CoreWar
Krieg der Kerne
Handbuch für Einsteiger.

Sascha Zapf

Version 0.7 Dezember 2002 / Januar 2003

Inhaltsverzeichnis

1. Über dieses Handbuch - Vorwort zur ersten Version

Ich selbst bin nicht allzu lange dabei, aber kenne Corewars schon länger. Ich schreibe dieses Handbuch aus verschiedenen Gründen.

- Es gibt viele gute Artikel und Berichte über CoreWar. Allerdings sind diese mit den Feinheiten des Spiels und dem CoreWar eigenen Slang gespickt. Hinzu kommt das praktisch alles in Englisch geschrieben ist. Man bleibt irgendwie aussen vor. Dadurch wird es sehr schwer alles zu verstehen bzw. den richtigen Einstieg zu kriegen. Mir ging es vor 2 Jahren genauso. Ich habe damals leider aufgegeben bevor ich ein gewisses Verständnis erreicht hatte.
- Ich möchte selber weiter lernen. Die beste Möglichkeit etwas zu lernen, ist für mich, zu lehren. In die Lage versetzt anderen korrekte Tatsachen präsentieren zu "müssen", muß ich mich tiefer denn je in den Standart einlesen. Muß ich einen Kampf schrittweise verfolgen. Wenn es um einen selbst geht, dann nimmt man schnell auch mal ein paar Tatsachen als gegeben an.
- Ich möchte die Hilfsbereitschaft die mir in der Diskussiongruppe `rec.games.corewar` entgegengebracht wurde mit anderen teilen. Also an dieser Stelle ein riesengroßes Dankeschön an alle aktiven und nicht-immer-aktiven in dieser Newsgroup.
- Dieses Handbuch soll aber auch ein bißchen von der Faszination CoreWar rüberbringen. Falls es zwischendurch ein wenig unsachlich klingt, falls Prozesse sterben oder Kämpfer betäubt werden, dann bitte ich das zu entschuldigen.

Also, ich bin alles andere als der erfahrene CoreWar-Spezialist. Ich möchte meine Erfahrungen und die Lösungen der Probleme in meiner Anfangszeit, die ja immer noch andauert, mit anderen teilen. Ich bin auf jeden Fall auf dem Weg zu den Hill's. Wer kommt mit?

Sascha Zapf
Köln im Januar 2003

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

2. Was ist CoreWar.

CoreWar ist ein Spiel oder ein Wettkampf für Leute die Spaß daran haben einen Gegner mit Mathematik, selbstmodifizierendem hochoptimierten Assemblercode und viel Einfallsreichtum zu bekämpfen. Es ist nicht nur ein Spiel für Programmierer, aber ich kann mir leider nur schwer jemand der nicht programmiert dabei vorstellen. Aber es ist auf jedem Fall einen Versuch Wert. CoreWar ist für heutige Maßstäbe schon relativ alt. Im Mai 1984 von A. K. Dewdney als Zerstreuung für Programmierer erfunden, war es aber ein Signal an die ganze Welt. Computer-Viren waren dabei sich einen Namen zu machen. Dewdney zeigte durch sein "Spiel" welche zerstörerische Kraft in nur wenigen Anweisungen stecken kann. Dadurch das die Kämpfer nur in einem virtuellen Computer existieren können, wurden die Bedenken der Leute wohl zerstreut.

Im Internet existieren verschiedene Server die einen oder mehrere Hill's betreiben. Dorthin kann man seine Kämpfer per eMail schicken und ihn gegen andere digitale Kämpfer antreten lassen. Die Formulierung Hill kommt daher das Kämpfer die nicht gut genug sind nicht an die Spitze der Rangliste gelangen können. Wenn ein Kämpfer älter ist, dann wird er wohl oder übel von den jüngeren oder den besseren, besiegt und vom Hill geschmissen.

Auch Zuhause kann man die Kämpfe austragen lassen. Nötig ist dazu nur ein Editor für Ascii-Text und eine MARS. Das ist ein virtueller Computer auf dem der Redcode, das ist Sprache in der die Kämpfer geschrieben werden, ausgeführt wird und die Kämpfer agieren können.

Der Speicher der MARS ist eindimensional und als Ring angelegt, so das kein Kämpfer oder Daten des Kämpfer vorne oder hinten rausfallen kann. Hinter der letzten Stelle kommt die erste.

Als Quasistandart hat sich die pMars etabliert. Sie hat einige Experimentelle Optionen und Befehle welche nicht im 94'er Standart verankert sind, welche aber von den 94'er Hills im Internet alle akzeptiert werden. Es gibt vereinzelt Hill's die nur Kämpfer die nach dem 88'er Standart programmiert wurden, akzeptieren. Dieses Handbuch erläutert CoreWar nach dem 94'er Standart mit den Erweiterungen der pMars 0.8.

3. Was ist Redcode?

Redcode ist eine im Umfang reduzierte Assemblersprache. Einige Befehlsgruppen sind komplett vorhanden, andere nur Teilweise. Registeroperationen oder ein Akkumulator fehlen auffallend. Insgesamt sind es momentan, nach dem ICWS'94 Standart mit den Erweiterung durch pMars, 19 Befehle. Jeder Befehl, oder besser jede Instruktion besteht aus ihrem Opcode, dem Modifizierer des Opcodes dem A-Feld und dem B-Feld mitsamt ihrer Adressierungsarten. Nicht jede Instruktion hat alle Felder. Desweiteren gibt es sogenannte Pseudo-Opcodes. Das sind Befehle die dem Ausführen der ersten Anweisung, das Ende des Quelltextes oder der Namensgebung für bestimmte numerischen Konstanten dienen. Jede Zeile im Quelltext kann mit einem Label bezeichnet werden, auf die sich dann auch die Instruktionen beziehen. Das macht die Arbeitsweise des Kämpfer um einiges leichter zu verstehen. Als dritte im Quelltext anzutreffende Erscheinung sind die Kommentare zu nennen. Ein Semikolon leitet einen Kommentar ein. In den Kommentaren gibt es spezielle Schlüsselwörter die z.B. der Benennung des Kämpfer oder des Autors dienen. Auch die Spezifikation der Umgebung für die der Kämpfer geschrieben worden ist kann in Kommentaren festgelegt werden.

4. Einfache Kämpfer.

Ich habe fast 10 Jahre in Assembler programmiert. 6 Jahre auf dem C64, Demo's und Intro's. 4 Jahre auf dem Amiga. Ich habe einige Tricks kennengelernt wie man kurz und schnelle Programme erstellen kann. Aber das man einen Kämpfer mit nur einem Befehl programmieren kann, das hätte ich mir nicht vorstellen können. Und doch geht es.

4.1. Der Imp

Durch die relative Adressierung innerhalb des Core's ist folgender Kämpfer möglich.

```
        mov 0,1
```

Oder, mit Label zur besseren Lesbarkeit.

```
        org imp

imp     mov.i  imp,imp+1

        end
```

Was macht dieser Kämpfer wenn er gestartet wird? Die mov-Instruktion verschiebt etwas, das ist klar. Der Modifizierer **.i** weist die mov-Instruktion an, die gesamte Instruktion zu verschieben. Die Adresse 0, beziehungsweise die Adresse des Labels **imp** ist die Instruktion selbst. Adresse 0 ist immer die aktuelle Instruktion. Adresse 1 ist die nächste Instruktion. Dort ist aber zur Zeit noch keine gültige Instruktion, nur der mit `dat $0,$0` vorinitialisierte Core. Die mov-Instruktion kopiert nun sich selbst auf die nächste Instruktion. Die Instruktion hat damit ihre Aufgabe erfüllt. Als letzte Tätigkeit wird nun noch die Adresse der Instruktion+1 zurück in die Prozeßwarteschlange gesteckt. Als nächste Instruktion wird die eben kopierte Instruktion ausgeführt. Die Instruktion verhält sich natürlich wie die erste. Instruktion für Instruktion kriecht der Imp (Knirps) durch den Speicher. Er tut das mit einer Geschwindigkeit die im CoreWarjargon 'c' (Lichtgeschwindigkeit) genannt wird. Bei CoreWar werden Geschwindgkeitsangaben meist mit der c-Notation angegeben. 100%c heißt 1/Zyklus, schneller geht es eigentlich nicht. Der Imp bewegt sich pro Zyklus eins weiter, also mit 100%c oder einfach c. Was passiert wenn der Imp auf einen Gegner trifft sehen wir nach der nächsten Vorstellung.

4.2. Der Dwarf

Wenn man mit einer Instruktion schon etwas konstruieren kann was kämpft, dann wird es bei drei Instruktionen schon etwas interessanter. Die vierte Instruktion (dat) ist eigentlich nicht nötig, wird aber trotzdem zum besseren Verständnis mit aufgeführt. Erst mal der Redcode

```
        add #4, 3
        mov 2, @2
        jmp -2
        dat #0,#0
```

oder

```
        step EQU 4

        ORG loop

loop    add.ab #step,   bomb
        mov.i  bomb,    @bomb
        jmp    loop
bomb    dat.f  #0,      #0

        END
```

Beschreiben wir die Aufgabe jeder Instruktion nacheinander.

add #4, 3 Die add-Instruktion addiert den Wert im A-Feld zu dem Wert auf den das B-Feld zeigt. In diesem Fall wird die 4 zur 0 in der dat-Instruktion addiert. Beim nächsten Durchlauf 4 zur 4 dann 4 zur 8 usw. Also, aus dat #0,#0 wird dat #0,#4.

mov 2, @2 Die Instruktion 2 Stellen weiter wird indirekt über das B-Feld der Anweisung zwei Stellen weiter kopiert. Im ersten Durchlauf steht dort die 4. Sie wird dadurch 4 Stellen weiter in den Core kopiert.

jmp -2 Ein unbedingter Sprung zur ersten Zeile startet das ganze von neuem.

dat #0, #0 Der dat-Befehl dient als Bombe und als Positionsgeber für die Bombe.

Was macht dieser Kämpfer? Er schmeißt an jede vierte Stelle im Core eine Bombe, die jedesmal veränderte dat-Instruktion. Die dat-Instruktion kennzeichnet Daten im Redcode. Ausserdem ist die dat-Instruktion ist eine Möglichkeit einen Prozeß zu eliminieren. Ein Prozeß der versucht eine dat-Instruktion auszuführen stirbt daran. Egal welche Techniken man anwendet um den Gegner zu finden, zu betäuben, zu manipulieren oder sonst etwas, getötet wird er nur dadurch das er versucht eine dat-Instruktion auszuführen.

Der Dwarf schmeißt also dat-Bomben an jede vierte Stelle im Core. Mit etwas Glück trifft er den Gegner bevor er ihn bekommt. Führt der Gegner diese Instruktion aus,

verliert er einen Prozeß. War es sein einziger dann war es das für den Kämpfer. Trifft er nicht dann hat er vielleicht keine Chance mehr. Wenn er die 8000 Stellen im Speicher beinahe überwunden hat, macht es sich bezahlt das wir 4 als Abstand der Bomben gewählt haben. In dieser Phase schmeißt er eine Bombe direkt vor seine add-Instruktion und die nächste landet auf seiner Bombe. Damit ist die kritische Phase dann auch schon wieder vorbei.

Noch einmal kurz zu der c-Notation. Der Dwarf wirft eine Bombe in einer Schleife von drei Befehlen, also mit 33%c.

4.3. Imp versus Dwarf

Lassen wir die zwei mal gegeneinander antreten. Den schleichenden Imp kann man nur mit einem Treffer an seine Spitze vernichten. Sollte der Dwarf das Glück haben, und trifft das vorderste mov 0,1 des Imp vor dessen Ausführung, so stirbt der Imp. Ein in größeren Abständen bombardierender Dwarf hätte eventl. noch ein zweite Chance auf einen direkten Treffer, da er den Core in weniger Zyklen umrundet. Aber unser Freund arbeitet nun mal mit vier. Davon ausgegangen den Imp haben die Bomben passiert und er lebt noch, dann kriecht er weiter auf den Dwarf zu. Sobald er ihn erreicht überschreibt er eine Anweisung nach der anderen. Spätestens nach dem unbedingten Rücksprung in Zeile 3 ist der ehemalige Dwarf ebenfalls ein Imp geworden. Der Kampf kann nur noch unentschieden ausgehen.

5. Entwicklung

Diese Situation ist in Corewars eigentlich nicht üblich. Normalerweise kennt man sein Gegner nicht "persönlich". Den Programmierer des Kämpfers vielleicht. Aber man bekommt den Quelltext eines Kämpfer meist erst zu Gesicht wenn er vom Hill geschmissen wurde. In der Vergangenheit, um die Zeit '94 bis '97 wurde zwar einige Kämpfer im Corewarrior veröffentlicht die noch aktiv waren, aber das waren Ausnahmen. Wir gehen hier einfach davon aus das die Programmierer von Imp und Dwarf befreundet sind und sich gegenseitig in den Quelltext schauen lassen.

5.1. Der Dwarf rüstet auf

Der Programmierer des Dwarf ist natürlich nicht zufrieden mit seinem Kämpfer und möchte den Imp gerne immer stoppen. Der Dwarf könnte seine Bomben an jede Stelle des Core's schmeißen. Allerdings, wenn er das vorwärts macht, könnte er den Imp Aufgrund seiner geringen Geschwindigkeit nicht ein-. bzw. überholen. Also muß er rückwärts arbeiten. Aber selbst mit einer Geschwindigkeit von c wäre es nicht 100% sicher das er den Imp voll erwischt. Leider ist unser Dwarf aber nicht so schnell, also fällt der Angriff aus wie vorher nur Rückwärts. Das wichtigste am Imp ist das Ziel seiner Kopieraktion. Wenn das Ziel nicht die nächste Instruktion im Speicher ist legt er sich nicht mehr selber seine Instruktionen bereit, sondern muß ausführen was dort liegt, bzw. andere dort haben liegen lassen. Den Dwarf um eine Instruktion vergrößern um bei jedem Schleifendurchlauf das B-Feld der letzten Instruktion vor seinen Instruktionen auf etwas anderes als 1 zu setzen, würde:

1. die Schleife des Dwarf um eine Anweisung vergrößern und damit die Bombardiergeschwindigkeit des Dwarf's auf 25%c reduzieren.

2. den Imp trotzdem nur mit 25% Wahrscheinlichkeit stoppen können.

3. den Imp in den meisten Fällen bei erreichen des Dwarfs selber zu einem Dwarf machen, womit es wieder zum unentschieden kommen würde.

Um die Instruktion vor dem Dwarf effektiv und stetig zu verändern müssen wir erreichen das mit jeder Instruktion des Dwarfs auch an diesem B-Feld operiert wird.

Als erstes können wir die sub-Instruktion, die ja die add-Instruktion ersetzt nicht auf der dat-Bombe im Code des Dwarf sondern im Core davor arbeiten lassen. Dazu müssen wir den Abstand der Bomben aber auf 5 erhöhen, da der Dwarf wächst. Desweiteren gibt es neben der indirekten auch noch die indirekt-predekrement-Adressierung, die die indirekte Sprungadresse vor dem Sprung dekrementiert. Wenn die mov-Instruktion

damit arbeitet, würden schon zwei Befehle an diesem B-Feld manipulieren. Die jmp-Instruktion schafft es durch eine besonderheit des Redcode. Die Predekrement-. bzw. Postinkrementfunktionen werden auch bei Befehlen oder in Feldern in denen sie eigentlich nichts bewirken sollten, ausgeführt. Selbst eine nop-Instruktion, die ja eigentlich nichts macht, wird in der folgenden Form A und B-Feld der Instruktion hinter sich inkrementieren.

```
        nop }-1,>-1
```

Somit kann der Dwarf mit jeder Instruktion die er ausführt das B-Feld der letzten Instruktion vor sich manipulieren. Die zwischenzeitlichen dekrementierungen müssen bei berechnung des nächsten Bombenziels natürlich berücksichtigt werden. Ausserdem haben wir den Start des Dwarf in die Schleife verlegt. Damit haben wird den neuen Dwarf.

```
        ORG     start

        dat     #0,     #-4
bomb    dat     #0,     #0
loop    sub.ab  #3,     bomb-1
start   mov.i   bomb,   <bomb-1
        jmp     loop,   <bomb-1

        END
```

Spielen wir jetzt nochmals entscheidende Phase des Kampfes durch. Der Dwarf hat den Imp mit seinen dat-Bomben verpasst. Der Imp hat seine Spitze gerade nach `bomb-1` kopiert. Folgende Situationen können eintreten.

1. Der Dwarf führt die sub-Instruktion aus. Das B-Feld der Spitze des Imp wird von 1 auf -2 geändert. Der Imp führt das aus, kopiert sein Spitze drei Instruktionen nach hinten. Damit überschreibt er nicht die dat-Bombe. Bei der nächsten Instruktion wird der Imp versuchen die Bombe auszuführen - Ruhe in Frieden!

2. Der Dwarf führt die mov-Instruktion aus. Diese dekrementiert die Adresse zuerst, bevor indirekt über sie, kopiert wird. Das B-Feld der Imp-Spitze ist 1, dekrementiert wird es zu 0. Das heißt das der Dwarf die Bombe sofort auf die Spitze des Imp schmeißt. Mit der nächsten Instruktion die der Imp ausführen wird ist sein Ende besiegelt.

3. Der Sprungbefehl veranlasst den Imp durch die dekrementierung des B-Feldes dazu die Spitze auf sich selbst zu kopieren. Damit tritt der gleiche Effekt wie bei 1 ein. Der Imp legt sich nicht mehr die nächste Instruktion zurecht und läuft auf die dat-Bombe.

Der Dwarf gewinnt immer. Trotz des intensiven Schutz gegen den Imp hat sich an seiner 33%c Bombardiergeschwindigkeit nichts geändert.

5.2. The Impire strikes back

Was kann am Imp getan werden damit er wenigstens wieder auf das unentschieden kommen kann?

```
        ORG imp

imp     mov 0,2
        mov 0,2

        END
```

Nun, das sieht nicht besonders spektakulär aus, aber betrachten wir die entscheidende Situation nochmals, immer davon ausgegangen der Dwarf hat den Imp mit seinen Bomben verfehlt.
Der Imp hat seine Spitze nach `bomb-1` gelegt:

1. Der Dwarf führt seine sub-Instruktion aus. Aus der Instruktion `mov 0,2` wird `mov 0,-1`. Die nächste Instruktion die der Imp ausführt liegt aber bei `bomb-2`. Also wird die Bombe des Dwarfs durch ein `mov 0,2` überschrieben. Der Dwarf schmeißt diese unwirksame Bombe über das durch die sub-Instruktion und sein Predekrement veränderte B-Feld der alten Imp-Spitze, wo jetzt -2 steht. Jetzt führt der Imp die Instruktion aus. Der Dwarf springt zu seiner sub-Instruktion. Der Imp führt nun die Instruktion die auf Bomb liegt aus, und überschreibt die mov-Instruktion des Dwarf mit seiner eigenen. Der Dwarf schmeißt keine Bomben und ist damit unschädlich gemacht. Der Imp, kommt kurz danach über die sub-Instruktion zur mov-Instruktion, schmeißt sich selbst hinter die jmp-Instruktion, wird diese aber nicht erreichen. Er wird wie der Dwarf in einer Endlosschleife bis zum unentschieden ausharren.

2. Der Dwarf führt seine mov-Instruktion aus. Vor der eigentlichen Kopieraktion wird das Predekrement ausgeführt. Aus der Instruktion `mov 0,2` wird `mov 0,1`. Die mov-Instruktion schmeißt die Bombe auf die Stelle an der die Bombe bereits liegt. Danach führt der Imp den Befehl bei `bomb-2` aus und überschreibt die Bombe. Der Dwarf springt zurück, wobei er durch das Predekrement die `mov 0,1` Anweisung zu `mov 0,0` verwandelt. Der Imp läuft über das `mov 0,0` und danach verwandelt er sich wie der Dwarf in einen ungefährlichen mov-Bomben-Werfer.

3. Die Situation läuft auf das gleiche hinaus wie b.

Bei 100 Kämpfen kommt dann auch dieses Ergebnis heraus. Der Dwarf gewinnt ca. 20% der Kämpfe durch direkte Treffer. Alle anderen Kämpfe enden unentschieden. Allerdings ist sichtbar das der Imp durchaus verbessert werden kann. Wenn die Spitze des Imp etwas vor seiner Ausführungsposition liegt, dann ist er etwas stabiler. Wie weit kann man das ausbauen?

```
     org imp

imp  mov 0,3
     mov 0,3
     mov 0,3

     end
```

Ausgangsituation Dwarf verfehlte Imp, Imp legte seine Spitze auf `bomb-1` der Dwarf führt seine sub-Instruktion aus. Die Instruktion `mov 0,3` auf `bomb-1` wird zu `mov 0,0`. Der Imp führt seine Instruktion bei `bomb-3` aus (Die Imp-Instruktion bei `bomb-4` legte die Instruktion auf `bomb-1`). Die Bombe wird durch ein `mov 0,3` ersetzt. Die schmeißt der Dwarf dann auch indirekt predekrement über `bomb-1` nach `bomb-2` (Wieso?). Nun ist der Imp mit seinem Befehl bei `bomb-2` dran. Schauen wir uns den Dwarf jetzt mal an.

```
        mov     0,      3           ; <---- Hier ist der Imp ---
        mov     0,      -1
bomb    mov     0,      3
loop    mov     0,      3
start   mov     bomb,   <bomb-1
        jmp     loop,   <bomb-1     ; <---- Hier arbeitet der Dwarf ---
```

Die jmp-Instruktion verzweigt nach `loop` (predekrement) wo er die neue Spitze des Imp vorfinden wird. Inzwischen führt der Imp das für das Geschehen uninteressante Instruktion bei `bomb-1` aus. Der Dwarf kopiert ein `mov 0,3` hinter die jmp-Instruktion. Mit der Anweisung bei `bomb` ersetzt der Imp die jmp-Instruktion durch sein `mov 0,3`. Wichtig ist, das in diesem Imp Ring eine Anweisung fehlt (Was den Imp im Moment noch nicht stört). Die mov-Instruktion schmeißt eine unschädliche Bombe aus dem Geschehen heraus. Der Imp führt nun ebenfalls die Anweisung bei `loop` aus und verändert damit die Situation aber nicht. Schauen wir uns den Dwarf nun nochmal an.

```
        mov     0,3
        mov     0, - irgendetwas
bomb    mov     0,3
loop    mov     0,3                 ; <--- Hier ist der Imp -----
start   mov     bomb, <bomb-1       ; <--- Hier ist der Dwarf ---
        mov     0,3
        mov     0,3
```

Wir können auf diesem Ausschnitt das Loch in dem Impring bereits erkennen. Das dritte `mov 0,3` hinter start wird nicht gelegt, somit rennt der Dwarf, der mittlerweile nur noch Imp-Instruktionen ausführt als erster in den Bereich in dem nur dat-Instruktionen (So wird der Core vorbelegt) liegen und stirbt. Somit kann der Imp trotz seiner eigentlich nicht aufs Gewinnen ausgelegte Strategie einige Punkte machen. Dem Leser bleibt es als Übung überlassen die Fälle b und c selbst zu Simulieren.

5.3. Ein Dwarf hat vier Leben

Wie ich bereits erwähnte zieht die Ausführung einer dat-Instruktion die eliminierung des Prozesses mit sich. Ob diese dat-Instruktion durch den Gegner in den Code des Kämpfer eingepflanzt wird, der Dwarf versucht genau das, oder ob der Gegner durch inkorrekte aber legale Instruktionen auf eine dat-Instruktion geführt wird spielt dabei keine Rolle. Der Kämpfer der zuerst alle seine Prozesse verliert, hat verloren.

Der Programmierer des Dwarf hat wahrscheinlich nicht schlecht gestaunt als der Imp ihn besiegte. In der Tat gewinnen die meisten Imps dadurch das sie den Gegner in dat-Instruktionen laufen lassen. Folgende Punkte tragen zum Sieg des Imp über den Dwarf bei.

1. Der Imp wird zwar durch die dekrementierung der Coreposition `bomb-1` verwundet, aber läuft dann noch so weit das er dem Dwarf schaden, ihn sogar töten kann. Tatsächliche ist der Imp durch diese "Verwundung" tötlich. Ohne sie würde es immer zu einem unentschieden kommen.

2. Der Dwarf trifft den Imp zu selten mit seine Bomben. Es müßte ein größerer Teppich anstatt einer einzelnen Bombe geschmissen werden.

3. Der Dwarf hat wie der Imp nur einen Prozeß. Wären es mehrere dann würde der Imp in der entscheidende Phase einige der Dwarf-Prozesse überholen und der Dwarf somit noch einige Prozesse haben wenn der Imp in sein Loch fällt.

Zum ersten Punkt fällt die Lösung recht schnell ein. Übrigens, diese ständige dekrementierung eines B-Feldes heißt Imp-Gate. Der Programmierer legt das Imp-Gate einfach etwas weiter nach hinten, so das der verwundete Imp nicht mehr in den Dwarf ausläuft.

Einen komplizierteren Bombenteppich kann man nur mit einer komplizierteren Schleife erstellen.

```
        ORG  start

        dat     #0,     #-4
bomb    dat     #0,     #0
loop    sub.ab  #3,     bomb-1
start   mov.i   bomb,   <bomb-1
        mov.i   bomb,   @bomb-1
        mov.i   bomb,   @bomb-1
        mov.i   bomb,   @bomb-1
        jmp     loop

        END
```

Das sieht auf den ersten Blick nicht so schlecht aus. Die Bombardiergeschwindigkeit steigt auf 4 Bomben/6 Instruktionen. Das macht 66%c. Allerdings können wir das Imp-Gate nicht mehr 100% aktiv halten, es wird nur noch mit 33% betrieben. Das ist

eigentlich ganz gut, wenn man den schnelleren Bombenteppich als Ausgleich hat. Aber wir haben noch Punkt 3 in der Liste. Mehr Prozesse heißt, den Imp überleben. Es zählt nur wer am Ende noch lebt. Wenn wir uns die Forderung: 4 Bomben im Abstand von 4 mit mehreren Prozessen nochmal durch den Kopf gehen lassen gestalten müssen, dann fällt es fast schon von alleine auf den Monitor

```
        ORG     start

; ===== Launch =====
start   mov.ab #8,      -12         ; Impfalle installieren  ; Bomb
        spl    1                    ; Prozeß teilen = 2 Prozesse
        spl    1                    ; 2 Prozesse teilen = 4 Prozesse

; ===== Kämpfer ====
loop    mov.i  start-1, <start-12
        jmp    loop,    <start-12

        END
```

Da die Falle nur durch die Predekrementadressierung lebt, muss bei der Positionierung ein bisschen gerechnet werden, damit Falle nicht durch die Bomben des Dwarf zerstört, bzw. die Werte durcheinander gebracht werden. Der Dwarf könnte sich dadurch selbst überschreiben.

Ich habe mit den Kommentarzeilen mal den Launchbereich, das ist der Teil des Codes der die Vorbereitungen für den Kämpfer schafft und dem eigentlichen Kämpfer unterteilt. Das ist für die Berechnung der Bombardiergeschwindigkeit wichtig. Eine Schleife ist jetzt nur noch 2 Instruktionen lang wirft nach wie vor eine Bombe. Dadurch das es 4 parallele Prozesse sind werden die Bomben in der Anordnung 4 Stück hintereinander und dann 4 Lücken, geschmissen. Also, wenn der Bombenteppich zum Imp kommt können nur die Orginal-Imp sowie die Miniringe mov 0,2 und mov 0,3 eventl. auch mov 0,4 das überleben. Alle größeren werden verwundet und laufen aus. Da nur die größeren die Falle überwinden könnten, bzw. verwundet noch in den Dwarf reinlaufen könnten, ist die Sache geritzt. Der Dwarf kann gegen den Imp und Impringe nicht mehr verlieren. Ganz nebenbei bemerkt liegt die Geschwindigkeit des Dwarf mit der er bombardiert bei 50%c oder .5c. Einen kleinen Vorgriff erlaube ich mir aber jetzt. Wollen wir dem Dwarf mal den Sonntagsanzug, das heißt ein komplettes Recode-Listing verpassen.

```
;redcode-94
;name Dwarf
;version 2.0
;date 12.04.2001
;author Sascha Zapf
;assert CORESIZE==8000
;strategy .5c dat 0,0 Core bombing
;strategy  100%c Imp Gate
```

```
        ORG     launch

        gatepos EQU launch-12           ; Abstand Impfalle vom Kämpfer

launch  mov.ab #8,      gatepos         ; Impfalle installieren
        spl 1                           ; Anzahl Prozesse verdoppeln
        spl 1                           ; Anzahl Prozesse verdoppeln
loop    mov.i  launch-1, <gatepos       ; Bombe schmeißen + Falle
        jmp    loop,     <gatepos       ; Rücksprung + Falle

        END
```

Wie dem einen oder anderen vielleicht auffällt könnte man den Code noch um die mov-Instruktion bei launch kürzen. Wenn man die Impfalle so positioniert das der Kämpfer sie nicht Bombardieren kann, kann die Adressierung auch mit der dort befindlichen 0 beginnen. Auch das ist wieder ein Aufgabe für den Leser. Schneller wird der Kämpfer dadurch nicht, aber der Launchcode wird verkürzt.

Der Imp hat bei der Taktik des Dwarf keine Chance. Um das Gate zu passieren und den Dwarf zu verletzen müßte der Imp seine Spitze weiter nach vorne legen. Allerdings würde er bei einer größe von über 4 das Bombardement des Dwarf nicht überleben.

Dem aufmerksamen Leser, bzw. den Leuten die die beiden Kämpfer im Debbuger verfolgt haben ist bestimmt aufgefallen das aus dem Imp-Gate noch etwas anderes wird. Der Dwarf kann das Imp-Gate nur mit 100% betreiben wenn er seine Bomben indirekt über das B-Feld schmeißt. Was passiert also wenn der Imp seine Spitze mit dem Wert 3 im B-Feld auf das Gate legt? Nun, in 50% der Fälle wird das B-Feld bei der nächsten Instruktion des Dwarf einfach nur dekrementiert. Allerdings könnte der Dwarf auch gerade seine Bomben schmeißen. Da der Imp die 3 im B-Feld trägt, wird der Teppich des Dwarf auf jeden Fall neu gestartet. Der Imp hat es dadurch also noch zusätzlich mit dem Bombenteppich zu tun.

5.4. Ein Imp wird Erwachsen

Als erstes wird sich der Programmieren des Imp mit einer der beiden Hürden beschäftigen müssen. Dazu simuliert er den Teil des Dwarf's dem er sich als erstes widmen will. Dazu kann er mit einer Zeile ein

6. Verschiedene Techniken

Der Angriff auf den Imp war einfach. Der Imp kommt immer von hinten, also muß man ihm mit den Bomben entgegengehen. Der Imp braucht ein bestimmte Anzahl intakter Instruktionen um weiter zu leben, also Bombardiert man in regelmäßigen Abständen. Der Imp legt sich sein Instruktionen zurecht, also muß man das Ziel dieser Kopieraktion verändert. Mit den drei Punkten habe wir unseren Dwarf zu einem Gegner für den Imp gemacht gegen den er nicht gewinnen kann. Selbst ein unentschieden kommt mit dem Imp-Gate nicht in Frage. Gegen unbekannte Gegner kämpfen ist um einiges schwieriger. Der Angriff sowie die Verteidigung kann nicht "maßgeschneidert" werden.

Im laufe der Zeit habe sich einige Strategien entwickelt wie man einen Kampf führen kann. Die Reihenfolge entspricht nicht dem ersten Auftreten.

6.1. Angriffsstrategien

6.1.1. Imp

Dieser Kämpfer arbeitet nur mit dem mov-Befehl. Er überschreibt den Gegner und erreicht oft ein Unentschieden. Alle Varianten ausser mov 0,1 haben zusätzlich die Chance den gegnerischen Prozeß auf eine dat-Instruktion zu leiten, woran dieser dann stirbt.

6.1.2. Imp-Spiralen

Es sind praktisch 3 oder mehr Imps die sich Gegenseitig die Instruktionen zurechtlegen. Obwohl ein Imp-Gate einen Imp verwundet, so lebt er doch einige Zeit durch die Instruktionen weiter die er durch den anderen Imp zurechtgelegt bekommt. Erst wenn die beschädigung des Imps, der das Gate passiert sich auf ihn auswirkt, hat er ein Loch und stirbt. Imp-Spiralen mit mehreren Prozessen sind allerdings mächtige Gegner die man nicht unterschätzen sollte. Ausserdem belegen sie nicht viel Platz, es ist also ein kleines Add-On für einen Kämpfer den man Programmiert hat. Dadurch das Imp-Spiralen mit mehreren Prozessen arbeiten sind sie langsamer als andere Gegner. Die Chance das die Gegner durch die mov-Instruktionen des Imp's auf eine dat-Instruktion geführt werden ist dadurch größer und hilft zudem die Spirale weiterleben zu lassen. Zudem gibt es die Form der Gate crashing Imp-Spiral. Ein Kombination von drei Imp-Spiralen deren erste beiden Spiralen bei Kontakt mit dem Gate verwundet werden. Allerdings verändert die Verwundung die Instruktion so das sie zu Instruktionen der dritten Spirale werden, die somit weiterlebt und das Tor passiert hat. Gegen eine 3 Punkt und 6 Prozeß Gate crashing Imp-Spiral hätte unser guter Dwarf wohl keine

Chance. Allerdings muß man dabei sagen das der Umfang an Recode für den Launch (ca. 60 Befehle) oder der doch sehr zeitaufwendige Launch dem Dwarf einiges an Zeit verschafft das ganze noch im Keim zu ersticken.

6.1.3. Coreclear

Den Core vorwärts oder rückwärts mit dat-Instruktionen überschreiben. Meisten ein Teil eines Kämpfer in den er nach verwundungen durch andere oder durch sich selbst, verzweigt. Vorstellbar wäre das unser Dwarf mit spl 0-Instruktionen um sich schmeißt. In der zweiten Phase tötet er die betäubten Gegner dann mit dem Coreclear.

6.1.4. D-Clear

Eine Weiterentwicklung des Coreclear ist der D-Clear. Durch überschreiben von Quell.- und Zielzeiger durch die eigenen Bomben kann der D-Clear verschiedene Phasen durchmachen.

```
dc      spl    #-3,       #-10
        spl    #-1,       #-10
        mov.i  *pointer,<pointer
        dat.f  #-1,       #-10
pointer dat.f  #-4,       #-10
```

Der Core wird zweimal mit spl-Instruktionen überschrieben, danach mit dat-Instruktionen.

6.1.5. Stone - Bomber

Der Name Stone bezeichnet die Gruppe von Kämpfer die nach dem Muster wie unser Dwarf arbeiten. Dabei wird der Core in regelmäßigen Abständen bombardiert. Es gibt einige Dwarfs die sich absichtlich selbst bombardieren, um dadurch zu mutieren um in eine zweite Angriffsphase zu wechseln. Skew Dwarf aus Recycled Bits mutiert durch selbstbombardieren von normalen Bomber zum Core Clear und dann zu Imp-Gate. Hier mal ein Beispiel wie ein Bomber durch einen Selbsttreffer seine zweite Phase einleiten kann. Später werden wir diesem Kämpfer noch mit Hilfe des Debuggers cdb verbessern.

```
;redcode-94
;name Selfmod-Dwarf
;author Sascha Zapf
;assert CORESIZE==8000

        ORG start

start   mov.i  splbomb, @splbomb
        add.ab #3044,   splbomb
selfhit jmp    start
```

```
        mov.i  datbomb, <selfhit
datbomb dat    #0,      #55
splbomb spl    0,       1

        END
```

Dieser leicht veränderte Dwarf schmeißt mit spl-Bomben um sich. Die Bombe Nr. 1159 trifft die jmp-Instruktion und der Dwarf beginnt mit dem CoreClear.

6.1.6. B-Scanner

Da der Core zu Beginn des Kampf aus dat $0,$0-Instruktionen besteht, kann man versuchen den Gegner zu suchen. Aus der Zeit in der man nur B-Felder aktiv bearbeiten konnte, kommt die Klasse der B-Scanner. Der Core mit folgender Schleife durchsucht.

```
loop    add #10,  scan
scan    jmz loop, 10
```

Der jmz-Befehl verzeigt auf das Ziel in seinem A-Feld nur dann wenn das B-Feld der Anweisung auf die sein B-Feld zeigt 0 ist. Wenn man im Anschluß an diese Schleife ein paar Bomben an die Stelle schmeißt, hat man den gegnerischen Kämpfer vielleicht schon erwischt. Würden wir diesen B-Scanner gegen unseren Dwarf antreten lassen würde er warscheinlich zuerst das Imp-Gate und dann den Dwarf selbst bombardieren.

6.1.7. CMP-Scanner

Die B-Scanner überprüfen immer nur eine Speicherstelle, finden 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 vergleichen. 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.

```
        ...
        ...
loop    add.f  incr,   cline
cline   cmp.i  $123,   $134
        jmp    loop
attack  ...
        ...
incr    dat.f  #11,    #11
```

6.1.8. Quick-Scanner

Wie in Assembler üblich kann man Schleifen aufrollen um noch etwas mehr Geschwindigkeit zu bekommen. Während die Cmp-Scanner noch möglichst gründlich den gesamten Core absuchen, versuchen Quickscanner sehr schnell grob über den Core zu scannen. Das hat in Zeiten in denen die Kämpfer meist den gesamten verfügbaren Platz ausnutzen durchaus seine Berechtigung. Hier mal ein einfacher Quickscanner.

```
        sne.i  $1000,   $1095
        seq.i  $1190,   $1285
        mov.ab #1000,   attack
        sne.i  $1380,   $1475
        seq.i  $1570,   $1665
        mov.ab #1380,   attack
        sne.i  $1760,   $1855
        seq.i  $1950,   $2045
        mov.ab #1760,   attack
        jmz.b  attack,  attack
        ...
        sne.i  $2140,   $2235
        seq.i  $2330,   $2425
        mov.ab #2140,   attack
        sne.i  $2520,   $2615
        seq.i  $2710,   $2805
        mov.ab #2520,   attack
        sne.i  $2900,   $2995
        seq.i  $3090,   $3185
        mov.ab #2900,   attack
        jmz.b  attack,  attack
        ...
        ...
        ...
attack  mov.i  bomb,    $0
```

Die erste Instruktion „Skip if not equal“ vergleicht zwei Speicherstellen im Core miteinander. Sollte dabei ungleichheit festgestellt werden, wird die seq-Instruktion übersprungen. Sollte gleichheit herrschen, so wird als nächstes „Skip if equal“ ausgeführt. Wenn diese ebenfalls zwei gleiche Speicherstellen findet wird die mov-Instruktion übersprungen. Die mov-Instruktion, wird also ausgeführt wenn eine der beiden Instruktion eine nicht übereinstimmung gefunden hat. Dann schreibt sie einen Startwert für die Angriffsphase an eine bestimmte Stelle im Kämpfer. Die jmz-Instruktion die zwischendurch anzutreffen sind, reagieren eben auf diesen Wert. Die Geschwindigkeit beim scannen schlägt den Cmp-Scanner um längen. Zwischen den jmz-Instruktionen wird der Core mit 200%c gescannt, während ein einfacher Cmp-Scanner es gerade auf 66%c bringt. Allerdings ist wirklich nur das scannen schneller. Sollte der Quickscanner etwas finden, so dauert es noch bis zur nächsten jmz-Instruktion bis er darauf reagieren kann.

Dazu kommt das vier mögliche Postionen in Frage kommen. Vieles der Geschwindigkeit geht durch diese beiden Tatsachen verloren. Denoch ist dieser Code sehr nützlich auch in anderer Hinsicht. Da er nur einmal durchlaufen wird, ist er als vergrößerung des Kämpfer stets willkommen. Wenn die anderen Teile des Kämpfers per bootstrap vom Basis-Kämpfer wegkopiert werden, bleibt ein großer Dummy zurück, der von anderen Quickscanner sicherlich gefunden wird.

6.1.9. Q^2-Scanner

Die große schwäche der Quickscanner ist die Verzögerung mit der sie einen gefundenen Gegner angreifen. Um den Gegner sofort angreifen zu können, müßte nach jedem sne/seq/mov-Block eine jmz-Instruktion zum Angriffscode verzweigen. Da das die Geschwindigkeit des scannens deutlich senken. Ein andere Weg wäre wenn die jmp-Instruktion verschiedene Stellen im Angriffscode aufrufen würde. Dort könnte dann auf die Aufrufe reagiert werden. Um die ganze Sache dann noch zu verkleinern, könnte die jmp-Instruktion mit ihrem unbenutzten B-Feld Zeiger auf Datentabellen, welche die Scanpositionen beinhalten beeinflussen.

```
          seq.i  $1000,$1250
          jmp    attack1             ; Alles bleibt wie es ist.
          seq.i  $1500,$1750
          jmp    attack1, <pointer1  ; Tabellenzeiger wird verändert
          seq.i  $2000,$2250
          jmp    attack1, >pointer1  ; Ebenfalls veränderung.
          ...
          ...
          ...
          dat.f  $1500,$1750
table     dat.f  $1000,$1250
          dat.f  $2000,$2250
          ...
pointer1  dat.f  $0,   table
target    dat.f  $0,   $0
attack1   mov.f  @pointer1, target
          mov.i  table, *target
          mov.i  table, @target
          ...
          ...
```

Die jmp-Instruktionen beeinflussen den Zeiger in die Datentabelle, der vor dem Bombardement nach **target** kopiert wird. Eine andere Möglichkeit wäre die Positionen der seq-Instruktionen selber für das Bombardement zu nutzen. Die Geschwindigkeit mit der Angegriffen wird hat sich damit auf max. 3 Instruktionen verkleinert. Das kann Kampfentscheident sein. Natürlich wird dieser Geschwindgkeitsvorteil mit Platz erkauft, weswegen Q^2 Scanner noch gröber über den Core scannen.

6.1.10. Q^3 Scanner

Beim Q^3 Scanner wurde versucht ein wenig des Platzverbrauchs des Q^2 Scanner's zurückzugewinnen. Anstatt die Zeiger aus einer Tabelle zu nehmen werden sie unter zuhilfenahme der B-Felder der jmp-Instruktionen, berechnet.

```
          seq.i  $1000,$1250
          jmp    attack1, {multip
          seq.i  $1500,$1750
          jmp    attack1
          seq.i  $2000,$2250
          jmp    attack1, }multip
          ...
          ...
          ...
attack1   mul.ab #1, pointer
          add.b  pointer, bomb
          mov.i  pointer, $1000
          ...
          ...
          ...
pointer   dat.f  $0,   $500
```

6.1.11. Q^4 Scanner

Ohne weiteren Beispielcode zu präsentieren, nur ganz kurz erklärt wurde bei Q^4 Scannern die Scanndichte etwas erhöht. Dazu werden sne/seq/jmp-Blöcke genutzt. Zusätzlich gibt es kleine Tabellen die bei der Berechnung der Zielposition helfen.

6.1.12. Paper - Replicatoren

Paper bezeichnet Kämpfer welche sich reproduzieren. Sie überschwemmen den Core mit Kopien von sich, die ebenfalls Kopien von sich erzeugen. Somit kämpft der Gegner nicht gegen einen sondern gegen viele. Tötet man eine Kopie, so werden die anderen nur schneller. Paper gewinnen meist dadurch das sie ihren Gegner teilweise überschreiben und auf dat-Instruktionen führen.

```
pointer  dat.f $,$1233
temp     dat.f $,$1233
start    mov.i temp, pointer
loop     mov.i }pointer,>pointer
         djn.b loop, #len
         spl   pointer+1233
         ...
         ...
         ...
```

Die Kopierschleife mit dem daranhängenden Kämpfer wird kopiert und danach ausgeführt. Wenn die Kopien sich nicht gegenseitig beschädigen, dann geht das ganze solange weiter bis die Prozesse ausgehen.

6.1.13. Silk

Wie Paper reproduziert sich Silk, springt aber die Kopie an bevor sie im Speicher ist. Das geht durch die Tatsache das der durch die spl-Instruktion erzeugte Prozeß in der Reihenfolge hinter dem erzeugenden Prozeß liegt.

```
; ----- Launch -----

       spl 1         ;\
       spl 1         ; \___ 4 parallele Prozeße erzeugen

; ----- Kämpfer ----

silk   spl 2534,0        ; vier neue Prozesse zur neuen Kopie abzweigen.
       mov >silk,}silk   ; vier Anweisungen kopieren
       mov bomb,>silk    ; dat-Bombe in die nächsten vier Felder
                         ; werfen.
bomb   dat #0,#0         ; Bombe und Selbstmord des Silks
```

Dieser Kämpfer reproduziert sich selbst 2534 Felder weiter, und setzt dann noch 4 dat-Bomben unmittelbar vor sich. Da dieser Kämpfer sich selbst nach dem werfen der dat-Bomben umbringt, führt er auch Gegner, die er eventl. überschrieben hat in den dat-Tot.

6.1.14. Vampire

Eine Strategie die mich immer besonders angesprochen hat, ist der Vampir. Der Vampir ist ein Kämpfer der mit jmp-Befehlen um sich schmeißt. Die jmp-Befehle werden vor dem Wurf so berechnet das sie auf eine bestimmte Stelle im Vampir zeigt. Dort steht meist als erster Befehl ein spl 0, damit der Gegner der über diesen Befehl dorthin geführt wird, anfängt immer weiter Prozesse zu erzeugen. Sollte der Gegner nur über einen Prozeß verfügen, so ist er damit gefangen. Gegner die mit mehreren Prozesse arbeiten werden so stark verlangsamt, das sie wohl kaum noch eine Gefahr darstellen. Wenn unser 4 Prozess-Dwarf in die Falle geht, dann ist er ebenfalls gefangen, da alle Prozesse parallel arbeiten. Im Anschluß an diese Vermehrung liegt dann meist weiterer Angriffscode, meist ein Coreclear, der dann ausgeführt wird ohne das der Vampir dadurch Arbeit hätte. Wichtig ist am Ende nur das der Vampir alle Gefangenen Prozesse zerstört, das kann mit einem Mini-Coreclear, der nur in dem Bereich der Falle arbeitet geschehen,

```
fang     jmp    8+trap, -8
loop     mov.i  fang, @fang
```

```
         add.f  incr, fang
         djn.b  loop, #999
         ...
         ...
         ...
incr     dat.f  #8,#-8
trap     spl    0
         mov.i  trap,<1
         dat.f  #0, #-8
```

Diese Technik ist in Multiwarriorfights echter Vorteil. Teile dieser Technik kann natürlich auch von anderen Kämpfern benutzt werden. Scanner schmeißen oft spl 0-Bomben um den Gegner zu betäuben oder zu verlangsamen, bevor sie in ihre Angriffphase wechseln.

6.1.15. p-Switcher

Diese Kämpfer bestehen eigentlich aus mehreren kleinen anderen. Ein p-Switcher überprüft zu Beginn eines Kampfes wie er im letzten Kampf angeschnitten hat. Wenn der Kampf erfolgreich war, dann behält er seine Technik bei. Andernfalls wählt er ein neue Technik aus. Dabei können sich die Techniken stark unterscheiden. Es kann sich aber auch nur um einzelne Werte handeln. Es gibt Kämpfer die nur Anhand der Tatsache ob sie im letzten Kampf gewonnen oder verloren haben entscheiden. Es gibt aber auch Kämpfer die einen kleinen Verlauf im pSpace ablegen, damit sie noch besser auswerten können wie sie den aktuellen Gegner besiegen können.

Viele kleine Details habe ich ausgelassen, meist handelt sich um einzelne Befehle die einen Angriff ausmachen. Aber das wird man sehr schnell selbst herausfinden. Wie auch immer, ist es sehr schwer gegen einen unbekannten zu Kämpfen. Man sollte sich deshalb nicht nur auf eine Technik verlassen. Hybriden unter den Kämpfer verwendet deshalb mehrere Techniken um des Duell zu gewinnen. Sehr erfolgreich waren Imp-Spiralen die mit einem Stone gekoppelt waren. Quick-Scanner mit einer anschließenden D-Clear-Phase sind auch schwer zu besiegen.

6.2. Verteidigung

Nun kommen wir zu den Verteidungsmechanismen. 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.

6.2.1. 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 Dekrement-operation 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.

6.2.2. Decoy

Je nach den Regeln des Hill’s auf dem der Kämpfer antritt darf er ein 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 ein 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.

6.2.3. Bomber-Dodger

Warscheinlich hat sein hier der Decoy weiterentwickelt. Die Idee ist den Bombenteppich des Dwarf zu scannen und kleine Kämpfer die 1 oder 2 Instruktionen große Decoys’ enthalten so zu plazieren das die Bomben des Dwarf’s in die Lücken fallen. **(Ist diese Technik noch neu, oder schon ausgereift?)** In der ersten Phase muss der Kämpfer möglichst schnell die Bomben erwischen. Danach den oder die Programmteile dementsprechend verschieben und starten.

6.2.4. Colour

Eine Verteidungstechnik 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.

6.2.5. 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 können 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.

6.2.6. Mirror oder Reflection.

Eine Technik gegen sogenannte On-axis Scanner. Das sind cmp-Scanner die einen Abstand von einer halben Corelänge 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.

6.2.7. 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 warscheinlich 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 veteilt 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, starkt 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 warscheinlich selbst bombardieren.

6.2.8. 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 den Versuche 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.

6.2.9. 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 jmz-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 jmz-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 meisten ein Coreclear gestartet. Sinnvoll wäre es aber auch ein Paper zu starten, da man davon ausgehen kann das 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.

6.2.10. 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.

6.3. Papier, Stein und Schere

Das alte Spiel was wir sicher alle kennen, hat auch Eingang in das Spiel gefunden. Hier werden verschiedene Kämpfertypen den einzelnen Symbolen zugeordnet.

Paper Alle Arten von Replikatoren bis hin zum Silk.

Stone Bomber

Scissors Die Gilde der Scanner und Vampires

Sicherlich gelten nicht immer die Regeln des alten Spiel, da auch mal ein Stone ein Paper besiegt. Es gibt auch einige Typen die sich nicht in dieses Schema fügen.

7. pSpace

pSpace ist ein Speicher der einem Kämpfer, wenn er will, alleine gehört. Es gibt zwei Kommandos mit denen er darauf zugreifen kann. Die Speicherstellen im pSpace unterscheiden sich vom Core dadurch das nur ein Wert und keine ganze Instruktion mit ihren Feldern A und B darin Platz haben. Das besondere an diesem Speicher ist aber das er bei Mehr-Runden-Kämpfen seinen Inhalt behält. Ein Kämpfer kann auf die Erfahrungen die er im letzten Kampf gemacht hat zurückgreifen. Ich beschreibe den pSpace dadurch wie ein Kämpfer mit dem Verteidigungmechanismus 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.

1. Abstand zwischen den Bomben
2. 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 "ausmisst". Das ist aber auch nicht immer so einfach. Wenn der Dwarf keine "coloured" Bomben wirft dann fällt die 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 versuche die Position des Gegner's 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 abscannen und auf den Teppich warten. Sobald die erste Bombe registriert wird, kann der Kämpfer anhand der gespeicherte Daten die Position des Dwarf ermitteln und ihn praktisch mit ein paar gezielte 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, in der Kämpferspezifische Daten stehen. 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 das noch jemand anderes darin arbeitet.

8. Die Prozeßwarteschlange

Kein Computer mit einem Prozessor kann wirklich gleichzeit 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 gepeichert. 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 dann aus und gibt den Befehl in den meisten Fällen zurück. Während die MARS den Befehl ausführt können verschieden Dinge passieren.

1. Der Befehl wird abgearbeitet. Die MARS verändert die Adresse des Prozesses in der Form das er nun auf die nächste Instruktion zeigt.

2. Der Befehl ist ein Sprungbefehl. Die MARS verändert die Adresse desd Prozesses wie es in dem Sprungbefehl geünscht 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 Prozeß zu erzeugen der seine nächste auzuführende Adresse durch das A-Feld der spl-Instruktion bekommt.

Die Tatsache das der neue Prozeß in der Warteschlange hinter dem erzeugenden steht ist sehr wichtig.

Nun arbeitet die MARS Kämpfer für Kämpfer und die Kämpfer Prozeß für Prozeß ab. Folgende Situation:

Kämpferwarteschlange der MARS : Kämpfer A, Kämpfer B, Kämpfer C

Anzahl Prozesse der Kämpfer: A = 2, B = 1 und 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 das Kämpfer alleine aus der Tatsache das sie mehrere Prozesse haben, keine 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 das alle seine Prozesse sterben werden, so dauert das noch eine gewisse Zeit.

9. Programmieren mit Redcode

Kommen wir nun zum Handwerkszeug des Spiels. 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 Befehlen und ihren Modifizierern ignoriert. Wie man schreibt ist einem selbst überlassen. Im letzten Teil dieses Kapitel werde ich ein gewissen Standart ansprechen der sich im laufe der Zeit verbreitet hat. Einen Standart zu unterstützen trägt dazu bei das sich Programmierer besser verstehen, und man selber leichter durch seine eigenen Kämpfer findet. Viele Programmierer kennen das Thema des Standarts sicher. Bei jedem neuen Job, oder gar bei jedem neuen Projekt hat man sich an einen neuen Standart zu gewöhnen. 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

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 verflogen sein.

Es ist wichtig die Formulierungen genau durchzulesen. Manchmal bemerkt man sofort wofür man einen Befehl einsetzen kann. Gerade wenn bei der Beschreibung der Instruktionen, wenn praktisch alles zusammen kommt ist es ganz gut wenn man Modifizierer und Adressierungsarten schon intus hat. Auf sie werde ich nämlich nur eingehen wenn sich Besonderheiten zeigen.

9.1. Die Operatoren

Folgende Operatoren können benutzt werden.
Arithmetische Operatoren

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

Vergleichsoperatoren

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

Logische Operatoren

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

Zuweisung

```
=
```

9.2. Die PseudoOpcodes

PseudoOcodes sind Anweisungen die den Kämpfer beeinflussen aber nicht im fertigen Code stehen. Sie werden während der Assemblierung in Adresse oder Quelltext umgewandelt.

9.2.1. ORG

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

9.2.2. END

END [**Adresse o. Label**] Hiermit wird das Ende des Kämpfer signalisiert. Optional kann hier ebenfalls die erste auszuführende Anweisung bezeichnet werden. Sollte hier ebenfalls keine Adresse oder kein Label angegeben werden, so wird die erste Anweisung im Quelltext angesprungen. Achtung, sollte bei ORG und END jeweils eine Adresse stehen, so wird die von ORG genommen.

9.2.3. EQU

text EQU ersatztext Textuelle Ersetzung des Labels durch den Text.

```
Beispiel:       Gate EQU Start-12
```

Würde jedes vorkommen des Wortes “Gate” im Quelltext durch den Text “Start-12” ersetzen. Achtung, textuelle Ersetzung bringt einige Gefahren mit sich. Will man den Abstand des Imp-Gates zum Kämpfer auf diese Art verdreifachen:

```
        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*-14 Positionen sondern 3*2-12= -6 Postionen 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:

```
                Gate EQU (Start-12)
```

Es können mehrere Zeilen benannt werden. Folgende Zeilen erzeugen einen 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
```

9.2.4. 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. Desweiteren ist es möglich eine Zählervariable anzugeben, welche dann mit dem & Zeichen mit dem zu erzeugenden Quelltext verbunden werden kann. Dass kann auch benutzt werden und ähnlich generiert Sequenzen mit Label anzuspringen.

Gegeben wird diese Sequenz.

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

```
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 gehangen wird kann man auch Werte “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 können man auch folgenden Block nutzen.

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

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

9.2.5. PIN

PIN Nummer
Dieser PseudoOpcode benennt den pSpace eines Kämpfer mit einer Nummer. Käpfer mit der gleichen PIN (PSpace Idenditification Number) teilen sich einen pSpace. Dieser PseudoOpcode ist ebenfalls nicht im 94er Standart enthalten. Wurde aber mit der pMars 0.8 als Quasistandart eingeführt und kann deshalb auch auf den 94er Hill's, mit Ausnahme der No-pSpace-Hill's benutzt werden.

9.3. Die vordefinierten Konstanten.

Die folgende Konstanten sind im Redcode nutzbar. Sie sind beim Assemblieren mit den tatsächlichen Werten initialisiert.

CORESIZE Größe des Core's. In einem Kampf gibt es keine größere Zahl als **CORESIZE-1**. Daran sollte man denken wenn man Werte definiert die in Kämpfern liegen die in verschiedene Größen des Core's agieren können.

MAXPROCESSES Maximale Anzahl der Prozesse eines Kämpfers.

MAXCYCLES Anzahl der Instruktionen die jeder Kämpfer ausführen darf.

MAXLENGTH Maximale Länge eines assemblierten Kämpfers.

MINDISTANCE 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 Kämpfer's im Core gemessen. Wenn

	man gegen Imp's kämpft dann kann man zu Beginn des Kämpfers eine Menge Freiraum lassen der nur mit dat-Instruktionen gefüllt ist. Das und **MINDISTANCE** geben ein bißchen Zeit.
ROUNDS	Anzahl der Runden bei Mehr-Runden-Kämpfen
PSPACESIZE	Größe des pSpace.
CURLINE	Number der Zeile im vorassemblierten Kämpfer.
VERSION	Version der pMars*100: 60 ist Version 0.6.0
WARRIORS	Anzahl der Kämpfer.

Die bekannteste Anwendung der Konstante **CORESIZE** ist wohl die in der Assert-Zeile jedes Kämpfers. Neben der Assert-Zeile können die Konstanten in Instruktionen eingesetzt werden. Beispiel der cmp-Anweisung eines On-axis cmp-Scanner's.

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

Es ist für unseren 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
```

hinter diesen Bereich in dem sich kein Gegner befindet setzen.

9.4. 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. Anders verhält es sich aber mit den besonderen Kommentaren im Redcode-File. Ein Kommentar wird mit den Semikolon eingeleitet. Alles was dahinter steht wird ignoriert. Das Ende der Zeile ist auch das Ende des Kommentars.

```
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 ein bisschen mehrarbeit kann man sich hier einiges an Denkarbeit sparen wenn man Teile des doch so erfolgreichen Kämpfer später in einem neuen Kämpfer "wiederverwerten" möchte. Wenn man bei der Entwicklung gewisse Handgriffe gut Kommentiert, dann können auch andere daraus lernen, was dafür sorgt das im wieder gute Kämpfer auf die Hill's geschickt werden.

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

9.4.1. ;redcode

Hiermit beginnt der Quelltext eines Kämpfer. Für den Falle das er 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.

9.4.2. ;name

Der Name eines Kämpfers ist nicht zu unterschätzen. Ein guter Name flößt dem gegnerischen Programmierer Respekt ein. Das wird sich bemerkbar machen wenn er Verteidigungsstrategien gegen diesen Kämpfer entwickeln muß ;-)

9.4.3. ;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 ältere Version zum Ende dieser Runde rausfliegen zu lassen. Sobald dieser Kämpfer erreicht wird, dann bekommt der zu löschende Kämpfer kein Punkte mehr gutgeschrieben. Dadurch fliegt er automatisch vom Hill.

9.4.4. ;author

Hier kann sich der Programmierer verewigen.

9.4.5. ;date

Das Schlüsseldatum der Erstellung des Kämpfers. Man kann hier auch den Text `@date@` eingeben, dann wird hier vom Server des Hill's auf den man den Kämpfer schickt das Datum an dem der Kämpfer auf den Hill's geschickt wurde eingetragen.

9.4.6. ;version

Versionsnummer des Kämpfers, nützlich bei Überarbeitungen.

9.4.7. ;assert

Dieser Ausdruck prüft ob ein Kämpfer in einer Umgebung kämpfen kann. Logisch Wahr heißt das der Kämpfer 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
```

Unser Dwarf, der sich bei Größen des Core's die sich nicht durch 4 teilen lassen selber bombardieren würde, könnte das mit -

```
;assert CORESIZE\%4==0
```

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

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

9.4.8. ;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 Kategorie angegeben. Je sicherer man sich bei seinem Kämpfer ist, umso mehr kann man über in verraten.
Beispiel:

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

9.5. 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.

9.5.1. Immediate - Unmittelbar

Der Wert um den es geht ist direkt im Feld abgelegt. Die Adresse die zum bearbeiten ist wird auf 0 gesetzt. Dadurch wird der angegebene Wert benutzt.

Die unmittelbare Adressierung wird durch # gekennzeichnet.

9.5.2. Direct - Direkt

Der Wert um den es geht liegt an der im Feld angegebenen Adresse. Die Direkte Adressierung ist voreingestellt. Sollte bei der Angabe eines Feldes keine Adressierung angegeben werden, dann wird die Direkte benutzt.

Die Direkte Adressierung wird durch $ gekennzeichnet.

9.5.3. 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. (Achtung, diese Addressierung ist nicht im 94er Standart festgelegt. Sie ist mit pMars 0.8 experimentell eingeführt worden. Alle 94er Hill's im Internet verarbeiten sie aber korrekt. Somit es ein Quasi-Standart der ruhigen Gewissens verwendet werden kann.)

Indirekte Adressierung über das B-Feld wird durch gekennzeichnet.

9.5.4. Predecrement Indirect - Predekremental Indirekt

Wie Indirekt, nur das der Index **vor** der Ermittlung der tatsächlichen Adresse um eins reduziert wird.

Prekremental Indirekt über das A-Feld wird mit { gekennzeichnet. (Achtung, diese Addressierung ist nicht im 94er Standart festgelegt. Sie ist mit pMars 0.8 experimentell eingeführt worden. Alle 94er Hill's im Internet verarbeiten sie aber korrekt. Somit es ein Quasi-Standart der ruhigen Gewissens verwendet werden kann.)

Prekremental Indirekt über das B-Feld wird mit ¡ gekennzeichnet.

9.5.5. 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. (Achtung, diese Addressierung ist nicht im 94er Standart festgelegt. Sie ist mit pMars 0.8 experimentell eingeführt worden. Alle 94er Hill's im Internet verarbeiten sie aber korrekt. Somit es ein Quasi-Standart der ruhigen Gewissens verwendet werden kann.)

Postinkremental Indirekt über das B-Feld wird mit ¿ gekennzeichnet.

9.6. Die Modifizierer

Hiermit wird angeben mit welchen Teilen der Instruktionen gearbeitet wird. Wichtig ist hierbei das 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

9.6.1. A

Bearbeitet den A-Wert der A-Instruktion und den A-Wert der B-Instruktion.

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.

9.6.2. B

Bearbeitet den B-Wert der A-Instruktion und den B-Wert der B-Instruktion.

Beispiele:

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

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

9.6.3. AB

Bearbeitet den A-Wert der A-Instruktion und den B-Wert der B-Instruktion.

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.

9.6.4. BA

Bearbeitet den B-Wert der A-Instruktion und den A-Wert der B-Instruktion.

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.

9.6.5. F

Bearbeitet A-Wert und B-Wert der A-Instruktion und A-Wert und B-Wert der B-Instruktion.

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.

9.6.6. X

Bearbeitet A-Wert und B-Wert der Instruktion und B-Wert und A-Wert der B-Instruktion. Praktisch wie der Modifizierer F, nur über Kreuz.

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.

9.6.7. I

Bearbeitet die komplette A-Instruktion und die komplette B-Instruktion.

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.

9.7. Die Opcodes

Kommen wir nun zum Eingemachten. Ich hoffe das bei Fragen im vorherigen Text ab und an mal ein Blick in diesen Teil geworfen wurde. Ich halte nicht davon die Befehle direkt zu Anfang zu beschreiben, da man noch nichts damit anfangen kann. Ich bin mir aber sicher das jetzt, nach dem man sich schon Gedanken macht welchen Kämpfer oder was für Techniken man Programmiert, jeder Befehl in der Liste sehr viel genauer studiert und besser behalten wird. Die Predekrementalen und Postinkrementalen Adressierung erwähne ich nur dann wenn sie nicht erwartet werden. Wie bei den Befehlen dat oder nop, sowie dem B-Feld der Befehle jmp oder spl. Ich habe versucht möglichst lebendige Beispiele zu finden. Manchmal sind es kleine einzeilige Kämpfer.

9.7.1. DAT

Hiermit können Daten im Code eingefügt werden. Der Core wird mit dat #0, #0-Instruktionen vorinitialisert. Ein Prozeß der versucht diesen Befehl auszuführen wird eliminiert. Er wird nicht mehr in die Prozeßwarteschlange eingefügt. Sollten Predekrementale oder Postinkrementale Adressierungen vorhanden sein, so werden diese dennoch abgearbeitet.

Beispiele:

dat 0, 0 Mit dieser Instrutkion 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 bei Ausführung A-Wert und B-Wert der Instruktion vor sich. 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 Prozeß stirbt, und dekrementiert zusätzlich zwei Stellen im Core. Sollte der ausführende Prozeß zu einer Imp-Spirale gehören, so wird diese durch den Verlust eines Prozesses und dadurch das zwei andere Stellen praktisch durch ein Imp-Gate gehen, stark beschädigt.

9.7.2. MOV

Überschreibt gemäß dem Modifizierer die Daten der B-Instruktion mit den Daten der A-Instruktion. Nach Abschluß wird der Prozeß um eine Instruktion weiter gestellt und in die Prozeßwarteschlange eingefügt.

Beispiele:

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 Den kennen wir ja schon!

mov.i }src,>dest Könnte ein Teil der Reproduktionsschleife eines Paper's sein. Die Zeiger auf die Quell- und Zielinstruktion werden nach jedem kopieren inkrementiert.

9.7.2.1. Unstimmigkeiten zwischen Modifizierer und Adressierung.

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

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 dem was sie tun 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.

9.7.3. ADD

Überschreibt gemäß dem Modifizierer 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:

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 Aufgrund 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.

9.7.4. SUB

Überschreibt gemäß dem Modifizierer 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:

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.

9.7.5. MUL

Überschreibt gemäß dem Modifizierer 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:

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.

9.7.6. DIV

Überschreibt gemäß dem Modifizierer die Daten der B-Instruktion mit dem ganzzahligen Quotient der 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 Prozeßwarteschlage eingeschleust. Ansonsten wird der Prozeß um eine Instruktion weiter gestellt und in die Prozeßwarteschlange eingefügt. Der Modifizierer I wird als F behandelt.

Beispiele:

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.

9.7.7. MOD

Überschreibt gemäß dem Modifizierer 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 Prozeßwarteschlage eingeschleust. Ansonsten wird der Prozeß um eine Instruktion weiter gestellt und in die Prozeßwarteschlange eingefügt. Der Modifizierer I wird als F behandelt.

Beispiele:

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.

9.7.8. JMP

Legt den Prozeß mit neuer Adresse wieder in die Prozeßwarteschlange. Predekrementale oder Postinkrementale im B-Feld werden ebenfalls ausgeführt.

Beispiele:

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.

9.7.9. JMZ

Prüft gemäß dem Modifizierer den Wert der B-Instruktion und handelt wie jmp wenn der Wert 0 ist. Ansonsten wird der Prozeß ein Instruktion weiter gestellt und wieder in die Prozeßwarteschlange eingefügt. Der Modifizierer I wird als F behandelt.

Beispiele:

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 wir 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.

9.7.10. JMN

Arbeitet wie JMZ nur das es bei nicht-null Wert der B-Instruktion nach A-Instruktion verzweigt.

9.7.11. DJN

Dekrementiert gemäß dem Modifizierer den Wert der B-Instruktion. Dann wird dieser auf 0 geprüft. Wenn nicht-null dann wird der Prozeß mit Ziel A-Instruktion wieder in die Prozeßwarteschlange eingschleust. Bei 0 wird der Prozeß eins weiter gestellt und in die Prozeßwarteschlange eingefügt.

Beispiele:

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.

9.7.12. CMP

Vergleicht gemäß dem Modifizierer 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 ein Instruktion übersprungen. Sollten sie nicht gleich sein so wird der Prozeß um eine Stelle weiter gestellt und in die Prozeßwarteschlange eingeschleust.

Beispiele:

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 wieter mit 20.

9.7.13. SLT

Vergleicht gemäß dem Modifizierer den 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.

Beispiele:

```
        slt.a werta,wertb
posa    nop #0
posb    nop #0
werta   dat #-2,#0
wertb   dat #5, #0
```

Der slt-Befehl wird bei einer CORESIZE von 8000 die -2 intern in 7998 umrechnen. Da das nicht kleiner als 5 ist wird der nächste Befehl nicht übersprungen und die Ausführung geht bei posa weiter. Anders wäre das wenn werta und wertb vertauscht wären. Dann würde die Ausführung bei posb weiter gehen.

9.7.14. 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 Addressierung im B-Feld wird abgearbeitet.

Beispiele:

spl 1 Da Ausführung des erzeugende Prozesses und die des neuen Prozesses bei der nächsten Instruktion weiter geht, verdoppelt diese Instruktion die Anzahl der passierende Prozesse.

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

9.7.15. SEQ

Synonym für CMP.

9.7.16. SNE

Vergleicht gemäß dem Modifizierer A-Instruktion und B-Instruktion. Sollten diese beiden nicht gleich sein, so wird der Prozeß 2 Stellen weiter gestellt und in die Prozeßwarteschlange eingeschleust. Es wird also ein Instruktion übersprungen. Sollten sie gleich sein so wird der Prozeß um eine Stelle weiter gestellt und in die Prozeßwarteschlange eingeschleust.

Beispiele:

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 Instrukion bei posa mit dem B-Wert der Instruktion bei posb. Beide Zeiger werden nach dem Vergleich um eins inkrementiert.

9.7.17. 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-Instrution zwei Imp-Gate's betreiben.

Beispiele:

nop <2667, <5334 Eben beschriebenes Imp-Gate gegen Spiralen.

nop <, }0 Diese Anweisung in einen Gegner eingepflanzt überschreibt eine seine Instruktionen und läßt ihn mit jedem Aufruf einen B-Wert einer Instruktion dekrementieren. Nach der dekrementierung wird der Zeiger inkrementiert. Somit

ist jeden durchlaufen ein andere Instruktion dran. Da ihn Verbindung mit einem Stealth-Kämpfer ist schon ein vielversprechende Taktik den Gegner zu beschädigen.

9.7.18. LDP

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.

Beispiele:

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.

9.7.19. STP

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

10. pMARS

Die pMARS (portable Memory Array Redcode Simulator) ist der Quasistandart von CoreWar. Sie unterstützt den 88er und 94er Standart. Es gibt Erweiterungen die nicht im 94er Standart enthalten sind. Die pMARS gibt es mittlerweile für viele Plattformen. Ich empfehle die Benutzung dieser MARS. Die meisten Hill's arbeiten mit ihr. Man kann also die Erweiterungen wie zum Beispiel FOR/ROF-Blöcke getrost im Kämpfer lassen.

Die pMARS wird aktiv entwickelt. Die aktuelle Version ist bei Sourceforge zu finden.

Die pMARS ist durch die Kommandozeile zu bedienen. Der Kampf selber kann auf dem Bildschirm dargestellt werden. Die pMARS kann bis zum 36 Kämpfern gleichzeitig in Multiwarrior-Kampf gegeneinander antreten lassen.

10.1. Kommandozeilenparameter von pMARS.

-r # Die Anzahl der Runden des Kampfes wird gesetzt. Die Voreinstellung ist 1. Ein Maximum von 32787 Runden kann hier eingeben werden.

-s # Größe des Core's. Die Voreinstellung ist 8000. Die maximale Coregröße ist Plattform-abhängig, normalerweise mindestens 65535.

-c # Legt die Anzahl der Instruktionen fest die ein Warrior ausführen kann. Der Kampf endet wenn einer der Kontrahenten stirbt oder die Kämpfer die maximale Anzahl an Instruktionen ausgeführt haben.

-l # Maximale Länge des Kämpfers nach der Assemblierung in Zeilen. Voreinstellung ist 100 und kann bis auf 500 erhöht werden.

-d # Minimal Distanz der Kämpfer zueinander zum Beginn einer Runde. Obwohl die Kämpfer normalerweise zufällig im Core positioniert werden, wird diese Distanz gewährleistet. Der Wert kann nicht kleiner sein als die maximale Größe der Kämpfer. Die Voreinstellung ist hier 100.

-S # Größe des pSpace. Voreinstellung ist hier 1/16 der Größe des Core's.

-F # Weist die pMARS an den zweiten Kämpfer an eine bestimmte Adresse zu setzen. Kämpfer 1 wird immer an Adresse 0 gesetzt.

-f Der Zufallsgenerator der für die Positionierung des zweiten Kämpfers genutzt wird, wird zu Beginn des Kampfes neu initialsiert. Das geschieht mit einer Checksumme die aus dem Quellcode der beiden Kämpfer gebildet wird. Das heißt das zwar die Positionen in jeder Runde immer noch zufällig sind, aber

beim nächsten Kampf sind die Startpostionen der Kämpfer je Runde immer gleich. Wenn man Kämpfe nachspielen will, dann muß man diese Option nutzen.

-e Nach der Assemblierung der Kämpfer in den Core wird der cdb Debugger gestartet. Mit ihm kann man den Kampf Schrittweise verfolgen.

-b Die pMARS gibt nur nach das Ergebnis des Kampfes aus. Diese Option ist in Verbindung mit Schedulern und anderen Steuerprogramme nötig.

-k Das Ausgabeformat des KotH-Programms wird benutzt. Damit kann die PMARS auch mit den Skripten der KotH-Server betrieben werden.

-8 Erzeugt den 88er Standart. Alle Erweiterungen des 94er Standart, sowie die der pMARS werden abgeschaltet und erzeugen Syntaxfehler bei der Assemblierung. Dies ist nur nötig wenn es ein 88er Kampf sein soll. Kämpfer die im 88er Standart geschrieben sind laufen immer im 94er weil er eine Übermenge des 88er ist.

-o Die Ausgabe des Ergebnis der Kämpfer wird in absteigender Sortierung der Punkte ausgegeben anstatt der Reihenfolge der Kämpfer in der Kommandozeile. Das ist bei Multiwarrior-Kämpfen nützlich, weil man sonst den erstplazierten erst suchen müßte.

-V Der Assembler gibt eine Menge Daten aus. Das ist für das Debuggen eines Kämpfers nützlich.

-v Setzt den Display Modus. Auflösung sowie Informationsdichte auf dem grafischen Coredisplay können hier eingestellt werden. Dazu sehen sie bitte im Anhang für ihre Plattform nach.

- <fn> Das mit fn benannte File wird ausgewertet als wenn es in der Kommandozeile stehen würde. Das wird hauptsächlich für Multiwarrior-Kämpfe genutzt. Kann aber auch für einen bestimmten Satz ein Einstellungen gebraucht werden.

-Q # Ebenfalls nützlich wenn die pMARS im Batch-Mode, also in Skripts eingesetzt wird. Hiermit können verschiedene Exitcodes erzwungen werden. Zum Beispiel “pmars -q 1000” gibt die Version der pmars als Rückgabewert wieder. Für eine komplette Aufstellung bitte in Anleitung zur pMARS nachschlagen.

10.2. cdb Der Debugger

Der Debugger bietet die Möglichkeit das Verhalten eines Kämpfer Schritt für Schritt zu beobachten. Den Core bei Bedarf zu verändern, auch direkt Befehle in ihn zu programmieren. Natürlich kann man auch zwei Kämpfer gegeneinander antreten lassen und sich den Kampf in Zeitlupe sehen. Durch Macro’s läßt sich der Debugger programmieren. Damit lassen sich Vorgänge innerhalb eines Kampfes in regelmäßigen

Abständen anzeigen oder Werte im Kämpfer verändern bis sie den gewünschten Effekt erzeugen.

10.2.1. Die Kommandos

Ein Druck auf Return/Enter wiederholt immer das letzte Kommando. Wird das Kommando mit einem führenden Space geschrieben, so bleibt das letzte Kommando erhalten. Alle Kommandos können auch abgekürzt werden. Die Abkürzung ist immer die erste nicht mehrdeutige Zeichenkette vom Anfang des Kommandos. Die meisten Kommandos wirken sich auf die nächste auszuführende Instruktion aus. Oft kann man eine Adresse mitangeben. Dadurch wird vor der Ausführung der Programmcounter, das ist der Name des Registers in dem die Adresse der als nächstes auszuführenden Instruktion steht, auf diese Adresse verbogen. Der Kämpfer verhält sich in dem Moment nicht mehr so wie es ursprünglich festgelegt war. Innerhalb der Kommandos haben folgende Symbole eine besondere Bedeutung.

. Die aktuelle, zuletzt dargestellte Adresse.

$ Die letzte Adresse.

A A-Wert der aktuellen Instruktion.

B B-Wert der aktuellen Instruktion.

PC Programmcounter des aktuellen Kämpfers.

PC1 Programmconter von Kämpfer 1.

PC2 Programmconter von Kämpfer 2.

LINES Anzahl der Zeilen auf dem Bildschirm.

Innerhalb der Modi pqueue und wqueue haben ähnliche Bedeutungen.

. Aktueller Prozeß/Kämpfer.

$ Letzter Prozeß/Kämpfer.

A A-Wert der Instruktion des aktuellen Prozeß.

B B-Wert der Instruktion des aktuellen Prozeß.

PC Sowie PC1 und PC2 sind 0.

Wird ein Kommando mit einem führenden @ eingegeben, so werden seine Ausgaben auf dem Bildschirm unterdrückt. Nicht aber die Ausgaben in die Protkolldatei. Ein führendes & unterdrückt beides. Kommandos können mit den sogenannten Chains verbunden werden. Sie werden dann hintereinander ausgeführt und bleiben komplett als letztes Kommando erhalten. Kommandos die Eingaben erwarten, bekommen diese nach den nächsten Chains

10.2.1.1. help

(h)elp

Hilfstexte mit kurzen Informationen zu jedem Kommando. Ich hoffe doch das man das nicht zu oft brauchen sollte wenn man das Handbuch auf dem Tisch liegen hat.

10.2.1.2. progress

(p)rogress

Zeigt den Punktestand der Kämpfer und die Gewinn/Niederlage/Unentschiedentabelle des Kampfes an.

10.2.1.3. registers

(r)egisters

Zeigt den Zustand der Simulation an. Folgende Daten werden angezeigt.

- Nummer der Runde von N Runden.
- Rückwärtszähler maximale Anzahl Instruktionen.
- Name des Kämpfers dessen Befehl als nächstes ausgeführt wird.
- Anzahl Prozesse des Kämpfers.
- Aktueller Ausschnitt der Prozeßwarteschlange.
- Ausschnitt aus dem pSpace
- Die letzten 4 Daten auch für jeden anderen am Kampf beteiligten Kämpfer.

10.2.1.4. go

(g)o [Adresse]

Läßt die Simulation ab der aktuellen Instruktion ablaufen. Durch angabe einer Adresse wird der Programmcounter erst auf diese Instruktion gestellt. Entweder bis zur nächsten Haltepunkt oder bis ans Ende des Kampfes. Danach springt der Befehl zum post-mortem debuggen.

10.2.1.5. step

(s)tep [Adresse]

Führt die nächste Auszuführende Instruktion aus und kehrt dann zum Prompt zurück. Wird eine Adresse angegeben so wird diese ausgeführt.

10.2.1.6. continue

(c)ontinue [Adresse]

Springt in den Simulator und beendet den Kampf ohne weitere Unterbrechungen. Sollte eine Adresse angegeben werden so wird diese als erste Instruktion ausgeführt. Die pMARS befindet sich nicht mehr im Debugger, deswegen werden auch keine Haltepunkte mehr beachtet.

10.2.1.7. thread

(th)read [Adresse]

Thread funktioniert wie step, arbeitet aber nur innerhalb eines Prozesses. Damit kann ein Prozess verfolgt werden. Man muß sich daran gewöhnen das zwischen den Schritten die man mit thread macht alle anderen Prozesse eine Instruktion ausführen dürfen. Dadurch kann sich der Core rund um den Befehl so manches mal verändern. Sollte eine Adresse mit angegeben werden so wird die nächste auszuführende Instruktion auf die Adresse geändert.

10.2.1.8. skip

(sk)ip [Anzahl]

Führt im Hintergrund die übergebene Anzahl an Instruktionen aus. Das Kommando `skip 10` ist damit vom Ergebnis wie 10 mal das step-Kommando.

10.2.1.9. execute

(e(x))ecute [Adresse]

Verändert die Adresse der nächsten auszuführenden Instruktion, und führt dann das step-Kommando auf dieser neuen Adresse aus.

10.2.1.10. quit

(q)uit

Verläßt den cdb und die pMARS und kehrt zurück.

10.2.1.11. trace

(t)race [Adresse|Bereich]

Diesem Kommando kann eine Adresse oder ein Bereich übergeben werden. Dann werden in allen betreffenden Feldern die Trace-Bits gesetzt. Das heißt das diese Instruktionen wenn sie in einer durch das go-Kommando gestarten Runden ausgeführt den Debugger-Prompt wieder erscheinen lassen.

10.2.1.12. untrace

(u)ntrace [Adresse|Bereich

Das Gegenstück zu trace. Setzt die Trace-Bits wieder zurück.

10.2.1.13. moveable

(mo)veable on|off

Dieser Befehl legt fest ob das Trace-Bit einer Instruktion mitkopiert wird wenn sie per `mov.i` kopiert wird. Dieser Befehl sieht zwar unscheinbar aus, kann aber von unschätzbarem Wert sein. Als Beispiel kann das austesten von verschiedenen dat-Bomben beim Gegner sein. Wenn die dat-Bomben ihre Trace-Bit mitnehmen, dann kann man den Kampf mit go weiterlaufen lassen. Sobald der Gegner versucht die dat-Bombe auszuführen wird man den Debugger-Prompt erblicken und kann sich den Todeskampf des Gegner in aller Ruhe Schritt für Schritt ansehen.

10.2.1.14. list

(l)ist [Adresse|Bereich]

Das wohl meistgenutzte Kommando. List zeigt Adresssen oder Bereiche des Core's an. Das Listkommando selbst muß nicht geschrieben werden. Sollte man am Prompt eine Adresse oder einen Bereich eingeben, so wird das List-Kommando damit ausgeführt. Das Trace-Bit wird angezeigt. Die Instruktion mit der der Core vorinitialisiert wird - `dat.f 0, 0` wird als leere Zeile angezeigt.

10.2.1.15. edit

(e)dit [Adresse|Bereich]

Mit diesem Kommando kann man einzelne Stellen oder Bereiche im Core mit neuen Instruktionen bestücken. Es wird immer der Inhalt der Instruktion angezeigt. Bei der Programmierung mit edit können labels und equ's benutzt werden.

10.2.1.16. fill

(f)ill [Adresse|Bereich]

Bei Anwendung auf eine Adresse arbeitet das Kommando wie edit. Sobald aber ein

Bereich als Ziel angegeben wird, dann wird das eingegebene Kommando an jede Stelle in dem Bereich kopiert.

10.2.1.17. search

(s)earch Muster

Sucht den Core nach der übergegebenen Zeichenkette ab. Die Suche bezieht das Trace-Bit mit ein. Mögliche Wildcards sind ? für ein beliebiges Zeichen und * für ein oder mehrere beliebige Zeichen.

Beispiel:

```
cdb>search mov.i 0,?
```

Sucht nach Anweisungen die aus einem Imp-Ring stammen könnten.

10.2.1.18. write

(w)rite [Protokolldatei]

Öffnet die Angegebene Protkolldatei und leitet die Ausgaben des Debugger dorhin um. Wird das Kommando ohne Dateinamen aufgerufen, wird die zur Zeit geöffnet Protokolldatei geschlossen und die Ausgabe wieder umgelenkt.

10.2.1.19. echo

(ec)ho [Zeichenkette]

Die übergegebene Zeichenkette wird gefolgt von einem CR ausgegeben.

10.2.1.20. clear

(cl)ear

Löscht den Bildschirm.

10.2.1.21. display

(d)isplay clear|on|off|nnn

Führt Veränderung am Display durch. Clear löscht das Display. Der off-Parameter friert den Bildschirm ein, keine Veränderungen im Core werden angezeigt. On hebt diesen Zustand wieder auf. Die dreistellige Zahlenkombination ist zusammengesetzt wie der Parameter -v von der Kommandozeile.

10.2.1.22. switch

(sw)itch [1|2]

Schaltet zwischen der linken und der rechten Bildschirmhälfte um. Sollte noch keine rechte Hälfte existieren, wird sie erzeugt.

10.2.1.23. close

(clo)se

Schließt die rechte Bildschirmhälfte und erweitert die linke auf den ganzen Bildschirm.

10.2.1.24. calc

(ca)lc Ausdruck1,[Ausdruck2]

Der Kommandozeilenrechner gibt das Ergebnis der Ausdrücke aus. In den Macros wird das Kommando zusätzlich für Zuweisungen an die Variablen c bis z genutzt.

10.2.1.25. macro

(m)acro [Name][,Datei]

Ein Macro ist ein Name für eine Kommandozeile des Debuggers. In der Zeile können mehrere mit sogenannten Chains (~) verbundene Kommandos stehen. Bei Aufruf mit Namen wird das betreffende Macro ausgeführt. Bei Angabe eines Dateinamens, das Komma darf nicht vergessen werden, wird die Datei, in der mehrere Macrodefinitionen nachgeladen. Die Macrodatei "pmars.mac" wird bei Start des Debuggers automatisch eingeladen. Bei Aufruf ohne Namen werden alle zur Zeit verfügbaren Macros aufgelistet. Wenn man bestimmte Macros sehr oft braucht, dann kann man diese auch in die Datei "pmars.mac" schreiben. Der Debugger kann maximal 100 Macros im Speicher halten.

10.2.1.26. if

(if) Ausdruck

Wenn der übergegebene Ausdruck mit 0 ausgewertet wird, dann wird das nächste Kommando übersprungen. Mehrere Kommandos können zwischen !! und ! eingeschlossen werden.

10.2.1.27. reset

(res)et

Bricht die Ausführung des Macros sofort ab und kehrt zum Prompt zurück.

10.2.1.28. pqueue

(pq)ueue 1|2|n|off

Schaltet in den Prozessqueuemodus um. Die Kommandos list, edit und fill arbeiten dann an der Prozeßwarteschlange des angegebenen Kämpfers. Mit off wechselt man wieder zum normalen Coremodus.

10.2.1.29. wqueue

(wq)ueue [off]

Schaltet in den Warriorqueuemodus. Die Kommandos list, edit und fill arbeiten dann an der Kämpferwarteschlange. Mit off kehrt man wieder zum Coremodus zurück.

10.2.1.30. pspace

(ps)pace 1|2|n|off

Das dritte Kommando ihm Bunde. Es wechselt in den pSpacemodus und zurück.

10.2.2. Arbeiten mit dem Debugger

Einen Kampf mit dem Debugger verfolgen gibt Aufschluß darüber was wirklich passiert. Man kann Auswirkungen von veränderten Konstanten sofort sehen. Man kann das Ende des eigenen Kämpfers nach dem Einschlag einer Bombe beobachten um dann geeignete Änderungen vorzunehmen. Die im Debugger vorhandene Kommandos versetzen den Benutzer in die Lage den Core und die Kämpfer zu untersuchen. Gerade bei Kämpfern mit selbstmodifikation muß man mit dem Debugger arbeiten um genau sagen zu können wie der Kämpfer funktioniert.

10.3. Zusatzprogramme

10.3.1. pShell

Als Zusatzprogramm zur pMARS empfehle ich die pShell. Sie stellt eine Pulldownmenugesteuerte Oberfläche für pMARS dar. Das Editieren von Kämpfer die geladen sind, sowie verschiedene Konfigurationen machen das Programmieren und Testen um einiges leichter. Die wichtigsten Kommandos liegen auf den Funktionstasten. Man kann verschiedene Konfigurationen abspeichern und so schnell verschiedene Umgebungen für verschiedene Kämpfer oder Tourniere verwalten.

10.3.2. MTS

Desweiteren ist da noch der MTS. Dieses Programm führt mithilfe der pMARS Tourniere durch. Da die pMARS nur einen Kampf ausführen kann, und bei mehr als Zwei

Kämpfern damit einen Multiwarrior-Kampf inszeniert wird hier ein Zusatzprogramm gebraucht welches aus einer Liste von Kämpfer alle Paarungen berechnet und die pMARS dementsprechend aufruft. Die pShell unterstützt ebenfalls die Bedienung von MTS

11. Die Benchmarks

In den vorherigen Kapiteln haben wir Tricks und Methoden kennengelernt mit denen man einen Kämpfer schneller oder kürzer machen kann. Wir haben uns eine effektive Bombe angesehen und wissen auch wie unser Kämpfer sich nicht selbst damit erledigt. Mit guten Schrittweiten können wir den Core schnell und gründlich abdecken. Jetzt wird es Zeit den Kämpfer gegen andere antreten zu lassen. Das Ziel ist es einen Kämpfer so gut zu machen das er gegen mehrere Arten von Gegnern bestehen kann. Deswegen müssen wir ihn auch gegen mehrere Kämpfer aus verschiedenen Sparten antreten lassen. Es gibt eine Archiv in dem beinahe 1700 Kämpfer gespeichert sind. Dort könnte man sich die verschiedenen Kämpfer runterziehen. Allerdings kann man vom bloßen hinsehen schlecht einschätzen wie gut ein Kämpfer ist. Es sollten schon die besten ihrer Zeit sein. Was nützt es gegen leichte Gegner zu gewinnen, wenn man auf den Hill sofort von jedem Kämpfer zermahlen wird.

Zu diesem Zweck wurden Benchmarks angelegt. Auch in anderem Zusammenhang hat man bei Computern oder Programmen mit Benchmarks zu tun. Man vergleicht das Abschneiden bei einer klar definierten Aufgabe. Es gibt zwei sehr bekannte Benchmarks: Wilkie's und Wilmoo's. Kommentare in einem Kämpfer in der Form: "Score 154 against Wilkies" oder so ähnlich geben das Abschneiden dieses Kämpfer gegen einen Hill mit allen Kämpfer des Wilkies wieder. Damit kann man den Kämpfer schon sehr gut einschätzen. Die folgenden drei Kapitel haben wohl die meiste Zeit gekostet, da ich teilweise recht viel im Einzelschritt überprüfen mußte. Ich versuche möglichst viele Details der Kämpfer zu beschreiben. Hier sollte einiges an guten Ideen zu holen sein.

11.1. Wilkies

Der Wilkies-Benchmark enthält jeweils 4 Kämpfer der drei Kategorien aus der Paper-Scissor-Stone Analogie.

11.1.1. Die Paper

11.1.1.1. Paperone

Paperone von Beppe Bezzi ist ein kompaktes Paper welches durch sein enorme Verbreitungsgeschwindigkeit auffällt.

```
start   spl     1,      <300    ;\
        spl     1,      <150    ;  generate 7 consecutive processes
        mov     -1,     0       ;/
```

Die ersten drei Zeilen erzeugen 7 aufeinander folgende Prozesse. Die B-Felder der beiden spl-Instruktionen sind zudem mit predekrementaler Addressierung angelegt. Dadurch dekrementieren sie bei ihrer Ausführung zwei B-Felder 300 und 150 Instruktionen voraus. Über die Effizienz kann man streiten, aber es kostet nichts und bringt vielleicht etwas.

```
silk    spl     3620,   #0      ;split to new copy
        mov.i   >-1,    }-1     ;copy self to new location
```

Der Kern des Silks ist das anspringen der Kopie bevor diese im Speicher existiert. Alle 7 Prozesse die die spl-Instruktion ausführen werden gesplittet. Der Orginalprozeß zeigt danach auf die mov-Instruktion. Der neu erzeugte Prozeß zeigt auf die Instruktion 3620 Instruktionen weiter vorne. Allerdings liegt dieser Prozeß in der Prozeßwarteschlange hinter dem Orginalprozeß, so das erst die mov-Instruktion ausgeführt wird. Nach dem splitten existieren also für diesem Punkt 14 relevante Prozesse. 7 stehen vor der Ausführung der mov-Instruktion und 7 stehen vor den noch nicht vorhandenen Instruktionen 3620 Stellen weiter vorne. Die 7 ersten Prozesse kopieren den Kämpfer über die beiden Felder der spl-Instruktion an ihr Ziel. Die Zeiger für die Kopieraktion werden somit auch mit der ersten kopierten Instruktion kopiert. Nach dem alle 7 Prozesse die mov-Instruktion ausgeführt haben, führen die 7 eben abgesplitteten Prozesse schon die spl-Instruktion der eben erstellten Kopie aus.

```
        mov.i   bomb,   >2005   ;linear bombing
```

Neben dem überschreiben des Core's mit Kopien von sich selbst, schmeißt Paperone pro Instanz auch noch zwei dat-Bomben. Die erste wird postinkremental über das B-Feld der Instruktion 2005 Felder weiter vorne geschmissen. Da die dat-Bombe als Anti-Imp-Bombe angelegt ist, sollte wir auch die Wirkung auf eine Imp-Spirale beschreiben.

Postinkremental über das B-Feld heißt das die Bombe auf den zu versorgenden Arm der Spirale geschmissen wird. Die wird einen Prozeß verlieren. Danach wird das B-Feld zusätzlich noch Inkrementiert, was wiederum für ein Loch in der Spirale sorgen kann. Wird die Bombe ausgeführt, werden B-Felder der anderen Arme der Spirale inkrementiert, was ebenfalls eine Beschädigung mit sich bringt.

```
        mov.i   bomb,   }2042   ;A-indirect bombing for anti-vamp
```

Die zweite Bombe wird postinkremental über das A-Feld geschmissen. Sie wird damit speziell gegen Vampire eingesetzt, welche ja meist mit jmp-Instruktionen die auf Teile aus ihrem Code zeigen, um sich schmeißen.

```
        add.a   #50,     silk    ;distance new copy
```

Da diese Zeile von allen 7 Prozessen ausgeführt wird, wird das Ziel der nächstem replikation um 350 Felder weiter nach vorne verlegt.

```
        jmp     silk,   <silk   ;reset source pointer, make new copy
```

Alle sieben Prozesse springen die spl-Instruktion an und setzen dabei den Quellzeiger durch die predekrementale Adressierung im B-Feld zurück.

```
bomb    dat.f   >2667,  >5334   ;anti-imp bomb
```

Wie bereits erwähnt ist diese Bombe speziell für den Einsatz gegen Imp-Spiralen konzipiert. Genauer noch: 3-Punkt Imp-Spiralen. Sollte der Arm einer getroffenen Spirale nur aus einem Prozess bestehen, stirbt er. Zusätzlich inkrementiert er noch die Zielfelder der beiden anderen Arme, wodurch in einem weiteren Arm ein Loch entsteht an dem die Spirale ebenfalls sterben wird.

Im Gegensatz zu den anderen Kämpfern aus dieser Kategorie leben die einzelnen Instanzen immer weiter. Dadurch ist Paperone gegen Kämpfer die mit Betäubungsbomben schmeißen fast machtlos.

11.1.1.2. Timescape (1.0)

11.1.1.3. Marcia Trionfale 1.3

11.1.1.4. nobody special

11.1.2. Die Scissors

11.1.2.1. Rave

Rave von Stefan Strack ist ein CMP-Scanner, der ausgemachte Ziele mit einem Teppich von `spl 0`-Instruktionen bedeckt. Nach dem der Scanvorgang abgeschlossen ist, vernichtet ein Coreclear die so betäubten Gegner. Am Ende bleibt einzig eine spl-Instruktion übrig, welche auf das Ende des Kampfes wartet.

```
CDIST   equ 12
IVAL    equ 42
FIRST   equ scan+OFFSET+IVAL
OFFSET  equ (2*IVAL)
DJNOFF  equ -431
BOMBLEN equ CDIST+2
```

Phase 1 - CMP-Scanner

```
scan    sub.f  incr,comp
```

Diese Zeile verändert Quelle und Ziel der cmp-Instruktion um die bei incr angegeben Werte.

```
comp    cmp.i  FIRST,FIRST-CDIST        ;larger number is A
```

Diese Zeile ist der Einsprung in den Kämpfer. Es werden zwei Instruktion die 12 Stellen auseinander liegen miteinander verglichen. Sollten die beiden Instruktionen gleich sein, wird die Ausführung direkt bei der djn-Instruktion weitergeführt.

```
        slt.a  #incr-comp+BOMBLEN,comp  ;compare to A-number
```

Wenn die cmp-Instruktion die Ausführung dieses Befehls erzwingt, dann hat sie etwas gefunden. Bevor jedoch das Bombardement losgeht, wird mit dieser Anweisung sichergestellt das sich der Scanner nicht selbst gefunden hat. Der Befehl veranlasst die djn-Instruktion zu überspringen wenn das A-Feld der cmp-Instruktion größer als 15 ist, was nicht der Fall ist wenn eine der Instruktionen vom Label comp an abwärts gefunden wird. Wichtig ist hierbei das diese Sicherheitseinrichtung die sub-Instruktion bei scan nicht mit einschließt.

```
        djn.f  scan,<FIRST+DJNOFF       ;decrement A- and B-number
```

Diese Instruktion verzweigt wieder nach vorne wenn bei einem Scanvorgang nichts oder der Kämpfer selbst gefunden wurde. Zusätzlich wird durch das predekrementale B-Feld eine sogenannter djn-Stream über einen Teil des Core's gelegt. Große Teile des djn-Stream sind für den Scanner unsichtbar da die Instruktionen die verglichen werden nur 12 Stellen auseinander liegen. Aber an den Enden des Bereich kommt es immer wieder zum Fehlalarm des Scanner und er vertrödelt viel Zeit damit seinen eigenen djn-Stream zu bombardieren. Im Leerlauf, also ohne Gegner macht er das genau 21 mal.

```
        mov.ab #BOMBLEN,count
```

Diese Zeile wird nur ausgeführt wenn der Scanner sowie die Sicherheitüberprüfung das Ziel erfasst haben. Die größe des zu legenden spl-Teppich wird in die djn-Instruktion geschrieben.

```
split   mov.i  bomb,>comp               ;post-increment
count   djn.b  split,#0
```

In dieser Schleife wird der komplette Bereich zwischen den beiden verglichenen Instruktion zuzüglich zwei weiterer Stellen mit der spl-Bombe überschrieben. Jeder aktive Code der sich dort befindet wird damit gezwungen sich in immer mehr Prozesse zu splitten. Der Gegner wird dadurch fast bis zum Stillstand verlangsamt.

```
        sub.ab #BOMBLEN,comp
```

Mit dieser Anweisung wird das B-Feld der cmp-Instruktion wieder korrigiert. Zugunsten der Geschwindigkeit wurde es von der mov-Instruktion genutzt.

```
        jmn.b  scan,scan
```

Nach erfolgtem Bombardement wird der Scanvorgang fortgesetzt. Allerdings wird der Scanner irgendwann die sub-Instruktion beim Label scan finden und nicht von der slt-Instruktion am Bombardement gehindert werden. In diesem Fall wird nicht mehr weiter gescannt da die Instruktion nicht mehr verzweigt.

Phase 2 - Coreclear

```
bomb    spl.a  0,0
        mov.i  incr,<count
incr    dat.f  <0-IVAL,<0-IVAL
```

Der Coreclear beginnt bei der djn-Instruktion mit seinem vernichtenden Werk. Nachdem der Core umrundet ist wird auch die mov-Instruktion durch eine dat-Bombe überschrieben. Damit mutiert der Coreclear zu einem Einzeller der möglichst klein darauf hofft von einem Gegner der bis dahin überlebt hat nicht gefunden zu werden.

11.1.2.2. Iron Gate

Iron Gate von Wayne Sheppard, scheint ein naher verwandter von Rave zu sein. Tatsächlich sind die einzigen Unterschiede in der Scanrichtung und Dichte und Zusammensetzung des Bombenteppich zu finden. Die Konstanten

```
dist    equ 73          ; Old scan distance = 98
scan    equ dist*2
```

Aufgrund der nahen Verwandtschaft, kann der geneigte Leser die Unterschiede zu Rave selber rausfinden.

```
        add     off,    @x
loc     cmp     dist-1, -1
        slt     #14,    @x
        djn     -3,     <7000
        mov     j,      @loc
x       mov     s,      <loc
        sub     new,    @x
        jmn     loc,    loc-1
s       spl     #0,     <1-dist
        mov     2,      <-2
j       jmp     -1
new     dat     <0-dist,<0-dist-1
off     dat     <scan,  <scan
        end
```

11.1.2.3. Thermite 1.0

Thermite von Robert Macrae ist eine gelungene Kombination aus Quickscanner, Vampire und Incendiary Bomber. Gefunden Ziele werden kurz Bombardiert um dann gründlich ausradiert zu werden.

```
SPC     equ 7700
STP1    equ 81
Lookat  equ look+237+8*(qscan-1)*STP1

traptr  dat     #0,      #trap
bite    jmp     @traptr, 0

look
qscan   for     6
```

```
         sne.i   Lookat+0*STP1, Lookat+2*STP1
         seq.i   Lookat+4*STP1, Lookat+6*STP1
         mov.ab  #Lookat-bite-2*STP1, @bite
        rof

        jmn      test+1, bite
```

Dies ist nur der erste von 4 Blöcken die den Quickscanner von Thermite ergeben. Ihre unglaubliche Geschwindigkeit im scannen verdanken Quickscanner solchen Konstruktionen. Das Macro wird von der Mars, genauer dem Assembler der Mars in 6 sne/seq/mov-Blöcke mit einer abschließenden jmn-Instruktion ausgerollt. Die verschiedenen Variablen ergeben die Scanpositionen. Wie der cmp-Scanner vergleicht der Quickscanner Positionen im Core miteinander. die sne-Instruktion vergleicht zwei Stellen im Core miteinander. Sollten sie nicht übereinstimmen, so wird die nächste Instruktion übersprungen. Die zweite Instruktion überspringt wenn die beide Stellen im Core gleich sind. Die dritte Instruktion speichert einen Startwert der als Anfangspunkt für ein Bombardement dienen kann. Wichtiger ist aber das erst die nächste jmn-Instruktion aufgrund dieses Wertes feststellt das die vor ihm stehende Block etwas gefunden hat. Sollte also eine der mov-Instruktionen ausgeführt worden sein, so wird die jmn-Instruktion daraufhin zur Entscheidungsphase springen.

```
test    jmz.b   blind,   bite
```

Da der letzte Quickscan-Block keine abschließende jmn-Instruktion hat, wird hier geprüft ob ein Gegner gefunden wurde. Falls nicht wird sofort der Bomber aufgerufen.

```
        add     #STP1*2, bite
        jmz.f   -1,      @bite
```

Dadurch das vier mögliche Stellen im Core betroffen sein können, folgt nun ein zweiter Scanloop, welcher das Ziel genau erfassen kann.

```
        mov     spb,     @bite
```

Da bis jetzt schon eine ganze Menge an Cycles vergangen sind, wird bevor das eigentliche Bombardement des Zielbereichs stattfindet, eine spl-Bombe im Zielbereich plaziert. Ein Gegner ist damit zumindest schon mal gefangen.

```
        add     #49,     bite
attack  sub.ba  bite,    bite
loop    mov     bite,    @bite
        add.f   step,    bite
        djn     loop,    #24
```

Nach einer kurzen Initialisierung in dem die jmp-Instruktion auf dem Gegner eingestellt wird, wird der Bereich in dem der Gegner vermutet wird mit den immer wieder neu berechneten jmp-Instruktionen die auf die Falle zeigen, gespickt.

Nachdem ein Gegner über die jmp-Instruktionen in die Falle gezwungen wird, bzw. bei der Auswertung des Quichscanners festgestellt wurde das kein Gegner gefunden worden ist, wird der Incendiary bomber gestartet.

```
bstp    equ    155
gap     equ    15
offset  equ    130
count   equ    1500

blind
spb     spl    #0,         <-gap+1
        add    #bstp,      1
        mov    spb,        @tgt-offset
        mov    mvb,        @-1
tgt     djn.f  -3,         >300
        mov    ccb,        >spb-1      ; Uses copied mvb for CC.
        djn.f  -1,         <spb-18     ; Aids clear.
mvb     mov    gap,        >gap        ; mov half of the incendiary
ccb     dat    0,          10          ; Core Clear.
```

Die beiden Teile der Bombe sind als Instruktionen im Code des Bombers vorhanden. Durch geschicktes Design sind so zwei Instruktionen gespart worden. Durch selbstbombardement verwandelt sich der Bomber in einen Coreclear, der dann die betäubten Gegner erledigt

```
trap    spl    0,      >-200         ; Lackadaisical attempt at gates.
        spl    -1,     >-200+2667    ; Each increments many times between
        jmp    -2,     >-200+2*2667  ; imp steps, but then the whole imp
                                     ; moves! I only blow away rings...

step    dat     #6,     #-6           ; QS step size. Up from 5 for speed.

        end    look
```

Das letzte Stück Code von Thermite ist die Falle in die gefundene Gegner geleitet werden soll. Diese ist ein Imp-Gate, welches speziell gegen Imp-Spiralen wirkt. Das erkennt man an für Dreipunkt-Impspiralen typische Werten von 2667.

Ich habe mir mal die mühe gemacht und die im Quickscanner durchsuchten Positionen ausgegangen von der ersten Instruktion des Kämpfer zu ermitteln.

239	1211	2183	3155	4127	5099	6071	7043
320	1292	2264	3236	4208	5180	6152	7124
401	1373	2345	3317	4289	5261	6233	7205
482	1454	2426	3398	4370	5342	6314	7286
563	1535	2507	3479	4451	5423	6395	7367
644	1616	2588	3560	4532	5504	6476	7448
725	1697	2669	3641	4613	5585	6557	7529
806	1778	2750	3722	4694	5666	6638	7610
887	1859	2831	3803	4775	5747	6719	7691
968	1940	2912	3884	4856	5828	6800	7772
1049	2021	2993	3965	4937	5909	6881	7853
1130	2102	3074	4046	5018	5990	6962	7934

Wie man sehr schnell sieht wir zwischen dem Bereich 239-7934 jede 81te Instruktion gescant. Daraus kann man zwei Tatsachen ableiten. Quickscanner sind nicht allmächtig. Kleine Gegner haben durchaus die Chance dem doch sehr grobmaschigen Netz des Scanners zu entgehen. Zweitens sollten Kämpfer die größer als 81 Instruktionen, was heute wo fast jeder Kämpfer einen Quickscanner bzw. einen seiner Nachkommen als erstes ausführt, zumindest Bootstrapping für seine Komponenten nutzen. So findet des Scanner einen Dummy den er gerne „zerbomben“ kann.

11.1.2.4. Porch Swing

Porch Swing ist ein harter Gegner für jeden Anfänger. Er kämpft in zwei Phasen. Die erste Phase ist eine Kombination von Stone und CMP-Scanner. Das ganze wird von einem kleinen DJN-Stream abgerundet. Alle Aktionen zusammengenommen arbeitet Porch Swing in dieser Phase mit .8c. Diese Geschwindigkeit erlaubt es ihm auch auf optimale Stepsizes zu verzichten. Er bombardiert und scannt einfach durch den Core. Sollte er keinen Gegner finden, wird er, wenn er bei seinem eigenen DJN-Stream angekommen ist in die zweite Phase springen. Die zweite Phase ist ein trickreicher Multipass-Coreclear, der durch einen DJN-Stream gesteuert wird.

Hier aber erstmal die Konstanten.

```
STEP    equ    12
EXTRA   equ    4
DJNOFF  equ    (-426-EXTRA)
```

Phase 1 - .8c Scan/Bomb/DJN-Stream

```
        dat.f  #gate-10,  step-gate+5
gate    dat.f  #gate-10,  sneer-STEP+1
dat2    dat.f  #gate-20,  step-gate+5
dat1    dat.f  #gate-25,  step-gate+5
site2   spl.a  #gate-30,  step-gate+5
site    spl.a  #gate-40,  step-gate+5
for EXTRA
```

```
        dat.f  0,         0
rof
```

Hier sind die verschiedene Bomben für den Multipass-Coreclear gelagert. Sie werden gleichzeitig auch als Index für die Positionierung eingesetzt.

```
adder   sub.f  sweeper,   sneer
```

Diese Instruktion berechnet die nächsten Bombenziele bzw. Scann -Positionen.

```
hithigh mov.i  step,      *sneer
hitlow  mov.i  step,      @sneer
```

Zwei Dat-Bomben werden indirekt über A.- und B-Feld der Anweisung bei `sneer` geworfen.

```
sneer   sne.i  @gate+STEP*6-1-EXTRA, *gate+STEP*3-EXTRA  ;so we bomb step
```

Der CMP-Scanner vergleicht die Instruktionen 12 Instruktionen vor der ersten und 12 Instruktionen nach der zweiten Bombe. Sollte hierbei eine nichtübereinstimmung gefunden werden, wird die zweite Phase eingeleitet.

```
looper  djn.b  adder,     <DJNOFF
```

Der DJN-Stream beginnt 430 Instruktionen vor dieser Anweisung und arbeitet Rückwärts dem Scan- und Bombenteppich entgegen. Sollte kein Gegner gefunden werden, so löst er dann auch die zweite Phase aus.

Phase 2 - Multipass-Coreclear

```
setup   add.f  sneer,     gate
```

Diese Instruktion macht den Coreclear "scharf". Die Positionen bei `sneer` werden zu den Werten bei `gate` addiert. Dadurch das `gate` 12 Instruktionen vor `sneer` liegt beginnt der Coreclear auf der vorderen Scanposition.

```
sweeper spl.a  #-STEP*4+1,<-STEP*4+1
```

Selfsplitting wie aus dem Lehrbuch.

```
mover   mov.i  @swinger,  >gate
```

Indirekt über das B-Feld der Instruktion bei `swinger` wird die zu werfende Bombe ermittelt. Das hört sich zwar umständlich an, ist aber einfach genial. Zu dem Zeitpunkt an dem der Coreclear läuft, ist er immun gegen seinen DJN-Stream. Darüber hinaus wird er durch diesem sogar noch gesteuert. Zu Beginn des Coreclear steht im B-Feld der Instruktion bei `swinger` noch der Verweis auf `site`. Das heißt das der Coreclear mit SPL 0- Instruktionen arbeitet. Wenn der DJN-Stream das B-Feld der Instruktion bei **swinger** zwei mal passiert hat dann verweist dieses auf die DAT-Instruktionen. Damit wird der Core zweimal mit SPL-Instruktionen und danach mit DAT-Instruktionen ge'cleart.

```
swinger djn.b  mover,     {site
```

Endlosschleife des Coreclears mit DJN-Stream.

```
step    dat.f  <STEP,     <-STEP
```

DAT-Bombe für die erste Phase.

11.1.3. Die Stones

11.1.3.1. Tornado

Tornado von Beppe Bezzi besitzt zwei Phasen. Die erste Phase ist ein recht schneller, 0.6c um genau zu sein, dat-Bomber. Die zweite Phase ist ein Coreclear. Diese Phase läuft bis ans Ende des Kampfes weiter. Zum besseren Verständnis hier die Konstanten des Kämpfers.

```
step    equ 95
count   equ 533
gate    equ start-1
```

Phase 1 - Mod-5 Bomber

```
start   mov     bombd,  *stone
        mov     bombd,  @stone
```

Zwei Bomben werden über A und B-Feld der Instruktion bei stone geschmissen.

```
stone   mov     step+2, @(2*step)+2
```

Der Kunstgriff, der dem Bomber die Geschwindigkeit von 0.6c verleiht, ist hier zu sehen. A und B-Feld über die eben noch geworfen wurde, sind ebenfalls Teile einer mov-Instruktion. Die Quelle ist jetzt direkt Adressiert, also die erste der eben geworfenen dat-Bomben. Das Ziel wird indirekt über das B-Feld der zweiten geworfenen dat-Bombe berechnet.

```
add     incr,   stone
jump    djn.b   start,  #count
```

Die berechnung der nächsten Koordinaten findet direkt in der mov-Instruktion bei stone statt. Dadurch werden die Koordinaten für drei Bomben mit einer add-Instruktion berechnet. Das ist sehr effektiv und schwer zu toppen. Die djn-Instruktion verzweigt 533 durchläufe wieder nach Start. Somit werden $3 * 533 = 1599$ dat-Bomben geschmissen. Die Koordinaten der ersten Bomben sind so gewählt, das die letzte Bombe den Kämpfer treffen würde nicht mehr geschmissen wird.

Phase 2 - Coreclear

```
incr    spl     #3*step,#3*step
```

Die spl-Instruktion diente bisher der Berechnung der nächsten Positionen. Durch die unmittelbare Adressierung wird jetzt praktisch der Befehl `spl 0` ausgeführt.

```
clr     mov     bombd,  >gate
```

Der Coreclear beginnt mit dem Feld auf den das B-Feld der Instruktion vor start verweist. Normalerweise würden wir jetzt den schnellen Tod des Kämpfers erwarten. Allerdings wurde in Phase 1 eine Bombe genau an diese Stelle geschmissen. Somit fängt der Coreclear 95 Instruktion entfernt von diesem Feld an. Damit ist keine Gefahr für den Kämpfer. Derselbe Mechanismus schützt den Kämpfer auch nachdem der Coreclear den Core umrundet hat. Sobald die dat-Instruktion mit ihrem 95er B-Feld auf das Feld in dem der Coreclear seine Positionen berechnet gelegt wird, entsteht automatisch eine Lücke von 95 Felder n. Somit kann der Coreclear immer weiter den Core löschen. Wenn es sein muß bis ans Ende von `MAXCYCLES`.

```
bombd   dat     step,   step
```

Die Bombe muß wohl nicht weiter beschrieben werden.

11.1.3.2. Blue Funk 3

```
step equ 3044
```

Jeder der mal einen Mod-4 Bomber geschrieben hat, wird diese Konstante wohl kennen.

```
for 78
dat <imp,<2667
rof
```

Ein großer Decoy, um Gegnerische Scannern ein gutes Ziel zu bieten. Durch den Bezug auf **imp** wird jede Zeile des Decoy's anders

Der erste Teil enthält den Stone der zu beginn des Kampfes per Bootstrapping woandershin verschoben wird.

```
emerald   SPL #-step,<step
```

Selfsplitting-Architektur gibt dem Stone etwas Power gegen Dat-Bomben. Ausserdem wird dadurch die Imp-Spirale stark verlangsamt, was grundsätzlich gut ist für Imp-Spiralen.

```
stone     MOV >-step,step+1
```

Da diese mov-Instruktion keine feste Stelle aus dem Kämpfer als Bombe nimmt, sondern eine Stelle aus dem Core postinkremental übers B-Feld holt, werden zwei Speicherstellen im Core manipuliert.

```
          ADD emerald,stone
```

Quell.- und Zielzeiger der mov-Instruktion werden verändert.

```
cnt       DJN.f  stone,<emerald-50
```

Klassische Situation für einen DJN-Stream. Dieser setzt 50 Instruktionen hinter dem Stone an.

```
cc   dat #-7

boot mov cc,out-200+5
out  spl @0,out-200
     mov emerald,>out
     mov emerald+1,>out
     mov emerald+2,>out
     mov emerald+3,>out
```

Das einzig erwähnenswerte an diesem Bootvorgang ist das die Kopie ähnlich wie bei einem Silk replicator angesprungen wird bevor sie existiert.

Nach dem Bootstrapping des Stone's wird eine 8 Prozeß 3 Punkt Imp-Spirale aufgebaut. Wichtig dabei ist das Timing. Die Prozesse müssen sich gegenseitig die Instruktionen zurechtlegen. Im Einzelschritt, mit dem Kommando „thread“ kann man sehr gut verfolgen wie ein einzelner Arm der Spirale gebildet und angesprungen wird.

```
     spl i
     spl i31
i12  spl imp2
imp1 jmp >0,imp
i31  spl imp1
imp3 jmp >0,imp+5334
i    spl i12
     spl imp3
imp2 jmp >0,imp+2667

imp  mov.i #3044,2667

end boot
```

11.1.3.3. Cannonade

11.1.3.4. Fire Storm v1.1

Die Konstanten..

```
boot    EQU Storm+4000
step    EQU 155
init    EQU step
away    EQU 18
```

```
        SPL 0, 1
        JMZ -1, 1
        JMP @-1, 1
        ...
        ...
        DAT #5, #1
        JMN -2, 1
        JMZ -5, 1
        DAT #2, #1
        DAT #5, #1
```

(Stark verkürzte Fassung)

Wenn man bei dat-Bomben mit nicht-null A.- und B-Felder von „coloured Bombs“ spricht, dann ist das wohl ein sehr bunter Decoy, nicht anderes. Das ganze ist nur dazu da Scanner möglichst lange zu beschäftigen.

```
launch  MOV Storm+4, boot+4
        MOV Storm+3, <launch
        MOV Storm+2, <launch
        MOV Storm+1, <launch
        MOV Storm+0, <launch
        MOV Storm-1, <launch
        MOV core, away+boot
        MOV fire, boot+Storm-ptr-away
        JMP boot, <2000
```

Ein simpler Bootstrap mit Ausführung nach der kopieren. Simpel ist keinesfalls negativ gemeint. Sollte genügend Platz vorhanden sein, so muß nicht getrickst werden. Der Code kann dadurch einfach ausfallen, was auf die Dauer leichter zu warten ist. Vielleicht kommt die Zeit an dem man doch noch kürzen muß, hier hätte man die reserve stecken.

```
ptr     SPL 0, <Storm-away
Storm   MOV <1-step, 2+step
        SUB Storm+away, -1
        JMP -2, <-2000
clear   MOV @Storm, <Storm+away
        DJN -1, <3975

core    DAT #step, #0-step
fire    DAT <Storm-ptr-away-1, #0

        END launch
```

Dieser Stone Manipuliert ebenfalls zwei Speicherstellen pro Loop. Die eine durch direktes Bombardement und die andere durch die predekrementale Adressierung. Nach

4806 Cycles wird die jmp-Instruktion durch die spl-Instruktion bei **ptr** überschrieben. Danach verwandelt sich der Stone in einen Zweiphasen-Coreclear. Im Gegensatz zum Bootloader ist dieser Code sehr kompakt und gut optimiert.

12. 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 auch übernehmen. Man sollte versuchen etwas neues zu finden. Die Geschwindigkeit zu optimieren oder einfach kürzer sein als alle anderen.

12.1. 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.

12.1.1. Parallele Prozesse

Besonders bei Kämpfern der Sorte Silk ist es wichtig das 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 wo die Prozesse erzeugt werden sollen. Nach jeder Ziffer in die nächste Zeile wechseln. In alle Zeilen mit einen 1 schreibt man dann `spl 1` und in alle Zeilen mit der 0 `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          ; 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 ist hier eher die Regeln als die Ausnahme.

12.1.2. Anzahl der Prozesse verkleinern.

Eine Sache die etwas bei Prozesse stört ist das man sie nur schwer wieder loswerden kann wenn sie einmal erzeugt hat. Als Beispiel muß hier wieder der Silk Replicator herhalten.

Nachdem die 7 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 zuviele Prozesse sind die den Code ausführen. 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.

12.1.3. 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 einen 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 Addressierung 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 das 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 einen 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 Core's. 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.

12.1.4. Effektive Bomben

Wenn wir nochmal 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 zuende. 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 her. Folgende Bombe aus dem CoreWarrior 18 ist sehr interessant.

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

Diese Bombe entfalten 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 das 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 das 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 abhängen das der Gegner auch wirklich bei der ersten Instruktion mit der Ausführung beginnt.

12.2. Stepsize - Schrittweite

Das Problem den Gegner zu treffen ist nicht zu unterschätzen. Beim Dwarf kommt auch noch das Problem das er sich nicht selber treffen darf, dazu. 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 besonders. 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	unentsch.
	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 das sie sich nicht treffen können. Alle anderen Kämpfe teilen die beiden sich, dort 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	unentsch.
	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 welcher Position 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 Kämpfes 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.

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 die richtige Schrittweite durch probieren finden. Allerdings ist der Aufwand sehr hoch. Nicht jeder Kämpfer ist so einfach gestrickt wie unsere Dwarf's. Ich möchte jetzt anhand des Programms Corestep von Jay Han zeigen wie man den Dwarf anhand besserer Schrittweiten tunen kann.

12.2.1. Corestep

Dieses Programm berechnet günstige Schrittweiten für Kämpfer. Das müssen nicht immer Bombardierungs-Schritte sein. Mit B-Scanner geht es genausogut. 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 anhand 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 aufaddiert. Durch die Anzahl der gelegten Bomben dividierd 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 auffaddiert und durch die Anzahl der gelegten Bomben dividiert. Je schneller die Schrittweite hier kleinere Ergebnisse erziehlt umso besser ist 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 treffer. 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 eventl. 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 Ausgeben.

-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 Kämfer 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 das wir richtigen Modulowert nicht Benutzen. Sonst würde sich der Dwarf nach kürzester Zeit selbst beschießen, 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 mal 100 Kämpfe. Der DwarfE hat die corestep-Konstanten.

	Dwarf1 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 das gute Schrittweiten bei der Kampfkraft eines Gegner ein sehr große Rolle spielen. Das ist bei einem simplen Kämpfer wie dem Dwarf genau wie bei einem Quickscanner oder einem Silk Replicator. Allerdings muß dabei gesagt werden das Corestep nicht hilfreich ist wenn es um Paper oder Silk geht. Die Schrittweiten müssen dort nach anderen Kriterien gewählt werden.

12.2.2. Optima

Ich gehe hier auf Optima nicht weiter ein, da es nach dem selben Algorithmus arbeitet wie Corestep mit -cc, kommt es auf die gleiche Ergebnisse. Allerdings bietet es nicht die möglichkeiten wie Corestep. Nur Coregröße, Modulo und max. Schrittweite können eingegeben werden.

12.2.3. mOpt

Dieses Programm unterscheidet sich in Bedienung und der Art der zu übergebende Parameter von den beiden 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 festgelgt 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 Bombardierungsinterval in der Increment/Offset Notation formulieren. Da wir beide Inkremente frei wählen lassen können müssen wir uns nur um den Modulowert Gedanken machen. Auch wenn man mit verschiedene Schrittweiten den Core besser abdecken können, so sollte doch keine 2 Bomben aufeinander fallen. Deshalb beschränken wir die möglichen Inkremente auf Modulo 8. Die beiden Offsets sollte um 4 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 etwas etwas länger dauern. Man kann die Berechnungen mit Control-C abbrechen, und bekommt dann die besten bisher ermittelten Kombinationen. Man muß 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.

13. King of the Hill

Zur Zeit gibt es vier Server mit verschiedenen "Bergen". King of the Hill ist der Kämpfer auf dem ersten Platz. Dort möglichst lange zu bleiben ist Ziel der Programmierer die ihre Kämpfer dort hin schicken. Je älter der Kämpfer wird, desto besser ist er. Ich beschreibe nun kurz die 4 Server bzw. die Hill's die auf ihnen zu finden sind.

13.1. www.Koth.org

Dieser Server bietet zur Zeit 6 Hill's an. Diese unterscheiden sich durch de Größe des Core's, des Instrukions-Standart oder der Anzahl der Kämpfer. Als Kurzbeschreibung nehme ich die Spezielle Besonderheit dieses Hill's.

ICWS'88 Standart Hill Der Orginal-Online Hill.

ICWS'94 NOP Hill Nutzung des pSpace nicht erlaubt.

ICWS'94 Experimental Hill Ein sehr viel größerer Core fordert smartere Kämpfer.

ICWS'88 Tourney Hill Orginal-Standart für ICWS-Turniere

ICWS'94 Multiwarrior Hill Alle Kämpfer kämpfen gleichzeitig gegeneinander.

ICWS'94 Multiwarrior X Hill Alle Kämpfer gleichzeitig und Experimentelle Einstellungen.

Wenn man einen Kämpfer auf einen der Hills schicken möchte so wird der Hill in der ersten Kommentarzeile benannt. Folgende Schritte um einen Kämpfer auf den ICWS'94 NOP Hill zu schicken.

1. Kämpfer darf keinen pSpace benutzen.
2. Die erste Zeile im Quellcode muß - `;redcode-94nop`, lauten.
3. Eine eMail, ohne Betreff an `koth@KOTH.org` schicken die den Quelltext des Kämpfers enthält.

Wenn das alles Korrekt erledigt ist, dann wird der Kämpfer auf dem Hill antreten.

Jetzt eine Übersicht aller Berge mit den Daten des Berges sowie der Anmeldungszeile. Damit die Berge auch zuhause Simuliert werden können sind die Parameternamen für die pMars mit dabei. Es reicht nicht die `;redcode`-Zeile einzusetzen, man muß alle Einstellungen eintragen.

	88 Std.	94 NOP	94 Exp.	88 Turn.	94 Multi	94 Multi X
Anmeldung	;redcode	...-94nop	...-94x	...-icws	...-94m	...-94xm
`Instr.Set`	-8			-8		
`CORESIZE (-s)`	8000	8000	55440	8192	8000	55440
`MAXCYCLES (-c)`	80000	80000	500000	100000	80000	500000
`MAXPROCESSES (-p)`	8000	8000	10000	8000	80000	10000
`MAXLENGTH (-l)`	100	100	200	300	100	200
`MINDISTANCE (-d)`	100	100	200	300	100	200

Darüberhinaus akzeptiert dieser Server folgende Kommandos innerhalb der Kommentarzeilen des Redcode-Listings.

;help Schickt eine Befehlsübersicht des Server zurück.

;status Statusbericht der Hills.

;statistics Statistik des Servers.

quite Dieses Kommando wird hinter die Anmeldung in der `;redcode`-Zeile geschrieben. Es vermindert die Anzhal der Benachrichtigungen die man über den Kämpfer bekommt während er auf dem Hill ist.

verbose Dieses Kommando wird ebenfalls hinter die Anmeldung in der `;redcode`-Zeile geschrieben. Es sorgt dafür das mehr Benachrichtigungen über den Kämpfer eingehen während er auf dem Hill ist.

test Ebenfalls hinter der Anmeldung einzutragen, wird der Kämpfer nur Testweise assembliert wird. Er wird auf keinen Hill transferiert.

;password Dieses Kommando ist eine Absicherung das ein Kämpfer nicht von einem anderen Programmierer mit dem `;kill`-Befehle vom Hill geschmissen werden kann. Es definiert ein Passwort für diesen Zugriff.

;newpasswd Legt ein neues Passwort fest. Funktioniert nur wenn neue `;redcode`, `;name` und `;password`-Zeilen mit angegeben werden.

;newredcode Setzt eine neue `;redcode`-Zeile im Kämpfer ein. Dies ist zum Beispiel nützlich wenn man die Häufigkeit der Benachrichtigungen ändern will. Das Kommando funktioniert nur mit neuen `;redcode`, `;name` und `;password`-Zeilen

;url Webadresse oder eMailadresse des Authors.

;version Versionen der Hill-Module?

13.2. Pizza Hill

Eine weiterer Server mit 4 Hill's ist unter `www.ecst.csuchico.edu/~pizza/koth` zu finden. Als Besonderheit möchte ich hier den Beginner's Hill nennen. Erst ist für Newbie's gedacht. Kämpfer die älter als 100 sind werden von diesem Hill entfernt. Man sollte den Kämpfer dann aber auf dem normalen 94'Standart laufen antreten lassen.

Eine weitere Besonderheit ist das der Befehl `test` in der `;redcode`-Zeile den Kämpfer gegen alle Kämpfer des Hill's antreten läßt, ohne dabei einfluß auf die Punkte zu nehmen. Man bekommt die Ergebnisse dann per eMail geschickt und weiß wie der Kämpfer abgeschnitten hätte. Seit der Einführung dieses Kämpfer ist es sehr viel schwieriger geworden einen Kämpfer mit sehr hohen Alter zu erstellen. Früher wurden alle Testläufe mitgezählt. Doch nun zu den Hills.

ICWS'94 Draft Hill Der Standart-Hill.

Experimental Hill Kleiner Core für kleine Kämpfer.

Beginner's Hill Daten wie der Standart-Hill. Die Kämpfer werden aber mit einem Alter von 100 Kämpfen vom Hill geschmissen.

Limited Process Hill Maximal 8 Prozesse pro Kämpfer. Die andere Werte leicht verändert.

Kämpfer die auf einem der Pizza Hill's antreten sollen müssen per eMail an `pizza@ecst.csuchico.edu` mit dem Betreff `koth` geschickt werden.

	Standart	Experimental	Beginner	Limited Process
Anmeldung	;redcode-94	...-94x	...-b	...-lp
`Instr.Set`				
`CORESIZE (-s)`	8000	800	8000	8000
`MAXCYCLES (-c)`	80000	80000	8000	80000
`MAXPROCESSES (-p)`	8000	800	8000	8
`MAXLENGTH (-l)`	100	20	100	200
`MINDISTANCE (-d)`	100	20	100	200
`ROUNDS (-r)`	200	200	200	200

Bei diesem Server gibt 2 Kommando die zu erwähnen sind.

show Dieses Kommando steuert ob der Quellcode des Kämpfers für andere sichtbar ist oder nicht. Zusätzlich kann man nur Teile des Quellcodes verstecken. Um den gewünschten Modus zu erreichen kannman folgende Parameter verwenden.

nosource Der Quellcode des Kämpfer ist nicht sichtbar.

source Der Quellcode ist sichtbar.

on/off Quellcode der sichtbar ist kann mit dem Parameter `off` teilweise versteckt werden. Dieses Kommando kann mitten im Quelltext stehen um ein ganz besonderes Implementationsdetail zu verbergen. Mit dem Parameter `on` wird der Quelltext wieder sichtbar:

change Mit dem `change`-Kommando kann man einzelne Zeilen im Kämpfer-Quellcode ändern, ohne den Kämpfer vom Hill nehmen zu müssen. Dies betrifft natürlich nur die Kommentarzeile zu Beginn des Quellcodes. Die `;redcode`-Zeile muß immer dabei stehen, damit der betreffende Hill ausgewählt werden kann. Die `;redcode`-Zeile kann ebenfalls geändert werden. Allerdings nur die Kommando `quite/verbose`.

13.3. Mount Olympus

Dieser Berg unterscheidet sich etwas von den ersten beiden. Er ist ein “infinite”-Hill. Das bedeutet das alle Kämpfer für immer auf dem Hill belieben. Die Kämpfer werden einfach an `Damien.Doligez inria.fr` geschickt. Hier die Daten des Hills.

Es würde sehr viel Zeit kosten bei jedem neu eintreffenden Kämpfer alle Kämpfe die nötig sind durchzuführen. Deshalb wird folgendes Verfahren angewendet.

1. Den Neuankömmling gegen 50 zufällig aus allen Bereichen des Hills ausgewählte Kämpfer antreten lassen.
2. Die Rangliste anhand dieser Ergebnisse neu berechnen.
3. Ermitteln aller Kämpfe die die Tops 25 gegen jeden anderen und jeder Kämpfer gegen die 10 über und unter ihm.
4. Alle Kämpfe laufen lassen die noch nicht liefen.
5. Neuberechnung der Rangliste.
6. Wiederholen der Schritte 3-5 bis keine Kämpfe mehr durchzuführen sind.

Am Ende hat jeder Kämpfer gegen die Top 25 und gegen die 10 über und unter ihm gekämpft. Dies ergibt sicher eine vernünftige Einschätzung des Kämpfer.

Wenn kein neuer Kämpfer eintrifft, nutzt das System die Zeit und läßt Kämpfe laufen die noch nicht stattgefunden haben. So kann das System sehr schnell die Insteressanten Kämpfe laufen lassen. Mit der Zeit werden alle Kämpfe durchgeführt.

13.4. Koenigstuhl

Der Königstuhl bietet gleich 7 “infinite” Hills an.

A. Opcode Referenz

DAT	Prozeß wird terminiert.
MOV	Kopiert A nach B.
ADD	Addiert A zu B, speichert Ergebnis in B.
SUB	Subtrahiert A von B, speichert Ergebnis in B.
MUL	Multipliziert A mit B, speichert Ergebnis in B.
DIV	Dividiert B durch A, Ergebnis nach B wenn A<>0, sonst Prozeßterminierung.
MOD	Dividiert B durch A, Rest nach B wenn A<>0, sonst Prozeßterminierung.
JMP	Ausführung nach A umleiten.
JMZ	Ausführung nach A umleiten wenn B=0.
JMN	Ausführung nach A umleiten wenn B<>0.
DJN	Dekrementiere B, wenn B<>0, Ausführung nach A umleiten.
SPL	Erzeuge neuen Prozeß bei A.
SLT	Überspringe nächste Instruktion wenn A<B.
CMP	Wie SEQ
SEQ	Überspringe nächste Instruktion wenn A = B.
SNE	Überspringe nächste Instruktion wenn A<>B.
NOP	Keine Operation (No OPeration).
LDP	Lade Inhalt des pSpace an Adresse A in die Coreadresse B.
STP	Speichere A in pSpace an Adresse B.