ergebnis 0 ist , sonst wird es gelöscht.
Carryflag wird gesetzt wenn das Ergebnis größer/gleich (2^32) ist, sonst wird es gelöscht.

ADDITION MIT CARRY: ADC AXY0 , AXY0 , AXYi

Erklärung:

Der Befehl dient dazu, eine Additionsoperation mit einem Übertrag durchzuführen. Wenn Carryflag gesetzt ist, wird es dazu addiert.

Beispiel:

ADC A, X, Y 010001ddsstt ----------

Beeinflussung der Flags:

Zeroflag wird nur gesetzt wenn das Ergebnis 0 ist , sonst wird es gelöscht.
Carryflag wird gesetzt wenn das Ergebnis größer/gleich (2^32) ist, sonst wird es gelöscht.

INKREMENTIERUNG: INC AXY0, AXY0

Erklärung:

Mit INC kann ein Register inkrementiert werden.

Beispiel:

INC X, Y ; wert von Y+1 in X speichern. Y=5 → X=6 010100ddsstt ----------

Beeinflussung der Flags:

Zeroflag wird gesetzt wenn 0 geschrieben wird.
Carryflag wird gesetzt wenn der Operand 0xffffffff ist

SUBTRAKTION: SUB AXY0, AXY0, AXYi

Erklärung:

Dieser Befehl wird dazu verwendet um 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 010010ddss00 000-nnnnnnnnnnnnnnn
SUB X, 5, 5 ; X = 5 – 5 = 0 Zeroflag gesetzt. Man kann überprüfen, ob die Operanden bzw. zwei Zahlen gleich sind.

Beeinflussung der Flags:

Zeroflag wird nur gesetzt wenn das Ergebnis 0 ist , sonst wird es gelöscht.
Carryflag wird gesetzt wenn Operand 2 <= Operand 1 ist

SUBTRAKTION MIT CARRY: SBC AXY0, AXY0, AXY

Erklärung:

Dieser Befehl wird verwendet, um eine Subtraktionsoperation mit einem Übertrag durchzuführen. Wenn das Carryflag nicht gesetzt ist, wird vom Ergebnis 1 subtrahiert.

Beispiel:

SBC A, Y, X 010011ddsstt ----------

Beeinflussung der Flags:

Zeroflag wird nur gesetzt wenn das Ergebnis 0 ist , sonst wird es gelöscht.
Carryflag wird gesetzt wenn das Ergebnis größer als 0 ist, sonst wird es gelöscht.

DEKREMENTIERUNG: DEC AXY0, AXY0

Erklärung:

Mit DEC kann ein Register dekrementiert werden.

Beispiel:

DEC X, Y ; wert von Y-1 in X speichern. Y=5 → X=4 010110ddsstt ----------

Beeinflussung der Flags:

Zeroflag wird gesetzt wenn 0 geschrieben wird.
Carryflag wird nicht gesetzt wenn Operand 1 = 0 ist.

UND-Verknüpfung**:** AND AXY0, AXY0, AXYi

Erklärung:

Mit dem AND Befehl können 2 Register miteinander verknüpft werden.

Beispiel:

AND A, A, Y ; (A=0) & (Y=1) = (A=0). 011000ddsstt ----------

Beeinflussung der Flags:

Zeroflag wird nur gesetzt wenn das Ergebnis 0 ist , sonst wird es gelöscht.
Carryflag wird immer gelöscht.

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. 011001ddsstt ----------
OR X, 0, $1234 ; belegt Register X mit 0x1234

Beeinflussung der Flags:

Zeroflag wird nur gesetzt wenn das Ergebnis 0 ist , sonst wird es gelöscht.
Carryflag wird immer gelöscht.

EXKLUSIV-ODER-Verknüpfung: XOR AXY0, AXY0, AXYi

Erklärung:

XOR Verknüpfung zweier Register.

Beispiel:

XOR A, A, Y ; 011010ddsstt ----------

Beeinflussung der Flags:

Zeroflag wird nur gesetzt wenn das Ergebnis 0 ist , sonst wird es gelöscht.
Carryflag wird immer gelöscht.

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. 011100ddsstt ----------
X=1 ; LSL X, X ; X=2

Beeinflussung der Flags:

Zeroflag wird nur gesetzt wenn das Ergebnis 0 ist , sonst wird es gelöscht.
Carryflag wird gesetzt, wenn eine 1 aus den 32 Bit geshiftet wird, sonst wird es gelöscht.

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. 011110ddsstt ----------
X=1; LSL X, X ; X=0

Beeinflussung der Flags:

Zeroflag wird nur gesetzt wenn das Ergebnis 0 ist , sonst wird es gelöscht.
Carryflag wird gesetzt, wenn eine 1 aus den 32 Bit geshiftet wird, sonst wird es gelöscht.

Schiebeoperation nach Links (Carry einfügen): ROL AXY0, AXY0

Erklärung:

Befehl um das Register nach links zu verschieben. Wenn das Carry-Flag gesetzt ist, wird es rechts hinzugefügt.

Beispiel:

ROL A,X; 011101ddsstt ----------

Beeinflussung der Flags:

Carryflag wird gesetzt, wenn eine 1 aus den 32 Bit geshiftet wird, sonst wird es gelöscht.
Carryflag wird gesetzt wenn eine 1 aus den 32 Bit geshiftet wird, sonst wird es gelöscht.

Schiebeoperation nach Rechts (Carry einfügen): ROR AXY0, AXY0

Erklärung:

Befehl um das Register nach links zu verschieben. Wenn das Carry-Flag gesetzt ist, wird es links hinzugefügt.

Beispiel:

ROR A, X; 011111ddsstt ----------

Beeinflussung der Flags:

Carryflag wird gesetzt, wenn eine 1 aus den 32 Bit geshiftet wird, sonst wird es gelöscht.
Carryflag wird gesetzt wenn eine 1 aus den 32 Bit geshiftet wird, sonst wird es gelöscht.

MULTIPLIKATION: MUL AXY0, AXY0, AXYi

Erklärung:

Befehl um eine Multiplikationsoperation durchzuführen. Muss hardwareseitig bzw. in der config.vhd aktiviert sein.

Beispiel:

MUL A, X, Y ; A = X * Y. 011011ddsstt ----------

Beeinflussung der Flags:

Zeroflag wird gesetzt wenn das Ergebnis 0 ist, sonst wird es gelöscht.
Die Multiplikation nimmt aus den Quellregistern nur je 16 Bit, das Ergebnis ist wiederum 32 Bit lang. Carry wird immer gelöscht.

Unbedingter Sprungbefehl: JMP [AXY0 + AXYi]

Erklärung:

Befehl um an eine gespeicherte Adresse im Speicher zu springen.

Beispiel:

JMP [0 + LABEL] ; Springe zu LABEL. 00100000ss00 000-nnnnnnnnnnnnnnn

Beeinflussung der Flags:

Keine Beeinflussung der Flags.

„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.
001000ddss00 000-nnnnnnnnnnnnnnn

Beeinflussung der Flags:

Keine Beeinflussung der Flags.

Sprung bei gesetztem Zero-Flag: 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 = True.
001111ddss00 000-nnnnnnnnnnnnnnn

Beeinflussung der Flags:

Keine Beeinflussung der Flags.

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 = False.
001110ddss00 000-nnnnnnnnnnnnnnn

Beeinflussung der Flags:

Keine Beeinflussung der Flags.

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.
001101ddss00 000-nnnnnnnnnnnnnnn

Beeinflussung der Flags:

Keine Beeinflussung der Flags.

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.
001100ddss00 000-nnnnnnnnnnnnnnn

Beeinflussung der Flags:

Keine Beeinflussung der Flags.

Lösche Zero-Flag: CLZ

Erklärung:

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

Setze Zero-Flag: SEZ

Erklärung:

Das Zero-Flag wird gesetzt.

Lösche Carry-Flag: CLC

Erklärung:

Das Carry-Flag wird gelöscht.

Setzte Carry-Flag: SEC

Erklärung:

Carry-Flag wird gesetzt.

Prozessor anhalten: HALT

Erklärung:

Durch den Befehl wird der Prozessor angehalten.

Beeinflussung der Flags:

Keine Beeinflussung der Flags.

Keine Skalar-Operation ausführen: NOP

Erklärung:

Es wird keine Skalar-Operation ausgeführt.

Beispiel: NOP

Beeinflussung der Flags:

Keine Beeinflussung der Flags.

Keine Vektor-Operation durchführen: VNOP

Erklärung:

Es wird kein Vektor-Befehl ausgeführt.

Beeinflussung der Flags:

Keine Beeinflussung der Flags.

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), Y ; Y nach Wort A in R0 kopieren
    MOV A, R1(A) ; Wort A in R1 nach A kopieren

Beeinflussung der Flags:

Keine Beeinflussung der Flags.

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

Beeinflussung der Flags:

Keine Beeinflussung der Flags.

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

Beeinflussung der Flags:

Keine Beeinflussung der Flags.

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

Beeinflussung der Flags:

Keine Beeinflussung der Flags.

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

Beeinflussung der Flags:

Keine Beeinflussung der Flags.

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

Beeinflussung der Flags:

Keine Beeinflussung der Flags.

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

Beeinflussung der Flags:

Keine Beeinflussung der Flags.

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 ; Die Vektorregister R1 und R2 werden addiert und in
das Register R3 gespeichert bei einer Wortlänge von 32 Bit.

Beeinflussung der Flags:

Keine Beeinflussung der Flags.

Vektor-Subtraktion: VSUB.breitevoll vregl, vregl, vregl

Erklärung:

Der Befehl ermöglicht es Vektorregister 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.

Beeinflussung der Flags:

Keine Beeinflussung der Flags.

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.

Beeinflussung der Flags:

Keine Beeinflussung der Flags.

Vektor-Oder-Verknüpfung: VOR.breitevoll vregl, vregl, vregl

Erklärung:

OR-Verknüpfung der Vektorregister.

Beispiel:

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

Beeinflussung der Flags:

Keine Beeinflussung der Flags.

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

Beeinflussung der Flags:

Keine Beeinflussung der Flags..

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.

Beeinflussung der Flags:

Keine Beeinflussung der Flags.

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.

Beeinflussung der Flags:

Keine Beeinflussung der Flags.

Vektor-Multiplikation: 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.

Beeinflussung der Flags:

Keine Beeinflussung der Flags.

Mischen von Vektorregistern: VSHUF vregl, vregl, vregl, perm

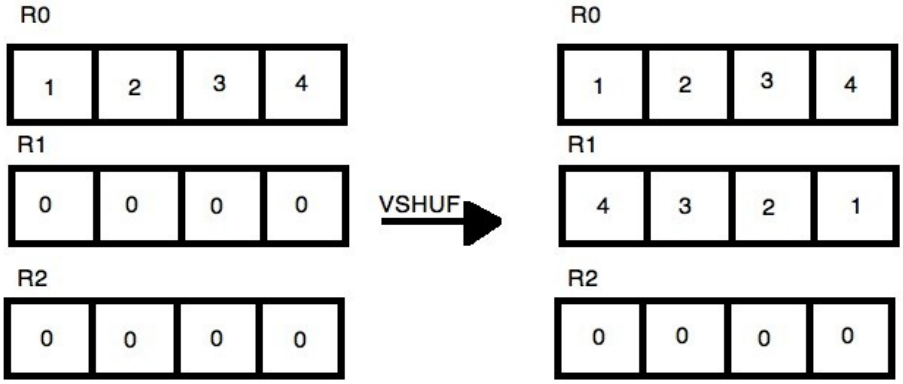
Erklärung:

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

Beispiel:

VSHUF R1,R0,R2,0b11000011100100
0b Bit Beschreibung des VSHUFFLE Befehls
11 Wortlänge (möglich 00-01-10-11)
0 Register Auswahl des erstes Datensatz (0=v , 1=w)
0 Register Auswahl des zweites Datensatz (0=v , 1=w)
0 Register Auswahl des drittes Datensatz (0=v , 1=w)
0 Register Auswahl des viertes Datensatz (0=v , 1=w)
11 Zielregisterwort Auswahl des erstes Datensatz
10 Zielregisterwort Auswahl des zweites Datensatz
01 Zielregisterwort Auswahl des drittes Datensatz
00 Zielregisterwort Auswahl des viertes Datensatz

Beispiel mit K=4 und N=20:



(Abb. 2.4 – Beispiel VSHUF)

Beeinflussung der Flags:

Keine Beeinflussung der Flags.

3) Installation und Benutzung (Sebastian Minning)

3.1) Einleitende Worte (Sebastian Minning)

Im folgenden Kapitel soll ein Überblick verschafft werden wie der Simulator installiert und die Installation geprüft wird, sowie die Nutzung der Konsolenanwendung bzw. der grafischen Oberfläche erläutert werden.

3.2) Installation (Sebastian Minning)

3.2.1) Voraussetzungen (Sebastian Minning)

Es werden keine besonderen Anforderungen an die Hardware gestellt. Eine Ausnahme stellt die Nutzung rechenintensiver Plugins wie z.B. das 'VGA-Plugin' dar. Hier ist es empfehlenswert ein System mit einer leistungsstarken CPU zu verwenden.

Da es sich um eine Java-Anwendung handelt ist eine aktuelle Sun Java Runtime Environment (JRE), ab Version „JRE 6“, erforderlich um den Simulator nutzen zu können. Die Verwendung anderer JREs kann zu unvorhergesehenen Problemen führen.

Optional, aber empfohlen ist es den Assembler SCOTCHas [2] zu installieren, da ansonsten keine „.asm“ Files assembliert werden können.

3.2.2) Installationsanleitung (Sebastian Minning)

Die Installation des HiCoSim ist einfach und kann in wenigen Schritten durchgeführt werden. Eine ausführliche Installationsanleitung findet sich im Ordner „HiCoSim“ in „LIESMICH.txt“.

3.2.3) Prüfen der Installation (Sebastian Minning)

Um zu Testen ob der HiCoSim ordnungsgemäß installiert ist und ob der Simulator fehlerfrei funktioniert, gibt es ein Testprogramm das ausgeführt werden kann – all.o.

Vorgehensweise (Auf der Konsole im Ordner hicosim):

- java -jar hicosim.jar → HiCoSim in der Konsole starten
- open <pfad>/all.o → Die Datei all.o öffnen
- run → Die Simulation startet → Es folgt die Ausgabe: „HALT instruction reached. Stopping processor“
- get 0x1000 0x100e → Den Speicherbereich 0x1000 bis 0x100e anzeigen

Hier muss bei jeder Speicheradresse der Wert 1 stehen! Ist dies der Fall, dann war der Test erfolgreich und der Simulator arbeitet fehlerfrei.

3.3) Benutzung (Sebastian Minning)

Der Simulator kann entweder in der Konsole ausgeführt, oder durch anhängen des Parameters „--gui“ mit der grafischen Oberfläche gestartet werden.

Ausführung unter Linux in der Konsole:

- java -jar hicosim.jar → öffnet die Konsolenanwendung
- java -jar hicosim.jar --gui → öffnet die Grafische Benutzeroberfläche

Es können natürlich auch Anwendungsstarter mit den entsprechenden Parametern angelegt werden um den Programmstart komfortabler zu gestalten

Die Ausführung unter anderen Betriebssystemen läuft weitestgehend analog zum Beispiel ab.

3.3.1) Konsole (Sebastian Minning)

3.3.1.1) Befehlsübersicht (Sebastian Minning)

Es stehen folgende Befehle in der Konsolenanwendung zur Verfügung:

- [q/exit] → Beendet das Programm
- [h/help] → Zeigt die Befehlsübersicht
- [cl/clear] → Löscht den Fensterinhalt
- [o/open] + <Pfad> → Lädt eine Objektdatei (.o)
- [l/list] + <Pfad> → Zeigt den Quellcode an
- [br/break] → Setzt einen Breakpoint
- [del/delete] → Löscht einen Breakpoint
- [wp/watchpoint] → Setzt einen Watchpoint
- [r/run] → Startet die Simulation
- [s/step] → Führt einen Einzelschritt aus
- [reset] → Zurücksetzen
- [i] → Interrupt (Sim. Unterbrechen)
- [get] + <address> → Speicherinhalt anzeigen
- [set] + <address> + <value> → Speicherinhalt setzen
- [setr] + <address> + <value> → Registerinhalt setzen
- [i/info] + <option> → Informationen anzeigen über:
  - regs — Register
  - ip — Programmzählen
  - labels — Labels
  - stats — Statistiken
  - config — Prozessorkonfiguration

– about → Aboutmessage
- [config] + <var> + <value> → Kofiguration ändern
- [d/disas] → Disassemblierten Code anzeigen
- [vga] → VGA-Plugin starten

Die Eingabe von „h“ bzw. „help“ + [Befehl] zeigt eine genauere Beschreibung des Befehls an.

3.3.2) Grafische Benutzeroberfläche (GUI) (Sebastian Minning)

3.3.2.1) Übersicht (Sebastian Minning)

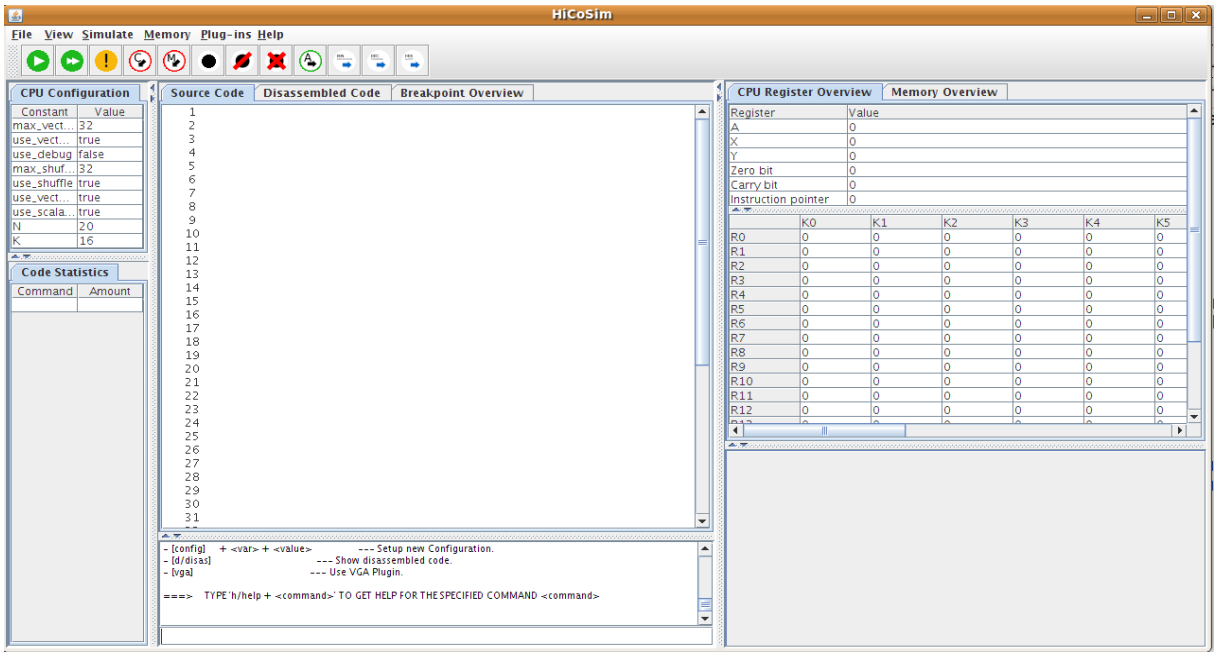
In der grafischen Benutzeroberfläche steht der komplette Funktionsumfang der auch auf der Konsole nutzbar ist zur Verfügung, ergänzt durch zahlreiche Features welche der Benutzerfreundlichkeit zugute kommen und die Arbeit erleichtern

3.3.2.2) Fensterlayout und Erklärung (Sebastian Minning)

Die GUI ist in acht Bereiche aufgeteilt, wovon abgesehen von der Menüleiste und der darunter befindlichen Toolbar alle in ihrer Größe anpassbar sind, sowie ganz aus- und wieder eingeblendet werden können.

Um Werte in editierbaren Tabellenfeldern zu ändern muss dieses mit einem Doppelklick angewählt werden und nach dem Editieren mit der Eingabetaste gesetzt werden.

Dies ist das Standardlayout der GUI:



(Abb. 3.1 – HiCoSim GUI Layout)

Menüleiste und Toolbar:

Im oberen Bereich befinden sich die Menüleiste und die Toolbar.
Über die Menüleiste lassen sich sämtliche Funktionen aufrufen, die wichtigsten sind zusätzlich mit einem Symbol auf der Toolbar vertreten.

Der linke Bereich der GUI wird von der CPU Konfiguration oben und den Code Statistiken unten eingenommen.

CPU-Konfiguration:

Im diesem Fensterbereich wird die aktuelle Konfiguration des Prozessors angezeigt und kann bei Bedarf angepasst werden.

Folgende Parameter können konfiguriert werden (Standardwert):

- max_vector_shift_width → Maximale
- use_vectorshift → Vektorshift nutzen (true)
- use_debug → Debugging aktivieren (false)
- max_shuffle_width → Maximale
- use_shuffle → Shuffle Einheit nutzen (true)
- use_vector_mult → Multiplizierer der Vektoreinheit nutzen (true)
- use_scalar_mult → Multplizierer der Skalareineheit nutzen (true)
- N → Anzahl der Vektorregister (20)
- K → Anzahl der Vektorscheiben (16)

Statistiken:

Hier wird die Anzahl der jeweils ausgeführten Instruktionen der zuletzt abgelaufenen bzw. der gerade laufenden Simulation angezeigt. Die Sortierung erfolgt nach Häufigkeit der vorgekommenen Typen. Die Gesamtanzahl der abgearbeiteten Instruktionen vervollständigt die Übersicht.
Die Statistiken können bei Bedarf über den Menüpunkt „File“ → „Save statistics as ...“ als Textdokument exportiert werden.

In der Mitte oben befindet sich der Fensterbereich in dem der Quellcode, der disassemblierte Code, sowie die Übersicht über die gesetzten Breakpoints angezeigt wird. Innerhalb der einzelnen Ansichten kann über die Reiter am oberen Rand navigiert werden. Darunter befindet sich die Konsole.

Quellcode/Disassembled Code/Breakpoint Overview:

Die Quellcode Ansicht bietet eine Zeilennummerierung, neben der Breakpoints als schwarze Punkte dargestellt werden.
Wird ein Breakpoint erreicht oder die Simulation unterbrochen, so wird die aktuelle Zeile mit roter Farbe gehighlightet. Dies geschieht in der Quellcode Ansicht, in der Ansicht für den disassemblierten Code, sowie in der Speicherübersicht (Instructionpointer).

Konsole:

Die Konsole der GUI kann genau so verwendet werden als ob der Simulator ohne grafische Oberfläche in der (System-)Konsole gestartet wurde.
Fehlermeldungen werden durch rote Schrift besonders hervorgehoben. Normale Ausgaben und Benutzereingaben erscheinen in schwarzer Schrift.
Beim Drücken der Eingabetaste wird der zuletzt ausgeführte Befehl wiederholt.
Die zuletzt eingegebenen Kommandos werden in einer Historie abgelegt und können wie es z.B. unter einem Linux Terminal bekannt ist mit den Pfeiltasten -oben/-unten durchgeblättert werden.
Der rechte Bereich der GUI beinhaltet oben die Übersicht über die CPU Register (Skalar- und Vektorregister), sowie im anderen Reiter die Speicherübersicht.
Unterhalb ist Platzt um Plugins wie z.B. das 'VGA-Plugin' anzuzeigen.

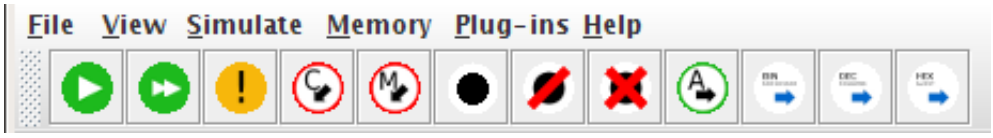
Registeranzeige/Speicherübersicht:

Der linke Reiter beinhaltet die CPU Register, sowohl die der Skalareinheit, als auch die der Vektoreinheit.
Es ist möglich bestimmte Bereiche der Verktorregister aus- bzw. einzublenden. Dies erfolgt über einen Dialog, der in der Menüleiste unter „View“ → „Hide Vector Registers“ bzw. „Show Vector Registers“ aufgerufen werden kann.
Somit ist es möglich uninterresante Bereiche der Vektorregister zu verstecken und die relevanten Daten kompakt darzustellen.
Unter dem rechten Reiter befindet sich die Speicherübersicht, die per Standard in noch einem Reiter: „Memory 1“ öffnet. Hier ist es möglich über die Menüleiste → „Add Memory Overview“ bzw. „Delete Memory Overwiew“ neue Reiter hinzuzufügen die wiederum eine Übersicht über den Speicher enthalten.
Sind mehrere Reiter geöffnet, so kann man in jedem einzelnen einen anderen Speicherbereich anzeigen und so über die Reiter komfortabel navigieren.
Die Buttons „Up“ und „Down“ ermöglichen es in der aktuellen Ansicht vertikal zu scrollen, mit dem links neben den Buttons befindlichen Feld können Adressen direkt angesprungen werden. Dazu muss lediglich die gewünschte Adresse eingegeben und mit Drücken der Eingabetaste bestätigt werden.
Mit den Checkboxen am rechten Rand einer jeden Speicheradresse können Watchpoints gesetzt werden.

3.3.2.3) Toolbar (Sebastian Minning)

Die Toolbar ermöglicht es häufig genutzte Aktionen mit nur einem Mausklick auszuführen. Standardmäßig erscheint sie unterhalb der Menüleiste, kann aber auch mit der Maus „herausgezogen“ werden und erscheint dann in einem eigenen Fenster. Beim Schließen des Toolbar-Fensters erscheint die Toolbar wieder am ursprünglichen Platz unterhalb der Menüleiste.

Die Toolbar:



(Abb. 3.2 – Die Toolbar)

Die Funktionen der Toolbar (von links nach rechts):

- Simulation starten
- Einzelschritt ausführen
- Die Simulation unterbrechen (Interrupt)
- Reset CPU
- Reset Speicher (Memory)
- Breakpoint setzen
- Breakpoint löschen
- Alle Breakpoints löschen
- Assemblieren
- Binäre Ansicht der CPU Register
- Dezimale Ansicht der CPU Register
- Hexadezimale Ansicht der CPU Register

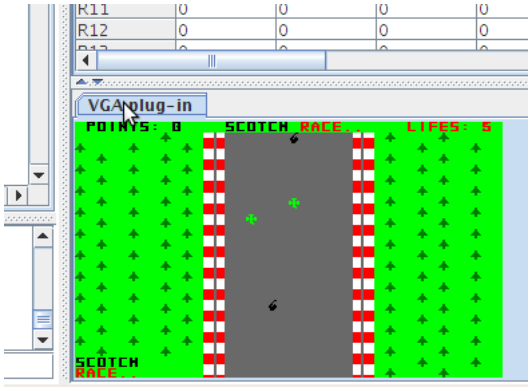
3.3.3) Plugins (Sebastian Minning)

Durch Plugins wird eine einfache Erweiterbarkeit des HiCoSim gewährleistet. Der rechte untere Bereich der GUI ist für die Anzeige von Plugins reserviert.

Als Beispiel soll das Spiel ScotchRace in der GUI mit dem 'VGA-Plugin' ausgeführt werden.

Wenn die Objektdatei geladen ist muss als nächstes über die Menüleiste → „Plug-ins“ → „Toggle vga plug-in“ das 'VGA-Plugin' gestartet werden.
Dann wird die Simulation über „Run“ gestartet. Um das Spiel steuern zu können muss der Reiter „VGA plug-in“ angeklickt werden. Jetzt kann das Spiel mit den Pfeiltasten gesteuert werden. Pfeil nach unten startet das Spiel und die Pfeiltasten nach links und rechts steuern das Fahrzeug.

Beispiel ScotchRace:



(Abb. 3.3 – ScotchRace [2] im 'VGA-Plugin')

4) Realisierung (Michael Wager, Andreas Weber, Daniel Obermüller)

4.1) Backend (Michael Wager, Andreas Weber)

4.1.1) Struktur (Andreas Weber)

Das Backend hat die Aufgabe, die Objektdatei einzulesen, zu interpretieren, zu simulieren und Statistiken zu sammeln. Außerdem wird eine Eingabe/Ausgabemöglichkeit über die Konsole bereitgestellt sowie Methoden, um mit der GUI zu kommunizieren.

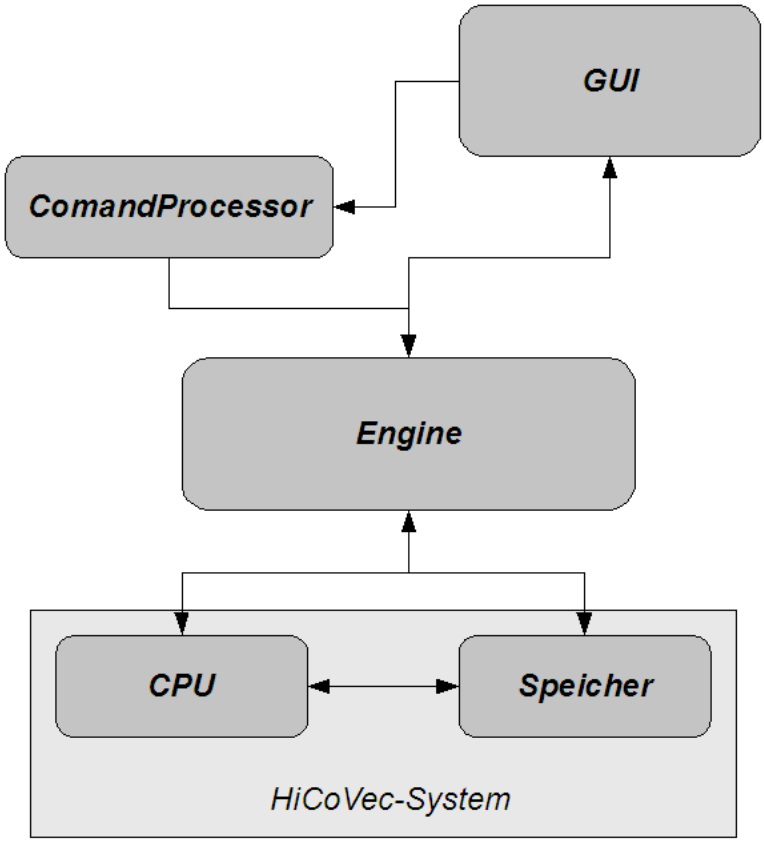
Die Ein- und Ausgabe (IO) auf Konsolenebene ist im Paket `backend.io` realisiert. Der User hat hier die Möglichkeit, den Simulator mit einfachen Kommandos zu bedienen. Auch die GUI greift zur Steuerung des HiCoSims auf diese Methoden zurück.

Der Prozessor selbst wird im Paket `backend.processor` simuliert. Neben dem eigentlichen HiCoVec-Prozessor wurde hier auch der Speicher (**Memory**), der PluginManager und der Disassembler implementiert.

Als zentrale Schnittstelle zwischen dem System und der Ein/Ausgabe (sowohl auf der Konsole als auch in der GUI) dient die **Engine** (Paket: `control`). Mit den Methoden der Klasse ist eine Steuerung des Simulators möglich.

Plugins (Erweiterungen) befinden sich im Paket `plugins`, beispielsweise das VGA-Plugin, mit welchem eine grafische Ausgabe ermöglicht wird.

Die Statistiken sowie weitere Hilfsklassen (z.B. für Bit-Operationen) finden sich im Paket `control.utils`.



(Abb. 4.1 - Struktur)

4.1.2) Engine – Die zentrale Schnittstellenklasse (Andreas Weber)

Um eine möglichst große Modularität zu erreichen und spätere Erweiterungen zu erleichtern, dient die Klasse **Engine** als Schnittstelle zwischen der Ein-/ Ausgabe und dem HiCoVec-System. Sie enthält Methoden zur Ablaufsteuerung und zur Ausgabe von Meldungen auf der GUI.

Beim Instantiieren der **Engine** legt diese auch die Objekte **Processor** und **Memory** an und speichert sie jeweils als Referenzen auf deren Interfaces (*IProcessor* und *IMemory*).

4.1.2.1) Kommunikation zwischen GUI und Engine (Andreas Weber)

Die Kommunikation zwischen der Engine und der grafischen Oberfläche funktioniert über ein Listener-Interface.
Die Engine kann genau eine Referenz auf einen *iListenerFrontend* verwalten.

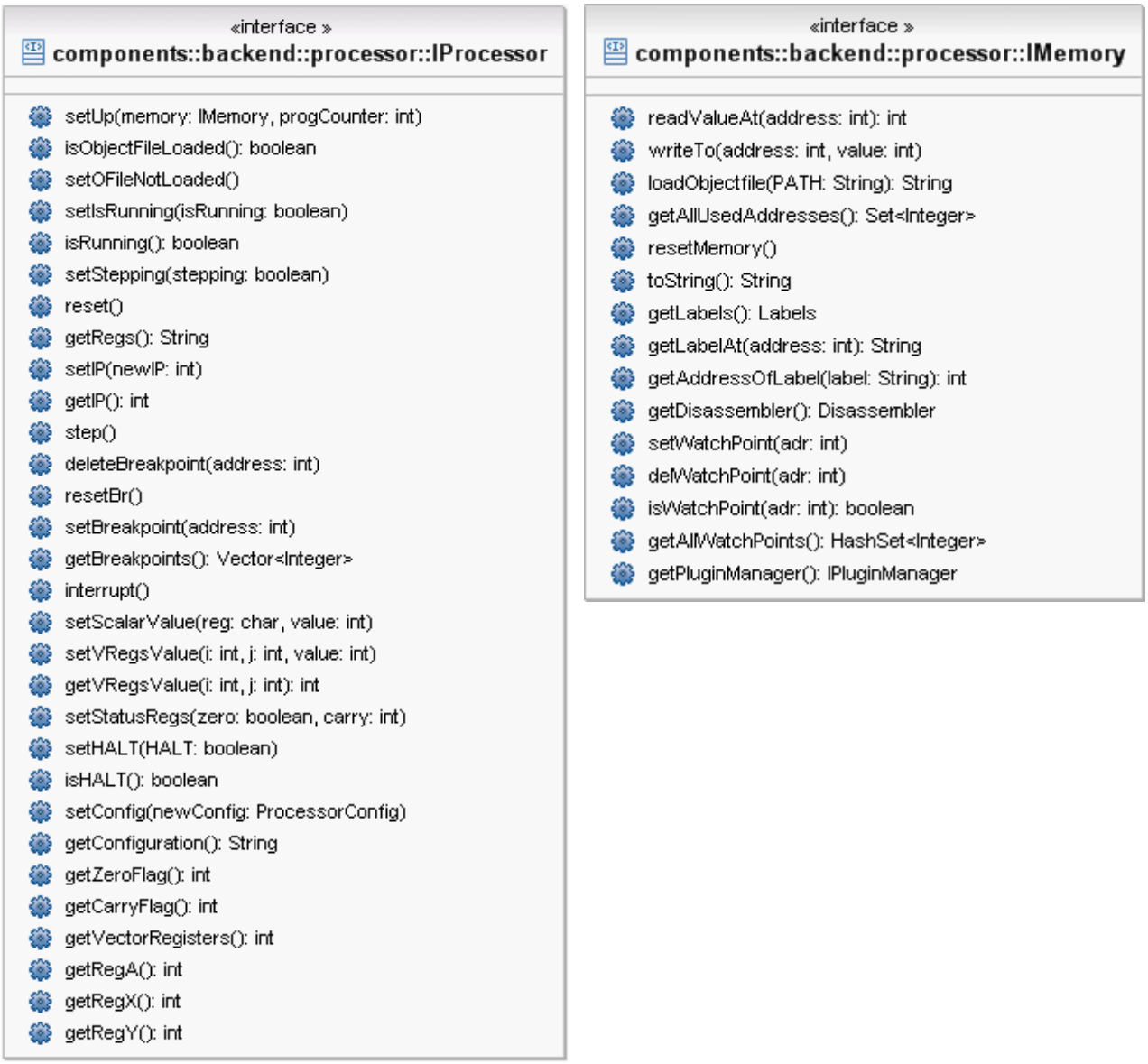
Das Interface wird von der GUI selbst implementiert und beinhaltet alle Methoden, um den aktuellen Zustand des HiCoVec (z.B. Registerinhalte) an die GUI zu schicken und darzustellen.
So bewirkt beispielsweise der Aufruf der Methode „`displayConsoleMessage(String text)`“ eine Anzeige des Strings in der Kommandozeile der GUI. Es wird jedoch zuerst überprüft, ob eine Referenz auf einen Listener vorhanden ist.

Dadurch ist gewährleistet, dass die Engine auch ohne eine grafische Oberfläche funktionsfähig ist.
Andere Komponenten, wie z.B. Prozessor und Speicher, nutzen ebenfalls die Listener-Methoden der Engine, um Nachrichten (z.B. Änderungen in Registern oder Fehlermeldungen) an die GUI zu schicken.

4.1.3) HiCoVec Realisierung (Michael Wager, Andreas Weber)

4.1.3.1) Übersicht und Klassendiagramm (Andreas Weber)

Der HiCoVec-Prozessor ist in die Komponenten CPU (**Processor**) und Speicher (**Memory**) unterteilt. Beide Klassen implementieren die Interfaces *IProcessor* und *IMemory*. Um eine möglichst hohe Modularität zu erreichen, werden von der Engine nur Referenzen auf diese Interfaces verwendet.



(Abb. 4.2 - Klassendiagramm)

Die Klasse **Processor** repräsentiert den gesamten HiCoVec-Prozessor. Dies umfasst sowohl die Skalar-, als auch die Vektoreinheit. Die entsprechenden Skalaregister A,X,Y werden in einer ENUM-Klasse dargestellt, die Vektorregister in einem Integer-Array. Der Befehlssatz des HiCoSims befindet sich, ebenfalls in Form einer ENUM-Klasse, im Paket `omponents.backend.processor.enums`.

Der Speicher wird in Form einer SortedMap in der Klasse Memory verwaltet. Besonders hervorzuheben sind hier die Methoden zum Schreiben (`writeTo(int address, int value)` bzw. zum Lesen (`readValueAt(int address)`) einer bestimmten Adresse. Außerdem unterstützt die Klasse Methoden, um Watchpoints anzulegen, abzufragen und zu löschen. Dadurch ist es möglich, eine bestimmte Speicheradresse zu markieren, bei der die Programmausführung im Falle einer Schreiboperation beendet wird. Der Aufruf der Watchpoint-Methoden erfolgt über die Engine.

4.1.3.2) Dekodierung der HiCoVec-Befehle (Michael Wager, Andreas Weber)

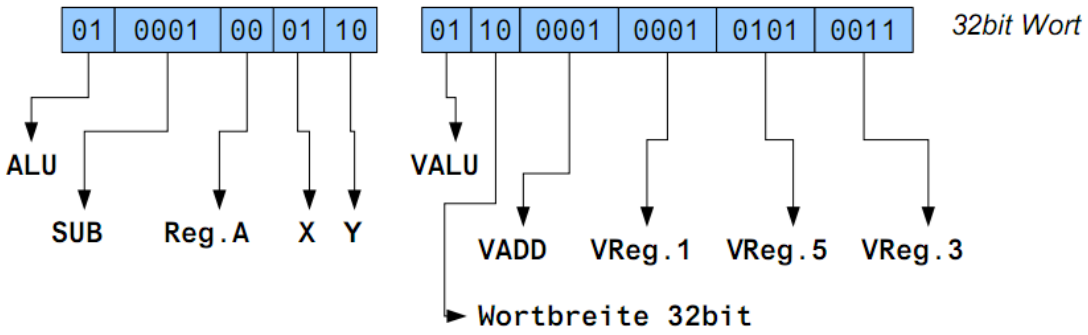
4.1.3.2.1) Einleitung (Andreas Weber)

Der HiCoVec verwendet eine Wortlänge von 32 Bit, d.h. jede Instruktion, die der Prozessor verarbeitet, hat eine Länge von genau 32 Bit. Wie bereits erwähnt verfügt die Skalareinheit des HiCoVec über 3 Register (A, X, Y), die Vektoreinheit kann in Hinsicht auf Anzahl und Breite der Vektorregister konfiguriert werden.

Die ersten 12 Bit eines Wortes werden von der Skalareinheit verwendet, die letzten 20 Bit für die Vektoreinheit. So ist es möglich, in einem Takt sowohl einen Skalar- als auch eine Vektorbefehl durchzuführen. Sollten die ersten 3 Bit eines Teilbefehls 000 sein, so führt der Prozessor eine „NOP“ durch (No Operation). Die restlichen Bits können nun für einen Direktwert verwendet werden.

Exemplarisch hier ein (paralleler) Assemblerbefehl:

SUB A, X, Y # VADD, R1, R5, R3



(Abb. 4.3 – 32Bit Wort)

Die weiteren Dekodierungen können der Diplomarbeit von Harald Manske [1] entnommen werden.

4.1.3.2.2) Dekodierung und Ausführung im Prozessor (Michael Wager)

Besonders wichtig für das Verständnis der Dekodierung und Simulation der Befehle ist folgende Funktion in
„`src/hicosim/components/backend/processor/Processor.java`“ :

```
public String decodeInstruction(int instr,boolean isdisas,boolean
stepping, int progCounter)
```

Diese Methode wird über *`processor.step()`* in *`fetchAndDecodeInstruction()`* aufgerufen, dekodiert das 32 Bit Wort „instr“ und führt dementsprechend den Befehl aus. Ist die Booleanvariable „steppping“ auf *true* gesetzt, so wird gesteppt und es sollen zusätzlich die disassemblierten Befehle ausgegeben werden.

Als erstes wird überprüft, ob der Befehl die „HALT Instruktion“ ist, d.h. der Prozessor wird angehalten. Ist dies nicht der Fall werden die Skalarkommando- und Vektorkommando-Bits gespeichert und die anhand der Bits für Skalarregister entsprechenden Skalarregister-ENUMS temporär instantiiert.

Danach werden alle MOVE Befehle (MOVA, MOV, VMOV, VMOL, VMOR) dekodiert und ggf. ausgeführt.
Ist dies ebenfalls nicht der Fall geht es weiter mit der Dekodierung der Befehle der Skalareinheit. Dies wird unter Verwendung einer switch-case Anweisung realisiert:

```
switch (this.decoder.decipherType(commandType)) {
    case LOAD_STORE:
     //code...
```

Hier kann man erkennen, dass dies unter Verwendung der Klasse **Decoder** im Paket `components.backend.processor` durchgeführt wird. Diese Klasse kümmert sich um die Überprüfung der einzelnen Bits und die Rückgabe von ENUMS, welche dann hier in der switch-Anweisung verwendet werden.

Als letztes werden nun die Vektorbefehle dekodiert. Dazu werden als erstes die Bits der Vektor - Ziel- und Quellregister geholt:

```
int rrrr = (instr>>>8) & 0xF;  //Ziel     -> 00000000
     00000000 00001111 vvvvwwww
int vvvv = (instr>>>4) & 0xF;  //Quelle1  -> 00000000
     00000000 0000rrrr 1111wwww
int wwww =  instr & 0xF;       //Quelle2  -> 00000000
     00000000 0000rrrr vvvv1111
```

Des weiteren werden Vektorbefehlstyp- und Vektorwort-Bits gespeichert.
Die Ablaufsteuerung erfolgt hier ebenfalls über eine switch-case Anweisung unter Verwendung der Klasse **Decoder**:

```
switch(this.decoder.decipherVectorType(vecCommandType))
{
            case VALU:
                //code...
```

Hier werden jetzt also noch alle Vektor-ALU Befehle und die Vektor-SHUFFLE Befehle dekodiert, ausgeführt und der Befehlszähler erhöht. Kooperationsbefehle werden somit in einem Durchgang der Methode `decodeInstruction()` ausgeführt.

4.1.3.2.3) Disassemblieren der Befehle (Michael Wager)

Der Disassembler nutzt, um Codeduplikation zu vermeiden, ebenfalls die Methode *decodeInstruction()* der Klasse **Processor**. Die Funktionsweise hier ist ziemlich genau dieselbe wie bei der Befehlsdekodierung und Ausführung:

Die Methode *decodeInstruction()* bekommt als Parameter eine Booleanvariable „*isdisas*“. Anhand dieser Variable wird immer überprüft, ob der Disassembler (src/hicosim/components/backend/processor/Disassembler.java) oder der Prozessor sie nutzt. Ist diese Variable auf true gesetzt, wird die Methode also vom Disassembler genutzt.

Dieser nutzt hier nun die gleiche Ablaufsteuerung auch unter Verwendung der switch-case Anweisungen und der Klasse Decoder.java, vermeidet jedoch die Ausführung des Befehls. Stattdessen wird ein String, welcher dem disassemblierten Befehl entspricht, erstellt und im Disassembler gespeichert, wodurch dieser disassemblierte Befehl dann über die Adresse des Befehls im Speicher abgeholt werden kann. Dieser String wird dann direkt zurückgegeben, wodurch die Methode zurückkehrt und die Ausführung eines Befehls vermieden wird.
Diese Funktion wird auch auf der GUI benötigt, um die disassemblierten Anweisungen einer geladenen Objektdatei anzuzeigen.

4.1.4) Laden der Prozessorkonfiguration (Andreas Weber)

Der HiCoVec kann durch Konfigurationen optimal an seine Aufgabe angepasst werden. Dies wird natürlich auch in der Simulation berücksichtigt. Neben der Anzahl der Vektorregister und Wörtern pro Vektorregister sind auch Shuffle- und Vektorshifteinheit und deren Breite einstellbar. Ausserdem kann die Multiplizierarithmetik für Vektor- und Skalareinheit bei Bedarf aktiviert oder deaktiviert werden.

Die gesamte Konfiguration wird in der Klasse ProcessorConfig verwaltet. Hier besteht über Getter/Setter-Methoden die Möglichkeit, einzelne Konfigurationsparameter zu ändern oder auszulesen. Der Prozessor selbst erhält bei der Instantiierung nur eine Referenz auf ein ProcessorConfig-Objekt.

Die Einstellungen für das reale HiCoVec-System werden in einer VHDL-Datei gespeichert (config.vhd). Der HiCoSim verfügt über Methoden, um diese Datei auszulesen, die gewünschte Konfiguration zu ermitteln und mit dieser Konfiguration eine neue Instanz der Klasse ProcessorConfig anzulegen. Diese Methoden sind Teil der Ein-/Ausgabe (`hicosim.component.backend.io.Configurationfile`). Es muss somit nur der Pfad zu einer gültigen config.vhd-Datei angegeben werden, um die Prozessorkonfiguration zu laden. Sollte diese nicht zur Verfügung stehen, wird sie automatisch mit Standardwerten angelegt.

4.1.5) Laden der Objektdatei (Michael Wager)

Die vom scotchAS [2] erstellte Objektdatei, welche die vom HiCoVec zur Ausführung benötigten 32 Bit Befehle enthält, hat eine ganz bestimmte Struktur, welche nun hier erläutert wird.
Wie man in Abb. 4.4 sehen kann gibt es N Sektionen mit je den Informationen zu Code, Labels, Zeilen, Breakpoints, Comments, Info und Register. Für die Simulation des HiCoVec mit dem HiCoSim sind nur folgende Sektionen relevant: Code (32 Bit), Labels, Register und Zeilen(32 Bit).
Alle anderen werden ignoriert.

In der **Codesektion** finden wir den Speicherinhalt und die Start - Indizes für die eigentlichen 32 Bit Befehle, in der **Labelsektion** die Indizes für die Labels und ihre zugehörigen Adressen und in der **Registersektion** steht der Programmzähler beim Start des Programms.

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

(Abb. 4.4 - Aufbau der Objektdatei [2])

Um das Einlesen der Objektdatei besser verstehen zu können, wird nun folgender Beispielquellcode betrachtet:

Beispiel Quellcode:

```
;########## test.asm ##########
.org 0xff
        LOAD:  .dc 0x7

.org 0x00
.start
        LD A, [0+LOAD]  ; lade A mit wert an Adresse "LOAD"=0xff=255 --> wert = 0x7
        HALT
.end
```

Assemblieren:
scotchas -o test.o test.asm

→ erstellt die Objektdatei test.o

Je 16 Bit eingelesen und ausgeben lassen (Hexadezimal):

Beispiele:

f00e	0000	0007	0012	0000	0026	0000	002b	→ 7 Sektionen; 0x12=18:
0013	0000	002c	0000	0040	0005	0000	0041	Beginn Label-Sektion
0000	0060	0006	0000	0061	0000	0063	0017	
0000	0064	0000	006b	0011	0000	006c	0000	→ 11: Beginn Code-Sektion HiCoVec 32 Bit
006f	0011	0000	0070	0000	0075	0001	0004	→ 11: Beginn Code-Sektion HiCoVec 32 Bit
0000	00ff	4c4f	4144	0001	0000	00ff	0002	
0000	00ff	0003	0000	0000	0004	0000	0000	
0005	0000	0000	0006	0000	0000	0007	0000	
0001	0001	003a	0006	206c	6164	6520	4120	\
6d69	7420	7765	7274	2061	6e20	4164	7265	\
7373	6520	224c	4f41	4422	3d30	7866	663d	> Kommentare, Infosektion etc.
3235	3520	2d2d	3e20	7765	7274	203d	2030	/
7837	564f	4443	4100	0000	0000	0000	0000	/
0000	0000	0000	0000	0000	00FF	0000	0007	→ BSP: je 2*16=32 Bit,Adresse:0xFF,wert: 7
0000	0000	8100	00ff	2800	0000	0000		

In einer Schleife werden jetzt also die N Sektionen immer wieder durchlaufen.
(Siehe src/hicosim/components/backend/io/ObjectfileReader.readUnsignedShortsAndGetMemory())

Als erstes wird die Anzahl der Sektionen geholt, siehe '7' in erster Zeile.

```
idx = 3 + (5 * secNum); //secNum beim ersten Mal == 0
secType    = shortsRead.get(idx); //idx also beim ersten Durchlauf == 3
```

0: An 3. Stelle steht eine 18, welche den Beginn der Label-Sektion für den HiCoVec darstellt.
Hier werden dann die startAddressIndexes für die Labels gespeichert.

1: Beim nächsten Durchlauf ist idx=8. An 8. Stelle steht eine 19, welche den Beginn der Zeilen-Sektion für den HiCoVec darstellt. Diese wird jedoch ignoriert und die Schleife beginnt von neuem.

2: und 3: In den nächsten 2 Durchläufen ist idx 13 und 18, dort stehen je der Beginn von Sektion Comments und Sektion Info, welche beide ignoriert werden.

4: Jetzt ist idx = 23 und an 23. Stelle steht eine 23, welche den Beginn der Register-Sektion darstellt. Hier wird nur die Startadresse des Programmzählers gespeichert.

5: idx=28. An Stelle 28 steht eine 17, welche nun die Sektion des reinen Maschinencodes, also die 32Bit Befehle, für den HiCoVec darstellt. Es werden die start- und end-Indizes der Befehle gespeichert und der Memory wird auch hier schon gefüllt.

Bei Durchlauf **6** ist idx = 33 und dort steht nochmal eine 17 welche die 2. Codesektion darstellt, da im Beispielquellcode ja mit .org zwei Codesektionen erstellt wurden.

Da wir jetzt die Start- Indizes der relevanten Informationen gespeichert haben, wird die Datei erneut eingelesen, diesmal jedoch je 8Bit. (Siehe ..Reader.readUnsignedBytes())
Hier werden nun anhand der Indizes erst die 32-Bit Befehle geholt und in den Speicher geschrieben, danach der Programmzähler geholt und gespeichert und zuletzt noch die Labels.
Lässt man sich dies dann ausgeben, kann man erkennen dass alle für die Simulation des HiCoVec relevanten Informationen nun vorhanden sind:

Programzähler beim Start: 0
Speicher nach Einlesevorgang: {0 = -2130706177, 1 = 671088640, 255=7}

→ „-2130706177“ representiert hier die Ladeoperation als 32 Bit Integer
→ „671088640“ == HALT
→ „7“ von der .dc Anweisung an Adresse 255

Codesektion #1 AB ADRESSE: 255=0xff
00000000 00000000 00000000 00000111 == 7
Codesektion #2 AB ADRESSE: 0
10000001 00000000 00000000 11111111 == -2130706177 == LD
00101000 00000000 00000000 00000000 == 671088640 == HALT

LABEL: 'LOAD' an Adresse: 255

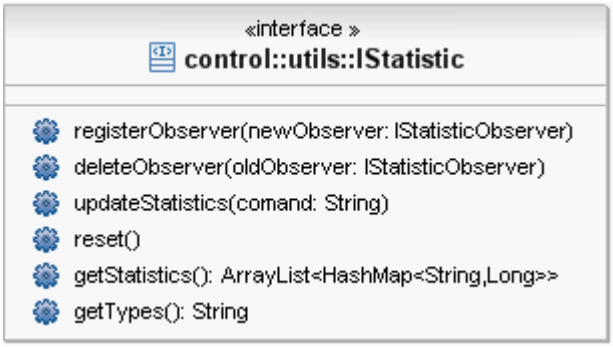
All dies wird gestartet in der Klasse src/hicosim/components/processor/Memory in
public String loadObjectfile(String PATH)

Nach erfolgreichem Einlesen wird nun in der Klasse *Memory* dem Prozessor der Programmzähler und der Speicher übergeben, wonach er bereit zur Ausführung ist.

4.1.6) Statistiken (Andreas Weber)

Um ein Programm in Hinsicht auf Geschwindigkeit und Effizienz zu beurteilen, sind aussagekräftige Statistiken ein unverzichtbares Hilfsmittel. Die Statistiken des HiCoSim werden im Hintergrund gesammelt, auf der GUI angezeigt und können auf Wunsch abgespeichert werden.

Statistiken werden in der Klasse **Statistics** (Interface: *IStatistics*) verwaltet und vom Prozessor angelegt.



(Abb. 4.8 - IStatistic)

Im Wesentlichen werden 2 Informationen genutzt: Der Typ und der Befehl. Führt der Prozessor beispielsweise den Befehl „ADD“ aus, erfolgt der Aufruf der Methode „updateStatistics(„ADD“). Um eine möglichst einfache Erweiterung zu ermöglichen, wird der Datentyp String verwendet. Die Statistik ermittelt nun den dazugehörigen Typ (in diesem Fall: „SkalarALU“) und erhöht die Anzahl für den Befehl. Die Zuordnung zwischen Typ und Befehl ist festgelegt, kann jedoch sehr einfach geändert und erweitert werden. Nicht zugeordnete Befehle erhalten standardmäßig den Typ „OTHER“.
Beide Werte benutzen als Datentyp eine HashMap, was ein effizientes Suchen und Einfügen ermöglicht. In einer überlagerten `toString()`-Methode ist auch eine textbasierte Ausgabe der schon gesammelten Statistiken möglich. Dazu werden diese erst der Häufigkeit nach sortiert und dann als String zurückgegeben.
Um eine Echtzeit-Anzeige, wie auf der GUI dargestellt, zu ermöglichen unterstützen die Statistiken das Observer-Pattern. Dazu ist die Funktionalität implementiert, eine oder mehrere Referenzen des Typs „*IStatisticObserver*“ aufzunehmen und diese bei einer Änderung zu benachrichtigen. Die weitere Verarbeitung kann dann von den jeweiligen Observern übernommen werden.

4.2) Frontend (Daniel Obermüller, Andreas Weber)

4.2.1) Einleitende Worte (Daniel Obermüller)

Wie bereits das Backend wurde auch das Frontend komplett in JAVA, unter Verwendung von Swing realisiert. Die wichtigsten Gründe hierfür waren einerseits, dass durch diverse Projekte im Rahmen des Studiums bereits eine gewisse Expertise in JAVA vorhanden war und andererseits beispielsweise Jython möglicherweise unvorhersehbare Probleme mit sich gebracht hätte.

Das Ziel war eine möglichst :

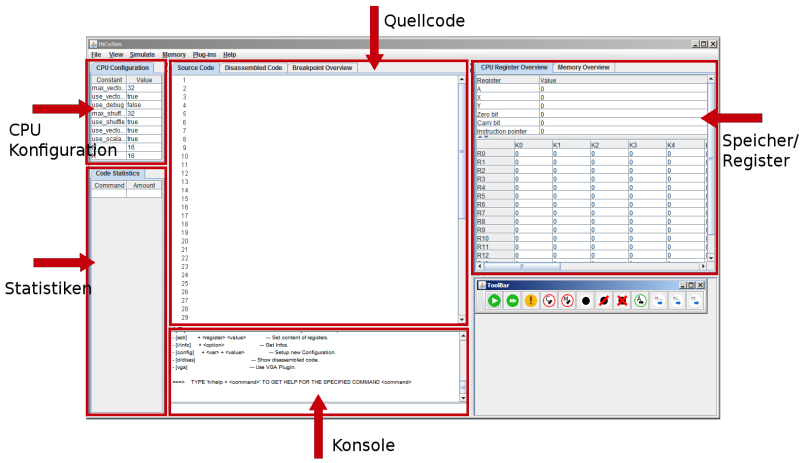
- intuitive,
- variable,
- erweiterbare,
- austauschbare
- und benutzerfreundliche

Benutzeroberfläche zu erstellen.

In den folgenden Abschnitten wird ein Überblick über die diversen Komponenten der grafischen Benutzeroberfläche gegeben und deren Funktionsweise erläutert.

Komponenten:

- Layout (Verschachtelung von "'JSplitPane'-Objekten")
- Textfelder („JTextAreaSourceCode“, „JTextAreaDiassembledCode“ und "JTextPaneConsole")
- Tabellen ("JTableVectorRegisterOverview", "JTableScalarRegisterOverview", "JTableCpuConfiguration" und "JTableCodeStatistics")
- Nachrichtendialoge ("CustomJDialog" und "JOptionPane")
- Speicherübersicht ("MemoryOverview")
- VGA-Plug-In
- ToolBar



(Abb. 4.9 – Grafische Benutzeroberfläche)

4.2.2) Layout (Daniel Obermüller)

4.2.2.1) Realisierung (Daniel Obermüller)

Um die variable Fensteranordnung zu ermöglichen und diese weitestgehend intuitiv zu halten, wurde das Hauptfenster „MainWindow“, welches ein abgeleitetes "JFrame" ist, aus einer Vielzahl verschachtelter „JSplitPane'-Objekten“ aufgebaut. Diese wurden in ein "BorderLayout" integriert. Da das "'JSplitPane'-Objekt" einen "Divider" besitzt und dieser via Maus verschoben werden kann, ist es dem Benutzer möglich die Benutzeroberfläche relativ gut seinen Bedürfnissen anzupassen. Es ist beispielsweise möglich jeden einzelnen Bereich komplett auszublenden, aber auch diesen lediglich zu verkleinern oder vergrößern.

Da das Frontend über ein „'Engine'-Objekt“ auf das Backend zugreift und somit keine direkte Abhängigkeit zwischen den zwei Komponenten besteht, kann das Backend jederzeit ausgetauscht werden. Allerdings nur sofern die Klasse "Engine" weiterhin alle Methoden, die das Frontend nutzt, beibehält. Um zu garantieren, dass das Frontend ohne größere Modifikationen ersetzt werden kann, wurde ein in JAVA gängiges Mittel dafür verwendet, "Interfaces". "Interfaces" deklarieren lediglich Methodenköpfe und keine Methodenrümpfe. Anders gesagt, sie zeigen dem Entwickler die Syntax einer Methode mitsamt Methodenname und aller Parameter. Das Backend beherbergt ein „'Interface'-Objekt“ "IListenerFrontend", welches Methoden des Frontends deklariert und über dieses wird auf Methoden der grafischen Benutzeroberfläche zugegriffen. Das Frontend muss dieses implementieren, d.h., dass im Frontend die Methodenrümpfe ausgearbeitet werden müssen. Durch diesen indirekten Zugriff des Backends auf das Frontend ist es möglich, dass das Frontend durch ein anderes ausgetauscht werden kann. Allerdings muss dafür das "Interface" "IListenerFrontend" im neuen Frontend implementiert sein. Jede Aktualisierung einer Komponente des Frontends wird durch das Backend ausgeführt.

Beispiel:

Ein Wert eines Vektorregisters wird geändert. Die Vektorregister werden innerhalb des Backends auf ein privates zweidimensionales "'Integer-Array'-Objekt" abgebildet. Aufgrund der Tatsache, dass das Frontend lediglich die Visualisierung der Vektorregister übernimmt und keine Kopie des "'Integer-Array'-Objekts", welches die Vektorregister darstellt, besitzt, muss eine Aktualisierung des entsprechenden Vektorregisters durch das Backend erfolgen. Diese Aktualisierung findet über einen Methodenaufruf des "'IListenerFrontend'-Objekts" im Backend statt.

4.2.2.2) Erweiterbarkeit (Daniel Obermüller)

Des Weiteren kann die Benutzeroberfläche jederzeit ohne größeren Aufwand erweitert werden, indem ein bereits vorhandenes "'JSplitPane'-Objekt" ein oder mehrere neue „'JSplitPane'-Objekte" aufnimmt. Außerdem wäre es möglich an geeigneter Stelle in das bereits vorhandene "BorderLayout" neue Komponenten aufzunehmen. Darüber hinaus könnte man das verwendete "BorderLayout" durch ein variableres Layout austauschen, beispielsweise das "GridBagLayout". Um spätere Änderungen des Layouts zu garantieren, wurde das Initialisieren jeder Grafikkomponente durch eine separate Methode, welche die gewünschte Komponente zurück gibt, realisiert und somit weitreichende Modularität gewährleistet.

Beispiel:

```
/**
* This method initializes the JMenuItem "jMenuItemExit".
*
* @author Daniel Obermueller 9th April 2009
* @return javax.swing.JMenuItem
*/
        private JMenuItem getJMenuItemExit() {
                if (jMenuItemExit == null) {
                        jMenuItemExit = new JMenuItem("Exit");
                        jMenuItemExit.setPreferredSize(new Dimension(55, 20));
                        jMenuItemExit.setMnemonic('X');
                        jMenuItemExit.setAccelerator(KeyStroke.getKeyStroke(
                                        java.awt.event.KeyEvent.VK_Q,
                                        java.awt.Event.CTRL_MASK));
                        ...
                                        {
                        System.exit(JFrame.DISPOSE_ON_CLOSE);
                                                }
                                        });
                }
                return jMenuItemExit;
        }
```

In dieser privaten Methode wird das „'JMenuItem'-Objekt" „jMenuItemExit", welches einen Menüpunkt der Menüleiste darstellt, initialisiert.

Zunächst wird überprüft ob das Objekt bereits initialisiert wurde. Falls es noch nicht initialisiert wurde, wird eine neues „'JMenuItem-Objekt“ erzeugt. Diesem Objekt werden Attribute und Methoden zugewiesen. Anschließend wird das Objekt zurückgeben. Falls es bereits initialisiert wurde, wird das vorhandene Objekt zurück gegeben.

Durch die Verwendung solcher Methoden sind die Komponenten nicht direkt vom Layout abhängig und können jederzeit in neue eingebettet werden.

4.2.3) Textkomponenten (Daniel Obermüller)

4.2.3.1) Konsole (Daniel Obermüller)

Um eine angemessene Konsolendarstellung für Fehlermeldungen und reguläre Ausgaben zu ermöglichen, musste eine Möglichkeit gefunden werden selbige visuell klar voneinander zu trennen. Dies wurde durch die Darstellung von Nachrichten in unterschiedlichen Farben realisiert; hier durch rot für Fehler- und schwarz für normale Meldungen.

Die Komponente, welche diese Methode bereitstellt, ist ein abgeleitetes "'JTextPane'-Objekt" „CustomJTextPane“ und verfügt über die Methode

```
public void append(String message, Color color).
```

Das Backend ruft via "CallBack" die Methoden

```
public void displayConsoleMessage(String message)

public void displayConsoleError(String message)
```

für die entsprechenden Meldungen auf.

4.2.3.2) Textfelder für Quell- und Disassemblierten Code (Daniel Obermüller)

Beide Textfelder basieren auf der von "JTextArea" abgeleiteten Klasse „JTextAreaWithLineNumbering“. Diese besitzt Methoden zur Darstellung von Zeilenindizes, Breakpoints und eine weitere, welche für das Code-Highlighting genutzt wird.

Zeilennummerierung:

Involviert an der Anzeige der Zeilen sind die folgenden zwei überladenen Methoden

```
public Insets getInsets(Insets insets)

public void paintComponent(Graphics g).
```

Die Methode, welche die "Insets" zurück gibt, ist verantwortlich für den nötigen Abstand in der Spalte mit den Nummerierungen und Breakpoints.

Die andere hingegen zeichnet die Zeilennummern und die Breakpoints für den momentan angezeigten Ausschnitt in den Bereich der "Insets". Hierbei wird aus der Höhe der Komponente sowie den Metriken der verwendeten Schriftart die benötigten Indizes kalkuliert und überprüft ob Breakpoints angezeigt werden müssen.
Die Breakpoints werden in Form von ganzzahligen Werten, welche Zeilen repräsentieren, innerhalb eines „'LinkedList<Integer>' -Objekts“ abgelegt.

Zeilenmarkierung:

Die Methode

```
public void highlightLine(int index)
```

markiert die übergebene Zeile in der Farbe rot. Hierbei wird zunächst der Text indiziert, d.h., dass der Text anhand der Zeilenumbruchzeichen in Abschnitte eingeteilt wird. Diese Abschnitte wiederum repräsentieren Zeilen.

Anschließend wird die übergebene Zeile unter Verwendung der geerbten Methode „getHighlighter().addHighlight(...)“ und dem erforderlichen Abschnitt markiert und der Fokus auf diese gelegt. Hierzu wird der Textcursor an den Anfang der entsprechenden Zeile gesetzt.

4.2.4) VGA-Plug-In (Daniel Obermüller)

Das VGA-Plug-In ist eine von der Klasse "JPanel" abgeleitete Klasse „VGAPlugin“. Diese implementiert das abstrakte "Interface" "KeyListener", welches Methoden zur Verarbeitung von Tastatureingaben bereitstellt. Falls die linke, rechte oder untere Pfeiltaste gedrückt wird, werden bestimmte Speicheradressen mit vorher definierten Werten beschrieben. Z.B. wird durch Druck der linken bzw. rechten Pfeiltaste, sofern das Spiel [ScotchRace] geladen und ausgeführt wird, das dargestellte Auto in die entsprechende Richtung gelenkt.

Die Darstellung des Bildes wird durch Verwendung eines „'BufferedImage'-Objekts“ im Zusammenhang mit einem „'WritableRaster'-Objekt“ realisiert.
Die Pixel des Bildes werden in einem zweidimensionalen "'Integer-Array'-Objekt" abgelegt. Das zugrunde liegende Farbmodell ist das "'ARGB'-Modell". Hierbei werden die Transparenz und die Farben, welche sich aus Kombinationen von Rot, Grün und Blau zusammensetzen, berücksichtigt.

Um beispielsweise den Pixel in der linken oberen Ecke einzufärben, muss das Feld [x=0] [y=0] des "'Integer-Array'-Objekts", welches die Pixel beherbergt, mit der gewünschten Farbe als "'Integer'-Repräsentation" überschrieben werden. Die Klasse "Color" besitzt 16 statische Variablen, welche grundlegende Farben darstellen und kann außerdem jeden beliebigen Farbwert annehmen, indem ein neues Objekt (mit den nötigen Rot-, Grün- und Blaufarbwerten) dieser Klasse instantiiert wird.

Um unnötige Aktualisierungen des Bildes zu vermeiden, wird die Methode

```
public void setIgnoreRepaint(boolean ignoreRepaint)
```

beim initialisieren des VGA-Plug-Ins aufgerufen.

Das Aktualisieren des Bildes übernimmt ein "Thread", der dauerhaft alle Speicheradressen, die für das VGA-Plug-In von Interesse sind, ausliest und die entsprechenden Pixel mit den jeweiligen Farben überschreibt. Nach Beendigung eines Durchlaufs über alle erforderlichen Speicheradressen wird manuell eine Aktualisierung des Bildes vorgenommen. Hierbei werden die Pixel in Form des "'Integer- Array'-Objekts" an das "'WritableRaster'-Objekt" übergeben und anschließend das "'BufferedImage'-Objekt" neu gezeichnet. Hierbei ist zu berücksichtigen, dass das Bild nur aktualisiert wird sofern Pixel verändert wurden. Falls kein Pixel verändert wurde, wird keine Aktualisierung vorgenommen. Diese Logik übernimmt das "'BufferedImage'-Objekt" voll automatisch ohne Zutun des Programmierers.

4.2.5) Tabellen (Daniel Obermüller)

4.2.5.1) Skalar- und Statusregister (Daniel Obermüller)

Die Visualisierung der Skalar- und Statusregister wurde durch die Verwendung eines "'JTable'-Objekts" realisiert. Dieses Objekt "jTableScalarCpuRegisterOverview" hat zwei Spalten und sechs Zeilen. Durch Verwendung der Klasse "CustomDefaultTableModel", welches eine abgeleitete Klasse der Superklasse "DefaultTableModel" ist, ist es möglich eine Editierung der ersten Spalte zu untersagen. Hierzu musste die geerbte Methode

```
public boolean isCellEditable(int rowIndex, int columnIndex)
```

entsprechend überschrieben werden.

Da aus gewissen Gründen die Spalten- sowie Zeilenköpfe unter bestimmten Umständen nicht angezeigt werden, mussten diese selber erstellt werden.
Hierzu wurde zuerst eine neue Zeile an erster Stelle eingefügt und diese durch die oben angeführte Methode nicht editierbar gemacht. Allerdings wich die Optik nun stark von den Standardspaltenköpfen ab, da die einzelnen Zellen weiß waren. Um die selbst erstellten Spaltenköpfe den regulären Spaltenköpfen der Klasse "JTable" anzugleichen, wurde ein eigener "TableCellRenderer" erstellt.

Diese Klasse "ColoredTableCellRendererScalarCpuRegisterOverview" basiert auf der Klasse "JTextArea", von welcher sie abgeleitet ist, und implementiert das "Interface" "TableCellRenderer".

Die vom „Interface“ „TableCellRenderer“ vorgegebene Methode muss wie folgt überschrieben werden:

```
/**
 * This method is used to set different background colors of JTable cells.
 *
 *
 * @param JTable table
 * @param Object value
 * @param boolean isSelected
 * @param boolean hasFocus
 * @param int row
 * @param int column
 * @return Component
 * @author Daniel Obermueller 18th June 2009
 */
public Component getTableCellRendererComponent(JTable table, Object value,
                boolean isSelected, boolean hasFocus, int row, int column) {
        if (row == 0)
                setBackground(new Color(0xffeeeeee));
        else
                setBackground(Color.WHITE);
        setText(value.toString());
        return this;
}
```

In dieser Methode werden die geerbten Methoden der Klasse „JTextArea“, auf welcher diese Klasse "TableCellRenderer" basiert, genutzt. Der Hintergrund der Zellen in der ersten Reihe wird in dem Grauton der regulären Spaltenköpfe der Klasse "JTable" dargestellt; die restlichen Zellen hingegen in weiß.

Die Werte der Register können direkt über das Frontend modifiziert werden. Dazu muss nur per Doppelklick in die zu bearbeitende Zelle geklickt werden und ein dezimaler, oktaler, hexadezimaler oder binärer Wert eingegeben werden und mit der Eingabetaste bestätigt werden. Das Frontend verfügt über eine Methode, welche den eingegebenen Wert in das Dezimalsystem konvertiert und anschließend über das "'Engine'-Objekt" im Backend den Wert im entsprechenden Register überschreibt.

4.2.5.2) Vektorregister (Daniel Obermüller)

Wie auch bei den Skalar- und Statusregistern wurde die Darstellung der Vektorregister in Form eines "'JTable'-Objekts" realisiert. Dieses Objekt "jTableVectorCpuRegisterOverview" hat eine variable Anzahl an Spalten und Zeilen. Das zugrunde liegende "TableModel" ist ein Objekt der Klasse "CustomAbstractTableModelVectorCpuRegisterOverview" und stellt

Methoden zum Ausblenden und wieder Einblenden von Spalten und Reihen bereit. Außerdem ermöglicht das Implementieren dieses Objekts die Möglichkeit Spalten dynamisch während der Laufzeit hinzuzufügen und zu entfernen. Durch Verwendung der Klasse "CustomAbstractTableModelWithEditableCells", welche eine abgeleitete Klasse der Superklasse "CustomAbstractTableModelVectorCpuRegisterOverview" ist, ist es möglich bestimmten Zellen die Eigenschaft zu nehmen editierbar zu sein [siehe Kapitel 4.5.1]. Die Werte der Vektorregister können direkt über das Frontend modifiziert werden. [siehe Kapitel 4.5.1]

4.2.5.3) Code-Statistiken (Andreas Weber)

Statistiken sind ein wichtiges Hilfsmittel, um [HiCoVec]-Programme hinsichtlich ihrer Geschwindigkeit und Effizienz zu analysieren und zu vergleichen. Die grafische Benutzeroberfläche bietet dem Nutzer daher stets einen Überblick über die gesammelten Statistiken, die bei Bedarf auch abgespeichert werden können.

Die Statistiken werden in Echtzeit bereitgestellt, von der grafischen Benutzeroberfläche entsprechend aufbereitet und im Programmfenster angezeigt. Um dies zu ermöglichen, implementiert das Frontend das "Interface" „IStatsObserver“. Eine entsprechende Methode im Backend benachrichtigt bei einer Änderung des Statistikwerts automatisch alle "Observer" durch einen Aufruf der Methode

```
public boolean updateStatsObserver(String command, String type)
```

Um diese Informationen zu erhalten, registriert sich die grafische Benutzeroberfläche als Beobachter. Das Frontend übernimmt nun die weitere Verarbeitung. In dem für die Anzeige zuständigen "'JTable'-Objekt" wird zunächst überprüft, ob der entsprechende Befehlstyp schon erfasst wurde. Falls der Befehlstyp bereits vorhanden ist, wird die entsprechende Zelle selektiert; falls nicht wird eine neue Zeile hinzugefügt.

Die Ausführungsanzahl des Typs wird erhöht und anschließend wird analog dazu der übergebene Befehl gesucht, sowie dessen Wert inkrementiert.

Um eine bessere Übersicht zu gewährleisten, werden die Statistiken der Häufigkeit nach sortiert und entsprechend ausgegeben. Dies geschieht direkt beim Einfügen der Werte in die Tabelle.

4.2.5.4) CPU-Konfiguration (Daniel Obermüller)

Die Darstellung der CPU-Konfiguration wurde in Form eines "'JTable'-Objekts" realisiert. Diesem Objekt "jTableCpuConfiguration" wird beim Starten der Applikation die CPU-Konfiguration durch das Backend übergeben. Die CPU-Konfiguration lässt sich während der Laufzeit durch den Benutzer modifizieren. Die Bearbeitung erfolgt hierbei analog zu allen anderen editierbaren Tabellen [siehe Kapitel 4.5.1]. Allerdings muss nach einer Änderung der Parameter "N" (Anzahl der Vektorregister) und "K" (Anzahl der Vektor-scheiben) die Anzeige der Vektorregister entsprechend aktualisiert werden.

Hierbei kommen die Methoden des in [Kapitel 4.5.2] erwähnten "CustomAbstractTableModel-VectorCpuRegisterOverview" zu tragen. Diese Aktualisierung nimmt das Backend durch einen indirekten Aufruf des Frontends über das Objekt "IListenerFrontend" vor.

4.2.5.5) Speicher (Andreas Weber)

Um eine aufschlussreiche Simulation eines [HiCoVec]-Programms zu ermöglichen, ist es nötig, jederzeit den aktuellen Stand des Systems überblicken zu können. Neben einer Registeranzeige kann auch die Speicherbelegung komfortabel überwacht werden.
Die Speicherübersicht (Memory Overview) stellt dem Benutzer Funktionen bereit, um den überwachten Speicherbereich mit Hilfe von "'JButton'-Objekten" zu ändern. Außerdem ist es möglich über ein "'JTextField'-Objekt" eine direkte Eingabe zu tätigen.

Der Inhalt der Speicheradresse wird in dezimaler, hexadezimaler und binärer Form angezeigt. Außerdem erfolgt eine Ausgabe des disassemblierten Maschinenbefehls. Diese Felder sind editierbar, d.h. eine direkte Eingabe eines neuen Wertes ist möglich. Jede Speicheradresse lässt sich über ein "'ComboBox'-Objekt" auch als "WatchPoint" markieren.

Bei der Implementierung wurde besonders darauf geachtet, die Eingabemöglichkeit so robust wie möglich zu gestalten. So akzeptiert das "'JTextField'-Objekt" als gültige Parameter nicht nur eine dezimale Adresse, sondern wandelt auch hexadezimale Werte sowie ganze Wertebereiche automatisch um. Bei der Eingabe von Text wird in dem "'Engine'-Objekt" überprüft, ob die Eingabe einem "Label" entspricht und die Übersicht dementsprechend angepasst.

Die Speicherübersicht wurde als Plug-In realisiert. Beim Hinzufügen einer weiteren Speicherübersicht wird eine neues Objekt der Klasse "WatchMemPlugin" instantiiert. Dieses registriert sich beim "Plug-In-Manager" des [HiCoSim]. Durch den "Observer-Mechanismus" wird die Ansicht sofort aktualisiert, sobald sich der Inhalt einer Speicherzelle im beobachteten Bereich ändert.
Bei Eingabe einer neuen Adresse oder wenn der Bildschirminhalt geblättert wird, teilt die Speicheranzeige dem "Plug-In-Manager" die erste und letzte angezeigte Adresse mit. So wird garantiert, dass die grafische Benutzeroberfläche nur Aktualisierungen für die momentan sichtbaren Adressen erhält.

4.2.6) Öfnen-/Speichern Dialoge (Daniel Obermüller)

Durch Verwendung von Dialogen wird die Benutzerfreundlichkeit erhöht, und mögliche Fehlerquellen ausgeschlossen, so sind bei sämtlichen Dialogen nur Dateien mit unterstützen Dateiendungen auswählbar. Wird eine Datei ohne Endung angegeben, so wird diese automatisch angefügt. Wird bei einem Speichern-Dialog eine bereits existierende Datei ausgewählt, so erscheint ein äquivalenter "Override-Dialog".

Die Dialoge „Open source code“, „Open object file“, „Save source code as“, sowie „Save statistics as“ sind allesamt "'JFileChooser'-Dialoge" mit analogem Aufbau.

Weiterhin gibt es noch den „Hide-“ bzw. „Show-Vector-Registers-Dialog", welcher in der Vektorregisterübersicht Zeilen bzw. Spalten ein- oder ausblenden kann. Es ist ein selbst erstellter Dialog, basierend auf einem "'JPanel'-Objekt" mit zwei Textfeldern um ein Intervall einzugeben (Start und Ende), zwei "'RadioButton'-Objekten um zwischen Zeilen und Spalten zu wählen und zwei "'JButton'-Objekte" zum Bestätigen und Abbrechen des Vorgangs.

4.2.7) Toolbar (Daniel Obermüller)

Um dem Benutzer den Umweg über die Menüleiste zu ersparen, werden die gängigsten Funktionen wie z.B. die Simulation zu starten, "BreakPoints" zu setzen usw. in einer Toolbar mit prägnanten Symbolen bereitgestellt. Die Toolbar ist ein "'JToolBar'-Objekt", dass im "BorderLayout" unterhalb der Menüleiste eingebettet wurde.

Die Toolbar kann nicht ausgeblendet werden, kann aber aus dem "BoderLayout" herausgelöst werden und wird dann in einem eigenen "'JFrame'-Objekt" dargestellt. Beim Schließen dieses Objekts gliedert sich die Toolbar wieder unter der Menüleiste ins Layout ein.

Um die Toolbar zu erweitern muss die private Klasse "ToolBar" innerhalb des Objekts der Klasse "MainWindow" modifiziert werden.

Die Symbole der Toolbar befinden sich im Ordner "icons" unter „../hicosim/bin/components/frontend/“.
An dieser Stelle müssen auch neue Symbole in der Größe 20x20 Pixel im "'GIF'-Format" eingefügt werden um in der Toolbar genutzt werden zu können.

In der Klasse "ToolBar" muss ein privates statisches "'String'-Objekt" initialisiert werden, welche die spätere Funktion des Symbols beschreibt.

Beispiel:

```
static final private String NEWACTION = "Neue ToolBar Aktion";
```

Als nächstes muss innerhalb der nachfolgenden Methode ein "'NavigationButton'-Objekt" angefügt werden und an das "'JToolBar'-Objekt" übergeben werden.

Daher muss der Eintrag wie folgt aufgebaut sein:

```
protected void addButtons(JToolBar toolBar) {
        JButton button = null;

        //makeNavigationButton(FILE NAME, ACTION, TOOLTIP, ALTERNATIVE
        TEXT)
        button = makeNavigationButton("new_action", NEWACTION, "New",
                "New");
        toolBar.add(button);
```

Zuletzt wird noch die auszuführende Aktion definiert, die beim Klick auf das Symbol ausgeführt werden soll.

Dies geschieht in der Methode "actionPerformed" durch Einfügen der gewünschten Funktion an geeigneter Stelle.

Beispiel:

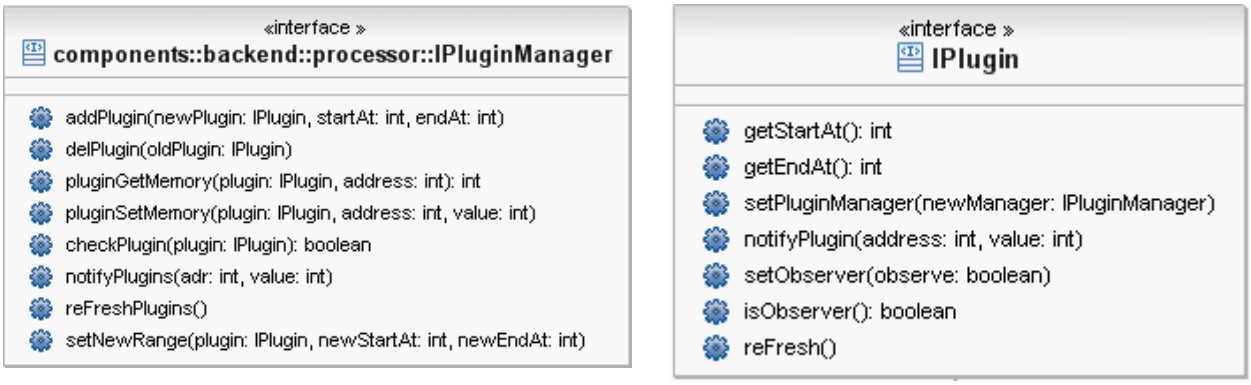
```
else if (NEWACTION.equals(cmd)) {          //New Action
        engine.doNewAction();
}
```

Durch Modifikationen an den aufgeführten Stellen ist es möglich die Toolbar um neue Funktionen zu erweitern.

4.3) Plugins (Andreas Weber)

Da der HiCoVec erst am Anfang seiner Entwicklung steht, ist es wahrscheinlich, dass weitere Ergänzungen realisiert werden. Der HiCoSim stellt hierfür einen Plugin-Mechanismus zur Verfügung, um neue Plugins mit möglichst geringen Änderungen am Code umsetzen zu können.

Zur Verwaltung der Plugins dient der **PluginManager** (Schnittstelle: *IPluginManager*). Er kann ein oder mehrere Referenzen auf Objekte verwalten, die das Interface *IPlugin* implementieren.



(Abb. 4.5 - Plugins)

Der Pluginmanager selbst enthält Referenzen auf die Engine sowie auf den Speicher. Jedes Plugin muss den Bereich, auf den Lese-/ oder Schreiboperationen durchgeführt werden sollen, definieren. Es kann auch festgelegt werden, ob das Plugin als Observer auftreten soll.

Das Plugin selbst benötigt nicht zwingend eine Referenz auf die Engine. Speicherzugriffe erfolgen über den PuginManager.

4.3.1) VGA-Plug-In (Michael Wager)

4.3.1.1) Einleitung (Michael Wager)

Um auch bei der Simulation des HiCoVec-Prozessors eine grafische Darstellung zu ermöglichen haben wir uns entschieden ein VGA-Plugin, basierend auf der Funktionsweise des VGA-Moduls der SCOTCH-Gruppe[2], zu implementieren.

Abb. 4.6 zeigt hier einen Screenshot des Orginal ScotchRace bei gewonnenem Spiel und zum direkten Vergleich Abb. 4.7 den Screnshot des ScotchRace auf dem HiCoSim VGA-Modul. Daran kann man erkennen, dass die Funktionsweise exakt übernommen wurde und man somit wie auf der ScotchHardware[2] einen vollständigen Grafikmodus sowie einen ASCII-Modus um ASCII Zeichen darzustellen, zur Verfügung hat.

(Abb. 4.6 - Screenshot Orginal ScotchRace [2])

(Abb. 4.7 - Screenshot ScotchRace [2] auf HICOSIM VGA-Modul)

4.3.1.2) Funktionsweise (Michael Wager)

Es gibt 3 relevante Speicherbereiche, einen für reinen Grafikmodus(0x2000-0x3F3F), einen für ASCII Modus(0x4000-0x43E7)und einen um die Pixelfarben (0-15 für 16 Farben) hineinzuschreiben.
Im reinen Grafikmodus kann man direkt die Pixel anhand eines 32 Bit Datenwortes hineinschreiben. Ein 32 Bit Datenwort wird hier als 8 Pixel ã 4 Bit interpretiert, d.h. man kann durch dieses eine Datenwort 8 Pixel zeichnen, je 4 Bit für einen Pixel, also 16 Farben.

Durch den ASCII-Modus ist es möglich, direkt ASCII-Zeichen bzw. deren Codes hineinzuschreiben, welche dann dementsprechend direkt in den Grafikspeicherbereich geschrieben werden.
Dort überwacht die Klasse ProcessASCII.java den ASCII Speicherbereich und erstellt bei Schreibzugriff auf diesen Bereich acht 32 Bit Datenwörter um ein 8x8 Pixel Zeichen darstellen zu können.

Will man zB ein 'A' mit der Vordergrundfarbe blau auf die Hintergrundfarbe weiß auf den VGA Screen schreiben, dies entspricht: .dc “A ”(0x3,0x0) im Assemblercode], so schreibt man folgendes 32 Bit Wort in den ASCII-Speicherbereich:

00000000 00000000 **0000** 0011 01000001
→ 65, ASCII Code für 'A'
→Vordergrundfarbe: 3 entspricht Blau in LookUpTable.java
→ **Hintergrundfarbe: 0 entspricht Weiß in LookupTable.java**

welches dann über die ALUT als 8 32Bit Wörter interpretiert wird. Danach werden diese 8 32Bit Wörter entsprechend in den Grafikspeicherbereich geschrieben um ein 8x8 Pixel-Zeichen darstellen zu können.

Um die ASCII Zeichen und Farben zu realisieren haben wir eine Klasse “LookUpTable” implementiert, welche 2 Methoden besitzt:

```
public static int  CLUT(int code)
public static String ALUT(int switcher)
```

CLUT – Color Look up Table:
Die erste Methode dient dazu einen 32Bit Farbwert anhand des übergebenen Wertes “code” zu bekommen wodurch dann ein Pixel in einer Farbe gezeichnet werden kann. “code” kann hier zwischen 0 bis 15 sein um 16 Farben zu realisieren.

Beispiel:
Ein 32Bit Wort: 0xff112233 → RGB: ff=Alpha, 11=Rot, 22=Grün, 33=Blau
Daraus setzt sich dann die Farbe für einen Pixel zusammen.

ALUT – ASCII Look up Table:
Diese Methode speichert je ein 8x8 ASCII Zeichen als String in einem Array, welches dann abholbar über die Variable “int switcher” ist.

Beispiel:

```
return arr[switcher];
```

Anhand dieses Strings könen dann die 8 32-Bit Datenwörter erstellt werden, welche dann in den Grafikspeicherbereich geschrieben werden.

Der 3. Speicherbereich ist nun dazu da, um die aus dem Grafikbereich gelesenen und aus einem 32 Bit Wort zerlegten acht 4-Bit Werte hineinzuschreiben.
In diesem Speicher stehen also die gesamten 320*240 (Pixel-)Farbwerte, welche gebraucht werden um über die CLUT (LookupTable.java) einen Farbwert von 16 zu holen und dadurch einen Pixel auf den Bildschirm zeichnen zu können.

Dieser Bereich wird also auch ständig überwacht ob sich darin Änderungen ergeben, wodurch diese Änderungen auch sofort aktualisiert werden und auf dem Screen neu angezeigt werden.

Scotch Race

Um ScotchRace spielen zu können, kann man die „Left/Rigth/Down“ Tasten verwenden. Down/Unten startet das Spiel, Left ← / Rigth → zur Steuerung des Autos.

Dabei werden bei gedrückter Taste vordefinierte Werte in den Speicher geschrieben, welche vom ScotchRace ausgelesen und dementsprechend, als unten (=start), rechts oder links, interpretiert werden.

5) Erweiterungen (Luca Calchera, Michael Wager, Michael Wager)

5.1) Einleitung

Eine wichtige Anforderung an den HiCoVec Simulator war von Anfang an eine möglichst problemlose Erweiterung. Dies betrifft besonders den Befehlssatz, die Möglichkeit, Plugins hinzuzufügen und das einfache Modifizieren der vom Programm gesammelten Statistiken.

5.2) Befehlssatz (Luca Calchera)

Man kann neue Befehle mit folgenden Anweisungen implementieren:

Zunächst sollte der Befehlsname in Instruction.java und in Decoder.java hinzugefügt werden, damit der Befehl vom Prozessor erkannt und ausgeführt wird.
Die Datei Instruction.java ist im **package** hicosim.components.backend.processor.enums gespeichert.
Die Datei Decoder.java ist im **package** hicosim.components.backend.processor gespeichert.

```
public enum Instruction {
    LOAD, STORE, ADD, SUB, ADC, SBC, INC, DEC, AND, OR, XOR, MUL,
    JMP,JAL,JNC,JC,JNZ,JZ,       CLC,SEC,CLZ,SEZ,      LSL, LSR, ROL, ROR,
    NOP, VNOP,
    MOV, MOVA,
    //VEKTORBEFEHLE:
    VLOAD, VSTORE, VMOV, VMOL, VMOR,

    //Vector-ALU:
    VADD, VSUB, VAND, VOR, VXOR, VMUL, VLSL, VLSR,
    VSHUF;
}
```

(Abb. 5.1 - Befehlssatz)

Anschließend spezifiziert man den Befehlstyp, um den Befehl einer bestimmten Kategorie zuordnen zu können. Es muss entschieden werden, in welcher Einheit der Befehl bearbeitet werden soll. Dies hängt davon ab, wo der Befehlstyp gespeichert wurde: entweder in VectorType.java oder in ScalarType.java. Beide Dateien befinden sich im **package** hicosim.components.backend.processor.enums.

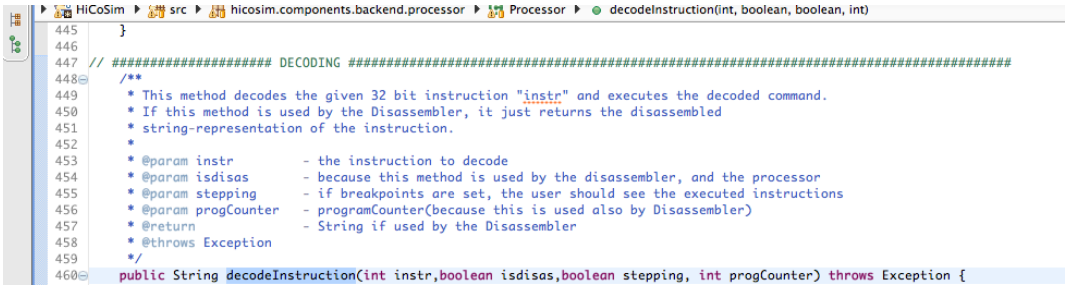
```
public enum ScalarType {
    LOAD_STORE, ALU, JMP_NOP,  FLAG;
}
```

(Abb. 5.2 - Skalarbefehle)

```
public enum VectorType {
    VALU, VSHUF;
}
```

(Abb. 5.3 - Vektorbefehle)

Darüber hinaus muss dem Prozessor erklärt werden, wie man den Befehl dekodiert: das Verfahren wird in der Methode decodeInstruction der Klasse Prozessor.java implementiert. Die Datei Prozessor.java ist im **package** hicosim.components.backend.processor gespeichert.



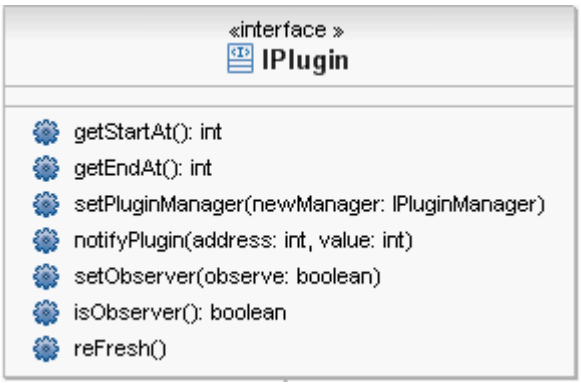
(Abb. 5.4 – Dekodierung)

5.3) Plugins (Michael Wager, Anderas Weber)

5.3.1) Die Schnittstellen IPlugin und IPluginManager (Andreas Weber)

Für ein möglichst problemloses Hinzufügen von Plugins, z.B. Eingabegeräten, stellt der HiCoSim die Schnittstellen IPlugin sowie IPluginManager zur Verfügung. Für den IPluginmanager steht bereits eine konkrete Implementierung , der `Pluginmanager`, bereit. Dieser enthält eine Referenz auf den Speicher, unterstützt Methoden zur Verwaltung von Plugins und ermöglicht ihnen Zugriff (lesend und schreibend) auf den Speicher. Durch die zentrale Verwaltung können auch einzelne Speicherbereiche für Lese/ Schreiboperationen gesperrt werden. Plugins benötigen somit nicht zwingend eine Referenz auf den HiCoVec-Speicher, sondern können über den Pluginmanager arbeiten.

Neue Plugins müssen, um vom PluginManager akzeptiert zu werden, das Interface *IPlugin* implementieren:



(Abb. 5.5 - IPlugin)

Beim Instantiieren muss jeder Erweiterung ein konkreter Speicherbereich im Konstruktor übergeben werden, in welchem der Schreib-/Lesezugriff erfolgen soll.

Eine Registrierung beim PluginManager und somit ein Zugriff auf den HiCoVec-Speicher erfolgt durch den Aufruf der Methode „*addPlugin(IPlugin plugin)*“ in der Klasse Engine. Die Engine übergibt das Plugin an den PluginManager. Dieser liest mit „*getStartAt*“ und „*getEndAt*“ die zulässigen Speicherbereiche des Plugins aus. Analog dazu kann das Plugin auch per „*delPlugin(IPlugin plugin)*“ entfernt werden.
Falls das Plugin als Observer (*setObserver(true)*) registriert wird, erfolgt bei jedem schreibenden Speicherzugriff auf den Memory automatisch ein vom Pluginmanager angestoßener Aufruf der „*notifyPlugin(int adr, int value)*“ Methode des Plugins, sofern sich die Adresse im festgelegten Bereich des beobachtenden Plugins befindet. Schreiboperationen über den Pluginmanager sind ebenfalls nur auf die festgelegten Adressbereiche möglich. Der gültige Speicherbereich kann vom der Erweiterung selbst zur Laufzeit durch Aufruf der Methode „*setNewRange(IPlugin plugin, int newStart, int newEnd*“) im PluginManager geändert werden.
Es ist auch möglich, den Speicher direkt über die Klasse Memory zu beschreiben (*engine.getMemory().writeAT(int adr, int value*). Dann allerdings können Schreibzugriffe nicht mehr zentral über den PluginManager verwaltet werden, außerdem muss jedes Plugin eine Referenz auf die Engine erhalten.

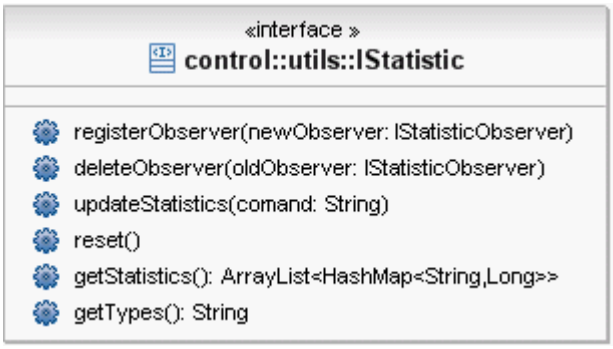
5.3.2) VGA-Plugin (Michael Wager)

Eine Erweiterung des VGA Plugins bedarf keines großen Aufwandes.
Um Zeichen hinzufügen zu können muss man nur die Klasse “LookUpTable.java” um (8x8) Strings erweitern und dies ggf. in der Klasse ProcessASCII anpassen.
Siehe:
(src/hicosim/components/plugins/vga/LookUpTable.java)
(src/hicosim/components/plugins/vga/ProcessASCII.java)

Implementierte Zeichen
A - Z
0 - 9
. : _ - ! ?
'b' : die Bombe im scotchRace
'k' : das Kleeblatt im scotchRace
't' : der Baum im scotchRace

5.4) Statistiken (Andreas Weber)

Im Hinblick auf zukünftige Ergänzungen des Befehlssatzes wurde auch die Erweiterbarkeit der Statistiken möglichst einfach gehalten. Statistiken werden in der Klasse `hicosim.control.utils.Statistics` gesammelt. Sie ist eine Implementierung der Klasse `IStatistic`.



(Abb. 5.6 – IStatistic)

Statistiken sind im HiCoSim nicht nur für konkret implementierte Befehle erzeugbar. Vielmehr kann jede Klasse, die zumindest Zugriff auf die Engine hat, neue Statistiken anlegen. Dies schließt natürlich auch Plugins, Speicherzugriffe in der Memory-Klasse etc. ein. Um einen Wert hinzuzufügen bzw. zu erhöhen genügt ein Aufruf der Methode: „*engine.getProcessor().getStatistics().updateStatistics(„Command“)*“.
Die Klasse Statistics prüft nun, ob dem übergebenen Wert ein Typ zugeordnet ist (z.B.: Command = „ADD“ → Type = „SKALARALU“). Falls keine Zuordnugn vorliegt, wird dem neuen Wert der Typ „OTHER“ zugewiesen. Für die aktuell implementierten Prozessorbefehle stehen hierfür bereits Default-Einstellungen bereit. Diese können aber sehr einfach in der Methode „decipherType(String comand)“ geändert und ergänzt werden. Es wird nun geprüft, ob und wie oft das Kommando schon einmal aufgerufen worden ist und die Häufigkeit dementsprechend erhöht.

Auch die Ausgabe der Statistiken kann leicht den individuellen Bedürfnissen angepasst werden. Neben einer rudimentären String-Ausgabe „toString()“, welche für die Darstellung auf der HiCoSim-Console ihre Anwendung findet, nutzt das Frontend den eingebauten Observer-Mechanismus der Statistiken, um eine sofortige Aktualisierung der Anzeige zu ermöglichen. Ausgangspunkt hierfür ist das Interface `IStatisticsObserver:`

(Abb. 5.7 - IStaticObserver)

Um eine neue Statistik-Auswertung zu realisieren, muss eine Klasse dieses Interface implementieren und sich selbst bei der Statistik-Klasse registrieren („`registerObserver(this)`“). Sie stellt nun einen Observer der Statistiken dar. Bei jeder Befehlserhöhung erfolgt der automatische Aufruf der Methode „`updateStatsObserver(String comand, String type)`“ aller Statistik-Observer. In dieser kann festgelegt werden, wie die konkrete Anzeige der Statstiken ausfallen soll. Möglich wäre auch die Weiterleitung in eine Datei o.ä. Im Falle der HiCoSim-GUI werden die Statistiken nach Häufigkeit sortiert und in einer JTable gespeichert.

Wichtig:
Die Methode „resetStats“ muss implementiert werden, um zu gewährleisten dass die Anzeige auch dann aktuell bleibt, wenn der Prozessor per reset zurückgesetzt wird. In diesem Fall wird automatisch die Methode `resetStats()` in allen Statistik-Observern aufgerufen.

6) Zusammenfassung (Müge Özugur, Hibetoullah Ftima)

Durch den in dieser Projektarbeit entstandenen „HiCoSim“ ist eine Simulation beliebiger HiCoVec-Programme möglich. Sämtliche Konfigurationen des Prozessors können nun getestet und mit Hilfe der Statistiken hinsichtlich ihrer Effizienz verglichen werden. Die möglichen Konfigurationen des HiCoVec können bequem über die grafische Benutzeroberfläche variiert oder automatisch in der config.vhd-Datei ausgelesen werden.

Der Simulator unterstützt den gesamten, zu diesem Zeitpunkt implementierten Befehlssatz einschließlich der Möglichkeit, parallele Befehle zu verarbeiten.

Die Softwareentwicklung für den HiCoVec ist durch den HiCoSim nun einfacher und komfortabler geworden. Die Ausführung von assemblierten Programmen ist nun ohne ein angeschlossenes FPGA-Board möglich, der Programmlauf kann jederzeit unterbrochen und fortgesetzt werden. Das setzen und löschen von Break- und Watchpoints stellen für den Entwickler eine große Hilfe bei der Fehlersuche und Optimierung dar.

Der aktuelle Zustand des HiCoVec, wie Register- und Speicherinhalte, kann jederzeit in der GUI überwacht werden. Das geladene Programm kann auch während der Ausführung durch die Änderung der Maschinenbefehle im Speicher modifiziert werden. Auch die Aufbereitung und Darstellung der Objektdatei in Form von disassemblierten Instruktionen bietet in Verbindung mit dem anzeigten Befehlsregister die Möglichkeit, den Programmablauf besser verfolgen zu können.

Wie schon in Kapitel 5 erläutert kann der HiCoSim in Hinsicht auf Befehlssatz, Plugins und Statistiken erweitert werden und kann so auch bei einer möglichen Weiterentwicklung des HiCoVec genutzt werden.

Das von der Projektgruppe „Der HiCoVec in Aktion **[2]**“ entwickelte Spiel „ScotchRace“ ist auch auf unserem Simulator lauffähig und wurde im Rahmen des Projekttages an der Hochschule Augsburg am 1. Juli 2009 demonstriert.

Aufgrund der Verwendung von Java als Programmiersprache und durch den Verzicht auf externe Bibliotheken ist der HiCoSim auf allen Systemen lauffähig, für die eine aktuelle Java Runtime Environment (ab Version 6) zur Verfügung steht. Getestet wurde dies unter Windows (XP/Vista), MacOS und Linux.

7) Anhang

7.1) Quellenverzeichnis

[1] Diplomarbeit Harald Manske – HiCoVec
http://www.hs-augsburg.de/~kiefer/hicovec/index.html

[2] „Der HiCoVec in Aktion“ - Projektarbeit im WS2007 ,
http://www.hs-augsburg.de/~kiefer/archiv/2008w-projekt/index.html

[3] Prof. Dr. Gundolf Kiefer: HiCoSim – Simulator für einen konfigurierbaren Vektorprozessor.
Projektarbeit SS2009, http://www.hs-augsburg.de/~kiefer/projekt-hicosim/index.htm

7.2) Befehlsreferenz [2]

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