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 [RS02b]) 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.

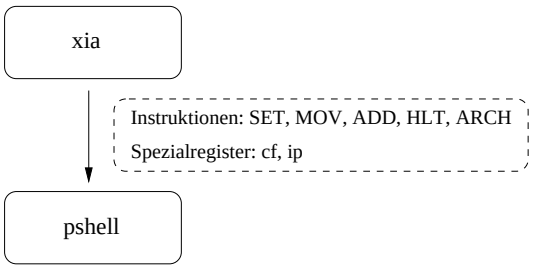

Abbildung 1: Schnittstelle zwischen den Paketen `xto4` und `pshell`

[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.

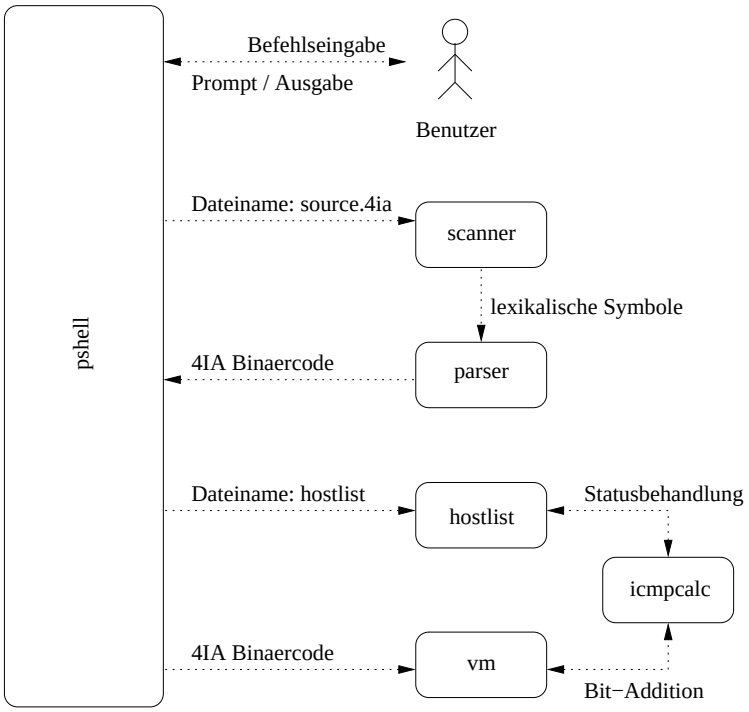

Abbildung 2: Zusammenarbeit der `pshell` Module

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 [RS02a] 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 [RS02c] 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 [RS02c] 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 [RS02c] 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.

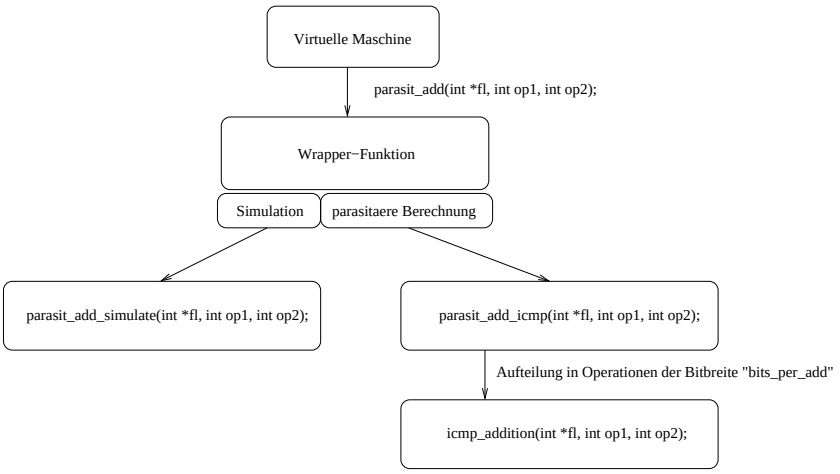

Abbildung 3: `parasit_add()` Wrapper-Funktionen

[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 [RS02c] 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.

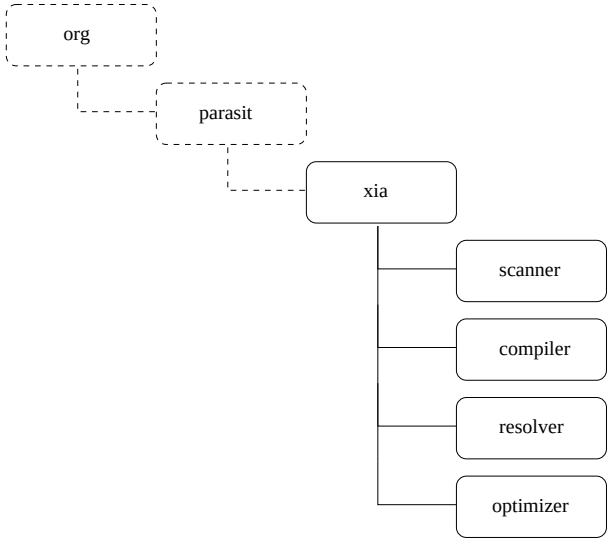

Abbildung 4: Package Hierarchie des Cross-Compilers `xto4`

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.

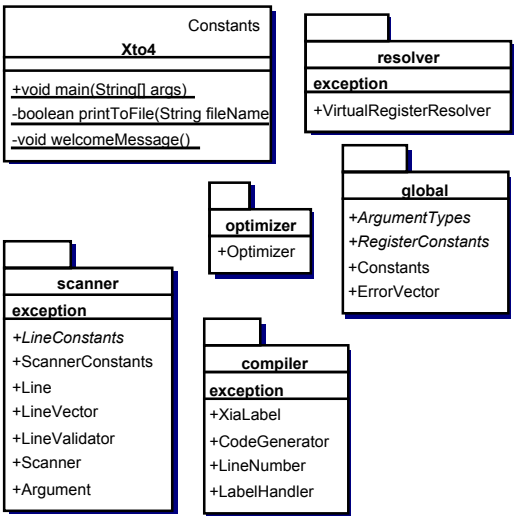

Abbildung 5: Übersicht der Packages

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.

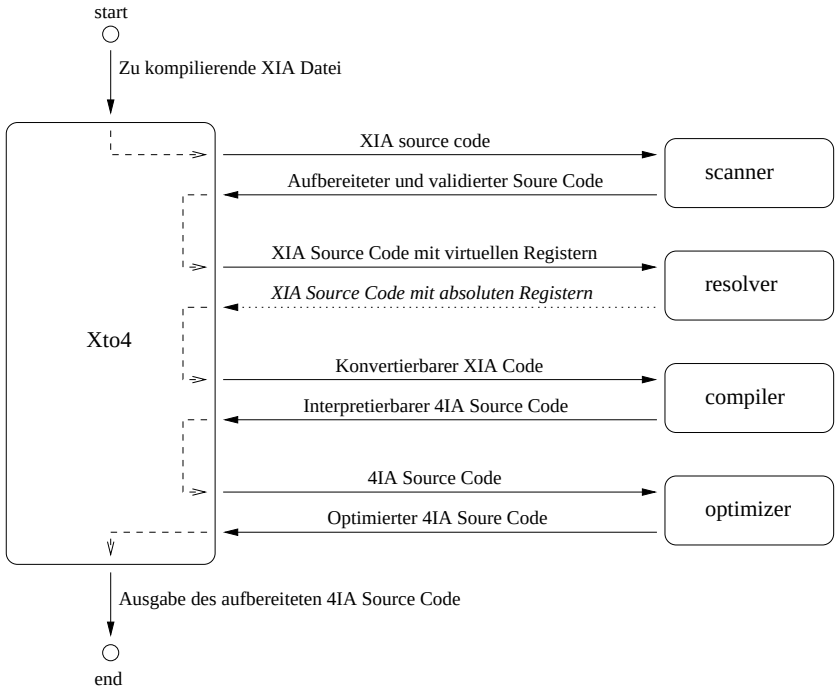

Abbildung 6: Sequentieller Ablauf des Programmes

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.

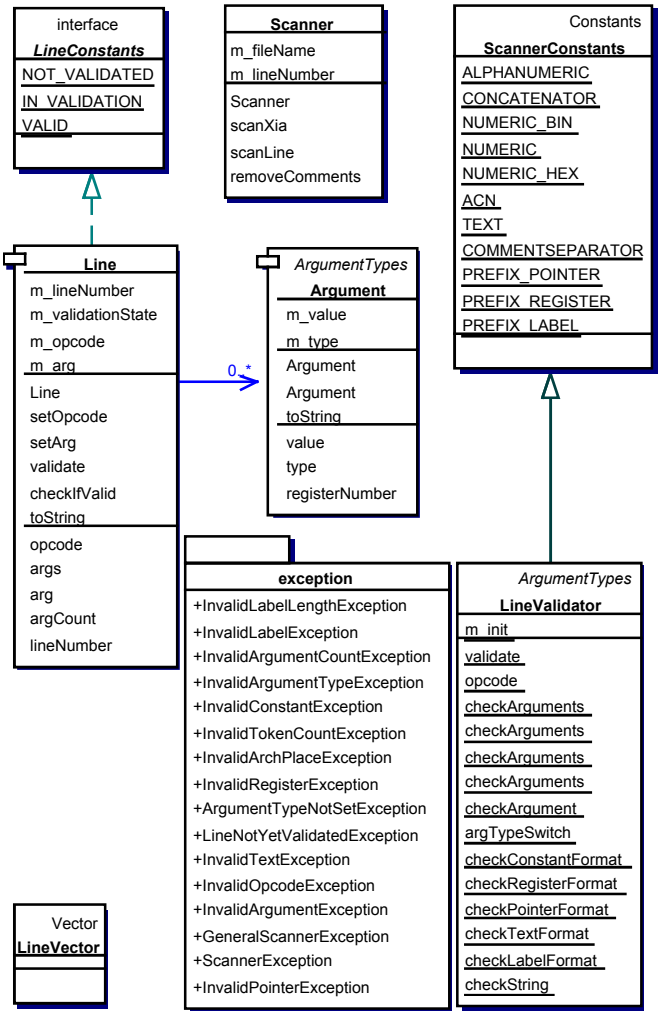

Abbildung 7: Klassendiagramm des Package `scanner`

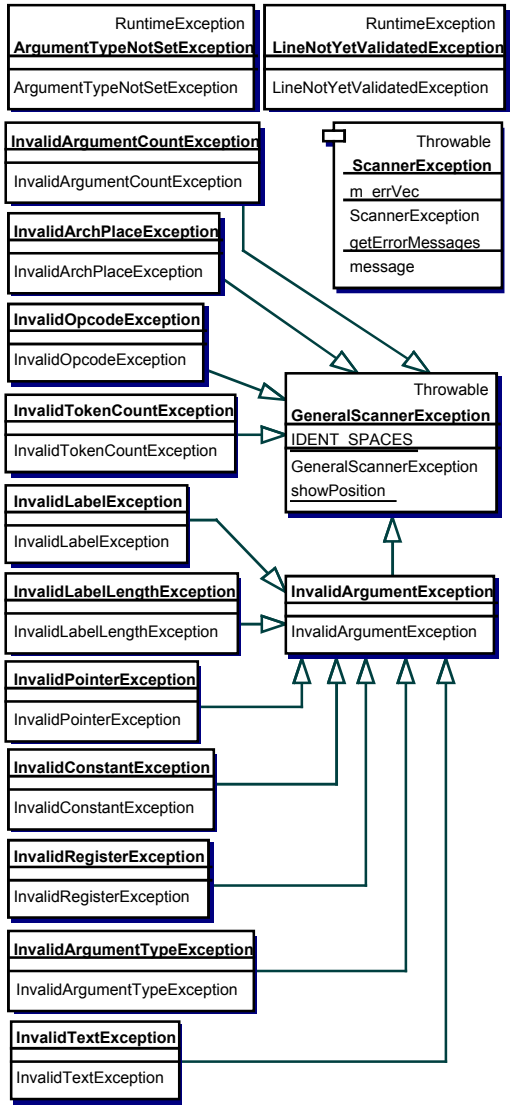

Abbildung 8: Klassendiagramm des Package `scanner.exception`

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.

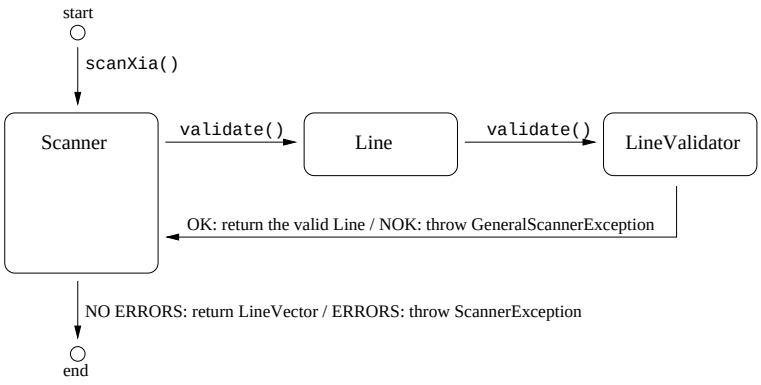

Abbildung 9: Vereinfachter Ablauf der Validierungsphase

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`

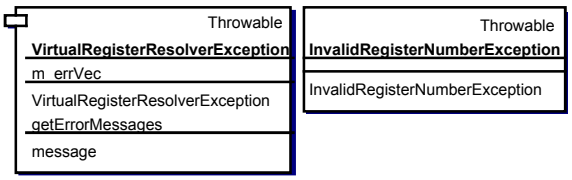

Abbildung 11: Klassendiagramm des Package `resolver.exception`

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 22), 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.

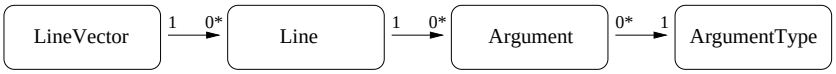

Abbildung 12: Prinzipieller Aufbau eines LineVector

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.

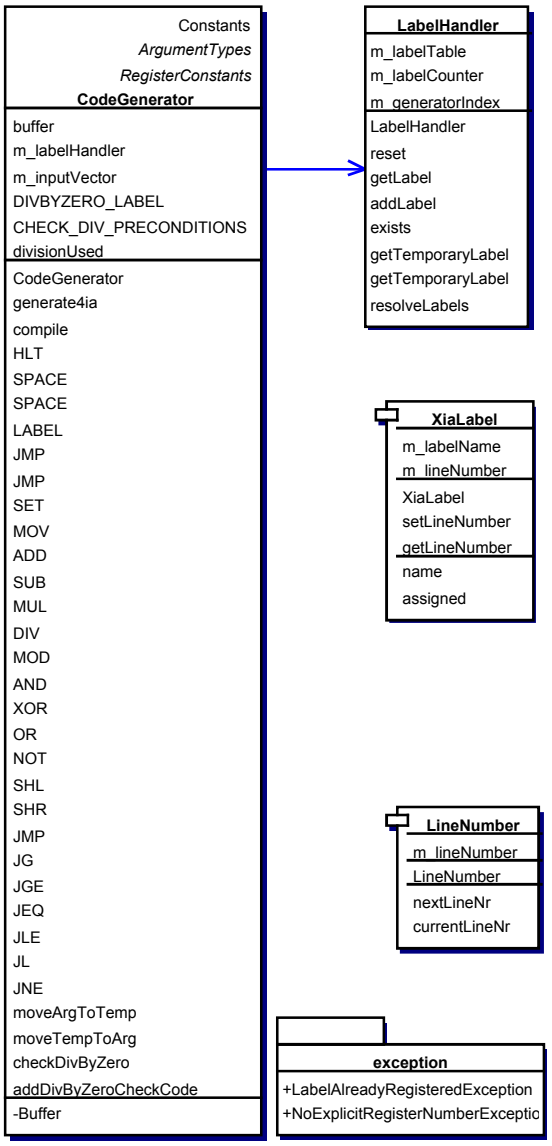

Abbildung 13: Klassendiagramm des Package `compiler`

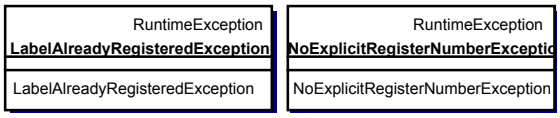

Abbildung 14: Klassendiagramm des Package `compiler.exception`

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 36.

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.

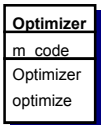

Abbildung 15: Klassendiagramm des Package `optimizer`

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.

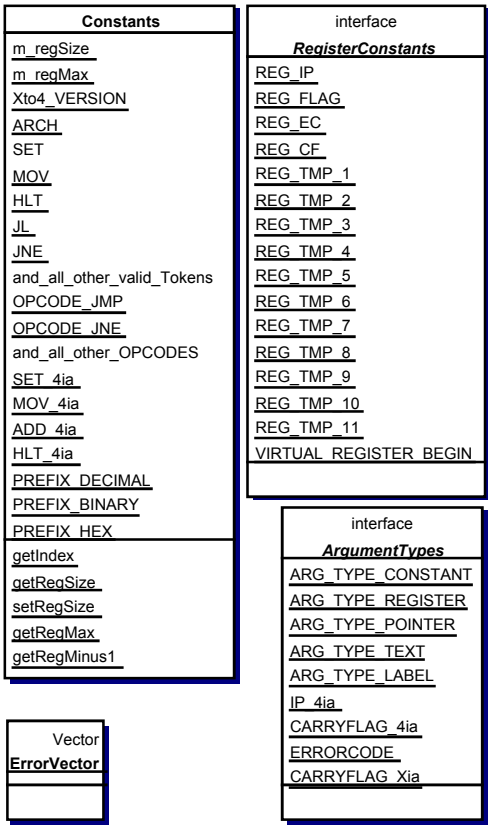

Abbildung 16: Klassendiagramm des Package `global`

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 18 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 18 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.

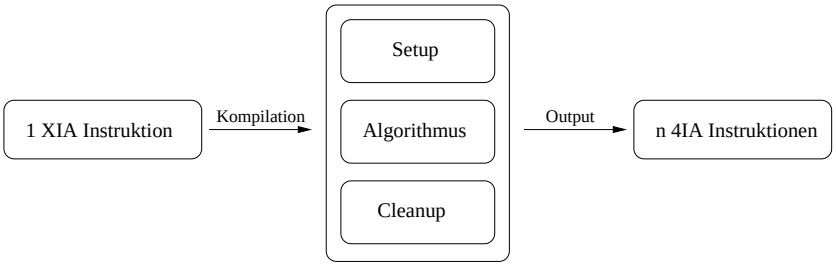

Abbildung 17: Kompilationsschritte

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 46):

```
((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 45) und `XOR` (Seite 46) 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 45) und XOR (Seite 46) 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.

Literatur

[RS02a] Juerg Reusser and Luzian Scherrer. Parasitic computing: Handbuch. `http://parasit.org/documentation`, Dezember 2002.

[RS02b] Juerg Reusser and Luzian Scherrer. Parasitic computing: Pflichtenheft. `http://parasit.org/documentation`, Juni 2002.

[RS02c] Juerg Reusser and Luzian Scherrer. Parasitic computing: Realisierungskonzept. `http://parasit.org/documentation`, Oktober 2002.