PARASITIC COMPUTING
Realisierungskonzept

Version 1.0

Jürg Reusser
Luzian Scherrer

29. Oktober 2002

Zusammenfassung

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

Inhaltsverzeichnis

1 Einleitung

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

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

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

1.1 Umfeld und Motivation

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

1.2 Verwandte Arbeiten

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

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

2 Analyse der drei Hauptkomponenten

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

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

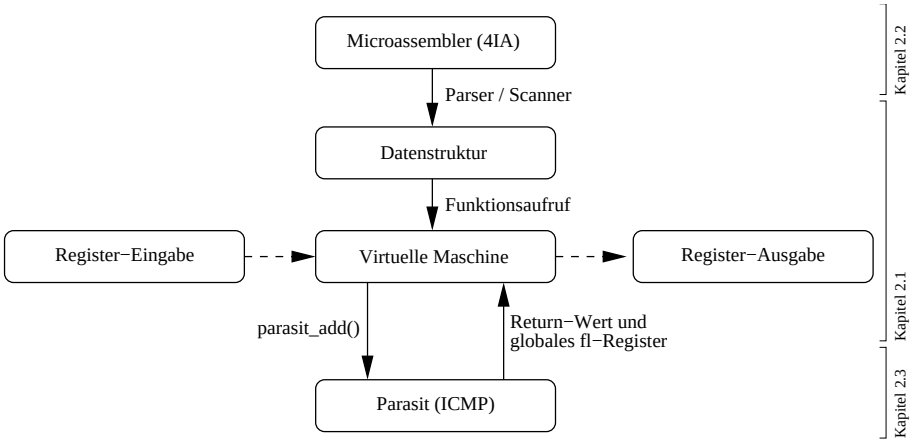

Abbildung 1: Gliederung der Hauptkomponenten und Schnittstellen

2.1 Die virtuelle Maschine

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

2.1.1 Register-Maschine

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

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

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

Tabelle 1: Register der virtuellen Maschine

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

2.1.2 Eingabe - Verarbeitung - Ausgabe

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

2.1.3 Instruktionen

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

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

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

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

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

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

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

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

Tabelle 2: Zusammenfassung der Instruktionen

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

2.1.4 Das Rechenwerk

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

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

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

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

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

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

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

2.1.5 Schnittstellen

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

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

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

2.2 Programmierbarkeit

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

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

2.2.1 Sequenz

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

2.2.2 Wiederholung

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

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

oder

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

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

2.2.3 Auswahl

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

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

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

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

Um so beispielsweise die Auswahl

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

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

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

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

2.2.4 Programmbeispiel: Schlaufe

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

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

2.2.5 Programmbeispiel: indirekte Adressierung

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

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

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

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

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

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

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

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

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

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

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

2.2.6 Weiterführende Arithmetik

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

2.2.7 Binäre boolsche Operationen

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

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

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

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

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

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

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

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

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

a NAND b = (a AND b) XOR 1

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

2.2.8 Höhere Programmiersprachen

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

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

2.3 Parasitäre, verteilte Berechnung

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

2.3.1 Die Internet Checksumme

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

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

Tabelle 3: Berechnung der Internet-Checksumme

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

2.3.2 Addition von Ganzzahlen

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

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

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

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

Tabelle 4: Datenpaket mit Operanden

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

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

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

Tabelle 5: Wahrheitstabelle für die Ganzzahladdition

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

2.3.3 Zeitkomplexität

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

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

2.3.4 Protokoll: ICMP

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

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

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

Tabelle 6: Vollständiges ICMP Paket

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

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

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

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

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

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

2.3.5 ICMP Pakete parallelisieren

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

2.3.6 Empfänger

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

2.3.7 Überwachung und Fehlerkontrolle

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

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

[5] „auspolstern"

2.3.8 Simulationsmodus

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

3 Design Ergebnisse

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

3.1 Systemarchitektur

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

3.1.1 Betriebssystem

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

3.1.2 Programmiersprachen

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

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

3.1.3 Netzwerk

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

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

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

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

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

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

3.2 Module

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

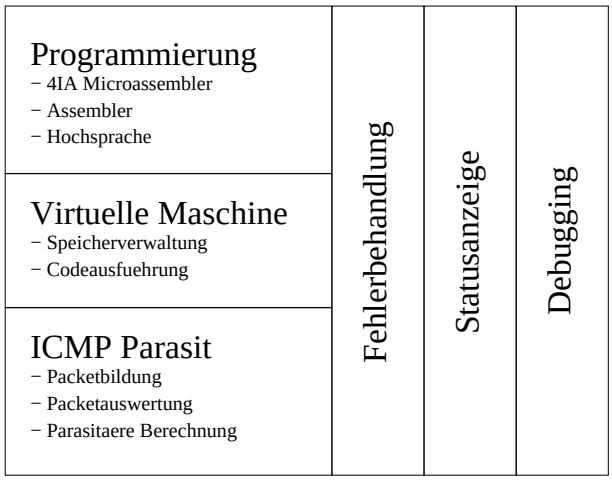

Abbildung 2: Framework der Applikation

4 Realisierungsvorhaben

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

4.1 Priorisierung

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

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

Tabelle 7: Priorisierung der Teilarbeiten

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

4.2 Optionale Ziele

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

- **Höhere Assemblersprache**
  Dieser Punkt wurde bereits in Abschnitt 2.2.8 auf Seite 11 erläutert.
- **Visualisierung**
  Ein graphisches Interface zur Visualisierung des Ablaufes (Registerinhalt, gesendete Netzwerkpakete, etc.), welches mit der virtuellen Maschine beispielsweise über ein Logfile unidirektional oder über einen Socket bidirektional kommuniziert, ist als optionales Ziel denkbar.
- **Hochsprache**
  Dies wurde bereits behandelt; eine Hochsprache zur Programmierung der parasitären Maschine, voraussichtlich auf Basis des YAMCC Paketes aus der Compilerbau Vorlesung der HTA-BE (vgl. [Boi01]), ist als optionales Ziel denkbar.

4.3 Risiken

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

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

Literatur

[BFJB01] Albert-László Barabási, Vincent W. Freeh, Hawoong Jeong, and Jay B. Brockman. Parasitic computing. *Nature*, 412:894–897, August 2001.

[Boi01] Dr. Jacques Boillat. Code generierung. *Script*, 2001.

[Fre02] Vincent W. Freeh. Anatomy of a parasitic computer. *Dr. Dobb's Journal*, 332:63–67, Januar 2002.

[GF89] Network Working Group and Internet Engineering Task Force. Rfc 1123, requirements for internet hosts. `http://www.faqs.org/rfcs/rfc1123.html`, Oktober 1989.

[Gro88] Network Working Group. Rfc 1071, computing the internet checksum. `http://www.faqs.org/rfcs/rfc1071.html`, September 1988.

[oND02] University of Notre Dame. Parasitic computing website. `http://www.nd.edu/~parasite/`, Juni 2002.

[Pos81] Network Working Group (J. Postel). Rfc 792, internet control message protocol. `http://www.faqs.org/rfcs/rfc792.html`, September 1981.

[Sch92] Franz Josef Schmitt. *Praxis des Compilerbaus*. Hanser Studienbuecher der Informatik, Muenchen, Wien, 1992.

[Ste94] Richard W. Stevens. *TCP/IP Illustrated, Volume 1*. Addison-Wesley, 1994.

[TL99] Frank Yellin Tim Lindholm. *The JAVA Virtual Machine Specification*. Addison Wesley, 1999.

[Web02] WebGain. Java compiler compiler - the java parser generator. `http://www.metamata.com/products/java_cc/`, August 2002.