

CoreWars

Quellen:

GUI-Programm:

This document is intended only as a user's guide to the simulator; it does not describe Core War, or how to write Core War programs. For these topics, please refer to the introductory materials at <http://www.corewar.info>, <http://www.koth.org>, and many other Core War sites.

Redcode-Beschreibung:

CoreWar
Krieg der Kerne
Handbuch für Einsteiger.
Sascha Zapf
Version 0.7 Dezember 2002 / Januar 2003

Heimatseite dieses Handbuchs ist www.netcologne.de/~nc-zapfsa

Zusammengefasst von: Dipl.-Inf., Dipl.-Ing. (FH) M. Wilhelm
HS Harz
Fachbereich Automatisierung und Informatik
Veranstaltung: Projektwoche SS 2005

Inhaltsverzeichnis

1	WinCors	5
1.1	Optionen	5
1.2	Breakpoints	6
1.3	Ergebnisse	8
1.3.1	Drei Arten des Spiels	8
1.4	Core View	8
1.4.1	Controls	9
1.5	Processes	9
1.5.1	Controls	10
1.6	Edit Core	10
1.7	P-Space View	11
2	Programmieren in Redcode	12
2.1	Die Operatoren	12
2.1.1	Arithmetische Operatoren	12
2.1.2	Vergleichsoperatoren	12
2.1.3	Logische Operatoren	13
2.1.4	Zuweisung	13
2.2	Die PseudoOpcodes	13
2.2.1	ORG	13
2.2.2	END	13
2.2.3	EQU	13
2.2.4	FOR/ROF	14
2.2.5	PIN-Nummer	15
2.3	Die vordefinierten Konstanten	16
2.3.1	CORESIZe	16
2.3.2	MAXPROCESSES	16
2.3.3	MAXCYCLES	16
2.3.4	MAXLENGTH	16
2.3.5	MINDISTANCE	16
2.3.6	ROUNDS	16
2.3.7	PSPACESIZE	16
2.3.8	CURLINE	17
2.3.9	VERSION	17
2.3.10	WARRIORS	17
2.3.11	Hinweise:	17
2.4	Kommentare	17
2.4.1	;redcode	17
2.4.2	;name	18
2.4.3	;kill	18
2.4.4	;author	18
2.4.5	;date	18
2.4.6	;version	18
2.4.7	;assert	18
2.4.8	;strategy	19
2.5	Die Adressierungsarten	19
2.5.1	Immediate – Absolute Adressierung	19
2.5.2	Direct – relative Adressierung	19
2.5.3	Indirect - Indirekt	20
2.5.4	Predecrement Indirect - Predekremental Indirekt	20
2.5.5	Postincrement Indirect - Postinkremental Indirekt	20
2.6	Die Modifizierer	21
2.6.1	A	21
2.6.2	B	21
2.6.3	AB	21
2.6.4	BA	22
2.6.5	F	22
2.6.6	X	22
2.6.7	I	22
2.7	Die Befehle - Opcodes	23
2.7.1	DAT	23

2.7.2	MOV	24
2.7.3	ADD	24
2.7.4	SUB	25
2.7.5	MUL	25
2.7.6	DIV	25
2.7.7	MOD	26
2.7.8	JMP	26
2.7.9	JMZ	26
2.7.10	JMN	26
2.7.11	DJN	27
2.7.12	CMP	27
2.7.13	SLT	27
2.7.14	SPL	28
2.7.15	SEQ	29
2.7.16	SNE	29
2.7.17	NOP	29
2.7.18	LDP	30
2.7.19	STP	30
3	P-Space	31
4	Die Prozesswarteschlange	33
5	Wie programmiert man einen Kämpfer	35
5.1	Die Trickkiste	35
5.1.1	Parallele Prozesse	35
5.1.2	Anzahl der Prozesse verkleinern	35
5.1.3	djn-Stream	36
5.1.4	Effektive Bomben	36
5.2	Stepsize - Schrittweite	37
5.2.1	Corestep	38
5.2.2	mOpt	40
6	Beispiele	42
6.1	Adressierungsbeispiele	42
6.1.1	Direkte oder absolute Adressierung	42
6.1.2	Relative Adressierung	42
6.1.3	Indirekte Adressierung	43
6.1.4	Predecrement Indirect - Predekremental Indirekt	44
6.1.5	Postincrement Indirect - Postinkremental Indirekt	45
6.2	Einfache Kämpfer	45
6.2.1	Imp	45
6.2.2	Dwarf	45
6.3	Suchen	48
6.3.1	B-Scanner	48
6.3.2	CMP-Scanner	48
6.4	Splitten eines Programms	48
6.4.1	Silk	49
6.4.2	Neue Imp-Version	49
7	Verteidigung	51
7.1	Imp-Gate	51
7.2	Decoy	51
7.3	Bomber-Dodger	51
7.4	Colour	51
7.5	Bootstrapping	52
7.6	Mirror oder Reflection	52
7.7	Stealth	52
7.8	Self splitting	52
7.9	Airbag	53
7.10	Brainwashing	53
8	Links	54
9	Indexverzeichnis	55

Abbildungen

Abbildung 1	Lauf eines Kampfes mit mehreren Viren.....	5
Abbildung 2	Anzeigemodi.....	5
Abbildung 3	Core-View	6
Abbildung 4	Liste aller definierten Breakpoints.....	6
Abbildung 5	Beispiel eines Breakpoints.....	7
Abbildung 6	Ergebnisse einer Runde	8
Abbildung 7	Anzeige eines Core-View	9
Abbildung 8	Eigenschaftsfenster eines Befehls.....	9
Abbildung 9	Anzeige eines Kämpfers im Prozessfenster	10
Abbildung 10	Anzeige von Befehlen im Editor	11
Abbildung 11	P-Spave Anzeige.....	11
Abbildung 12	Bearbeiten des P-Spaces eines Prozesses	11
Abbildung 13	Beispiel der direkten Adressierung.....	42
Abbildung 14	Beispiel einer indirekten Adressierung im Core	43
Abbildung 15	Indirekte Adressierung über das A-Feld.....	44
Abbildung 16	Dwarf im Core	46
Abbildung 17	Ergebnis des ersten Dwarfs.....	47

1 WinCors

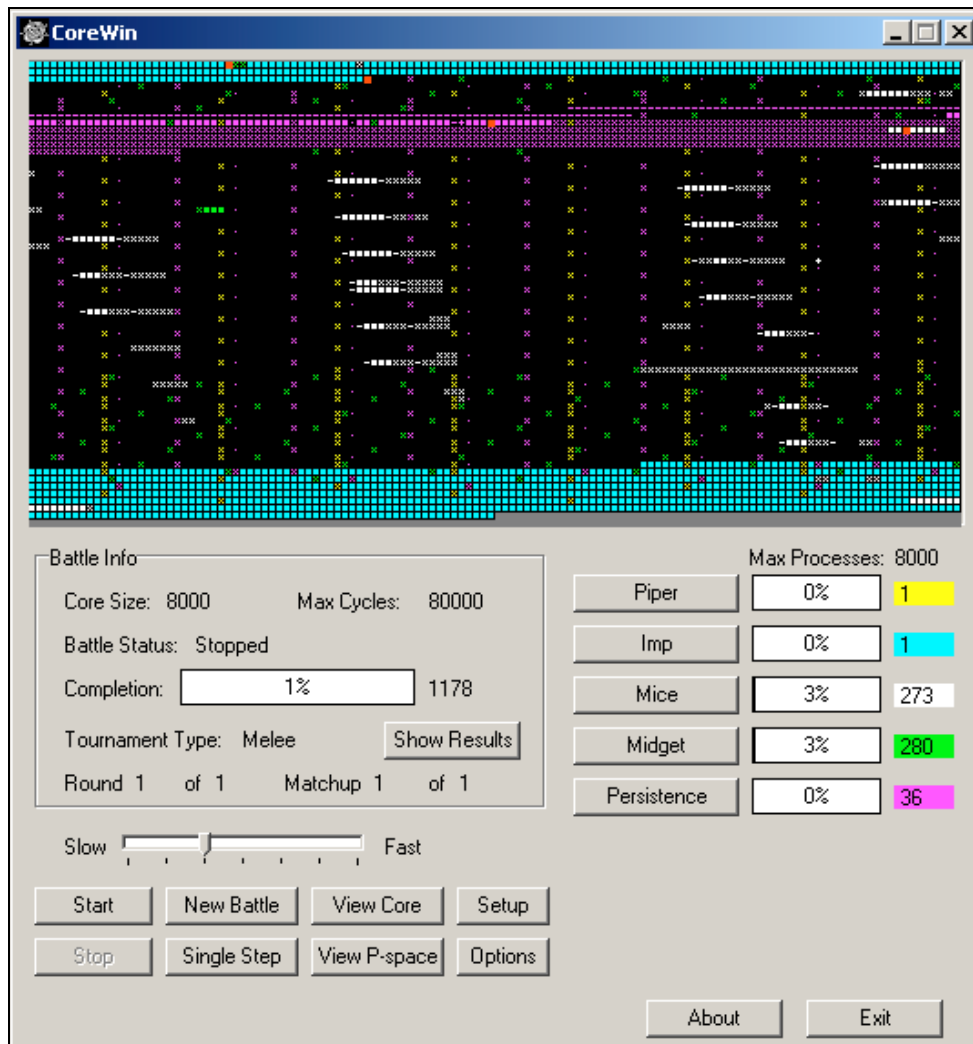
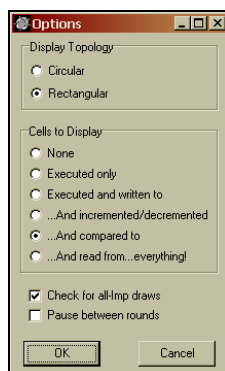


Abbildung 1 Lauf eines Kampfes mit mehreren Viren

1.1 Optionen

Dieses Dialogfenster zeigt die Anzeige-Einstellungen des „Runmoduls“.



Calls to Display

- Nichts
- Beim Ausführen
- Ausführen und Schreiben
- + Addieren, Subtrahieren (Inc, Dec)
- + Vergleich (Compare)
- Immer, wenn eine Zelle adressiert wird

Abbildung 2 Anzeigemodi

1.2 Breakpoints

Ein Breakpoint kann in jeder Anweisung mit optionalen Bedingungen gesetzt werden. Das Core-Fenster erhält man mit dem Schalter „View Core“.

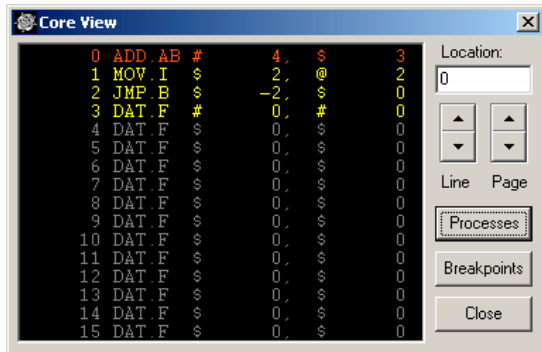


Abbildung 3 Core-View

Mit den Schalter "Line" und "Page" kann der Ausschnitt verändert werden. Der Schalter "Process" zeigt ein Fenster, in der alle "Kämpfer" mit ihren jeweiligen Teilprozessen dargestellt sind.

Der Schalter "Breakpoints" erlaubt die Definition von Haltepunkten.

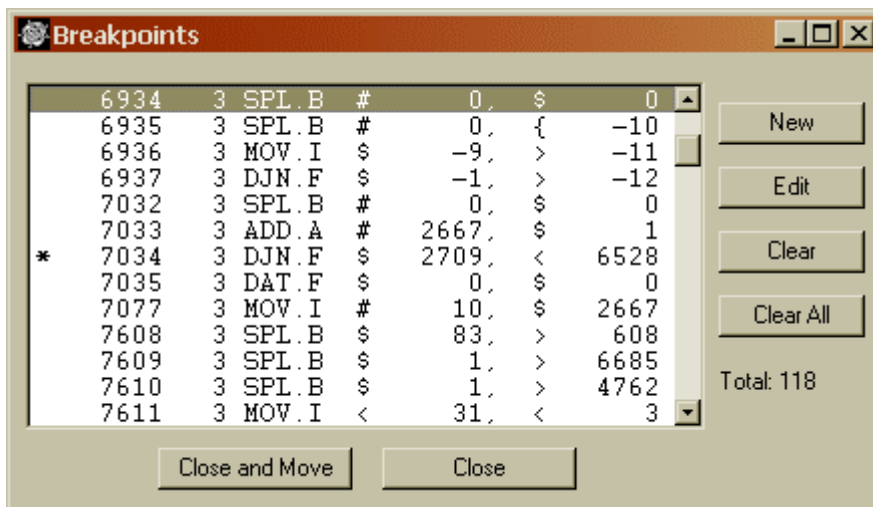


Abbildung 4 Liste aller definierten Breakpoints

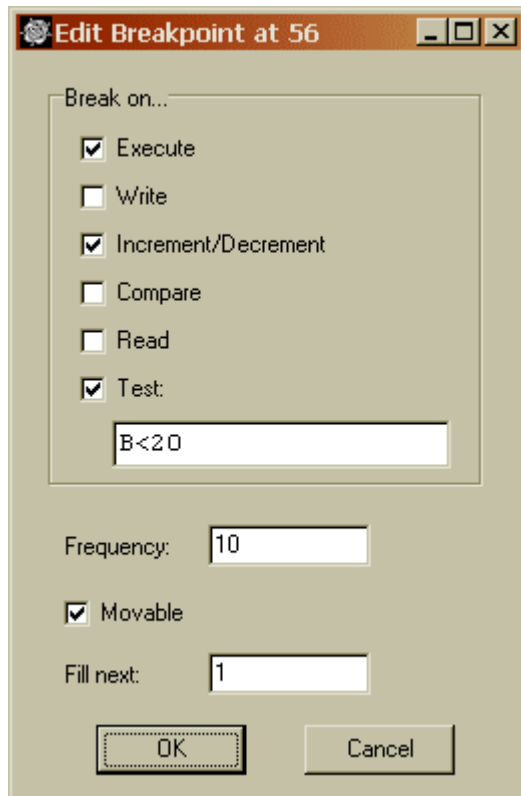


Abbildung 5 Beispiel eines Breakpoints

Der Inhalt der Frequenz erlaubt die Steuerung, wie oft ein Breakpoint gesetzt werden soll. Mit dem Wert Null, wird der Breakpoint immer ausgeführt.

Der Schalter „Moveable“ erlaubt das „Kopieren“ des Breakpoints

Breakpoints können mit vier Compiler Direktiven gesetzt werden:

- `;debug [static | off]` – Erlaubt das Setzen oder Deaktivieren der Breakpoints. Mit dem Attribut „static“ werden alle „beweglichen“ Breakpoints fest.
- `;trace [off]` – Mit „trace“ werden alle Haltepunkte erlaubt. Mit „trace off“ werden die Haltepunkte ignoriert.
- `;condition [EWICRT] [“<condition>”] [M | F] [<frequency>]` – Setzt den Defaultwert für alle Breakpoints. Jede Kombination mit den Buchstaben „E, W, I, C, R und T“ ist erlaubt. Ohne Angabe ist der Default-Wert der Fall „E“.

In der Eingabezeile können alle Ausdrücke eingetragen werden. Die Buchstaben A und B dienen als Referenz für beiden Operanden. Die Standardbedingung ist „B==0“. Mit Hilfe dieser Anweisung kann der Ablauf angehalten werden.

M oder F spezifiziert einen beweglichen oder festen Haltpunkt. Der Defaultfall ist „M“.

- `;break [EWICRT] [“<condition>”] [M | F] [<frequency>]` – Setzt einen Haltpunkt mit der eingegebenen Bedingung auf die nächste Anweisung. Wenn keine Bedingung eingegeben wurde, wird die letzte gültige genommen oder „E M 1“.

Beispiel:

```
;break WT "A<20" F 10
```

Diese Anweisung definiert einen Haltpunkt, wenn der nächste Befehl geschrieben wird, oder der A-Operand kleiner zwanzig ist. Ausgeführt wird der Breakpoint nur jedes zehnte Mal. Eine alternaive Schreibweise wäre:

```
;break fw "a < 20" 10t
```

Breakpoints können natürlich auch mittels des Dialogfenster eingefügt werden.

Es gibt zwei vordefinierte „breakpoint type“. Typ eins hält bei Ausführung, ist fest und hat eine Frequenz von eins (E F 1). Der Typ zwei unterscheidet sich nur darin, dass er bewegbar - movable – ist (E M 1).. Jeder andere Haltepunkt hat eine größere Nummer (ab drei). Insgesamt dürfen es aber maximal 255 Haltepunkte eingesetzt werden.

1.3 Ergebnisse

Mit dem Schalter „Sow Results“ werden die Ergebnisse dargestellt. Die Checkbox „Show W%, L%, T%“ zeigt die Daten in prozentualer Aufteilung.

ID	Warrior	Length	Score	Given	W%	L%	T%
5	Reepicheep	0.54414	159.8	98.5	39.4	18.9	41.7
9	Son of Vain	0.59732	152.3	102.3	35.6	18.9	45.5
10	Uninvited	0.50032	146.2	116.7	36.4	26.5	37.1
12	Willow	0.37467	144.7	142.4	43.9	43.2	12.9
4	nPaper II	0.62935	137.9	115.2	30.3	22.7	47.0
6	Revenge of the Papers	0.62776	137.1	118.9	31.1	25.0	43.9
7	RotF Copy	0.78049	127.3	100.0	18.2	9.1	72.7
11	Vanquisher II	0.46675	122.0	149.2	31.1	40.2	28.8
2	CrazyShot 2	0.45654	121.2	157.6	33.3	45.5	21.2

Abbildung 6 Ergebnisse einer Runde

Je nach Typ des Spiels, erhält man detailliertere Tabellen.

1.3.1 Drei Arten des Spiels

- Melee: Alle kämpfen gleichzeitig gegeneinander
- Round-Robin: Jeder gegen jeden (N*N-1) Kämpfe
- Challenge: Ein Virus kämpft nacheinander gegen alle

1.4 Core View

Dieses Dialogfenster zeigt die Befehle Core-View bzw. im „Hauptspeicher“. Ein Befehl, der ausgeführt wird, erscheint in der Farbe rot. Leere Befehle erscheinen dagegen grau. Da das Fenster „nicht-modal“ ist, kann es während der Berechnung offen bleiben. Der Inhalt des Fensters wird in regelmäßigen Abständen erneuert. Mit jedem Klick auf den Schalter „View Core“ wird ein neues Fenster erzeugt (MVC-Architektur).

Befehle mit einem Haltepunkt haben ein Symbol am linken Rand. Breakpoints mit einem Ausführungsstopp haben die Farbe rot, alle anderen die Farbe grün. Quadrate symbolisieren den Typ 1 (E F 1), Kreise den Typ 2 (E M 1) und Rauten alle anderen.

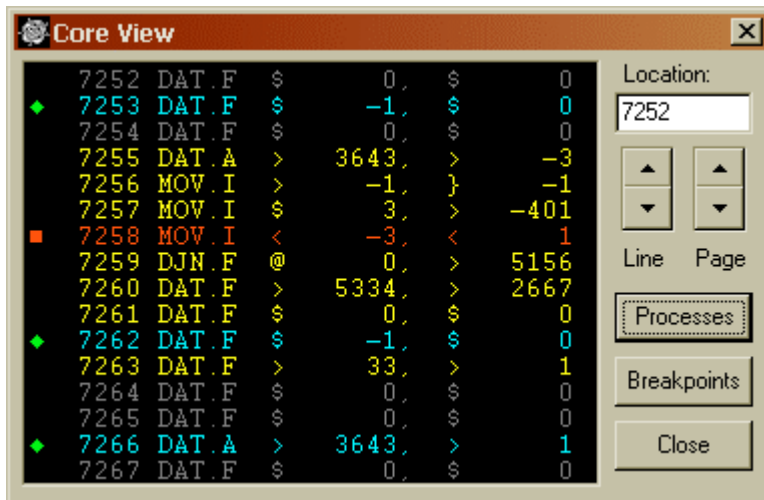


Abbildung 7 Anzeige eines Core-View

1.4.1 Controls

Die „spin controls“ bewegen die aktuelle Zeile oder Seite. Entspricht den Pfeilen bzw. den Tasten „PgUp, PgDn“. Um eine Zeile direkt anzuspringen, gib man die Zeile in das Editorfeld (Dezimal oder \$Hexadezimal) und betätigt die Tab-taste“.

Ein Mausklick auf das Symbol oder die Zahl wechselt zwischen den drei Haltepunkt-Typen. Ein Doppelklick auf einem Befehl öffnet ein separates Fenster, in dem den Befehl bearbeiten kann.

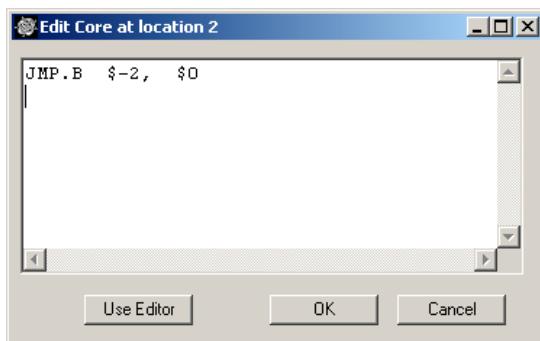


Abbildung 8 Eigenschaftsfenster eines Befehls

1.5 Processes

Dieses Fenster zeigt alle Teilprozesse aller Kämpfer an. In der „ComboBox“ in der Mitte kann man den Kämpfer auswählen. Pro Kämpfer wird der nächste Teilprozess angezeigt. Dieses kann man aber mit den Schalter „Move Up“ und „Move Down“ ändern. Abbildung 9 sind 21 Teilprozesse aufgelistet. Jeder Teilprozess hat eine andere Adresse und einen Befehl. Mit dem Schalter „Roll to Top“ können Prozesse an den Anfang des „Taskmanagers“ gelanden.

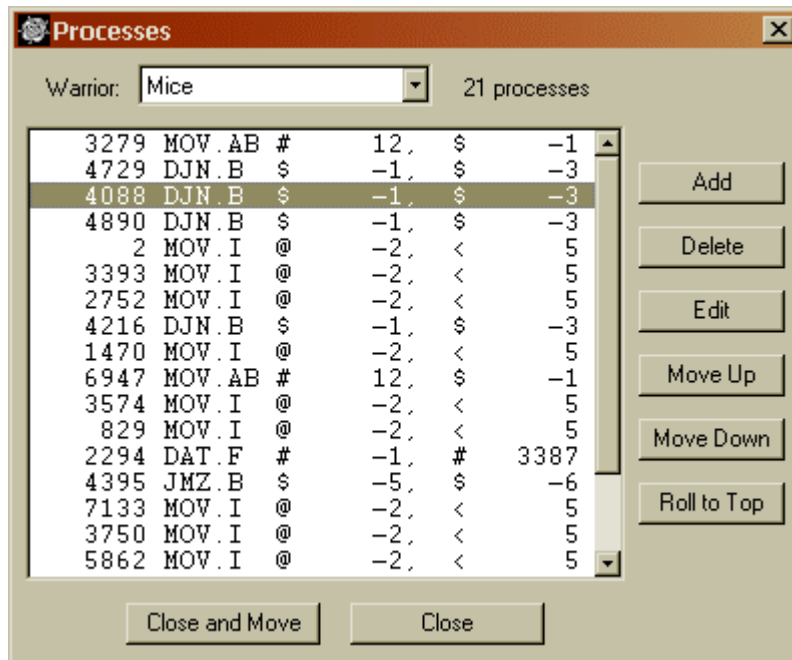


Abbildung 9 Anzeige eines Kämpfers im Prozessfenster

1.5.1 Controls

Erklärung der Schalter:

Add:

Erzeugt einen neuen Prozess mit einer Startadresse. Der neue Prozess wird direkt nach dem aktuellen ausgeführt.

Delete:

Löscht den selektierten Teilprozess.

Edit:

Setzt den Befehlszähler des aktellen Prozess auf einen Wert.

Move Up:

Der aktuelle Teilprozess wird um eine Position nach oben geschoben. Damit wird er früher ausgeführt.

Move Down:

Der aktuelle Teilprozess wird um eine Position nach unten geschoben. Damit wird er später ausgeführt.

Roll to Top:

Der aktuelle Prozess wird an die Spitze der Ausführungsliste gesetzt. Wichtig dabei, die relative Reihenfolge der Teilprozesse bleibt dabei erhalten.

Close:

Entspricht den Abbruch-Schalter

Close and Move:

Entspricht den Ok-Schalter. Alle vorgenommenen Änderung werden in den „Core-View“ eingetragen.

1.6 Edit Core

Diese Dialogfenster zeigt den aktuellen Befehl. Dabei kann dieser durch gültige „Befehle“ geändert werden. Dies schliesst Schleifen etc. mit ein. Mit Betätigen des Schalters „Ok“ wird der Quellcode übersetzt und ggfs wird eine Fehlermeldung ausgegeben. Mit dem Schalter „Use Editor“ kann der große GUI-Editor aufgerufen werden.

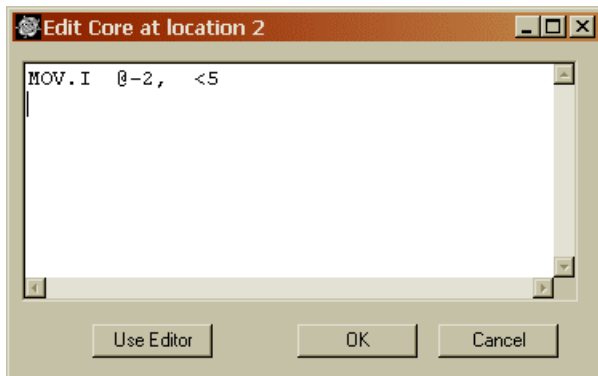


Abbildung 10 Anzeige von Befehlen im Editor

1.7 P-Space View

Der P-Space ist ein Speicherbereich, auf dem alle Kämpfer zugreifen können. Jeder Kämpfer hat seinen eigenen Speicherbereich. Ein Kämpfer kann aber auch mit Hilfe einer ID einen Speicherplatz zugeordnet bekommen. Diese ID ist aber nicht eindeutig, so dass auch andere Kämpfer auf diesen Speicher zugreifen können. Der Speicher kann benutzt werden, um ermittelte Information von einer Runde in die nächste zu retten.

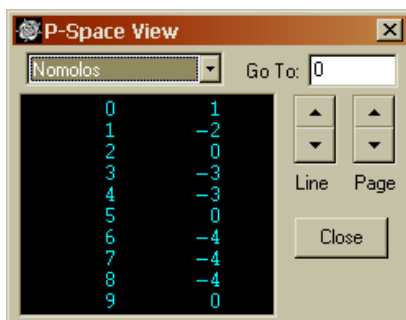


Abbildung 11 P-Spave Anzeige

Mit einem Doppelklick auf eine Zeile, öffnet sich ein neues Dialogfenster, siehe Abbildung 12, in dem Änderung vornehmen kann. Dabei bedeutet der Eintrag „Fill Next“, wieviele Inhalte überschrieben werden sollen.

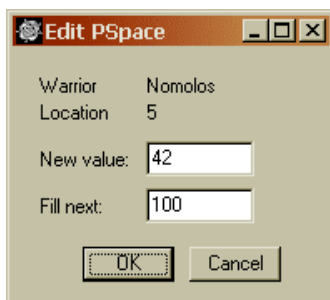


Abbildung 12 Bearbeiten des P-Spaces eines Prozesses

Siehe auch Kapitel 2.2.5, Seite 15.

2 Programmieren in Redcode

Dieses Kapitel erläutert detailliert die Programmierung in Redcode, der Assemblersprache des CoreWars. Durch 19 Instruktionen, 8 Adressierungsarten und 7 verschiedenen Modifizierer ist Redcode eine Sprache mit der man fast alles ausdrücken kann was ein Kämpfer können sollte. Es gibt sicherlich Befehle die noch vermisst werden, aber ich denke die wichtigsten sind vorhanden. Groß/Kleinschreibung wird bei den Befehlen und ihren Modifizierern ignoriert. Wie man schreibt ist einem selbst überlassen. Im letzten Teil dieses Kapitel werde ich ein gewissen Standard ansprechen der sich im Laufe der Zeit verbreitet hat. Einen Standard zu unterstützen trägt dazu bei, dass die Programmierer den Quellcode besser verstehen

Wichtig:

Bei Labeln wird die Groß/Kleinschreibung allerdings beachtet. Damit sind folgende Label alle verschieden.

- Label mov 0,3
- label mov 0,3
- LABEL mov 0,3

Wenn ein Label in einem Operand eingesetzt wird, so wird die Bezeichnung durch die Anzahl der Befehle ersetzt, die zwischen diesen Anweisungen sind.

Beispiel:

```
Dest    DAT #0,    #Source
Source  MOV $Source, $Dest
```

Wird übersetzt nach

```
DAT #0,    #1
MOV $0,    $-1
```

Ich warne dennoch ausdrücklich davor Label zu nutzen die sich nur durch ihre Groß- und Kleinschreibung unterscheiden. Situationen wie: "Habe ich die dat-Bombe jetzt Bomb, bomb oder BOMB genannt", halten den Programmierer oft sekundenlang auf. Meist wird er nach oben scrollen um nachzuschauen. In der Zeit kann schon wieder ein genialer Gedankengang verloren sein.

Es ist wichtig die Formulierungen genau durchzulesen. Manchmal bemerkt man sofort, wofür man einen Befehl einsetzen kann. Gerade wenn bei der Beschreibung der Befehle, ist es ganz gut wenn man die verschiedenen Modifizierer und Adressierungsarten schon kennen gelernt hat. Auf sie wird in den Befehlen nur eingegangen, wenn sich Besonderheiten zeigen.

2.1 Die Operatoren

Folgende Operatoren können benutzt werden.

2.1.1 Arithmetische Operatoren

```
+    Addition
-    Subtraktion
*    Multiplikation
/    Division
%    Modulo - Rest
```

2.1.2 Vergleichsoperatoren

```
=    gleich
!=   nicht gleich
<    kleiner als
>    größer als
<=   kleiner oder gleich
>=   größer oder gleich
```

2.1.3 Logische Operatoren

```
&&  UND
||   ODER
!    NICHT
```

2.1.4 Zuweisung

= Zuweisung, Breakpoints

2.2 Die PseudoOpcodes

PseudoOpcodes sind Anweisungen die den Quellcode beeinflussen, aber nicht im fertigen Code stehen. Sie werden während der Assemblierung in Adressen oder in Quelltext umgewandelt.

2.2.1 ORG

ORG Adresse

Die Adresse oder das Label hinter dem ORG-Opcode bezeichnet die Stelle, an der der erste auszuführende Befehl des Kämpfers liegt. Sie kann mitten im Quellcode liegen. Sollte diese Angabe fehlen, werden die Daten hinter dem end-PseudoOpcode genommen.

2.2.2 END

END [Adresse o. Label]

Hiermit wird das Ende des Quellcode signalisiert. Optional kann hier ebenfalls die erste auszuführende Anweisung eingetragen werden. Sollte hier ebenfalls keine Adresse oder kein Label angegeben werden, so wird zur ersten Anweisung im Quelltext gesprungen. Achtung, sollte bei ORG und END jeweils eine Adresse stehen, so wird die von ORG genommen.

2.2.3 EQU

text EQU ersatztext

Textuelle Ersetzung des Labels „text“ durch den Text „ersatztext“.
Beispiel: Gate EQU Start-12

Ersetzt jedes im Text vorkommene Wort „Gate“ durch den Text „Start-12“.

Hinweis:

Die Textuelle Ersetzung bringt auch einige Gefahren mit sich. Will man den Abstand des Imp-Gates zum Kämpfer auf diese Art verdreifachen, so könnte man schreiben:

```
Gate EQU Start-12
start NOP #0, #0
NOP
JMP start, <3*Gate           ; <--- 3*Gate wird (3*Start)-12
```

Dann wird das Gate nicht $3*(2-12) = -14$ Positionen, sondern $3*2-12 = -6$ Positionen nach hinten gelegt. In bestimmte Fällen kann das Gate dann auch im Kämpfer liegen. Da Punkt vor Strichrechnung geht, sollte man solche zusammengesetzten Werte immer Klammern:

Abhilfe:

Gate EQU (Start-12)

Es können auch mehrere Zeilen benannt werden. Folgende Zeilen erzeugen 12 Instruktionen langen „nicht-null-B-Feld Decoy“.

```
d4 EQU dat #0, #12
dat #0, #23
dat #0, #34
dat #0, #45
...
...
...
decoy d4
d4
d4
```

2.2.4 FOR/ROF

FOR/ROF ist eine sehr mächtige Version von EQU. Folgender Quellcode,

```
FOR 3
  dat #0, #0
ROF
```

erzeugt drei „dat #0, #0 Instruktionen“. Des Weiteren ist es auch möglich, eine Zählervariable anzugeben, die dann durch das &-Zeichen mit dem zu erzeugenden Quelltext verbunden wird. Das kann auch dazu benutzt werden automatisch generierte Sequenzen mit einem Label anzuspringen.

Gegebenist die Sequenz.

```
N FOR 5
  slab&N spl lab&N,&N*5
ROF
```

In einem Beispiel:

```
bomb dat 0,0
I FOR 5
lab&I mov bomb, <slab&I
jmp lab&I, <slab&I
ROF
```

Vor der eigentlichen Assemblierung wird sie zu folgendem Quelltext ausgewertet.

```
slab01 spl lab01, 5
slab02 spl lab02, 10
slab03 spl lab03, 15
slab04 spl lab04, 20
slab05 spl lab05, 25
bomb dat 0,0
lab01 mov bomb, <slab01
jmp lab01, <slab01
lab02 mov bomb, <slab02
jmp lab02, <slab02
lab03 mov bomb, <slab03
jmp lab03, <slab03
lab04 mov bomb, <slab04
```

```

jmp lab04, <slab04
lab05 mov bomb, <slab05
jmp lab05, <slab05

```

Da die Laufvariable immer auf die gleiche Weise an das Label angefügt wird kann man die Werte auch „importieren“.

Folgende Sequenz:

```

step01 EQU 3044
step02 EQU 1644
step03 EQU 1704
step04 EQU 928

N FOR 4
mov.i datb, step&N
add.ab #step&N, -1
ROF

```

Wird so erweitert.

```

mov.i datb, 3044
add.ab #3044, -1
mov.i datb, 1644
add.ab #1644, -1
mov.i datb, 1704

add.ab #1704, -1
mov.i datb, 928
add.ab #926, -1

```

FOR/ROF Blöcke können auch geschachtelt werden. Um den 12 Instruktionen „nicht-null-B-Feld Decoy“ von eben zu erzeugen, kann man folgenden Block benutzen.

```

N FOR 3
  I FOR 4
    dat #0, #&I+11
  ROF
ROF

```

Die FOR/ROF PseudoOpcodes sind nicht Teil des 94er Standard. Da sie aber im Quasistandard, der pMars enthalten sind, können sie bedenkenlos genutzt werden.

2.2.5 PIN-Nummer

PIN Nummer

Dieser PseudoOpcode benennt den pSpace eines Kämpfer mit einer Nummer. Kämpfer mit der gleichen PIN (PSpace Identification Number) teilen sich einen pSpace. Dieser PseudoOpcode ist ebenfalls nicht im 94er Standard enthalten. Er wurde aber mit der „pMars 0.8“ als Quasistandard eingeführt, und kann deshalb auch auf den 94er Hill's, mit Ausnahme der No-pSpace-Hill's benutzt werden.

Beispiel:

```
PIN 3
```

2.3 Die vordefinierten Konstanten

Die folgende Konstanten sind im Redcode nutzbar. Sie werden beim Assemblieren durch die tatsächlichen Werten ersetzt.

2.3.1 CORESIZE

Zeigt die Größe des Core's an. In einem Kampf gibt es keine größere Zahl als CORESIZE-1. Der Core fängt bei Null an! Daran sollte man denken wenn man Werte in Befehlen definiert, die mit Core's in unterschiedlichen Größen des Core's agieren sollen.

2.3.2 MAXPROCESSES

Bestimmt die maximale Anzahl der Prozesse eines Kämpfers.

2.3.3 MAXCYCLES

Bestimmt die Anzahl der Instruktionen, die jeder Kämpfer ausführen darf.

2.3.4 MAXLENGTH

Bestimmt die maximale Länge eines assemblierten Kämpfers.

2.3.5 MINDISTANCE

Dieser Wert ist die kleinste Entfernung, die Kämpfer zu Beginn eines Kampfes von einander haben dürfen. Diese Konstante wird von der MARS gebraucht. Alle Kämpfer werden zu Beginn des Kampfes an zufällige Positionen im Core's gesetzt. Dieser „Sicherheitsabstand“ wird aber auf jeden Fall eingehalten. Der Abstand wird vom tatsächlichen Anfang und Ende des Kämpfers im Core gemessen. Wenn man gegen Imp's kämpft, dann kann man zu Beginn des Kampfes eine Menge Freiraum lassen, der nur mit dat-Instruktionen gefüllt ist. Das und MINDISTANCE geben dann ein bisschen Zeit.

2.3.6 ROUNDS

Wert für die Anzahl der Runden bei Mehr-Runden-Kämpfen.

2.3.7 PSPACE SIZE

Größe des pSpace.

2.3.8 CURLINE

Aktueller Befehl im vorassemblierten Kämpfer (Current Line)

2.3.9 VERSION

Die Version der pMars*100: 60 ist Version 0.6.0

2.3.10 WARRIORS

Zeigt die Anzahl der Kämpfer.

2.3.11 Hinweise:

Die bekannteste Anwendung der Konstante CORESIZE ist wohl die in der Assert-Zeile jedes Kämpfers. Neben der Assert-Zeile können weitere Konstanten in Instruktionen eingesetzt werden. Beispielsweise in einer cmp-Anweisung eines „On-axis cmp-Scanner’s“:

```
cmp.f scanpos,scanpos+CORESIZE/2
```

Es ist für einen Dwarf unnötig, MINDISTANCE Instruktionen vor oder nach sich mit Bomben zu bewerfen. Jedenfalls zu Beginn des Kampfes. Deshalb könnte man die Position der ersten Bombe mit:

```
dat #0,#-MINDISTANCE-5
```

definieren.

2.4 Kommentare

Kommentare sind ein leidiges Thema, über das schon viel diskutiert worden ist. Man kann während der Entwicklung praktisch Notizen im Redcode hinterlassen, um später bestimmte Tricks sofort wiederzufinden. Aber das ist natürlich jedem selbst überlassen. Ein Kommentar wird mit den Semikolon eingeleitet. Alles was dahinter steht wird ignoriert. Das Ende der Zeile ist auch das Ende des Kommentars. Anders verhält es sich aber mit den besonderen Kommentaren in einer Redcode-Datei.

Beispiel:

```
loop  ADD #step,bomb      ; Alles was hier steht ist Kommentar
      MOV bomb,@bomb      ; Hier könnte auch die ein oder andere jmp loop ; nützliche Info stehen
      bomb dat #0,#0      ; Die neue Bombe (war spl 0 ) - z.B.
```

Mit etwas Mehrarbeit kann man sich hier einiges an Denkarbeit sparen, wenn man Teile des doch so erfolgreichen Kämpfers später in einem neuen Kämpfer „wiederverwerten“ möchte. Wenn man bei der Entwicklung gewisse Handgriffe gut kommentiert, dann können auch Andere daraus lernen.

Hinweis:

Besondere Kommentare werden durch das Semikolon und eines der folgenden Schlüsselwörter eingeleitet.

2.4.1 ;redcode

Hiermit beginnt der Quelltext eines Kämpfers. Für den Fall, dass dieser per eMail übermittelt wird, kann man noch begleitenden Text davor schreiben. Alles was vor dieser Zeile steht, wird bei der Assemblierung des Kämpfers ignoriert. Mit dieser Zeile wird zusätzlich gesteuert, auf welchen Hill der Kämpfer gesetzt werden soll. Aber dazu später mehr.

2.4.2 ;name

Der Name eines Kämpfers ist nicht zu unterschätzen. Ein guter Name stösst dem gegnerischen Programmierer Respekt ein. Das wird sich bemerkbar machen, wenn er Verteidigungsstrategien gegen diesen Kämpfer entwickeln muss.

2.4.3 ;kill

Schleust man einen Kämpfer auf einen Hill, auf dem sich noch ältere Versionen des Kämpfers befinden, so kann man den Hill dazu auffordern, die älteren Versionen zum Ende der Runde zu entfernen. Sobald dieser Kämpfer den Run-Modus erreicht hat, bekommt der zu löschende Kämpfer keine Punkte mehr gutgeschrieben. Dadurch fliegt er automatisch vom Hill.

2.4.4 ;author

Hier kann sich der Programmierer verewigen.

2.4.5 ;date

Das Schlüsseldatum der Erstellung des Kämpfers. Man kann hier auch den Text @date@ eingeben, dann wird automatisch beim Server des Hill's das aktuelle Datum eingetragen.

2.4.6 ;version

Die Versionsnummer des Kämpfers, nützlich bei Überarbeitungen.

2.4.7 ;assert

Dieser Ausdruck prüft, ob ein Kämpfer in einer Umgebung kämpfen kann. Logisch „Wahr“ heisst, dass der Kämpfer für den aktuellen Core geeignet ist. Bei „Falsch“ wird der Kämpfer nicht zum Kampf zugelassen.

Beispiele:

Der Imp, der in jeder Umgebung kämpfen könnte, signalisiert das mit immer Wahr.

```
;assert 1  
oder  
;assert 0==0
```

Der Dwarf, der sich bei Größen des Core's die sich nicht durch 4 teilen lassen selber bombardieren würde, könnte das mit folgender Anweisung überprüfen:

```
assert CORESIZE%4==0
```

Eine „Imp-Spirale“ ist meist auf eine bestimmte Größe des Core festgelegt.

```
; assert CORESIZE==8000
```

2.4.8 ;strategy

In einer oder mehreren Strategie-Zeilen kann man die Art des Angriffs und wenn man will, auch die Art der Verteidigung beschreiben. Dabei werden aber keine Implementationsdetails, sondern nur die Kategorien angegeben. Je sicherer man sich bei seinem Kämpfer ist, umso mehr kann man über ihn verraten.

Beispiel:

```
;name Dwarf
...
...
...
;strategy Stone with .5c-Bomber and 100%c Imp-Gate
```

2.5 Die Adressierungsarten

Die Adressierungsarten bestimmen, wie die Daten in A-Feld und B-Feld zu behandeln sind. Sie bestimmen, mit welcher Adresse die Instruktion tatsächlich arbeitet.

Insgesamt existieren acht Adressierungsarten:

- # -- immediate
- \$ -- direct (the \$ may be omitted) , relative Adressierung
- * -- A-field indirect
- @ -- B-field indirect
- { -- A-field indirect with predecrement
- < -- B-field indirect with predecrement
- } -- A-field indirect with postincrement
- > -- B-field indirect with postincrement

2.5.1 Immediate – Absolute Adressierung

Der Wert um den es geht, ist direkt im Feld abgelegt. Die Adresse, die zu bearbeiten ist, wird auf 0 gesetzt. Dadurch wird der angegebene Wert benutzt. Die unmittelbare Adressierung wird durch # gekennzeichnet.

Beispiel:

```
bomp DAT #0, #0
label ADD #2, bomp
      JMP label
```

Der Additions-Befehl addiert zur Zelle „bomp“ jeweils den Wert zwei.

2.5.2 Direct – relative Adressierung

Die Adresse wird direkt angegeben (Zahl von 0 bis Coresize-1). Die unmittelbare Adressierung wird durch \$ gekennzeichnet. Sollte bei der Angabe eines Feldes keine Adressierung angegeben werden, dann wird die direkte Adressierung benutzt.

Beispiel:

```
; Direkte Adressierung
ORG label
      bomp DAT #0, #0
label ADD.A #2, $0 // Relative Adressierung $0
      JMP label
```

Mit dieser Anweisung wird das A-Feld um zwei addiert. Damit ist aber das A-Feld der aktuellen Anweisung gemeint. Besser wäre der Befehl

```
label ADD.A #2, $-1 // Relative Adressierung $-1 zum Label
```

2.5.3 Indirect - Indirekt

Die Adresse im Feld zeigt auf ein Feld das nochmals einen Index enthält. Er muss noch dazuaddiert werden. Indirekte Adressierung über das A-Feld wird durch * gekennzeichnet. Indirekte Adressierung über das B-Feld wird durch @ gekennzeichnet.

```
; Indirekte Adressierung
ORG loop
loop MOV.i    bomb,  @bomb
    JMP loop
bomb DAT.f    #8,     #4
    END
```

Die Anweisung „DAT“ wird mit dem Befehl „MOV“ kopiert. Dabei wird indirekt über das B-Feld adressiert.

2.5.4 Predecrement Indirect - Predekremental Indirekt

Wie Indirekt, nur dass der Index vor der Ermittlung der tatsächlichen Adresse um eins reduziert wird. Prekremental Indirekt über das A-Feld wird mit „{“ gekennzeichnet. Prekremental Indirekt über das B-Feld wird mit „<“ gekennzeichnet.

2.5.5 Postincrement Indirect - Postinkremental Indirekt

Wie Indirekt, nur dass der Index nach der Adressermittlung um eins erhöht wird. Postinkremental Indirekt über das A-Feld wird mit „}“ gekennzeichnet. Postinkremental Indirekt über das B-Feld wird mit „>“ gekennzeichnet.

Der „Postincrement Modus“ ist gleichwertig dem „Predecrement Modus“, aber der Pointer, die Adresse, wird nach der Ausführung erhöht

Beispiel:

```
DAT #5, #-10
MOV -1, }-1 ; Aktueller Befehl
```

Nach der Ausführung ergibt sich folgende Core-Code:

```
DAT #6, #-10 ←
MOV -1, }-1 ←
;
;
;
;
DAT #5, #-10 ←
```

Selbst wenn kein Befehl vorhanden ist, werden diese Anweisungen ausgeführt.

Beispiele:

```
JMP -1, <100 // decrement den Befehl 100
DAT <50, <60 // decrement the addresses in addition to killing the process.
```

2.6 Die Modifizierer

Hiermit wird angegeben, mit welchen Teilen der Instruktionen gearbeitet wird. Wichtig ist hierbei dass die Adressierungsart eines Feldes erhalten bleibt, wenn es überschrieben wird. Wenn also ein Wert aus einem unmittelbaren Feld in ein Feld mit indirekter Adressierung geschrieben wird, so bleibt die indirekte Adressierung des Ziels bestehen. Die Ausnahme bildet nur der Modifizierer I, der die komplette Instruktion behandelt

Aufbau der Befehle:

Befehl	Modifizierer	A-Inhalt	B-Inhalt
--------	--------------	----------	----------

There are 7 different modifiers available:

- MOV.A -- moves the A-field of the source into the A-field of the destination
- MOV.B -- moves the B-field of the source into the B-field of the destination
- MOV.AB -- moves the A-field of the source into the B-field of the destination
- MOV.BA -- moves the B-field of the source into the A-field of the destination
- MOV.F -- moves both fields of the source into the same fields in the destination
- MOV.X -- moves both fields of the source into the *opposite* fields in the destination
- MOV.I -- moves the whole source instruction into the destination

2.6.1 A

Befehl bezieht sich auf beide Felder A

Beispiele:

cmp.a vergleicht den A-Wert der A-Instruktion mit dem A-Wert der B-Instruktion.
mov.a überschreibt den A-Wert der B-Instruktion mit dem A-Wert der A-Instruktion.

mov.a i1, i2 $\Leftrightarrow A[i2] = A[i1]$

2.6.2 B

Befehl bezieht sich auf beide Felder B

Beispiele:

cmp.b vergleicht den B-Wert der B-Instruktion mit dem B-Wert der A-Instruktion.
mov.b überschreibt den B-Wert der B-Instruktion mit B-Wert der A-Instruktion.

mov.b i1, i2 $\Leftrightarrow B[i2] = B[i1]$

2.6.3 AB

1. Befehlsteil bezieht sich auf das Feld A
2. Befehlsteil bezieht sich auf das Feld B

Beispiele:

cmp.ab vergleicht den A-Wert der A-Instruktion mit dem B-Wert der B-Instruktion.
mov.ab überschreibt den B-Wert der B-Instruktion mit dem A-Wert der A-Instruktion.

mov.ab i1, i2 $\Leftrightarrow B[i2] = A[i1]$

2.6.4 BA

1. Befehlssteil bezieht sich auf das Feld B
2. Befehlssteil bezieht sich auf das Feld A

Beispiele:

cmp.ba vergleicht den B-Wert der A-Instruktion mit dem A-Wert der B-Instruktion.
 mov.ba überschreibt den A-Wert der B-Instruktion mit dem B-Wert der A-Instruktion.

mov.ba i1, i2 $\Leftrightarrow A[i2] = B[i1]$

2.6.5 F

Der Befehl bezieht sich auf beide Felder A und B.

i1.A -> i2.A

i1.B -> i2.B

Beispiele:

cmp.f vergleicht den A-Wert der A-Instruktion mit dem A-Wert der B-Instruktion
 und den B-Wert der A-Instruktion mit dem B-Wert der B-Instruktion.

mov.f überschreibt den A-Wert der B-Instruktion mit dem A-Wert der A-Instruktion
 und den B-Wert der B-Instruktion mit dem B-Wert der A-Instruktion.

mov.f i1, i2 $\Leftrightarrow A[i2] = A[i1]$
 $B[i2] = B[i1]$

2.6.6 X

Der Befehl bezieht sich auf beide Felder A und B, nur ist die Richtung vertauscht.

i1.A -> i2.B

i1.B -> i2.A

Beispiele:

cmp.x vergleicht den A-Wert der A-Instruktion mit dem B-Wert der B-Instruktion
 und den B-Wert der A-Instruktion mit dem A-Wert der B-Instruktion.

mov.x überschreibt den B-Wert der B-Instruktion mit dem A-Wert der A-Instruktion
 und den A-Wert der B-Instruktion mit dem B-Wert der A-Instruktion.

mov.f i1, i2 $\Leftrightarrow A[i2] = A[i1]$
 $B[i2] = B[i1]$

2.6.7 I

Die komplette Daten werden verglichen bzw. kopiert (Alle vier Teile).

Beispiele:

cmp.i vergleicht die A-Instruktion mit der B-Instruktion.

mov.i überschreibt die B-Instruktion mit der A-Instruktion. Es wird die komplette Instruktion, inklusive Adressierung bearbeitet.

```
mov.i i1, i2    ⇔ A[ i2 ] = A[ i1 ]
                B[ i2 ] = B[ i1 ]
                Befehl[ i2 ] = Befehl[ i1 ]
                Modifizierer[ i2 ] = Modifizierer [ i1 ]
```

2.7 Die Befehle - Opcodes

Nach den Adressierungen und Modifizierern werde nun die eigentlichen Assembler-Befehle vorgestellt. Dabei kann man sich beim Lesen der Definitionen der Befehle Gedanken machen, in welcher Weise die Befehle bei den Kämpfern wirken könnten. Die Predekrementalen und Postinkrementalen Adressierungen werden nur erwähnt, wenn sie direkt benötigt werden, beispielsweise bei den Befehlen DAT oder NOP, sowie dem B-Feld der Befehle JMP oder SPL. Einige Befehlen werden mit Beispielen, kleine einzeilige Kämpfer, näher erläutert

Standards:

'88 opcodes: DAT, MOV, ADD, SUB, JMP, JMZ, JMN, DJN, CMP, SPL, SLT

'94 opcodes: MUL, DIV, MOD, SEQ, SNE, NOP, LDP, STP

Welche Modifizierer eignen sich für welchen Befehl:

- DAT, NOP
Always .F, but it's ignored.
- MOV, SEQ, SNE, CMP
If A-mode is immediate, .AB,
if B-mode is immediate and A-mode isn't, .B,
if neither mode is immediate, .I.
- ADD, SUB, MUL, DIV, MOD
If A-mode is immediate, .AB,
if B-mode is immediate and A-mode isn't, .B,
if neither mode is immediate, .F.
- SLT, LDP, STP
If A-mode is immediate, .AB,
if it isn't, (always!) .B.
- JMP, JMZ, JMN, DJN, SPL
Always .B (but it's ignored for JMP and SPL).

2.7.1 DAT

Hiermit können Daten im Code eingefügt werden. Der Core wird mit dat #0, #0-Instruktionen vorinitialisiert. Ein Prozess, der versucht diesen DAT-Befehl auszuführen, wird eliminiert. Er wird aus der Prozesswarteschlange entfernt.

Wichtig:

Sollten Pre-dekrementale oder Post-inkrementale Adressierungen vorhanden sein, so werden diese dennoch abgearbeitet.

Beispiele:

Befehl	Beschreibung
dat 0, 0	Mit dieser Instruktion wird der Core zu Beginn des Kampfes initialisiert
dat #23, #0	Speichert die Werte 23 und 0
dat { -1, <-1	Speichert die Werte -1 und -1. Dekrementiert vor der Ausführung den A-Wert und B-Wert der Instruktion. Das nennt sich auch „echte Bombe“, da bei der Ausführung nicht nur der Prozess eliminiert wird, sondern auch noch Instruktionen in seiner Umgebung manipuliert werden.
dat <2667, <5334	Ein typische Anti-Imp-Bombe. Der ausführende Teilprozess stirbt und zusätzlich werden

	zwei Stellen im Core verändert. Sollte der ausführende Prozess zu einer Imp-Spirale gehören, so wird dieser durch den Verlust eines Prozesses und dadurch das zwei andere Stellen praktisch durch ein Imp-Gate gehen, stark beschädigt.
--	---

2.7.2 MOV

Überschreibt gemäß des Modifizierers die Daten der B-Instruktion mit den Daten der A-Instruktion. Nach Abschluss des Befehls wird der Befehlszähler um eine Position weiter gesetzt.

Beispiele:

Befehl	Beschreibung
mov.a 20, 2	Überschreibt den A-Wert der Instruktion 2 Stellen weiter mit dem A-Wert der Instruktion 20 Stellen weiter.
mov.i 0, 1	Kopiert sich komplett in den nächsten Befehle
mov.i } src,>dest	Könnte ein Teil der Reproduktionsschleife eines Paper's sein. Die Zeiger auf die Quell- und Zielinstruktionen werden nach jedem Kopieren inkrementiert.

2.7.2.1 Unstimmigkeiten zwischen Modifizierer und Adressierung.

Aus der Tatsache, dass bei unmittelbarer Adressierung die Adresse der Instruktion auf 0 gesetzt wird, ergeben sich einige interessante Seiteneffekte. Diese ich hier kurz beschreiben möchte.

Bevor die eigentliche Instruktion ausgeführt wird, müssen erst die Adressen der beteiligten Instruktionen ermittelt werden. In dieser Phase sind die Adressierungsarten der Felder wichtig. Wenn die Adressierungsart eines Feldes unmittelbar ist, dann muss die Adresse auf 0 gesetzt werden. Somit kann jeder beliebige Wert dort stehen. Gearbeitet wird mit Adresse 0. Folgende „spl-Instruktionen“ sind somit in ihrer Funktion völlig gleich.

```
SPL 0
und
SPL #42
```

Bei der Ausführung wird der unmittelbare Wert im A-Feld nicht beachtet. Man kann so zum Beispiel mehr Daten in einem Befehl speichern.

Eine mögliche Anwendung wäre:

```
SPL #-1, }0
mov.i -1, }-1
mov.i -2, }-2
```

Dadurch das im A-Feld der spl-Instruktion die „Unmittelbare Adressierung“ genutzt wird, steht als Adresse dort immer 0. Die ändert sich auch nicht, wenn das B-Feld der spl-Instruktion oder die B-Felder der mov-Instruktionen prederemental indirekt darüber arbeiten. Wert und Adresse – damit befinden sich zwei Daten im selben Feld. Das die Adresse immer 0 ist stört in diesem Fall nicht. Der kleine Code-Schnipsel ist ein 66%c spl 0-Core Bomber. Er „betäubt“ seine Gegner.

2.7.3 ADD

Der Befehl überschreibt gemäß des Modifizierers die Daten der B-Instruktion mit der Summe der Werte von A-Instruktion und B-Instruktion. Nach Abschluss wird der Prozess um eine Instruktion weiter gestellt und in die Prozesswarteschlange eingefügt. Der Modifizierer I wird wie F behandelt.

Beispiele:

Befehl	Beschreibung
ADD.ab 20, 2	Addiert 20 zum B-Wert der Instruktion 2 Stellen weiter und legt das Ergebnis auch dort ab.
ADD.f 30, 2	Überschreibt das A-Feld der Instruktion 2 Stellen weiter mit der Summe der A-Felder der Instruktionen 30 und 2 Stellen weiter. B-Feld ebenso.
ADD.f #3, 5	Auf Grund der unmittelbaren Adressierung im A-Feld addiert diese Instruktion 3 zum A-Wert der Instruktion 5 Stellen weiter und 5 zum B-Wert der Instruktion 5 Stellen weiter.

2.7.4 SUB

Der Befehl überschreibt gemäß des Modifizierers die Daten der B-Instruktion mit der Differenz der Werte von A-Instruktion und B-Instruktion (B-Instruktion - A-Instruktion). Nach Abschluss wird der Prozess um eine Instruktion weiter gestellt und in die Prozesswarteschlange eingefügt. Der Modifizierer I wird als F behandelt.

Beispiele:

Befehl	Beschreibung
SUB.ab 2, 3	Zieht den A-Wert der Instruktion 2 Stellen weiter vom B-Wert der Instruktion 3 Stellen weiter ab und speichert das Ergebnis im B-Wert der Instruktion 3 Stellen weiter.
SUB.f #2607, 17	Zieht 2607 vom A-Wert der Instruktion 17 Stellen weiter und 17 vom B-Wert der Instruktion 17 Stellen weiter ab.

2.7.5 MUL

Der Befehl überschreibt gemäß des Modifizierers die Daten der B-Instruktion mit dem Produkt der Werte von A-Instruktion und B-Instruktion. Nach Abschluss wird der Prozess um eine Instruktion weiter gestellt und in die Prozesswarteschlange eingefügt. Der Modifizierer I wird als F behandelt.

Beispiele:

Befehl	Beschreibung
MUL.ba 2, 3	Multipliziert die B-Werte der Instruktionen 2 und 3 Stellen weiter miteinander und speichert das Ergebnis im A-Wert der Instruktion 3 Stellen weiter.
MUL.a #23, 4	Multipliziert den A-Wert der Instruktion 4 Stellen weiter mit 23

2.7.6 DIV

Der Befehl überschreibt gemäß des Modifizierers die Daten der B-Instruktion mit der ganzzahligen Division die Werte von A-Instruktion und B-Instruktion (B-Instruktion /A-Instruktion). Sollte der Wert der A-Instruktion 0 sein, so wird dieser Prozess nicht mehr in die Prozesswarteschlange eingeschleust. Ansonsten wird der Prozess um eine Instruktion weiter gestellt und in die Prozesswarteschlange eingefügt. Der Modifizierer I wird als F behandelt.

Beispiele:

Befehl	Beschreibung
DIV.ab 2, 3	Dividiert den B-Wert der Instruktion 3 Stellen weiter durch den A-Wert der Instruktion 2 Stellen weiter und legt das Ergebnis im B-Wert der Instruktion 3 Stellen weiter ab.
DIV.a #20, 6	Dividiert den A-Wert der Instruktion 6 Stellen weiter durch 20.

2.7.7 MOD

Der Befehl überschreibt gemäß des Modifizierers die Daten der B-Instruktion mit dem ganzzahligen Rest der Division B-Instruktion / A-Instruktion. Sollte der Wert der A-Instruktion 0 sein, so wird dieser Prozess nicht mehr in die Prozesswarteschlange eingeschleust. Ansonsten wird der Prozess um eine Instruktion weiter gestellt und in die Prozesswarte-schlange eingefügt. Der Modifizierer I wird als F behandelt.

Beispiele:

Befehl	Beschreibung
MOD.ab 2, 3	Dividiert den B-Wert der Instruktion 3 Stellen weiter durch den A-Wert der Instruktion 3 Stellen weiter und speichert den Rest im B-Wert der Instruktion 3 Stellen weiter.
mod.x #22, 12	Dividiert den B-Wert in der Instruktion 12 Stellen weiter durch 22 und legt den Rest dort ab und dividiert den A-Wert der Instruktion 12 Stellen weiter durch 12 und legt den Rest dort ab.

2.7.8 JMP

Der Befehlszähler des Teilprozesses erhält einen neuen Wert. Predekrementale oder Postinkrementale im B-Feld werden ebenfalls ausgeführt.

Beispiele:

Befehl	Beschreibung
JMP -3	Verändert die Adresse der nächsten auszuführenden Instruktion so, das als nächsten die Instruktion 3 Stellen zurück ausgeführt wird.
JMP >0, #1	Der Sprung wird indirekt über den B-Wert der Instruktion berechnet. Danach wird der B-Wert um eines erhöht, so das bei der nächsten Ausführung der jmp-Instruktion das Ziel um eins erhöht ist. Damit kann man z.B. einen sogenannten Vector-Launch realisieren.

2.7.9 JMZ

Prüft gemäß des Modifizierers den Wert der B-Instruktion und handelt wie jmp wenn der Wert 0 ist. Ansonsten wird der Prozess eine Instruktion weiter gestellt und wieder in die Prozesswarteschlange eingefügt. Der Modifizierer I wird als F behandelt.

Kurzform: Jump, wenn $A[i] == \text{Null}$.

Beispiele:

Befehl	Beschreibung
JMZ.ab -5, 2	Springt 5 Instruktionen zurück wenn der B-Wert der Instruktion 2 Stellen weiter 0 ist.
JMZ.f #-1, { 0	Wieder die Seiteneffekte durch die unmittelbare Adressierung ausnutzend, haben wir hier einen Mini-AB-Scanner. Solange er 0 als A-Wert und B-Wert vorfindet, springt er sich selber an. Vor jeder Prüfung wird der A-Wert um eins dekrementiert.

2.7.10 JMN

Arbeitet wie JMZ, nur dass es bei einem „nicht-null Wert“ der B-Instruktion nach A-Instruktion verzweigt.

Kurzform: Jump, wenn $A[i] \neq \text{Null}$.

Befehl	Beschreibung
JMN.ab -5, 2	Springt 5 Instruktionen zurück wenn der B-Wert der Instruktion 2 Stellen ungleich 0 ist.

2.7.11 DJN

Dekrementiert gemäß des Modifizierers den Wert der B-Instruktion. Dann wird dieser auf 0 geprüft. Wenn er ungleich Null ist, wird der Prozess mit Ziel A-Instruktion wieder in die Prozesswarteschlange eingeschleust. Bei 0 wird der Prozess eins weiter gestellt und in die Prozesswarteschlange eingefügt.

Kurzform: a) decrement,
b) jump if not zero

Beispiele:

Befehl	Beschreibung
DJN.ab -5, #20	Diese Instruktion dekrementiert ihren B-Wert und springt die Instruktion 5 Stellen zurück an. Sollte der B-Wert bei 0 angelangt sein, dann geht die Ausführung mit der Instruktion nach der DJN-Instruktion weiter.
DJN.f -4,4	Diese Instruktion dekrementiert A-Wert und B-Wert der Instruktion 4 Stellen weiter. Wenn beide nicht-null sind, dann wird zur Instruktion 4 Stellen vorher verzweigt.

2.7.12 CMP

Vergleicht gemäß des Modifizierers die A-Instruktion und B-Instruktion. Sollten diese beiden gleich sein, so wird der Prozess 2 Stellen weiter gestellt und in die Prozesswarteschlange eingeschleust. Es wird also eine Instruktion übersprungen. Sollten sie nicht gleich sein, so wird der Prozess um eine Stelle weiter gestellt und in die Prozesswarteschlange eingeschleust.

Beispiele:

Befehl	Beschreibung
CMP.i >targetA, >targetB	Vergleicht die komplette Instruktion bei targetA mit der Instruktion bei targetB.
CMP.a #20, 2	Vergleicht den A-Wert der Instruktion 2 Stellen weiter mit 20.

2.7.13 SLT

Vergleicht gemäß des Modifizierers den Wert der A-Instruktion mit dem Wert der B-Instruktion. Sollte der Wert in der A-Instruktion kleiner sein, wird die nächste Instruktion übersprungen. Ansonsten wird der Prozess eins weitergestellt und in die Prozesswarteschlange eingeschleust. Der Modifizierer I wird wie F behandelt. Diese Instruktion ist ein wenig kompliziert wenn man ihn das Erste Mal anwendet. Die größte Zahl im Spiel ist immer die Größe des Core's. Wenn man im Redcode größere Zahlen eingibt, dann wird diese Zahl durch CORESIZE dividiert und der Rest steht dann im Code. Um den SLT-Befehl zu benutzen, muss man alle Zahlen ins positive holen. Dann zählt nur noch die Tatsache das keine Zahl kleiner 0 sein kann.

Kurzform: Jump, wenn $A[i] < B[i]$
skip if less than

Beispiele:

Befehl	Beschreibung
SLT.a werta, wertb	Der SLT-Befehl wird bei einer CORESIZE von 8000 die -2 intern in 7998

posa nop #0 posb nop #0 werta dat #-2,#0 wertb dat #5, #0	umrechnen. Da dieser Wert nicht kleiner als 5 ist, wird der nächste Befehl nicht übersprungen und die Ausführung geht bei „posa“ weiter. Anders wäre es, wenn werta und wertb vertauscht wären. Dann würde die Ausführung bei posb weiter gehen.
--	--

2.7.14 SPL

Der ausführende Prozess wird eins weitergestellt und wieder in die Warteschlange eingeschleust. Sollte die Warteschlange ihre maximale Größe noch nicht erreicht haben, so wird ein neuer Prozess mit der A-Instruktion als Adresse in die Warteschlange eingefügt. Predrementale oder Postinkrementale Adressierung im B-Feld werden noch abgearbeitet. Dieser Befehl erzeugt also einen neuen Teilprozess mit dem Anfangsbefehl A-Instruktion.

Kurzform: split execution

Beispiele:

Befehl	Beschreibung
SPL 1	Da die Ausführung des alten Prozesses und die des neuen Prozesses bei der nächsten Instruktionen weiter gehen, verdoppelt dieser Befehl die Anzahl der vorhandenen Teilprozesse.

Bemerkung:

Die folgenden 5 Befehle sind nicht im 94er Standard enthalten. Sie sind mit der „pMars 0.8“ eingeführt worden. Da auch die 94er Hill's im Internet mit pMars arbeiten, kann man diesen Quasistandard annehmen und benutzen. Ausnahme: Es gibt spezielle Hill's, unter anderem auch den „NopSpace-Hill“ in denen der pSpace nicht erlaubt ist.

A SPL command adds an extra process to the program by splitting. The "existing" process acts as if the SPL command was "NOP" or "JMP 1". The additional process acts as if the command was "JMP dest".

The complicated bits are sorting out the order that new processes execute in, and what happens if multiple processes run the same code.

Here is a sample of code:

```
bomb DAT #0, #0
loop MOV bomb, <bomb
      SPL 1
      JMP 1
```

If I repeatedly list the code sample with the position of the program counter for each process and the program counter queue shown, it should be possible to follow what's happening:

```
Q=A MOV:A SPL: JMP:
Initially only process A exists. It executes MOV
(to bomb-1).
Q=A MOV: SPL:A JMP:
Then it executes a split creating process B.
Q=AB MOV:B SPL: JMP:A
Then process A executes a JMP.
Q=BA MOV:BA SPL: JMP:
Process B executes MOV to (bomb-2).
Q=AB MOV:A SPL:B JMP:
Process A executes MOV to (bomb-3).
Q=BA MOV: SPL:BA JMP:
```

B splits into B & C.
 Q=ABC MOV:C SPL:A JMP:B
 A splits into A & D.
 Q=BCAD MOV:CD SPL: JMP:BA
 B JMPs.
 Q=CADB MOV:CDB SPL: JMP:A
 C executes MOV to (bomb-4).
 Q=ADBC MOV:DB SPL:C JMP:A
 A JMPs.
 Q=DBC A MOV:DBA SPL:C JMP:
 D executes MOV to (bomb-5)
 Q=BCAD MOV:BA SPL:CD JMP:
 B executes MOV to (bomb-6)
 Q=CADB MOV:A SPL:CDB JMP:
 C splits into C & E
 Q=ADBCE MOV:AE SPL:DB JMP:C
 etc.

2.7.15 SEQ

Synonym für CMP.

Kurzform: skip if equal

2.7.16 SNE

Vergleicht gemäß des Modifizierers A-Instruktion und B-Instruktion. Sollten diese beiden nicht gleich sein, so wird der Prozess um 2 Stellen weiter gestellt und in die Prozesswarteschlange eingeschleust. Es wird also eine Instruktion übersprungen. Sollten sie gleich sein so wird der Prozess um eine Stelle weiter gestellt und in die Prozesswarteschlange eingeschleust.

Kurzform: Jump, wenn $A[i] \neq B[i]$
 skip if not equal

Beispiele:

Befehl	Beschreibung
SNE.f >posa, >posb	Vergleicht den A-Wert der Instruktion bei posa mit dem A-Wert der Instruktion bei posb und den B-Wert der Instruktion bei posa mit dem B-Wert der Instruktion bei posb. Beide Zeiger werden nach dem Vergleich um eins inkrementiert.

Hinweis:

Posa und posb müssen vorhandene Labels sein.

2.7.17 NOP

Dieser Befehl – „No Operation“ tut nichts. Predekrementale oder Postinkrementale A-Felder oder B-Felder werden dennoch abgearbeitet. Nach Abschluss wird der Prozess um eine Instruktion weiter gestellt und in die Prozesswarteschlange eingefügt. Dieser Befehl ist beim Debuggen eines Kämpfers sehr nützlich. Man kann ohne Nebenwirkungen Daten im Kämpfer ablegen und trotzdem die Ausführung darüberlaufen lassen. Zusätzlich kann man mit einer nop-Instruktion zwei Imp-Gate's betreiben.

Beispiele:

Befehl	Beschreibung
NOP <2667, <5334	Ein Imp-Gate gegen Spiralen.
NOP <, } 0	Diese Anweisung in einen Gegner eingepflanzt überschreibt eine seiner Instruktionen und lässt ihn mit jedem Aufruf einen B-Wert einer Instruktion dekrementieren. Nach der Dekrementierung wird der Zeiger inkrementiert. Somit ist nach jedem Durchlaufen ein andere Instruktion dran. Das ist in Verbindung mit einem Stealth-Kämpfer schon eine vielversprechende Taktik, den Gegner zu beschädigen.

2.7.18 LDP

Der Befehl überschreibt die Daten der B-Instruktion mit dem Wert aus der Stelle A des *pSpace*. Um die Nutzung des pSpace etwas einzuengen, werden alle Modifizierer ignoriert. Die Instruktion arbeitet praktisch immer wie Modifizierer B. Werte aus dem pSpace können vielfältig genutzt werden. Wenn man den Kämpfer dementsprechend gestalten können Werte aus dem pSpace als Sprungziele, Schleifenzähler oder Bombardierabstände in den Kämpfer eingesetzt werden. Durch die fehlenden Modifizieren ist man zwar auf einige Tricks angewiesen, aber das ist vertretbar. Der pSpace sollte nur im Launch des Kämpfer genutzt werden, damit bei Beschädigungen des Kämpfer nicht auch noch Gedächtnisverlust durch fehlerhafte Zugriffe auf den pSpace hinzu kommen.

Kurzform: load from p-space (loads a number from private storage space)

Beispiele:

Befehl	Beschreibung
LDP 20, 13	Der B-Wert der Instruktion 20 Stellen weiter, gibt die Adresse im pSpace wieder. Deren Wert wird im B-Wert der Instruktion 13 Stellen weiter gespeichert.

2.7.19 STP

Speichert gemäß des Modifizierers den Wert der A-Instruktion in der Stelle B des pSpace.

Kurzform: save to p-space (saves a number to private storage space)

Beispiel:

Befehl	Beschreibung
STP 20, 13	Der B-Wert der Instruktion 20 Stellen weiter, gibt die Adresse im pSpace wieder. Deren Wert wird im B-Wert der Instruktion 13 Stellen weiter aus dem pSpce geladen.
STP.AB #4, #5	P-Space[5] = 4

3 P-Space

pSpace ist ein Speicher, der einem Kämpfer zugeordnet wurde. Es gibt zwei Kommandos mit denen er darauf zugreifen kann (LDP, STP). Die Speicherstellen im pSpace unterscheiden sich vom Core dadurch, dass nur ein Wert und keine ganze Instruktion mit ihren Feldern A und B darin Platz haben. Das Besondere an diesem Speicher ist aber, dass er bei „Mehr-Runden-Kämpfen“ seinen Inhalt behält. Ein Kämpfer kann auf die Weise Erfahrungen, die er im letzten Kampf gemacht hat, zurückgreifen.

Ich beschreibe den pSpace dadurch, wie ein Kämpfer mit dem Verteidigungsmechanismus „Bomber Dodger“ ihn benutzen könnte. Ein Kämpfer mit Bomber Dodger scannt den Speicher um zwei Informationen über die Bomben des gegnerischen Dwarf's zu bekommen.

- Der Abstand zwischen den Bomben
- Die Position der Bomben relativ zu ihm. Sobald der Scanner also die erste Bombe erwischt, muss er eine zweites Modul starten

welches dann den Abstand zur nächsten Bombe ermittelt. Das ist aber auch nicht immer so einfach. Wenn der Dwarf keine „coloured“ Bomben wirft, fällt diese Möglichkeit aus. Aber davon ausgegangen, es sind coloured Bomben, hat der Kämpfer jetzt die Möglichkeit, die Angabe des Abstands im pSpace zu speichern. Danach kann er den passenden Minikämpfer um die Bomben herumlegen und seinen Angriff starten. Beim nächsten Kampf kennt er bereits den Abstand der Bomben und kann den Bomben entgegen scannen. Sobald er die erste hat kann er sofort die Minikämpfer installieren. Man kann das Ganze natürlich noch verbessern. Dadurch das der Kämpfer den Abstand der Bomben kennt, könnte er versuchen, die Position des Gegners zu bestimmen. Während er nach Bomben scannt könnte ein Zähler mitlaufen, über den die Zeit bestimmt wird. den die Bomben gebraucht haben. Dann berechnet er aus Abstand der Bomben und verschiedenen Bombardiergeschwindigkeiten die Entfernung des Kämpfers und scannt verstärkt in diesen Bereichen. Wenn er den Kämpfer lokalisiert hat, dann kann er auch die Angabe über die Geschwindigkeit des Gegners im pSpace abspeichern. Mit diesen Angabe kann er in der nächsten Runde einen Bereich vor sich, der in der Größe dem Abstand der Dwarfbomben entspricht, absキャン und auf den Teppich warten. Sobald die erste Bombe registriert wird, kann der Kämpfer an Hand der gespeicherte Daten die Position des Dwarf ermitteln und ihn praktisch mit ein paar gezielten Schüssen erledigen. Nun, das war wieder etwas nach der Art - Kampf gegen bekannten Gegner. Der Kämpfer findet zwar Intervall und Geschwindigkeit des Gegner selbst raus, aber gegen etwas anderes als einen Dwarf ist er machtlos. Interessant wird es wenn auch der Gegner den pSpace nutzt.

Nun, aber auch der pSpace hat Angriffspunkte. Vampire können Kämpfer die den pSpace nutzen, auf Anweisungen lenken, welche den Speicher löschen oder sinnlose Werte in ihn schreiben. Dann ist der Kämpfer wieder auf sich gestellt. Diese Technik wird Brainwashing genannt. Heutige Kämpfer sind immer gut beraten, Brainwashing zu beherrschen.

Durch einen bestimmten PseudoOpcode kann man den pSpace mit einer bestimmten Identifikationsnummer benennen. Ein anderer Kämpfer mit der selben Nummer hat dann auch den gleichen pSpace, mit Ausnahme der Speicherstelle 0, auf die jeder Kämpfer alleiniges Zugriffsrecht hat. Somit könnten Allianzen zwischen Kämpfern gebildet werden. Wer aber keine Allianz eingehen möchte, der sollte den pSpace nicht benennen, da dadurch automatisch die Gefahr gegeben ist, dass noch jemand anderes darin arbeitet.

-space -- the final frontier

P-space is the latest addition to Redcode, introduced by pMARS 0.8. The "P" stands for private, permanent, personal, pathetic and so on, whichever you like. Basically, the P-space is an area of memory which only your program can access, and which survives between rounds in a multi-round match.

1. Beispiel:

Die Anweisung „STP.AB #4, #5“ fügt den Wert 4 in das Feld mit der Nummer 5.

2. Beispiel:

```
STP.B  2,   3
...
DAT    #0,  #10
```

DAT #0, #7

Diese Anweisung fügt den Wert 10 in das siebte P-Space-Feld. Dies ist also eine indirekte Adressierung.

Wichtig:

Man kann mit dem Befehl LDP nur jeweils einen Wert aus dem P-Space holen. Dementsprechend sind folgende Befehle gleich (LDP.F, .LDP.X und LDP.I) sind identisch mit LDP.B!

Meistens wird der P-Sapce zum Speichern der Strategie verwendet. Solche Programme heißen „P-warriors“, „P-switchers“ oder „P-brains“. Der P-Space ist leider nicht so privat, wie man es gerne hätte. Selbst ohne der Angabe der PIN-Nummer. Ein Programm kann nämlich von einem anderen Programm übernommen werden. Dann hat der „Gegner“ auch Zugriff auch diese Daten. Diese Technik wird „brainwashing“ genannt.

4 Die Prozesswarteschlange

Kein Computer mit einem Prozessor kann wirklich gleichzeitig mehrere Sachen machen. So auch die „MARS“ nicht. So wird jeder Kämpfer in einer Warteschlange gespeichert. So kommen sie immer nach der Reihe dran. Jeder Kämpfer selbst hat ebenfalls eine Warteschlange für die Teilprozesse gespeichert. Seine Prozesswarteschlange hat jeden Prozess des Kämpfers in sich gespeichert. Kommt ein Kämpfer an die Reihe, so gibt er der MARS einfach den nächsten Prozess. Meist ist das nur die Adresse der nächsten Instruktion für diesen Prozess. Die MARS führt den Befehl aus und gibt den Befehl in den meisten Fällen zurück. Während die MARS den Befehl ausführt, können verschiedene Dinge passieren.

1. Der Befehl wird abgearbeitet. Die MARS verändert die Adresse des Prozesses in der Form, dass er nun auf die nächste Instruktion zeigt.
2. Der Befehl ist ein Sprungbefehl. Die MARS verändert die Adresse des Prozesses wie es in dem Sprungbefehl gewünscht ist.
3. Der Befehl `dat, div 0` oder `mod 0` veranlasst die MARS den Prozess nicht mehr an den Kämpfer zurückzugeben, der Prozess ist tot.
4. Der `spl`-Befehl veranlasst die MARS, nachdem sie den normalen Prozess um eine Stelle weiter gestellt hat und an den Kämpfer zurückgegeben hat, einen neuen Teilprozess zu erzeugen der seine nächste auszuführende Adresse durch das `A`-Feld der `spl`-Instruktion bekommt. Die Tatsache, dass der neue Prozess in der Warteschlange hinter dem Erzeugenden steht, ist sehr wichtig, da dieser beim nächsten Durchlauf ausgeführt wird.

Nun arbeitet der „Taskmanager“ alle Kämpfer nacheinander B: Existieren Teilprozesse, so werden diese nacheinander ausgeführt. Folgende Situation ist möglich:

Kämpferwarteschlange der MARS :

Kämpfer A,

Kämpfer B,

Kämpfer C

Anzahl Prozesse der Kämpfer: A = 2,

Anzahl Prozesse der Kämpfer: B = 1,

Anzahl Prozesse der Kämpfer: C = 5,

Die Kämpfer, sowie die Prozesse in ihnen werden der Reihe nach abgearbeitet.

A - Prozess 1

B - Prozess 1

C - Prozess 1

A - Prozess 2

B - Prozess 1

C - Prozess 2

A - Prozess 1

B - Prozess 1

C - Prozess 3

A - Prozess 2

B - Prozess 1

C - Prozess 4

A - Prozess 1

B - Prozess 1

C - Prozess 5

A - Prozess 2

B - Prozess 1

C - Prozess 1

A - Prozess 1

B - Prozess 1

C - Prozess 2

A - Prozess 2

B - Prozess 1

C - Prozess 3

Man kann also ganz schnell sehen, dass Kämpfer alleine aus der Tatsache, dass sie mehrere Prozesse haben, keinen Geschwindigkeitsvorteil haben. Geschickte Programmierung macht „Mehrprozesskämpfer“ zu dem, was

sie sind. Im Kampf hat der „Mehrprozesskämpfer“ den Vorteil, das er nicht sofort stirbt, wenn er auf eine dat-Instruktion läuft. Selbst, wenn der Kämpfer so unglücklich getroffen wurde, dass alle seine Prozesse sterben, dauert das dann noch eine gewisse Zeit.

5 Wie programmiert man einen Kämpfer

Es gibt kein Geheimrezept dafür, wie man einen Kämpfer programmiert. Man sucht sich das Konzept aus, welches Einem am Meisten zusagt und legt los. Man sollte sich andere Kämpfer des gleichen Typs ansehen, um zu verstehen, worauf es ankommt. Die eine oder andere Idee kann man so auch übernehmen. Man sollte aber immer versuchen, etwas Neues zu finden. Die Geschwindigkeit zu optimieren oder einfach kürzer sein, als alle anderen Kämpfer.

5.1 Die Trickkiste.

Beim Ausprobieren von Kämpfern kommt es manchmal zu Situationen die nicht erwartet wurden. Manchmal verursacht ein Tippfehler die seltsamsten Ergebnisse. Man sollte hier immer aufpassen, ob man nicht etwas davon gebrauchen kann. Jeder Programmierer in jeder Programmiersprache hat sein eigene Trickkiste. Dort liegen kleine Codeschnipsel, die nur darauf warten, benutzt zu werden. Ich möchte hier ein paar Schnipsel oder Tricks zum Besten geben, die ich in meiner kurzen Zeit bei CoreWar bereits sammeln konnte. Einige nehme ich auch aus dem CoreWarrior, einem CoreWar Newsletter, der in unregelmäßigen Abständen selbst heute noch erscheint.

5.1.1 Parallele Prozesse

Besonders bei Kämpfern der Sorte Silk, ist es wichtig das eine bestimmte Anzahl von Prozessen erzeugt werden, die dann auch noch parallel zueinander laufen müssen. Um die Anzahl der Prozesse nicht unnötig zu vergrößern, sollte man dann auch wirklich die genaue Anzahl erreichen. Eine Technik, die genau das ermöglicht ist folgende. Angenommen man will n parallele Prozesse erzeugen, so schreibt man die binäre Darstellung von n-1, angefangen mit der ersten 1 von Links, an die Stelle im Quellcode wo die Prozesse erzeugt werden sollen. Nach jeder Ziffer wird in die nächste Zeile gewechselt. In allen Zeilen mit einer 1 schreibt man dann „spl 1“ und in alle Zeilen mit der 0 „mov.i -1, 0“. Durch den so entstandenen Quellcode, werden n parallele Prozesse erzeugt.

Beispiel:

Für einen Silk Replicator mit der Länge 7 möchte wir 7 Prozesse erzeugen. Also erzeugt man 0 bis n-1 Teilprozesse.

Die binäre Darstellung von 6 ist 110.

```
Launch spl 1 ; 1
      spl 1 ; 1
      mov.i -1, 0 ; 0
```

Ich habe diesen Trick in vielen Kämpfern gesehen. Ich glaube ursprünglich stammt er von Beppe Bezzi. Das Ganze ist recht einfach von der Funktion, aber die Idee war wohl schon genial. Das ist allerdings auch ein gutes Beispiel für CoreWar. Selbstmodifizierender Code ist hier eher die Regeln als die Ausnahme.

5.1.2 Anzahl der Prozesse verkleinern.

Eine Sache, die bei der Prozesserzeugung stört, ist dass man die neuen Prozesse nur schwer wieder loswerden kann. Als Beispiel muss hier wieder der Silk Replicator herhalten. Nachdem die sieben Prozesse die nächste Kopie gestartet und kopiert haben, führen sie den Rest dieser Kopie aus. Sollte das eine Endlosschleife sein, die einen kleinen Bereich vor sich bombardiert oder ein kleiner B-Scanner sein, so ist es nicht nötig, bzw. manchmal störend, wenn es zu viele Prozesse vorhanden sind. Hier eine Möglichkeit, die Prozesse auf einen zu reduzieren.

```
silk spl 1234,0
     mov.i >silk, }silk
onep mov.i datb, 0
...
```

```
...
  datb  dat #0,#0
```

Alle Prozesse laufen parallel. Der erste Prozess der die Instruktion bei onep ausführt, kopiert die dat-Bombe auf sich selbst. Dadurch werden alle Prozesse, die nach ihm kommen, eliminiert.

5.1.3 djn-Stream

Am Besten sind immer die Sachen, die den Gegner schädigen und dabei nichts kosten. Sollte man in seinem Programm irgendwo einen unbedingten Rücksprung an einem Schleifenanfang haben, dann kann man dort mit der djn-Instruktion einen sogenannten djn-Stream erzeugen lassen. Wenn wir uns die Funktionsweise der DJN-Instruktion nochmal ansehen, dann sollten folgende Zeilen eigentlich klar sein.

```
DJN.b loop,<MINDISTANCE
jmp loop,<-1
```

Die DJN-Instruktion dekrementiert den Wert aus dem B-Feld und zweigt nach loop wenn der Wert 0 ist. Durch die Predekrement-Indirekt Adressierung ist das jedesmal ein anderes Feld. Der djn-Stream dekrementiert damit die Felder im Core. Durch verschiedene Adressierungen geht das vor.- wie rückwärts. Damit der djn-Stream den Kämpfer nicht selbst dekrementiert, sollte man vor den Kämpfer einen „decoy“ mit „dat #1,#1“-Anweisungen setzen, da das den Stream unterbricht. Damit der Stream, sowie die Schleife, nach einer Unterbrechung des djn-Streams weiterläuft, kann dann die ursprüngliche jmp-Instruktion leicht modifiziert dafür sorgen, dass die Schleife weiterläuft, und der Stream wieder von Vorne beginnt. Die Adresse im B-Wert der djn-Instruktion wird durch das B-Feld der jmp-Instruktion um eines weiter gestellt. Wenn man die Adresse nicht verändert, wird der djn-Stream über den Schutzdecoy schnell doch noch in den Kämpfer laufen und ihn zerstören. Mit dem Modifizierer .f ist der djn-Stream sehr effektiv weil er beide Werte der Instruktion dekrementiert. Ein damit verwundeter Kämpfer wird höchstwahrscheinlich nicht mehr richtig funktionieren oder sogar sterben. Gegen folgenden Kämpfer hat selbst unser erweiterter Dwarf nicht den Hauch einer Chance. Der Kämpfer besteht aus 2 * 2 Instruktionen. Er dekrementiert aber nur über eine Stelle. Da der Dwarf ein mod-4 Bomber ist, wird er schnell eine der beide Schleife zerschießen. Die andere läuft aber weiter und dekrementiert den Core mit der Geschwindigkeit c immer weiter. Durch den dat #1,#1-decoy gegen sich selbst geschützt, läuft die überlebende Schleife immer weiter. Da der Dwarf nur über einen Prozeß verfügt wird er sehr schnell in Anweisungen springen, die schwer zu verdauen sind.

```
org launch
launch  spl loop2, #-1
loop1   djn.f loop1, <launch
        jmp loop1, <-1
loop2   djn.f loop2, <launch
        jmp loop2, <-1
        for 50
            dat #1,#1
        rof
        end
```

Die Laufzeitkosten des djn-Stream sind zu vernachlässigen. Ab und an wird die jmp-Instruktion ausgeführt. Die Größe des nötigen Sicherheitsdecoys ergibt sich aus der Geschwindigkeit und der Größe des Cores. Läuft der Stream mit 100%c, dann kann er den Core höchstens 10 mal umrunden. Also ergibt sich ein Größe von 10. Wenn die Schleife allerdings mehrere Anweisungen hat, dann wird der Stream langsamer und der decoy kann kleiner ausfallen.

5.1.4 Effektive Bomben

Wenn wir noch mal zurückgehen zu unserem Dwarf, dann sehen wir die dat-Bombe, mit der er um sich geschmissen hat. Wenn sie richtig sitzt, dann ist der Kampf zu Ende. Doch was ist mit Gegnern die verschiedene Module besitzen. Ein Kämpfer könnte sich selbst reparieren, oder wir treffen nur einen active Decoy. Wenn wir

eine "Instanz" eines Silk's treffen, dann werden die Anderen nur schneller. Effektive Bomben sind ihrem Zweck entsprechend. Einen Silk Replicator zum Beispiel, sollte man mit seinen Bomben dazu bringen, sehr viele unnötige Prozesse zu erzeugen. Als Beispiel könnte man folgende Bombe nennen.

```
spl 0
jmp -1
```

Diese Bombe erzeugt im Laufe der Zeit eine Menge Prozesse. Allerdings könnte die Zahl größer sein. Die jmp-Instruktion kostet viel Zeit. Mehr Prozesse in kürzerer Zeit, könnten sicherlich durch mehrere spl 0-Zeilen vor der jmp-Instruktion generiert werden. Allerdings würde die Bombe größer und die Bombardiergeschwindigkeit darunter leiden, so das der Kämpfer wohl keine Chance mehr hätte. Also muß eine smartere Bombe entwickelt werden. Folgende Bombe aus dem CoreWarrior 18 ist sehr interessant.

```
spl #2, 0
mov.i -1, }-1
```

Diese Bombe entfaltet sich bei der Ausführung immer weiter. Es werden dabei enorm viele Prozesse erzeugt, da hinter der mov-Instruktion ein Teppich aus lauter spl-Instruktionen entsteht. Man könnte meinen, dass wir mit dieser Bombe dem Gegner einen stabilen CoreClear geben, mit dem er den eigenen Kämpfer erwischen könnte. Aber das ist nicht richtig. Der Gegner wird durch diese Bombe so stark verlangsamt, dass der Teppich nicht sehr lang wird. Davon ausgangen, die erste Bombe trifft den Gegner und er führt die Bombe mit einem Prozess aus, dann ist der Teppich am Ende des Kampfes gerademal 14 Instruktion lang. Man kann mit allen Instruktionen bombardieren. Effektive Bomben können auch aus zwei oder sogar drei Instruktionen bestehen. Zu lang sollten die Bomben nicht sein. Ausserdem dürfen sie nicht zu stark davon abhären, dass der Gegner auch wirklich bei der ersten Instruktion mit der Ausführung beginnt (siehe Option ORG)

5.2 Stepsize - Schrittweite

Das Problem den Gegner zu treffen, ist nicht zu unterschätzen. Beim Dwarf kommt auch noch das Problem dazu, dass er sich nicht selber treffen darf. Eine Bombe an jede vierte Stelle im Core. Das müsste eigentlich reichen, um einen Gegner zu erwischen. Nun mal abgesehen, man kämpft gegen einen Bomber-Dodger, ist es sicherlich ausreichend, jede vierte Stelle im Core mit einer Bombe zu bestücken. Durch die vier Stellen Abstand geht der Dwarf sicher das er nicht durch die Bomben getroffen wird. Allerdings wird eine Bombe immer vier Stellen neben der letzten gelegt. Wir arbeiten uns vor wie ein Imp. Sehr langsam geht es durch den Core. Wenn der Gegner genau auf der anderen Seite sitzt, dann müssen wir erst den halben Core mit unseren Bomben treffen, bevor wir den Gegner erwischen. Das ist nicht besonders gut. Selbst wenn wir den Mindestabstand bei der ersten Bombe mit einrechnen, ist das nicht sonderlich gut. Man könnte auf die Idee kommen, den Dwarf zu erweitern:

```
org loop
loop  mov.i datb,@datb      ; Bombe B-Indirekt " über datb legen.
      add.ab #4,datb        ; Nächste Position berechnen.
      jmp weiter           ; Decoy " überspringen.
      datb dat #4000,#0     ; Bombe, Bombenposition und Decoy
weiter mov.i datb,*datb     ; Bombe A-Indiekt " über datb legen.
      add.a #4,datb        ; Nächste Position berechnen.
      jmp loop             ; wieder von vorne.
end
```

So würde an zwei verschiedenen Seiten des Core's mit der Bombardierung angefangen. Das ist schon besser. Schauen wir uns mal an, wie zwei dieser Dwarf gegeneinander kämpfen. Ich lasse jetzt zwei völlig identische Dwarfs 10 mal 100 Kämpfe gegeneinander antreten. Hier sind die Ergebnisse.

	Dwarf1	Dwarf2	unentschieden
	39	27	34
	37	38	25
	33	43	24
	31	35	34
	41	36	23
	28	46	26
	45	35	20

	34	35	31
	33	45	22
	43	35	22
Gesamt	364	375	261

Das Ergebnis war zu erwarten. In ca. 25% der Kämpfe liegen die Kämpfer so zueinander, dass sie sich nicht treffen können. Alle anderen Kämpfer gewinnt mal der Eine, mal der Andere. Es gewann immer der Kämpfer mit der besseren Position. Jetzt wird Dwarf2 modifiziert. Er lässt einen Bombenteppich jetzt rückwärts laufen und beachtet auf beiden Seiten die MINDISTANCE.

```

org loop
loop    mov.i datb,@datb
        add.ab #4,datb
        jmp weiter
        datb dat #-MINDISTANCE,#MINDISTANCE
weiter  mov.i datb,*datb
        add.a #-4,datb
        jmp loop
end

```

Die Ergebnisse sprechen für sich.

	Dwarf1	Dwarf2	Unentschieden
	17	57	26
	31	50	19
	20	63	17
	14	57	29
	21	58	21
	17	53	30
	20	57	23
	18	59	23
	15	55	30
	18	49	33
Gesamt	191	558	251

Gleichbleibend ist hier die Anzahl der Unentschieden. Dwarf2 ist jetzt um einiges besser als Dwarf, nur dadurch das er die MINDISTANCE mit einbezieht, dadurch einen kleinen Vorsprung hat. Aber eines ist immer noch geblieben; Der Ausgang des Kampf hängt sehr stark davon ab, an welche Positionen die Kämpfer liegen, die Unverwundbarkeit durch Bombardierung der dat-Instruktion mal ausgenommen.

Besser wäre es, wenn der Core möglichst gleichmäßig mit den dat-Bomben belegt werden könnte. Dann wäre die Position nicht mehr so wichtig. Der Platz in den sich der Gegner befinden kann, müsste mit jeder Bombe verkleinert werden. Wenn wir den Dwarf 1 mal als Beispiel nehmen, so kann aus seiner Sicht der Gegner zu Beginn des Kampfes in 7800 Stellen des Core's versteckt sein. Nach seinem ersten Durchlauf sind es zwei Stücke mit je 3896 Stellen. Das ist nicht schlecht, allerdings wird mit der nächsten Bombe daran nicht viel geändert. Dann sind es zwei Stücke mit je 3892 Stellen. Wenn man mit jeder Bombe größere Stücke abtrennen könnte, dann würde der Gegner unabhängig von seiner Position schneller eingeengt bzw. die Wahrscheinlichkeit ihn zu treffen wäre höher (Wahrscheinlich binäre Suche). Das ist genau das was sich schon viele andere CoreWar-Anhänger gedacht haben. Die Suche nach der richtigen Schrittweite, den „optimal constants“, ist etwas, was jeden Programmierer von digitalen Kämpfer beschäftigt. Man kann die richtige Schrittweite durch Probieren ermitteln. Allerdings ist der Aufwand sehr hoch. Nicht jeder Kämpfer ist so einfach gestrickt, wie unsere Dwarf's. Ich möchte jetzt an Hand des Programms Corestep von Jay Han zeigen, wie man den Dwarf an Hand besserer Schrittweiten tunen kann.

5.2.1 Corestep

Dieses Programm berechnet günstige Schrittweiten für Kämpfer. Das müssen nicht immer Bombardierungsschritte sein. Mit B-Scanner geht es genauso gut. Als Erstes wollen wir uns die Parameter ansehen die Corestep für seine Berechnung braucht. Der erste Wert hinter dem Kommando sollte die Größe des Core's sein. Wird hier

keine Größe angegeben, so wird per Voreinstellung 55440 eingesetzt. Danach können folgende Parameter genutzt werden:

-m #	Modulowert. Als Schrittweite nur Zahlen ausgeben die durch # restlos teilbar sind. Das ist beim Dwarf wichtig um selbstbombardierung zu verhindern. Erlaubt sind hier nur echte Teiler der Coresize.
-l #	Größte Schrittweite Nur Schrittweiten ausgeben die diesen Wert nicht überschreiten. Per Voreinstellung die Hälfte der Coresize
-b #	Ausgabeanzahl Die besten # Schrittweiten ausgeben. 0 bedeutet alle und nicht sortiert.
-c	Die beste Schrittweite muss an Hand eines Bewertungssystem ausgewählt werden. Hierbei wird für jede Schrittweite nach einem bestimmten Algorithmus eine Bewertung abgegeben. Grundsätzlich dreht sich dabei alles darum wie gleichmäßig und wie schnell der Core abgedeckt wird. Wenn wir mit mod-4 Stepsizes arbeiten, dann ist klar das wenn jede mögliche Bombe gelegt ist der Core gleichmäßig abgedeckt ist. Die Bewertungsalgorithmen sind aber darauf aus eine Stepsize zu finden, welche auch nach wenigen Bomben den Core gleichmäßig bedecken. Der Bewertungsalgorithmus kann unter folgenden ausgewählt werden.
c	Klassische Bewertung: Nach jeder theoretisch gelegten Bombe werden die Längen der Lücken zwischen ihnen auf addiert. Durch die Anzahl der gelegten Bomben dividiert ergibt die durchschnittliche Größe. Die Schrittweiten die hier auch sehr schnell niedrige Werte haben sind die besseren.
a	Alternative Bewertung: Da bei einem kompletten Durchlauf keine zwei Bomben auf eine Stelle gelegt werden ist es automatisch so das eine Bomben immer zwischen 2 andere gelegt wird. Die Abstände der Bomben zu den Mittelpunkten der Lücken werden auf addiert und durch die Anzahl der gelegten Bomben dividiert. Je schneller die Schrittweite hier kleinere Ergebnisse erzielt um so besser ist.
-f #	Finden. Bei diesem Bewertungsalgorithmus gibt man die Größe des Zielbereichs an. Je kleiner die Lücken zwischen den Bomben werden, desto größer ist die Wahrscheinlichkeit den Gegner zu treffen. Die Schrittweiten die hier schnell ein große Wahrscheinlichkeit erreichen liegen weiter vorn.
-s #	Test. Hier kann man eine eigene Schrittweite berechnen lassen. Das ist ganz nützlich wenn man durch andere Module im Kämpfer eventuell. als Nebenprodukt eine Schrittweite vorgegeben bekommt. Hier könnte man Testen wie gut diese ist, indem man erst die 10 besten und dann die eigene berechnen lässt.
-q	Minimale Ausgabe. Nur die durch -b bestimmte Anzahl an Schrittweiten wird ausgegeben.
-v	Erweiterte Ausgabe. Neben den Besten werden noch einige andere Werte ausgegeben.
-o	Alternativ. Wie -q nur als Redcode-Kommentar formatiert. So kann man die Schrittweiten direkt in den Kämpfer übernehmen und sie nach und nach ausprobieren ohne jedesmal eine Berechnung starten zu müssen.

Wenn wir nun Schrittweiten für unseren Dwarf erhalten wollen, dann ist es natürlich wichtig, dass wir den richtigen Modulowert benutzen. Sonst würde sich der Dwarf nach kürzester Zeit selbst beschließen, denn vor Selbstverstümmelung schützen uns die Schrittweiten nicht. Nun folgt die Eingabe und das Resultat.

```
D:\Temp\corestep>corestep 8000 -m 4 -cc -l 8000 -b 5
corestep v3: Non-simulating optimal step finder by Jay Han, 5/13/1994.
Coresize 8000. Mod-4. Maximum 8000. Best 5. Scoring: classic.
Step 3044 Score: 17.11
Step 3364 Score: 17.11
Step 4636 Score: 17.11
Step 4956 Score: 17.11
Step 2876 Score: 17.02
```

Die beide ersten Werte übernehmen wir jetzt in einen der beiden Dwarfs. Somit gibt es zwei bis auf die Schrittweiten identische Dwarf's.

```
org loop
loop    mov.i datb,@datb
        add.ab #3044,datb ; <--- 1. optimale Konstante ---
        jmp weiter
        datb dat #-MINDISTANCE,#MINDISTANCE
weiter  mov.i datb,*datb
        add.a #3364,datb ; <--- 2. optimale Konstante ---
        jmp loop
end
```

Jetzt die Ergebnisse der 10×100 Kämpfe. Der DwarfE hat die corestep-Konstanten.

	DwarfI gewinnt	DwarfE gewinnt	unentschieden
	24	50	26
	31	45	24
	12	53	35
	26	48	26
	27	53	20
	28	48	24
	20	50	30
	28	48	24
	30	48	22
	23	49	28
Gesamt	249	492	249

Man kann sehr deutlich erkennen, dass gute Schrittweiten bei der Kampfkraft eines Gegner ein sehr große Rolle spielen. Das ist bei einem simplen Kämpfer wie dem Dwarf genauso, wie bei einem Quickschanner oder einem Silk Replicator. Allerdings muss dabei gesagt werden, dass Corestep nicht hilfreich ist, wenn es um Paper oder Silk geht. Die Schrittweiten müssen dort nach anderen Kriterien gewählt werden.

5.2.2 mOpt

Dieses Programm unterscheidet sich in Bedienung und der Art der zu übergebenden Parameter vom vorherigen. Nach dem Start wird der Benutzer nach der Coregröße und der Größe des Ziels gefragt. Danach kommt die Eingabe eines Inkrement/Offset-Pärchens. Hier kann man nun die Werte eingeben, die man hat, oder einen Generator nutzen. Ein Generator ist der Parameterliste einer for-Schleife in C sehr ähnlich. Sie besteht aus folgenden Teilen.

Anfangsbedingung

Die Variable wird auf den Anfangswert gesetzt.

Endbedingung

Der Generator läuft solange wie dieser Ausdruck mit wahr ausgewertet wird.

Variablen-Verlauf

Hiermit wird festgelegt wie sich die Variable nach jedem Durchlauf verändert.

Während man bei Corestep einfach mit dem Modulo-Parameter arbeiten konnte, so muss man hier einen Generator formulieren. Das ist für diesen Fall zwar mehr Arbeit, aber macht das Programm sehr flexibel. Man kann zum Beispiel unregelmäßige Schrittweiten formulieren oder eine Schrittweite die man praktisch als Nebenprodukt eines anderen Teil des Kämpfers mitnutzen, aber nicht verändern kann. Man kann mehrere Inkrement/Offset-Pärchen angeben. Das ist für Kämpfer, die mehrere Bomben in einer Schleife werfen, nötig. Unser erweiterter Dwarf, den wir mit den Corestep Schrittweiten getunt haben, ist schon so ein Kandidat. Die Schrittweiten sind einzeln gesehen sehr gut. Aber im Zusammenspiel ergibt sich ein anderes Bild. mOpt könnte unseren Dwarf mit besseren Schrittweiten versorgen. Dazu müssen wir den Bombardierungsintervall in der

Increment/Offset Notation formulieren. Da wir beide Inkremente frei wählen lassen können, müssen wir uns nur um den Modulwert Gedanken machen. Auch wenn man mit verschiedene Schrittweiten den Core besser abdecken können, so sollten doch keine zwei Bomben aufeinander fallen. Deshalb beschränken wir die möglichen Inkremente auf Modulo acht. Die beiden Offsets sollten um vier voneinander versetzt sein.

mopt v1.2 - multiple constant optimizer for corewar (c) 1994-96 Stefan Strack

Core size: [8000]

Target size: [100]

Increment value or generator #1: 0

Offset value or generator #2: a=0,a<8000,a=a+8

Increment value or generator #2: 4

Offset value or generator #3: b=0,b<2000,b=b+8

Increment value or generator #3:

Nach dieser Eingabe wirft das Programm Zahlkombinationen aus. Da alle Kombinationen berechnet werden, kann die Berechnung etwas länger dauern. Man kann die Berechnungen mit Control-C abbrechen, und bekommt dann die Besten bisher ermittelten Kombinationen. Man muss also nicht bis zum Ende der Berechnungen warten, da auch schon am Anfang sehr gute Werte auftreten können. Mit den Werten die mopt auswirft, könnten wir unseren Dwarf noch um ein paar Prozent verbessern.

6 Beispiele

6.1 Adressierungsbeispiele

6.1.1 Direkte oder absolute Adressierung

Der Wert um den es geht, ist direkt im A- oder B-Feld abgelegt. Die Adresse, die zu bearbeiten ist, wird auf 0 gesetzt. Dadurch wird der angegebene Wert benutzt. Die unmittelbare Adressierung wird durch # gekennzeichnet.

Beispiel:

```
; Direkte Adressierung
ORG label
bomp DAT #0, #0
label ADD #2, bomp
JMP label
```

Ergebnis der Übersetzung:

```
0000 DAT.F      #0    #0
0001* ADD.AB   #2    $-1    // A[b] = A[b] + A[a]
0002 JMP.B     $-1   $0
```

Nach dem Start wird der Inhalt der Adresse, B-Teil, DAT jeweils um zwei addiert.

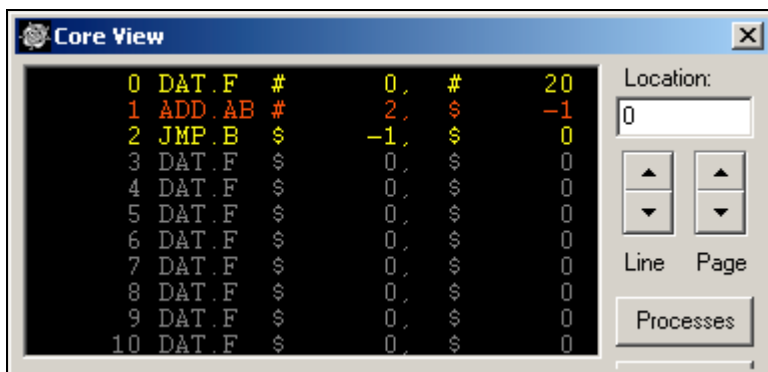


Abbildung 13 Beispiel der direkten Adressierung

Beispiel:

```
; Direkte Adressierung
ORG label
bomp DAT #0, #0
label ADD.A #2, bomp
JMP label
```

Das zweite Beispiel demonstriert die Addition in das A-Feld.

6.1.2 Relative Adressierung

Die Adresse wird direkt angegeben (Zahl von 0 bis Coresize-1). Die unmittelbare Adressierung wird durch \$ gekennzeichnet. Sollte bei der Angabe eines Feldes keine Adressierung angegeben werden, dann wird die direkte Adressierung benutzt.

Beispiel:

```

; Direkte Adressierung
ORG label
    bomp DAT #0
label ADD.A #2, $0    // Relative Adressierung $0
    JMP label

```

Mit dieser Anweisung wird das A-Feld um zwei addiert. Damit ist aber das A-Feld der aktuellen Anweisung gemeint. Besser wäre der Befehl

```
label ADD.A #2, $-1    // Relative Adressierung $-1 zum Label
```

6.1.3 Indirekte Adressierung

Die Adresse im Feld zeigt auf ein Feld das nochmals einen Index enthält. Er muss noch dazuaddiert werden. Indirekte Adressierung über das A-Feld wird durch * gekennzeichnet. Indirekte Adressierung über das B-Feld wird durch @ gekennzeichnet.

1. Beispiel:

```

; Indirekte Adressierung
ORG loop
loop MOV.i    bomb,    @bomb
    JMP loop
bomb DAT.f    #8,     #4
END

```

Die Anweisung „DAT“ wird mit dem Befehl „MOV“ kopiert. Dabei wird indirekt über das B-Feld adressiert.

NACH DEM Compiler ergibt sich:

```

0000* MOV.I    $2    @2
0001 JMP.B    $-1    $0
0002 DAT.F    #8    #4

```

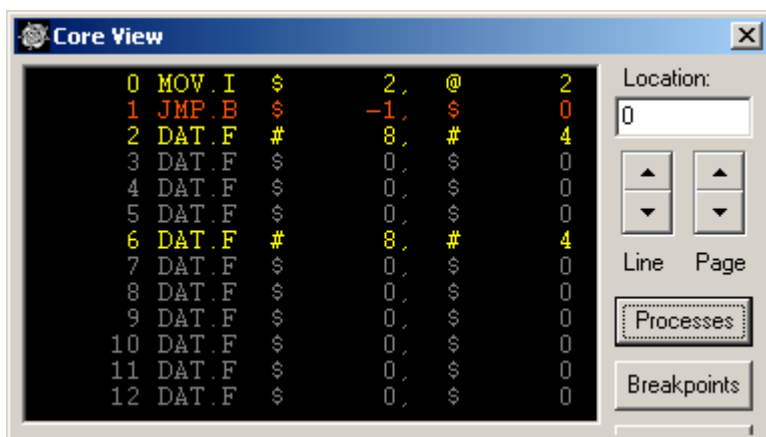


Abbildung 14 Beispiel einer indirekten Adressierung im Core

Hinweis:

Die Zieladresse zählt aber vom DAT-Feld !

Würde man das B-Feld in der DAT-Anweisung ändern, so würde die „Bombe“ sich im Core verteilen.

2. Beispiel:

```

; Indirekte Adressierung
ORG loop
loop MOV.i    bomb,    *bomb    // Adresse über A-Feld
    JMP loop

```

```
bomb DAT.f      #8,      #4
      END
```

Die Abbildung 15 zeigt die Änderung bei einer Adressierung mit dem A-Feld. Hier werden acht Befehle übersprungen.

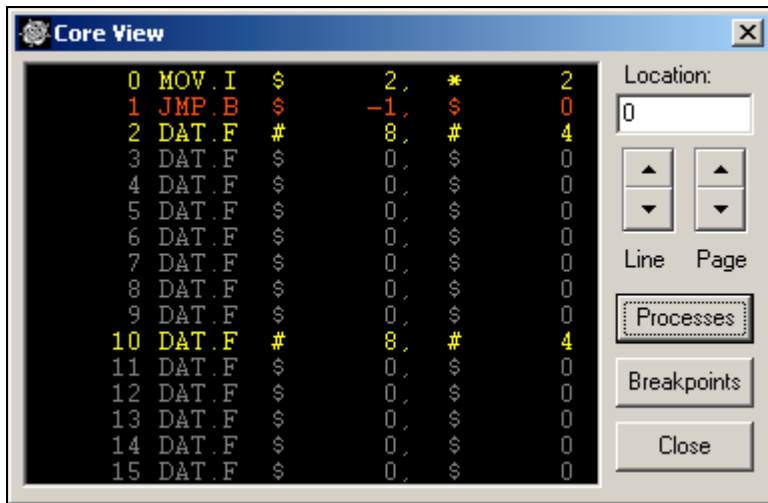


Abbildung 15 Indirekte Adressierung über das A-Feld

6.1.4 Predecrement Indirect - Predekremental Indirekt

Wie Indirekt, nur dass der Index vor der Ermittlung der tatsächlichen Adresse um eins reduziert wird. Predekremental Indirekt über das A-Feld wird mit „<“ gekennzeichnet. Predekremental Indirekt über das B-Feld wird mit „<“ gekennzeichnet.

1. Beispiel:

```

ORG start
start MOV.I <2, $3 ; relative Adresse im B-field mit predecrement des A-Felds
      ADD.F d1, start
      JMP start
      DAT #0, #0 // Dieser „Befehl wird überschrieben
d1 DAT # -50, #50
      END

```

Der MOV-Befehl hat die relative Adresse „3“. Das entspricht dem leeren Dat-Befehl. Nun ist noch die Frage, welcher Befehl kopiert werden soll. Dazu dient das A-Feld, mit dem aktuellen Wert „zwei“. Vor der Ausführung wird aber der Wert um eins erniedrigt. Damit ergibt sich folgender Core-Inhalt.

```

start MOV.I <2, $3 ; relative Adresse im B-field mit predecrement des A-Felds
      ADD.F d1, start
      JMP start
      ADD.F d1, start // neuer Inhalt
d1 DAT # -50, #50
      END

```

2. Beispiel:

```

; Beispiel 3
ORG start
Start MOV <2, 3 ; B-field indirect with predecrement
      ADD d1, start

```

```

        JMP start
        DAT      #0,  #0
d1     DAT      #-50, #50 ; speichert die Abstände
        END

```

6.1.5 Postincrement Indirect - Postinkremental Indirekt

Wie Indirekt, nur das der Index nach der Adressermittlung um eins erhöht wird.
 Postinkremental Indirekt über das A-Feld wird mit „}“ gekennzeichnet.
 Postinkremental Indirekt über das B-Feld wird mit „>“ gekennzeichnet.

6.2 Einfache Kämpfer

Dieses Kapitel zeigt eine einfache Kämpfer, vom Imp bis zum Dwarf.

6.2.1 Imp

Der „Imp“ ist das einfachste Programm im Core-Wars. Es hat nur einen Befehl.

```

ORG imp
imp  MOV imp,  imp+1
        END

```

ergibt folgenden Assemblercode:

```
0000*  MOV.I    $0  $1
```

Es wird der aktuelle Befehl in \$0 nach der relativen Adresse \$1 kopiert. Danach wird automatisch der IP weitersetzt.

Erweiterungen:

```

ORG imp
imp  MOV imp,  imp+1
        DAT #0,  #0
        END

```

Dieser Kämpfer lebt nicht sehr lange !

Ebenso dieser:

```

ORG imp
imp  MOV imp,  imp+2
        add  12,  23
        DAT #0,  #0
        END

```

6.2.2 Dwarf

Der Dwarf ist relativ klein und dementsprechend schnell. Er versucht mit Bomben den anderen Kämpfer zu treffen.

Autor: A. K. Dewdney

```

ORG loop
loop MOV.i    bomb,    @bomb    ; B-Feld indirekt
    JMP loop
bomb DAT.f    #8,      #4
END

```

Assemblercode:

```

0000*  MOV.I  $2  @2          ; $2= rel. Adresse,  B-Feld indirekt
0001   JMP.B  $-1 $0
0002   DAT.F  #8  #4

```

Wie im Kapitel 2.5.3 geschildert, wirft dieses Programm an einer bestimmten Stelle, vier Stellen nach der DAT-Anweisung eine Bombe. Dieses System kann man nun ausbauen. Dazu muss der Abstand erhöht werden.

Erste Version:

```

ORG loop
adr  DAT    #8,    #4
bomp DAT    #0,    #0
loop ADD    #4,    adr
    MOV.i  bomb,    @adr ; B-Feld indirekt
    JMP loop
END

```

Assemblercode:

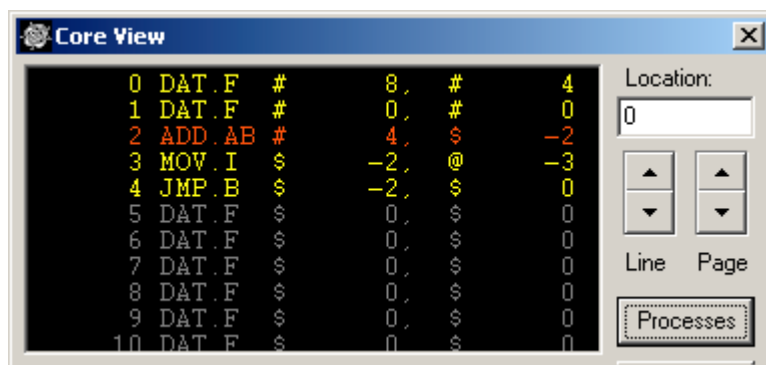


Abbildung 16 Dwarf im Core

Nach kurzer Zeit lebt der Dwarf nicht mehr. Die Ursache kann man in der nächsten Abbildung betrachten.

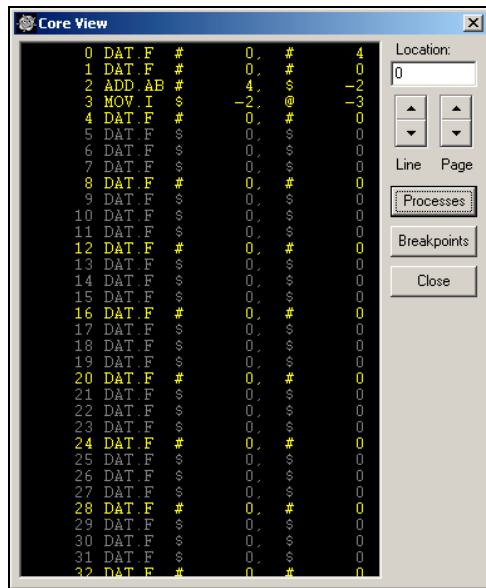


Abbildung 17 Ergebnis des ersten Dwarfs

Der Dwarf hat sich selber abgeschossen.

Abhilfe:

1) Erhöhung des Abstandes

Problem: Es kann trotzdem noch zu „Unfällen“ kommen“. Der Abstand muss in Anhängigkeit der Größe des Core, der Länge des Programms und der Position berechnet werden.

2) Verkleinern des Programms

Alte Version:

```

ORG loop
adr DAT #8, #4
bomp DAT #0, #0
loop ADD #4, adr
      MOV.i bomb, @adr ; B-Feld indirekt
      JMP loop
      END
  
```

Wichtig dabei ist die Tatsache, dass ein Programm, was auf eine DAT-Anweisung stößt immer beendet wird. Dementsprechend kann man die beiden DAT-Anweisungen zusammenlegen.

```

ORG loop
bomp DAT #0, #0
loop ADD #4, bomp
      MOV.i bomb, @bomp ; B-Feld indirekt
      JMP loop
      END
  
```

Assembler-Code:

```

0000* ADD.AB #4 $3
0001 MOV.I $2 @2
0002 JMP.B $-2 $0
0003 DAT.F #0 #0
  
```

Nun hat das Programm nur eine Länge von drei Befehlen. In jedem vierten Block wird eine Bombe geschmissen. Damit gibt es keine Probleme.

6.3 Suchen

6.3.1 B-Scanner

Mit Hilfe des Vergleichsbefehl kann man den Gegner suchen. Dabei ist es wichtig zu wissen, das der Core am Anfang immer auf „DAT #0, #0“ gesetzt wird. Damit kann man den Core mit folgendem Code durchsuchen:

```
loop  add  #10, scan
scan  jnz  loop, 10
```

Der Jmp-Zero verzweigt nur dann wieder zum Label „loop“, wenn der Inhalt des B-Felds gleich null ist. Durch die Addition mit #10 erhöht sich fortlaufend. Wenn ein Programmteil gefunden wurde, kann man es mit Bomben zerstören.

6.3.2 CMP-Scanner

Die B-Scanner überprüfen immer nur eine Speicherstelle, finden dementsprechend auch nur Gegner oder Code mit „nicht-null B-Feldern“. CMP-Scanner vergleichen einfach zwei Instruktionen im Core miteinander. Da zu Beginn des Kampfs die meisten Stellen die „DAT \$0,\$0“-Instruktion enthalten, wird der Scanner meist zwei dat \$0,\$0 beiden miteinander verglichen. Sollte eine der beiden Stellen etwas anderes enthalten, dann verzweigt der Scanner in seine Attacke und bombardiert einfach beide Positionen. CMP-Scanner sind dadurch doppelt so schnell im Suchen, brauchen aber länger, um den Gegner auszuschalten, falls sie zuerst die falsche Position bombardieren.

Beispiel:

```
...
...
loop  ADD.f  incr, cline
cline  CMP.i  $123, $134
        JMP loop
attack ...
...
incr  DAT.f  #11, #11
```

6.4 Splitten eines Programms

Mit der SPL-Anweisung kann das aktuelle Programm einen neuen Teilprozess erzeugen.

```
ORG start
start  SPL    #0,    0
        END
```

Im Debugger erkennt man, dass zwei Prozesse erzeugt werden, aber das der neue immer auf eine DAT-Anweisung läuft.

Verbesserung:

```
ORG start
start  SPL    #0
        jmp start
        END
```

Nun werden „unendlich viele“ neue Teilprozesse erzeugt.

6.4.1 Silk

Besonders bei Kämpfern der Sorte Silk ist es wichtig, dass eine bestimmte Anzahl von Prozesse erzeugt wird, die dann auch noch parallel zueinander laufen müssen. Um die Anzahl der Prozesse nicht unnötig zu vergrößern, sollte man dann auch wirklich die genaue Anzahl erreichen. Eine Technik, die genau das ermöglicht ist folgende:

Angenommen man will n parallele Prozesse erzeugen, so schreibt man die binäre Darstellung von $n-1$ angefangen mit der ersten 1 von Links, an die Stelle im Quellcode, an der die Prozesse erzeugt werden sollen. Nach jeder Ziffer wird in die nächste Zeile gewechselt. In alle Zeilen mit einem eins schreibt man dann den Befehl „spl 1“ und in alle Zeilen mit der Null den Befehl „mov.i -1, 0“. Durch das so entstandene Listing werden, wenn ein Prozeß die erste Instruktion ausführt, n parallele Prozesse erzeugt.

Beispiel:

Für einen Silk Replicator mit der Länge 7 möchte wir 7 Prozesse erzeugen. Die binäre Darstellung von 6 ist 110.

```
launch    spl 1
          spl 1
          mov.i -1, 0 ; 0
```

Mit diesem Teilcode kann man sieben Prozesse erzeugen. Leider stoßen alle dann auf die nächste Dat-Anweisung und werden beendet.

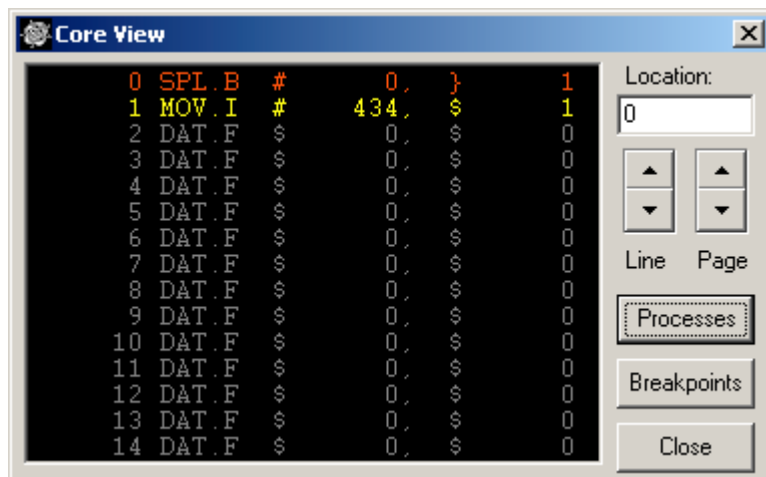
```
ORG launch
bomp     DAT     #0,      #0
launch   spl     1 ;     1
          spl     1 ;     1
          mov.i   -1,     0
loop     ADD     #4,     bomp
          MOV.i   bomb,   @bomp ; B-Feld indirekt
          JMP    loop
          END
```

6.4.2 Neue Imp-Version

Mit diesen Kenntnissen kann man auch den Imp verbessern.

Beispielcode:

```
ORG start
start   SPL     #0,      }1
          MOV.I  #1234,  1
          END
```



7 Verteidigung

Nun kommen wir zu den Verteidigungsmechanismen. Natürlich gibt es keinen Kämpfer der sich nur verteidigt, weshalb es auch schwer ist Verteidigungs-Code im Kämpfer zu lokalisieren. Verteidigt wird immer „nebenbei“, sowie das Imp-Gate bei unserem Dwarf, nur durch Design und ausnutzen von Seiteneffekten bei Befehlen lebt.

7.1 Imp-Gate

Das Imp-Gate ist eine Verteidigung gegen alle Arten von Imp's. Es besteht alleine dadurch das ein bestimmte Speicherstelle immer wieder dekrementiert wird. Der Imp, der durch die korrekte Positionierung seiner eigenen Anweisungen lebt wird "verwundet" und läuft aus. Da selbst verwundete Imp's noch eine gewisse Zeit weiter wandern sollte das Imp-Gate niemals zu nahe vor dem Kämpfer sein.

```
;Dwarf mit Imp-Gate
loop MOV.i    bomb,    @bomb
      ADD.ab   #4,      bomb
      JMP loop, <loop-10 ;Predekrement B-Feld wird ausgeführt.
bomb DAT.f    #0,      #0
```

Dieser Dwarf ist das typische Beispiel für „Verteidigung nebenbei“. Die jmp-Instruktion führt die Dekrementoperation aus, obwohl sie im B-Feld eigentlich nicht benutzt wird. Wenn man irgendwo im Code ein freies B-Feld bei einer jmp.- oder spl-Instruktion hat, dann ist es immer eine gute Idee dort ein kleines Imp-Gate zu installieren.

7.2 Decoy

Je nach den Regeln des Hill's auf dem der Kämpfer antritt darf er eine bestimmte Länge nicht überschreiten. Aber alles was bis zu dieser Länge geht ist erlaubt. Man kann zwischen den einzelnen Modulen des Kämpfers Freiräume schaffen, die eine Verteidigung gegen verschiedene Arten von Gegnern bieten. Freiräume mit nicht-null B-Feldern werden sicherlich B-Scanner dazu veranlassen ihre Bomben zu schmeißen. Kleinere Freiräume zwischen den Modulen und auch innerhalb der Module vergrößern die Chance das ein bombardierender Gegner nur Instruktionen zerstört die nicht gebraucht werden.

7.3 Bomber-Dodger

Dies ist wahrscheinlich eine Weiterentwicklung des Decoy. Die Idee ist, den Bombenteppich des Dwarf zu scannen und kleine Kämpfer die 1 oder 2 Instruktionen große Decoys' enthalten, so zu platzieren, dass die Bomben des Dwarf's in die Lücken fallen. In der ersten Phase muss der Kämpfer möglichst schnell die Bomben erwischen. Danach den oder die Programmteile dementsprechend verschieben und starten.

7.4 Colour

Eine Verteidigungstechnik, die ein Bomber, also auch unser Dwarf gegen Scanner anwenden kann, sind Bomben die eine „Farbe“ besitzen. B-Scanner oder auch cmp-Scanner sprechen auf nicht leere Felder an. Dat 0,0-Instruktionen als Bomben sind für die Scanner unsichtbar. Schmeisst der Dwarf mit Dat 5,5 oder allem anderem mit nicht-null A- und B- Feld, so wird der Scanner, wenn er nicht entsprechend spezialisiert ist, die Bomben des Dwarf's als Ziel erkennen. Das gibt dem Dwarf viel Zeit um den Scanner zu erwischen.

7.5 Bootstrapping

Bevor das Modul eines Kämpfers, welches den eigentlichen Angriff führt gestartet wird, wird es an eine andere Stelle kopiert. Wenn der Kämpfer aus mehreren Modulen besteht kann das mit diesen ebenfalls passieren. Das Ziel ist es, einen großen Strohmann, den die Scanner finden und beschießen, im Speicher zu lassen, während die kleinen Kämpfer über den gesamten Core verteilt sind, ein weniger gutes Ziel abgeben und dadurch besser überleben. Das Zusammenarbeiten der einzelnen Module, beziehungsweise das nicht überschreiben oder Angreifen der einzelnen Teile, erfordert natürlich etwas mehr Arbeit beim Entwickeln des Kämpfers.

7.6 Mirror oder Reflection.

Eine Technik gegen sogenannte On-axis Scanner. Das sind cmp-Scanner die einen Abstand von einer halben Corelänge ange zwischen den beiden zu vergleichenden Instruktionen haben.

```
incr  DAT.f#4,  #4
loop  ADD.f$incr, $loop+1
      CMP.i $10,  $4010
      JMP loop
```

Ein Kämpfer der sich mit dieser Technik verteidigt, legt eine exakte Kopie von sich selbst genau eine halbe Corelänge vor sich. Dadurch vergleicht der cmp-Scanner wieder zwei gleiche Felder und geht davon aus das es sich um dat 0,0-Instruktionen handelt. Der Kämpfer ist unsichtbar, jedenfalls für cmp-Scanner. Wenn zusätzlich noch die vom Kämpfer veränderten Variablen etwas ausserhalb liegen, bzw. ein Decoy mit verschiedenen Werten, oder sogar einer dritten Kopie angelegt wird, wird der Scanner sicherlich dort angreifen.

7.7 Stealth

Gegen B-Scanner kann man sich schützen in dem man Module des Kämpfers nur aus Instruktionen mit null-B-Felder erstellt. Diese Module in Verbindung mit einem „nicht-null-B-Feld Decoy“ veranlasst die meisten B-Scanner wahrscheinlich immer „danebenzuschießen“. Auch wenn es nicht elegant ist, so kann man gegen einen B-Scanner mit folgender Instruktionen, die an ein paar Stellen im Speicher verteilt und ausgeführt wird, zumindest sehr oft ein Unentschieden herausholen.

```
spl 0
```

Das erzeugt bei jedem Aufruf einen neuen Prozess während der alte stirbt. Ausserdem wird ein Gegner der durch diese Bombe „versehentlich“ getroffen wurde, stark verlangsamt, da bei ihm alle neuentstehenden Prozesse weiterleben. So mancher Kämpfer bringt sich alleine durch die Timingprobleme schon selbst um. Unser Dwarf würde sich durch eine zusätzlichen Prozess wahrscheinlich selbst bombardieren.

7.8 Self splitting

Wenn man in einem Kampfmodul eine nicht zu lange Angriffsschleife hat, dann kann man sie mit einem jmp-Befehl, der vom Ende wieder zum Anfang verzweigt, gestalten. Eine andere Möglichkeit, zeigt sich wohl am Besten durch folgende Gegenüberstellung.

Kämpfer A	Kämpfer B
SPL 0, 4	MOV 4, 4
MOV -2, @-1	ADD #4, -4
ADD #4, -1	JMP -2

Diese beiden Kämpfer tun im Prinzip das Gleiche. Wird aber Kämpfer B durch eine dat-Bombe getroffen, so stirbt er an dem Versuch diese auszuführen. Kämpfer A hingegen verträgt zumindest dat-Bomben auf seine zweite und dritte Position. Er bombardiert zwar nicht mehr, aber er überlebt es. Wichtig ist hierbei das Kämpfer A praktisch rückwärts abgearbeitet wird. Während bei Kämpfer B der Prozeß-Counter von einer Instruktion zur

nächsten, bzw. an den Anfang der Schleife gestellt wird, so werden die einzelnen Prozesse nacheinander, der erzeugte Prozess immer nach dem erzeugenden abgearbeitet. Das heißt das nach dem Prozess A mit der spl-Instruktion Prozeß B erzeugt hat, wird er erst die mov-Instruktion ausführen, bevor B mittel spl C erzeugt. Bei solch einfachen Kämpfern spielt das wohl nur eine untergeordnete Rolle, aber wenn der Code größer wird, kann eine kleine Unachtsamkeit den Kämpfer sehr instabil machen.

7.9 Airbag

Diese Technik geht noch einen Schritt weiter als Selfsplitting. Durch geschicktes Manipulieren von Feldern, kann ein Kämpfer erkennen, ob er von einer Dat-Bombe getroffen worden ist. Folgendes Beispiel ist zwar sehr vereinfacht, funktioniert aber.

```
Incr  DAT.f  #1,    #3044
bomb  DAT.f  #0,    #0
loop  MOV.i  bomb,  @bomb
      ADD.f  incr,  bomb
      JMZ   loop,  {bomb
```

Dieser Code ist der eine Teil der Technik. Der zweite, wichtigere Teil ist die Anzahl und die Ausführungsreihenfolge der Prozesse, die in diesen Kämpfern arbeiten. Kämpfer mit dieser Art von Airbag, werden mit einem Prozess mehr ausgeführt, als sie Instruktionen in der Schleife haben. Zudem müssen die Prozesse den Kämpfer so abarbeiten, wie es ein Prozess tun würde. Dazu wird eine zusätzliche Instruktion, die nur Anfang gebraucht wird in den Kämpfer eingefügt. In unserem Fall ist es die NOP-Instruktion. Das Verteilen der Prozesse besorgt ein Vector-Launch.

```
;redcode
;name Airbag Stone
;assert CORESIZE==8000
  bomb  DAT.f  #jinst,  #0
  incr  NOP.f  #1,    #3044
  loop  MOV.i  bomb,  @bomb
        ADD.f  incr,  bomb
        JMZ.a loop,  {bomb
  Jinst JMP    0,    incr      ; Ausführung wenn DAT-Treffer erkannt.
  Launch SPL 1
        SPL 1
        JMP    >jinst
  end launch
```

Der Kern des Ganzen ist das Zusammenspiel der Instruktionen bei loop+1 und loop+2. Die jnz-Instruktion verzweigt nur dann nach loop, wenn im A-Feld der Instruktion auf die das A-Feld von bomb nach vorhergegangener dekrementierung zeigt, Null ist. Das ist aber nur der Fall, wenn die add-Instruktion den Zeiger vorher Inkrementiert, also verstellt hat. Sollte also eine dat-Instruktion in den Kämpfer als Bombe einschlagen, so läuft alles erstmal normal weiter. Wenn dann der letzte Prozess die jnz-Instruktion ausführt, gab es keinen Prozess der mit der add-Instruktion die richtige Ausgangssituation geschaffen hat und die Bedingung trifft nicht zu. So kann dann in den nächsten Teil verzweigt werden. Hier wird meistens ein Coreclear gestartet. Sinnvoll wäre es aber, auch ein Paper zu starten, da man davon ausgehen kann, dass man einen Bomber also einen Stone zum Gegner hat. Mit ein bisschen Fingerspitzengefühl lässt sich dieser Airbag in fast jeden Kämpfer der mit einer Schleife arbeitet einbauen. Wenn man die Technik noch weiter durchdenkt, dann könnte man auch jmp-, oder spl-Bomben erkennen und entsprechend reagieren. Airbag werden meist mit einer Kombination aus Postinkrement- und Predekrement-Adressierungen realisiert, das so noch ein weitere Bombe geschmissen werden kann.

7.10 Brainwashing

Im Zuge von der Einkehr des pSpace kam diese Verteidigungstechnik beinahe zwangsläufig auf den Plan. Da man nur bei identischer PIN-Nummer auf den selben pSpace wie der Gegner zugreifen kann, wird der Gegner dazu gebracht einige Instruktionen im Speicher auszuführen. Diese Instruktionen veranlassen oft das Löschen

des pSpace oder das Verändern einzelner Stellen. Sollte der Gegner sich allerdings schon „entschieden“ haben mit welcher Technik er kämpft, dann ist das Brainwashing für diese Runde zu spät. Erst in der nächsten Runde kann der Kämpfer der mit Brainwashing verteidigt auf Erfolge hoffen.

8 Links

- <http://www.corewars.org/>
- <http://www.corewars.org/docs/dummies.html>
- <http://www.stormking.com/~koth>
- Steve Bailey 101374.624@compuserve.com
- sgb@zed-inst.demon.co.uk
- <http://ourworld.compuserve.com/homepages/SGBailey>
- <http://mcraeclan.com/Graeme/CoreWars.htm>
- <http://corewars.sourceforge.net/>

9 Indexverzeichnis

A

Adressierungsarten	19
Direct.....	19
Immediate.....	19
Indirect	20
Postincrement Indirect.....	20
Predecrement Indirect	20
Adressierungsbeispiele	42
Arithmetische Operatoren.....	12

B

Befehle.....	23
ADD.....	24
CMP	27
DAT	23
DIV	25
DJN	27
JMN.....	26
JMP	26
JMZ.....	26
LDP	30
MOD	26
MOV	24
MUL.....	25
NOP.....	29
SEQ.....	29
SLT.....	27
SNE.....	29
SPL.....	28
STP.....	30
SUB.....	25
Beispiele	42
Breakpoints.....	6
B-Scanner	48

C

CMP-Scanner	48
Core View.....	8
Corestep.....	38

D

DJN-Stream	36
Dwarf.....	45

E

Effektive Bomben.....	36
Einfache Kämpfer.....	45

F

For-Schleife.....	14
-------------------	----

G

GUI

Breakpoints	6
Core View	8
Drei Arten des Spiels.....	8
Edit Core	10
Ergebnisse	8
Haltepunkte	6
Optionen.....	5
Prozesse.....	9
P-Space View	11

H

Haltepunkte.....	6
------------------	---

I

Imp.....	45
----------	----

K

Kommentare	17
assert.....	18
author	18
date	18
kill	18
name	18
redcode.....	17
strategy	19
version.....	18
Konstanten	
CoreSize	16
CURLINE	17
MAXCYCLES	16
MAXLENGTH.....	16
MAXPROCESSES	16
MINDISTANCE	16
PSPACESIZE.....	16
ROUNDS	16
VERSION	17
WARRIORS.....	17

L

Label.....	12
Links	54

M

Modifizierer	21
mOpt.....	40

O

Operatoren	12
------------------	----

P

Paralle Prozesse	35
PIN	15
Programmieren	12, 35
Prozesswarteschlange	33
P-Space	31

R

Redcode	12
---------------	----

S

Schrittweite	37
Silk	49
Splitten eines Programms	48
Suchen im Core	48

T

Trickkiste	35
------------------	----

V

Vergleichsoperatoren	12
Verteidigung	51
Airbag	53
Bootstrapping	52
Brainwashing	53
B-Scanner	51
Colour	51
Decoy	51
Imp-Gate	51
Mirror	52
Reflection	52
Self splitting	52
Stealth	52
vordefinierten Konstanten	16