Hochschule für Technik und Architektur Bern

DIPLOM 2002
Abteilung Informatik

– PARASITIC COMPUTING –

Verfasser:	Jürg Reusser, Luzian Scherrer
Betreuer:	M. Dürig, Dr. J. Boillat, Dr. J. Eckerle
Experte:	Walter Eich

PARASITIC COMPUTING
Aufgabenstellung

Umfeld
Ende letzten Jahres haben Wissenschaftler der Universität Notre Dame im US-Bundesstaat Indiana eine Methode demonstriert, mit der man ohne Wissen und Einwilligung der Anwender deren Rechnerkapazität im Internet nutzen kann. Sie lösten ein mathematisches Problem mit Hilfe von Web-Servern in Nordamerika, Europa und Asien, ohne dass die Betreiber dieser Sites etwas davon merkten. Dazu nutzten sie TCP aus: Um die Integrität von Daten sicherzustellen, plaziert der Sender eines Datenpakets eine Prüfsumme über die im Datenpaket enthaltenen Datenbits im Header. Auf der Empfängerseite wird diese Prüfsumme nachgerechnet und je nach Ergebnis wird das Datenpaket als korrekt akzeptiert, oder es wird verworfen. Durch geeignete Modifikation der Pakete und deren Header gelang es einfache Berechnungen durchzuführen. Siehe auch `http://www.nd.edu/~parasite/` und Dr. Dobb's Journal, Anatomy of a Parasitic Computer. Vincent W. Freeh. Januar 2002.

Aufgabenstellung & Zielsetzung
Die mittels dieses Missbrauchs durchführbaren Operationen reichen aus um jedes Problem das auf einem klassischen Computer lösbar ist zu lösen. In dieser Arbeit soll nun ein Compiler oder ein Interpreter entwickelt werden, der eine einfache Sprache für parasitäre Programme implementiert. Das heisst, ein solches Programm würde durch den Compiler/Interpreter automatisch in geeignete Grundoperationen übersetzt die dann mittels dem gezeigten Missbrauch der TCP-Prüfsumme ausgeführt werden können.

Lerninhalte
Compilerbau, verteiltes Rechnen, TCP/IP, Computer-Netzwerke und Protokolle

Hardware & Software
Linux, Windows, C/C++, Java, JavaCC, YACC, LEX, ...

Bemerkungen
Meines Wissens existiert bislang kein Compiler oder Interpreter dieser Art. Ein entsprechendes Programm könnte durchaus Aufsehen erregen.

Personen
Aufgabensteller ..... Michael Dürig
Bearbeiter ..... Jürg Reusser, Luzian Scherrer
Betreuer ..... Michael Dürig, Dr. J. Boillat, Dr. J. Eckerle
Experte ..... Walter Eich

Referenzen
Die Originalfassung der Aufgabenstellung – dieses Dokument ist eine inhaltlich identische Kopie – ist im Internet unter folgender Adresse verfügbar:

`http://www.hta-be.bfh.ch/~wwwinfo/di/02/Parasite.shtml`

PARASITIC COMPUTING
Dokumentenübersicht

Diese Seite enthält eine kurze Zusammenstellung aller beiliegenden Dokumente. Die dargestellte Reihenfolge entspricht dem empfohlenen Lesefluss. Der abschliessende Projektbericht kann optional auch als erstes Dokument gelesen werden.

PARASITIC COMPUTING
CD-ROM Distribution

Auf dieser CD-ROM ist die komplette Software-Distribution inklusive elektronischer Versionen aller Dokumente der Diplomarbeit „Parasitic Computing" enthalten. Eine Übersicht bietet die sich im Wurzelverzeichnis befindende Datei `index.html`.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

PARASITIC COMPUTING
Pflichtenheft

Version 1.0

Jürg Reusser
Luzian Scherrer

11. Juni 2002

Zusammenfassung

Das Prinzip „Parasitic Computing" wurde erstmals im August 2001 öffentlich erwähnt [BFJB01], als es Wissenschaftlern der Notre Dame Universität im US-Bundesstaat Indiana gelang, fremde Rechenkapazität ohne Wissen und Einwilligung derer Besitzer zur Lösung mathematischer Probleme zu nutzen. Die Vorliegende Diplomarbeit der Hochschule für Technik und Architektur Bern setzt auf dieser Forschungsarbeit auf und entwickelt Möglichkeiten, das vorgestellte Prinzip zu einem vollständig programmierbaren, parasitären Rechenwerk zu erweitern, welches sämtliche klassischen Probleme der Informatik berechnen könnte.

Inhaltsverzeichnis

1 Einleitung

1.1 Ausgangslage und Umfeld

Ende des Jahres 2001 haben Wissenschaftler der Universität Notre Dame im US-Bundesstaat Indiana eine Methode demonstriert, mit der man ohne Wissen und Einwilligung der Anwender deren Rechnerkapazität im Internet nutzen kann [BFJB01]. Sie lösten ein mathematisches Problem mit Hilfe von Web-Servern in Nordamerika, Europa und Asien, ohne dass die Betreiber dieser Sites etwas davon merkten. Dazu nutzten sie TCP aus: Um die Integrität von Daten sicherzustellen, platziert der Sender eines Datenpakets eine Prüfsumme über die im Datenpaket enthaltenen Datenbits im Header. Auf der Empfängerseite wird diese Prüfsumme nachgerechnet und je nach Ergebnis wird das Datenpaket als korrekt akzeptiert, oder es wird verworfen. Durch geeignete Modifikation der Pakete und deren Header gelang es einfache Berechnungen durchzuführen.

Diese Forschungsarbeit wurde in diversen Artikeln öffentlich publiziert und besprochen [Fre02], sie ist weitgehend auch aus technischer Sicht detailliert dokumentiert [oND02].

1.2 Vorhaben

Die mittels dieser Methode durchführbaren Operationen reichen prinzipiell aus, um jedes auf einem klassischen Computer lösbare Problem parasitär, sprich mit fremder Rechenkapazität, zu berechnen. In dieser Arbeit soll nun ein Compiler oder ein Interpreter entwickelt werden, der eine einfache Sprache für parasitäre Programme implementiert. Das heisst, ein solches Programm würde durch den Compiler/Interpreter automatisch in geeignete Grundoperationen übersetzt, die dann mittels dem gezeigten Missbrauch der TCP-Prüfsumme ausgeführt werden können.

1.3 Ziele

Es soll ein virtuelles, auf den parasitär durchführbaren Grundoperationen aufbauendes Rechenwerk (ALU) entwickelt werden, welches eine Teilmenge der gängigen arithmetischen und logischen Operationen zur Verfügung stellt. Diese Teilmenge muss mindestens dem Umfang an Instruktionen entsprechen, welcher benötigt wird, um sämtliche klassischen Probleme der Informatik zu lösen.

Um mit dieser virtuellen ALU arbeiten zu können, soll eine entweder durch sie interpretierbare oder für sie kompilierbare Assemblersprache entwickelt werden, die zusätzlich zu allen Instruktionen der ALU auch die nötigen Kontrollflussmöglichkeiten bietet. Es sind dies Sequenz, Auswahl und Wiederholung.

Da die virtuelle ALU in ihren Grundoperationen auf Netzwerkprotokollen und Verbindungen beruht und von deren Korrektheit abhängt, sollen geeignete Mechanismen zur Fehlererkennung und Fehlerbehebung untersucht und gegebenenfalls implementiert werden.

Neben der Muss-Anforderung einer Assemblersprache zur Programmierung des Rechenwerkes besteht optional die Möglichkeit eine Hochsprache (evtl. auf bereits existierenden Grundlagen der HTA-BE [Boi01]) zu erstellen. Eine solche Hochsprache

soll, falls sie realisiert wird, mittels eines Compilers in die Assemblersprache übersetzbar sein. Dies jedoch ohne die Anforderung, Effizienz steigernde Übersetzungsalgorithmen anzuwenden.

Die zum Abschluss resultierende Software soll in einem iterativen Zyklus aus sich erweiternden Prototypen erarbeitet werden. Im Realisierungskonzept (siehe Abschnitt 3.2 Seite 23), welches als nächstfolgender Meilenstein im Projekt gilt, werden die geforderten Ziele der Prototypen und des Endproduktes im Detail definiert.

1.4 Abgrenzung

1.4.1 Parasitäre Methode

Als parasitäre Methode wird für diese Arbeit primär das bekannte Vorgehen (vgl. [BFJB01]) mittels der Prüfsummen der Internetprotokolle auf den Layern 3 (IP, ICMP) und 4 (UDP, TCP) verwendet. Die Auswahl innerhalb dieser Kategorie wird im Realisierungskonzept (siehe Abschnitt 3.2 Seite 23) getroffen.

Die Erforschung und Evaluation weiterer, potenziell parasitär nutzbarer Protokolle (wie etwa solcher kryptographischer Natur) ist nicht vorgesehen, kann aber optional einbezogen werden falls entsprechende Varianten in Aussicht stehen.

1.4.2 Rechenwerk und Speicher

Das Rechenwerk soll ausschliesslich arithmetische und logische Operationen sowie Kontrollflussfunktionalitäten wie beispielsweise einen Programm-Counter bieten, die Register und der weiter benötigte Primärspeicher hingegen sind lokal abgebildet und nicht parasitär (sprich verteilt) zu implementieren.

1.4.3 Portabilität

Der Programmcode soll portabel und auf Linux sowie gängigen UNIX Systemen kompilierbar sein. Nicht unterstützt werden andere, sich wesentlich unterscheidende Betriebssysteme wie Microsoft Windows oder Apple MacOS.

1.4.4 Effizienz

Es ist nicht Ziel dieser Arbeit, die parasitären Berechnungen dahingehend zu entwickeln, dass sie dank ihrer Verteilung Effizienzvorteile gegenüber einer lokalen ALU bieten. Die Arbeit soll in diesem Sinne als ,,Proof of Concept” dienen und nicht in einem in der Praxis einsetzbaren High-Performance Rechner resultieren.

1.4.5 Parallelität

Wir beschränken uns auf die sequenzielle Abarbeitung von Programmcode und verfolgen keine Parallelisierung auf Level des Rechenwerkes wie dies etwa bei Multipipe-Prozessoren angewendet wird.

2 Projektorganisation

Beide Team Mitglieder (siehe Abschnitt 2.1 Seite 21) sind gleichberechtigt und neue Verantwortungsbereiche werden im Dialog fortlaufend aufgeteilt. Die Verantwortung für einen Bereich ist nicht gleichbedeutend mit alleiniger Durchführung.

2.1 Projektteam

Name	*E-Mail*
Luzian Scherrer	ls@parasit.org
Juerg Reusser	jr@parasit.org

2.2 Betreuer

Name	*E-Mail*
Dr. Jacques Boillat	jacques.boillat@hta-be.bfh.ch
Dr. Jürgen Eckerle	juergen.eckerle@hta-be.bfh.ch
Michael Dürig	michael.duerig@hta-be.bfh.ch

2.3 Experten

Name	*E-Mail*
Walter Eich	we@zuehlke.com

2.4 Arbeitszeiten des Projektteams

2.4.1 8. Semester, Projektphase

Während des 8. Semesters ist das in die Arbeit investierte Wochenpensum folgendermassen aufgeteilt:

Dauer	*Wochentag*	*Zeit*
6 Lektionen	Montag	16:15 bis 21:30
4 Lektionen	Dienstag	18:10 bis 21:30
6 Lektionen	Freitag	12:45 bis 17:50

Die Wochenarbeitszeit beträgt somit 16 Lektionen zu 45 Minuten, was 12 Stunden entspricht. Der Dienstag Abend wird für Besprechungen mit den Betreuern reserviert.

2.4.2 Diplomarbeitsphase

In der die Arbeit abschliessenden Diplomphase wird das Leistungspensum je nach Anforderungen und Stand des Projektes erhöht, wobei ein durchschnittlicher Aufwand von ungefähr 20 Wochenstunden anzustreben ist.

Besprechungen werden nach Rücksprache mit dem Experten respektive den Betreuern abgehalten.

2.5 Sitzungen

2.5.1 Teamsitzungen

Zweck	Projektstand, Terminüberwachung, Entscheidungen, aktuelle Probleme und weiteres Vorgehen
Teilnehmer	Projektteam
Häufigkeit	In der Regel jeden zweiten Dienstag Abend, 20:00 Uhr
Dauer	Nach Bedarf

2.5.2 Reviews

Zweck	Projektstand
Teilnehmer	Projektteam, Betreuer
Häufigkeit	Zwischen den Meilenstein-Reviews alle zwei bis drei Wochen
Dauer	Nach Bedarf

2.5.3 Meilenstein Review

Zweck	Meilenstein Abnahme
Teilnehmer	Projektteam, Betreuer, Experte
Häufigkeit	Beim erreichen eines Meilensteins
Dauer	Nach Bedarf

3 Projekspezifisches Vorgehensmodell

3.1 Projektstruktur

Die folgenden Abschnitte gliedern die Arbeit in ihre logischen Projektphasen ein.

3.1.1 Projekt-Setup

Das Projekt-Setup beinhaltet folgende Aktivitäten:

- Formelle Details klären
- Einarbeitung in die Materie
- Bereitstellung der Infrastruktur
- Konkretisierung der Problemstellung
- Definition der Ziele

Aus dieser Phase resultiert das fertige Pflichtenheft.

3.1.2 Erarbeitung Realisierungskonzept

Prinzipiell besteht die Implementation aus drei fundamentalen, erstrebenswerterweise möglichst lose gekoppelten, Schichten. Diese drei Teilbereiche sollen mittels vor der Implementierungsphase definierter Programmierschnittstellen verbunden werden können.

- Assembler
  - Literaturstudium
  - Existierende Sprachen auf Adaptierbarkeit und Umfang evaluieren
  - Benötigten Instruktionssatz festlegen
- Aufbau CPU/ALU
  - Literaturstudium
  - Studium und Festlegung der Architektur
  - Benötigte Datenstrukturen definieren
- Verteiltes Rechnen
  - Literaturstudium
  - Studium parasitär nutzbarer Netzwerkprotokolle
  - Potenzielle Fehlerquellen im Netzwerkteil eruieren
  - Untersuchung geeigneter Fehlerbehandlungsmechanismen

Aus dieser Phase resultiert das die Projektziele detailliert beschreibende Realisierungskonzept mit sämtlichen Angaben über die bevorstehende Implementation.

3.1.3 Implementation

In dieser Phase wird die effektive, auf dem Realisierungskonzept basierende Implementation durchgeführt. Dabei wird pro Modul sowie gesamthaft ein Vorgehen nach folgenden Punkten angestrebt:

- Analyse und Design
- Aufbau
- Durchführung
- Bewertung

3.1.4 Auswertung

Die Auswertungsphase umfasst folgende Teilbereiche:

- Gesamtbewertung
- Erreichte Ziele
- Erfahrungen
- Schlussbemerkungen

Aus der Auswertungsphase resultiert der abschliessende Projektbericht. Weitere Details zur Bewertung sind in Kapitel 5 festgehalten.

3.2 Meilensteine

Meilenstein	*Datum*
Abnahme Pflichtenheft	Mitte Juni 2002
Abnahme Realisierungskonzept	Vor den Herbstferien
Abschluss Implementation	Zwei Wochen vor Projektabnahme
Abnahme Projektbericht	Kalenderwoche 2, Januar 2003

3.3 Risikoanalyse: Risiken

3.3.1 Personenausfall (Krankheit, Militär, Beruf)

Faktor	Mittel
Auswirkung	Gross
Massnahmen	Permanentes Know-How-Sharing im Projektteam

3.3.2 Mathematisches Neuland

Faktor	Klein
Auswirkung	Gross
Massnahmen	Zusammenarbeit mit Mathematikern, entsprechende Know-How Beschaffung

3.3.3 Technisches Neuland

Faktor	Mittel
Auswirkung	Mittel
Massnahmen	Feingliedrige Bildung der Module, entwickeln von Simulatoren, falls Performanceprobleme

Im Realisierungskonzept werden Erfolgsaussichten und Risiken für die einzelnen Module/Komponenten separat abgeschätzt.

4 Ergebnisse

4.1 Besonderheit dieses Projekts

Bei dieser Diplomarbeit liegt das Schwergewicht nicht primär im Software Design sondern mehr im konzeptionellen und experimentellen Bereich. Dies hat natürlich auch Auswirkungen auf die zu erstellende Dokumentation. Insbesondere bedeutet dies, dass wir nicht eine Dokumentation im Sinne eines klassischen Softwareprojektes erstellen werden. Stattdessen werden Dokumente mit der Beschreibung der von uns durchgeführten Testimplementationen, Resultate und der gemachten Erfahrungen entstehen.

4.2 Dokumente

Bezeichnung	*Inhalt*
Pflichtenheft	Beschreibung der Ziele, Projektorganisation (gilt als definitive Aufgabenstellung)
Realisierungskonzept	Siehe Abschnitt 3.1.2 Seite 22
Implementation und Testbericht	Vorhaben Testen, Experimentieren, Resultate, Beurteilung
Lab Journal	Chronologische Notizen zum Projektverlauf, den Besprechungen und sonstigen Aktionen
Projektbericht	Projektverlauf, Erreichte Ziele, Bemerkungen, Schlusswort

4.3 Source-Code

Wir streben danach, die Applikation funktional betrachtet in verschiedene Module zu unterteilen, welche durch experimentieren und erforschen von Gegebenheiten und Ansätzen zur Lösungsfindung entwickelt werden. Wir behalten uns die Möglichkeit vor – falls dies als dringlich notwendig erscheint – vor Schluss der Diplomarbeit ein Redesign der ganzen Applikation vorzunehmen, da zum Zeitpunkt des Realisierungskonzeptes nicht hundertprozentig abgesehen werden kann, in welche exakten Richtungen sich verschiedene Lösungsansätze bewegen werden.

5 Beurteilung und Bewertung

Da diese Diplomarbeit nicht einem Projekt im Sinne der klassischen Software Entwicklung entspricht, bedarf es Korrekturen betreffend der Gewichtung der einzelnen Kriterien, welche in nachfolgender Bewertungsliste aufgeführt sind.

5.1 Bewertungsliste

	Arbeitsschritt	*Gewicht*
Vorbereitungsphase	Aufbau und Vollständigkeit des Pflichtenheftes	2
Durchführung	Arbeits und Zeitplanung	1
	Kreativität (Initiative, Selbstständigkeit)	3
	Wahl und Anwendung der (Arbeits-)Methodik	2
	Implementation, Robustheit, Programmierstil	2
	Systemtest (Verfahren, Durchführung, Bericht)	2
	Kommunikation mit Experten und Betreuer	1
Ergebnis	Übereinstimmung Endprodukt/Pflichtenheft bzw. Realisierungskonzept	5
	Allgemeiner Eindruck aus der Besichtigung	1
Projektbericht	Inhalt korrekt, vollständig, verständlich	3
	Sprache, Stil, Übersichtlichkeit	1
	Klare, aussagekräftige Zusammenfassung	1

6 Konfigurationsmanagment

6.1 Projektablage und Publizierung

Sämtliche Ergebnisse des Projektes werden laufend auf der Website `http://www.parasit.org` in ihrer jeweils aktuellsten Version verfügbar gemacht. Dokumente hierbei in den Formaten HTML und PDF, Source-Code und Binärdateien in TAR Archiven.

Über entscheidende Neuerungen wird mittels der Mailingliste `all@parasit.org` informiert. Dieser Verteiler beinhaltet das Projektteam und die Betreuer, optional auch den Experten.

Der Server wird voraussichtlich mit all seinen Funktionalitäten auch über Diplomschluss hinaus vollumfänglich zur Verfügung stehen.

6.2 Änderungskontrolle

Sämtliche Dateien tragen während der Bearbeitungsphase CVS interne Versionsnummern. Jeweils bei Abschluss einer Phase (Minor Release) sowie bei Abnahme (Major Release) wird eine symbolische Versionsnummer generiert. Minor Releases erhöhen hierbei die symbolische Versionsnummer in der Nachkommastelle, Major Releases in der Vorkommastelle. Die Änderungen zwischen Releases werden in Changelogs (in die Dokumente bzw. den Source-Code integriert) festgehalten und mitpubliziert.

6.3 Datensicherung

Sämtliche elektronisch gespeicherten Daten werden ausschliesslich mit dem Concurrent Versioning System [CVS02] verwaltet, womit jederzeit auf jede jemals existiert habende Version zugegriffen werden kann. Die persistente Datensicherheit ist durch das nächtliche Sichern des kompletten CVS Repositories in ein gewartetes Tape-Archive gewährleistet.

PARASITIC COMPUTING
Realisierungskonzept

Version 1.0

Jürg Reusser
Luzian Scherrer

29. Oktober 2002

Zusammenfassung

Das Prinzip „Parasitic Computing" wurde erstmals im August 2001 öffentlich erwähnt [BFJB01], als es Wissenschaftlern der Notre Dame Universität im US-Bundesstaat Indiana gelang, fremde Rechenkapazität ohne Wissen und Einwilligung derer Besitzer zur Lösung mathematischer Probleme zu nutzen. Die vorliegende Diplomarbeit der Hochschule für Technik und Architektur Bern setzt auf dieser Forschungsarbeit auf und entwickelt Möglichkeiten, das vorgestellte Prinzip zu einem vollständig programmierbaren, parasitären Rechenwerk zu erweitern, welches sämtliche klassischen Probleme der Informatik berechnen könnte.

Inhaltsverzeichnis

1 Einleitung

Ende des Jahres 2001 haben Wissenschaftler der Universität Notre Dame im US-Bundesstaat Indiana eine Methode demonstriert, mit der man ohne Wissen und Einwilligung der Anwender deren Rechnerkapazität im Internet nutzen kann [BFJB01]. Sie lösten ein mathematisches Problem mit Hilfe von Web-Servern in Nordamerika, Europa und Asien, ohne dass die Betreiber dieser Sites etwas davon merkten. Dazu nutzten sie TCP aus: Um die Integrität von Daten sicherzustellen, platziert der Sender eines Datenpakets eine Prüfsumme über die im Datenpaket enthaltenen Datenbits im Header. Auf der Empfängerseite wird diese Prüfsumme nachgerechnet und je nach Ergebnis wird das Datenpaket als korrekt akzeptiert, oder es wird verworfen. Durch geeignete Modifikation der Pakete und deren Header gelang es einfache Berechnungen durchzuführen.

Diese Forschungsarbeit wurde in diversen Artikeln öffentlich publiziert und besprochen [Fre02], und ist weitgehend auch aus technischer Sicht detailliert dokumentiert [oND02].

Die mittels dieser Methode durchführbaren Operationen reichen im Prinzip aus, um jedes auf einem klassischen Computer lösbare Problem parasitär, sprich mit fremder Rechenkapazität, zu berechnen.

1.1 Umfeld und Motivation

Die Motivation dieser Arbeit ist in zwei Hauptpunkten angesiedelt. Es ist dies einerseits das Design und die Implementierung einer virtuellen Maschine von Grund auf inklusive dem Design ihrer Steuerung mittels Assembler und optional einer Hochsprache. So fliessen verschiedenste Bereiche der Informatik in die Arbeit ein, unter anderem sind dies elektrotechnische Grundlagen und boolsche Schaltungslogik, induktive Rückführung sämtlicher Arithmetik auf die Addition, Prozessor-Design und Implementierung, Scanning und Parsing Techniken, Compilerbau und Programmiersprachenkonstruktion. Auf der anderen Seite der Motivation steht der Teil der verteilten parasitären Berechnung mit dem Ziel, ein „Proof of Concept" über die Vollständigkeit des „Parasitic Computing" zu liefern und die theoretische Machbarkeit in die Praxis umzusetzen und zu demonstrieren.

1.2 Verwandte Arbeiten

Auf der Seite des „Parasitic Computing" sind zum momentanen Zeitpunkt an verwandten Arbeiten lediglich die Versuche der Forscher der Notre Dame Universität zu nennen, welche über die Website [oND02] verfügbar sind. Weitere Anwendungen des parasitären Prinzips, wie es dort vorgestellt wird, sind nicht bekannt.

Der restliche Teil der Arbeit, die um den parasitären Kern gebaute virtuelle Maschine inkl. deren Programmiersprache(n), kennt viele Gemeinsamkeiten mit anderen Arbeiten. Als prominentestes Beispiel ist die Programmiersprache JAVA zu nennen, welche komplett auf einer virtuellen Maschine aufgebaut ist [TL99]. Ausserdem haben wir diesbezüglich auf die in Kapitel 6 des Buches [Sch92] erläuterten Ideen zurückgegriffen.

2 Analyse der drei Hauptkomponenten

Grundlegend wird die Arbeit in drei strukturell unabhängige Teilbereiche gegliedert. Es sind diese erstens die virtuelle Maschine, zweitens deren Programmierbarkeit und drittens die parasitäre Berechnung. In den folgenden Abschnitten wird genauer auf diese drei Schichten eingegangen und deren Schnittstellen werden definiert. Die Reihenfolge der nachfolgend erläuterten Unterkapitel ist des besseren Verständnisses wegen vertauscht: Damit der Microassembler-Code richtig interpretiert werden kann, sollte zuerst die Funktionsweise der virtuellen Maschine klar sein. Zuletzt wird der Netzwerkteil erklärt.

Abbildung 1 soll das Zusammenspiel der Hauptkomponenten mit den entsprechenden Schnittstellen verdeutlichen, die rechtsliegende Zusammenfassung verweist auf das jeweils entsprechende Kapitel:

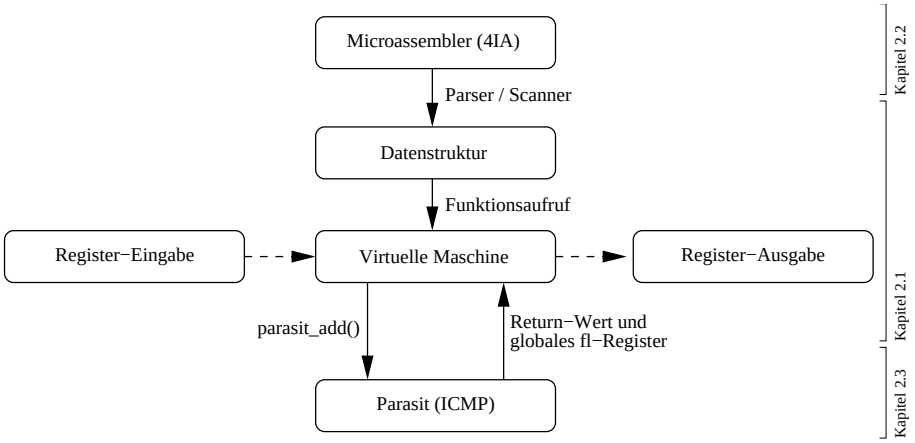

Abbildung 1: Gliederung der Hauptkomponenten und Schnittstellen

2.1 Die virtuelle Maschine

Dieser Abschnitt beschreibt die der Arbeit zugrundeliegende virtuelle Maschine sowie deren Schnittstellen einerseits zur Steuerung mittels einer Microassemblersprache, andererseits zur Durchführung der arithmetischen Grundoperationen.

2.1.1 Register-Maschine

Die virtuelle Maschine dieser Arbeit entspricht dem klassischen Register-Maschinen Modell. Diese zeichnet sich primär dadurch aus, dass der gesamte Speicher lediglich durch Register und nicht noch zusätzlich durch von diesen getrennten Hauptspeicher abgebildet ist. Die Register-Maschine verfügt über eine definierbare Anzahl n Register (genannt `r0` - `r`n), wovon zweien eine besondere Bedeutung zukommt: Das Register `r0` (im Folgenden auch `ip` genannt) repräsentiert den Instruktionszeiger, das Register `r1` (im Folgenden auch `fl` genannt) dient als binäres Flag-Register (enthält also die Werte 0 oder 1) um Additionsüberläufe anzuzeigen.

Register können direkt oder indirekt angesprochen werden. Indirekter Zugriff wird durch das Zeichen * markiert. So bedeutet beispielsweise der Ausdruck *r7 „das Register, dessen Nummer in Register 7 abgelegt ist".

Register Name	Register Alias	Bedeutung
r0	ip	Instruktionszeiger
r1	fl	Binäres Flag-Register für Überlauf
r2 ... r*n*		Weitere Register
*r*n		Darstellung indirekter Adressierung

Tabelle 1: Register der virtuellen Maschine

In diesem Dokument wird eine Registerbreite von 8 Bit angenommen. Diese Definition ist entscheidend, um bestimmen zu können, wo Additionsüberläufe stattfinden. Die effektive Implementation ist jedoch nicht zwingend an diese 8 Bit gebunden.

2.1.2 Eingabe - Verarbeitung - Ausgabe

Die virtuelle Maschine verfügt grundsätzlich nicht über Ein- und Ausgabe Funktionalität. Der konkrete Ablauf des E-V-A Prozesses gliedert sich in drei Phasen: In der ersten Phase, der Eingabe-Phase, werden die Register unabhängig von der virtuellen Maschine mit den gewünschten Werten initialisiert. Danach folgt die effektive Verarbeitungs-Phase, in welcher das Rechenwerk autonom den eingegebenen Programmcode abarbeitet. Abschliessend folgt die Ausgabe-Phase, in welcher, wieder unabhängig von der virtuellen Maschine, die Registerwerte zur Interpretation durch den Anwender ausgelesen werden.

2.1.3 Instruktionen

Das Rechenwerk ist mittels folgender Sprache, welche wir von nun an 4IA[1] nennen wollen, steuerbar:

- HLT
  Die Instruktion HLT (halten) terminiert den Ausführungsprozess des Rechenwerkes.
- SET *dst*, *const*
  Die Instruktion SET (setzen) setzt den Wert *const* ins Register *dst*. Dabei ist *const* eine ganze Zahl zwischen 0 und $2^{RB} - 1$, wobei RB die definierte Registerbreite ist. *dst* ist immer ein direkt adressiertes Register.
- MOV *dst*, *src*
  Die Instruktion MOV (verschieben) kopiert den Inhalt von Register *src* nach Register *dst*, so dass Register *src* erhalten bleibt. Sowohl *src* als auch *dst* können direkt oder indirekt adressierte Register sein.[2]

[1] Die Abkürzung 4IA steht für „4 Instruktionen Assembler".

[2] Die Instruktion MOV müsste eigentlich eher CPY heissen, da sie kein „move", sondern ein „copy" durchführt. Wir haben uns aber an die übliche Namensgebung MOV gehalten.

- `ADD` *dst*, *src*
  Die Instruktion `ADD` (addieren) addiert den Wert von Register *src* zum Wert von Register *dst* und legt das Resultat in Register *dst* ab, wobei sowohl *src* als auch *dst* ganze Zahlen sind (Ganzzahladdition). Falls die Summe grösser als $2^{RB} - 1$ (RB = Registerbreite) ist, findet ein Überlauf statt, das heisst die Addition ergibt ein Resultat modulo 2^{RB} und setzt im Falle des Überlaufes das Register `fl` auf 1. Findet kein Überlauf statt, so wird das Register `fl` nicht implizit auf 0 gesetzt (muss also gegebenenfalls vorher gelöscht werden damit ein eindeutiger Zustand resultiert). Sowohl *src* als auch *dst* sind immer direkt adressierte Register.

Es ist zu erwähnen, dass die Instruktionen `HLT`, `SET` und `MOV` für das Modell nicht zwingend notwendig sind. Sie dienen primär der Verständlichkeit der Darstellung des virtuellen Rechenwerkes im Pseudocode und vor allem der Verständlichkeit des Microassemblers 4IA. `HLT` könnte beispielsweise durch einen Sprung zu einer übermässig hohen Instruktionszeile repräsentiert werden und `SET` kann grundsätzlich durch korrekte Vorbereitung in der Eingabe-Phase (siehe 2.1.2) ersetzt werden, indem nämlich alle wärend des Ablaufes jemals benötigten Konstanten in Registern abgelegt werden. `MOV` schliesslich kann als `ADD` repräsentiert werden: `MOV r3, r4` ist äquivalent zu `ADD r3, r4`, sofern das Register `r3` vorgängig (mittels eines Subtraktionsloops bestehend aus einer Folge von `ADD r3, r5` mit `0xFF` initialisiertem `r5`) auf 0 gesetzt wird. Loops werden später genauer erläutert.

Voraussetzung für die Reduktion aller Instruktionen auf eine einzige Instruktion `ADD` ist, dass die verbleibende Instruktion `ADD` mit den Registern *src* und *dst* sowohl direkt als auch indirekt adressiert umgehen kann.

Opcode	Mnemonic	Argumente	Beschreibung	Adressierung
00	`HLT`	keine	Terminieren	
01	`SET`	*reg*, *const*	Konstante laden	direkt
10	`MOV`	*dst*, *src*	Register kopieren	direkt od. indirekt
11	`ADD`	*dst*, *reg*	Register addieren	direkt

Tabelle 2: Zusammenfassung der Instruktionen

Durch die verwendeten vier Instruktionen wird versucht, ein Optimum zwischen Verständlichkeit einerseits und Simplizität der virtuellen Maschine andererseits zu erreichen. Sie sind in Tabelle 2 nochmals zusammengefasst inklusive Opcodes, erwarteter Argumente, kurzer Beschreibung und Adressierungsart.

2.1.4 Das Rechenwerk

Nachfolgend wird nun die virtuelle Maschine in Pseudocode dargestellt. Das Array `code` enthält hierbei den abzuarbeitenden 4IA-Code.

```
register r[0..n] = 0;
integer ip = 0;

while(forever)
{
  switch(code[r[ip]])
```

```
  {
    ADD:
      r[dst] = parasit_add(r[src], r[dst]);
    MOV:
      r[r[dst]] = r[r[src]];
    SET:
      r[dst] = const;
    HLT:
      terminate;
  }
  r[ip] = parasit_add(r[ip], 1);
}
```

Das Rechenwerk führt also solange einzelne Zyklen durch, bis es auf die Instruktion `HLT` im Programmcode trifft. Dabei wird pro Zyklus die aktuelle Instruktion inklusive ihrer Argumente gelesen und entsprechend ausgeführt. Die die Addition repräsentierende Funktion `int parasit_add(int, int)` wird im Abschnitt 2.3 genau erläutert, an dieser Stelle kann von einer „normalen" Addition wie in Abschnitt 2.1.3 beschrieben ausgegangen werden. Nach jedem Zyklus wird der Instruktionszeiger `ip` um 1 inkrementiert.

Im gezeigten Pseudocode implementiert die Instruktion `MOV` ausschliesslich indirekte Adressierung. Die direkte Adressierung, wie sie vorgängig definiert wurde, wird dabei als Spezialfall der indirekten Adressierung angesehen: so kann man beispielsweise in der Eingabephase die Register 1000 bis 1999 statisch mit den Werten 0 bis 999 initialisieren. Damit lässt sich darauffolgend ein Register r, wobei $(0 \leq r \leq 999)$, mittels $r + 1000$ direkt adressieren. Dieses Vorgehen wird in unserer Implementation vom Compiler – für den Programmierer also absolut transparent – durchgeführt.

Dieses im Pseudocode dargestellte Modell soll primär verdeutlichen, welche Operationen lokal in der virtuellen Maschine, und welche extern (durch Simulation oder später parasitär im Netzwerk) ausgeführt werden. Lokal finden, wie im Pflichtenheft gefordert, ausschliesslich Speicheroperationen statt (dies entspricht allen mit „=" beschriebenen Zuweisungen im Pseudocode). Die Arithmetik ist komplett auf die Funktion `int parasit_add(int, int)` ausgelagert und geschieht ausserhalb des Rechenwerks.

2.1.5 Schnittstellen

Die Schnittstelle zur „Programmierschicht" wird durch einen Scanner/Parser gegeben. Dieser hat die Aufgabe, den auszuführenden 4IA-Code in eine der virtuellen Maschine verständliche Form zu bringen (also in eine geeignete Datenstruktur, im gezeigten Pseudocode dem Array `code[...]` entsprechend). Ob der Scanner/Parser programmtechnisch direkt in die virtuelle Maschine integriert wird, oder ob eine Übergabe dieser Datenstruktur mittels Funktionsaufruf stattfindet, ist an dieser Stelle nicht entscheidend und wird sich je nach Realisierung zeigen.

Die Schnittstelle zur verteilten Berechnung, also der parasitären Schicht, erfolgt ausschliesslich über die bereits erwähnte Funktion `int parasit_add(int, int)`. Diese erwartet als Argumente zwei ganze Zahlen und liefert deren Summe modulo 2^{RB} zurück. Falls ein Überlauf stattgefunden hat, die Summe ohne Modulerechnung

also grösser $2^{RB} - 1$ wäre, setzt diese Funktion ausserdem das globale Flag-Register `fl` auf 1. Im anderen Fall, bei einer Summe kleiner $2^{RB} - 1$, bleibt das Flag-Register unverändert.

2.2 Programmierbarkeit

In diesem Abschnitt soll gezeigt werden, wie die virtuelle Maschine mittels der 4IA-Sprache den Anforderungen im Pflichtenheft entsprechend programmiert werden kann. Die im Folgenden dargestellten Code-Beispiele enthalten der besseren Lesbarkeit wegen am Anfang jeder Zeile eine Zeilennummer gefolgt von einem Doppelpunkt und optional einen durch ein Semikolon abgetrennten Kommentar am Ende der Zeile[3]. Dazwischen befindet sich der eigentliche Code.

Sequenz, Wiederholung und Auswahl sind die drei Kontrollkonstrukte einer Programmiersprache, die gemäss Pflichtenheft implementierbar sein müssen und mit denen sich prinzipiell jedes klassische Problem der Informatik darstellen und lösen lässt. Diese Konstrukte werden im Folgenden erläutert.

2.2.1 Sequenz

Die Sequenz ergibt sich implizit durch die Inkrementierung des Instruktionszeigers `ip` in jedem Zyklus wie dies aus dem Pseudocode der virtuellen Maschine leicht ersichtlich ist.

2.2.2 Wiederholung

Eine Wiederholung ist ein unbedingter Sprung im Programmcode. Dieser kann entweder an eine direkte oder an eine als Offset angegebene Adresse durchgeführt werden:

`0:  SET ip,` *adresse-1*

oder

`0:  ADD ip,` *offset-1*

Sowohl der Offset wie auch die Adresse werden hier bei der Programmierung dekrementiert. Grund dafür ist die implizite Inkrementierung des Instruktionszeiger `ip` pro Zyklus, welche von der virtuellen Maschine vorgenommen wird. Die (vom Programmierer vorgenommene) Subtraktion von 1 bei den Sprungzielen gleicht dies aus.

2.2.3 Auswahl

Die Auswahl kann aufgrund des Wertes des binären Flag-Registers `fl` ausgeführt werden. Dabei sind verschiedene Vergleichskonstrukte denkbar, welche in einem entsprechend gewünschten Zustand des `fl` Registers enden:

```
0:  ADD ip, fl  ; fl-Reg. zum Instruktionszeiger addieren
1:  ...         ; Diese Zeile wird bei fl = 0 angesprungen
2:  ...         ; Diese Zeile wird bei fl = 1 angesprungen
```

[3] Zeilennummerierung und Kommentar werden auch in der effektiven Implementation durch den Parser als syntaktisch gültig akzeptiert, in der Codegenerierung jedoch ignoriert.

Zu beachten gilt auch hier, dass eine Addition von 1 zum Instruktionszeiger effektiv in einem Sprung um zwei Zeilen resultiert, denn `ip` wird in jedem Zyklus zusätzlich zu dieser Addition implizit inkrementiert. Analog bedeutet die Addition von 0 die Weiterführung bei der nachfolgenden Zeile und nicht etwa ein Stehenbleiben des Programmzählers.

Um so beispielsweise die Auswahl

```
if ( x >= 10 ) {
  Auswahl TRUE;
} else {
  Auswahl FALSE;
}
```

zu realisieren, ist folgender 4IA-Code denkbar (wobei das Register `r5` der Variable `x` entspricht):

```
0:  SET fl, 0    ; fl-Reg. mit 0 initialisieren
1:  SET r3, 246  ; 246 = 0xFF - 10 + 1
2:  MOV r4, r5   ; r5 sichern
3:  ADD r4, r3   ; Addition, bei r5 >= 10 Ueberlauf
4:  ADD ip, fl   ; siehe vorangehendes Beispiel
5:  ...          ; Auswahl FALSE
6:  ...          ; Auswahl TRUE
```

Die Zeilen 5 und 6 können hier natürlich wiederum Sprünge sein, welche zu anderen Positionen im Code zeigen – dies wäre dann ein Beispiel für die Implementation eines bedingten Sprunges.

2.2.4 Programmbeispiel: Schlaufe

Um die vorgestellten Kontrollkonstrukte in einer häufig gebrauchten Form der Anwendung zu demonstrieren, wird nun ein erstes Programmbeispiel vorgestellt. Es handelt sich dabei um eine einfache Schlaufe, die – in diesem gezeigten Beispiel – exakt fünf mal durchlaufen wird. Der Rumpf der Schlaufe (Zeile 3) wird ausgelassen:

```
0:  SET r3, 0xFF ; Integerkonstante fuer Dekrementierung laden
1:  SET r2, 5    ; Anzahl Durchgaenge ins Zaehler-Register
                 ; laden, hier werden 5 Durchgaenge gemacht
2:  SET ip, 3    ; An Start des Loops springen: Zeile 4 (=3+1)
3:  ...          ; Loop-Body
4:  SET fl, 0    ; Flag-Register Loeschen
5:  ADD r2, r3   ; Zaehler dekrementieren, Flag-Register
                 ; wird 1 falls Ueberlauf stattfindet
6:  ADD ip, fl   ; Falls Ueberlauf in Zeile 5, Sprung Zeile 8
7:  SET ip, 8    ; Loop beenden: Zeile 9 (=8+1)
8:  SET ip, 2    ; Loop wiederholen: Zeile 3 (=2+1)
9:  HLT          ; Programm beenden
```

2.2.5 Programmbeispiel: indirekte Adressierung

Noch nicht eingegangen wurde bis jetzt auf die Notwendigkeit der indirekten Adressierung von Registern (dargestellt durch ein `*`-Präfix). Die indirekte Adressierung ist

notwendig, um nicht skalare Datenstrukturen wie beispielsweise ein Array oder einen Stack zu implementieren.

Das folgende Beispiel zeigt die Implementation eines mit `r10` als Stackzeiger und `r11` - `r`*n* als effektive Stackdaten realisierten Stacks. Die Initialisierung könnte etwa so aussehen:

```
0: SET  r10, 11    ; Stackzeiger r10 mit Konstante fuer
                   ; Stackinitialisierung laden
```

Die Operation *push* legt den Inhalt von Register `r3` auf den Stack:

```
0: SET  r2,  1     ; Integerkonstante fuer Addition +1 laden
1: MOV *r10, r3    ; Inhalt von r3 auf den Stack legen
2: ADD  r10, r2    ; Stackpointer erhoehen
```

Die Zeile 1 ist der springende Punkt; es wird hier der Inhalt von Register `r3` in dasjenige Register geladen, auf welches der Stackzeiger `r10` zeigt. Ohne die indirekte Adressierung würde an dieser Stelle der Stackzeiger verändert und nicht das durch ihn referenzierte Register.

Ähnlich ist nun die Operation *pop*, bei welcher der oberste sich auf dem Stack befindende Wert ins Register `r3` geladen und der Stackzeiger dekrementiert wird, implementiert:

```
0: SET r2, 0xFF    ; Integerkonstante fuer Dekrementierung laden
1: ADD r10, r2     ; Stackzegier dekrementieren
2: MOV r3, *r10    ; Oberstes Element vom Stack nach r3
                   ; verschieben
```

Die Zeile 2 ist hier interessant; es wird analog zu *push* der Inhalt des durch `r10` referenzierten Registers und nicht dessen eigener Wert nach `r3` verschoben.

Fehlerprüfungen wie beispielsweise das Verhindern eines Stackunderflows wurden in diesem Beispiel der besseren Verständlichkeit wegen weggelassen.

2.2.6 Weiterführende Arithmetik

Durch die vorgestellten Konstrukte können nun auch die arithmetischen Möglichkeiten der 4IA-Sprache erweitert werden. So ist beispielsweise eine Subtraktion $a - b$ in dieser Sprache als $a + (-1) \times b$ realisierbar, denn die Subtraktion um 1 ist durch `SET r`*x*`, 0xFF` gefolgt von `ADD r`*a*`, r`*x* und die Multiplikation durch die Kombination von Wiederholung und Auswahl gegeben. Aus der Subtraktion lässt sich analog zur Multiplikation, also mittels Wiederholung, schliesslich die Division realisieren womit die vier grundlegenden Operationen abgedeckt wären: Addition, Subtraktion, Multiplikation und Division.

2.2.7 Binäre boolsche Operationen

Binäre boolsche Operationen lassen sich in unserem Modell durch Addition des jeweils höchstwertigsten Bits pro Operand erreichen. Es entspricht nämlich das resultierende Summenbit der Bit-Addition der Operation `XOR`, das resultierende Übertragsbit (in unserem Fall das fl-Register) der Operation `AND`.

Es folgen Beispiele für die Operationen AND und XOR mit den Operanden 1 und 0. Dazu werden vorgängig die Operanden in die Register r3 und r4 geladen:

```
0:  SET r3, 0x80 ; 1. Operand: 1 (hoechstwertigstes Bit setzen)
1:  SET r4, 0    ; 2. Operand: 0
```

Es folgt die Operation AND. Im Code findet nur dann ein Additionsüberlauf statt, wenn beide Operanden 1 sind:

```
2:  SET fl, 0    ; fl-Register loeschen
3:  ADD r3, r4   ; Addition mit Ueberlauf
```

Zu diesem Zeitpunkt liegt das Resultat der Operation AND im Register fl und kann weiterverwertet werden. Ausserdem liegt in Register r3 auch bereits das Resultat der Operation XOR derselben Operanden vor.

Möchte man das XOR-Resultat nun zur Weiterverwertung beispielsweise ins fl Register schieben, so ist folgender Code denkbar:

```
4:  SET r5, 0x80 ; Konstante fuer Verschiebung laden
5:  SET fl, 0    ; fl-Register loeschen
6:  ADD r3, r5   ; Hoechstwertigstes Bit von r3 durch
                 ; Ueberlauf nach fl schieben
```

Bekanntlich lassen sich aus der einzelnen Operation NAND alle 16 möglichen binären Operationen herleiten, womit gemäss der Gleichung

a NAND b = (a AND b) XOR 1

die boolsche Logik durch das gezeigte XOR und AND komplett erschlossen ist.

2.2.8 Höhere Programmiersprachen

Das Entwerfen nichttrivialer Programme mit der in diesem Kapitel vorgestellten 4IA-Sprache ist durchaus möglich, offensichtlicherweise aber sehr aufwendig. Es ist also wünschenswert, eine Assemblersprache höheren Levels (etwa mit einem Instruktionssatz vom Umfang der in [Sch92] vorgestellen Sprache REGS) für dieses Rechenwerk nutzbar zu machen. Dazu müsste ein Compiler geschrieben werden, der die REGS-ähnliche Sprache gemäss der vorgängig beschriebenen Abbildung von höheren programmiertechnischen Konzepten in die hier vorgestellte 4- Instruktionen-Sprache übersetzen kann.

In einem weiteren Schritt wäre dann auch die Implementation eines Compilers für eine höhere Sprache der dritten Generation denkbar. Falls im Laufe der Arbeit genügend Zeit zur Verwirklichung dieses optionalen Zieles zur Verfügung steht, dürfte ein solches Vorhaben voraussichtlich auf der Basis des in der Compilerbau-Vorlesung vorgestellten YAMCC Paketes realisiert werden (vgl. [Boi01]).

2.3 Parasitäre, verteilte Berechnung

Dieser Abschnitt beschreibt das Prinzip des ,,Parasitic Computing", wie es von [BFJB01] veröffentlicht und in einer etwas abgewandelten Form in dieser Arbeit zur Anwendung kommt. Wie bereits erwähnt wurde, bildet es die arithmetische Grundlage der vorgestellten virtuellen Maschine.

2.3.1 Die Internet Checksumme

Bei der Kommunikation im Internet mittels der Standardprotokolle IP, UDP, TCP und ICMP wird durch die sogenannte Internet Checksumme (vgl. [Gro88]) die Korrektheit der Datenübertragung geprüft und sichergestellt. Zur Berechnung dieser Checksumme teilt der Absender jedes zu sendende Datenpaket in 16-Bit Worte auf, welche er binär zusammenaddiert und schliesslich das Gesamtresultat invertiert (0 wird zu 1, 1 wird zu 0). Dieses invertierte Resultat ist nun die effektive Checksumme, welche im Header des entsprechenden Datenpaketes abgelegt wird. Dargestellt ist dies in Tabelle 3.

Wert	Bedeutung
`00110111 10010001`	erstes 16-Bit Wort
`01101110 11011001`	zweites 16-Bit Wort
...	weitere 16-Bit Worte
`10111110 10010101`	Summe aller Worte
`01000001 01101010`	Komplement der Summe

Tabelle 3: Berechnung der Internet-Checksumme

Die Aufgabe des Empfängers ist es nun, das erhaltene Datenpaket wieder in 16-Bit Worte aufzuteilen und diese analog dem Absender zu addieren. Da diese Addition nun zusätzlich die vom Absender errechnete Checksumme als Summanden enthält, muss das Resultat (ohne Invertierung) logischerweise `1111111111111111` ergeben. Ist dies der Fall, so wird von einer korrekten Datenübertragung ausgegangen und die Verarbeitung kann weitergeführt werden (was sich schliesslich in einer Antwort an den Absender auf dessen Anfrage im Nutzdatenteil des gesendeten Paketes äussert). Ergibt die Summe des Empfängers nicht das erwartete Resultat, so muss – dem Protokollstandard entsprechend – von Datenkorruption ausgegangen und das fehlbare Datenpaket stillschweigend verworfen werden. Für den Absender äussert sich dies darin, dass er keine Antwort des Empfängers auf das fehlbare Paket erhält (vgl.[Ste94]). Diese Tatsache wird uns im folgenden Abschnitt von entscheidendem Nutzen sein.

2.3.2 Addition von Ganzzahlen

Unser Modell benötigt, wie bereits gezeigt, lediglich die Addition als parasitäre Grundoperation. Wir zeigen daher im folgenden, wie sich die Addition zweier Binärzahlen durch Ausnützung dieser Checksummenberechnung ausführen lässt.

Bekanntlich wird die Addition zweier Binärzahlen bei der Schulmethode so durchgeführt, dass sukzessive von rechts nach links, das heisst von der niederwertigsten zur höchstwertigsten Bitposition, jeweils die Bitwerte (an der selben Position) unter Berücksichtigung des Übertragsbits addiert werden. Die Addition einer n-stelligen Binärzahl wird somit auf n-malige Bit-Addition zurückgeführt. Für die Bit-Addition benötigen wir also drei Eingänge (die zwei Operandenbits und das vorgängige Übertragsbit) und zwei Ausgänge (das Resultatbit und das nachfolgende Übertragsbit).

Wir können uns nun ein Datenpaket vorstellen, welches in seinen Nutzdaten drei 16-Bit Worte enthält, deren jeweils niederwertigstes Bit einem dieser drei genannten Eingänge (Operand A, Operand B, Übertragsbit Ü) entspricht. Die restlichen Bits sind, wie in Tabelle 4 gezeigt, alle auf 0 gesetzt.

16-Bit Wort	Bedeutung
`00000000 0000000Ü`	Übertragsbit
`00000000 0000000A`	Operand A
`00000000 0000000B`	Operand B
`11111111 111111??`	Checksumme

Tabelle 4: Datenpaket mit Operanden

Bei Betrachtung der Checksumme dieses Paketes wird sofort ersichtlich, dass nur die zwei rechtsliegenden Bits relevant sind, die sich nämlich je nach Zustand der drei Eingänge (Ü, A und B) unterscheiden können. Ferner ist offensichtlich, dass vier Varianten von Checksummen möglich sind: 00, 01, 10 und 11. In Tabelle 5 sind alle denkbaren Varianten dargestellt.

Wenn wir nun die Möglichkeit haben, für ein konkretes Eingangstripel Ü, A und B aus diesen vier denkbaren Kandidatchecksummen die einzelne korrekte zu finden, so ist das Resultat der Addition eindeutig bestimmt. Und genau an diesem Punkt kommt das parasitäre Prinzip zum Zuge: wir schicken vier Pakete mit je einer der Kandidatlösungen an einen oder mehrere Empfänger und warten auf die aus dem davon einzigen korrekten Paket resultierende Antwort (denn nur das korrekte Paket wird, wie in 2.3.1 erläutert, auch wirklich beantwortet).

Übertrag Ü	Operand A	Operand B	Summe	Komplement (Checksumme)
0	0	0	00	11
0	0	1	01	10
0	1	0	01	10
0	1	1	10	01
1	0	0	01	10
1	0	1	10	01
1	1	0	10	01
1	1	1	11	00

Tabelle 5: Wahrheitstabelle für die Ganzzahladdition

Das rechte Bit der Spalte „Summe" in Tabelle 5 repräsentiert das Resultatbit, das linke das Übertragsbit. Letzterer Wert schlägt sich nach der Addition der beiden höchstwertigen Bits der Ganzzahladdition im Register `fl` nieder. Damit ist die Funktion `int parasit_add(int, int)` als Schnittstelle zur virtuellen Maschine definiert.

2.3.3 Zeitkomplexität

Wie wir oben gesehen haben, benötigen wir für jede Bitaddition mit Übertragsbit nur 4 Kandidaten, also für die Addition zweier n-stelliger Binärzahlen $4 \times n$ Kandidaten. Im Gegensatz dazu führt die in [BFJB01] vorgeschlagene Additionsmethode zu einem Aufwand bzw. zu einer Anzahl von Kandidaten, die exponentiell mit der Stellenzahl n wächst. Das von uns vorgeschlagene Additionsverfahren reduziert den Berechnungsaufwand gegenüber [BFJB01] ganz enorm. Ein kleiner, hier jedoch nicht ins Gewicht fallender Nachteil, betrifft die Parallelisierbarkeit. Charakteristisch für die Addition nach der Schulmethode ist die Tatsache, dass die Bit-Additionen sequentiell durchgeführt werden müssen, da eine Bit-Addition (an der Position i) das Übertragsbit der

vorangegangenen Addition (an der Position $i-1$) benötigt. Eine Parallelisierung der Addition ist daher im Gegensatz zu [BFJB01] nicht möglich. Unter Verwendung der Präfixsummation liesse sich auch für die Schulmethode eine Parallelisierung mit Aufwand *O(n)* erzielen.

2.3.4 Protokoll: ICMP

Als die vorgestellte Internet-Checksumme benutzendes Protokoll wird ICMP mit den sogenannten ICMP ECHO und ICMP ECHO_REPLY Messages verwendet (vgl. dazu auch [Pos81]) – von gängigen Betriebssystemen mit dem Befehl Ping implementiert – und nicht wie in der in [BFJB01] vorgestellten Methode TCP. ICMP ist vorzuziehen, weil es erstens einen wesentlich geringeren Protokolloverhead mit sich bringt, zweitens nicht von Protokollen höherer OSI-Layer abhängt und drittens gemäss [GF89] durch jeden Host im Internet zur Verfügung gestellt werden muss. Die in den vorgehenden Kapiteln genannten Datenpakete sind also konkret als ICMP ECHO Pakete zu verstehen.

In der folgenden Tabelle 6 erweitern wir das in Tabelle 4 gezeigte, theoretische Paket zu einem vollständigen, gültigen ICMP Paket wie es in der Implementation zur Anwendung kommen wird.

16-Bit Wort	Bedeutung
00001000 00000000	Typ (linke 8 Bit) und Code (rechte 8 Bit)
00000000 00000000	Checksumme
xxxxxxxx xxxxxx00	Identifier
xxxxxxxx xxxxxx00	Sequence-Number
...	Padding
00000000 0000000Ü	Übertragsbit
00000000 0000000A	Operand A
00000000 0000000B	Operand B

Tabelle 6: Vollständiges ICMP Paket

Es folgt eine kurze Erläuterung der einzelnen Felder (gem. [Ste94]):

- **Typ**
  Das Feld spezifiziert den Typ der ICMP Message. In unserer Anwendung ist dies immer 8 (ECHO_REQUEST).
- **Code**
  In unserer Anwendung nicht benötigt.
- **Checksumme**
  Wird zur Checksummenberechnung mit 0 initialisiert und nach deren Kalkulation durch sie ersetzt. Da wir bei der Paketbildung in dieser spezifischen Applikation auch auf die Checksummenberechnung der 14 linksliegenden Bits verzichten wollen, wird die effektive Implementation mit einer vorkalkulierten Menge an passenden Checksummen und Sequence-Number Paaren realisiert.
- **Identifier**

Üblicherweise wird dieses Feld mit der PID[4] des sendenden Prozesses gefüllt, damit der Kernel die erhaltene Antwort diesem wieder zuweisen kann. Wir benutzen hier eine Pseudo-ID um unsere Pakete bei Empfang von anderen ICMP ECHO_REPLY Meldungen unterscheiden zu können. Diese Pseudo-ID soll die für die parasitäre Addition relevanten Bits nicht tangieren.

- **Sequence-Number**
  Individuell benutzbares Feld zur Numerierung der Pakete. Hier gilt für unsere Anwendung – analog zum Identifier Feld –, die zwei rechtsliegenden Bits nicht zu verwenden.

- **Padding**
  Je nach Implementation muss das Packet mindestens 32 16-Bit Worte enthalten, daher wird ein Padding[5] mit 0 vorgenommen.

Die Restlichen drei Felder mit dem Übertragsbit und den Operanden wurden bereits beschrieben.

2.3.5 ICMP Pakete parallelisieren

Pro atomare Operation sind, wie beschrieben wurde, vier Netzwerkpakete notwendig, von welchen eines eine gültige Checksumme haben muss. Jedes dieser Pakete ist durch eine ICMP Sequence Number eindeutig identifizierbar (vgl. [Pos81]). Die vier Kandidatlösungen (Netzwerkpakete) können also parallel verschickt und überprüft werden, da anhand der Sequence Number eine zurückkehrende Antwort eindeutig der gesendeten Anfrage zugeordnet werden kann.

2.3.6 Empfänger

Die Empfänger der Datenpakete, also sozusagen die Wirte des Parasiten, werden in einer Priority-Liste geführt. Möglicherweise kann diese Liste auch mittels Broadcasts erstellt werden, sofern nur im lokalen Netz gerechnet und die erreichbaren Maschinen nicht bekannt sind. Die Reihenfolge in der Priority-Liste ergibt sich aus der durchschnittlichen Antwortgeschwindigkeit der Maschinen, welche vorgängig ebenfalls durch ICMP Messages ermittelt werden kann. Der genaue Algorithmus, wie die Kandidatlösungen auf die Empfänger aufgeteilt werden, ist nicht von entscheidender Bedeutung für die Arbeit und wird erst in der Implementationsphase entschieden.

2.3.7 Überwachung und Fehlerkontrolle

Um die Korrektheit der Berechnungen sicherzustellen, muss eine Fehlerkontrolle stattfinden. Diese wird im Minimum prüfen, ob die Anzahl retournierter Datenpakete der erwarteten (immer eines) entspricht und gegebenenfalls fehlbare Kommunikationspartner aus der genannten Priority-Liste entfernen. Falls es sich zeigt, dass die Fehlerrate generell zu gross wird, so ist auch denkbar jede als korrekt gemeldete Checksumme durch einen weiteren Kommunikationspartner zu verifizieren und bei Nichtübereinstimmung die zweifelhafte Operation zu wiederholen.

[4] PID steht für „Process-Identifier", also eine auf dem Betriebssystem eindeutige Identifikation für jeden laufenden Prozess.

[5] „auspolstern"

2.3.8 Simulationsmodus

Auf dieser untersten Ebene wird ein optional ein- und ausschaltbarer Simulationsmodus implementiert. Dabei werden wohl noch die Datenpakete wie beschrieben gebildet, allerdings nicht mehr tatsächlich über das Netzwerk gesendet, sondern lokal auf gültige Checksummen geprüft. Dieser Modus soll vorallem in der Entwicklungsphase dienlich sein, ist aber auch für die Fehlersuche unverzichtbar.

3 Design Ergebnisse

Dieses Kapitel beschreibt die Umsetzung des Analysemodells aus Kapitel 2 in ein Designmodell, welches bereits bei der Implementation der Prototypen so weit als möglich berücksichtigt wurde. Das Schwergewicht in der Beschreibung widmet sich in erster Linie der Aufteilung der Funktionszuständigkeiten in die drei genannten Schichten sowie der Faktorisierung der Applikation in geeignete Module.

3.1 Systemarchitektur

In diesem Abschnitt werden technische Aspekte der konkreten Umsetzung wie beispielsweise das verwendete Betriebssystem, die Programmiersprachen und so weiter erläutert und die Wahl jeweils begründet.

3.1.1 Betriebssystem

Die Entwicklung der Applikation erfolgt unter UNIX mit Unterstützung möglichst aller gängigen UNIX-Systeme sowie Linux. Microsoft Windows, Apple Macintosh und andere Architekturen werden nicht berücksichtigt, da sich diese in einem für die Applikation äusserst zentralen Punkt, nämlich dem direkten Socket-Zugriff, sehr stark unterscheiden. Zum momentanen Zeitpunkt ist dafür kein systemübergreifendes API bekannt.

3.1.2 Programmiersprachen

Die Implementation des parasitären Teils sowie der virtuellen Maschine wird in der Programmiersprache C durchgeführt. Dies primär aus dem Grund, weil C mit den unter UNIX standardisierten Bibliotheken praktisch die einzige Möglichkeit darstellt, direkten Socket-Zugriff (BSD Raw-Sockets) zu erzielen.

Compiler für eine höhere Assemblersprache oder Compiler für eine Hochsprache werden mit JAVA in Kombination mit dem JAVA Compiler Compiler (vgl. [Web02]) entwickelt.

3.1.3 Netzwerk

Die unabdingbare Voraussetzung der Applikation, Operationen verteilt berechnen zu können, ist die Verbindung des ausführenden Rechners zu einem oder mehreren anderen Maschinen, welche die generierten in ICMP Pakete codierten Kandidatlösungen auswerten und gegebenenfalls beantworten.

Folgende Möglichkeiten werden primär zur Auswertung von ICMP Paketen in Betracht gezogen:

- **Simulation**
  Simulation ohne Netzwerkzugriff zur autonomen Berechnung von Checksummen der ICMP Pakete und entsprechender Antwort. Dies ermöglicht die Entwicklungsarbeit ausserhalb eines Netzwerkes.

- **Localhost**
  Effiziente und von Netzwerkproblemen unabhängige Variante, ICMP Pakete über das Loopback-Interface des ausführenden Hosts auszuwerten.

- **LAN**
  Nicht sonderlich performante aber problemarme Variante, mit welcher die Applikation im Gegensatz zu den vorgängig genannten Berechnungsmethoden bereits auf ein funktionierendes Netzwerk mit mindestens einem weiteren verfügbaren Rechner angewiesen ist. Dafür werden bei dieser Variante aber bereits Rechenkapazitäten im Sinne von parasitärer Berchnung genutzt, wenn auch nur im lokalen Netzwerk.

- **Internet**
  Ineffizienteste Variante der Auswertung von ICMP Paketen mit grösserer Fehleranfälligkeit und der Bedingung, dass eine zuverlässige Verbindung ans Internet gewährleistet ist. Mit dieser Art der Berechnung kommt der Ansatz und der Grundgedanke der parasitären Nutzung von Rechenkapazität voll zur Geltung.

3.2 Module

Die nachfolgend dargestellte Abbildung 2 soll einen Überblick über das Framework der Applikation mit den zugrundeliegenden Modulen verschaffen.

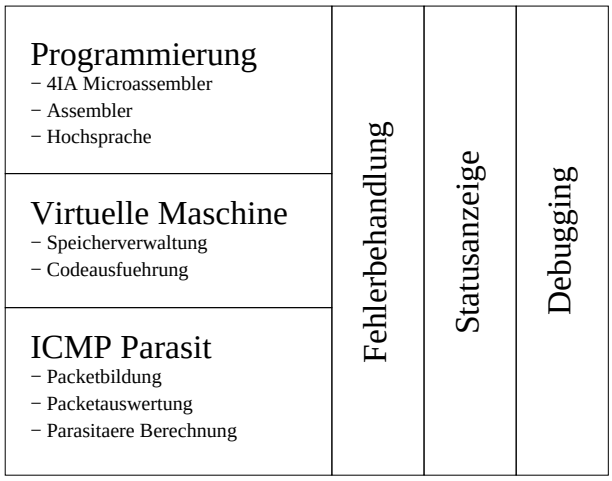

Abbildung 2: Framework der Applikation

4 Realisierungsvorhaben

In diesem Kapitel wird beschrieben, welche Ansätze angewendet und in welcher Reihenfolge diese für die Realisierung der Applikation entwickeln werden sollen. Die einzelnen Ansätze beziehen sich auf die vorangegangenen Kapitel.

4.1 Priorisierung

In nachfolgender Tabelle sind die in Kapitel 3 erläuterten Module den Entwicklungszeitpunkt betreffend nach Wichtigkeit und Priorität klassifiziert.

Modul Name	Priorität	Relisierungsschritt
Virtuelle Maschine, Speicherverwaltung, Codeausführung, Eingabe, Ausgabe	hoch	erste Phase
Paketbildung, ICMP-Simulation	hoch	erste Phase
4IA-Microassembler	hoch	erste Phase
parasitäre ICMP-Berechnung, Fehlerbehandlung	mittel	zweite Phase
Hosts Priority-Queue, Statusanzeige	mittel	zweite Phase
Höhere Assemblersprache, Visualisierung, Hochsprache	gering	optionale dritte Phase

Tabelle 7: Priorisierung der Teilarbeiten

Erstes Ziel ist es, die Applikation möglichst früh als Prototypen in Betrieb zu nehmen und dann in einem iterativen Entwicklungssyklus laufend zu erweitern.

4.2 Optionale Ziele

Folgende Auflistung soll kurz die optionalen Ziele darstellen, welche nur bei Vorhandensein genügender Zeit realisiert werden:

- **Höhere Assemblersprache**
  Dieser Punkt wurde bereits in Abschnitt 2.2.8 auf Seite 37 erläutert.
- **Visualisierung**
  Ein graphisches Interface zur Visualisierung des Ablaufes (Registerinhalt, gesendete Netzwerkpakete, etc.), welches mit der virtuellen Maschine beispielsweise über ein Logfile unidirektional oder über einen Socket bidirektional kommuniziert, ist als optionales Ziel denkbar.
- **Hochsprache**
  Dies wurde bereits behandelt; eine Hochsprache zur Programmierung der parasitären Maschine, voraussichtlich auf Basis des YAMCC Paketes aus der Compilerbau Vorlesung der HTA-BE (vgl. [Boi01]), ist als optionales Ziel denkbar.

4.3 Risiken

Durch geeignete Priorisierungen der zu entwickelnden Module konnte das Risiko, die Projektziele zu verfehlen, stark minimiert werden. Das einzige verbleibende, projekt-

gefährdende Restrisiko besteht darin, dass wir zur Implementationszeit auf noch nicht erkannte, gravierende Probleme stossen könnten.

PARASITIC COMPUTING
Handbuch

Version 1.0

Jürg Reusser
Luzian Scherrer

5. Januar 2003

Zusammenfassung

Dieses Dokument dient als Benutzerhandbuch für die im Rahmen der Diplomarbeit „Parasitic Computing" entwickelten Softwarekomponenten. Es beginnt bei der Kompilation und Installation der Programme und umfasst im Weiteren die Bedienung der `pshell`-Umgebung und des `xto4` Cross-Compilers. Den Hauptteil bilden die Beschreibungen der beiden Sprachen 4IA (4 Instruction Assembler) und XIA (Extended Instruction Assembler), welche zur Programmierung der virtuellen Maschine verwendet werden.

Inhaltsverzeichnis

1 Distribution

In den folgenden Unterkapiteln wird erläutert, wie die Source-Codes vom Internet (bzw. von der beiliegenden CD-ROM) heruntergeladen, entpackt, übersetzt und installiert werden können. Die Übersetzung ist dabei optional; die Softwarepakete sind sowohl im Quellcode wie auch als vorkompilierte Binärdateien erhältlich.

1.1 Download

Alle Source-Codes sowie die fertig übersetzten und direkt installierbaren Pakete können unter folgender Adresse bezogen werden:

```
http://parasit.org/code
```

Die beiliegende CD-ROM enthält einen zur Onlineversion identischen Abzug. Der Zugriff darauf erfolgt mittels Webbrowser über die Datei `index.html`, welche sich im Wurzelverzeichnis befindet.
Das Source-Code Paket enthält alle Komponenten. In den Binärversionen sind die `pshell`-Umgebung und der `xto4` Cross-Compiler getrennt verfügbar, da sie in verschiedenen Programmiersprachen realisiert sind. Die genannte Website gibt einen genauen Überblick über alle Komponenten einschliesslich einer kurzen Beschreibung.

1.2 Entpacken

Die Distributionen sind mit GNU Gzip[1] und Tar[2] komprimiert. Zum entpacken dient folgende Zeile:

```
gzip -dc <filename> | tar xvf -
```

Daraus resultiert ein dem Distributionsnamen entsprechendes, temporäres Verzeichnis, in welchem die nachfolgend beschriebene Übersetzung und Installation ausgeführt wird.

2 Installation

Dieses Kapitel beschreibt die Kompilation und Installation von Source-Code und Binärdateien. Bei den Binärdateien entfällt der Kompilationsschritt.

2.1 Systemanforderungen

Dieser Abschnitt gibt einen kurzen Überblick über die zur Übersetzung und dem Betrieb benötigte Software.

[1] siehe `http://www.gnu.org/software/gzip/`
[2] siehe `http://www.gnu.org/software/tar/`

2.1.1 Betriebssysteme

Es werden folgende Betriebssysteme unterstützt:

- GNU Linux
- Sun Microsystems Solaris
- Silicon Graphics IRIX

Der `xto4` Cross-Compiler, eine reine Java Applikation, kann zusätzlich auch auf allen weiteren Plattformen eingesetzt werden, welche ein Java Runtime Environment zur Verfügung stellen.

2.1.2 Software zur Übersetzung

Die nachfolgend aufgelistete Software muss zur Übersetzung installiert sein. Es handelt sich dabei um Standardkomponenten, welche auf einem gängigen Entwicklungssystem bereits vorhanden sein sollten. Die zur Entwicklung benutzten Versionen sind jeweils in eckigen Klammer angegeben; in Problemfällen ist auf diesbezügliche Versionskompatibilität zu achten.

- C Compiler [3.0.3]
  `http://gcc.gnu.org/`
- Flex (Fast Lexical Analyser Generator) [2.5.4]
  `http://www.gnu.org/software/flex/`
- GNU Readline Library [4.3]
  `http://cnswww.cns.cwru.edu/~chet/readline/rltop.html`
- Ncurses (new curses) [5.2]
  `http://www.gnu.org/software/ncurses/ncurses.html`
- Java Compiler [1.3.1_03]
  `http://Java.sun.com`
- ANT [1.5.1]
  `http://jakarta.apache.org/ant/`

2.2 Übersetzung

Dieser Abschnitt kann für die Installation von Binärdateien übersprungen werden. Die Übersetzung erfolgt mittels dem im Paket enthaltenen `Makefile`. Der Übersetzungsprozess wird mit folgendem Befehl ausgeführt:

```
make
```

Bei erfolgreicher Übersetzung endet der Prozess mit der Meldung:

```
Compilation finished; all done.
```

2.3 Installation

Die Installation geschieht durch das im Paket enthaltene `Makefile`. Sie wird mit folgendem Befehl ausgeführt:

```
make install
```

Das Zielverzeichnis der Installation ist `/usr/local`, wobei alle Binärdateien nach `/usr/local/bin` und Codebeispiele nach `/usr/local/share/parasit` installiert werden. Es ist vor Ausführung der Installation darauf zu achten, dass Schreibrechte im Zielverzeichnis existieren. Falls die Software in ein anderes als das vorgegebene Zielverzeichnis installiert werden soll, so kann im `Makefile` die Variable `PREFIX` entsprechend angepasst werden.

Nach der Installation sind folgende Dateien vorhanden:

- `/usr/local/bin/pshell`
  Das `pshell` Executable
- `/usr/local/bin/xto4`
  Das `xto4` Executable (Wrapper-Script)
- `/usr/local/bin/Xto4.jar`
  Das `xto4` JAR Archive
- `/usr/local/share/parasit/4ia/`...
  Codebeispiele in der 4IA-Sprache
- `/usr/local/share/parasit/xia/`...
  Codebeispiele in der XIA-Sprache

3 Betrieb

Die folgenden Unterkapitel erklären, wie die installierten Pakete betrieben und mit den entsprechenden Programmiersprachen gearbeitet werden kann.

3.1 Die `pshell`-Umgebung

Die `pshell`-Umgebung bildet den Kern und das primäre Benutzerinterface zur parasitären virtuellen Maschine. Sie verfügt über ein eingebautes Hilfe-System, welches mit dem Befehl `help` angezeigt wird. Die `pshell`-Umgebung wird mit folgendem Befehl gestartet:

```
pshell
```

Da die `pshell` auf der GNU Readline Library basiert, entsprechen die Kommandoeingabe und die Zeileneditierungsmöglichkeiten dem bekannten Prinzip anderer UNIX Shells, wie beispielsweise der `bash`. Das GNU Readline Manual gibt einen Überblick über die vielfältigen Möglichkeiten:

`http://cnswww.cns.cwru.edu/~chet/readline/rluserman.html`

Auf den nächsten Seiten werden in alphabetischer Reihenfolge die `pshell` spezifischen Befehle im Detail erläutert.

3.1.1 Edit

Beschrieb

Edit startet den durch die Umgebungsvariable `EDITOR` definierten Editor mit der als `filename` angegebenen Datei als Argument. Dieser Befehl dient dazu, direkt aus der `pshell`-Umgebung 4IA-Programmcode oder eine Hostliste zu erstellen oder editieren.

Syntax

```
edit <filename>
```

Kurzform

```
ed <filename>
```

Argumente

Das Argument `<filename>` entspricht der zu editierenden Datei.

3.1.2 Execp

Beschrieb

Execp führt das angegebene 4IA-Programm im parasitären Betriebsmodus aus. Weil zur Ausführung dieser Aktion Raw-Sockets vom Kernel angefordert werden müssen, steht das Kommando `execp` nur dem Benutzer `root` mit UID 0 zur Verfügung.

Vor der parasitären Ausführung eines Programmes muss mit dem Befehl `loadhosts` eine Liste von Hosts geladen werden. Mit diesen wird die Berechnung ausgeführt.

Weitere beeinflussende Werte der parasitären Ausführung, können vorgängig mit den Befehlen `setbits`, `settimeout` und `setthres` definiert werden.

Die aktuelle Programmausführung kann jederzeit duch das Senden eines SIGINT Signales unterbrochen werden. SIGINT entspricht üblicherweise der Tastenkombination `CTRL` + `c`, kann aber auf durch den Befehl `kill -INT` <`pid`> gesendet werden.

Syntax

```
execp <file.4ia>
```

Kurzform

```
xp <file.4ia>
```

Argumente

Das Argument `<file.4ia>` entspricht dem Pfad einer Datei, welche in der 4IA-Sprache geschriebenen Programm-Code enthält.

3.1.3 Execs

Beschrieb

Execs führt das angegebene 4IA-Programm im Simulationsmodus aus. Diese Möglichkeit steht allen Benutzern zur Verfügung und erfordert keine speziellen Berechtigungen.

Die aktuelle Programmausführung kann jederzeit duch das Senden eines SIGINT Signales unterbrochen werden. SIGINT entspricht üblicherweise der Tastenkombination `CTRL` + `c`, kann aber auf durch den Befehl `kill -INT <pid>` gesendet werden.

Syntax

```
execs <file.4ia>
```

Kurzform

```
xs <file.4ia>
```

Argumente

Das Argument `<file.4ia>` entspricht dem Pfad einer Datei, welche in der 4IA-Sprache geschriebenen Programm-Code enthält.

3.1.4 Help

Beschrieb

Help zeigt die Kurzübersicht aller verfügbaren Befehle an.

Syntax

```
help
```

Argumente

Keine Argumente.

3.1.5 History

Beschrieb

History zeigt eine Liste der bisher eingegebenen Befehle an.

Syntax

```
history
```

Kurzform

```
h
```

Argumente

Keine Argumente.

3.1.6 Loadhosts

Beschrieb

Loadhosts lädt eine Liste von Hostnamen (oder IP-Adressen) zur parasitären Ausführung. Je nach Grösse der Hostliste kann die Ausführung dieses Befehls einige Sekunden in Anspruch nehmen, weil die Hostnamen zum Zeitpunkt des Aufrufes direkt von der Resolver-Library aufgelöst werden.

Die Hostliste ist eine durch Zeilenumbrüche separierte Liste nach folgendem Beispiel:

```
www.isbe.ch
parasit.org
192.168.100.2
www.google.com
```

Syntax

```
loadhosts <filename>
```

Kurzform

```
lh <filename>
```

Argumente

Das Argument `<filename>` entspricht dem Pfad zu einer Datei, welche eine Liste von Hostnamen (oder IP-Adressen) enthält.

3.1.7 Quit

Beschrieb

Beenden der `pshell`.

Syntax

```
quit
```

Kurzform

```
q
```

Argumente

Keine Argumente.

3.1.8 Setbits

Beschrieb

Setbits setzt die Anzahl der parallel zu addierenden Bits für die nächste parasitäre Ausführung. Es werden pro Sendeimpuls soviele ICMP-Pakete verschickt, wie notwendig sind, um eine Addition von der angegebenen Bitbreite auszuführen. Da alle Additionen somit in Teiladditionen der angegebenen Bitbreite aufgeteilt werden, muss diese ein ganzer Teiler der definierten Registerbreite sein.

Syntax

```
setbits <number>
```

Kurzform

```
sb <number>
```

Argumente

Das Argument `<number>` ist eine der folgenden Konstanten: 1, 2, 4.

3.1.9 Settimeout

Beschrieb

Settimeout definiert die Anzahl Sekunden, die maximal auf eine ICMP Antwort gewartet werden soll, bevor die Operation wiederholt und der den Timeout verursachende Host entsprechend markiert wird. Der definierte Wert kann mit dem Befehl `showconfig` abgefragt werden.

Syntax

```
settimeout <number>
```

Kurzform

```
sti <number>
```

Argumente

Das Argument `<number>` ist eine Konstante grösser als 0; die Einheit sind Sekunden.

3.1.10 Setthreshold

Beschrieb

Auf die Antwort eines Sendeimpulses wird pro Host um den durch `settimeout` definierten Timeout abgewartet. Falls innerhalb dieser Frist keine Antwort eintrifft, wird ein Timeout-Zähler inkrementiert als Attribut des den Timeout verursachenden Hosts. Wenn dieser Timeout-Zähler die angegebene Höchstgrenze `setthres` erreicht, so wird der entsprechende Host für weitere Berechnungen nicht mehr benutzt.

Der Zustand des Timeout-Zählers kann mit dem Befehl `showhosts` abgefragt werden.

Syntax

```
setthres <number>
```

Kurzform

```
str <number>
```

Argumente

Das Argument `<number>` ist eine Konstante grösser als 0.

3.1.11 Showconfig

Beschrieb

Showconfig zeigt die Konfiguration der `pshell`-Umgebung an. Die angezeigten Werte könnnen an Hand von den Befehlen `settimeout`, `setthres` und `setbits` verändert werden.

Syntax

```
showconfig
```

Kurzform

```
sc
```

Argumente

Keine Argumente.

3.1.12 Showhosts

Beschrieb

Showhosts zeigt die durch den Befehl `loadhosts` geladenen Hosts und deren Stati an. Beispielsweise die Anzahl gesendeter ICMP Pakete oder die Anzahl eingetroffener Timeouts, an.

Um die Hostliste und somit die Stati pro Host neu zu Initialisieren (was bei Codeausführung nicht ausdrücklich geschieht), kann die Hostliste mit dem Befehl `loadhosts` erneut geladen werden. Dadurch werden alle Zähler und Zustandsvariablen auf deren Initialwerte zurückgesetzt.

Syntax

```
showhosts
```

Kurzform

```
sh
```

Argumente

Keine Argumente.

3.1.13 Showregisters

Beschrieb

Dieser Befehl bietet die einzige Möglichkeit zur Auswertung der Resultate der ausgeführten Programme. Es können sämtliche Register der virtuellen Maschine aufgelistet werden.

Syntax

```
showreg [reg-list]
```

Kurzform

```
sr [reg-list]
```

Argumente

Das optionale Argument `[reg-list]` ist eine space-separierte Liste von Registernamen. Wird das Argument weggelassen, so zeigt der Befehl alle Register an.

Um nur eine Teilmenge aller Register anzuzeigen:

```
showreg r10 r12 r15 r20
```

Um alle Register anzuzeigen:

```
showreg
```

3.1.14 Statistics

Beschrieb

Statistics zeigt eine Statistik über den letzten Programmablauf der virtuellen Maschine an.

Bei dem Wert „Anzahl lokal benötigte Prozessorzyklen" handelt es sich um eine Approximation; Dieser Wert kann nicht mit absoluter Genauigkeit ermittelt werden.

Die verschiedenen Ausführungszeiten (User, Kernel, Total) arbeiten ebenfalls mit begrenzter Auflösung. So kann es vorkommen, dass bei einem sehr kurzen Programm die Ausführungszeit 0 Sekunden beträgt. Die Auflösung dieser Werte entspricht der Genauigkeit, welche vom Betriebssystem zur Verfügung gestellt wird.

Syntax

```
stats
```

Kurzform

```
ss
```

Argumente

Keine Argumente.

3.1.15 Eine `pshell` Beispielssitzung

Um den Einstieg zu erleichtern, wird eine Beispielssitzung in der `pshell` dargestellt:

```
Parasitic Computing (pshell 1.0)
Copyright (c) 2002 Juerg Reusser, Luzian Scherrer
Determining CPU clockspeed... 167.00 Mhz
Type help for help.
> ed /tmp/myhosts
```

Das File `/tmp/myhosts` wird mit folgendem Inhalt erstellt:

```
www.isbe.ch
www.google.com
www.microsoft.com
```

Danach wird diese Liste von Hosts geladen und verifiziert:

```
> lh /tmp/myhosts
Hostlist loaded
> sh
```

Nun wird das Beispielprogramm `binary_and.4ia` zuerst als Simulation, dann parasitär ausgeführt:

```
> xs /usr/local/share/code/4ia/binary_and.4ia
Execution successfully terminated.
> xp /usr/local/share/code/4ia/binary_and.4ia
Execution successfully terminated.
```

Darauffolgend können nun Statistiken über die Ausführung und der Zustand der Register ausgelesen werden:

```
> sr
> ss
```

Danach wird die `pshell` Sitzung beendet:

```
> quit
```

3.2 Der `xto4` Cross-Compiler

Der `xto4` Cross-Compiler wird mit dem Wrapper-Script `xto4` gestartet[3]:

```
xto4
```

Folgende Information wird bei Programmstart ausgegeben:

```
Parasitic Computing, Xto4 Cross-Compiler (Xto4 1.0)
Copyright (c) 2002 Luzian Scherrer, Juerg Reusser
Check out http://www.parasit.org for information.

Usage: xto4 <input.xia> [output.4ia]
```

3.2.1 Übergabeparameter

Als erstes Argument <`input.xia`> wird die zu kompilierende, den 4IA-Code enthaltende Datei übergeben; der Dateinamen kann dabei als absoluter oder relativer Pfad angegeben werden. Bei Dateinamen ohne Pfadangabe wird das aktuelle Verzeichnis abgesucht.

Das zweite optionale Argument `[output.4ia]` definiert, in welche Datei der kompilierte XIA-Code geschrieben werden soll. Diese Datei wird im Filesystem neu erstellt. Die Angabe ist optional; erfolgt sie nicht, so wird der generierte Code auf die standard Ausgabe geschrieben.

4 Programmiersprachen

Folgende Unterkapitel beschreiben die Syntax und die Semantik der beiden Programmiersprachen und zeigen neben diversen Programmbeispielen auch Möglichkeiten, wie Effizienzsteigerungen und weitere Optimierungen vorgenommen werden können.

4.1 Allgemeines

Die Programmiersprachen 4IA und XIA besitzen diverse gemeinsame Eigenschaften, die in den folgenden Abschnitten aufgezeigt werden.

4.1.1 Hierarchischer Aufbau

Der hierarchische Aufbau der Sprachen und der dazugehörigen virtuellen Maschine ist in Abbildung 1 ersichtlich. Die zuoberst dargestellte Sprache der dritten Generation wurde im Rahmen dieses Projektes nicht umgesetzt.

[3] Falls das Paket auf einem anderen als den ausdrücklich unterstützten Systemen installiert wurde, kann das Wrapper-Script möglicherweise nicht benutzt, der Cross-Compiler aber trotzdem „direkt" gestartet werden: `Java -jar Xto4.jar`

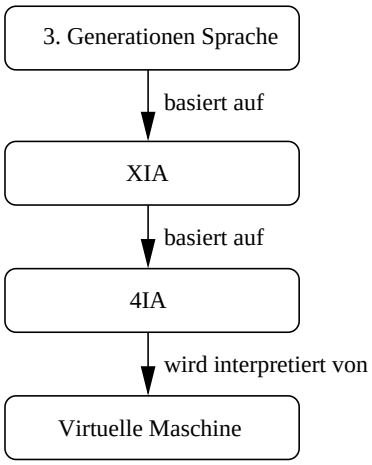

Abbildung 1: Hierarchischer Aufbau von `pshell`, 4IA und XIA

4.1.2 Konstanten

In beiden Programmiersprachen 4IA und XIA werden Konstanten verwendet, welche folgende Kriterien erfüllen:

- Positive ganze Zahl
- Wertebereich zwischen 0 und $2^{Registerbreite} - 1$
- Zahlensystem dual, dezimal oder hexadezimal

Der syntaktische Aufbau einer Konstante in EBNF-Notation lautet wie folgt:

```
((0d)?[0-9]+)    |          ; decimal
(0b[01]+)        |          ; binary
(0x[0-9a-fA-F]+)            ; hexadecimal
```

Um also beispielsweise die Zahl 213 darzustellen, sind folgende Varianten möglich:

```
213
0d213
0xD5
0xd5
0b11010101
```

4.1.3 Speicher und Adressierungsarten

Beide Sprachen sind Registermaschinensprachen: Die einzige Möglichkeit des Speicherzugriffes geschieht über eine (unbegrenzte) Anzahl von Registern. Registernamen beginnen immer mit einem kleinen `r` gefolgt von einer Ganzzahl, welche der Nummer des Registers entspricht. Wird dem Registernamen ein `*` vorangestellt, so dient das Register der indirekten Adressierung und referenziert dasjenige Register, dessen Nummer

es enthält. Enthält beispielsweise das Register `r12` den Wert 17, so ist der Ausdruck `*r12` gleichwertig zum Ausdruck `r17`.

In beiden Programmiersprachen gibt es eine begrenzte Anzahl an Spezialregistern, welche bestimmte Zustände und Metainformationen der virtuellen Maschine widerspiegeln. Auf diese Register wird in den entsprechenden Abschnitten der einzelnen Sprachen genauer eingegangen.

Alle Register werden bei Programmstart immer mit dem Wert 0 initialisiert.

4.2 Die 4IA-Sprache

Die 4IA-Sprache entspricht in ihrem Instruktionsumfang den vier Operationen der virtuellen Maschine. In dieser Sprache kann, wie bereits im Realisierungskonzept gezeigt wurde, sämtliche Programmlogik abgebildet werden.

4.2.1 Spezielle Register

Folgende zwei Spezialregister sind für die 4IA-Sprache von Bedeutung:

- **IP** (Instruktionszeiger-Register)
  Das Instruktionszeiger-Register enthält die Zeilennummer des jeweils zur Ausführung stehenden Codes und wird von der virtuellen Maschine in jedem Zyklus implizit inkrementiert. Um in 4IA den gegebenen sequentiellen Programmfluss zu alternieren (beispielsweise um Sprünge auszuführen), kann das Instruktionszeiger-Register entsprechend beeinflusst werden. Es ist entweder unter dem Namen `r0` oder dem Alias-Namen `ip` ansprechbar. Im Realisierungskonzept finden sich ausführliche Beispiele dazu.

- **FL** (Flag-Register)
  Das Flag-Register (`r1` oder `fl`) steht dem Programmierer als „normales" Register zur Verfügung. Lediglich nach einer Addition (Instruktion `ADD`) hat es eine besondere Bedeutung : Verursacht eine Addition einen Überlauf, so wird das Flag-Register auf den Wert 1 gesetzt. Findet kein Überlauf statt, so bleibt das Register unverändert. Ein Überlauf resultiert aus einer Addition, deren Resultat grösser als $2^{Registerbreite} - 1$ wäre, wobei das effektive Resultat der Addition in einem solchen Fall modulo $2^{Registerbreite}$ entspricht.

Die restlichen Register `r2` bis `rn` haben keine speziellen Funktionen und können uneingeschränkt verwendet werden.

4.2.2 Zeilenaufbau

Der grundlegende Aufbau einer Zeile 4IA-Code entspricht folgendem Muster:

```
ZEILENNUMMER:  INSTRUKTION <ARGUMENTE>   ; KOMMENTAR
```

Dabei ist die Zeilennummer eine optionale Angabe; Sie hat für die Sprache keine Bedeutung, empfiehlt sich aber als Programmierhilfe zur Berechnung von Sprüngen. Sprünge werden in der 4IA-Sprache, wie oben erwähnt, durch absolute Adressierung mittels des Instruktionszeigers ausgeführt.

4.2.3 EBNF

Die 4IA-Sprache ist syntaktisch durch folgende EBNF-Notation definiert, wobei `4ia` das Startsymbol darstellt.

```
4ia         ::= arch (line)* ;
arch        ::= "ARCH" dec dec ;
const       ::= dual | dec | hex ;
alpha       ::= [a-zA-Z] ;
numeric     ::= [0-9]+ ;
otherchar   ::= [#@+*/^?!><.,;:()[]{}' ']
comment     ::= ";" (otherchar | alpha | numeric)* ;
dual        ::= (0b[0,1]+) ;
dec         ::= ((0d)?[0-9]+) ;
hex         ::= (0x[0-9a-fA-F]+) ;
line        ::= ( numeric ":" )?
                instruction (comment)?
reg         ::= dirreg | indirreg | specialreg;
dirreg      ::= "r"  (numeric)+ ;
indirreg    ::= "*r" (numeric)+ ;
specialreg  ::= "ip" | "fl" ;
instruction ::= set | mov | hlt | add ;
set         ::= "SET" (dirreg | specialreg ) ","
                const ;
mov         ::= "MOV" reg "," reg ;
hlt         ::= "HLT" ;
add         ::= "ADD" (dirreg | specialreg) ","
                (dirreg | specialreg);
```

4.2.4 Architekturdefinition

Wie aus der EBNF Darstellung ersichtlich ist, beginnt jedes 4IA-Programm mit der Instruktion `ARCH`. Diese definiert dynamisch die Architektur der virtuellen Maschine für das nachfolgende Programm. Die beiden Argumente dieser Spezialinstruktion bestimmen die Registerbreite und die Anzahl verfügbarer Register. So ist beispielsweise die Definition

```
ARCH 8 15
```

an den Beginn eines Programmes zu setzen, welches eine Registerbreite von 8 Bit und eine Anzahl von 15 Registern benötigt.

4.2.5 Fehlererkennung

Bei der Ausführung von 4IA-Programmen in der `pshell`-Umgebung kommen zwei verschiedene Fehlererkennungsmechanismen zum tragen:

- **Übersetzungsfehler**
  Während der Übersetzungsphase wird der Code auf syntaktische, und logische

Programmierfehler überprüft. Wird ein solcher gefunden, so bricht der Übersetzungsprozess mit einer den Fehler und dessen Zeile im Programmcode referenzierenden Meldung ab.

- **Laufzeitfehler**
  Zur Laufzeit sind zwei Arten von Fehlern möglich, welche nicht in der Übersetzungsphase erkannt werden: Einerseits der Zugriff auf nichtexistierende Register mit hilfe von indirekter Adressierung, andereseits des Setzen des Instruktionszeigers auf einen Wert ausserhalb des gültigen Bereiches. Beide Fehler werden von der virtuellen Maschine abgefangen und führen zu einem Programmabbruch mit entsprechender Meldung.

4.2.6 Instruktionsreferenz: ARCH

Beschrieb

Definiert die Architektur der virtuellen Maschine. *arg1* entspricht der geforderten Registerbreite, *arg2* der Anzahl Register.

Syntax

ARCH *arg1 arg2*

Argumente

arg1	Registerbreite
arg2	Anzahl Register

Rückgabewert

Unverändert

Flag

Alle Flags bleiben unverändert.

Beispiele

Programmausführung mit 15 Registern von jeweils 8-Bit Breite:

```
ARCH 8, 15    ; Architekturdefinition
              ; (8 Bits, 15 Register)
```

4.2.7 Instruktionsreferenz: SET

Beschrieb

Die Set Instruktion weist dem direkt adressierten Register *arg1* den Wert der Konstanten *arg2* zu.

Syntax

`SET` *arg1* , *arg2*

Argumente

arg1	Das direkt adressierte Register, welchem der Wert von *arg2* zugewiesen werden soll.
arg2	Eine Konstante, welche ins Register *arg1* geladen werden soll.

Rückgabewert

arg1	Dem Register wird der Wert von *arg2* zugeordnet.

Flag

Alle Flags bleiben unverändert.

Beispiele

Dem Register `r5` den Wert 4 zuweisen:

```
SET r5, 4  ; Ordne Register r5 den Wert 4 zu
```

Dem Instruktionszeiger den Wert 11 zuweisen:

```
SET ip, 11 ; Sprung an Position 11+1
           ; (Das +1 resultiert durch die implizite
           ; Inkrementierung des Instruktionszeigers
           ; pro Zyklus der virtuellen Maschine)
```

4.2.8 Instruktionsreferenz: MOV

Beschrieb

MOV kopiert den Wert eines direkt oder indirekt adressierten Registers *arg2* in ein direkt oder indirekt adressiertes Register *arg1* .

Syntax

`MOV` *arg1* , *arg2*

Argumente

arg1	Das direkt oder indirekt adressierte Register, in welches der Wert von *arg2* kopiert werden soll.
arg2	Das direkt oder indirekt adressierte Register, dessen Wert nach *arg1* kopiert werden soll.

Rückgabewert

arg1	Dem Register wird der Wert von *arg2* zugeordnet.

Flag

Alle Flags bleiben unverändert.

Beispiele

Den Inahlt von Register `r44` nach Register `r6` kopieren:

```
MOV r6, r44    ; Kopiere Inahlt von r44 nach r6
```

Dem Register r22 den Wert 1 zuweisen mittels indirekter Adressierung:

```
SET r5, 1
SET r6, 22
MOV *r6, r5
```

4.2.9 Instruktionsreferenz: ADD

Beschrieb

ADD addiert den Inhalt zweier Register und legt das Resultat im ersten Register ab. Falls ein Additionsüberlauf stattfindet, wird dies im Carry-Flag `fl` ersichtlich.

Syntax

`ADD` *arg1* , *arg2*

Argumente

arg1	Summand; direkt adressiertes Register.
arg2	Summand; direkt adressiertes Register.

Rückgabewert

arg1	Die Summe der Addition wird in Register *arg1* abgelegt.

Flag

`cf`	Das Carry-Flag Register wird 1 gesetzt, falls die Instruktion einen Überlauf verursachte. Andernfalls wird das `cf` nicht verändert.

Beispiele

Eine Addition $r5 = 3 + 4$. Das Carry-Flag bleibt in diesem Fall unverändert.

```
SET r5  3    ; Dem Register den Wert 3 zuordnen
SET r7  4    ; Dem Register den Wert 4 zuordnen
ADD r5, r7   ; Addition von r5 mit r7
```

4.2.10 Instruktionsreferenz: HLT

Beschrieb

HLT veranlasst die virtuelle Maschine zum Stopp. Sämtliche Resultate sowie die eventuell gesetzten Flags können danach ausgewertet werden. Die Instruktion `HLT` ist am logischen Ende eines Programmes notwendig.

Syntax

`HLT`

Argumente

Keine

Rückgabewert

Unverändert

Flag

Alle Flags bleiben unverändert.

Beispiele

Nach der Addition $r4 = 4 + 5$ wird die virtuelle Maschine mit `HLT` gestoppt:

```
SET r4, 4   ; Dem Register den Wert 4 zuordnen
SET r5, 5   ; Dem Register den Wert 5 zuordnen
ADD r4, r5  ; Register r4 addieren mit r5
HLT
```

4.2.11 Programmbeispiel: Division

Im Folgenden wird ein 4IA-Programm zur Durchführung von Divisionen gezeigt. Das Programm ist in der Distribution enthalten:

```
;
; $Id: Handbuch.tex,v 1.33 2003/01/05 17:23:13 jr Exp $
;
; Ganzzahldivision mit Dividend in r10, Divisor in r11, Resultat in r7 und
; Divisonsrest in r8 (oder in Registern ausgedrueckt: r10/r11 = r7 Rest r8).
; Ein Divisor von 0 ist nicht zulaessig und wuerde in einer Endlosschlaufe
; resultieren.
;
;.....................................................................

ARCH 8 12          ; Architekturdefinition (8 Bits, 12 Register)

00:  SET r10, 110  ; Dividend
01:  SET r11, 12   ; Divisor (darf nicht 0 sein!)
02:  SET r4, 0xFF  ; Subtraktionskonstante
03:  SET r5, 1     ; Additionskonstante
04:  MOV r6, r11   ; Innerer Schlaufenzaehler initialisieren (r6)
05:  SET ip, 9     ; An ende der inneren Schlaufe springen
06:  SET fl, 0     ; Flag loeschen
07:  ADD r10, r4   ; Dekrement Dividend
08:  ADD ip, fl    ; Bei Dividend > 0 weiter
09:  SET ip, 16    ; Bei Dividend == 0 ende
10:  SET fl, 0     ; Flag loeschen
11:  ADD r6, r4    ; Dekrement innerer Schlaufenzaehler
12:  ADD ip, fl    ; Ueberlauf verarbeiten fuer Auswahl in Zeile 13/14
13:  SET ip, 14    ; Bei Schlaufenzaehler == 0 ende innere Schlaufe
14:  SET ip, 5     ; Bei Schlaufenzaehler > 0 weiter
15:  ADD r7, r5    ; Inkrement Quotient
16:  SET ip, 3     ; Innere Schlaufe neu beginnen
17:  MOV r8, r11   ; Divisor nach r8 kopieren
18:  SET ip, 19    ; An start von Rest-Subtraktionsloop springen
19:  ADD r8, r4    ; Kopie von Divisor dekrementieren
20:  SET fl, 0     ; Flag loeschen
21:  ADD r6, r4    ; Schlaufenzaehler dekrementieren
22:  ADD ip, fl    ; Ueberlaufcheck
23:  SET ip, 24    ; Schlaufe beenden
24:  SET ip, 18    ; Schlaufe wiederholen
25:  ADD r8, r4    ; r8 dekrementieren -> Divisonsrest
26:  HLT
```

In der Distribution finden sich weitere Programmbeispiele, welche die diversen Programmiertechniken der 4IA-Sprache erklären. Sämtlicher Code ist ausführlich dokumentiert.

4.3 Die XIA-Sprache

Die XIA-Sprache (Extended Instruction Assembler) baut auf der vorgestellten 4IA-Sprache auf und erweitert diese. Das Ziel der XIA-Sprache ist es, einerseits die Entwicklung von komplexeren Programmen zu vereinfachen und andererseits eine Sprache zur Verfügung zu stellen, in welche eine Hochsprache anhand eines Compilers übersetzt werden könnte (siehe Grafik auf Seite 68).

In Tabelle 1 ist ersichtlich, wie die Instruktionen der XIA-Sprache in vier Gruppen unterteilt werden.

Gruppe	*Name*	*Funktion*

Generelles	`SET`
	`MOV`
	`HLT`
	`SPACE`
		Grundoperationen der virtuellen Maschine und Generierung von XIA Kommentaren.

Mathematik	`ADD`
	`SUB`
	`MUL`
	`DIV`
	`MOD`
		Die vier grundlegenden mathematischen Operationen zusätzlich der „Modulo" Operation.

Bit-Logik	`AND`
	`OR`
	`XOR`
	`NOT`
	`SHL`
	`SHR`
		Die logischen Operationen, mit welchen sich mit geringem Aufwand sämtliche noch fehlenden bitweisen Operationen programmieren lassen.

Sprünge	`JMP`
	`JG`
	`JGE`
	`JEQ`
	`JLE`
	`JL`
	`JNE`
		Bedingte und unbedingte Sprünge, wobei bereits sämtliche Möglichkeiten implementiert sind. Es sollte kein Bedarf bestehen, Umwandlungen wie beispielsweise `JG r1 r2` → `JLE r2 r1` vorzunehmen.

Tabelle 1: Zusammenfassende Gruppierung der XIA-Instruktionen

4.3.1 Zeilenaufbau

Der grundlegende Aufbau einer Zeile XIA-Code entspricht folgendem Muster:

```
INSTRUKTION <Argumente> <Kommentar>
```

Die Anzahl der Argumente hängt direkt von der entsprechenden Instruktion ab und muss in jedem Fall eingehalten werden. Kommentare werden in den generierten 4IA-Code nicht übernommen, sondern vom Scanner gelöscht. Da der `xto4` Cross-Compiler den Code automatisch mit entsprechenden Kommentaren versieht. Die Instruktion `SPACE` (s. Seite 87) kann benutzt werden, um den generierten Code mit eigenen Kommentaren zu versehen.

4.3.2 Labels und Sprünge

XIA benötigt im Gegensatz zu 4IA keine absoluten Zeilennummern zur Realisierung von Sprüngen, sondern arbeitet mit positionsunabgängigen Labels. Diese werden mittels der Instruktion `LABEL` gesetzt.

Beispiel:

```
SET r1 10   ; Ordne Register r1 den Wert 10 zu
LABEL L1    ; Setzen eines Sprungziels
SUB r1 1    ; Dekrementieren von r1
JGE r2 1 L1 ; Springe zu Label L1
            ; falls r2 groesser gleich 1
HLT         ; Programm beenden
```

4.3.3 Spezielle Register

Die Tabelle 2 zeigt diejenigen Register, denen in der die XIA-Sprache eine besondere Bedeutung zukommt.

Identifier	*Beschreibung*
`ip`	Der Instruktionszeiger der virtuellen Maschine
`fl`	Das Carry-Flag, welches direkt von der virtuellen Maschine verwaltet wird.
`ec`	Error-Code nach Programm-Terminierung. Für Details, siehe Tabelle 4.
`cf`	Carry-Flag XIA, welches je nach ausgeführter Operation einen entsprechenden Wert gesetzt hat. Erläutert in den Beschreibungen der jeweiligen Instruktionen.

Tabelle 2: Verfügbare spezielle Register und deren Funktion

Durch die Kompilation von XIA zu 4IA besteht ein Bedarf an temporären, für compilerinterne Berechnungen benutzte Register. Aus diesem Grund stehen in der XIA-Sprache die Register `r0` bis `r14` *nicht* zur Verfügung.

Benutzbare Register der XIA-Sprache: `r15` bis `r`n

4.3.4 Adressierungsarten

Register können grundsätzlich direkt sowie indirekt adressiert werden. Bei einigen Instruktionen sind zusätzlich Konstanten erlaubt. Ausnahmen – beziehungsweise alle erlaubten Zugriffsarten – sind pro Instruktion der Tabelle 3 zu entnehmen. Indirekte Adressierungsarten sind für alle speziellen Register vollumfänglich erlaubt.

Beispiel:
Folgender XIA-Code zeigt, wie indirekte Adressierung angewendet wird. Dabei wird der Inhalt desjenigen Registers, auf welches `r1` zeigt, in das Register, auf welches `r2` zeigt, kopiert:

```
SET r1 4     ; Ordne Register r1 den Wert 4 zu
SET r2 3     ; Ordne Register r2 den Wert 3 zu
SET r4 10    ; Ordne Register r4 den Wert 10 zu
MOV *r2 *r1  ; Kopiere Inhalt von *r1 in *r2
HLT          ; Programm beenden
```

Obiges Programm hat die gleiche Bedeutung wie folgendes Codefragment:

```
SET r4 10    ; Ordne Register r4 den Wert 10 zu
MOV r3 r4    ; Kopiere Inhalt von r4 in r3
HLT          ; Programm beenden
```

Zur Verdeutlichung: Register `*r1` zeigt auf Register `r4`, Register `*r2` zeigt auf Register `r3`.

		Argument 1			*Argument 2*		
			Register			Register	
		Konstante	direkt	indirekt	Konstante	direkt	indirekt
Name	SET		x	x	x		
	MOV		x	x		x	x
	ADD		x	x	x	x	x
	SUB		x	x	x	x	x
	MUL		x	x	x	x	x
	DIV		x	x	x	x	x
	MOD		x	x	x	x	x
	AND		x	x	x	x	x
	OR		x	x	x	x	x
	XOR		x	x	x	x	x
	NOT		x	x	x	x	x
	SHL		x	x	x	x	x
	SHR		x	x	x	x	x
	JGE	x	x	x	x	x	x
	JEQ	x	x	x	x	x	x
	JLE	x	x	x	x	x	x
	JL	x	x	x	x	x	x
	JNE	x	x	x	x	x	x

Tabelle 3: XIA-Instruktionen und deren Adressierungsarten

4.3.5 Error-Codes

Tabelle 4 erklärt die möglichen Error-Codes, welche nach Programmende im entsprechenden Register gesetzt sein können. Dieses Register kann in der `pshell`-Umgebung durch den Befehl `showreg` ausgelesen werden (s. Abschnitt 4.3.3). Beim Start eines Programmes wird der Error-Code immer auf `0` initialisiert.

Error-Code	*Bedeutung*
`0`	Programm ohne Fehler terminiert
`1`	Division durch 0 führte zu Programmabbruch

Tabelle 4: Mögliche Error-Codes und deren Bedeutung

4.3.6 Architekturdefinition

Als optionale Architekturdefinition kann zu Beginn eines XIA-Programmes folgende Instruktion stehen:

`ARCH` [registerbreite] < höchstes benutztes Register >

Das erste obligatorische Argument wird unverändert dem generierten 4IA-Code weitergegeben und so der virtuellen Maschine mitgeteilt. Zulässig sind Registerbreiten von 2 bis 16. Das zweite optionale Argument kann verwendet werden, um der virtuellen Maschine mitzuteilen, wieviele Register alloziiert werden sollen. Dies ist nützlich, falls indirekt adressierte Register verwendet werden, welche vom Cross-Compiler nicht eruiert werden können. Falls das zweite optionale Argument nicht gesetzt wurde, bestimmt der Compiler selbst die Anzahl der verwendeten Register, indem er mittels Programmcode berechnet, welches das höchste verwendete Register ist.

Wichtig: Falls diese optionale Definition *nicht* gesetzt ist, generiert der `xto4` Cross-Compiler automatisch eine Architekturdefinition mit einer Registerbreit von 8 Bit.

4.3.7 EBNF

Die syntaktische Definition der XIA-Sprache in EBNF-Notation[4], wobei `xia` das Startsymbol darstellt:

```
xia           ::= (arch)? (line)* ;
arch          ::= "ARCH" dec (dec)?;
const         ::= dual | dec | hex ;
alpha         ::= [a-z, A-Z] ;
numeric       ::= [0-9] ;
concat        ::= [_, -];
comment       ::= ";" ([#@+*/^?!><.,;:()[]{}' '] |
                  alpha | numeric)* ;
label         ::= "L" (alpha | numeric)+ ;
dual          ::= (0b[0,1]+) ;
dec           ::= ((0d)?[0-9]+) ;
hex           ::= (0x[0-9, a-f, A-F]+) ;
line          ::= instruction (comment)?
reg           ::= dirreg | indirreg | specialreg;
dirreg        ::= "r"  (numeric)+ ;
indirreg      ::= "*r" (numeric)+ ;
specialreg    ::= "ip" | "fl" | "ec" | "cf" ;
instruction   ::= set | mov | hlt | space |
                  add | sub | mul | div | mod |
```

[4] Extended Backus-Naur Form

```
                and | or | xor | not | shl | shr |
                jmp | jg | jge | jeq | jle |
                jl | jne ;
set           ::= "SET" reg (,)? const ;
mov           ::= "MOV" reg (,)? reg ;
hlt           ::= "HLT" ;
space         ::= "SPACE" ;
add           ::= "ADD" reg (const | reg) ;
sub           ::= "SUB" reg (const | reg) ;
mul           ::= "MUL" reg (const | reg) ;
div           ::= "DIV" reg (const | reg) ;
mod           ::= "MOD" reg (const | reg) ;
and           ::= "AND" reg (const | reg) ;
or            ::= "OR"  reg (const | reg) ;
xor           ::= "XOR" reg (const | reg) ;
not           ::= "NOT" reg (const | reg) ;
shl           ::= "SHL" reg (const | reg) ;
shr           ::= "SHR" reg (const | reg) ;
jmp           ::= "JMP" (const | reg)
                (const | reg) label ;
jg            ::= "JG"  (const | reg)
                (const | reg) label ;
jge           ::= "JGE" (const | reg)
                (const | reg) label ;
jeq           ::= "JEQ" (const | reg)
                (const | reg) label ;
jle           ::= "JLE" (const | reg)
                (const | reg) label ;
jl            ::= "JL"  (const | reg)
                (const | reg) label ;
jne           ::= "JNE" (const | reg)
                (const | reg) label ;
```

4.3.8 Instruktionsreferenz: SET

Beschrieb

Die Set Instruktion weist dem Register von *arg1* den Wert von *arg2* zu. Beim Kompilieren wird diese Funktion unverändert weitergegeben.

Syntax

SET [*arg1*] [*arg2*]

Argumente

arg1	Das Argument, welchem der Wert von *arg2* zugewiesen wird. Dieses Argument kann ein direktes Register oder ein Pointer sein, jedoch keine Konstante.
arg2	Eine Konstante, welche ins Argument *arg1* geladen werden soll. Dieses Argument muss vom Typ Konstante sein. Gültige Zahlensysteme sind das Dual-, das Dezimal- sowie das Hexadezimalsystem.

Rückgabewert

arg1	Dem Register wird der Wert von *arg2* zugeordnet.
arg2	Unverändert.

Flag

Alle Flags bleiben unverändert.

Beispiele

Dem Register `r1` den Wert 4 zuweisen:

```
SET r1  4       ; Ordne Register r1 den Wert 4 zu
```

Dem Register `*r1` den Wert 5 zuweisen:

```
SET r1  4       ; Ordne Register r1 den Wert 4 zu
SET *r1 5       ; Ordne Register *r1(=r4) den Wert 5 zu
```

4.3.9 Instruktionsreferenz: MOV

Beschrieb

Kopiert den Wert eines Registers in ein anderes.

Syntax

`MOV` [*arg1*] [*arg2*]

Argumente

arg1	Das Argument, in welches der Wert von *arg2* kopiert werden soll. Dieses Argument kann ein direktes Register oder ein Pointer sein, jedoch keine Konstante.
arg1	Das Argument, dessen Wert in *arg1* kopiert werden soll. Dieses Argument kann ein direktes Register oder ein Pointer sein, jedoch keine Konstante.

Rückgabewert

arg1	Dem Argument wird der Wert von *arg2* zugeordnet
arg2	Unverändert.

Flag

Alle Flags bleiben unverändert.

Beispiele

Kopieren des Wertes von Register `r2` in `r1`. Beide Register sind direkt adressiert. Nach Ausführung enthalten sowohl Register `r1` wie auch Register `r2` den Wert 4.

```
SET r2  4       ; Dem Register den Wert 4 zuordnen
MOV r1  r2      ; Registerinhalt von r2 nach r1 kopieren
```

Kopieren des Wertes von Pointer `r2` in Register `r1`. Danach steht in Register `r1` der Wert, welcher in `*r2` beziehungsweise in `r3` gespeichert ist.

```
SET r2  3       ; Dem Register den Wert 3 zuordnen
SET r3  4       ; Dem Register den Wert 4 zuordnen
MOV r1  *r2     ; Registerinhalt von *r2 (=r3) nach r1 kopieren.
```

4.3.10 Instruktionsreferenz: HLT

Beschrieb

Veranlasst die virtuelle Maschine zum Stopp. Sämtliche Resultate sowie die gegebenenfalls gesetzten Flags, insbesondere das Error-Flag, können danach ausgewertet werden.

Syntax

`HLT`

Argumente

Keine

Rückgabewert

Unverändert

Flag

Alle Flags bleiben unverändert.

Beispiele

Nach der Addition $r1 = 4 + 5$ wird die virtuelle Maschine mit `HLT` gestoppt. Nach Auflistung der Register mittels `pshell` kann festgestellt werden, dass das Register `r1` nun den Wert 9, das Resultat der Addition, enthält.

```
SET r1  4       ; Dem Register den Wert 4 zuordnen
ADD r1  5       ; Register r1 addieren mit 5
HLT
```

4.3.11 Instruktionsreferenz: SPACE

Beschrieb

Fügt eine Leerzeile in generierten 4IA-Code ein. Optional kann die Zeile auch mit Kommentaren versehen werden, falls ein mentsprechendes Argument mitgegeben wird. Auf diese Weise ist es möglich, selbst automatisch generierten Code zu kommentieren.

Syntax

`SPACE <aI>`

Argumente

arg1 Das optionale Argument kann Text enthalten, welcher im generierten 4IA-Code wieder ersichtlich wird. Folgende Zeichen sind erlaubt: Alle Klein- und Grossbuchstaben, alle Zahlen sowie einige Sonderzeichen, welche der EBNF (siehe Seite 82) zu entnehmen sind.

Rückgabewert

arg1 Unverändert

Flag

Alle Flags bleiben unverändert.

Beispiele

Übersichtlicher 4IA-Code, welcher zwei Divisionen durchführt.

```
SET r1  32       ; Dem Register den Wert 32 zuordnen
SET r2  4        ; Dem Register den Wert 4 zuordnen
SPACE   Nachfolgend die erste Division: r1 / r2
DIV r1  r2       ; Die erste Division
SPACE            ; Zwei Leerzeilen
SPACE            ; ...
SPACE   Nun die zweite Division
DIV r1  r2       ; Die zweite Division
SPACE   Hier sind beide Divisionen beendet.
```

4.3.12 Instruktionsreferenz: ADD

Beschrieb

Addiert zwei Zahlen miteinander. Falls ein Überlauf stattfindet, wird dies im Carry-Flag `cf` ersichtlich.

Syntax

`ADD` [*arg1*] [*arg2*]

Argumente

arg1	Erster Summand: Dieses Argument kann ein direktes Register oder ein Pointer sein, jedoch keine Konstante.
arg1	Zweiter Summand: Dieses Argument darf sowohl vom Typ Konstante sein, wie auch ein direkt adressiertes Register oder ein Pointer. Gültige Zahlensysteme der Konstanten sind das Dual-, das Dezimal- sowie das Hexadezimalsystem.

Rückgabewert

arg1	Die Summe der Addition wird in Argument *arg1* geschrieben.
arg2	Unverändert.

Flag

`cf`	Das Carry-Flag Register wird 1 gesetzt, falls die Instruktion einen Überlauf verursachte. Andernfalls wird das `cf` nicht verändert.

Beispiele

Eine Addition $r1 = 3 + 4$. Das Resultat steht danach in `r1`.

```
SET r1  3       ; Dem Register den Wert 3 zuordnen
ADD r1  4       ; Addition von r1 mit 4.
```

4.3.13 Instruktionsreferenz: SUB

Beschrieb

Subtraktion (Differenz) zweier Zahlen. Falls ein Unterlauf stattfindet, wird dies im Carry-Flag `cf` ersichtlich.

Syntax

SUB [*arg1*] [*arg2*]

Argumente

arg1	Der Minuend: Dieses Argument kann ein direktes Register oder ein Pointer sein, jedoch keine Konstante.
arg2	Der Subtrahend: Dieses Argument darf sowohl vom Typ Konstante sein, wie auch ein direkt adressiertes Register oder ein Pointer. Gültige Zahlensysteme der Konstanten sind das Dual-, das Dezimal- sowie das Hexadezimalsystem.

Rückgabewert

arg1	In diesem Argument wird die Differenz, also das Resultat, gespeichert.
arg2	Unverändert.

Flag

`cf`	Das Carry Flag Register wird 1 gesetzt, falls die Instruktion einen Unterlauf verursachte. Andernfalls wird das `cf` nicht verändert.

Beispiele

Eine Subtraktion $r1 = 9 - 5$. Das Resultat steht danach in `r1`.

```
SET r1  9       ; Dem Register den Wert 9 zuordnen
SUB r1  5       ; Subtraktion von r1 mit 5
```

4.3.14 Instruktionsreferenz: MUL

Beschrieb

Multiplikation (Produkt) zweier Zahlen. Falls ein Überlauf stattfindet, wird dies im Carry-Flag `cf` ersichtlich.

Syntax

`MUL` [*arg1*] [*arg2*]

Argumente

arg1	Erster Faktor der Multiplikation: Dieses Argument kann ein direktes Register oder ein Pointer sein, jedoch keine Konstante.
arg1	Zweiter Faktor der Multiplikation: Dieses Argument darf sowohl vom Typ Konstante sein, wie auch ein direkt adressiertes Register oder ein Pointer. Gültige Zahlensysteme der Konstanten sind das Dual-, das Dezimal- sowie das Hexadezimalsystem.

Rückgabewert

arg1	In diesem Argument wird das Produkt, also das Resultat, gespeichert.
arg2	Unverändert.

Flag

`cf`	Das Carry-Flag Register wird 1 gesetzt, falls die Instruktion einen Überlauf verursachte. Andernfalls wird das `cf` nicht verändert.

Beispiele

Multiplikation von $r1 = 4 * 5$. Das Resultat steht danach in `r1`.

```
SET r1  4       ; Dem Register den Wert 4 zuordnen
MUL r1  5       ; Multiplikation von r1 mit 5
```

4.3.15 Instruktionsreferenz: DIV

Beschrieb

Division (Quotient) zweier Zahlen. Bei Divisionen mit `0` wird das Error Code Register `1` gesetzt und die virtuelle Maschine gestoppt. Der Divisionsrest, auch Übertrag genannt, wird im `cf` Register gespeichert.

Syntax

`DIV` [*arg1*] [*arg2*]

Argumente

arg1	Der Dividend: Dieses Argument kann ein direktes Register oder ein Pointer sein, jedoch keine Konstante.
arg1	Der Divisor: Dieses Argument darf sowohl vom Typ Konstante sein, wie auch ein direkt adressiertes Register oder ein Pointer. Gültige Zahlensysteme der Konstanten sind das Dual-, das Dezimal- sowie das Hexadezimalsystem.

Rückgabewert

arg1	In diesem Argument wird der Quotient, also das Resultat, gespeichert.
arg2	Unverändert.

Flag

`cf`	Der Übertrag der Division wird in diesem Register gespeichert
`ec`	Bei einer versuchten Division mit `0` wird diesem Register der Error Code Wert 1 zugeordnet und das Programm beendet.

Beispiele

Division von $r1 = 32/10$. Das Resultat `3` ist danach in `r1`, der Übertrag von `2` im Register `cf` gespeichert.

```
SET r1  32      ; Dem Register den Wert 32 zuordnen
DIV r1  10      ; Division von r1 mit 10
```

4.3.16 Instruktionsreferenz: MOD

Beschrieb

Errechnet den Restbetrag, welcher übrig bleibt bei einer Division zweier Zahlen. Bei Divisionen mit `0` wird das Error Code Register `1` gesetzt und die virtuelle Maschine gestoppt.

Syntax

`MOD` [*arg1*] [*arg2*]

Argumente

arg1	Der Dividend: Dieses Argument kann ein direktes Register oder ein Pointer sein, jedoch keine Konstante.
arg1	Der Divisor: Dieses Argument darf sowohl vom Typ Konstante sein, wie auch ein direkt adressiertes Register oder ein Pointer. Gültige Zahlensysteme der Konstanten sind das Dual-, das Dezimal- sowie das Hexadezimalsystem.

Rückgabewert

arg1	In diesem Argument wird der Restwert, also der Übertrag, gespeichert.
arg2	Unverändert.

Flag

`cf`	Der Übertrag der Division wird in diesem Register gespeichert
`ec`	Bei einer versuchten Division mit `0` wird diesem Register der Error Code Wert 1 zugeordnet und das Programm beendet.

Beispiele

Modulo von $r1 = 32 : 10$. Das Resultat `3` wird verworfen und der Übertrag von `2`, welcher bereits im Register `cf` gespeichert ist, kopiert nach *arg1* respektive im Beispiel nach `r1`.

```
SET r1  32      ; Dem Register den Wert 32 zuordnen
MOD r1  10      ; Modulo Operation von r1 mit 10
```

4.3.17 Instruktionsreferenz: AND

Beschrieb

Logisches AND, welches für jedes Bit der Argumente die bitweise AND Operation durchführt. Falls beide höchstwertigen Bit gesetzt sind, geht der Übertrag verloren.

Syntax

`AND` [*arg1*] [*arg2*]

Argumente

arg1	Die erste Zahl: Dieses Argument kann ein direktes Register oder ein Pointer sein, jedoch keine Konstante.
arg1	Die zweite Zahl: Dieses Argument darf sowohl vom Typ Konstante sein, wie auch ein direkt adressiertes Register oder ein Pointer. Gültige Zahlensysteme der Konstanten sind das Dual-, das Dezimal- sowie das Hexadezimalsystem.

Rückgabewert

arg1	Das Resultat der bitweisen AND Operationen.
arg2	Unverändert.

Flag

Alle Flags bleiben unverändert.

Beispiele

Bitweise AND Operation mit folgenden zwei Zahlen: `0b0101` und `0b0110`. Das Resultat in `r1` ist `0b0100`.

```
SET r1  0b0101  ; Dem Register den binaeren Wert 0b0101 zuordnen
AND r1  0b0110  ; Bitweise AND Operation mit r1 und 0b0110
```

4.3.18 Instruktionsreferenz: OR

Beschrieb

Logisches OR, welches für jedes Bit der Argumente die bitweise OR Operation durchführt.

Syntax

`OR` [*arg1*] [*arg2*]

Argumente

arg1	Die erste Zahl: Dieses Argument kann ein direktes Register oder ein Pointer sein, jedoch keine Konstante.
arg1	Die zweite Zahl: Dieses Argument darf sowohl vom Typ Konstante sein, wie auch ein direkt adressiertes Register oder ein Pointer. Gültige Zahlensysteme der Konstanten sind das Dual-, das Dezimal- sowie das Hexadezimalsystem.

Rückgabewert

arg1	Das Resultat der bitweisen OR Operationen.
arg2	Unverändert.

Flag

Alle Flags bleiben unverändert.

Beispiele

Bitweise OR Operation mit folgenden zwei Zahlen: `0b0101` und `0b0110`. Das Resultat in `r1` ist `0b0111`.

```
SET r1  0b0101  ; Dem Register den binaeren Wert 0b0101 zuordnen
OR  r1  0b0110  ; Bitweise OR Operation mit r1 und 0b0110
```

4.3.19 Instruktionsreferenz: XOR

Beschrieb

Logisches XOR, welches für jedes Bit der Argumente die bitweise XOR Operation durchführt.

Syntax

`XOR` [*arg1*] [*arg2*]

Argumente

arg1	Die erste Zahl: Dieses Argument kann ein direktes Register oder ein Pointer sein, jedoch keine Konstante.
arg1	Die zweite Zahl: Dieses Argument darf sowohl vom Typ Konstante sein, wie auch ein direkt adressiertes Register oder ein Pointer. Gültige Zahlensysteme der Konstanten sind das Dual-, das Dezimal- sowie das Hexadezimalsystem.

Rückgabewert

arg1	Das Resultat der bitweisen XOR Operationen.
arg2	Unverändert.

Flag

Alle Flags bleiben unverändert.

Beispiele

Bitweise XOR Operation mit folgenden zwei Zahlen: `0b0101` und `0b0110`. Das Resultat in `r1` ist `0b0011`.

```
SET r1  0b0101  ; Dem Register den binaeren Wert 0b0101 zuordnen
XOR r1  0b0110  ; Bitweise OR Operation mit r1 und 0b0110
```

4.3.20 Instruktionsreferenz: NOT

Beschrieb

Logisches NOT, welches für jedes Bit des Arguments die bitweise NOT Operation durchführt.

Syntax

`NOT` [*arg1*]

Argumente

arg1	Die Zahl, welche negiert werden soll: Dieses Argument kann ein direktes Register oder ein Pointer sein, jedoch keine Konstante.

Rückgabewert

arg1	Das Resultat der bitweisen NOT Operationen.

Flag

Alle Flags bleiben unverändert.

Beispiele

Bitweise NOT Operation mit folgender Zahl: `0b0101`. Das Resultat in `r1` ist `0b1010`.

```
SET r1  0b0101  ; Dem Register den binaeren Wert 0b0101 zuordnen
NOT r1          ; Bitweise NOT Operation mit r1
```

4.3.21 Instruktionsreferenz: SHL

Beschrieb

Logisches SHL (Shift-Left), welches den Wert von *arg1* um *arg2* Bit nach links schiebt. Die Anzahl *arg2* höchstwertigsten Bits von *arg1* gehen gegebenenfalls verloren und das `cf` wird gesetzt.

Syntax

`SHL` [*arg1*] [*arg2*]

Argumente

arg1	Die Zahl, welche um *arg2* bit nach links geschoben werden soll. Dieses Argument kann ein direktes Register oder ein Pointer sein, jedoch keine Konstante.
arg2	Die Anzahl Verschiebungen nach links. Dieses Argument darf sowohl vom Typ Konstante sein, wie auch ein direkt adressiertes Register oder ein Pointer. Gültige Zahlensysteme der Konstanten sind das Dual-, das Dezimal- sowie das Hexadezimalsystem.

Rückgabewert

arg1	Die Zahl, um *arg2* Bits nach links geschoben.
arg2	Unverändert

Flag

`cf`	Das Carry-Flag Register wird 1 gesetzt, falls die Instruktion einen Überlauf verursachte. Andernfalls wird das `cf` nicht verändert.

Beispiele

Shift left Operation mit der Zahl 4 um 2 bit. Das Resultat in `r1` ist $4*2*2 = 16$.

```
SET r1  4       ; Dem Register den Wert 4 zuordnen
SHL r1  2       ; shift left von r1 um 2 Bit
```

4.3.22 Instruktionsreferenz: SHR

Beschrieb

Logisches SHR (shift right), welches den Wert von *arg1* um *arg2* Bit nach rechts schiebt. Die Anzahl *arg2* niederwertigster Bit von *arg1* gehen gegebenenfalls verloren und das `cf` wird gesetzt.

Syntax

`SHL` [*arg1*] [*arg2*]

Argumente

arg1	Die Zahl, welche um *arg2* Bit nach rechts geschoben werden soll. Dieses Argument kann ein direktes Register oder ein Pointer sein, jedoch keine Konstante.
arg2	Die Anzahl Verschiebungen nach rechts. Dieses Argument darf sowohl vom Typ Konstante sein, wie auch ein direkt adressiertes Register oder ein Pointer. Gültige Zahlensysteme der Konstanten sind das Dual-, das Dezimal- sowie das Hexadezimalsystem.

Rückgabewert

arg1	Die Zahl, um *arg2* Bits nach rechts geschoben.
arg2	Unverändert

Flag

`cf`	Das Carry Flag Register wird 1 gesetzt, falls die Instruktion einen Unterlauf verursachte. Andernfalls wird das `cf` nicht verändert.

Beispiele

Shift right Operation mit der Zahl 4 um 2 bit. Das Resultat in `r1` ist $16/2/2 = 4$.

```
SET r1  16      ; Dem Register den Wert 16 zuordnen
SHL r1  2       ; shift right von r1 um 2 Bit
```

4.3.23 Instruktionsreferenz: JMP

Beschrieb

Mit `JMP` (Jump) wird ein unbedingter Sprung erzwungen zum Label *arg1* .
Falls kein entsprechendes Label definiert ist, wird das Argument durch den Compiler nicht aufgelöst und direkt der Name als Sprungadresse beibehalten, was zur Laufzeit `pshell`-seitig einen Fehler verursacht.

Syntax

`JMP` [*arg1*]

Argumente

arg1	Das Label, zu welchem gesprungen werden soll: Dieses Argument muss zwingend mit dem Zeichen `L` beginnen, gefolgt von grundsätzlich beliebig vielen Sonderzeichen (siehe Seite 82).

Rückgabewert

arg1	Unverändert

Flag

Alle Flags bleiben unverändert.

Beispiele

Folgendes Beispiel führt solange Divisionen aus, bis eine Division mit `0` ausgeführt wird, und bricht deshalb das Programm ab.

```
SET r1  256       ; Dem Register den Wert 256 zuordnen
LABEL   L1        ; Setzen der Sprungadresse L1
DIV r1  2         ; Division von r1 mit 2 bis DivByZero
JMP     L1        ; Unbedingter Sprung zu Label L1
```

4.3.24 Instruktionsreferenz: JG

Beschrieb

Mit `JG` (Jump Greater) wird ein bedingter Sprung gemacht zum Label *arg3* , falls der Wert des Arguments *arg1* grösser ist als der von Argument *arg2* .
Falls kein entsprechendes Label definiert ist, wird das Argument durch den Compiler nicht aufgelöst und direkt der Name als Sprungadresse beibehalten, was zur Laufzeit `pshell`-seitig einen Fehler verursacht.

Syntax

`JG` [*arg1*] [*arg2*] [*arg3*]

Argumente

arg1	Erster Wert, welcher mit *arg2* verglichen wird: Dieses Argument darf sowohl vom Typ Konstante sein, wie auch ein direkt adressiertes Register oder ein Pointer. Gültige Zahlensysteme der Konstanten sind das Dual-, das Dezimal- sowie das Hexadezimalsystem.
arg2	Zweiter Wert, welcher mit *arg1* verglichen wird: Dieses Argument darf sowohl vom Typ Konstante sein, wie auch ein direkt adressiertes Register oder ein Pointer. Gültige Zahlensysteme der Konstanten sind das Dual-, das Dezimal- sowie das Hexadezimalsystem.
arg3	Das Label, zu welchem gesprungen werden soll: Dieses Argument muss zwingend mit dem Zeichen `L` beginnen, gefolgt von grundsätzlich beliebig vielen Sonderzeichen (siehe Seite 82).

Rückgabewert

arg1	Unverändert
arg2	Unverändert
arg3	Unverändert

Flag

Alle Flags bleiben unverändert.

Beispiele

Folgendes Beispiel zählt das Register `r1` von `10` hinunter bis `5` und verlässt dann die Zählschlaufe.

```
SET r1  10        ; Dem Register den Wert 10 zuordnen
LABEL   L1        ; Setzen der Sprungadresse L1
SUB r1  1         ; Dekrementieren von Schlaufenvariable
JG  r1  5   L1    ; Solange wiederholen bis r1 == 5 ist.
```

4.3.25 Instruktionsreferenz: JGE

Beschrieb

Mit `JGE` (Jump Greater Equals) wird ein bedingter Sprung gemacht zu Label *arg3* , falls der Wert des Arguments *arg1* grösser oder gleich ist wie der von Argument *arg2* .
Falls kein entsprechendes Label definiert ist, wird das Argument durch den Compiler nicht aufgelöst und direkt der Name als Sprungadresse beibehalten, was zur Laufzeit `pshell`-seitig einen Fehler verursacht.

Syntax

`JGE` [*arg1*] [*arg2*] [*arg3*]

Argumente

arg1	Erster Wert, welcher mit *arg2* verglichen wird: Dieses Argument darf sowohl vom Typ Konstante sein, wie auch ein direkt adressiertes Register oder ein Pointer. Gültige Zahlensysteme der Konstanten sind das Dual-, das Dezimal- sowie das Hexadezimalsystem.
arg2	Zweiter Wert, welcher mit *arg1* verglichen wird: Dieses Argument darf sowohl vom Typ Konstante sein, wie auch ein direkt adressiertes Register oder ein Pointer. Gültige Zahlensysteme der Konstanten sind das Dual-, das Dezimal- sowie das Hexadezimalsystem.
arg3	Das Label, zu welchem gesprungen werden soll: Dieses Argument muss zwingend mit dem Zeichen `L` beginnen, gefolgt von grundsätzlich beliebig vielen Sonderzeichen (siehe Seite 82).

Rückgabewert

arg1	Unverändert
arg2	Unverändert
arg3	Unverändert

Flag

Alle Flags bleiben unverändert.

Beispiele

Folgendes Beispiel zählt das Register `r1` von `10` hinunter bis `5` und verlässt dann die Zählschlaufe.

```
SET r1  10        ; Dem Register den Wert 10 zuordnen
LABEL   L1        ; Setzen der Sprungadresse L1
SUB r1  1         ; Dekrementieren von Schlaufenvariable
JGE r1  6   L1    ; Solange wiederholen bis r1 == 5 ist.
```

4.3.26 Instruktionsreferenz: JEQ

Beschrieb

Mit `JEQ` (Jump Equals) wird ein bedingter Sprung gemacht zu Label *arg3* , falls der Wert des Arguments *arg1* gleich ist wie der von Argument *arg2* .
Falls kein entsprechendes Label definiert ist, wird das Argument durch den Compiler nicht aufgelöst und direkt der Name als Sprungadresse beibehalten, was zur Laufzeit `pshell`-seitig einen Fehler verursacht.

Syntax

`JEQ` [*arg1*] [*arg2*] [*arg3*]

Argumente

arg1	Erster Wert, welcher mit *arg2* verglichen wird: Dieses Argument darf sowohl vom Typ Konstante sein, wie auch ein direkt adressiertes Register oder ein Pointer. Gültige Zahlensysteme der Konstanten sind das Dual-, das Dezimal- sowie das Hexadezimalsystem.
arg2	Zweiter Wert, welcher mit *arg1* verglichen wird: Dieses Argument darf sowohl vom Typ Konstante sein, wie auch ein direkt adressiertes Register oder ein Pointer. Gültige Zahlensysteme der Konstanten sind das Dual-, das Dezimal- sowie das Hexadezimalsystem.
arg3	Das Label, zu welchem gesprungen werden soll: Dieses Argument muss zwingend mit dem Zeichen `L` beginnen, gefolgt von grundsätzlich beliebig vielen Sonderzeichen (siehe Seite 82).

Rückgabewert

arg1	Unverändert
arg2	Unverändert
arg3	Unverändert

Flag

Alle Flags bleiben unverändert.

Beispiele

Folgendes Beispiel zählt das Register `r1` von `10` hinunter bis 5 und verlässt dann die Zählschlaufe.

```
SET r1  10        ; Dem Register den Wert 10 zuordnen
LABEL   L1        ; Setzen der Sprungadresse L1
SUB r1  1         ; Dekrementieren von Schlaufenvariable
JEQ r1  5   Lend  ; Solange wiederholen bis r1 == 5 ist.
JMP     L1        ; Immer zu L1 springen, Abbruch ist anderswo
LABEL   Lend      ; Hierher wird bei Schlaufenabbruch gejumped
```

4.3.27 Instruktionsreferenz: JLE

Beschrieb

Mit `JLE` (Jump Less Equals) wird ein bedingter Sprung gemacht zu Label *arg3* , falls der Wert des Arguments *arg1* kleiner oder gleich ist wie der von Argument *arg2* .
Falls kein entsprechendes Label definiert ist, wird das Argument durch den Compiler nicht aufgelöst und direkt der Name als Sprungadresse beibehalten, was zur Laufzeit `pshell`-seitig einen Fehler verursacht.

Syntax

`JLE` [*arg1*] [*arg2*] [*arg3*]

Argumente

arg1	Erster Wert, welcher mit *arg2* verglichen wird: Dieses Argument darf sowohl vom Typ Konstante sein, wie auch ein direkt adressiertes Register oder ein Pointer. Gültige Zahlensysteme der Konstanten sind das Dual-, das Dezimal- sowie das Hexadezimalsystem.
arg2	Zweiter Wert, welcher mit *arg1* verglichen wird: Dieses Argument darf sowohl vom Typ Konstante sein, wie auch ein direkt adressiertes Register oder ein Pointer. Gültige Zahlensysteme der Konstanten sind das Dual-, das Dezimal- sowie das Hexadezimalsystem.
arg3	Das Label, zu welchem gesprungen werden soll: Dieses Argument muss zwingend mit dem Zeichen `L` beginnen, gefolgt von grundsätzlich beliebig vielen Sonderzeichen (siehe Seite 82).

Rückgabewert

arg1	Unverändert
arg2	Unverändert
arg3	Unverändert

Flag

Alle Flags bleiben unverändert.

Beispiele

Folgendes Beispiel zählt das Register `r1` von `5` hinauf bis 10 und verlässt dann die Zählschlaufe.

```
SET r1  5         ; Dem Register den Wert 5 zuordnen
LABEL   L1        ; Setzen der Sprungadresse L1
ADD r1  1         ; inkrementieren von Schlaufenvariable
JLE r1  9   L1    ; Solange wiederholen bis r1 == 10 ist.
```

4.3.28 Instruktionsreferenz: JL

Beschrieb

Mit `JL` (Jump Less) wird ein bedingter Sprung gemacht zu Label *arg3* , falls der Wert des Arguments *arg1* kleiner ist als der von Argument *arg2* .
Falls kein entsprechendes Label definiert ist, wird das Argument durch den Compiler nicht aufgelöst und direkt der Name als Sprungadresse beibehalten, was zur Laufzeit `pshell`-seitig einen Fehler verursacht.

Syntax

`JL` [*arg1*] [*arg2*] [*arg3*]

Argumente

arg1	Erster Wert, welcher mit *arg2* verglichen wird: Dieses Argument darf sowohl vom Typ Konstante sein, wie auch ein direkt adressiertes Register oder ein Pointer. Gültige Zahlensysteme der Konstanten sind das Dual-, das Dezimal- sowie das Hexadezimalsystem.
arg2	Zweiter Wert, welcher mit *arg1* verglichen wird: Dieses Argument darf sowohl vom Typ Konstante sein, wie auch ein direkt adressiertes Register oder ein Pointer. Gültige Zahlensysteme der Konstanten sind das Dual-, das Dezimal- sowie das Hexadezimalsystem.
arg3	Das Label, zu welchem gesprungen werden soll: Dieses Argument muss zwingend mit dem Zeichen `L` beginnen, gefolgt von grundsätzlich beliebig vielen Sonderzeichen (siehe Seite 82).

Rückgabewert

arg1	Unverändert
arg2	Unverändert
arg3	Unverändert

Flag

Alle Flags bleiben unverändert.

Beispiele

Folgendes Beispiel zählt das Register `r1` von `5` hinauf bis 10 und verlässt dann die Zählschlaufe.

```
SET r1  5         ; Dem Register den Wert 5 zuordnen
LABEL   L1        ; Setzen der Sprungadresse L1
ADD r1  1         ; inkrementieren von Schlaufenvariable
JL  r1  10  L1    ; Solange wiederholen bis r1 == 10 ist.
```

4.3.29 Instruktionsreferenz: JNE

Beschrieb

Mit `JNE` (Jump Equals) wird ein bedingter Sprung gemacht zu Label *arg3* , falls der Wert des Arguments *arg1* nicht gleich dem von Argument *arg2* ist.
Falls kein entsprechendes Label definiert ist, wird das Argument durch den Compiler nicht aufgelöst und direkt der Name als Sprungadresse beibehalten, was zur Laufzeit `pshell`-seitig einen Fehler verursacht.

Syntax

`JNE` [*arg1*] [*arg2*] [*arg3*]

Argumente

arg1	Erster Wert, welcher mit *arg2* verglichen wird: Dieses Argument darf sowohl vom Typ Konstante sein, wie auch ein direkt adressiertes Register oder ein Pointer. Gültige Zahlensysteme der Konstanten sind das Dual-, das Dezimal- sowie das Hexadezimalsystem.
arg2	Zweiter Wert, welcher mit *arg1* verglichen wird: Dieses Argument darf sowohl vom Typ Konstante sein, wie auch ein direkt adressiertes Register oder ein Pointer. Gültige Zahlensysteme der Konstanten sind das Dual-, das Dezimal- sowie das Hexadezimalsystem.
arg3	Das Label, zu welchem gesprungen werden soll: Dieses Argument muss zwingend mit dem Zeichen `L` beginnen, gefolgt von grundsätzlich beliebig vielen Sonderzeichen (siehe Seite 82).

Rückgabewert

arg1	Unverändert
arg2	Unverändert
arg3	Unverändert

Flag

Alle Flags bleiben unverändert.

Beispiele

Folgendes Beispiel zählt das Register `r1` von `10` hinunter bis `5` und verlässt dann die Zählschlaufe.

```
SET r1  10        ; Dem Register den Wert 10 zuordnen
LABEL   L1        ; Setzen der Sprungadresse L1
SUB r1  1         ; Dekrementieren von Schlaufenvariable
JNE r1  5   L1    ; Solange wiederholen bis r1 == 5 ist.
```

4.3.30 Instruktionsreferenz: ARCH

Beschrieb

Mit der optionalen Instruktion ARCH (Architecture) kann der virtuellen Maschine mitgeteilt werden, welche Registerbreite sie zum interpretieren und ausführen des generierten 4IA-Codes benutzen soll. Diese Instruktion darf nur einmal und nur ganz zu Beginn (abgesehen von Kommentarlinien, siehe Seite 87) verwendet werden. Falls nicht gesetzt, wird eine standardmässige Registerbreite von 8 Bit angenommen.
Das optionale zweite Argument kann dazu benutzt werden, der VM mitzuteilen wieviele Register sie alloziieren soll. Diese Option ist sinnvoll, falls indirekt addressierte Register verwendet werden, welche der Compiler nicht eruieren kann und der virtuellen Maschine eine zu geringe Anzahl verwendeter Register mitteilt, was zu Laufzeitfehlern führen würde.

Syntax

ARCH [*arg1*] <*arg2* >

Argumente

arg1	Die Registerbreite in Bit, im Minimum 2. Falls die `pshell` im Simulationsmodus betrieben wird existiert grundsätzlich keine Grenze, was die Registerbreite nach oben betrifft. Bei parasitären Betriebsarten allerdings wird die Registerbreite wegen der Breite der Checksummen begrenzt auf maximal 16 Bit.
arg2	Optional, teilt der virtuellen Maschine mit, wieviele Register alloziiert werden sollen.

Rückgabewert

arg1	Unverändert.
arg2	Unverändert.

Flag

Alle Flags bleiben unverändert.

Beispiele

Setzen der Registerbreite der VM (virtuelle Maschine) auf 9 Bit.

```
ARCH  9         ; Benutze VM mit Registerbreite 9 Bit
```

Setzen der Registerbreite der VM (virtuelle Maschine) auf 9 Bit und verwenden von 100 Registern, welche alloziiert werden sollen.

```
ARCH  9 100     ; Benutze VM mit Registerbreite 9 Bit
                ; und alloziiere 100 Register
```

4.3.31 Programmierhinweise

Sparsame Registernutzung
Sparsamer Umgang mit Registern trägt zur Entlastung der virtuellen Maschine bei. Eine Möglichkeit, die Anzahl verwendeter Register klein zu halten, bietet sich, indem die speziellen Register in Codeabschnitten, welche diese nicht verwenden, ebenfalls wie gewöhnliche Register mitbenutzt werden.

Das `ec` Register beispielsweise wird nur von der Division benutzt, die restlichen Instruktionen benötigen es nicht. Das Flag-Register `cf` wird von diversen Instruktionen verwendet, nicht so aber von den Instruktionen `SET` und `MOV`. Das `cf` eignet sich demnach bestens zum Zwischenspeichern temporärer Werte, welche innerhalb von `SET` und `MOV` Instruktionen gehalten werden müssen.

Effiziente Dekrementierung
Das Dekrementieren um eins entspricht einer Addition von $2^{Registerbreite} - 1$, bedarf jedoch mehr an zusätzlichem Aufwand. Folgende zwei Programmausschnitte bewirken dasselbe, wobei aber die zweite Variante wesentlich effizienter ist:

```
; Konventionelle Dekrementierung
ARCH  4       ; Benutze VM mit Registerbreite 4 Bit
SET   r1  10  ; Dem Register den Wert 10 zuordnen
SUB   r1  1   ; r1 dekrementieren
HLT           ; VM stoppen
```

```
; Effiziente Dekrementierung
ARCH  4       ; Benutze VM mit Registerbreite 4 Bit
SET   r1  10  ; Dem Register den Wert 10 zuordnen
ADD   r1  15  ; r1 dekrementieren. Die 15 errechnet
              ; sich wie folgt: 2^(Registerbreite)-1
HLT           ; VM stoppen
```

Carry-Flag Initialisierung
Ein häufiger Fehler, scheinbar fälschlich gesetztes Carry-Flag, kann verhindert werden, indem dieses vor der beeinflussenden Instruktion (wie beispielsweise `ADD`, `SUB`, `MUL` oder `DIV`) immer explizit initialisiert wird:

```
SET   cf  0 ; Carry-Flag mit 0 initialisieren
```

Das Carry-Flag wird, sofern kein Überlauf stattfindet, nicht auf 0 gesetzt, sondern behält seinen vorherigen Zustand. Das gleiche gilt für das Flag-Register der virtuellen Maschine `fl`.

4.3.32 Programmbeispiel: Bubblesort

Das nachfolgende Programm sortiert eine Anzahl von `r30` Ganzzahlen, beginnend bei `*r31` in aufsteigender Reihenfolge inplace nach dem Bubblesort-Algorithmus. Im gezeigen Beispiel also die fünf Integer in `r20` bis `r24`.

```
  SET r20, 3    ; Die zu sortierenden Zahlen werden in
  SET r21, 7    ; die Register r20 bis r24 gelegt
  SET r22, 5
  SET r23, 4
  SET r24, 0

  SET r30, 5    ; Anzahl der zu sortierenden Elemente
  SET r31, 20   ; Zeiger auf das erste Element

  SUB r30, 1    ; Initialisierung der Schlaufenzaehler
  MOV r34, r30
  SET r32, 0

LABEL Lot       ; Aeussere Schlaufe
  SET r33, 0

LABEL Lin       ; Innere Schlaufe

  MOV r35, r31
  ADD r35, r33  ; *r35 zeigt auf das aktuelle Element
  MOV r36, r35
  ADD r36, 1    ; *r36 zeigt auf das naechste Element

  ; Elemente vergleichen, falls
  ; Reihenfolge korrekt zu Lnswp springen:
  JLE *r35, *r36, Lnswp

  MOV r37, *r35  ; Elemente *r35 und *36 austauschen
  MOV *r35, *r36
  MOV *r36, r37

LABEL Lnswp
  ADD r33, 1

  ; Innere Schlaufe Abbruchsbedingung:
  JL  r33, r34, Lin

  SUB r34, 1
  ADD r32, 1

  ; Aeussere Schlaufe Abbruchsbedingung:
  JL  r32, r30, Lot

  HLT
```

PARASITIC COMPUTING
System Design

Version 1.0

Luzian Scherrer
Jürg Reusser

3. Januar 2003

Zusammenfassung

Dieses Dokument beschreibt das Design und den Aufbau der im Rahmen der Diplomarbeit „Parasitic Computing" realisierten Software `pshell` und `xto4`. Es werden die inneren Abläufe sowie die Interaktionen der einzelnen Softwarekomponenten aufgezeigt und alle zentralen Algorithmen im Detail dargestellt.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Übersicht

Das implementierte System, welches ermöglicht, jedes klassische Problem der Informatik (siehe [RS02c]) durch parasitäre Nutzung verteilter Ressourcen zu lösen, ist unterteilt in zwei voneinander unabhängige, installier- und benutzbare Pakete, die im Folgenden hinsichtlich ihrer Zuständigkeiten prägnant erläutert werden:

pshell[1]
Das Paket `pshell` ist zuständig für die Ausführung von in der 4IA-Sprache geschriebenem Quellcode innerhalb der virtuellen Maschine und dient als generelle Benutzerschnittstelle zu dieser. Die Software ist ausschliesslich in C geschrieben.

xto4[2]
Das Paket `xto4` besteht aus einem Cross-Compiler, welcher in der XIA-Sprache geschriebenen Quellcode in die 4IA-Sprache übersetzt. Die Software ist ausschliesslich in der Programmiersprache Java geschrieben und kann auf jeder Java-fähigen Plattform betrieben werden.

1.1 Schnittstelle `xto4` ↔ `pshell`

Die Schnittstelle zwischen dem Cross-Compiler `xto4` und der `pshell` definiert sich durch den vom `xto4` generierten Programm-Code, welcher die `pshell` interpretiert. Dieser sogenannte 4-Instruction-Assembler (kurz 4IA) besteht im Wesentlichen aus folgenden vier Instruktionen: `SET`, `MOV`, `ADD` und `HLT`. Die zusätzliche Instruktion `ARCH`, welche einmalig zu Beginn eines Programmes verwendet wird, bestimmt die Registerbreite, womit die virtuelle Maschine bei der Programmausführung arbeiten soll.

Im Weiteren gehören zur Schnittstelle zwischen dem `xto4` Cross-Compiler und der ausführenden `pshell` zwei spezielle Register. Dies ist zum einen das Carry-Flag (`cf`), mit welchem Register-Überläufe, verursacht durch die Addition von grossen Zahlen, angezeigt werden, zum anderen der Instruction-Pointer (`ip`), welcher die aktuelle Programmzeile des sich in Ausführung befindenden Codes enthält.

Zusammenfassend lässt sich die Schnittstelle zwischen `xto4` und `pshell` wie in Abbildung 1 aufgezeigt darstellen.

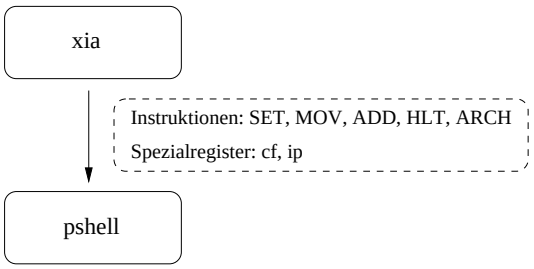

Abbildung 1: Schnittstelle zwischen den Paketen `xto4` und `pshell`

[1] `pshell` ist eine Abkürzung für „Parasitic Shell".
[2] Der Name `xto4` steht als Abkürzung für „XIA to 4IA Compiler".

1.2 Abhängigkeiten

Die beiden Pakete `xto4` und `pshell` können unabhängig voneinander kompiliert, installiert und betrieben werden. Sämtliche Abhängigkeiten ergeben sich somit ausschliesslich durch die im vorgängigen Kapitel 1.1 beschriebenen Schnittstellen.

2 Die `pshell`-Umgebung

2.1 Allgemeines

Das Paket `pshell` ist in die sieben in Tabelle 1 dargestellten, die Kernfunktionalitäten kapselnden Module unterteilt. Jedes dieser Module ist in der Sprache C geschrieben und wird bei der Übersetzung in ein Object kompiliert. Im letzten Schritt der Übersetzung werden dann alle Objects zu einer ausführbaren Binärdatei (mit Namen `pshell`) zusammengelinkt.

Modul	*Funktion*
pshell.c	Benutzerschnittstelle: Befehlseingabe und Verarbeitung, Register- und Statusausgabe. Dieses Modul enthält die `main()` Funktion.
scanner.l	Lexikalischer Scanner: Übersetzt einen 4IA-Quelltext in entsprechende, vom Parser interpretierbare Symbole.
parser.c	4IA-Parser: Synatktische und semantische Validierung von 4IA-Quellcode und Generierung von entsprechendem ausführbarem Binärcode.
vm.c	Die virtuelle Maschine: Verwaltung der Registerspeicher, Interpretation des ausführbaren Codes, Abfangen von Laufzeitfehlern.
icmpcalc.c	Berechnung der Grundaddition (parasitär oder als Simulation): erstellen, versenden, empfangen und validieren der ICMP Pakete, welche die effektive Addition bilden.
hostlist.c	Hostverwaltung: Verwaltung und Statusbehandlung der an der parasitären Berechnung beteiligten Hosts.
debug.c	Fehlerausgabe: nur zur Entwicklung der Applikation benutzt.

Tabelle 1: Übersicht der `pshell` Module

Die Kommunikation dieser Module untereinander basiert auf Funktionsaufrufen und deren Rückgabewerten sowie auf globalen Variablen. Abbildung 2 zeigt eine abstrahierte Darstellung der Zusammenhänge. In den Nachfolgenden Unterkapiteln werden darauffolgend die Funktionalitäten und Algorithmen der einzelnen Module im Detail erläutert.

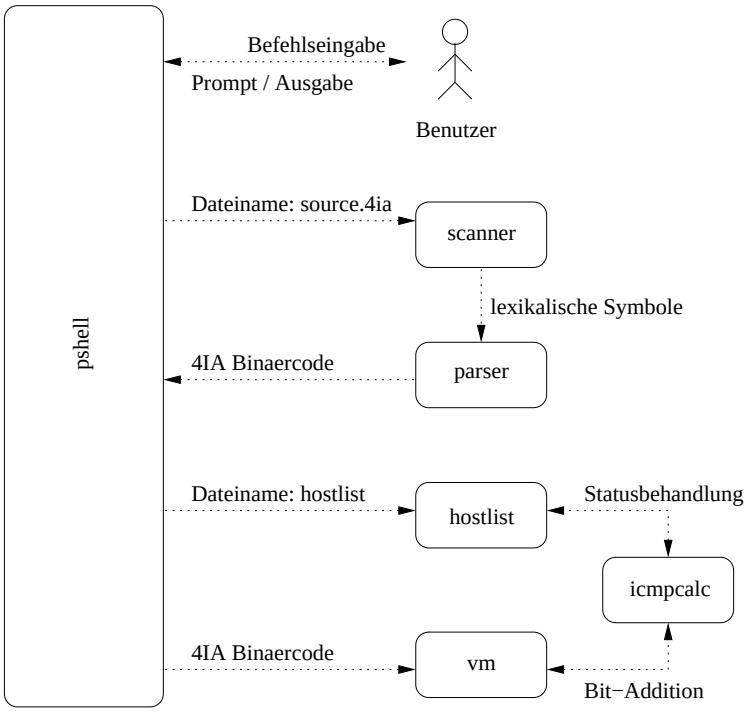

Abbildung 2: Zusammenarbeit der `pshell` Module

2.2 Modul: pshell

Das Modul `pshell` enthält die Funktion `main()` der Applikation und bildet somit den Eintrittspunkt.

2.2.1 Hauptschlaufe

Das Modul besteht primär aus einer Eingabe-Verarbeitungs-Ausgabe-Schlaufe, welche im Pseudocode folgendermassen dargestellt werden kann:

```
print_welcome_message();
while(command != QUIT)
{
  print_prompt();
  line = readline();
  command = parse_command(line);
  switch(command)
  {
    case BEFEHL1:
      aktion_1();
      break;
    case BEFEHL2:
      aktion_2();
      break;
```

```
    ...
    case BEFEHLn:
      aktion_n();
      break;
  }
}
```

Die Funktion `readline()` liest eine Zeile von der standard Eingabe. Die Funktion `parse_command()` implementiert einen trivialen, mit `strncmp()` realisierten Parser und retourniert ein dieser Eingabezeile entsprechendes, interpretierbares Token.

Die Bezeichner `BEFEHL`*n* entsprechen den in [RS02b] aufgelisteten `pshell`-Befehlen, welche durch die dazugehörenden Funktionen `aktion_`*n*`()` ausgeführt werden. Diese Funktionen sind in den im Folgenden beschriebenen Modulen enthalten.

Im Weiteren enthält das Modul `pshell` alle Fuktionen zur Anzeige auf der standard Ausgabe, so beispielsweise die Funktionen `print_stats()` oder `print_help()`.

2.2.2 Algorithmus: CPU-Taktratenermittlung

Für den Vergleich von virtuellen zu reellen CPU-Zyklen ist die Ermittlung der Taktrate des ausführenden Prozessors notwendig. Dies geschieht in der `main()` Funktion und ist – durch bedingte Kompilation realisiert – stark Plattformabhängig. Unter Solaris kann diese Information mittels der Funktion `processor_info()` ausgelesen werden. Unter LINUX wird die Funktion `get_cycles()`, welche die seit Systemstart durchgeführten Zyklen zählt, folgendermassen benutzt:

```
c_start = get_cycles();
sleep(1);
c_end   = get_cycles();
cpu_clockspeed = c_end-c_start;
```

Die Funktion `get_cycles()` liest das `RDTSC` Register der Maschine aus. Es ist nur auf Intel Prozessoren ab Pentium III verfügbar. Auf allen anderen Prozessoren wird, um dennoch eine ungefähre Grössenordnung von virtuellen zu reellen CPU-Zyklen angeben zu können, eine Taktrate von 1 GHz angenommen. Dies ist auch das Vorgehen auf der IRIX Plattform, wo keine Methode zur Ermittlung der Taktrate zur Verfügung steht.

Die in der Statistik angegebene Zahl an reellen Zyklen ergibt sich aus einer Multiplikation des oben ermittelten Wertes mit den Returnwerten der vom System zur Verfügung gestellten Funktion `getrusage()`. Letztere misst die Ausführungszeit eines Prozesses.

2.3 Modul: scanner

Die Eingabedatei des Scanners besteht aus einer Definition aller gültigen lexikalischen Symbole der 4IA-Sprache als reguläre Ausdrücke. Aus dieser Eingabedatei wird mit-

tels des „Fast Lexical Analyser Generator”[3] der effektive Scanner generiert, welcher dem Parser die erkannten Symbole als Tokens liefert.

2.4 Modul: parser

Das Modul Parser implementiert einen simplen Top-Down Parser für die 4IA-Sprache. Die lexikalischen Symbole werden dabei vom generierten Scanner mittels des Funktionsaufrufs `yylex()` geliefert.

Die Semantik der 4IA-Sprache ist sehr einfach, da sich erstens semantisch zusammenhängende Teile nur über jeweils eine Zeile erstrecken und zweitens keine Lookaheads benötigt werden. Durch das erste Symbol einer Zeile ist also die zu verarbeitende Instruktion bereits definiert und es kann eine entsprechende Funktion die Argumente der Zeile verarbeiten. So wird beispielsweise eine mit `ADD` beginnende Instruktionszeile in Pseudocodedarstellung folgendemassen verarbeitet:

```
void parse_add()
{
  c = yylex();
  if(c == DREGISTER) {
    op1 = parse_register();
  } else {
    syntactic_error("direct register");
  }

  c = yylex();
  if(c != COMMA) {
    syntactic_error(",");
  }

  c = yylex();
  if(c == DREGISTER) {
    op2 = parse_register();
  } else {
    syntactic_error("direct register");
  }

  emit_code(OP_ADD, op1, op2);
}
```

Die am Ende stehende Funktion `emit_code()` geneiert nun den binären Code zur Ausführung. Dieser besteht aus einem Array der nachfolgend definierten Struktur:

```
struct code4ia
{
  int opcode;
  int operand1;
  int operand1;
```

[3] siehe `http://www.gnu.org/software/flex/`

```
};
```

Dieses Array wird mit jedem Aufruf von `emit_code()` dynamisch vergrössert.

2.5 Modul: vm

Die virtuelle Maschine ist für die effektive Ausführung des im Array vom Typ `code4ia` enthaltenen binären Codes verantwortlich.

Vor der Ausführung werden alle benötigten Register mit dem Wert 0 sowie alle Statistikzähler, wie beispielsweise der Zähler der virtuellen Prozessorzyklen, initialisiert.

Der Kern des Moduls bildet die Funktion `execute()`, welche bereits in [RS02d] ausführlich beschrieben und hier nun der Vollständigkeit wegen nocheinmal im Pseudocode dargestellt wird:

```
int execute(code4ia code[])
{
  register r[0..n] = 0;
  integer ip = 0;

  while(forever)
  {
    switch(code[r[ip]])
    {
      ADD:
        r[dst] = parasit_add(r[src], r[dst]);
      MOV:
        r[r[dst]] = r[r[src]];
      SET:
        r[dst] = const;
      HLT:
        terminate;
    }
    r[ip] = parasit_add(r[ip], 1);
  }
}
```

Da zur Laufzeit eines 4IA-Programmes zwei mögliche Fehler auftreten können, ist der Code in der effektiven Implementation (hier nicht dargestellt) um zwei dementsprechende Fehlerprüfungen erweitert.

Es muss erstens vor jeder `MOV` Instruktion geprüft werden, ob mittels indirekter Adressierung nicht auf Register zugegriffen wird, die nicht existieren:

```
if(r[code[r[ip]].operand1] >= number_of_regs ||
   r[code[r[ip]].operand2] >= number_of_regs )
{
  /* Fehlermeldung ausgeben */
} else {
```

```
  /* Adressierung i.O. */
}
```

Zweitens muss sichergestelt werden, dass der Instruktionszeiger nicht auf eine Position im Code (Array `code[]`) gesetzt wird, die nicht existiert:

```
if(r[ip] >= codesize || r[ip] < 0)
{
  /* Fehlermeldung ausgeben */
} else {
  /* Instruktionszeiger i.O. */
}
```

Die Variable `codesize` enthält dabei die Anzahl der im Array `code[]` enthaltenen Elemente.

2.6 Modul: icmpcalc

Das Modul icmpcalc stellt dem Modul vm die Funktion `parasit_add(int *flag, int op1, int op2)` zur Verfügung. Durch diese Funktion kann die virtuelle Maschine die der Berechnung zugrundeliegenden Additionen durchführen. Die Funktion `parasit_add(...)` ist dabei als Wrapper implementiert und ruft, je nach Zustand der globalen Variable `executiontype` (mit den möglichen Werten `SIMULATION` oder `ICMPCALC`) die entsprechende Additionsfunktion auf.

2.6.1 Simulation

Die Additionsfunktion der Simulation ist denkbar einfach und besteht aus folgendem Codeabschnitt:

```
int parasit_add_simulate(int *fl, int op1, int op2)
{
  int result;
  int maxint;

  maxint = (1<<register_width)-1;

  result = op1 + op2;
  if(result > maxint) {
    result = result%(maxint+1);
    *fl = 1;
  }

  return result;
}
```

Die Variable `maxint` enthält den mit der definierten Registerbreite `register_width` höchstmöglichen Ganzzahlwert. Anhand dieses Wertes wird auf Additions-Überlauf geprüft und, falls ein solcher stattfindet, das Flag-Register (`fl`) auf 1 gesetzt.

2.6.2 Parasitäre Berechnung

Die parasitäre Berechnung basiert auf dem in [RS02d] erläuterten Prinzip der Kandidatlösungsprüfung. Der exakte Algorithmus zur Aufteilung der Bit-Additionen auf die involvierten Hosts ist in Abschnitt 2.7.2 aufgeführt, da die dort beschriebene Hostliste für die eigentliche Aufteilung zuständig ist.

Die in [RS02d] erläuterte Berechnung basiert darauf, dass pro Sendeimpuls[4] jeweils nur eine 1-Bit Addition durchgeführt wird. Es ist aber auch denkbar, in einem Sendeimpuls die Lösungen von breiteren als 1-Bit Additionen zu prüfen und somit eine Parallelisierung der Grundoperation zu erreichen. Dabei entspricht die Anzahl der betötigten Netzwerkpakete

$$AnzahlPakete = \frac{RB}{x} \times 2^{(x+1)}$$

wobei *RB* die Registerbreite der virtuellen Maschine in Bits und *x* die Parallelisierungskonstante ist. Letztere definiert, wieviele Bitstellen pro Sendeimpuls berechnet werden sollen. Für die Parallelisierungskonstante gibt es zwei Einschränkungen. Erstens muss sie ein ganzer Teiler der Registerbreite sein, denn nur so lässt sich das ganze Register gleichmässig aufteilen, zweitens darf sie nicht höher als 4 sein. Der nächstmögliche Wert nach 4 wäre 8, dies geht aber bereits nicht mehr, da sich – die Netzwerkpakete sind 16-Bit breit – Operatoren und Checksummen überschneiden würden.

Die Operanden-Argumente der vorhin genannten Funktion `parasit_add(...)` sind also nicht in jedem Falle in 1-Bit Operationen aufzuteilen, sondern in so grosse Operationen, wie dies die globale Variable `bits_per_add` (definiert durch den `pshell`-Befehl `setbits`) definiert. Die durch `parasit_add(...)` aufgerufene Funktion `parasit_add_icmp(...)` führt diese Aufteilung durch. Dieser Ablauf ist in Abbildung 3 dargestellt.

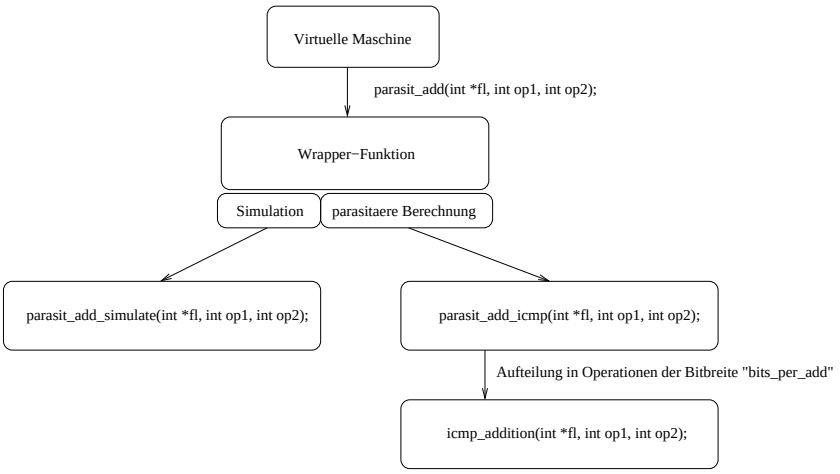

Abbildung 3: `parasit_add()` Wrapper-Funktionen

[4]Nach dem beschriebenen Prinzip der Kandidatlösungsprüfung benötigt die Berechnung eines Teilresultates mehrere Netzwerkpakete. Diese Ansammlung an gesendeten Paketen pro Teilresultat bezeichnen wir als Sendeimpuls.

Durch den modularen Aufbau mittels dieser Wrapper-Funktion kann das Softwarepaket – sollte wie im Pflichtenheft angetönt eine weitere Methode parasitärer Berechnungsart gefunden werden – jederzeit mit geringstem Aufwand um neue Additionsarten erweitert werden.

Es folgt nun ein kurzer Auszug aus der Funktion `parasit_add_icmp(...)` um die Aufteilung der Operandenbits zu veranschaulichen:

```
int parasit_add_icmp(int *fl, int op1, int op2)
{
  for(i=0; i<register_width; i+=bits_per_add)
  opA = op1&((1<<bits_per_add)-1);
  opB = op2&((1<<bits_per_add)-1);

  res = icmp_addition(&opU, opA, opB, hostno);

  complete_result += (res<<i);

  op1>>=bits_per_add;
  op2>>=bits_per_add;

  return complete_result;
}
```

Die Funktion `icmp_addition()` führt nun die effektive Berechnung mittels der ICMP Pakete durch. Dieses Vorgehen ist in [RS02d] bereits ausführlich erläutert.

2.6.3 Vorkalkulierte Sequenznummern

Wie im Realisierungskonzept erklärt wurde, muss jedes gesendete ICMP Paket mit einer eindeutigen Sequenznummer versehen werden. Diese Sequenznummer wird in der Antwort der Gegenseite auf das eine korrekte Paket an den Absender zurückgeschickt, womit im Modul icmpcalc die richtige Kandidatlösung eindeutig identifiziert werden kann.

Die benötigten Sequenznummern (ihre Anzahl entspricht der Anzahl Netzwerkpakete gemäss der Formel in Abschnitt 2.6.2) werden nicht zu Laufzeit kalkuliert, sondern durch den Header `precalc.h` importiert. Im Kommentarteil dieses Headers ist ein PERL[5] Programm enthalten, welches zur Generierung benutzt wurde.

Der nur durch die unterschiedlichen Sequenznummern beeinflusste Teil der Checksummen (die höherwertigen 8 Bit) ist ebenfalls vorkalkuliert und in `precalc.h` definiert.

2.6.4 Plattformabhängigkeiten

Das Modul icmpcalc gewinnt Aufgrund entscheidender Unterschiede zwischen den einzelnen Plattformen (namentlich den Unterschieden von LINUX zu anderen UNICES) zusätzlich an Komplexität. So sind die von den Systemen benutzen Datenstrukturen für ICMP Pakete sehr unterschiedlich, wie beispielsweise folgender Ausschnitt zeigt:

[5] PERL: „Practical Extraction and Reporting Language", siehe `http://www.perl.com/`.

```
#if !defined (LINUX)
  memset(header, 0x0, sizeof(struct icmp));
  header->icmp_type = ICMP_ECHO;
  header->icmp_id = htons(PARASIT_ID);
  header->icmp_seq = htons(PARASIT_ID);
  header->icmp_cksum = htons(PARASIT_ID);
  memcpy(buff, header, sizeof(struct icmp));
#else
  memset(header, 0x0, sizeof(struct icmphdr));
  header->type = ICMP_ECHO;
  header->un.echo.id = htons(PARASIT_ID);
  header->un.echo.sequence = htons(PARASIT_ID);
  header->checksum = htons(PARASIT_ID);
  memcpy(buff, header, sizeof(struct icmphdr));
#endif
```

Aus dem abgebildeten Code wird ersichtlich, wie diese Probleme in der Arbeit gelöst wurden: die unterschiedlichen Strukturen der einzelnen Plattformen sind durch C Präprozessormakros umgangen worden.

2.7 Modul: hostlist

Das Modul Hostlist dient zur Verwaltung der in die parasitäre Berechnung involvierten Hosts und deren Attributen. Es kapselt dazu folgende Datenstruktur:

```
struct hostlist
{
  char hostname[MAXHOSTNAMELEN];
  char hostaddress[MAXHOSTNAMELEN];
  int enabled;
  int timeouts;
  int packetcount;
  int false_positive;
};
```

Diese Struktur wird durch Aufruf der Funktion `read_hostlist()`, welche ihrerseits durch den `pshell`-Befehl `loadhosts` ausgelöst wird, gefüllt. Dabei werden die Hostnamen durch die Resolver-Library zu IP-Adressen aufgelöst und die Attributfelder initialisiert. Letztere haben folgende Bedeutungen:

- `enabled`
  Das Feld `enabled` besagt, ob ein Host für weitere Berechnungen benutzt werden soll oder nicht. Es wird mit 1 initialisiert, was dem Wert TRUE entspricht.
- `timeouts`
  Das Feld `timeouts` enthält die Anzahl Timeouts, welche ein Host verursachte.
- `packetcount`
  Das Feld `packetcount` enthält die Anzahl ICMP Pakete, welche dem entsprechenden Host geschickt wurden. Es dient zu statistischen Zwecken.

- `false_positive`
  Das Feld `false_positiv` gibt Auskunft, ob für den entsprechenden Host ein sogenannter False-Positive Test bereits ausgeführt wurde oder nicht. Der False-Positive Test wird im nachfolgenden Abschnitt erläutert.

Die Datenstruktur `hostlist` wird dynamisch alloziert und bleibt solange mit sämtlichen Attributen erhalten, bis das die `pshell`-Umgebung entweder terminiert oder eine neue Hostliste geladen wird.

Das Modul icmpcalc kommuniziert mit dem Modul hostlist ausschliesslich über die selbsterklärende Funktion `get_next_host()`.

2.7.1 False Positives

In der Testphase hat es sich herausgestellt, dass es vereinzelte Hosts gibt, welche auch ICMP Pakete mit falschen Checksummen als korrekt beantworten. Solche falschen Antworten sind für die Berechnung fatal und müssen erkannt werden. Dazu wird jedem Host vor dem ersten realen Paket ein Testpaket mit einer falschen Checksumme geschickt um zu verifizieren, ob er dieses beantwortet oder nicht. Wird das falsche Paket beantwortet, so ist der Host für weitere Berechnungen unbrauchbar und wird deshalb mittels dem Attribut `enabled` als ungültig markiert. Das Attribut `false_positive` dient dabei lediglich dazu anzuzeigen, ob für einen Host bereits ein False-Positive Test durchgeführt wurde oder nicht.

2.7.2 Algorithmus: parasitäre Verteilung

Im folgenden wird der Algorithmus, wie die Kandidatlösungen auf die involvierten Hosts der Hostliste aufgeteilt werden, dargestellt:

```
while(vorhanden: hosts[].enabled == true)
{
  host := naechster host mit enabled == true;
  if(host.false_positive checked == false)
  {
    status := false_positive_check(host);
    if(status == false)
    {
      host.enabled := false;
      continue;
    }
  }
  generiere(icmp_pakete);
  sende(icmp_pakete, host);
  inkrementiere(host.packetcount);
  warte(timeout);
  if(antwort innerhalb timeout)
  {
    if(antwort == genau 1 paket)
    {
      auswerten(antwort);
    } else {
```

```
      host.enabled := false;
    }
  } else {
    inkrementiere(host.timeout);
    if(host.timeout == timeout_threshold)
    {
      host.enabled := false;
    }
  }
}
```

Der Algorithmus läuft also solange, wie es in der Hostliste noch gültige, für die Berechnung brauchbare Hosts gibt. Es ist durchaus möglich, dass ein Programm abgebrochen werden muss, wenn alle Hosts der Hostliste das Attribut `enabled` auf 0 (FALSE) gesetzt haben.

Fehler in der Berechnung sind nach diesem Algorithmus nur noch möglich, wenn ein Host eines (und genau eines) der falschen Pakete als korrekt und alle anderen Pakete (inklusive das einzelne korrekte) als falsch behandelt. Ein solcher Fall ist in der Praxis kaum denkbar und konnte in der Testphase nicht prozudiert werden. Durch gewollte Manipulation der beteiligten Hosts kann dieser Zustand allerdings relativ leicht erreicht werden.[6]

2.8 Modul: debug

Das Modul Debug wird nur für Entwicklungszwecke benötigt und dient der Fehlersuche in der Applikation. Es besteht lediglich aus einer Funktion mit folgendem Prototyp:

```
void debug(int level, char *message);
```

Dabei entspricht `level` demjenigen Debug-Level, zu welchem die übergebene Meldung `message` gehört. Folgende Debug-Level sind definiert:

```
#define DEBUG_NONE        0
#define DEBUG_PARSER      1
#define DEBUG_CODEGEN     2
#define DEBUG_EXECUTION   4
#define DEBUG_DUMPREGS    8
#define DEBUG_HOSTLIST   16
#define DEBUG_ICMP       32
#define DEBUG_TRACE      64
```

So würde beispielsweise eine Debug-Meldung aus dem Modul Parser die Funktion `debug()` mit dem Wert `DEBUG_PARSER` als Argument `level` aufrufen. Für die Applikation wird nun global ein Ausgabe-Debug-Level als binäre-oder Kombination

[6] Die einzige Möglichkeit, um solche Fehler abzufangen, wäre die Verifizierung jeder Aktion durch einen weiteren Host, was die Zeitkomplexität der Berechnung verdoppeln, beziehungsweise im Zweifelsfall vervielfachen würde.

aus den dargestellten Werten definiert. Um beispielsweise alle Debug Meldungen der Codegenerierung und des Parsers zu erhalten:

```
#define DEBUG_LEVEL DEBUG_CODEGEN|DEBUG_PARSER
```

Damit lässt sich nun das Vorgehen zur Entscheidung, ob eine Meldung ausgegeben wird oder nicht, der Funktion `debug()` veranschaulichen:

```
void debug(int level, char *message)
{
  if((level)&(DEBUG_LEVEL))
  {
    printf("%s", message);
  }
}
```

Hierbei ist anzumerken, dass die vielen Aufrufe der Funktion `debug()` eine nicht unwesentliche Performance-Einbusse verursachen. Dies liesse sich mittels bedingter Kompilation umgehen, was allerdings aus Transparenzgründen nicht realisiert wurde.

2.9 Weiterführende Dokumentation

Auf der sich im Umfang dieser Arbeit befindenden CD-ROM ist eine mit Doxygen[7] direkt aus dem Source-Code generierte Referenz abgelegt, welche als Erweiterung des gesamten Kapitels 2 dieses Dokumentes dient. Zusätzlich sind im Source-Code selber sämtliche Funktionen und alle relevanten Code-Abschnitte mit ausführlichen Kommentaren versehen.

3 Der `xto4` Cross-Compiler

Das Paket `xto4`, zuständig für die Kompilation von XIA zu 4IA-Code, wird in den folgenden Subkapiteln zerlegt in einzelne Teilpakete. Diese werden jeweils im Detail erklärt und ihre Schnittstellen zu den andern Teilpaketen aufgezeigt.

3.1 Allgemeines

Das Paket `xto4` besteht im Wesentlichen aus vier Teilpaketen, nachfolgend Packages[8] respektive Haupt-Packages genannt, sowie zwei weiteren kleinen Packages (*util* und *global*), welche von diesen vier Haupt-Packages benutzt werden. In Tabelle 2 werden alle sechs Packages erläutert und kurz deren Aufgaben vorgestellt:

[7] siehe `http://www.doxygen.org/`

[8] Der Name *Package* kommt von Seiten Java. Packages definieren eine Kollektion von Klassen und gegebenenfalls weiteren Sub-Packages (Unterpakete), welche zusammen bestimmte Aufgaben und Funktionen erledigen. Typischerweise wird ein Package bestimmt durch einen Ordner, worin sich die Klassen sowie weitere Sub-Packages befinden.

Package	*Funktion*
Scanner	Öffnet die Datei mit dem XIA-Code, liest Zeile um Zeile und bildet daraus eine Folge gültiger Tokens.
Resolver	Löst die relativen Sprungadressen auf in absolute Zeilennummern und ersetzt die virtuellen Register (`fl`, `cf`, `ip` und `ec`) durch ihre entsprechenden, absoluten Register.
Compiler	Verarbeitet Tokens und bildet daraus 4IA-Code, welcher aber noch relative, nicht aufgelöste Sprungadressen enthält.
Optimizer	Ist nicht implementiert. Würde den an sich lauffähigen 4IA-Code bezüglich Doppelsprüngen, indirekten Verweisen auf andere Register und so weiter erkennen und eliminieren.
Global	Enthält Konstanten und Methoden, welche packageübergreifend benutzt werden.
Util	Kollektion von Methoden, welche die Funktionalitäten im Zusammenhang mit Strings und String Arrays erweitern.

Tabelle 2: Übersicht der vorhandenen `xto4` Packages

Ein weiteres wichtiges Package, welches alle oben genannten Packages bis auf das Package *util* enthält, lautet `xia`, nachfolgend auch Main-Package genannt. Dieses beinhaltet alle Klassen, die spezifisch für diesen Cross-Compiler entwickelt worden sind. Im Weiteren befindet sich die Klasse `Xto4` direkt im Main-Package, welche die vier Haupt Packages miteinander arbeiten lässt. Diese Klasse beinhaltet die `main`-Methode.

Abbildung 4 verdeutlicht die Package-Hierarchie und zeigt vereinfacht auf, in welche Teilaufgaben der Cross-Compiler unterteilt wurde.

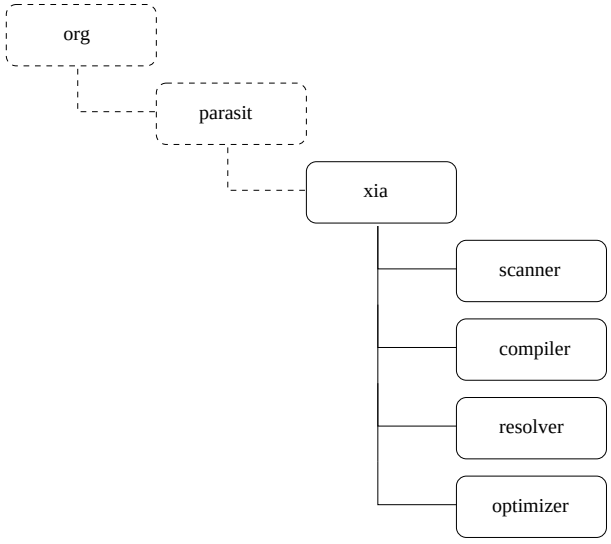

Abbildung 4: Package Hierarchie des Cross-Compilers `xto4`

Die Packages `xia` und `util` befinden sich gemäss Java Spezifikation im Wurzelver-

zeichnis: `org.parasit`.

3.2 Interfaces

Zum besserern Verständnis der Klassendiagramme ist es hilfreich zu wissen, dass Interfaces nicht nur zur Deklaration gewisser Gemeinsamkeiten innerhalb von Klassen dienen, sondern darüber hinaus auch dazu benutzt werden können, Konstanten, welche von mehreren Klassen verwendet werden, elegant zur Verfügung zu stellen. `xto4` macht sich diese Eigenschaft von Interfaces in Folgenden Klassen zu nutze: `ArgumentTypes` und `RegisterConstants`.

3.3 Packages

Nachfolgende Unterkapitel erklären den Aufbau und die Funktionsweise des jeweiligen Packages und gegebenenfalls seinen Sub-Packages[9]. Die Klassendiagramme sollen aufzeigen, welche Klassen sich in den jeweiligen Packages befinden und welche Beziehungen Klassen innerhalb eines Package zueinander haben.

Packages haben zum Ziel, Klassen zu paketieren, sprich zu ordnen nach ihren Aufgaben und Funktionen, die sie wahrnehmen. Gut zu erkennen ist dies an den Namen der vier Haupt-Packages sowie deren Sub-Packages. So ist etwa das Haupt-Package `compiler` dafür verantwortlich, den `XIA`-Code in `4IA`-Code zu kompilieren. Das Sub-Package `compiler.exception` enthält dabei alle Typen von Exceptions, welche im Zusammenhang mit der Kompilation auftreten können. Das Package `xia` wiederum dient als Haupt-Package für alle Klassen, welche spzifisch entwickelt worden sind im Zusammenhang mit der Diplomarbeit „Parasitic Computing". Das Sub-Package `util`, das sich direkt im Package `xia` befindet, wurde nicht spezifisch für die Diplomarbeit entwickelt und gehört demnach nicht ins Verzeichnis `xia`.

3.3.1 main

Das Package Main enthält, wie bereits erläutert, alle Haupt-Packages sowie das Hilfspackage *global*. Im weiteren befindet sich die Klasse `Xto4` darin, welche die `main` Methode enthält. Dies bedeutet, dass diese Klasse primär die Benutzerschnittstelle bildet und zuständig für die Zusammenarbeit aller andern Packages und Klassen ist.

Package Übersicht

[9] Ein Sub-Package ist ein Package innerhalb eines Packages.

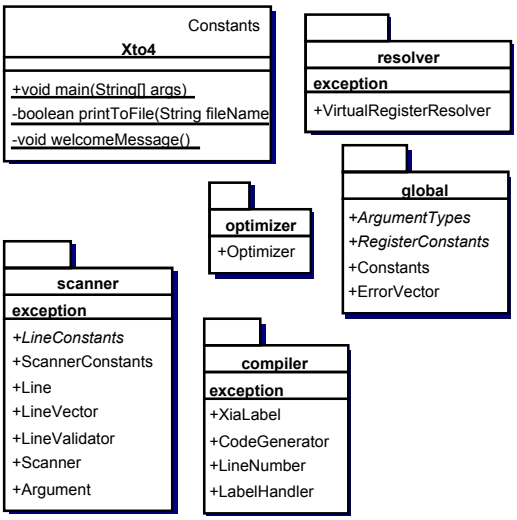

Abbildung 5: Übersicht der Packages

Die Klasse Xto4

Die Klasse `Xto4`, nachfolgend Main-Klasse genannt, ist zuständig für die Kontrolle und die Koordination der Funktionalitäten, welche die vier Haupt-Packages anbieten. Die Schnittstelle zu den Haupt- Packages ist prinzipiell immer die gleiche (mit Ausnahme von Resolver, welches keinen Rückgabewert hat, sondern vielmehr die entsprechenden Werte der Objekte direkt ändert, erkennbar an der gestrichelten Linie, welche den Rückgabeweg darstellt). Ein String[10] oder ein Vector mit Strings, der jeweils eine Zeile repräsentieren, wird der Instanz der verantwortlichen Klasse des entsprechenden Packages übergeben, woraus als Rückgabewert wiederum ein entsprechend verarbeiteter String resultiert. Der sequentielle Ablauf ist in Abbildung6 dargestellt.

[10]Unter String versteht sich eine Abfolge von Characters, also Zeichen. Ein String ist zu Beginn des Programmes typischerweise XIA-Code, der nach erfolgreicher Beendigung des Programmes kompiliert ist zu 4IA-Code.

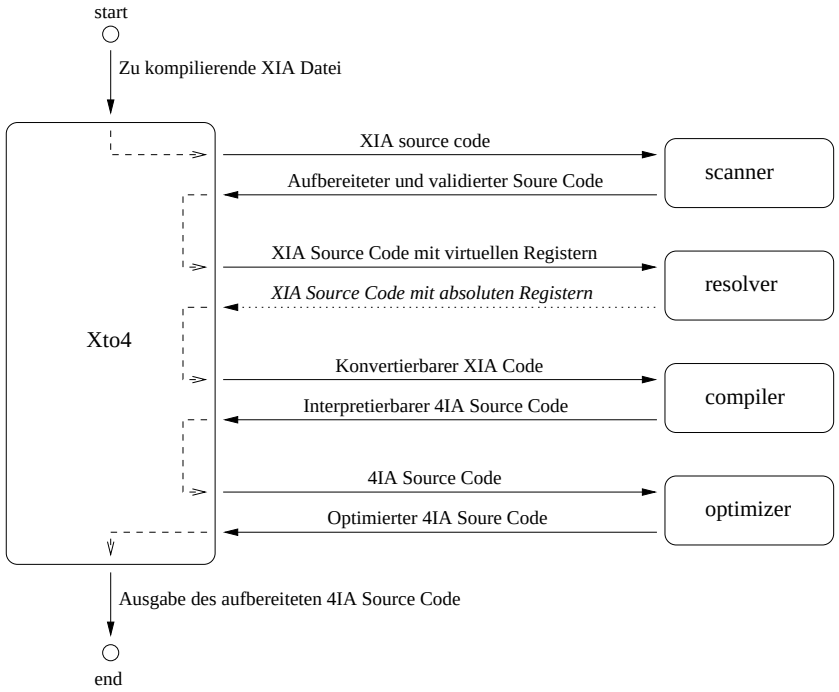

Abbildung 6: Sequentieller Ablauf des Programmes

Im Falle, dass ein Schritt bei der Kompilation nicht erfolgreich durchgeführt werden konnte, wirft die entsprechende Klasse eine Exception, welche von der Main Klasse abgefangen und als entsprechende Fehlermeldung auf der Konsole ausgegeben wird. Andernfalls wird je nach Anzahl Argumenten, welche dem Programm übergeben wurden, der kompilierte 4IA-Code auf der Konsole ausgegeben oder aber in eine Datei gespeichert.

3.3.2 Scanner

Das Package `Scanner` mit den dazugehörigen Klassen ist dafür verantwortlich, eine Datei mit XIA-Code einzulesen und die darin enthaltenen Instruktionen mit ihren Argumenten und Kommentaren auf ihre syntaktische Korrektheit zu validieren, so dass das Package `Resolver` den XIA-Code weiter verarbeiten kann. Das Package `Scanner` übernimmt damit auch die Aufgaben des Parsers.

Klassendiagramm
Die Klassendiagramme in Abbildung 7 und 8 zeigen auf, welche Klassen involviert sind zum Scannen und Parsen von XIA-Code und dessen Aufbereitung in einen Vector.

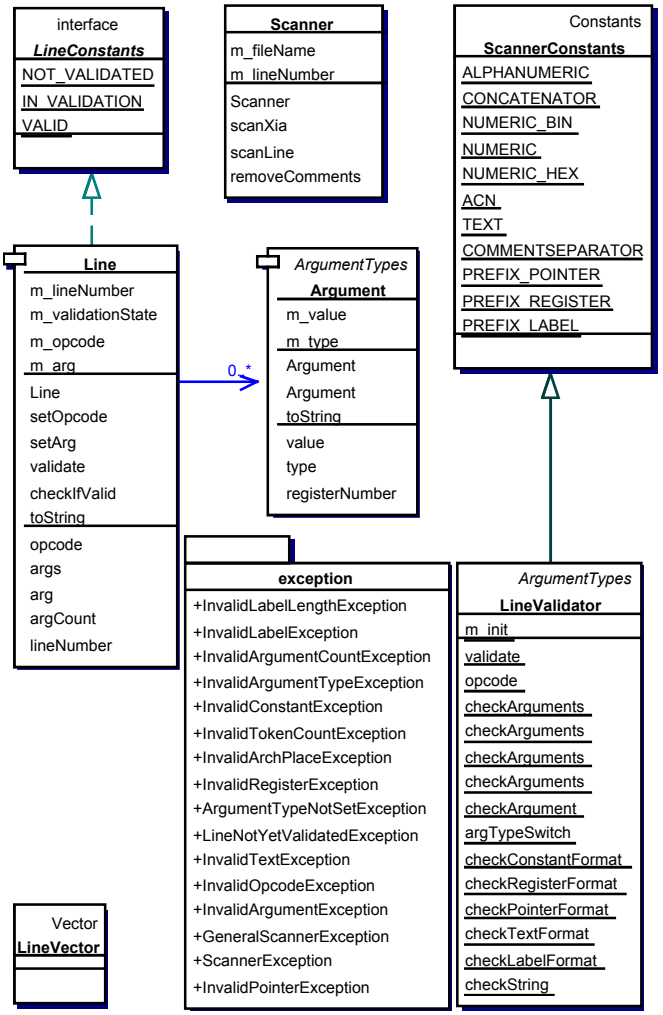

Abbildung 7: Klassendiagramm des Package `scanner`

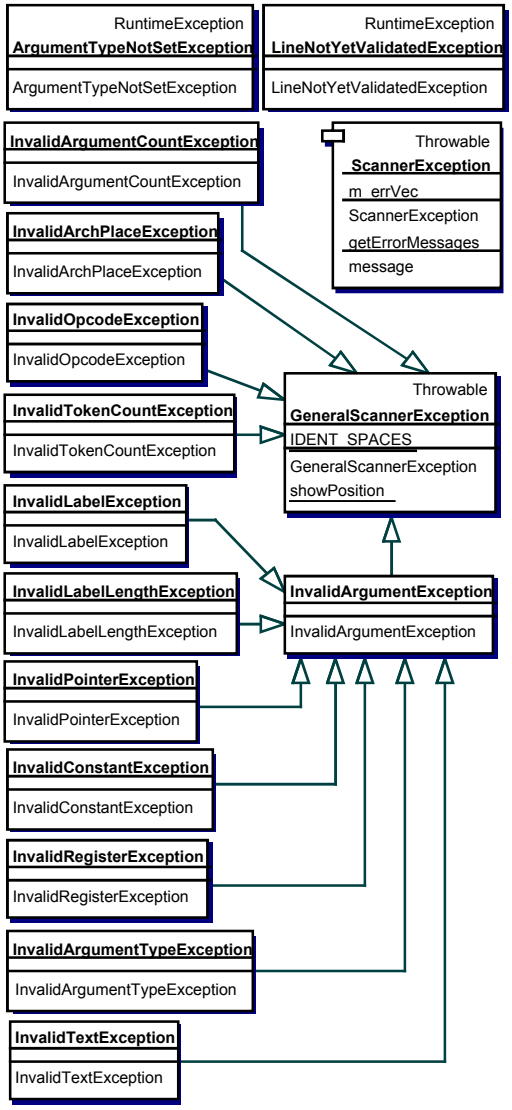

Abbildung 8: Klassendiagramm des Package `scanner.exception`

Schnittstelle
Die von der Main-Klasse `xto4` zum Scannen verwendete Schnittstelle definiert sich zum einen durch den Konstruktor und zum andern durch die Methode `scanXia()`. Ersterer erwartet als Argument einen String mit einem relativen Pfad auf die zu scannende Datei, worauf letztere den Scanning-Prozess startet.

Scanning Prozess
Die Klasse *Scanner* ist dafür verantwortlich, Linie um Linie der Datei zu lesen. Jede gelesene Linie wird bereinigt, sprich Kommentare und Leerzeilen werden gelöscht,

und mit einem StringTokenizer werden Tokens gebildet. Mit diesen Tokens und der aktuellen Zeilennummer wird eine Instanz vom Typ *Line* geschaffen, die dann dafür verantwortlich ist, sich zu validieren oder gegebenenfalls eine *GeneralScannerException* zu werfen. Der implementierte Scanner ist demnach denkbar einfach und basiert nicht auf einer bereits bestehenden Scanner Technik im klassischen Sinne, zumal die zu scannende Sprache weder Verschachtelungen noch zeilenübergreiffende Instruktionen kennt. Vielmehr sind der Aufbau einer Zeile und die zu erwartenden Tokens wohl bekannt und streng definiert.

Linien Validierung
Jede erzeugte *Line*-Instanz validiert ihren Opcode mit den dazugehörenden Argumenten anhand der statischen Methode `validate(Line)` der Klasse *LineValidator*. *LineValidator* besitzt Informationen darüber, welcher Opcode wieviele Argumente von welchem Typ besitzen muss. Folgende Kontrollen werden durchgeführt, bevor eine Linie für gültig deklariert werden kann:

- Opcode gültig
- Anzahl Argumente korrekt
- Entspricht das Argument einem gültigen Typ

Falls alle oben genannten Kontrollen erfolgreich durchgeführt werden konnten, wird die Linie für gültig deklariert und einem *LineVector* in der Klasse *Scanner* hinzugefügt. Falls die Kontrolle fehlgeschlagen hat, wird eine Exception mit entsprechender Nachricht geworfen und vom *Scanner* aufgefangen. Diese Exception wird einem *ErrorVector* hinzugefügt.

Abbildung 9 verdeutlicht in vereinfachter Weise den Ablauf einer *Line* Validierung.

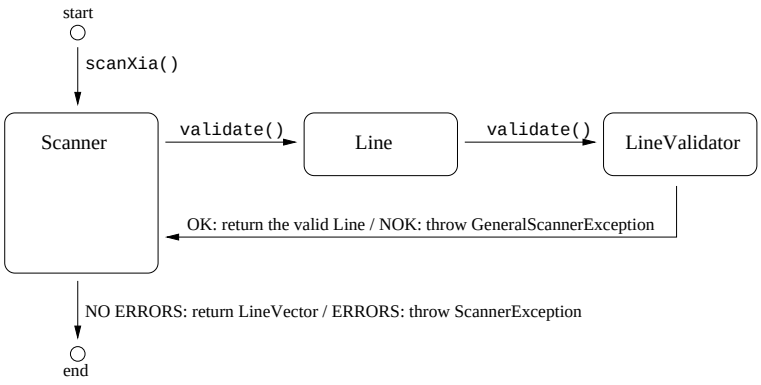

Abbildung 9: Vereinfachter Ablauf der Validierungsphase

Rückgabewert
Wie bereits erwähnt, werden im Fall von fehlgeschlagenen Kontrollen zur Validierung von Linien *GeneralScannerException* geworfen, die vom *Scanner* aufgefangen und in einem *ErrorVector* gesammelt werden. Anhand der Elemente in diesem *ErrorVector*

entscheidet der *Scanner* nach dem Scannen jeder Linie, ob der *LineVector* an die Main Klasse zurückgegeben oder eine Exception geworfen werden soll. Enthält der *ErrorVector* mindestens ein Element, so wirft die `validate()`-Methode des *Scanner* eine *ScannerException*, welche den *ErrorVector* auswertet und die entsprechenden Fehlermeldungen aufbereitet zu einem String, welcher dann auf der Konsole ausgegeben werden kann und den Benutzer informiert, welche Zeilen fehlerhaft sind.

In jedem Fall ist damit die Aufgabe des *Scanner* abgeschlossen.

3.3.3 Resolver

Das Package Resolver mit den dazugehörigen Klassen ist dafür verantwortlich, die virtuellen Register zu ersetzen in Registernamen, welche von der virtuellen Maschine verstanden werden. Zu den virtuellen Register zählen der Instruction Pointer `ip`, die zwei Carry-Flag `fl` (4IA-seitig) und `cf` (XIA-seitig) sowie das Error-Code Register. Im Weiteren werden die Prefixes für Register und Pointer, konkret sind dies die Zeichen `'r'` und `'*'`, konvertiert in 4IA spezifische Prefixes. Die Registernummern werden überprüft, ob sie sich in einem gültigen Bereich befinden und es werden Informationen darüber gesammelt, so dass ein aussagekräftiger Header generiert werden kann mit Statistiken über die benutzten Register.
Das Package Resolver ist auch dafür verantwortlich, die Zeile mit dem Opcode `ARCH` und seinen zwei Parametern zu generieren, besitzt doch *VirtualRegisterResolver* die nötigen Informationen dazu.

Klassendiagramm
Die Klassendiagramme in Abbildung 10 und 11 zeigen auf, welche Klassen involviert sind im Zusammenhang mit dem *VirtualRegisterResolver*.

Abbildung 10: Klassendiagramm des Package `resolver`

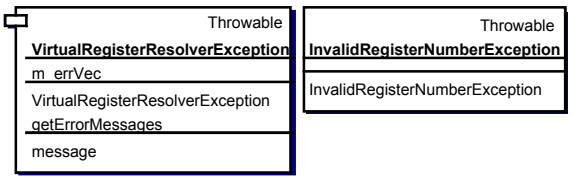

Abbildung 11: Klassendiagramm des Package `resolver.exception`

Schnittstelle
Die Schnittstelle, welche von der Main Klasse Xto4 verwendet wird zur Auflösung der virtuellen Register, wird definiert durch die Methode `resolve(`*LineVector*`)`. Dieses Package gibt keinen Vector oder sonstige Strings zurück. Vielmehr werden alle Änderungen direkt an den jeweiligen Argument-Objekten vorgenommen, nach dem Prinzip *call by reference*. Somit erübrigen sich Rückgabewerte. Einzig falls Fehler auftraten beim Auflösen und Validieren von Registern und deren Nummern, wird eine Exception vom Typ *VirtualRegisterResolverException* geworfen.

Funktionsweise
Der erhaltene *LineVector*, welcher den aufbereiteten XIA-Code Linie für Linie enthält

(siehe Abschnitt 3.3.2 auf Seite 130), wird durchenumeriert und jedes Argument *Argument* pro Linie (Objekt Typ *Line*) wird entsprechend seinem Typ behandelt. Folgende Typen existieren jeweils als Pointer wie auch als direktes Register: Virtuelle Register `ip, fl, cf, ec` sowie Register (Im Code XIA-weit werden eben erwähnte Register wie folgt benutzt: `ip, *ip, fl, *fl, cf, *cf, ec, *ec, rx, *rx`). Abbildung 12 verdeutlicht die Zusammenhänge.

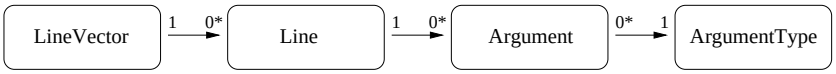

Abbildung 12: Prinzipieller Aufbau eines LineVector

Falls es sich um ein virtuelles Register handelt, wird das Token des Arguments ersetzt mit demjenigen Token, welches 4IA-seitig Gültigkeit besitzt. Dabei wird der allenfalls vorhandene Pointer-Prefix beibehalten. Falls es sich um ein absolutes Register oder um einen Pointer auf ein Register handelt, werden folgende Schritte durchgeführt:

- Prüfen, ob sich Register-Nummer im gültigen Bereich befindet
- Gegebenenfalls Statistiken betreffend der benutzten Register nachführen
- Register- respektive Pointer-Prefix konvertieren

Der gültige Bereich für verwendbare Register liegt zwischen denjenigen Registern, die der Kompiler braucht um temporäre Werte zwischenzuspeichern, und der maximal adressierbaren Register Anzahl, die XIA-seitig mittels der Instruktion `ARCH` indirekt überschrieben werden kann. Indirekt deshalb, weil nur die Registerbreite der virtuellen Maschine definiert werden kann, welche aber wie folgt vorgibt, wieviele Register adressierbar sind: $2^{Registerbreite} - 1$. Überschreiben, weil standardmässig eine Registerbreite von 8 Bit gesetzt wird, was eine Register Obergrenze von 255 ergibt.

Statistiken betreffend der benutzten Register werden nur nachgeführt, wenn ein Register verwendet wird, dessen Nummer kleiner respektive grösser ist als diejenigen der bereits benutzten.

Mit der definierten Syntax der XIA-Sprache erübrigen sich Konvertierungen von Register- und Pointer-Prefixen zu 4IA, da beide Sprachen die selbe Notation verwenden.

Statistiken
Nachfolgend ein generierter Header, welcher Aufzeigen soll, welche statistischen Informationen gesammelt werden:

```
; This 4ia code has been compiled by Xto4 Cross-Compiler, Version x
; Copyright (c) 2002 Luzian Scherrer, Juerg Reusser
; Check out http://www.parasit.org for further informations.
;
; VM settings to run the following code:
;   Lowest useable register : r15
;   Really used registers   : 20
;     lowest  register used : r20
;     highest register used : r37
;   Register width in bits  : 8
;   Addressable registers   : 256
```

```
;   decrementing constant   : 255
;   Opcodes used 4ia wide   : SET, MOV, ADD, HLT
;
; NOTE: Statistics considers only directly addressed registers.
; Initialization parameters for the virtual machine
ARCH 8 38
```

Rückgabewerte
Wie bereits erwähnt gibt der *VirtualRegisterResolver* der Main Klasse weder einen Vektor noch einen String zurück, da sämtliche Werte direkt *by reference* verändert werden. Falls sich aber während des Auflösungsprozesses Fehler ereigneten, dann wirft diese Methode eine Exception vom Typ *VirtualRegisterResolverException*, welche als „Message" die Informationen über die fehlerhaften Linien und die Gründe dafür enthält. Im weiteren besitzt die Klasse *VirtualRegisterResolver* nebst „private" Methoden die „public" Methode `getHeaderInfos()`, welche nach dem Auflösungsprozess die generierten Headerzeilen zur Verfügung stellt.

Damit endet der Zuständigkeitsbereich des Package `resolver`.

3.3.4 Compiler

Das Package `compiler` mit den dazugehörigen Klassen ist dafür verantwortlich, den aufbereiteten XIA-Code zeilenweise zu kompilieren in 4IA-Code, welcher interpretierbar ist für die virtuelle Maschine.
Bei erfolgreichem Kompilieren wird der generierte rohe 4IA-Code in Form eines String Objekts retourniert, das dann vom „Optimizer" gegebenenfalls weiter verarbeitet werden kann.

Klassendiagramm
Die Klassendiagramme in Abbildung 13 und 14 zeigen auf, welche Klassen zur Kompilation von XIA-Code involviert sind.

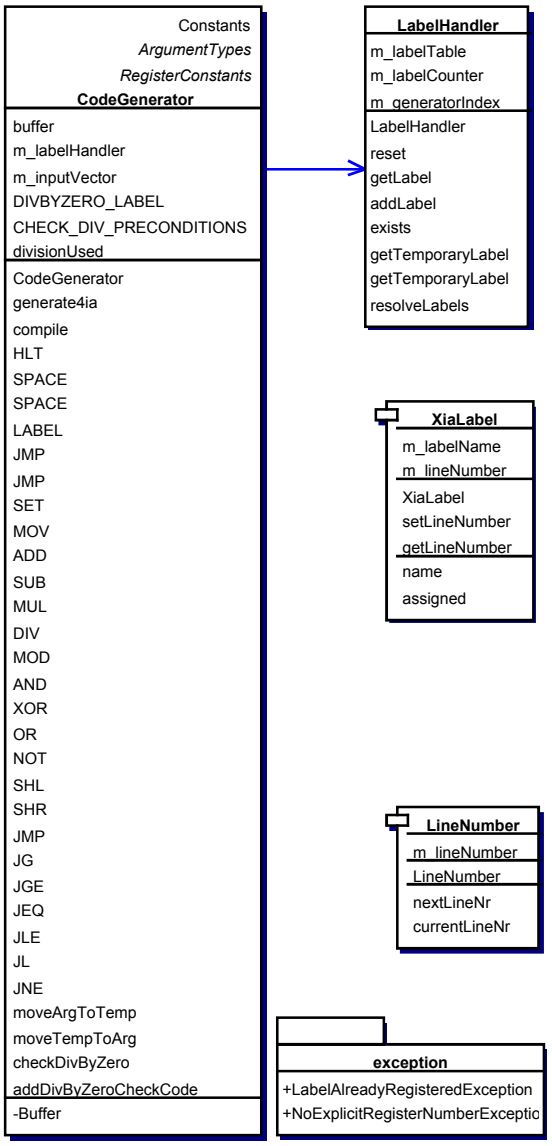

Abbildung 13: Klassendiagramm des Package `compiler`

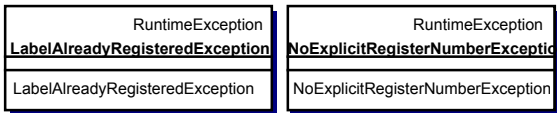

Abbildung 14: Klassendiagramm des Package `compiler.exception`

Schnittstelle
Die Schnittstelle, welche von der Main Klasse Xto4 verwendet wird zum Kompilieren, wird eingangsseitig definiert durch den Konstruktor der Klasse *CodeGenerator*, der als Argumente einen *LineVector* erwartet mit den darin enthaltenden *Line* Objekten sowie einem String, der dem generierten 4IA-Code am Anfang eingefügt wird und Header-Informationen beinhaltet.
Abschliessend zur Schnittstellendefinition gehört die Methode `generate4ia()` dazu, welche den eigentlichen Kompilationsprozess startet. Dies mit Hilfe der Daten, welche der Instanz *CodeGenerator* zur Zeit der Konstruktion mitgegeben wurden, sprich mit Aufruf des Konstruktors.

Kompilationsprozess
Mit dem Starten des Kompilationsprozesses durch Aufrufen der entsprechenden Methode `generate4ia()` der Klasse *CodeGenerator* werden im Wesentlichen folgende Schritte für jede Instruktion, respektive für jedes *Line* Objekt, ausgeführt:

- Bestimmen des Opcodes
- Gegebenenfalls die Register, in welchen die Werte zur Berechnung stehen, in temporäre Register kopieren
- Gegebenenfalls Preconditions überprüfen
- Generieren des 4IA-Codes
- Gegebenenfalls Werte von temporären Registern zurück in die benutzten Register kopieren

Beispiel
Nachfolgend ein kleines Beispiel, welches exemplarisch aufzeigen soll, wie eine Methode, in diesem Beispiel die Methode `SUB(`*Argument* `arg1,` *Argument* `arg2)`, in etwa aussieht, welche 4IA-Code generiert. Die Zeilennummern sollen spätere Erklärungen vereinfachen.

```
 1: private void SUB(Argument arg1, Argument arg2) {
 2:   moveArgToTemp(arg1, REG_TMP_7);
 3:   moveArgToTemp(arg2, REG_TMP_8);
 4:   String SUBLabel[] = m_labelHandler.getTemporaryLabel(3);
 5:   buffer.writeLine(SET_4ia, REG_TMP_1, getRegMinus1());
 6:   buffer.writeLine(SET_4ia, REG_IP, SUBLabel[1]);
 7:   LABEL(SUBLabel[0]);
 8:   buffer.writeLine(ADD_4ia, REG_TMP_7, REG_TMP_1);
 9:   LABEL(SUBLabel[1]);
10:   buffer.writeLine(SET_4ia, REG_FLAG, 0);
11:   buffer.writeLine(ADD_4ia, REG_TMP_8, REG_TMP_1);
```

```
12:   buffer.writeLine(ADD_4ia, REG_IP, REG_FLAG);
13:   buffer.writeLine(SET_4ia, REG_IP, SUBLabel[2]);
14:   buffer.writeLine(SET_4ia, REG_IP, SUBLabel[0]);
15:   LABEL(SUBLabel[2]);
16:   moveTempToArg(arg1, REG_TMP_7);
17: }
```

Erläuterungen zum Code Fragment

Zeile	*Erläuterungen*
1	Einstiegspunkt zur Kompilation eines konkreten Opcodes mit seinen Argumenten.
2-3	Die Übergaberegister, welche durch die Argumente `arg1` (Minuend) und `arg2` (Subtrahend) definiert sind, werden in interne, temporäre Register kopiert, so dass Erstere ihre Werte nach Bedarf beim Verlassen der Methode beibehalten soweit gefordert. `REG_TMP_8` ist der Schlaufenzähler, das heisst die Dekrementierung vom Minuenden wird `REG_TMP_8` mal erfolgen.
4	Erstellen von temporären Labeln, welche innerhalb der Subtraktion verwendet werden.
5	Subtraktionskonstante `-1` in temporäres Register kopieren.
6	Ersten Subtraktionszyklus auslassen.
7	Relative Sprungadresse definieren zum Dekrementieren des Minuenden.
8	Dekrementieren des Minuenden.
9	Relative Sprungadresse definieren um Dekrementierung des Minuenden zu überspringen.
10	Das Carry-Flag der virtuellen Maschine zurücksetzen.
11	Schlaufenzähler dekrementieren.
12	Das Carry-Flag VM-seitig dem Instruction Pointer VM-seitig hinzuaddieren.
13	Kein Überlauf bei der Addition, was bedeutet, dass der Schlaufenzähler bei 0 angelangt ist.
14	Überlauf bei der Addition, was bedeutet, dass der Schlaufenzähler noch nicht bei 0 angelangt ist: Fortfahren mit der Dekrementierung.
15	Relative Sprungadresse zum Beenden der Schlaufe.
16	Resultat kopieren in Register, welches in `arg1` definiert ist.
17	Der 4IA-Code ist generiert.

Abschliessende Erklärungen zu allen Methoden, welche XIA in 4IA-Code kompilieren, finden sich im Kapitel ab Seite 144.

Die Klasse *CodeGenerator* ist demnach dafür verantwortlich, ein *Line* Objekt zu verarbeiten, so dass daraus 4IA-Code entsteht. Zum temporären Zwischenspeichern von Registern sowie intern benutzten Werten, zum Beispiel der Dekrementationskonstante, des Schlaufenzählers und so weiter, werden die Register `r0` bis `r14` benutzt, welche dem Programmierer nicht zugänglich sind. Unter diesen Registern befindet sich auch der Instruction Pointer sowie die Carry-Flags und das Error-Code Register.

Die Klasse CodeGenerator$Buffer [11]
Die Inner Class *Buffer* erfüllt im wesentlichen folgende Aufgaben:

[11] Das Dollarzeichen $ bedeutet, dass nachfolgender Text der Name einer Inner Class, also einer Klasse innerhalb einer Klasse, ist.

- 4IA-Code zu formatieren
- 4IA-Code Fragmente zwischenspeichern
- Einfügen der Headerinformationen zu Beginn des 4IA-Codes

Dabei stellt die Klasse *Buffer* Methoden zur Verfügung, welche es erlauben, eine 4IA Zeile sowohl mit Konstanten, Registern oder Kommentar in verschiedener Anzahl dem Buffer hinzuzufügen. Die Methode nennt sich immer `writeLine(...)`, wobei die Übergabeparameter und deren Anzahl variieren können. Sämtliche Methoden bis auf `writeLine(`*String*`, `*String*`, `*String*`, `*String*`)` sind Delegatoren auf eben genannte. Diese implementierende Methode ist zuständig zur Formatierung der Source-Code Linie. Zum fortlaufenden Numerieren der Zeilen bedient sich *Buffer* den Methoden `getNextLineNr()` der Klasse *LineNumber*, welche von *Buffer* instantiiert wird.

Label Behandlung
Labels werden zur Generierungszeit gehandhabt in Form von relativen Sprungadressen, welche zum Schluss der 4IA Code-Generierung aufgelöst werden in absolute, für die virtuelle Maschine benutzbare Zeilennummern. Zuständig für die Verwaltung der Labels ist die Klasse `LabelHandler`, welche instantiiert wird von *CodeGenerator*. Im wesentlichen deckt *LabelHandler* folgende Funktionalitäten ab:

- Ein neues Label instantiieren
- Ein Label zur Liste der verwendeten Labels hinzufügen
- Temporäre, eindeutige Labels instantiieren
- Labels auflösen in absolute Sprungadressen

Ein Label repräsentiert sich in Form einer Instanz von *XIALabel*, welches ”getter” und ”setter” Methoden zur Verfügung stellt, die Informationen handeln betreffend dem relativen Label Namen sowie der absoluten Liniennummer, welche vorerst nicht definiert ist.

Die Methode `resolveLabels(`*String*`)` der Klasse *LabelHandler* ist zuständig für die Auflösung von relativen in absolute Sprungadressen. Verwendet vom *CodeGenerator*, wird sie unmittelbar vor dem Beenden des Kompilationsprozesses aufgerufen. Übergeben wird der Methode der 4IA-Code, welcher relative Labels enthält.
`resolveLabels(...)` löst diese Labels nun auf und ersetzt sie durch die entsprechenden absoluten Zeilennummern. Zum Schluss wird dieser bearbeitete String dem *CodeGenerator* retourniert.

Mehrfachverwendung von 4IA-Code Fragmenten
Im Zusammenhang mit der Precondition, dass ein Divisor nicht null sein darf bei Divisionen, wird das entsprechende 4IA-Code Fragment, welches diese Überprüfung vornimmt, mehrfach verwendet. Dies wurde wie folgt realisiert: Indem die Division vor deren Aufruf in definierte Register einerseits die Zeilennummer speichert, zu welcher zurück gesprungen werden soll, falls der Divisor nicht null ist, andererseits den Wert des Divisors selbst in ein Register ablegt, welches ausgelesen und ausgewertet werden kann von oben genanntem 4IA-Code Fragment. Falls eine Division mit null versucht wurde, wird ins Error-Code Register der entsprechende Wert gespeichert und danach das Programm beendet.

Rückgabewert
Nach der Generierung von 4IA-Code für jede Zeile des *LineVectors* werden noch folgende drei Aufgaben erledigt, bevor der Code String der Main Klasse zurückgegeben wird:

- Gegebenenfalls das Code Fragment hinzufügen, welches überprüft ob ein Divisor null ist
- Den generierten Code aus dem Buffer holen
- Die Labels mit relativen Sprungadressen auflösen

Die Entscheidung ist trivial, ob das Code Fragment benötigt wird, welches überprüft, ob ein Divisor null ist, . Zur Instanziierungszeit von *CodeGenerator* wird das Flag `divisionUsed` auf FALSE gesetzt. Jedesmal wenn eine Divisionsinstruktion kompiliert wird, wird diesem Flag der Wert true zugeordnet.

Die überschriebene Methode `toString()` von *Buffer* liefert den formatierten und korrekt durchnummerierten 4IA-Code String. Dieser enthält nur noch relative Sprungadressen.

Die Auflösung der relativen Labels in absolute Sprungadressen übernimmt wie bereits vorgängig erwähnt die Methode `resolveLabels(`*String*`)` der Member-Instanz [12] *LabelHandler* der Klasse *CodeGenerator*. Nach deren Auflösung gibt der *CodeGenerator* den 4IA-Code zurück in Form eines Strings. Auf der virtuellen Maschine ist dieser bereits ausführbar und kann betreffend Mehrfach Jumps optimiert werden.

Damit endet der Zuständigkeitsbereich des Package Compiler.

3.3.5 Optimizer

Das Package `optimizer` mit den dazugehörigen Klassen ist dafür verantwortlich, lauffähigen 4IA-Code in folgenden Punkten zu optimieren:

- Sprünge auf andere Sprünge eliminieren
- Werte von temporären Registern, welche in weiteren temporären Registern sind, nur einmal zu speichern
- Mehrfach benutzte Routinen nur einmal dem 4IA-Code hinzufügen

Die Funktionalitäten des Package Optimizer wurden nicht implementiert, da dieses weitläufige Gebiet des Compilerbau nicht viel mit dem eigentlichen Kernthema der Diplomarbeit „Parasitic Computing" zu tun hat. Alleine die Entwicklung von Algorithmen, welche Code Optimieren bezüglich Performance und Dateigrösse, könnten als eigentsändige Diplomarbeit angegangen werden.

Wir entschieden uns, das Package der Code Optimierung zu berücksichtigen, um damit aufzuzeigen, welche Schritte notwendig wären, um einen leistungsfähigen Cross-Compiler zu realisieren. Der Cross-Compiler `xto4` wurde dementsprechend so modular aufgebaut, dass Erweiterungen problemlos machbar wären, was dieses Package mit seiner minimalen, definierten Schnittstelle aufzeigen soll.

[12] Member Instanz bedeutet, dass eine Instanz einer Klasse nur innerhalb einer weiteren Klasse zugänglich ist.

Klassendiagramm
Das Klassendiagramm soll grundsätzlich die Schnittstelle aufzeigen, welche nach aussen besteht.

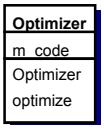

Abbildung 15: Klassendiagramm des Package `optimizer`

Implementierungsansatz
Bei einer effektiven Implementation wäre es denkbar, den String, welcher dem *Optimizer* mitgegeben wird im Konstruktor, erneut zu scannen und in eine strukturierte Form zu bringen, welche es bedeutend leichter ermöglichen würde, logische Abfolgen und Muster zu erkennen. Der übergebene String ist bereits lauffähiger, aber ineffizienter 4IA-Code, welcher optimiert werden soll.

3.3.6 Global

Das Package Global enthält Konstanten und packageübergreifend benutzte Methoden.

Klassendiagramm
Folgendes Klassendiagramm soll einen Überblick verschaffen, welche Konstanten und Methoden dieses Package zur Verfügung stellt.

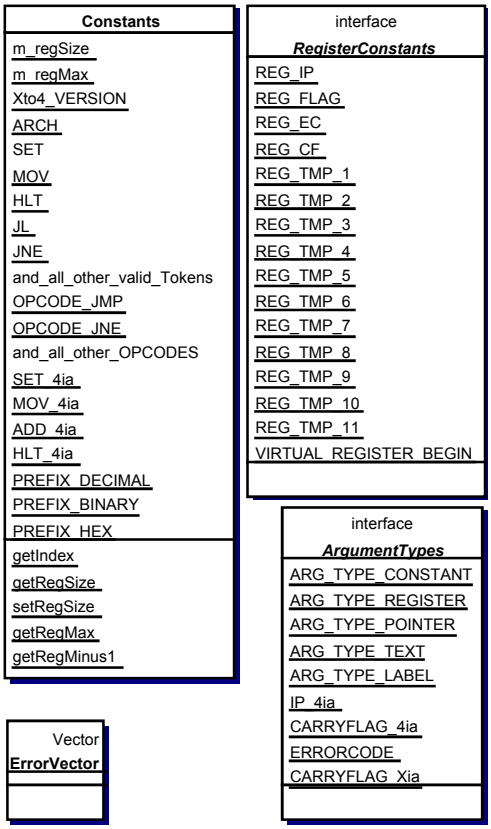

Abbildung 16: Klassendiagramm des Package `global`

Das Interface ArgumentTypes
Das Interface *ArgumentTypes* stellt Konstanten zur Verfügung, welche alle gültigen Argument Typen deklarieren und XIA-weit benutzt werden. Die Wahl, dazu ein Interface zu verwenden, wird in Abschnitt 3.2 auf Seite 126 erklärt.

Das Interface RegisterConstants
Das Interface *RegisterConstants* stellt Konstanten zur Verfügung, welche die Namen sämtlicher Hilfs- und der virtuellen Register definieren. Die Wahl, dazu ein Interface zu verwenden, wird in Abschnitt 3.2 auf Seite 126 erklärt.

Die Klasse ErrorVector
Die Klasse *ErrorVector* dient dazu, einen Vector zu deklarieren, welcher Exceptions gespeichert hat. *ErrorVector* erweitert die Klasse *java.util.Vector*, implementiert selbst keine weiteren Methoden und überschreibt auch keine. Der Vorteil von *ErrorVector* gegenüber der Benutzung von *java.util.Vector* ist der, dass sich die Schnittstelle verkleinert, ist doch auch *LineVector* abgeleitet von *java.util.Vector* und muss nicht speziell

behandelt werden. Schöner ist diese implementierte Variante gegenüber der Alternative, *Object* Übergabe- und Return-Values zu benutzen.

Die Klasse Constants
Die Aufgabe der Klasse *Constants* ist es, Integer- sowie String-Konstanten zur Deklaration der verschiedenen Opcodes zur Verfügung zu stellen. Im weiteren bietet *Constants* eine Reihe `public static getters()`, welche Informationen liefern betreffend maximal adressierbaren Registern, Dekrementierungskonstanten oder Stringsauflösung in Opcode-Indexes.

Constants ist eine Klasse und kein Interface, weil sie nebst den Konstanten, welche es erlauben würden, *Constants* als Interface zu handhaben, zusätzlich eine Reihe von Methoden zur Verfügung stellt, die zwingendermassen in einer Klasse untergebracht werden müssen.

3.3.7 JavaDoc Erläuterungen

Die automatisch generierte Java Dokumentation[13] des Cross-Compiler `xto4` ist sowohl online verfügbar unter `http://parasit.org/code/Xto4_API/` sowie auch auf der CD-ROM, welche zum Umfang dieser Dimplomarbeit gehört, im Unterverzeichnis `code/Xto4_API`.

Die Java Dokumentation beschreibt die Anwendung, die Zuständigkeiten und die Funktionsweise der Packages, Klassen, Konstanten und Methoden. Vom Detaillierungsgrad her knüpft die JavaDoc direkt an an dieses Dokument und bietet im wesentlichen folgende zusätzlichen Informationen:

- Informationen betreffend des Zusammenspiels und der Abhängigkeiten der einzelnen Klassen und Methoden, welche zum kompilieren von 4IA-Code benötigt werden. Ersichtlich anhand der Übergabe-Argumente der entsprechenden Methoden.
- Welche temporären, internen Register jeweils zur Übersetzung einer XIA in 4IA-Code Zeilen benutzt werden (siehe Klasse *CodeGenerator* aus dem Package `org.parasit.xia.compiler`).
- Welche Fehler auftreten können innerhalb der vier Kompilationsschritte (Ersichtlich in den Sub-Packages der vier Haupt-Packages compiler, scanner, optimizer und resolver).
- Schritte, in welche die Aufgaben der Methoden unterteilt wurden.
- Wo welche Konstanten definiert sind, welche benutzt werden. Sämtliche Konfigurationen wie beispielsweise die gültigen Namen der Opcodes oder der Registerprefix sind in Klassen definiert, welche diese Informationen verwalten und allen Klassen zur Verfügung stellen (siehe Klassen, welche den Namen *Constants* beinhalten).

[13] JavaDoc Website: `http://java.sun.com/j2se/javadoc/`

4 Kompilationsalgorithmen xto4

4.1 Allgemeines

In nachfolgenden Unterkapiteln werden sämtliche Instruktionen erläutert und der Input, das heisst die XIA-Code Zeile, mit dem daraus resultierenden Output werden aufgezeigt. Diejenigen Zeilen, welche nebst der generierten 4IA Instruktion mit der Funktion an sich eine übergeordnete Funktionalität haben wie beispielsweise Schlaufenzähler, werden zusätzlich kommentiert, so dass der generierte Code vollständig und klar nachvollzogen werden kann. Abbildung 17 zeigt prinzipiell auf, welche Schritte ausgeführt werden zum kompilieren einer XIA Instruktion in 4IA-Code.

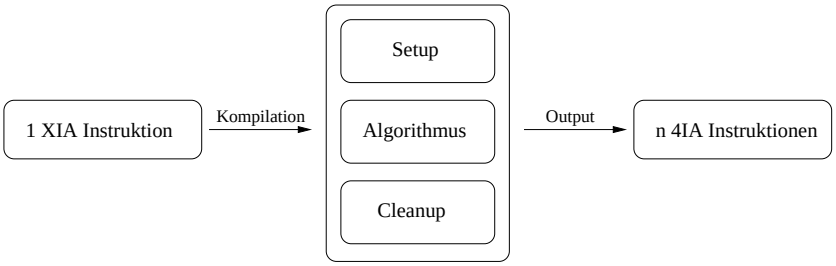

Abbildung 17: Kompilationsschritte

4.1.1 Einleitende Code-Fragmente

Bei genauer Betrachtung des 4IA-Codes in nachfolgenden Erläuterungen fällt auf, dass gewisse 4IA-Code Fragmente sich stehts ähnlich sehen. So zum Beispiel zu Beginn von etlichen 4IA-Code Fragmenten, wo auf den ersten Blick der Eindruck entsteht, Werte würden sinnlos von Registern in andere Register kopiert, welche diese Werte wiederum in dritte Register kopieren würden. Beispielsweise

```
0001: MOV r10,  r15        ; Wert von r15 in tmp. Register r10 kopieren
0002: SET r10,  20
0003: MOV r15,  r10        ; Wert von temp. Reg. r10 zurueck kopieren in r15
0004: MOV r10,  r15        ; Wert von r15 in tmp. Register r10 kopieren
0005: SET r11,  4          ; Konstante in temp. Reg. r11 laden
```

hätte durch folgende Zeilen vereichfacht implementiert werden können:

```
0004: SET r10,  20         ; Wert von r15 in tmp. Register r10 laden
0005: SET r11,  4          ; Konstante in temp. Reg. r11 laden
```

Der Grund für zunächst zu häufiges Mehrfachkopieren von Registerwerten in andere ist der, dass die Java Methoden, welche diesen Code generiert haben, sich weiterer Hilfsmethoden bedienen, welche es erlauben, gewisse Abstraktionen vorzunehmen. Die Zeile respektive die damit definierte Java Methode

```
moveArgToTemp(argX, REG_TMP_X);
```

nimmt beispielsweise die Aufgabe war, die Übergaberegister, welche durch die Argumente `arg1` und `arg2` definiert sind, in interne, temporäre Register zu kopieren, so dass Erstere ihre Werte nach Bedarf beim Verlassen der Methode beibehalten soweit gefordert. Im weiteren nimmt sie folgende Aufgabe wahr: Die aufrufende Methode, also diejenige Methode, welche die Methode `moveArgToTemp(...)` benutzt, kann abstrahieren, ob ihr Argument ein Pointer, ein direktes Register respektive ein virtuelles Register oder eine Konstante ist, welche unterschiedlich behandelt werden müssten, was die Wertzuweisung betrifft [14] in das entsprechende temporäre Register. Nach dem Aufruf kann das temporäre Register in jedem Fall als ein direkt adressierbares Register benutzt werden. Dabei werden aber überflüssige `MOV` und `SET` Instruktionen generiert. Die XIA Instruktion

```
SET  *r15   20
```

beispielsweise liefert folgenden Output:

```
0001: MOV r10,  *r15     ; Wert von *r15 in tmp. Register r10 kopieren
0002: SET r10,  20
0003: MOV *r15, r10      ; Wert von temp. Reg. r10 zurueck kopieren in *r15
```

Deutlich zu erkennen in diesem Beispiel ist hier, dass die Java Methode SET nicht zu wissen braucht, ob es sich beim Argument 1, also *r15, um eine Konstante oder einen Pointer handelt. Nach Aufruf von `moveArgToTemp(...)`, welcher Zeile 1 generiert, kann davon ausgegangen werden, dass nun die Konstante in Argument 2, also 20, direkt in das temporäre Register r10 geladen werden kann, ersichtlich in Zeile 2. Zum Schluss wird dann das temporäre Register r10, ersichtlich in Zeile 3, wiederum kopiert ins eigentliche Zielregister respektive in unserem Beispiel nach Pointer *r15. Auch hier wiederum braucht sich die SET Methode Java Seitig nicht darum zu kümmern, von welchem Typ Argument 1 ist. Die bereits erläuterte Java Methode sieht wie folgt aus:

```
private void SET(Argument arg1, Argument arg2) {
  moveArgToTemp(arg1, REG_TMP_7);
  buffer.writeLine(SET_4ia, REG_TMP_7, arg2.getValue());
  moveTempToArg(arg1, REG_TMP_7);
}
```

Ein Nachteil der Methode `moveArgToTemp(...)` ist wie bereits erwähnt der, dass sie wie bereits erläutert überflüssige Mehrfachkopieren von Registerwerten in andere verursacht, welche dann von einem anderen Package, dem `optimizer`, wieder rückgängig gemacht werden müssen.
Nachfolgend der Java Source-Code dieser Methode zur besseren Verständlichkeit.

```
private void moveArgToTemp(Argument arg, String register) {
  if(arg.getType() == ARG_TYPE_CONSTANT) {
    buffer.writeLine(SET_4ia, register, arg.getValue(),
                     "Konstante in temp. Reg. "+register+" laden");
  }else if(arg.getType() == ARG_TYPE_REGISTER) {
    buffer.writeLine(MOV_4ia, register, arg.getValue(),
                     "Wert von "+arg.getValue()+" in tmp. Register "+
                     register+" kopieren");
  } else if(arg.getType() == ARG_TYPE_POINTER) {
```

[14] Eine Konstante wird einem Register durch die `SET` Instruktion zugewiesen, hingegen wird ein Registerwert beim Kopieren in ein anderes Register durch eine `MOV` Anweisung vorgenommen

```
    buffer.writeLine(MOV_4ia, register, arg.getValue(),
                     "Wert von "+arg.getValue()+" in temp. Reg. "+
                     register+" kopieren");
  }
}
```

4.1.2 Instruktionen zur Resultat Speicherung

Nach Beendigung der Generierung von 4IA-Code der entsprechenden Java Methoden tauchen häufig Zeilen respektive Methodenaufrufe auf in folgender Form, bevor die Java Methode verlassen wird:

```
moveTempToArg(argX, REG_TMP_X);
```

Diese Java Methode hat die Aufgabe, ein Resultat, welches noch in einem internen temporären Register gespeichert ist, zurück zu kopieren ins Register, in welchem das zu erwartende Resultat gespeichert werden soll, also ins Register, welches durch Argument `argX` definiert ist.
Solange bei keinem Opcode XIA-seitig als erstes Argument eine Konstante sein darf, solange kann ja davon ausgegangen werden, dass das Zielargument immer ein Register oder ein Pointer ist. Dementsprechend muss nicht unterschieden werden, sondern es kann vielmehr direkt das temporöre Register mittels `MOV` Instruktion ins Zielregister, welches durch Argument 1 definiert ist, kopiert werden. Die Java Methode sieht demnach wie folgt aus:

```
private void moveTempToArg(Argument arg, String register) {
  buffer.writeLine(MOV_4ia, arg.getValue(), register,
                   "Wert von temp. Reg. "+register+
                   " zurueck kopieren in "+arg.getValue());
}
```

4.1.3 Zeilennummerierung im Java-Code

Zur einfacheren und verständlicheren Erläuterung von Java Code sind nachfolgende Zeilen nummeriert. Dies soll aber keine Verwirrung stiften bezüglich falschen Annahmen, wir hätten einen eigenen Java Parser mit einer speziellen virtuellen Maschine entwickelt...

4.1.4 FEO

Die 4IA-Code Beispiele, welche in nachfolgenden Unterkapiteln aufgelistet und erläutert sind, können nicht eins zu eins übernommen und ausgeführt werden von der virtuellen Maschine, sind also FEO[15]. Dies, weil aus bewusst der Opcode `ARCH` sowie die 4IA Instruktion `HLT` weggelassen wurden.
Im weiteren fehlten teilweise Kontexte, welche zum Ausführen zusätzlich benötigt würden, sowie sämtliche Headers, welche statistische Aufschlüsse darlegen würden.

4.1.5 Bitweise Operatoren

Da die Registerbreite variabel ist, wird in den folgenden Erklärungen darauf verzichtet, korrekte binäre Werte zu verwenden in Algorithmen, welche die Funktionsweise und insbesondere das Zusammenspielen von Makros erläutern. Die Zeile

[15] FEO (*engl.*): „for exposition only"; zu Demonstrationszwecken, nicht direkt übertragbar auf die Realität.

```
((a XOR 1) AND (b XOR 1)) XOR 1
```

beispielsweise müsste korrekterweise wie folgt aussehen, falls mit einer Registerbreite von 8 Bit gearbeitet wird (Beispiel entnommen von OR auf Seite 154):

```
((a XOR 11111111) AND (b XOR 11111111)) XOR 11111111
```

Die 1 repräsentiert symbolisch immer die Anzahl aufeinanderfolgenden 1 gemäss Registerbreite, mit welcher gefahren wird. Diese Anzahl eins, also die Registerbreite, wird zu Beginn in der Scanning Phase von der Klasse *LineValidator* bestimmt und als Konstante allen andern Packages mittels Konstante, festgehalten in Klasse *Constants*, zur Verfügung gestellt.

4.2 Instruktionen

Nachfolgend aufgelistet sämtliche Instruktionen mit Erläuterungen betreffend den verwendeten Kompilationsalgorithmen, Beispielen von In- und Output, welcher eingespeist respektive generiert wird sowie Kommentaren zu generierten 4IA-Code Zeilen, welche nebst ihrer direkten Funktionen übergeordnete Aufbaben haben wie Schlaufenzähler und so weiter.

4.2.1 SET

Algorithmus
Die Instruktion SET wird eins zu eins weitergegeben.

Input

```
SET r3   20        ; Dem Register r3 den Wert 20 zuweisen
```

Output

```
0001: MOV r10,  r15        ; Wert von r15 in tmp. Register r10 kopieren
0002: SET r10,  20
0003: MOV r15,  r10        ; Wert von temp. Reg. r10 zurueck kopieren in r15
```

Erläuterungen

Zeile	*Funktion*
2	Abbilden der SET Instruktion auf den 4IA Code

4.2.2 MOV

Algorithmus
Die Instruktion MOV wird eins zu eins weitergegeben.

Input

```
MOV r15  r16       ; Register r16 in r15 kopieren
```

Output

```
0001: MOV r15,  r16        ; Reg. Wert r16 in Reg. r15 kopieren
```

Erläuterungen

Zeile *Funktion*

1 Abbilden der MOV Instruktion auf den 4IA Code

4.2.3 HLT

Algorithmus

Die Instruktion HLT wird eins zu eins weitergegeben.

Input

```
HLT               ; Stoppt die virtuelle Maschine
```

Output

```
0001: HLT                 ; Stoppt die VM
```

Erläuterungen

Zeile *Funktion*

1 Abbilden der HLT Instruktion auf den 4IA Code

4.2.4 SPACE

Algorithmus

Die Instruktion SPACE fügt eine Leerzeile ohne Zeilennummer ein in den 4IA-Code, damit dieser besser leserlich wird. Die Kommentarzeile hat keine Zeilennummer, weil diese ja von der virtuellen Maschine ignoriert werden soll.

Input

```
SPACE Kommentar Zeile im 4IA Code...
```

Output

```
                          ; Kommentar Zeile im 4IA Code...
```

Erläuterungen

—

4.2.5 ADD

Algorithmus
Die Instruktion ADD wird eins zu eins weitergegeben.

Input
```
ADD r15  12        ; 12 zu r15 addieren
```

Output
```
0001: MOV r10,  r15        ; Wert von r15 in tmp. Register r10 kopieren
0002: SET r11,  12         ; Konstante in temp. Reg. r11 laden
0003: ADD r10,  r11        ; Addition
0004: MOV r15,  r10        ; Wert von temp. Reg. r10 zurueck kopieren in r15
```

Erläuterungen
Zeile *Funktion*
3 Abbilden der ADD Instruktion auf den 4IA Code

4.2.6 SUB

Algorithmus
Das Argument *arg1* wird *arg2* mal dekrementiert.

```
int i = arg2;
while(i-- > 0) --arg1;
```

Input
```
SUB r15  5         ; 5 von r15 subtrahieren
```

Output
```
0001: MOV r10,  r15        ; Wert von r15 in tmp. Register r10 kopieren
0002: SET r11,  5          ; Konstante in temp. Reg. r11 laden
0003: SET r4,   255        ; Integerkonstante fuer -1 Subtraktion
0004: SET ip,   5          ; Erste Subtraktion ueberspringen
0005: ADD r10,  r4         ; Register r10 dekrementieren
0006: SET fl,   0          ; Flag zuruecksetzen auf 0
0007: ADD r11,  r4         ; Zaehler r11 dekrementieren
0008: ADD ip,   fl         ; Schlaufe verlassen falls Zaehler r11 auf 0
0009: SET ip,   10
0010: SET ip,   4
0011: MOV r15,  r10        ; Wert von temp. Reg. r10 zurueck kopieren in r15
```

Erläuterungen
Zeile *Funktion*
2 Schlaufenzähler initialisieren
4 Erste Subtraktion überspringen, weil die letzte Dekrementierung erfolgt, wenn der Schlaufenzähler bereits be 0 angekommen ist.
5 Wiederholtes dekrementieren des Minuenden
6 Flag wird zurückgesetzt, damit nach Dekrementierung vom Schlaufenzähler festgestellt werden kann, ob Schlaufe verlassen werden kann.
9 Schlaufe verlassen. Das Resultat steht temporär in r10
10 Schlaufe wiederholen, da der Schlaufenzähler noch keinen Überlauf verursachte.

4.2.7 MUL

Algorithmus
Das Argument *arg1* wird *arg2* mal zu sich selbst addiert.

```
int i = arg2;
while(i-- > 0) {
  arg1 += arg1;
}
```

Input

```
MUL r15  5        ; r15 mit 5 multiplizieren
```

Output

```
0001: MOV r10,  r15       ; Wert von r15 in tmp. Register r10 kopieren
0002: SET r11,  5         ; Konstante in temp. Reg. r11 laden
                          ; Beginn der Multiplikation
0003: SET r5,   0         ; Resultat Register initialisieren
0004: SET r4,   255       ; Integerkonstante fuer -1 Subtraktion
0005: SET ip,   6         ;   Addition wiederholen
0006: ADD r5,   r10       ; Addition
0007: SET fl,   0         ; Flag zuruecksetzen auf 0
0008: ADD r11,  r4        ; Zaehler dekrementieren, Flag setzen falls Unterlauf
0009: ADD ip,   fl        ; Schlaufe verlassen falls Flag gesetzt
0010: SET ip,   11        ;   Schlaufe verlassen
0011: SET ip,   5         ;   Addition wiederholen
0012: MOV r15,  r5        ; Wert von temp. Reg. r5 zurueck kopieren in r15
```

Erläuterungen

Zeile	*Funktion*
2	Schlaufenzähler initialisieren
6	Addition von *arg1* mit sich selbst.
9-11	Abbruchbedingung der Schlaufe.

4.2.8 DIV

Algorithmus
Der Divisor *arg2* wird solange den Dividenden *arg1* abgezogen, bis dieser kleiner 0 ist. Bei jedem Schlaufendurchgang wird das Resultat um eins erhöht. Nach Verlassen der Schlaufe wird der Rest berechnet, indem das Vorzeichen des Wertes, welcher zuviel subtrahiert wurde vom Dividenden, getauscht, und dieser Betrag vom Divisor subtrahiert wird.

```
resultat = -1;
while(dividend >= 0) {
  dividend -= divisor;
  resultat++;
}
rest = divisor - (-1 * dividend);
```

Input

```
DIV r15  5         ; r15 mit 5 dividieren
```

Output

```
0001: MOV r10,  r15        ; Wert von r15 in tmp. Register r10 kopieren
0002: SET r11,  5          ; Konstante in temp. Reg. r11 laden
                           ;
                           ; check ob Division mit 0
0003: MOV r5,   r11        ; Register, welches auf 0 getestet werden soll
0004: SET r4,   5          ; Zeilennummer in r4 fuer Ruecksprung von zero-check
0005: SET ip,   31         ; Unbedingter Sprung zu Zeile 31
0006: SET r4,   255        ; Subtraktionskonstante -1
0007: SET r5,   1          ; Additionskonstante 1
0008: SET r7,   0          ; Resultat Register mit 0 initialisieren
0009: MOV r6,   r11        ; Innerer Schlaufenzaehler initialisieren (r6)
0010: SET ip,   14         ; Unbedingter Sprung zu Zeile 14
0011: SET fl,   0          ; Flag loeschen
0012: ADD r10,  r4         ; Dekrement Dividend
0013: ADD ip,   fl         ; Bei Dividend > 0 weiter
0014: SET ip,   21         ; Bei Dividend == 0 ende
0015: SET fl,   0          ; Flag loeschen
0016: ADD r6,   r4         ; Dekrement innerer Schlaufenzaehler
0017: ADD ip,   fl         ; Ueberlauf verarbeiten fuer Auswahl in Zeile 18/19
0018: SET ip,   19         ;   Bei Schlaufenzaehler == 0 ende innere Schlaufe
0019: SET ip,   10         ;   Bei Schlaufenzaehler > 0 weiter
0020: ADD r7,   r5         ; Inkrement Quotient
0021: SET ip,   8          ; Innere Schlaufe neu beginnen
0022: MOV r15,  r7         ; Wert von temp. Reg. r7 zurueck kopieren in r15
0023: MOV r3,   r11        ; Divisor ins Carry-Register kopieren
0024: SET ip,   25         ; Unbedingter Sprung zu Zeile 25
0025: ADD r3,   r4         ; Kopie von Divisor dekrementieren
0026: SET fl,   0          ; Flag loeschen
0027: ADD r6,   r4         ; Schlaufenzaehler dekrementieren
0028: ADD ip,   fl         ; Ueberlaufcheck
0029: SET ip,   30         ; Unbedingter Sprung zu Zeile 30
0030: SET ip,   24         ; Unbedingter Sprung zu Zeile 24
0031: ADD r3,   r4         ; Carry-Reg. dekrementieren -> Divisionsrest
                           ; ENDE DER DIVISION
                           ;
                           ; Routine, die ueberprueft, ob Division
                           ; mit 0 vorliegt. Wert aus Register r5
                           ; wird auf 0 geprueft. Falls dies der Fall
                           ; ist, dann Sprung zu Zeile 37  fuer Programmstopp
                           ; und Error-Flag=1 setzen.
0032: SET r6,   255        ; Register r6 zum addieren von 255 um r5
0033: SET fl,   0          ; Ueberlauf register fl auf 0 setzen
0034: ADD r5,   r6         ; Register r5 addieren um 255
0035: ADD ip,   fl         ; Checken ob Addition Ueberlauf verursachte
0036: SET ip,   37         ; Register r5 war 0: Abbruch
0037: MOV ip,   r4         ; Weiterfahren mit der Division
0038: SET r2,   1          ; Error-Code setzen: Division mit 0
0039: HLT                  ; Stoppt die VM
```

Erläuterungen

Zeile	*Funktion*
1	Dividend temporär in r10 speichern.
2	Divisor temporär in r11 speichern.
3	Register initialisieren, welches auf 0 überprüft werden soll (”division by zero” check).
5	”Division by zero” Check Subroutine ausführen.
6	Rücksprungadresse, falls Divisor nicht gleich 0 ist und Division durchgeführt werden kann.
8	Der innere Schlaufenzähler dient der Dekrementierung vom Dividenden mit dem Divisor.
11-19	Die innere Schlaufe, welche den Dividenden mit dem Divisor subtrahiert.
14	Abbruchbedingung der äusseren Schlaufe, welche das Resultat inkrementiert.
21	Sprung zum äussere Schlaufe wiederholen.
22	Resultat zurückkopieren in `arg1`.
23-31	Berechnung des Rests, indem der Divisor so oft dekrementiert wird, bis der Schlaufenzähler r6 0 ist.
20	Inkrementieren des Quotienten.

4.2.9 MOD

Algorithmus

Das Argument *arg1* wird dividiert mit dem Argument *arg2* und der Rest, der von der Division übrig bleibt, wird zurückkopiert ins Zielregister definiert durch Argument *arg1*.
Zum Dividieren wird das Makro `DIV` (siehe 4.2.8) verwendet, der Divisionsrest jedoch zurück kopiert ins Resultat Register von *arg1*.

Input

```
MOD r15  5         ; Restwert von r15 mod 5 berechnen
```

Output

Ist der gleiche wie der der Division, zusätzlich wird aber vor Beendigung des Makros noch folgende Zeile angehängt, damit der Restwert zurück ins Resultat Register kopiert wird:

```
...
                          ; Divisionsrest in r15 kopieren
0034: MOV r15,  r3        ; Wert von temp. Reg. r3 zurueck kopieren in r15
...
```

Erläuterungen

Zeile	*Funktion*
34	Der Divisionsrest, welcher das `DIV` Makro ins Register `r3` schreibt, wird ins Resultat Register r15 kopiert.

4.2.10 AND

Algorithmus

Bei jedem Bit-Paar der beiden Argumente wird geschaut ob beide 1 sind. Falls ja, wird dies dem Resultat addiert.

```
int i = Registerbreite;
int resultat = 0;
while(i-- > 0) {
  shiftLeft(resultat);

  if(highestBit(arg1) == true) op1 = 1;
  else op1 = 0;

  if(highestBit(arg2) == true) op2 = 1;
  else op2 = 0;

  if(op1 == op2 == 1) resultat += 1;

  shiftLeft(op1);
  shiftLeft(op2);
}
```

Input

```
AND r15  0b01101   ; Bitweises and mit r15 und 0b01101
```

Output

```
0001: MOV r9,   r15        ; Wert von r15 in tmp. Register r9 kopieren
0002: SET r10,  0b01101    ; Konstante in temp. Reg. r10 laden
0003: SET r11,  128        ; Konstante fuer Fixaddition (hoechstes Bit auf 1)
0004: SET r5,   1          ; Konstante fuer Fixaddition (tiefstes Bit auf 1)
0005: SET r4,   255        ; Konstante fuer Fixaddition (maximale Ganzzahl)
0006: SET r12,  7          ; Schlaufenzaehler (Registerbreite - 1)
0007: ADD r6,   r6         ; Resultat nach links schieben

0008: SET fl,   0          ; Flag zuruecksetzen
0009: MOV r7,   r9         ; Aktuelles hoechstwertiges Bit von Op 1 ermitteln
0010: ADD r7,   r11
0011: ADD fl,   fl         ; Flag mit 2 multiplizieren, optimierter bedingter
0012: ADD ip,   fl         ; Sprung
0013: SET r7,   0
0014: ADD ip,   r5
0015: MOV r7,   r11

0016: SET fl,   0
0017: MOV r8,   r10        ; Aktuelles hoechstw. Bit von Op 2 ermitteln
0018: ADD r8,   r11
0019: ADD fl,   fl         ; s. oben
0020: ADD ip,   fl
0021: SET r8,   0
0022: ADD ip,   r5
0023: MOV r8,   r11

0024: SET fl,   0
0025: ADD r7,   r8         ; Addition der ermittelten hoechstwertigen Bits
0026: ADD r6,   fl         ; Flag enth. Resultat "AND" der obigen zwei Bits
0027: ADD r9,   r9         ; Operand 1 nach links schieben
```

```
0028: ADD r10,  r10        ; Operand 2 nach links schieben
0029: SET fl,   0          ; Flag loeschen
0030: ADD r12,  r4         ; Schlaufenzaehler dekrementieren
0031: ADD ip,   fl         ; Bei Ueberlauf Schlaufe wiederholen:
0032: SET ip,   33         ; Srung ans Ende (Zeile 33      )
0033: SET ip,   6          ; Sprung zu Zeile 6
0034: MOV r15,  r6         ; Wert von temp. Reg. r6 zurueck kopieren in r15
```

Erläuterungen

Zeile	*Funktion*
1-6	Temporäre Register initialisieren.
8-15	Aktuelles hoechstwertiges Bit von Argument 1 ermitteln.
16-23	Aktuelles hoechstwertiges Bit von Argument 2 ermitteln.
24-26	Addition von 1 zum Resultat, falls beide Bits 1 sind.
29-33	Schlaufe wiederholen bis alle Bit ausgewertet sind.

4.2.11 OR

Algorithmus

Dieser Algorithmus benutzt die Makros `AND` und `XOR` folgendermassen:

```
((a XOR 1) AND (b XOR 1)) XOR 1
```

Input

```
OR  r15  0b01101   ; Bitweises XOR mit r15 und 0b01101
```

Output

Eine Zusammenstellung von generietem 4IA-Code der Makros `AND` und `XOR`, wie sie im Abschnitt Algorithmen erläutert wurde.

Erläuterungen

Siehe Makro `AND` (Seite 152) und `XOR` (Seite 154) für Code Details.

4.2.12 XOR

Algorithmus

Der Algorithmus der Operation `XOR` entspricht dem der Operation `AND`, mit dem Unterschied, dass dem Resultat nur eine 1 hinzugefügt wird, falls nur *eine* der beiden Bit aus `arg1` und `arg2` gesetzt ist.

```
int i = Registerbreite;
int resultat = 0;
while(i-- > 0) {
  shiftLeft(resultat);

  if(highestBit(arg1) == true) op1 = 1;
  else op1 = 0;
```

```
  if(highestBit(arg2) == true) op2 = 1;
  else op2 = 0;

  if((op1 + op2) == 1) resultat += 1;

  shiftLeft(op1);
  shiftLeft(op2);
}
```

Input

```
XOR r15  0b01101   ; Bitweises XOR mit r15 und 0b01101
```

Output

```
0001: MOV r9,   r15        ; Wert von r15 in tmp. Register r9 kopieren
0002: SET r10,  0b01101    ; Konstante in temp. Reg. r10 laden
0003: SET r11,  128        ; Konstante fuer Fixaddition (hoechstes Bit auf 1)
0004: SET r5,   1          ; Konstante fuer Fixaddition (tiefstes Bit auf 1)
0005: SET r4,   255        ; Konstante fuer Fixaddition (maximale Ganzzahl)
0006: SET r12,  7          ; Schlaufenzaehler (Registerbreite - 1)
0007: ADD r6,   r6         ; Resultat nach links schieben

0008: SET fl,   0          ; Flag zuruecksetzen
0009: MOV r7,   r9         ; Aktuelles hoechstwertiges Bit von Op 1 ermitteln
0010: ADD r7,   r11
0011: ADD fl,   fl         ; Flag mit 2 multiplizieren, optimierter bedingter
0012: ADD ip,   fl         ; Sprung
0013: SET r7,   0
0014: ADD ip,   r5
0015: MOV r7,   r11

0016: SET fl,   0          ; Flag zuerueckzusetzen
0017: MOV r8,   r10        ; Aktuelles hoechstwertiges Bit von Op 2 ermitteln
0018: ADD r8,   r11
0019: ADD fl,   fl         ; s. oben
0020: ADD ip,   fl
0021: SET r8,   0
0022: ADD ip,   r5
0023: MOV r8,   r11

0024: ADD r7,   r8         ; Addition der zwei ermittelten hoechstw. Bits
0025: SET fl,   0          ; Flag loeschen
0026: ADD r7,   r11        ; Hoechstwertiges Bit nach Flag transferieren
0027: ADD r6,   fl         ; Flag enth. Resultat "XOR" der obigen zwei Bits
0028: ADD r9,   r9         ; Operand 1 nach links schieben
0029: ADD r10,  r10        ; Operand 2 nach links schieben
0030: SET fl,   0          ; Flag loeschen
0031: ADD r12,  r4         ; Schlaufenzaehler dekrementieren
0032: ADD ip,   fl         ; Bei Ueberlauf Schlaufe wiederholen:
0033: SET ip,   34         ; Srung ans Ende (Zeile 34      )
0034: SET ip,   6          ; Sprung zu Zeile 6
0035: MOV r15,  r6         ; Wert von temp. Reg. r6 zurueck kopieren in r15
```

Erläuterungen

Zeile	*Funktion*
1-6	Temporäre Register initialisieren.
8-15	Aktuelles höchstwertiges Bit von Argument 1 ermitteln.
16-23	Aktuelles höchstwertiges Bit von Argument 2 ermitteln.
24-27	Addition von 1 zum Resultat, falls nur 1 Register gesetzt ist.
30-34	Schlaufe wiederholen bis alle Bit ausgewertet sind.

4.2.13 NOT

Algorithmus
Dieser Algorithmus benutzt das Makro XOR wie folgt:

```
a XOR 1
```

Input

```
NOT r15  0b01101   ; Bitweises NOT mit r15 und 0b01101
```

Output
Eine Zusammenstellung von generietem 4IA-Code der Makros AND und XOR, wie sie im Abschnitt Algorithmen erläutert wurde.

Erläuterungen
Siehe Makro AND (Seite 152) und XOR (Seite 154) für Code Details.

4.2.14 SHL

Algorithmus
Der Wert in *arg1* wird *arg2* mal mit sich selbst addiert.

```
for(int i=1; i<=arg2; ++i) {
  arg1 += arg1;
}
```

Input

```
SHL r15  3         ; Shift left r15 um 3 Bit
```

Output

```
0001: MOV r10,  r15        ; Wert von r15 in tmp. Register r10 kopieren
0002: SET r11,  3          ; Konstante in temp. Reg. r11 laden
0003: SET r3,   0          ; Carry-Flag XIA-seitig zuruecksetzen
0004: SET r4,   255        ; Subtraktionskonstante -1
0005: MOV r5,   r11        ; Anzahl shifts (3) initialisieren

0006: SET ip,   14         ; erstes SHL ueberspringen -> falls 0 shifts gewuenscht
0007: SET fl,   0          ; Ueberlauf flag auf 0 setzen
0008: ADD r10,  r10        ; shift left r10 um 1 bit

0009: ADD ip,   r3         ; Checken ob cf bereits gesetzt
0010: SET ip,   11         ;   cf noch nicht gesetzt: neu checken
0011: SET ip,   14         ;   cf bereits gesetzt: weiterfahren ohne check
0012: ADD ip,   fl         ; Checken ob shift left overflow verursachte
0013: SET ip,   14         ;   kein overflow erfolgt -> weiterfahren
0014: SET r3,   1          ; SHL verursachte overflow. cf setzen

0015: SET fl,   0          ; Unterlauf flag auf 0 setzen
0016: ADD r5,   r4         ; Schlaufenzahler dekrementieren
0017: ADD ip,   fl         ; Checken ob Schlaufe beendet
0018: SET ip,   19         ;   SHL verlassen
0019: SET ip,   6          ;   SHL wiederholen
0020: MOV r15,  r10        ; Wert von temp. Reg. r10 zurueck kopieren in r15
```

Erläuterungen

Zeile	*Funktion*
1-5	Temporäre Register initialisieren.
7	Schlaufen Sprung Adresse zum wiederholen der Addition.
8	Shift Left um 1 Bit des Arguments *arg1*.
9-14	Mechanismus, welcher es erlauben würde, gezielt auf Überläufe zu reagieren. Wird nicht verwendet. Wird in der aktuellen Version nicht weiter verwendet.
15-19	Schlaufenwiederholungs-Konditionen prüfen und ausführen.

4.2.15 SHR

Algorithmus

Der Wert in *arg1* wird *arg2* mal mit 2 dividiert, was jeweils einem Shift Right um 1 Bit entspricht.

```
for(int i=1; i<=arg2;++i) {
  arg1 = arg1 / 2;
}
```

Input

```
SHR r15  3         ; Shift right r15 um 3 Bit
```

Output

```
0001: MOV r12,  r15        ; Wert von r15 in tmp. Register r12 kopieren
0002: SET r13,  3          ; Konstante in temp. Reg. r13 laden
0003: SET r4,   255        ; Subtraktionskonstante -1
0004: MOV r8,   r13        ; Anzahl shifts (3) initialisieren
0005: SET ip,   36         ; erstes SHR ueberspringen -> falls 0 shifts gewuenscht

...
... Makro DIV(r12, r13)    ; shift right r12 um 1 bit
...

0037: SET fl,   0          ; Unterlauf flag auf 0 setzen
0038: ADD r8,   r4         ; Schlaufenzahler dekrementieren
0039: ADD ip,   fl         ; Checken ob Schlaufe beendet
0040: SET ip,   41         ;   SHL verlassen
0041: SET ip,   5          ;   SHL wiederholen
0042: MOV r15,  r12        ; Wert von temp. Reg. r12 zurueck kopieren in r15
```

Erläuterungen

Zeile	*Funktion*
1-4	Temporäre Register initialisieren.
5-36	Division von `arg1` mit 2, welche einem Shift right um ein Bit entspricht.
37-41	Schlaufenwiederholungs-Konditionen prüfen und ausführen. Anzahl gewünschter Verschiebungen nach rechts entspricht Schlaufendurchgängen.

4.2.16 JMP

Algorithmus
Der unbedingte Sprung `JMP` ist speziell, weil die der einzige Sprung an eine Adresse ist, welcher unbedingt erfolgt. **Input**

```
JMP L1            ; Unbedingter Sprung zu Label L1
```

Output

```
0001: SET ip,   L1         ; Unbedingter Sprung zu Zeile L1
```

Erläuterungen

Zeile *Funktion*

1 Abbilden der HLT Instruktion auf den 4IA Code.

4.2.17 JG

Algorithmus
Beide Variablen werden inkrementiert, bis ein Überlauf stattgefunden hat. Falls nur die erste Variable einen Überlauf verursachte, wird gesprungen.

```
while(!overflow(arg1) && !overflow(arg2)) {
  ++arg1;
  ++arg2;
}

if(overflow(arg1) && !overflow(arg2)){
  return true;
} else {
  return false;
}
```

Input

```
JG  r15 2 L1          ; Unbedingter Sprung zu Label L1
```

Output

```
0001: MOV r10,  r15        ; Wert von r15 in tmp. Register r10 kopieren
0002: SET r11,  2          ; Konstante in temp. Reg. r11 laden
0003: SET r5,   1          ; Register r5 mit Wert 1 zum inkrementieren

0004: SET fl,   0          ; Flag zuruecksetzen auf 0
0005: ADD r10,  r5         ; Register r10 inkrementieren
0006: ADD ip,   fl         ; Jump zu Verzweigung ob r10 Ueberlauf oder nicht
0007: SET ip,   8          ;   Kein Ueberlauf: weiterfahren mit r11 inkrementieren
0008: SET ip,   12         ;   Ueberlauf: checken ob r11 auch Ueberlauf...

0009: ADD r11,  r5         ; Register r11 inkrementieren
0010: ADD ip,   fl         ; Jump zu Verzweigung ob r11 Ueberlauf oder nicht
0011: SET ip,   3          ;   r11 auch kein Ueberlauf: naechster Durchlauf
```

```
0012: SET ip,   18         ;   Abbruch: Wert in Register 2 grosser als in r15

0013: SET fl,   0          ; Flag zuruecksetzen auf 0
0014: ADD r11,  r5         ; Register r11 inkrementieren
0015: ADD ip,   fl         ; Jump zu Verzweigung ob r11 Ueberlauf oder nicht
0016: SET ip,   17         ;   Abbruch: Wert in r15 ist groesser als der in 2
0017: SET ip,   18         ;   Abbruch: Wert in 2 grosser als in r15
0018: SET ip,   L1
```

Erläuterungen

Zeile	*Funktion*
1-3	Initialisierungen.
4-8	Inkrementieren und auf Überlauf prüfen von *arg1*.
9-12	Inkrementieren und auf Überlauf prüfen von *arg2*.
13-18	Überprüfen, ob die Bedingung eingetroffen ist und dementsprechend den Instruction Pointer setzen.

4.2.18 JGE

Algorithmus

Beide Variablen werden inkrementiert, bis ein Überlauf stattgefunden hat. Falls nur die erste oder beide Variable einen Überlauf verursachten, wird gesprungen.

```
while(!overflow(arg1) && !overflow(arg2)) {
  ++arg1;
  ++arg2;
}

if((overflow(arg1) && !overflow(arg2)) ||
   !overflow(arg2)){
  return true;
} else {
  return false;
}
```

Input

```
JGE r15 2 L1          ; Unbedingter Sprung zu Label L1
```

Output

```
0001: MOV r10,  r15        ; Wert von r15 in tmp. Register r10 kopieren
0002: SET r11,  2          ; Konstante in temp. Reg. r11 laden
0003: SET r5,   1          ; Register r5 mit Wert 1 zum inkrementieren

0004: SET fl,   0          ; Flag zuruecksetzen auf 0
0005: ADD r10,  r5         ; Register r10 inkrementieren
0006: ADD ip,   fl         ; Jump zu Verzweigung ob r10 Ueberlauf oder nicht
0007: SET ip,   8          ;   Kein Ueberlauf: weiterfahren mit r11 inkrementieren
0008: SET ip,   12         ;   Ueberlauf: checken ob r11 auch Ueberlauf...

0009: ADD r11,  r5         ; Register r11 inkrementieren
0010: ADD ip,   fl         ; Jump zu Verzweigung ob r11 Ueberlauf oder nicht
0011: SET ip,   3          ;   r11 auch kein Ueberlauf: naechster Durchlauf
```

```
0012: SET ip,   18        ;   Abbruch: Wert in Register 2 grosser als in r15

0013: SET fl,   0         ; Flag zuruecksetzen auf 0
0014: ADD r11,  r5        ; Register r11 inkrementieren
0015: ADD ip,   fl        ; Jump zu Verzweigung ob r11 Ueberlauf oder nicht
0016: SET ip,   17        ;   Abbruch: Wert in r15 ist groesser als der in 2
0017: SET ip,   17        ;   Abbruch: Wert in 2 grosser als in r15
0018: SET ip,   L1
```

Erläuterungen

Zeile	*Funktion*
1-3	Initialisierungen.
4-8	Inkrementieren und auf Überlauf prüfen von *arg1*.
9-12	Inkrementieren und auf Überlauf prüfen von *arg2*.
13-18	Überprüfen, ob die Bedingung eingetroffen ist und dementsprechend den Instruction Pointer setzen.

4.2.19 JEQ

Algorithmus

Beide Variablen werden inkrementiert, bis ein Überlauf stattgefunden hat. Falls beide Variablen einen Überlauf verursachten, wird gesprungen.

```
while(!overflow(arg1) && !overflow(arg2)) {
  ++arg1;
  ++arg2;
}

if(overflow(arg1) && overflow(arg2)){
  return true;
} else {
  return false;
}
```

Input

```
JEQ r15 2 L1          ; Unbedingter Sprung zu Label L1
```

Output

```
0001: MOV r10,  r15       ; Wert von r15 in tmp. Register r10 kopieren
0002: SET r11,  2         ; Konstante in temp. Reg. r11 laden
0003: SET r5,   1         ; Register r5 mit Wert 1 zum inkrementieren

0004: SET fl,   0         ; Flag zuruecksetzen auf 0
0005: ADD r10,  r5        ; Register r10 inkrementieren
0006: ADD ip,   fl        ; Jump zu Verzweigung ob r10 Ueberlauf oder nicht
0007: SET ip,   8         ;   Kein Ueberlauf: weiterfahren mit r11 inkrementieren
0008: SET ip,   12        ;   Ueberlauf: checken ob r11 auch Ueberlauf...

0009: ADD r11,  r5        ; Register r11 inkrementieren
0010: ADD ip,   fl        ; Jump zu Verzweigung ob r11 Ueberlauf oder nicht
0011: SET ip,   3         ;   r11 auch kein Ueberlauf: naechster Durchlauf
0012: SET ip,   18        ;   Abbruch: Wert in Register 2 grosser als in r15
```

```
0013: SET fl,   0          ; Flag zuruecksetzen auf 0
0014: ADD r11,  r5         ; Register r11 inkrementieren
0015: ADD ip,   fl         ; Jump zu Verzweigung ob r11 Ueberlauf oder nicht
0016: SET ip,   18         ;   Abbruch: Wert in r15 ist groesser als der in 2
0017: SET ip,   17         ;   Abbruch: Wert in 2 grosser als in r15
0018: SET ip,   L1
```

Erläuterungen

Zeile	*Funktion*
1-3	Initialisierungen.
4-8	Inkrementieren und auf Überlauf prüfen von *arg1*.
9-12	Inkrementieren und auf Überlauf prüfen von *arg2*.
13-18	Überprüfen, ob die Bedingung eingetroffen ist und dementsprechend den Instruction Pointer setzen.

4.2.20 JLE

Algorithmus

Beide Variablen werden inkrementiert, bis ein Überlauf stattgefunden hat. Falls nur die zweite oder beide Variablen einen Überlauf verursachten, wird gesprungen.

```
while(!overflow(arg1) && !overflow(arg2)) {
  ++arg1;
  ++arg2;
}

if((overflow(arg1) && overflow(arg2)) ||
   !(overflow(arg1)){
  return true;
} else {
  return false;
}
```

Input

```
JLE r15 2 L1          ; Unbedingter Sprung zu Label L1
```

Output

```
0001: SET r10,  2          ; Konstante in temp. Reg. r10 laden
0002: MOV r11,  r15        ; Wert von r15 in tmp. Register r11 kopieren
0003: SET r5,   1          ; Register r5 mit Wert 1 zum inkrementieren

0004: SET fl,   0          ; Flag zuruecksetzen auf 0
0005: ADD r10,  r5         ; Register r10 inkrementieren
0006: ADD ip,   fl         ; Jump zu Verzweigung ob r10 Ueberlauf oder nicht
0007: SET ip,   8          ;   Kein Ueberlauf: weiterfahren mit r11 inkrementieren
0008: SET ip,   12         ;   Ueberlauf: checken ob r11 auch Ueberlauf...

0009: ADD r11,  r5         ; Register r11 inkrementieren
0010: ADD ip,   fl         ; Jump zu Verzweigung ob r11 Ueberlauf oder nicht
0011: SET ip,   3          ;   r11 auch kein Ueberlauf: naechster Durchlauf
0012: SET ip,   18         ;   Abbruch: Wert in Register r15 grosser als in 2
```

```
0013: SET fl,   0          ; Flag zuruecksetzen auf 0
0014: ADD r11,  r5         ; Register r11 inkrementieren
0015: ADD ip,   fl         ; Jump zu Verzweigung ob r11 Ueberlauf oder nicht
0016: SET ip,   17         ;   Abbruch: Wert in 2 ist groesser als der in r15
0017: SET ip,   17         ;   Abbruch: Wert in r15 grosser als in 2
0018: SET ip,   L1
```

Erläuterungen

Zeile	*Funktion*
1-3	Initialisierungen.
4-8	Inkrementieren und auf Überlauf prüfen von *arg1*.
9-12	Inkrementieren und auf Überlauf prüfen von *arg2*.
13-18	Überprüfen, ob die Bedingung eingetroffen ist und dementsprechend den Instruction Pointer setzen.

4.2.21 JL

Algorithmus

Beide Variablen werden inkrementiert, bis ein Überlauf stattgefunden hat. Falls nur die zweite Variablen einen Überlauf verursachte, wird gesprungen.

```
while(!overflow(arg1) && !overflow(arg2)) {
  ++arg1;
  ++arg2;
}

if(!(overflow(arg1)) {
  return true;
} else {
  return false;
}
```

Input

```
JL  r15 2 L1          ; Unbedingter Sprung zu Label L1
```

Output

```
0001: SET r10,  2          ; Konstante in temp. Reg. r10 laden
0002: MOV r11,  r15        ; Wert von r15 in tmp. Register r11 kopieren
0003: SET r5,   1          ; Register r5 mit Wert 1 zum inkrementieren

0004: SET fl,   0          ; Flag zuruecksetzen auf 0
0005: ADD r10,  r5         ; Register r10 inkrementieren
0006: ADD ip,   fl         ; Jump zu Verzweigung ob r10 Ueberlauf oder nicht
0007: SET ip,   8          ;   Kein Ueberlauf: weiterfahren mit r11 inkrementieren
0008: SET ip,   12         ;   Ueberlauf: checken ob r11 auch Ueberlauf...

0009: ADD r11,  r5         ; Register r11 inkrementieren
0010: ADD ip,   fl         ; Jump zu Verzweigung ob r11 Ueberlauf oder nicht
0011: SET ip,   3          ;   r11 auch kein Ueberlauf: naechster Durchlauf
0012: SET ip,   18         ;   Abbruch: Wert in Register r15 grosser als in 2

0013: SET fl,   0          ; Flag zuruecksetzen auf 0
0014: ADD r11,  r5         ; Register r11 inkrementieren
```

```
0015: ADD ip,   fl         ; Jump zu Verzweigung ob r11 Ueberlauf oder nicht
0016: SET ip,   17         ;   Abbruch: Wert in 2 ist groesser als der in r15
0017: SET ip,   18         ;   Abbruch: Wert in r15 grosser als in 2
0018: SET ip,   L1
```

Erläuterungen

Zeile	*Funktion*
1-3	Initialisierungen.
4-8	Inkrementieren und auf Überlauf prüfen von *arg1*.
9-12	Inkrementieren und auf Überlauf prüfen von *arg2*.
13-18	Überprüfen, ob die Bedingung eingetroffen ist und dementsprechend den Instruction Pointer setzen.

4.2.22 JNE

Algorithmus

Beide Variablen werden inkrementiert, bis ein Überlauf stattgefunden hat. Falls nur eine Variable einen Überlauf verursachte, wird gesprungen.

```
while(!overflow(arg1) && !overflow(arg2)) {
  ++arg1;
  ++arg2;
}

if((!overflow(arg1) && overflow(arg2)) ||
   (!overflow(arg1) && overflow(arg2))) {
  return true;
} else {
  return false;
}
```

Input

```
JNE r15 2 L1          ; Unbedingter Sprung zu Label L1
```

Output

```
0001: MOV r10,  r15        ; Wert von r15 in tmp. Register r10 kopieren
0002: SET r11,  2          ; Konstante in temp. Reg. r11 laden
0003: SET r5,   1          ; Register r5 mit Wert 1 zum inkrementieren

0004: SET fl,   0          ; Flag zuruecksetzen auf 0
0005: ADD r10,  r5         ; Register r10 inkrementieren
0006: ADD ip,   fl         ; Jump zu Verzweigung ob r10 Ueberlauf oder nicht
0007: SET ip,   8          ;   Kein Ueberlauf: weiterfahren mit r11 inkrementieren
0008: SET ip,   12         ;   Ueberlauf: checken ob r11 auch Ueberlauf...

0009: ADD r11,  r5         ; Register r11 inkrementieren
0010: ADD ip,   fl         ; Jump zu Verzweigung ob r11 Ueberlauf oder nicht
0011: SET ip,   3          ;   r11 auch kein Ueberlauf: naechster Durchlauf
0012: SET ip,   17         ;   Abbruch: Wert 2 grosser als r15

0013: SET fl,   0          ; Flag zuruecksetzen auf 0
0014: ADD r11,  r5         ; Register r11 inkrementieren
```

```
0015: ADD ip,   fl         ; Jump zu Verzweigung ob r11 Ueberlauf oder nicht
0016: SET ip,   17         ;   Abbruch: Wert in r15 ist groesser als der in 2
0017: SET ip,   18         ;   Abbruch: Wert in 2 grosser als in r15
0018: SET ip,   L1
```

Erläuterungen

Zeile	*Funktion*
1-3	Initialisierungen.
4-8	Inkrementieren und auf Überlauf prüfen von *arg1*.
9-12	Inkrementieren und auf Überlauf prüfen von *arg2*.
13-18	Überprüfen, ob die Bedingung eingetroffen ist und dementsprechend den Instruction Pointer setzen.

4.2.23 ARCH

Algorithmus

Die Instruktion ARCH wird nach Hinzufügen von einem zweiten durch den Kompiler bestimmten Argument weitergegeben in den 4IA-Code.

Input

```
ARCH  279
```

Output

```
ARCH 279 0
```

Erläuterungen

Zeile	*Funktion*
–	Das zweite Argument ist die durch den Kompiler definierte Anzahl Register, welche benutzt wird im XIA Programm.

PARASITIC COMPUTING
Auswertungen & Systemtest

Version 1.0

Jürg Reusser
Luzian Scherrer

5. Januar 2003

Zusammenfassung

Dieses Dokument zeigt die Resultate und Erkenntnisse, welche sich aus der im Rahmen der Diplomarbeit „Parasitic Computing" entwickelten Software ergeben haben. Im Folgenden werden verschiedene Parameterisierungen und Ausführungsarten der Software auf ihre Effizienz getestet und verglichen. Den zweiten Teil dieses Dokumentes bildet ein Bericht über das Vorgehen und die Resultate des Systemtests, mittels welchem die Korrektheit der Programme sichergestellt wurde.

Inhaltsverzeichnis

1 Auswertungen

1.1 Parallelisierung der Addition

Die der virtuellen Maschine zugrundeliegende Operation ist die Ganzzahladdition auf Bitebene. In ihrer atomaren Form geschieht diese, wie gezeigt wurde, sequentiell für jede Bitstelle der zu addierenden Zahlen. Es ist aber auch denkbar, in einem Sendeimpuls[1] die Lösungen von breiteren als 1-Bit Additionen zu prüfen und somit eine Parallelisierung der Grundoperation zu erreichen.

1.1.1 Anzahl benötigter Netzwerkpakete

Die Anzahl betötigter Netzwerkpakete der Addition entspricht

$$AnzahlPakete = \frac{RB}{x} \times 2^{(x+1)}$$

wobei *RB* die Registerbreite der virtuellen Maschine in Bits und *x* die Parallelisierungskonstante ist. Letztere definiert, wieviele Bitstellen pro Sendeimpuls berechnet werden sollen. Für die Parallelisierungskonstante gibt es zwei Einschränkungen. Erstens muss sie ein ganzer Teiler der Registerbreite sein, denn nur so lässt sich das ganze Register gleichmässig aufteilen, zweitens darf sie nicht höher als 4 sein. Der nächstmögliche Wert nach 4 wäre 8, dies geht aber bereits nicht mehr, da sich – die Netzwerkpakete sind 16-Bit breit – Operatoren und Checksummen überschneiden würden.

Somit sind, von einer 8-Bit Registermaschine ausgehend, nur die ersten drei in Tabelle 1 gezeigten Kombinationen möglich. Die restlichen Werte dienen er Anschaulichkeit.

RB	*x*	Anzahl Pakete
8	1	32
8	2	32
8	4	64
8	8	512
8	16	65’536
8	32	2’147’483’648

Tabelle 1: Anzahl Pakete abhängig von der Parallelisierung

1.1.2 Laufzeit und Prozessorzyklen

Die folgende Abbildung 1 zeigt die unterschiedlichen Laufzeitresultate desselben Programmes, ausgeführt in den drei verschiedenen Parallelisierungsmodi. Das Testprogramm ist eine einfache Multiplikation, wie sie in den 4IA-Codebeispielen zu finden ist. Die abgebildeten Ergebnisse sind die aus 15 sukzessiven Durchläufen ermittelten Durchschnittswerte im parasitären Modus. Als Hardware für diesen Test diente eine Sun Ultra Sparc 1 mit einer Taktrate von 167 MHz.

[1] Nach dem beschriebenen Prinzip der Kandidatlösungsprüfung benötigt die Berechnung eines Teilresultates mehrere Netzwerkpakete. Diese Ansammlung an gesendeten Paketen pro Teilresultat bezeichnen wir als Sendeimpuls.

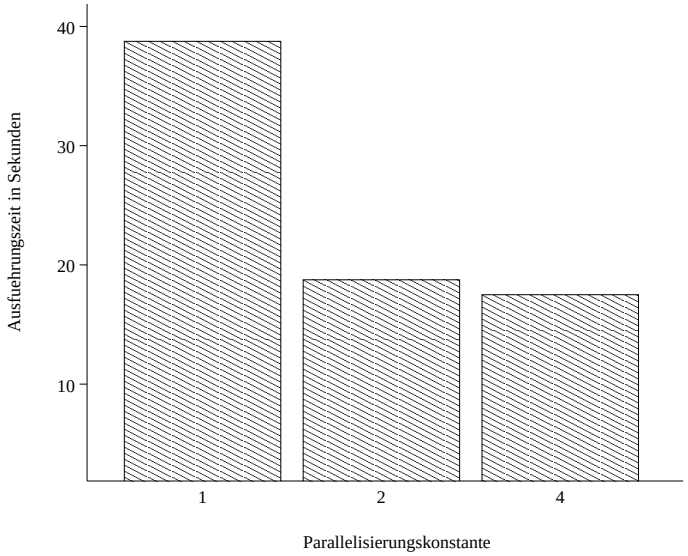

Abbildung 1: Unterschiedliche Laufzeiten je nach Parallelisierungsart

Es zeigt sich, dass die höchstmögliche Parallelisierung das beste Laufzeitverhalten ergibt. Demgegenüber muss jedoch auch der Aufwand an Prozessorzyklen der ausführenden Maschine betrachtet werden. Es entspricht dem eigentlichen Sinn des parasitären Rechenwerkes, diesen Wert so klein wie möglich zu halten. Die Abbildung 2 zeigt die benötigten Prozessorzyklen pro Parallelisierungsmodus für das gleiche Testprogramm, ermittelt jeweils wieder als Durchnittswerte von 15 Durchläufen.

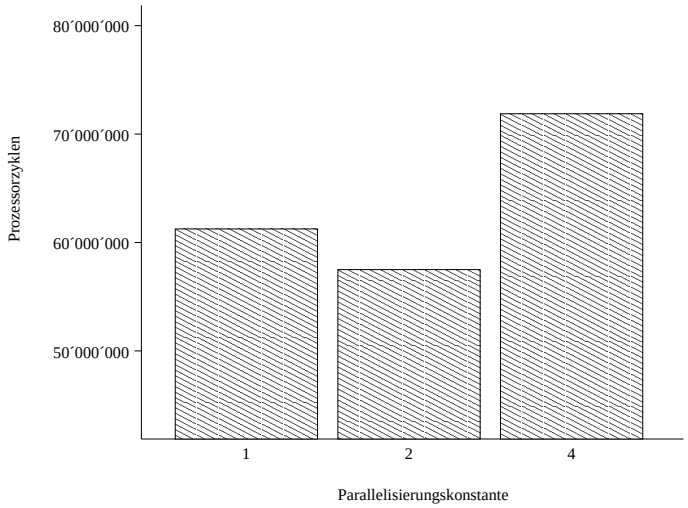

Abbildung 2: Benötigte Prozessorzyklen je nach Parallelisierungsart

Aus den beiden Grafiken wird ersichtlich, dass die Variante mit einer Parallelisierungskonstante von 2 die optimalsten Resultate liefert. Dies entspricht auch den Erwartungen

aus obengenannter Formel, denn $x = 2$ ist der Optimalfall aus den Anforderungen:

- Geringe Anzahl an ICMP Paketen
  Die Werte $x = 1$ und $x = 2$ benötigen beide die gleiche, kleinstmögliche Anzahl an ICMP Paketen. Das Bilden dieser Pakete ist betreffend der benötigten Prozessorzyklen eindeutig die „teuerste" Operation.

- Bestmögliche Parallelisierung
  Eine höhere Parallelisierung hat zur Folge, dass weniger oft auf die Timeouts eines Sendeimpulses gewartet werden muss. Deshalb gilt – wie aus Abbildung 1 ersichtlich – der Grundsatz: je höher die Parallelisierung, desto kleiner die Ausführungszeit.

Der Defaultwert für die Parallelisierungskonstante ist somit bei Applikationsstart immer 2.

1.2 Gegenüberstellende Messungen

In den nachfolgenden Abschnitten sind zwei gegenüberstellende Messungen und deren Resultate als Verhältnisse aufgezeigt. Es ist zu beachten, dass es sich bei den aufgeführten Werten um statistische Ergebnisse von Grössen handelt, welche in der Praxis variieren können. Das Hauptziel dieser Messungen ist es, den ungefähren Rahmen der Grössenordnungen abzustecken.

1.2.1 Reelle vs. virtuelle Prozessorzyklen

Jede Iteration der in [RS02d] gezeigten Schlaufe der virtuellen Maschine entspricht zwei virtuellen Prozessorzyklen. Ein Zyklus wird dabei für die jeweils ausgeführte Instruktion, ein weiterer für die Inkrementierung des Instruktionszeiger-Registers gezählt.

Das Verhältnis aus den benötigen Prozessorzyklen der virtuellen und denjenigen der diese ausführenden Maschine kann nicht generell angegeben werden, da es erstens, wie vorgängig erläutert, von der Parallelisierungskonstante und zweitens vom aktuell berechneten Problem, sprich dessen Instruktionen, abhängt. Um jedoch eine ungefähre Grössenordnung als Anhaltspunkt zu nennen, sei hier auf das der Distribution beiliegende Programm `bubblesort.xia` verwiesen, welches im Simulationsmodus mit einem ungefähren Verhältnnis von

1 : 7’000

virtuellen zu reellen Prozessorzyklen ausgeführt werden kann. Dieses Resultat hat sich auf einem Pentium III mit einer Taktrate von 1 GHz und dem Betriebssystem Linux 2.4.7 ergeben. Im parasitären Modus mit gleichem Programm und gleicher Konfiguration beläuft sich das Verhältnis auf ungefähr

1 : 600’000

virtuelle zu reellen Prozessorzyklen (bei einer Ausführungszeit von etwas mehr als 13 Minuten, 8’760’289 gesendeten und 1’095’036 empfangenen ICMP Paketen).

In der selben Umgebung erzielte das ebenfalls in der Distribution enthaltene Programm `fibonacci.xia` ein Verhältnis von

1 : 9’500

virtuellen zu reellen Prozessorzyklen, ebenfalls im Simulationsmodus. Dieses Programm lässt sich im parasitären Modus nicht in sinnvoller Zeit ausgeführen, weshalb hier kein demenstprechender Prozessorzyklen-Vergleich angegeben werden kann.

1.2.2 Häufigkeit falscher Checksummen

Um eine Ahnung davon zu bekommen, wie oft Pakete mit falschen Checksummen in einem durchschnittlich belasteten Netzwerk auftreten, ist für diese Arbeit eine Testmessung vorgenommen worden. Dabei wurde an einem Cisco Catalyst Switch mit 9 angeschlossenen Hosts ein SPAN[2] konfiguriert und daran mit der Software *Etherreal*[3] wärend der Dauer von einer Woche alle Pakete mit falschen Checksummen gefiltert. Dabei wurden von insgesammt 22’912’281 gemessenen Paketen gerade mal 12 mit nicht korrekten Checksummen vom Filter aufgefangen. Dies entspricht einem ungefähren Verhältnis von

1 : 2’000’000

Paketen mit falschen zu Paketen mit korrekten Checksummen. Das Verhältnis schient gering, allerdings benötigt beispielsweise die Ausführung des der Distribution beiliegenden Programmes `fibonacci.xia` zur Berechnung der ersten 20 Fibonacci-Zahlen bereits 201’752’880 ICMP Pakete.

Pakete mit falschen Checksummen können die Berechnung der virtuellen Maschine allerdings nur dann zu Fall bringen, wenn auf einen Sendeimpuls eines (und genau eines) der falschen Pakete als korrekt und alle anderen Pakete (inklusive das einzelne korrekte) als falsch behandelt werden. Es müssten also die Checksummen von exakt zwei Paketen verändert werden, nämliche diejenige des korrekten Paketes als falsch und diejenige eines falschen Paketes als korrekt. Ein solcher Fall ist kaum denkbar (siehe Abschnitt ,,Algorithmus: parasitäre Verteilung” in [RS02e]).

[2] Switched Port Analyzer
[3] siehe `http://www.ethereal.com/`

2 Systemtest

2.1 Generelles Vorgehen

Wärend der gesamten Entwicklungsphase wurde der Fehlerprüfung stets höchste Gewichtung zugemessen, denn die Korrektheit ist bei Softwareentwicklungen das wichtigste und schlussendlich über Erfolg oder Misserfolg entscheidende Kriterium. Da diese Diplomarbeit im Team von zwei Personen entwickelt wurde, konnten wir durch gegenseitiges Audit der jeweils neu erstellten Codeabschnitte potentiell eingeführte Fehler immer kurzerhand eliminieren. Ausserdem haben wir den Testprozess der „Ausführungskorrektheit" weitgehend automatisiert, wie im nachfolgenden Unterkapitel beschrieben wird.

Vor Abschluss der Implementationsphase wurde nochmals ein komplettes, die gesamten Softwarepakete umfassendes Code-Audit durchgeführt, wobei alle erstellten Programme Zeile für Zeile verifiziert wurden.

Mit der endgültigen Fassung der Software ist es uns nicht mehr gelungen – sei es im parasitären- oder im Simulationsmodus – falsche Resultate bei der Programmausführung zu erhalten.

2.2 Das `pshell_test.pl` Utility

Die in der Distribution enthaltenen 4IA und XIA Programmbeispiele umfassen die kompletten Möglichkeiten der beiden Sprachen. Wenn alle diese Programme korrekt ausgeführt werden können, so gilt der Systemtest als bestanden. Um diesen Prozess wärend der Entwicklungphase – nach jeder Änderung der Software musste deren Korrektheit weiterhin sichergestellt werden können – zu automatisieren, haben wir das Utility `pshell_test.pl` geschrieben. Es ist eine einfache, in PERL[4] implementierte State-Event Maschine, welche für jedes enthaltene Programmbeispiel eine `pshell` Sitzung durchführt und die erwarteten Resultate überprüft. Die entsprechenden Ergebnisse des Tests werden dabei auf der standard Ausgabe angezeigt.

Das `pshell_test.pl` Utility ist auf der beiliegenden CD-ROM, beziehungsweise unter `http://parasit.org/code/`, verfügbar.

[4] PERL: „Practical Extraction and Reporting Language", siehe `http://www.perl.com/`.

PARASITIC COMPUTING
Semesterarbeit Ethik

Version 1.1

Luzian Scherrer
Jürg Reusser

4. August 2002

Zusammenfassung

Das Prinzip „Parasitic Computing" wurde erstmals im August 2001 öffentlich erwähnt [BFJB01], als es Wissenschaftlern der Notre Dame Universität im US-Bundesstaat Indiana gelang, fremde Rechenkapazität ohne Wissen und Einwilligung derer Besitzer zur Lösung mathematischer Probleme zu nutzen. In dem vorliegenden Dokument soll nun die auf dieser Grundlagenarbeit aufbauende Diplomarbeit „Parasitic Computing" der Hochschule für Technik und Architektur Bern in ihren ethischen Aspekten genauer betrachtet werden. Wir wollen uns im folgenden Text den potentiellen ethischen Problemen parasitärer Rechenmethoden annehmen, indem wir diese systematisch aufzeigen und mögliche Lösungswege darstellen.

Inhaltsverzeichnis

1 Einleitung

1.1 Was ist ein Parasit?

Das Wort ,,Parasit" findet seinen geschichtlichen Ursprung im antiken Griechenland. Damals trugen gewisse Volksvertreter auf bestimmte Zeit den Status des ,,Parasiten" und waren als solche geheissen, die Auswahl des Getreides, des Brotes und der Speise für das kutlische Opfermahl zu treffen und dieses mit dem Priester einzunehmen. Darin erklärt sich auch gleich die Herkunft des Wortes: Neben-, mit- oder bei- (pará) speisender (sitós). Lateinisch bedeutet ,,parasitus" Tischgenosse oder eben, wie bekannt, Schmarotzer. Ursprünglich trug das Wort, ganz im Gegensatz zu heute, keinen negativen Beigeschmack; die Parasiten im antiken Griechenland waren hochgeachtete, religiöse Beamte.

Parasitentum wurde früher auch als Lebenskunst betrachtet. Lukian von Samosata [1] hielt vor knapp 2000 Jahren fest: ,, Alle anderen Künste sind ohne gewisse Werkzeuge (die mit Kosten angeschafft werden müssen) ihrem Besitzer unnütz; niemand kann ohne Flöte flöten, ohne Violine geigen, oder ohne ein Pferd reiten: einzig die Parasitenkunst ist sich selber so genug und macht es ihrem Meister so bequem. Andere Kunstverwandte arbeiten nicht nur mit Mühe und Schweiss, sondern grösstentheils sogar sitzend oder stehen, und zeigen dadurch, dass sie gleichsam Sclaven ihrer Kunst sind: der Parasit hingegen treibt die seinige auf eben die Art wie Könige Audienz geben, – liegend" (vgl. [Enz01]).

Im Laufe der Jahrhunderte änderte sich diese Ansicht jedoch, und Parasit verkam zum Schimpfwort. War beim antiken ,,Parásitós" noch ganz klar, dass es sich dabei um eine Person aus Fleisch und Blut handelte, driftete man mit der Bezeichnung zunächst weit in die Botanik, beispielsweise zu Misteln, Moosen und Flechten – den heutigen Phytoparasiten, dann in in die niedere Tierwelt zu Läusen, Bandwürmern und Bazillen – heute Zooparasiten, ab.

1.2 Parasitismus

Die Ernährungsbeziehung zwischen dem Parasiten und seinem Wirt trägt die Bezeichnung Parasitismus. Der Parasit kann den Wirt in Einzahl (Solitärparasitismus) oder in Mehrzahl (Gregärparasitismus) befallen. Zu starker Gregärparasitismus (Superparasitismus) schadet dem Parasiten, weil er sich selbst die Lebensgrundlage entzieht. Wenn verschiedene Parasitenarten gleichzeitig in einem Wirt schmarotzen (Multiparasitismus), bleibt meist nur einer am Leben, oder alle gehen zugrunde. Parasiten können auch selbst wieder von Parasiten befallen werden (Hyperparasitismus). Sonderfälle von Parasitismus sind Cleptoparasitismus und Staatsparasitismus; im weiteren Sinne auch Brutparasitismus und Nahrungsparasitismus.

Der Parasitismus stellt gewissermassen eine Alternative zur klassischen Räuber-Beute-Beziehung dar. Er ist eine Symbiose, bei der ein Organismus indirekten Nutzen aus einem anderen zieht. Genauergenommen handelt es sich beim Parasitismus um eine kommensalistische, also eine einseitige Symbiose, bei welcher im Gegensatz zur mutualistischen Variante nur einer der Beteiligten vom anderen profitiert und kein gegenseitiger Austausch besteht. Entscheidendes Merkmal parasitärer Beziehungen ist immer, dass der Parasit seinem Wirt naturgemäss mit Schonung begegnet (oder begenen

[1] griechischer Schriftsteller, geb. in Samosata am Euphrat um 120 n. Chr., gest. nach 180.

muss), da er von diesem entscheidend abhängt, ja in seiner Existenzgrundlage auf ihn angewiesen ist.

1.3 Der Computer als Parasit

Der parasitäre Computer (vgl. [BFJB01]) nun, oder etwas genauer ausgedrückt das parasitäre Rechenwerk (welches nur einen Teil eines Computers im eigentlichen Sinne darstellt) kann am ehesten mit dem Solitärparasitismus verglichen werden. Es besteht aus einem einzelnen Knoten im Netzwerk (Parasit), das beliebige andere Knoten (Wirte) für sich arbeiten lässt. Das von Forschern der amerikanischen Notre Dame University definierte Verfahren (vgl. [Fre02]), welches hierzu benutzt wird, soll nun zum weiteren Verständnis im folgenden Abschnitt seinem Prinzip nach kurz erläutert werden.

1.4 Parasitismus in Computernetzwerken

Die Kommunikation innerhalb von Computernetzwerken wie beispielsweise dem Internet, basiert aus Gründen der Kompatibilität zwischen unzähligen verschiedenen Hardwareherstellern auf wohldefinierten, durch internationale Gremien beglaubigten und veröffentlichten Standards (vgl. [For02]). Bei diesen Standards handelt es sich um sogenannte Kommunikationsprotokolle, welche exakt vorgeben, in welcher Form ein Computer im Netzwerk Nachrichten zu verschicken, und wie er sich beim Empfang von Nachrichten zu verhalten hat. Solche Nachrichten sind Datenpakete, welche einerseits aus den frei definierbaren Nutzdaten, andererseit aus den sich aus eben diesen Protokollstandards ergebenden Kontrolldaten bestehen.

Einen Teil der Kontrolldaten bildet bei den weitverbreitetsten Kommunikationsprotokollen (namentlich IP, UDP, TCP, ICMP – dies sind die dem gesamten Internet zugrundeliegenden Standards [Ste94]) die sogenannte Internet-Checksumme. Die Checksumme dient der Überprüfung von Datenpaketen auf ihre Korrektheit. Der Absender bildet also über die Nutzdaten seiner zu sendenden Nachricht eine nach einem bestimmten Algorithmus berechnete Prüfsumme (Checksumme) und schickt diese zusammen mit der eigentlichen Nachricht an den Empfänger. Dieser ist nun vom Protokollstandard aufgefordert, zuerst die Checksumme nachzurechnen, und erst wenn diese mit den Nutzdaten übereinstimmt, je nach Anwendung mit der Verarbeitung der restlichen Daten weiterzufahren. Stimmt die Checksumme nicht mit den gesendeten Nutzdaten überein, so muss das Datenpacket stillschweigend verworfen werden.

Bei der Checksummenprüfung auf Empfängerseite kommt das Prinzip des „Parasitic Computing" zum tragen, durch welches eine Kandidatlösung eines durch den Checksummenalgorithmus abbildbaren Problemes vom Emfpänger auf Korrektheit überprüft werden lassen kann. Wird das Datenpacket vom Empfänger weiterverarbietet (dies lässt sich Aufgrund dessen weiteren Verhaltens mit den Nutzdaten bestimmen), so war die zu prüfende Kandidatlösung korrekt, andernfalls wird sie vom Empfänger verworfen und war nicht korrekt. Konkret lassen sich mit diesem Prinzip die binären Grundoperationen XOR und AND ausführen, und auf dieser Grundlage widerum sind sämtliche Operationen eines klassischen Computers aufbaubar (siehe auch [Wir95]). Es lässt sich auf diese Weise also ein parasitäres Rechenwerk realisieren, welches in seinem Kern ausschliessich auf Fremdressourcen aufgebaut ist.

Dabei ist von entscheidender Bedeutung, dass die vom Empfänger – also dem Wirt – genutzten Ressourcen von diesem willentlich zur Verfügung gestellt und benutzt werden dürfen, allerdings durch den Parasiten anders als ursprünglich vorgesehen verwendet werden.

1.5 Rentabilität des Parasitic Computing

An dieser Stelle muss noch erwähnt werden, dass das ,,Parasitic Computing" in seiner momentan bekannten Form nicht rentabel einsetzbar ist. Der Aufwand, Datenpackete für die beschriebene Umnutzung zu generieren, zu versenden und die Ergebnisse zu prüfen, übersteigt den daraus zu gewinnenden Nutzen um ein Vielfaches. ,,Parasitic Computing" ist daher in seiner aktuellen Variante nur ein ,,Proof of Concept", womit allerdings nicht festgelegt ist, dass nicht in Zukunft gewinnbringende Anwendungen gefunden werden könnten. Die potentiellen Ausnutzungsmöglichkeiten sind sehr weitreichend und eine Vielzahl an Protokollen, interessant dürften beispielsweise solche kryptographischer Natur sein, könnten auf ähnliche Weise Rechner zu Wirten von Parasitismus machen.

2 Auswirkungen parasitärer Rechnenmethoden

2.1 Potentielle Probleme

In den folgenden Abschnitten sollen die sich aus der parasitären Berechnung mittels Fremdressourcen ergebenden Problematiken genauer aufgezeigt werden.

2.1.1 (Um)nutzung fremder Ressourcen

Wie bereits in Abschnitt 1.4 angetönt, basiert das Parasitic Computing auf Mechanismen, die der Öffentlichkeit vom Wirt zur Verfügung gestellt und so prinzipiell durch jedermann – implizit erlaubt – benutzt werden dürfen. Dies tut der Parasit allerdings nicht zur vorgesehenen Anwendung der Fehlerprüfung der Kommunikation mit dem Wirt, sondern ausschliesslich für seine eigenen, dem Wirt absolut unnützen Zwecke. Der Wirt verliert also den seinen Nutzen an den Ressourcen die er verfügbar macht, nämlich der von ihm naturgemäss gewollten fehlerfreien Kommunikation mit anderen Rechnern.

Es drängt sich demnach die Frage auf, ob nun effektiv noch die ursprünglich zur Verfügung gestellten Ressourcen benutzt werden, oder eher, bedingt durch die Umnutzung des Parasiten, unwissentlich ausnutzbare Ressourcen beansprucht werden.

2.1.2 Beeinträchtigung des Wirtes

Nun ist es natürlich entscheidend zu wissen, in welchem Masse der Wirt diesen Parasitismus zu spühren bekommt, oder etwas expliziter ausgedrückt, in welchem Ausmass er dadurch beeinträchtigt wird.

Auf der untersten Ebene der durch den Parasiten ausgelösten Operationen liegen zwei atomare Grundfunktionen (siehe Abschnitt 1.4). Jede parasitäre Einheit besteht aus ausschliesslich einem dieser Atome, und erst wenn ein Wirt eine ganze Serie – es sind

dies selbst für primitive Operationen wie etwa die Multiplikation zweier Zahlen bereits mehrere tausende – solcher Einheiten ,,bewirtet", lässt sich so für den Parasiten etwas sinnvolles erreichen. Eine einzelne solche Einheit fällt für den Wirt kaum ins Gewicht, kann also aus praktischer Sicht nicht, wohl aber aus ethisch-theoretischer Sicht problematisch sein.

Das dem Solitärparasitismus entsprechende ,,Parasitic Computing" bedient sich nun allerdings nicht eines einzigen Wirtes, sondern verteilt die Arbeit auf eine Unmenge von Rechnern, im Idealfall so, dass pro Wirt nur eine der atomaren Grundoperationen ausgeführt wird. Ein einzelner Parasit wird in der Praxis also kaum feststellbaren Schaden anrichten können.

2.2 Reichweite parasitärer Rechenmethoden

Auf die Nichtrentabilität der Methode wurde bereits in Abschnitt 1.5 eingegangen, und daraus folgt, dass ,,Parasitic Computing" zum momentanen Zeitpunkt nur für Experimente eingesetzt werden kann.

Denkbar ist aber natürlich auch, dass Applikationen entwickelt werden, welche bösartig Datenpakete nach dem parasitären Prinzip verteilen, mit dem Ziel, einen fremden Rechner zu überlasten und diesen somit auszuschalten. Solche Missbräuche sind in Computernetzwerken weitreichend bekannt unter dem Namen ,,denial of service attack" (vgl. [Fer00]).

3 Erweiterte Zusammenhänge

Nachfolgend einige Punkte, welche erweiterte Zusammenhänge und Erklärungen zum Thema "Parasitic Computing" und Ethik liefern sollen.

3.1 Protokolle mit Sicherheitslücken?

Mittlerweilen ist es bekannt und öffentlich, dass zum Beispiel mittels ICMP oder TCP primitive Grundoperationen (XOR und AND) berechnet werden können. Naheliegenderweise sollten mit dieser Erkenntnis derartige Protokolle für unsicher, oder zumindest problematisch erklärt deshalb nicht mehr verwendet werden. Dies ist aber nicht praktisch durchführbar, denn solche weitverbreiteten, etablierten Kommunikationsmechanismen können unmöglich kurzerhand für ungültig erklärt oder ausgeschaltet werden. Das halbe Internet beispielsweise basiert auf TCP. Falls nun TCP nicht mehr verwendet werden sollte, würde das Internet nicht mehr in diesem Sinne existieren können. Eine Änderung oder Anpassung in den Kommunikationsstandards würde weltweit enorme Auswirkungen mit sich bringen und auf Hersteller- wie auch auf Anwenderseite auf heftigen Widerstand stossen.

3.2 Abwägen zwischen Schaden und Nutzen

Erwiesen ist, dass Ressourcen, wie etwa Berechnungen mittels ICMP oder TCP, nicht rentabel sind. Das bedeutet, dass zum zusammenstellen eines Paketes, dass eine einzige atomare Grundoperation (binäres XOR oder AND) verteilt berechnen kann, ein Vielfaches an Rechenleistung erforderlich ist. Zudem stellt die Internetverbindung in

jedem Fall einen Flaschenhals dar, denn um Rechenkapazitäten zu erreichen wie diese etwa von einem modernen Prozessor zur Verfügung gestellt werden, würden auch die momentan schnellsten gebräuchlichen lokalen Netze mit Übertragungsraten von bis zu 1'000 MBit in der Sekunde bei Weitem nicht ausreichen. Kurz gesagt, die atomaren Grundoperationen der momentan bekannten Methode geben (noch) zu wenig her.

3.3 Vorsorgliche Massnahmen

Bereits haben einige Computerfirmen wie beispielsweise Microsoft oder Oracle, welche vielbesuchte und weltbekannte Internetseiten anbieten, die verzichtbareren Protokolle wie ICMP abgestellt (dies lässt sich mittels dem auf üblichen Bestriebssystemen unter dem Namen `ping` bekannten Befehl nachvollziehen). Das Protokoll ICMP dient primär zu Diagnose- und Kontrollzwecken und ist für den Anbieter einer Website nicht zwingend zu implementieren.

Solches Handeln hat natürlich verschiedene Gründe (siehe auch [Lut01]), einerseits versucht man auf diese Weise sämtlichen nicht zwingend nötigen Datenverkehr zu unterbinden, andererseits will man sich damit vor möglicherweise zukünftig bekanntwerdenden Sicherheitslücken schützen. Dies rein nach dem Prinzip, je kleiner die Möglichkeit, desto kleiner auch die Wahrscheinlichkeit. Fast täglich entdeckt die Internetgemeinde neue Sicherheitsrisiken in allen Arten von Programmen, Protokollen und Algorithmen, welche auf einschlägigen Websites – sei dies der Warnung oder der Bekanntmachung aus negativen Gründen wegen – publiziert werden (vgl. [(CE02]).

3.4 Gesetzliche Grundlagen

Wie nirgendswo anders als im diesbezüglich blutjungen Informatikbereich hinkt die Gesetzgebung den stetig und rasant wachsenden Technologieneuerungen hinten nach. Dies äussert sich primär dann auf fatale Weise, wenn in einem Land mit schlecht an neue Technologiemöglichkeiten angepasser Gesetzgebung zum Beispiel ein Virus entwickelt wird, welches landesgrenzenübergreifend Schäden in Milliardenhöhe anrichtet, der Täter aber durch das Netz der Gesetze fällt. Bestes Exempel dafür ist das erst kürzlich in den Medien ausgiebig behandelte Virus „I Love You" (siehe [Oeb00]).

Oben genanntes Beispiel des Entwicklers *Onel de Guzman* zeigt die typische Problematik auf, wie Gesetzgebungen, die in ihrem Sinne abschreckende Wirkungen haben sollten auf ethische Zuwiederhandlungen, im Bereich Informatik versagen und so einschlägige Entwickler und Hacker geradezu anspornen, EDV-Anlagen lahmzulegen, weil ja in den meisten Fällen viel Ruhm und Medienpräsenz und auf sie zukommt, meist jedoch ohne gesetzliche Konsequenzen, falls den Fehlbaren überhaupt faktisch ein Verbrechen nachgewiesen werden kann.

3.5 „Erlaubte" verteilte Berechnungen

Das zur Zeit populärste Projekt, welches fremde Rechenkapazitäten im professionellen Sinne legal nutzt, ist zweifelsohne das Unternehmen SETI (The Search for Extraterrestrial Intelligence [SET01]) der amerikanischen Berkeley Universität. Beim Projekt SETI gilt es, Unmengen von durch Radarstationen aus dem Weltall aufgezeichneten

Daten nach bestimmten Mustern abzusuchen. Dabei handelt es sich um riesige Datenmengen, die einzelner Computer mit heutiger technologie unmöglich bewältigen kann – es muss die Aufgabe also auf tausende von Maschinen verteilt werden.

Eine Applikation, welche von den Teilnehmenden auf deren Rechnern (freiwillig) installiert wird, lädt hierbei jeweils ein Datenpaket mit vorgängig aufgezeichneten Signalen zur lokalen Auswertung herunter. Diese Signale werden mit kleinster Priorität immer dann schrittweise verarbeitet, wenn der eigene Computer über freie Rechenressourcen verfügt und nicht anderweitig genutzt wird. Sobald ein Paket fertig ausgewertet ist, schickt die Applikation dieses wiederum via Internet zum Server zurück, welcher die Applikation mit einem ächsten Paket zur Auswertung versorgt, und so weiter.

3.6 Komerzielle Nutzung

Das Nutzen fremder Rechenkapazität bietet auch einen aus komerzieller Sicht interessanten Ansatzpunkt. So könnte beispielweise freie Rechenkapazität – und diese übersteigt die produktive Phase des Rechenwerkes auf den meisten Maschinen um ein vielfaches – gegen eine Entgeltung angeboten werden. Denkbar wäre, dass Privatpersonen Rechenzeit gegen Bezahlung auf Stundenbasis an Firmen verkaufen, was Letzteren die Möglichkeit eröffnet, massive verteilte Rechenwerke ohne Wartungs- und Betreibungsaufwand aufzubauen.

3.7 Künstliche Intelligenz und parasitäre Methoden

Momentan vielleicht grossenteils noch Science Ficiton, möglicherweise aber bald schon in der Realität im Einsatz sind Programme mit echter sogenannter künstlicher Intelligenz. Wie lange es dauern wird, bis sich solche Programme dann, so wie dies die Forscher von heute tun, durch eigene Methoden fremde Rechenkapazität verschaffen, wird sich wohl oder übel zeigen. Das Verhalten solcher Programme wird mit ihrem weiteren Fortschritt schwerer und schwerer Vorhersagbar und allfällige Auswirkungen sind kaum abzuschätzen.

4 Lösungen

Nachfolgend einige Punkte, welche aufzeigen sollen, dass sich die Suche nach Lösungen, parasitäre Rechenkapazitäten nicht zu ermöglichen respektive Missbräuche zu verhindern, alles andere als einfach gestaltet.

Folgende Überlegungen sind also nicht zwingendermassen als pfannenfertige Lösungsansätze zu verstehen, sondern auch als Denk- und Wegfindungsilfe, wie mit einem Zwiespalt zwischen Gutmütigkeit einerseits (Bereitstellung eines Service, beispielsweise ICMP) und einseitiger Ausnutzung („Parasitic Computing”) umgegangen werden kann.

4.1 Zwiespalt

Das ganze Prinzip der Vernetzung von Computern und Maschinen beruht darauf, irgendwelche Datenpakete, welche hier nicht näher erläutert werden sollen, sicher, performant und vor allem fehlerfrei durch ein Netzwerk von Knoten zu transportieren. Dazu müssen die beteiligten Rechner zwingendermassen Mechanismen und Services zur

Verfügung stellen, welche Möglichkeiten bieten, fehlerhafte Datenpakete zu identifizieren und entsprechend darauf zu reagieren. Nur so kann ein korrekter Datentransport gewährleistet werden.

Parasitäre Methoden der Fremdkapazitätsnutzung zielen genau darauf ab, solche Mechanismen der Transportkontrolle von oben erwähnten Datenpaketen zu missbrauchen.

Offensichtlicherweise können Services und Mechanismen, welche die unabdingbare Voraussetzungen bilden zur konsistenten Kommunikation innerhalb eines Netzwerkes, nicht abgeschaltet werden, da ja sonst der korrekte Transport von Datenpaketen nicht mehr gewährleistet wäre. Vielmehr müssen allgemeinverträgliche Kompromisse gefunden und realisiert werden, welche eine gute Balance zwischen Einschränkungen und Funktionalität bilden.

4.2 Zutrittskontrollen in Netzwerken

Ein Rechner, der an ein Netzwerk angeschlossen wird, soll so konfigueriert sein, dass auf ihm keine den andern Rechnern schädlichen Programme zum laufen gebracht werden können.

Ein gutes Beispiel für einen kontrollierten Zugang in ein LAN sind Computer, deren MAC-Adressen (hardwareabhängige, weltweit eindeutige Adresse pro Maschine) vorgängig beim zuständigen Verbindungsglied registriert werden müssen, bevor der Rechner überhaupt auf dem Netzwerk kommunizieren kann. Realisiert so beispielsweise in der Postfinance in Bern.

Fazit: Dieser Ansatz ist insofern lösungsrelevant, weil er eine Möglichkeit aufzeigt, wie unter normalen Umständen sicher verhindert werden kann, dass sich fehlbare Maschinen bösartigerweise in einem Netzwerk betätigen können.

4.3 Kontrollierte Nutzung parasitärer Methoden

Falls parasitäre Methoden innerhalb eines Netzwerkes angewendet werden, dann sollen diese Varianten der Rechenkapazitätsnutzung bewusst, kontrolliert und ohne schädliche Nebeneffekte vor sich gehen. Unerwünschte Effekte bilden dabei folgende möglichen Punkte:

- Überlastung des Netzwerkes
- Ungleiche Ausnutzung freier Rechenressourcen

Fazit: Dieser Ansatz soll verdeutlichen, dass freie Ressourcen *sinnvoll* genutzt werden sollten. Damit kann aber keineswegs verhindert werden, dass diese Ressourcen auch anderweitig genutzt werden könnten. Desshalb appelliert diese Idee über den Umgang mit parasitären Methoden einmal mehr an unsere ethischen Prinzipien, welchen wir Folge leisten sollten.

4.4 Überwachung der Rechner im Netzwerk

Eine Variante, fehlbare Netzwerkteilnehmer zu identifizieren, könnte wie folgt beschrieben realisiert werden:

Bekanntlicherweise kommunizieren Rechner via einen oder mehrere Knotenpunkte innerhalb eines Netzwerkes miteinander. Jeder Knoten könnte seinen Verkehr kontrollieren und abnormale Aktivitäten wie beispielsweise auffallend viele ICMP Pakete eines einzigen Rechners oder einer Grupper von Rechnern einem speziell dafür eingerichteten Server melden, welcher die potentiellen „Hackermaschinen" entlarvt und entsprechende Alarme auslösen kann.

Fazit: Die Idee dieses Ansatzes verhindert zwar die Missnutzung fremder Rechenressourcen nicht, will aber einen Weg aufzeigen, wie Missbräuche von gutmütigen Mechanismen rasch aufgedeckt und entsprechend behandelt werden können.

4.5 Abschalten von nicht essentiellen Services

Eine wirksame Möglichkeit, sich gegen eine vielzahl von potentiellen Schwachstellen zu schützen, bietet die Variante, Services wie beispielsweise ICMP, welche nicht essentiell sind für die korrekte Übertragung von bestimmten Daten, gar nicht erst zur Verfügung zu stellen. Dies birgt zwar einerseits offensichtlich gewisse Nachteile mit sich, was etwa die Kontrolle von Netzwerkverbindungen anbelangt, andererseits aber kann so verhindert werden, dass triviale bösartige Programme einem Server ohne weiteres Schaden hinzufügen oder ihn lahmlegen können (siehe Abschnitt 3.3 Seite 179).

Fazit: Dieser Lösungsansatz bietet in seiner Idee keinen absoluten Schutz gegen sämtliche Missbräuche von Services, will aber aufzeigen, dass mit verkraftbaren Einschränkungen, welche im jedem Falle einen Komfort- und Funktionsverlust zur Folge haben, einiges unternommen werden kann gegen unerwünschten Ressourcendiebstahl und weitere ungewollte, durch fremde Maschinen gesteuerte Aktionen.

5 Anhang

5.1 Abkürzungsverzeichnis

LAN	Abkürzung für „Local Area Network". Ein lokal angelegtes Netzwerk - im Gegensatz zu WAN, das überregional Arbeitsstationen und Netzwerke verbindet. „Lokal" bezieht sich in diesem Sinne auf einen gemeinsamen Standort, wie beispielsweise ein Firmengelände oder einen Raum.
MAC Adresse	Abkürzung für „Media Access Control". MAC wird im Netzwerk-Umfeld allgemein als „MAC-Adresse" einer Netzwerkkarte verstanden. Sie ist fest auf der Karte gespeichert und weltweit eindeutig; es handelt sich sozusagen um die unverwechselbare Seriennummer einer Netzwerkarte.
TCP/IP	Abkürzung für „Transmission Control Protocol/Internet Protocol". Bezeichnet zumeist die ganze Familie von Protokollen, die ursprünglich für das US-Verteidigungsministerium (Department of Defence - DoD) entwickelt wurden, um Computer in verschiedenen Netzwerken miteinander zu verbinden.
UDP	Abkürzung für „User Datagram Protocol". Bezeichnet ein übertragungsprotokoll. Es kann anstatt des TCP aus den TCP/IP-Protokollen verwendet werden, mit dem Unterschied, dass es nicht wartet, bis es eine Bestätigung erhält, ob ein Paket angekommen ist, oder nicht.
ICMP	Abkürzung für „Internet Control Message Protocol". Ist ein Nachrichten- und Fehler-Protokoll (Bestandteil von TCP/IP) zwischen Gateway und Host. Es ist jedoch nicht für den Benutzer erkennbar, sondern verrichtet seine Arbeit, nämlich eine fehlerfreie Übertragung zu gewährleisten bzw. den Sender über Probleme im Netzwerk zu informieren.
Internet	Das Internet ist eine dezentrales, weltumspannendes Netzwerk, d.h. es ist von keinem einzelnen Computer abhängig. Ursprünglich als ARPANet für das Militär in den USA entwickelt, ist es heute für Millionen Benutzer zugänglich. Das Netz besteht aus einer Reihe von Unternetzen, den Subnets; als Netzwerkprotokoll wird immer TCP/IP (Transmission Control Protocol / Internet Protocol) verwendet.

Erläuterungen entnommen von `http://www.computerlexikon.com`.

PARASITIC COMPUTING
Labjournal

Luzian Scherrer
Jürg Reusser

6. Januar 2003

Zusammenfassung

Das vorliegende Labjournal der Diplomarbeit ,,Parasitic Computing" gibt einen chronologischen Überblick über alle durchgeführten Aktivitäten.

April 2002

26. April

- Projektbeginn: Kick-off Meeting mit JB, MD, JR und LS. Sitzungsprotokoll auf Webserver abgelegt (`Sitzung_20020426.pdf`).

Mai 2002

2. Mai

- Entwicklungsmaschine (CVS, etc) installiert und ans Netz gehängt: Sun Ultra Enterprise 1, Solaris 8, 512 MB RAM, 2 mal 18GB Disks mit Mirror für Datensicherheit.
- Webserver installiert: `http://parasit.org`.
- Mailingliste `all@parasit.org` eingerichtet.
- Erste Rohversionen vom Sitzungsprotokoll und Labjournal sind auf dem Webserver, muss später alles auf DocBook reformatiert werden.

4. Mai

- DocBook installiert und getestet; JR und LS nicht zufrieden. Dokumente werden nicht nach unseren Erwartungen formatiert, Installation äusserst aufwendig, Benutzung sehr komplex. LaTeX wird als Alternative diskutiert.

6. Mai

- Erste Rohversion des Pflichtenheftes erstellt.
- Bisherige Dokumente (Sitzungsprotokoll, Labjournal) umgeschrieben mit LaTeX und auf Webserver gestellt.

7. Mai

- Sitzung mit JE, MD, LS und JR.

8. Mai

- Simulation für XOR und AND Operationen anhand der TCP Checksumme erstellt, das Programm ist auf dem Webserver: `parasimu.c`.

10. Mai

- Erste Studie zum parasitären Computing. Das Programm `paraicmp.c` lässt mittles ICMP Messages XOR und AND berechnen.

12. Mai

- Zweite Studie zum parasitären Computing. Eine minimale virtuelle Maschine welche auf den Instruktionen XOR und AND aufbaut und so eine erweiterbare virtuelle ALU zur Verfügung stellt. Der ICMP Code konnte wesentlich optimiert werden mit dem Prinzip der Kennzeichnung der Messages mittels Sequence Nummer. Das Packet ist auf dem Webserver abgelegt.

17. Mai

- Das CVS Repository ist nun auch remote verfügbar:

CVSROOT	:ext:login@parasit.org:/usr/local/repository/parasit
CVS_RSH	ssh
Modules	src, doc, html, para

- LS und JR haben die Präsentation für den kommenden Dienstag vorbereitet.
- Wenn ein Host zuviele ICMP Packete innert kürzester Frist erhält, scheint er auch Packete mit falschen Checksummen zu beantworten. LS erstellt ein Testprogramm um der Sache auf den Grund zu gehen.
- JR schreibt die Draftversion des Pflichtenheftes auf LaTeX um.

20. Mai

- JR hat die Rohfassung des Pflichtenheftes nach LaTeX übersetzt.
- Bis in einer Woche werden die Korrekturen, die sich aus der Sitzung vom 7. Mai 2002 ergaben, in das Pflichtenheft einbezogen sein.

21. Mai

- LS und JR präsentieren `paraicmp` und erläutern die Funktionsweise des Source-Code.

24. Mai

- JR arbeitet das Pflichtenheft weiter aus und befasst sich mit darstellungsspezifischen Details von LaTeX.

28. Mai

- Fertigstellung Draftversion des Pflichtenheftes zur Besprechung.
- Sitzung mit JE, JB, MD, JR und LS. Das Pflichtenheft wurde besprochen und Änderungsvorschläge seitens der Betreuer festgehalten. JR und LS werden am kommenden Freitag eine diese Änderungen enthaltende Neufassung publizieren. Da an dieser Sitzung abgesehen vom Pflichtenheft nichts besprochen wurde, gibt es kein spezielles Sitzungsprotokoll.

30. Mai

- Pflichtenheft komplett überarbeitet.

Juni 2002

3. Juni

- E-Mail an Herrn Eich, mit der Bitte, das Pflichtenheft durchzulesen. Experte befindet Pflichtenheft für gut.

5. Juni

- Besprechung des komplett überarbeiteten Pflichtenheftes im Rahmen JE, MD, LS, JR. Bis auf kleinere Korrekturen ist das Pflichtenheft nun fertiggestellt.
- E-Mail an `all@parasit.org` mit dem Hinweis, dass das Pflichtenheft nun als Version 1.0 vorliegt und mit Terminvorschlag, wann das Pflichtenheft durch die Betreuer und den Experten abgenommen werden kann.

11. Juni

- Offizielle Übergabe des Pflichtenheftes an die Betreuer. Leider kann Herr Eich, unser Experte, nicht teilnehmen.
- Es wurde festgelegt, dass es sich nicht lohnt spezielle Protokolle für jede Sitzung zu schreiben. Die besprochenen Punkte fliessen jeweils direkt in die Dokumente ein.

14. Juni

- Struktur und Aufbau des Dokumentes für die Ethik Semesterarbeit diskutiert.
- Erste Vorversion mit der grundlegenden Gliederung erstellt.

16. Juni

- Kapitel 1 und 2 der Ethik-Arbeit geschrieben und korrigiert.

27. Juni

- Debugging der Prototypen.
- Grobplan weiteres Vorgehen.

30. Juni

- Austausch gewonnener Erkenntnisse beim Literaturstudium.

Juli 2002

03. Juli

- JR und LS: Besprechung Implementation ALU.

09. Juli

- JR und LS erstellen Detaillierten Vorgehensplan fr die kommenden Wochen. Koordination auch mit den anderen Aufgaben (Ethik, Seminar).

17. Juli

- Draftversion des Realisierungskonzeptes in Angriff genommen.

18. Juli

- Überarbeitung Themenaufbau des Realisierungskonzept.

23. Juli

- Arbeiten am Realisierungskonzept.

24. Juli

- LS und JR treffen sich in Bern und besprechen die Problematik, dass beide zu unterschiedlichen Zeiten in den Ferien sein werden. Beschluss: JR wird am Montag, 29. Juli 2002 versuchen, Herr Eckerle zu kontaktieren.

28. Juli

- Realisierungskonzept Draft 1.0 fertiggestellt.

August 2002

2. August

- LS und JR treffen sich in Bern und besprechen den Terminplan.
- Semesterarbeit Version 1.0 fertiggestellt.
- Weiterarbeit am Realisierungskonzept.

3. August

- Gemeinsame Weiterarbeit am Realisierungskonzept.
- Ethik Arbeit Version 1.0 fertiggestellt.

15. August

- Besprechung Realisierungskonzept mit Betreuern.
- Vortrag Seminar
- Vortrag Ethik
- Abgabe Ethik Seminararbeit

20. August

- LS/JR besprechen weiteres Vorgehen und Zeitplan.
- Neue Draftversion Realisierungskonzept und Überarbeitung des Vorschlages der Assemblersprache.

27. August

- LS Besprechung Realisierungskonzept mit Betreuern, neuer Vorschlag VM und Assembler.

28. August

- Realisierungskonzept gem. Besprechung vom 27. ausarbeiten

30. August

- Realisierungskonzept gem. Besprechung vom 27. ausarbeiten

31. August

- Realisierungskonzept gem. Besprechung vom 27. ausarbeiten

September 2002

03. September

- Sitzung mit Betreuern zur besprechung des komplett überarbeiteten Realisierungskonzeptes $\Rightarrow$ ausgefallen, Missverständnis.

08. September

- Die meisten Kritikpunkte, welche Michael Dürig zum aktuellen Realisierungskonzept angebracht hatte, sind verarbeitet.

10. September

- Sitzung Eckerle, Dürig, Reusser und Scherrer. Realisierungskonzept besprochen, Änderungswünsche diskutiert und vorzunehmende Änderungen festgehalten.

11. September

- Mail von Eckerle mit weiteren Details im Realisierungskonzept, die noch anzupassen sind.

17. September

- Diplomprüfungen – Pause Diplomarbeit

19. September

- Diplomprüfungen – Pause Diplomarbeit

22. September

- Mail von Herr Eckerle (siehe 11. September) begonnen zu verarbeiten.

24. September

- Weiterarbeit am Realisierungskonzept.
- Meeting zur Besprechnung des Realisierungskonzeptes. Ziel: Abgabe ⇒ Ausgefallen, Betreuer nicht aufgetaucht :(
- Mail an W. Eich mit Terminanfrage für Besprechung.

28. September

- Besprechung Realisierungskonzept mit Eckerle und Dürig. CVS Revision 1.60 ist Release-Candidate; keine Änderungen mehr gewünscht.
- Dokument zur Korrekturlesung an zwei unabhängige Personen gegeben.

Oktober 2002

3. Oktober

- Start der Implementationsphase. LS und JR diskutieren generelles Vorgehen.

9. Oktober

- Scanner (flex) und Parser (eigenbau) für die 4IA-Sprache implementiert.

14. Oktober

- Simulationsmodus der ICMP Berechnung implementiert.
- Virtuelle Maschine implementiert ⇒ 4IA-Programme (Beispiele aus dem Script) funktionieren einwandfrei!
- Debugging mechanismen implementiert.

17. Oktober

- LS/JR: Review über virtuelle Maschine, Scanner und Parser.
- LS/JR: Diskussion weiteres Vorgehen und Arbeitsteilung.

19. Oktober

- Design für Xto4 Cross-Compiler besprechen.

22. Oktober

- Design für Xto4 Cross-Compiler ausarbeiten.

25. - 27. Oktober

- Xto4 in den Grundzügen implementieren.

28. Oktober

- Besprechung und Vorbereitung für morgige Präsentation.

29. Oktober

- Meeting mit den Herren Eich, Eckerle und Dürig.
- Abnahme Realisierungskonzept.
- Demonstration Prototypen.
- Besprechung weiteres Vorgehen:
  - Parallelisierung anschauen für Addition
  - Statistiken erweitern (parasitär vs. lokal)
  - XIA abschliessen

November 2002

1. bis 24. November

- pshell: Fertigstellen des Release 1.0
- Xto4: Fertigstellen des Release 1.0
- `API` für Xto4 nachgeführt
- Source-Code Dokumentation der `pshell` mit Doxygen[1] erstellt.
- Chora[2] installiert; das CVS Repository kann nun öffentlich eingesehen werden.
- Zugriffsstatistiken der Website sind online und aktualisiert.

25. November

- Besprechung mit Eckerle und Dürig, weiteres Vorgehen besprechen.
- Nächste Priorität ist die Dokumentation; Anforderungen und Umfang abklären. Zusätzlich ist erwünscht, eine Addition mehrer Bits gleichzeitig berechnen zu können.

[1] siehe `http://www.doxygen.org`
[2] siehe `http://horde.org/chora/`

29. November

- Handbuch in Angriff genommen. Installation und Kompilation ist nun beschrieben.
- Addition mehrerer Bits in Angriff genommen.

30. November

- Parallelisierung der Addition auf Bitebene weitgehend als Prototyp implementiert. Eine höhere Parallelisierung als 4-Bit lässt sich allerdings nicht erreichen, da bereits bei 8-Bit Operatoren und Sequenznummern überlappen würden (ICMP Paket ist 16-Bit breit, Bitbreite der Parallelisierung muss ein Teiler der Registerbreite sein).
- False positive check eingebaut: es gibt Hosts, welche ICMP Pakete mit falschen Checksummen beantworten.

Dezember 2002

1. Dezember

- Script entwickelt, welches Distributionen erstellt.
- Homepage den neuen Gegebenheiten angepasst.
- Am Handbuch weitergeschrieben, insbesondere an XIA Programmiersprache.

2. Dezember

- LS und JR haben in Bern kurz den Aufbau des Handbuches besprochen.
- Weiterarbeit an Handbuch.tex, insbesondere am Kapitel 4.

5. Dezember

- Website neu gestaltet.

6. Dezember

- LS und JR treffen sich in Bern und besprechen Handbuch.
- Weiterarbeit an Dokumentationen.

7. Dezember

- Handbuch, Teil xto4, ist in einer Rohfassung fertig.
- doc Verzeichnis hat nun ein Unterverzeichnis, in welchem sämtliche Includes (Figures) abgelegt werden können.

8. Dezember

- System Design in Angriff genommen.

10. Dezember

- Handbuch: An `pshell` Teil weiter geschrieben.
- Homepage mit Links auf Handbuch und System Design erweitert, shell scripts angepasst.

11. Dezember

- System Design: Klassendiagramme eingefügt.

12. bis 19. Dezember

- Am System Design weitergearebeitet.

20. Dezember

- Meeting in Bern: System Design, Teil `xto4` soweit ok.
  Nächstes und letztes Meeting findet am 6. Januar 2003 um 18 Uhr 15 statt inklusive Herr W. Eich.
- Fehlerkorrekturen System Design

21. Dezember

- Handbuch: Release-Candidate 1.0 fertiggestellt.

23. Dezember

- Projektbericht erstellt.

25. Dezember

- Weiterarbeit am Projektbericht.

26. Dezember

- Weiterarbeit am Projektbericht.
- Dokumentenuebersicht erstellt.
- Update des documentation-index (HTML). Enthält nun Links auf alle Dokumente.
- Telefonkonferenz: Stand der Dinge festhalten und Planung der letzten Woche. Weiteres Vorgehen: Voraussichtlich Samstag, den 28. Dezember 2002, werden sich LS und JR in Bern treffen und das Dokument Auswertung in Angriff nehmen.

28. Dezember

- Weiterarbeit am Projektbericht. Alpha release zum durchlesen.
- Dokument Auswertungen gem. den im Laufe der letzten Woche ermittelten Resultaten erstellt.
- CD-ROM Distribution besprochen und Rohlinge bestellt.

29. Dezember

- Korrekturen Projektbericht, Auswertungen und Handbuch.

30. Dezember

- Binaries von `pshell` und `xto4` erstellt.
- `Makefile` für die Installation der Binaries erstellt.
- `README` für die Distribution der Binaries erstellt.

31. Dezember

- Mit den LaTeX Contribution Packages `combined` und `combnat` und `combinet` ein zusammenfassendes Dokument mit Titelblatt und Haupindex erstellt. Einige ästhetische Änderungen an `combined` vorgenommen.
- Gesamtausdruck erstellt zur Korrekturlesung.

Januar 2003

2. Januar

- Korrekturlesung

3. Januar

- Website mit W3C HTML-Validator geprüft und entsprechend korrigiert.
- Links auf der Website erweitert (Notredame University Dokumente).
- Webstatistik aktualisiert.

5. Januar

- Letzte Korrekturen durchgeführt.
- Dokumente in mehrfacher Fassung ausgedruckt und gebunden (HTA).
- Website bereinigt für die Abgabe.

6. Januar

- Formelles Abgabemeeting an der HTA-BE mit den Betreuern.

7. Januar

- Offizielle Abgabe der Diplomarbeit.

PARASITIC COMPUTING
Projektbericht

Version 1.0

Luzian Scherrer
Jürg Reusser

6. Januar 2003

Zusammenfassung

Das Prinzip „Parasitic Computing" wurde erstmals im August 2001 öffentlich erwähnt [BFJB01], als es Wissenschaftlern der Notre Dame Universität im US-Bundesstaat Indiana gelang, fremde Rechenkapazität ohne Wissen und Einwilligung derer Besitzer zur Lösung mathematischer Probleme zu nutzen.

Der vorliegende Projektbericht zieht eine Schlussbilanz über die gleichnamige Diplomarbeit der Hochschule für Technik und Architekt Bern, in welcher die vorgestellte Methode zu einem vollständig programmierbaren, parasitären Rechenwerk ausgebaut wurde.

Inhaltsverzeichnis

1 Parasitic Computing

„Parasitic Computing" ist die Bezeichnung für ein Vorgehen, mittels welchem durch Ausnutzung externer Ressourcen Berechnungen durchgeführt werden können. Obwohl diese Ausnutzung ohne Wissen und Einwilligung der Betroffenen geschieht, ist „Parasitic Computing" nicht mit einem Computervirus oder ähnlichem vergleichbar. Das Prinzip beruht ausschliesslich auf der Nutzung von willentlich angebotenen Kommunikationsprotokollen, dies allerdings zu anderem als dem ursprünglich vorgesehenen Zwecke.

1.1 Ausgangslage

Ende des Jahres 2001 haben Wissenschaftler der Universität Notre Dame im US-Bundesstaat Indiana eine Methode demonstriert, mit der man ohne Wissen und Einwilligung der Anwender deren Rechnerkapazität im Internet nutzen kann [BFJB01]. Sie lösten ein mathematisches Problem mit Hilfe von Web-Servern in Nordamerika, Europa und Asien, ohne dass die Betreiber dieser Sites etwas davon merkten. Dazu nutzten sie das Kommunikationsprotokoll TCP aus: Um die Integrität von Daten sicherzustellen, platziert der Sender eines Datenpakets eine Prüfsumme (Checksumme) über die im Datenpaket enthaltenen Datenbits in dessen Header. Auf der Empfängerseite wird diese Prüfsumme nachgerechnet und je nach Ergebnis wird das Datenpaket als korrekt akzeptiert, oder es wird verworfen. Durch geeignete Modifikation der Pakete und deren Header gelang es, einfache Berechnungen durchzuführen.

Diese Forschungsarbeit wurde in diversen Artikeln öffentlich publiziert und besprochen [Fre02], sie ist weitgehend auch aus technischer Sicht detailliert dokumentiert [oND02].

1.2 Inhalt der Arbeit

Im Unterschied zu der vorgestellten Methode, welche mit TCP Datenpaketen arbeitet, verwenden wir in dieser Arbeit ICMP Pakete. Dies, weil das ICMP-Protokoll erstens einen wesentlich geringeren Overhead mit sich bringt, ausserdem nicht von Protokollen höherer OSI-Layer abhängt und schliesslich gemäss [GF89] von jedem Host im Internet zur Verfügung gestellt werden muss.

Die mittels Modifikation von ICMP Datenpaketen verteilt durchführbaren Bit-Operation AND und XOR reichen aus, um jedes auf einem klassischen Computer lösbare Problem parasitär, sprich mit fremder Rechenkapazität, zu berechnen. Mit diesen beiden Operationen können zwei ganze Zahlen addiert werden, indem die zwei Summanden bitweise, beginnend mit dem niederwertisten Bit, zueinander addiert werden (XOR) und das Übertragungsbit (AND) jeweils in die nächste Bit-Addition übernommen wird.

Aufbauend auf dieser Addition von ganzen Zahlen mit Übertragungsbit wurde eine virtuelle Maschine nach dem Modell der Registermaschine entwickelt, welche eine Additions-Instruktion einer Assemblersprache verteilt parasitär berechnen kann. Diese Assemblersprache trägt den Namen 4-Instruction-Assembler (4IA) und besteht, wie ihr Name besagt, lediglich aus der Additions- und drei weiteren Instruktionen. Diese weiteren Instruktionen dienen dem Speicherzugriff auf die virtuellen Register (Wert setzen, Wert kopieren) beziehungsweise dem Stoppen der Maschine (Halt-Instruktion)

und benötigen somit keine Leistung des rechnenden (parasitären) Prozessors. Wie im Realisierungskonzept [RS02d] dieser Arbeit gezeigt wurde, reicht der Instruktionssatz der 4IA-Sprache aus, jedes auf einem klassischen Computer berechenbare Problem zu programmieren und lösen.

Zur Steuerung der virtuellen Maschine durch den Benutzer wurde das Software-Paket `pshell` entwickelt. Es bietet, von der Bedienung einer klassischen UNIX-Shell ähnlich, umfangreiche Parameterisierungs-, Auswertungs- und Kontrollmöglichkeiten, um in der 4IA-Sprache geschriebenen Programmcode parasitär auszuführen.

Eine höhere Assemblersprache mit wesentlich erweitertem Instruktionsumfang ist die Sprache XIA[1], welche mit dem im Rahmen dieser Arbeit entwickelten Cross-Compiler `xto4` in 4IA-Code übersetzt werden kann. Das Instruktionsset der XIA-Sprache ist einerseits darauf ausgelegt, die Entwicklung von nicht trivialen Programmen zu vereinfachen, andererseits um als mögliches Backend für Compiler von Sprachen der dritten Generation zu dienen.

Abbildung 1 zeigt den Aufbau und die Abhängigkeiten der verschiedenen Schichten. Die Schicht „3GL" mit der entsprechenden Abhängigkeit ist kursiv dargestellt, weil sie im Rahmen dieser Arbeit nicht implementiert wurde.

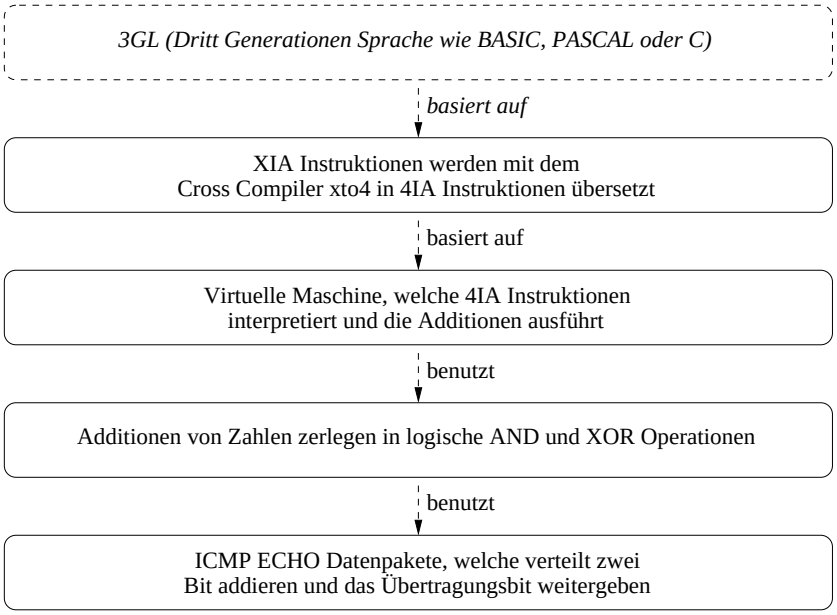

Abbildung 1: Prinzipieller Aufbau der zusammenwirkenden Schichten

[1] Extended-Instruction-Assembler: Instruktionsumfang bestehend aus den mathematischen Grundoperationen `ADD`, `SUB`, `MUL`, `DIV` und `MOD`, Vergleichsoperatoren wie `JG`, `JGE`, `JEQ`, `JNE`, etc., sowie den wichtigsten bitweisen Operationen `AND`, `OR`, `XOR`, `NOT`, `SHL`, `SHR`.

2 Projektverlauf

Das Thema dieser Dimplomarbeit wurde uns am 1. März 2002 zugeteilt. Am 26. April 2002 fand ein Kick-off Meeting statt, welches den Start des Projektes darstellte.

Die Diplomarbeit wurde unterteilt in fünf zentrale Phasen: Projekt-Setup, Realisierungskonzept, Implementation, Auswertung und Dokumentation. In den nachfolgenden Abschnitten werden die Arbeiten dieser Phasen kurz aufgezeigt.

2.1 Projekt-Setup

Die erste Phase „Projekt-Setup" diente dazu, die Infrastruktur wie beispielsweise ein Versionsverwaltungs-Repository, Mailinglisten oder die LaTeX Umgebung zur Dokumentation einzurichten. Ferner bildete das zusammentragen und studieren von themembezogenen Texten einen wesentlichen Teil dieses Projektabschnittes.

Während der Einarbeitung in die Unterlagen entwickelten wir einen ersten Prototypen, mit welchem wir die aus den vorhandenen Dokumentationen gewonnenen Ideen und Lösungsansätze nachvollziehen und ausbauen konnten. Die daraus gewonnenen Erkenntnisse waren ausschlaggebend für die weiteren Iterationen des Prototyps und schliesslich die endgültige Implementation.

2.2 Realisierungskonzept

Die Realisierungskonzept-Phase dauerte länger als angenommen, da aus der ursprünglichen Idee, lediglich Grundzüge der Implementation zu definieren, nach und nach ein Dokument entstand, welches einen konkreten Realisierungsweg mit hohem Detaillierungsgrad aufzeigte. Dies geschah einerseits Aufgrund der Erkenntnisse der fortschreitenden Prototypen, andererseits aber auch grossenteils auf rein theoretischer Grundlage.

Wärend sich das Realisierungskonzept praktisch ausschliesslich auf die virtuelle Maschine und die 4IA-Sprache bezog, wurde über den Cross-Compiler `xto4` zu diesem Zeitpunkt noch kaum diskutiert. Falls ein Compiler entwickelt werden sollte, welcher auf der 4IA-Sprache aufbaute, dann mit dem Ziel, durch einen mächtigeren Instruktionsumfang – vergleichbar etwa mit der in [Sch92] vorgestellten Sprache REGS – die Entwicklung nicht trivialer Programme zu erleichtern.

2.3 Implementation

Bereits zu Projektbeginn haben wir uns entschieden, die Software der virtuellen Maschine ausschliesslich für UNIX und Linux Umgebungen zu realisieren. Da die Anforderung bestand, direkt auf den Kommunikations-Stack des Betriebssystems zuzugreiffen, und da dies eine sehr systemspezifische Aufgabe ist, mussten wir uns auf *eine* Umgebung einschränken. Mit dieser Anforderung war auch die zur Realisierung benutzte Programmiersprache (C) definiert.

Als Programmiersprache zur Entwicklung des Cross-Compilers `xto4` entschieden wir uns für Java, dies einerseits um diesen Teil der Arbeit Plattformunabhängig zu halten und andererseits, um das technische Spektrum aus Eigeninteresse etwas zu erweitern.

Dabei haben wir auf weitere Abhängigkeiten zu Drittprodukten, wie etwa JavaCC, bewusst verzichtet, um die Software nicht zuletzt schlank und effizient zu halten.

In den realisierten Paketen legten wir grossen Wert darauf, die einzelnen Teile immer absolut modular und gekapselt zu gestalten. Dies bewahrte uns davor, zeitraubendes „Code Refactoring" zu betreiben oder Source-Code mehrfach neu zu schreiben. Der Code der Prototypen, welcher zu Beginn der Diplomarbeit geschrieben wurde, konnte grösstenteils wiederverwendet werden.

Die Implementationsphase war wesentlich weniger zeitintensiv als wir zu Beginn des Projektes angenommen hatten. Dies weil wir bereits zum Zeitpunkt der Realisierungskonzept-Erstellung die zu implementierenden Algorithmen und Module bestimmten und während der Implementierung keine nennenswerten Überraschungen auftauchten. Einzig die Entwicklung der Funktionalität, mehrstellige Bit-Additionen in einem Schritt berechnen zu können und somit eine Parallelisierung zu erreichen, wurde erst wärend der Implementationsphase diskutiert, entworfen und ausgeführt.

2.4 Auswertung

Auswertungen über die Effizient und Fehleranfälligkeit der Berechnungsmethode zu erstellen gestaltete sich aufgrund der implementierten Monitoring-Funktionalitäten als einfach und wenig zeitintensiv.

2.5 Dokumentation

Die Dokumentation wurde weitgehend parallel zur Entwicklung der Software laufend nachgeführt. Viel Zeit in Anspruch nahm das nach Abschluss der ersten Beta-Versionen erstellte Dokument „Handbuch", worin sich ausführliche Referenzen und Programmieranleitungen zu den beiden definierten Sprachen finden.

Alle Dokumente wurden mit dem Textsatzsystem LaTeX erstellt. Der Gedankte hinter diesem Entscheid war primär, uns mit dieser bis ahnhin nicht gut bekannten Dokumentations-Umgebung vertraut zu machen.

3 Ergebnisse

In den nachfolgenden Abschnitten wird aufgezeigt, wie der Terminplan eingehalten werden konnte, welche Ziele erreicht respektive nicht erreicht wurden und was wir zusätzlich implementierten.

3.1 Terminplan

Der Terminplan, welcher sich definierte durch die Meilensteine[2], erwies sich bereits kurz nach den Sommerferien als ungünstig. Durch die viel detailliertere Ausarbeitung des Realisierungskonzeptes als ursprünglich vorgesehen verschob sich die Abgabe dieses Dokumentes nach hinten, kurz vor Beginn der Herbstferien. Dafür dauerte die Implementationsphase, welche planungsgemäss zwei Wochen vor Projektabnahme abgeschlossen sein sollte, weniger lange als angenommen, so dass wir bedeutend mehr Zeit zur Verfügung hatten, die Dokumentationen Handbuch, Systemdesign, Auswertungen und Projektbericht zu erstellen und auszuarbeiten.

3.2 Ziele

Als Ziele wurden einzelne Module definiert, welche es zu erarbeiten galt. Tabelle 1 gibt Aufschluss darüber, welche Module mit welcher Priorität zur Realisierung definiert wurden.

Modul Name	*Priorität*	*Realisierungsschritt*
Virtuelle Maschine, Speicherverwaltung, Codeausführung, Eingabe, Ausgabe	hoch	erste Phase
Paketbildung, ICMP-Simulation	hoch	erste Phase
4IA-Microassembler	hoch	erste Phase
parasitäre ICMP-Berechnung, Fehlerbehandlung	mittel	zweite Phase
Hosts Priority-Queue, Statusanzeige	mittel	zweite Phase
Höhere Assemblersprache, Visualisierung, Hochsprache	gering	optionale dritte Phase

Tabelle 1: Definierte Ziele des Pflichtenheftes

Erreichte Ziele

Sämtliche Ziele mit Priorität hoch und mittel sowie ausserdem zwei Ziele mit Priorität gering wurden erreicht.

Nicht erreichte Ziele

Das Modul „Hochsprache" mit Priorität gering wurde zu Gunsten eines zusätzlichen Moduls (siehe nachfolgendes Unterkapitel 3.2) nicht implementiert, da dieses für den eigentlichen Kern der Arbeit, die parasitäre Berechnung, wenig relevant ist.

[2] Vier Meilensteine wurden festgehalten: Abnahme Pflichtenheft, Abnahme Realisierungskonzept, Abschluss Implementation und Abnahme Projektbericht.

Zusätzliche Ziele

Zusätzlich zu den in Tabelle 1 genannten Zielen entstand inmitten der Diplomarbeit, aus Überlegungen der Effizienzsteigerung der parasitären Rechennutzung, ein weiteres Modul, welches zum Ziel hatte, mehrere bitweise Additionen pro Instruktion ausführen zu können. Dieses Modul wurde gegen Ende der Implementationsphase erfolgreich umgesetzt.

4 Ausblick

Wie bereits aus der Aufgabenstellung deutlich wird, ist das in dieser Arbeit zur Anwendung kommende parasitäre Prinzip nicht rentabel einsetzbar. Die virtuelle Maschine ist ungefähr um einen Faktor von 7'000 ineffizienter als der Prozessor eines durchschnittlichen Homecomputers (vgl. [RS02a]). Es ist aber durchaus denkbar, die Methode im Zusammenhang mit anderen Protokollen (etwa solcher kryptographischer Natur) so einzusetzen, dass die Effizienz der parasitären Berechnungen diejenige des lokalen Prozessors übersteigt.

Sollte eine solche Methode gefunden werden, so kann sie mit geringem Aufwand auch in die virtuelle Maschine dieser Diplomarbeit integriert werden; Es besteht diesbezüglich eine wohldefinierte, leicht integrierbare Schnittstelle.

5 Fazit

Durch die Diplomarbeit ,,Parasitic Computing" konnten wir aufzeigen, dass jedes auf einem klassischen Computer lösbare Problem parasitär, sprich mit fremder Kapazität, berechenbar ist. Die Nutzung dieser Rechenressourcen ist jedoch höchst ineffizient, denn ein Vielfaches an lokaler Rechenleistung ist notwendig, um lediglich das Resultat einer einfach Bit-Operation zu erhalten. Die Arbeit dient somit ausschliesslich als ,,Proof of Concept" der Aufgabenstellung.

Literatur

[BFJB01] Albert-László Barabási, Vincent W. Freeh, Hawoong Jeong, and Jay B. Brockman. Parasitic computing. *Nature*, 412:894–897, August 2001.

[Boi01] Dr. Jacques Boillat. Code generierung. *Script*, 2001.

[(CE02] Computer Emergency Response Team (CERT). Advisories and incident notes. `http://www.cert.org`, August 2002.

[CVS02] CVS. Concurrent versioning system. `http://www.cvshome.org/`, Juni 2002.

[Enz01] Christian Enzensberger. *Wir Parasiten. Ein Sachbuch.* Eichborn, Frankfurt am Main, 2001.

[Fer00] Paul Ferguson. Denial of service attack resources. `http://www.denialinfo.com/`, Juli 2000.

[For02] The Internet Engineering Task Force. Ietf website. `http://www.ietf.org/`, Juni 2002.

[Fre02] Vincent W. Freeh. Anatomy of a parasitic computer. *Dr. Dobb's Journal*, 332:63–67, Januar 2002.

[GF89] Network Working Group and Internet Engineering Task Force. Rfc 1123, requirements for internet hosts. `http://www.faqs.org/rfcs/rfc1123.html`, Oktober 1989.

[Gro88] Network Working Group. Rfc 1071, computing the internet checksum. `http://www.faqs.org/rfcs/rfc1071.html`, September 1988.

[Lut01] Joerg Luther. Microsoft offline - anatomie eines gaus. `http://www.tecchannel.de/internet/640/`, Januar 2001.

[Oeb00] Alfons Oebbeke. E-mail-virus: I love you. `http://sicheres.web.glossar.de/glossar/1frame.htm?http%3A//sicheres.web%.glossar.de/glossar/z_loveletter.htm`, Mai 2000.

[oND02] University of Notre Dame. Parasitic computing website. `http://www.nd.edu/~parasite/`, Juni 2002.

[Pos81] Network Working Group (J. Postel). Rfc 792, internet control message protocol. `http://www.faqs.org/rfcs/rfc792.html`, September 1981.

[RS02a] Juerg Reusser and Luzian Scherrer. Parasitic computing: Auswertungen. `http://parasit.org/documentation`, Dezember 2002.

[RS02b] Juerg Reusser and Luzian Scherrer. Parasitic computing: Handbuch. `http://parasit.org/documentation`, Dezember 2002.

[RS02c] Juerg Reusser and Luzian Scherrer. Parasitic computing: Pflichtenheft. `http://parasit.org/documentation`, Juni 2002.

[RS02d] Juerg Reusser and Luzian Scherrer. Parasitic computing: Realisierungskonzept. http://parasit.org/documentation, Oktober 2002.

[RS02e] Juerg Reusser and Luzian Scherrer. Parasitic computing: Systemdesign. http://parasit.org/documentation, Dezember 2002.

[Sch92] Franz Josef Schmitt. *Praxis des Compilerbaus*. Hanser Studienbuecher der Informatik, Muenchen, Wien, 1992.

[SET01] SETI@home. The search for extraterrestrial intelligence. http://setiathome.ssl.berkeley.edu/, 2001.

[Ste94] Richard W. Stevens. *TCP/IP Illustrated, Volume 1*. Addison-Wesley, 1994.

[TL99] Frank Yellin Tim Lindholm. *The JAVA Virtual Machine Specification*. Addison Wesley, 1999.

[Web02] WebGain. Java compiler compiler - the java parser generator. http://www.metamata.com/products/java_cc/, August 2002.

[Wir95] Niklaus Wirth. *Digital Circuit Design*. Springer Verlag, August 1995.