PARASITIC COMPUTING
Handbuch

Version 1.0

Jürg Reusser
Luzian Scherrer

3. 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`

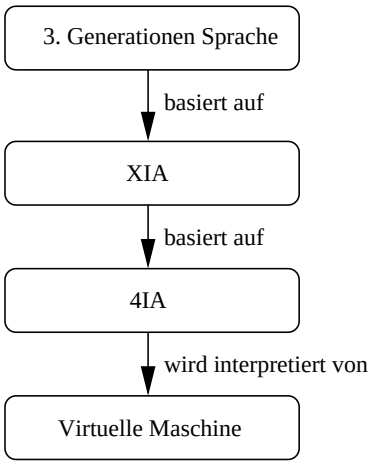

Abbildung 1: Hierarchischer Aufbau von `pshell`, 4IA und XIA

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 1.31 2003/01/05 14:55:34 ls 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 22).

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 41) 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 36) 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 36).

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 36).

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 36).

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 36).

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 36).

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 36).

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 36).

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 41) 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 eines Registers 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
```