Miscellaneous tips & tricks

- Index
- Bootstrapping
- Core-clear
- Self-splitting
- Bombing/scanning pattern
- Self-mutate
- Online processes

Bootstrapping

Some warriors bootstrap their codes by copying their functional ones to somewhere distant from their original ones. Its general purposes are to arrange a better setup and to leave the original codes as decoys.

(back to Index)

Core-clear

A core clear is basically a linear bombing stone. Its size and function fit in many warriors as the routine that completely clear the arena.

Typical core clear is:

```
        mov 2, <1
        jmp -1, -1
```

or:

```
        spl 0, 0
        mov 2, <-1
        jmp -1, -1
```

The presence of *SPL 0* is to prevent the warrior from self termination.

(back to Index)

Self-splitting

Self-splitting or *SPL 0* is such a peculiar instruction that can be used as either a weapon or a protection.

As a weapon, SPL 0 is usually accompanied by JMP -1 and thrown together into the core. These two instructions are very lethal against replicators because they can hinder and eventually stall their progress.

Self-splitting can also be used to improve the warrior endurance. Consider the following example:

```
        MOV 3, 3
        ADD #165, -1
        JMP -2

If any part of this code is hit, the program will terminate immediately.
Now compare it with the following:

        SPL 0
        MOV 3, 3
        ADD #165, -1
        JMP -2

After few cycles are running, this small module have accumulated some
processes in its loop. A hit to any of its parts will not stop it
from running. Furthermore, a single hit to either the first or the
last instruction will still let the program remain operational.
```

There is a notable difference between the following routines:

```
                           SPL 0              SPL 0
        MOV 3, 3           MOV 3, 3           ADD #165, -1
        ADD #165, -1       ADD #165, -1       MOV 3, 3
        JMP -2             JMP -2             JMP -2

<hr>
```

- The difference is due to the way SPL works. For comparison:
- No self-splitting routine works as:
- MOV, ADD, MOV, ADD, MOV, ADD, MOV, ADD, MOV, ADD, MOV, ADD, ...
- Self-splitting routine 1 works as:
- MOV, ADD, MOV, ADD, MOV, *MOV*, ADD, MOV, ADD, MOV, ADD, MOV, ...
- Self-splitting routine 2 works as:
- ADD, MOV, ADD, MOV, ADD, *ADD*, MOV, ADD, MOV, ADD, MOV, ADD, ...

(back to Index)

Bombing/scanning pattern

Such kind of pattern is typically shown in: (*C* * *i*) mod *coresize*.
C is an integer number that is repetitively added to or subtracted from another number. If C is one, then the formed pattern will be similar to that of linear bombing or scanning.
i is the nth times of addition or subtraction.
coresize is the size of the arena.

The measurement of good constant number C is generally characterized by how swift the module employing C can fully break the arena down into smaller fragments whose size is equal or less than N. N here is the expected fragmentation size. The smaller the N, the slower the break down process. This is understandable as the smaller the N, the more the fragmentation is needed.

There is also a lower limit in which a fragment cannot be broken into smaller ones. Thus this limit number sets as the minimum fragment size and it depends on the chosen constant number. Modulo is associated with this limit number. A pattern of modulo 5 means that the arena cannot be broken into fragments whose size is smaller than 5. For every five cells in a row, four of them cannot be touched and left as gaps. Another term for modulo is the greatest common divisor or gcd between the constant C and the coresize.

- Below is the list of some constants that produce best pattern for any given modulo in coresize 8000:
- 1 3039 3359
- 2 2234 3094
- 4 3044 3364
- 5 2365 3315
- 8 2376 2936

There are C source code and DOS executable file that compute best constant for any given modulo.

(back to Index)

Self-mutate

This is a beauty in corewar programming. A simple few lines of code can have more than one function or behavior. Not all of them are readily apparent until the code undergoes self-mutation or hits itself on its own purpose during its course. This is cheap and highly effective. Twill, for example is designed as a very capable stone that changes its behavior four times during its full course.

(back to Index)

On-Line Processes

- On-Line processes are several processes that run at the same line everytime. They are very useful in many prototypes especially paper. The formula to create N number of these processes:
- Subtract N by 1.
- Encode (N-1) into its equivalent binary value.
- Starting from left to right, replace every 1 with "SPL 1" and every 0 with "MOV -1, 0".

Example:
For N = 10, its (N-1 or 9) equivalent binary value is: 1001. The sequence is:

```
        SPL 1     ;1
        MOV -1, 0 ;0
        MOV -1, 0 ;0
        SPL 1     ;1
```

```
(back to Index)
```

PREFACE

The following part of introduction describes about some paper warriors. It covers about: basic paper style, improving paper, winning with paper, and checksum design.

- Table lookup
- Basic paper style
- Checksum design

Basic paper style

A paper warrior is basically a replicator. Its ability to spread in the core and to disrupt opponent code is its main strength. By replicating and spreading, it assures itself a long endurance. The negative effect is it exposes and renders itself vulnerable to warriors that apply stun attack. Hence paper is a formidable opponent against a single process stone and at the same time is an easy target against scissors-type warriors.

A basic paper program might look like this:

```
;name Paper 1

  cnt EQU lst-src     ; number of code in paper

  src DAT #cnt        ; source pointer
  dst DAT #1222       ; destination pointer
  pap MOV #cnt, src   ; #cnt is number of lines to be copied
      MOV <src, <dst  ; copy a code...
      JMN -1, src     ; once at a time and
                      ; loop back until all lines copied
      SPL @dst        ; split the process to a new copy
      SUB #23, dst    ; give more distance to the next copy
      JMP pap         ; make other copies
  lst END pap
```

As early as the development of ’88 had gone, this kind of program was also known as mice program. They could fill-up the core faster than what early stones or dwarfs could cover.

Nowadays, there is little to be expected from it as far as efficiency is concerned. Notice that some of the codes have only one functional operand. By utilizing the unused operands, improvement on both speed and size can be gained.

```
;name Paper 2

  cnt EQU lst-src     ; number of code in paper

  src MOV #cnt, 0     ; source pointer
      MOV <src, <dst  ; copy the code...
      JMN -1, src     ; once at a time
  dst SPL @0, 1222    ; destination pointer
      SUB #23, dst    ; give more distance to next copy
      JMZ src, src    ; redo
  lst END src
```

The second paper doesn’t need DAT for its pointers. Its pointers are used in double usages with others. Instead of replicating 8 lines of code, it now replicates 6 lines. This means smaller module and faster progress.

Checksum

As paper modules replicate, their growth rate decreases proportionally. There is a side effect to this kind of event. Since paper warrior overides its adversary codes with its own replicating codes, there is a chance that its adversary becomes converted into another working paper instead of getting clobbered and terminated. For their best advantage, many paper warriors include in them checksum.

Like others, paper's goal is to terminate all of its adversary processes. Although it is lacking of what other warriors have: core-clear, it can win mainly just by overwriting the opponent code with its own paper code. An ideal paper module should provide dual functions: replicator and terminator. Checksum, as part of codes being copied, controls all the processes executing it to choose among the two functions. They are allowed to continue replicating if they can identify themselves as their own processes or forced terminated otherwise.

One way to design checksum is by observing how distinct own processes from opponent ones when running in a paper module. They are:

Initial location.
    The opponent process may start at anywhere in the copied module.
Number.
    There are more processes executing the module (own's + opponent's).

A checksum can be implemented effectively with only few additional codes. The first warrior that demonstrates checksum is note paper.

The concept is as follow:

```
;name Paper 3

  cnt  EQU dt - src

  init SPL 1
       MOV -1, 0
       SPL 1          ; Create 6 on-line processes

  src  MOV #cnt, 0    ; Init number of lines to be copied
                      ; This also serves as a source pointer
       MOV <src, <dst ; Copy a line 6 times (make one full copy)
  dst  SPL @0, #1222  ; Split 6 times
       MOV dt, <-1    ; Give more distance to next copy
       JMZ src, src   ; Test for checksum
       MOV 0, -1      ; Attempt to erase that module
  dt   END init
```

The new warrior requires initial set-up that create 6 processes that have to be executing on the same line. This paper module is equipped with checksum and self-erase routine. The self-erase routine is intended for all alien processes.

In order to make a full copy, there has to be at least 6 processes in any loop or module. The checksum checks for exactly 6 processes being present in that loop. If it is, the processes continue the copy routine. Otherwise, their progress are simply denied and forced to activate the self-erase routine.

Replicator Warriors

- Index
- Replicator framework
- Checksum design

Replicator framework

A replicator is also referred as paper. Its ability to spread in the core and to disrupt opponent code is its main strength. By replicating and spreading, it assures itself a long endurance. The negative effect is it exposes and renders itself vulnerable to warriors that apply stun attack. Hence replicator is a formidable opponent against a single process stone and at the same time is an easy target against scissors-type warriors.

Basically, a replicator might look like this:

```
        ;name Paper 1

         cnt EQU lst-src     ; number of code in paper

         src DAT #cnt        ; source pointer
         dst DAT #1222       ; destination pointer
         pap MOV #cnt, src   ; #cnt is number of lines to be copied
             MOV <src, <dst  ; copy a code...
         JMN -1, src     ; once at a time and
                         ; loop back until all lines copied
         SPL @dst        ; split the process to a new copy
         SUB #23, dst    ; give more distance to the next copy
         JMP pap         ; make other copies
     lst END pap</listing>
```

```
At earlier time, this kind of warrior was also known as mice warrior.
Against stones or dwarfs, they could easily
overrun them.
```

```
        ;name Paper 2

         cnt EQU lst-src     ; number of code in paper

         src MOV #cnt, 0     ; source pointer
             MOV <src, <dst  ; copy the code...
         JMN -1, src     ; once at a time
     dst SPL @0, 1222    ; destination pointer
         SUB #23, dst    ; give more distance to next copy
         JMZ src, src    ; redo
     lst END src</listing>
```

```
The second replicator doesn’t need DAT for its pointers. Its pointers are used
in double usages with others. Instead of replicating 8 lines of code, it now
replicates 6 lines. This means smaller module and faster progress.
```

Checksum

Like others, paper’s goal is to terminate all of its adversary processes. Although it is lacking of what other warriors have: core-clear, it still can win just by mainly overwriting the opponent code with its own code. One hindrance with this is that simple plain of copying is not sufficient for paper to win alone. There is a chance that its adversary becomes converted into another working paper instead of getting clobbered. Thus, an ideal paper module should be able to self-terminate as well as to replicate. Here is the importance of checksum. It allows paper to check all the executing processes and direct their continuation. Those who can identify themselves as original paper’s processes are allowed to continue replicating. Those who can’t are forced to terminated.

One way to design checksum is by observing how distinct own processes from opponent ones when running in a paper module. They are:

Initial location.
    The opponent process may start at anywhere in the copied module.
Number.
    There are more processes executing the module (own’s + opponent’s).

A checksum can be implemented effectively with only few additional codes. The first warrior that demonstrates checksum is note paper.

The concept is as follow:

```
        ;name Paper 3

         cnt  EQU dt - src

         init SPL 1
              MOV -1, 0
              SPL 1          ; Create 6 on-line processes

         src  MOV #cnt, 0    ; Init number of lines to be copied
                             ; This also serves as a source pointer
              MOV <src, <dst ; Copy a line 6 times (make one full copy)
     dst  SPL @0, #1222  ; Split 6 times
          MOV dt, <-1    ; Give more distance to next copy
          JMZ src, src   ; Test for checksum
          MOV 0, -1      ; Attempt to erase that module
     dt   END init</listing>
```

```
The new warrior requires an initial set-up that creates 6
online processes. Down at the bottom is
a neat single piece of code. It is intended for all alien processes.
The checksum is such as in order to replicate successfully, there have to be
exactly 6 processes running synchronously. Failing the requirement should
trigger the self-erase routine at the bottom.
```

PREFACE

This part of introduction describes what and how the scanner warriors work. The scope in this document covers only classical scanners. Classical scanners are scanners that are designed to specifically catch paper style warriors by throwing self-splitting instructions. This includes two scanner prototypes: B-scanner and CMP-scanner. They are detailed in two separate parts. Comparison between scanners can be found afterward. Further details on other types of scanners can be obtained from various collection of articles in ftp.CSUA.berkeley.edu under directory pub/corewar/redcode.

- Table lookup:
- Scanner Prototypes
  - B-scanner
    - Components:
    - B-scan phase
    - Stunning phase
    - Looping solution
    - End phase
    - Overall
  - CMP-scanner
    - Components:
    - CMP-scan phase
    - Stunning phase
    - Looping solution
    - End phase
    - Overall
- Comparison between B-scanners and CMP-scanners

Scanner prototypes

Scanner warriors are those that are configured to detect the presence of opponent before laying down their bombs on any suspicious locations. Aside from scanning, it is also important that the scanners are able to avoid messing up their own code.

There are two distinct prototypes for scanners. They are B-scanner and CMP-scanner. Their names were derived from their functions that do scanning. B-scanners detect their opponent by searching for any non-zero B-field in their code. CMP-scanners provide more rigid detection by comparing (CMP) for any non-identical instructions between two different locations. In the extent of their functional differences, both kinds of scanners avoid self-attack in interestingly different manner. (For further detail, see Comparison between B-scanners and CMP-scanners).

B-scanner

B-scanners assumes that at least one of their opponent code has non-zero value in their B-field.

One of B-scanners' duties might be as follow:

```
   ; ... B-scan
    ; ... throw self-splitting instructions
    ; ... redo before finish
    ; ... core-clear</listing>
```

<h3>B-scan</h3>

Its main instruction is: *JMZ scan, ptr* where scan refers to
the scanning instructions and ptr refers to the current scanning location.
The scanning instructions update the scanning pointer and test if it points
to a non-zero B-field. The instructions might be:

```
<listing>   scan ADD #const, ptr
         JMZ scan, ptr</listing>
```

During scanning phase, the scanner shouldn’t be mistaken with any of its
own codes. An easy way to do it is to add SLT after JMZ, e.g:

```
<listing>   scan ADD #const, ptr
         JMZ scan, ptr
         SLT #num, ptr ; num is number of codes...
                       ; ...in-between ptr and last line</listing>
```

<h3>Throw self-splitting instructions</h3>

These are the two instructions:

```
<listing>   MOV jmp_i, @ptr
    MOV spl_i, <ptr</listing>
```

spl_i refers to *SPL 0* and jmp_i refers to *JMP -1*.
<h3>Redo before finish</h3>

Against replicator warriors or other
warriors that execute more than one modules, it is neccessary to scan as
many locations as possible before core-clearing. A simple test to see whether
it has undergone a self-modification or
not is sufficient. This test could be a single instruction:

```
<listing>   JMN scan, scan</listing>
```

<h3>Core-clear</h3>

This is to clear away all the opponent stunned processes and to convert
tie into winning:

```
<listing>   SPL 0, 0
    MOV dat_i, <-1
    JMP -1, 0</listing>
```

<h3>Overall</h3>

Putting up together, here is the first version of B-scanner:
This version uses *SLT* to avoid self-attack.

```
<listing>   ; name B-scanner 1
    const   EQU 3094
    init    EQU scan
    scan    ADD #const, ptr
    ptr     JMZ scan, ptr+init
            SLT #dat_i, ptr
    throw   MOV jmp_i, @ptr
            MOV spl_i, <ptr    ; pointer is decremented by 1
            ADD #1, ptr        ; needed to readjust the pointer
    redo    JMN scan, scan
    spl_i   SPL 0, 0
            MOV dat_i, <-1
    jmp_i   JMP -1, 0
    dat_i   END scan</listing>
```

Here is a much more elegant solution to B-scanner, blatantly taken from a successful classical B-scanner:
B-scanner live in vain.

```
<listing>   ; name B-scanner 2
    const   EQU 2234
    init    EQU scan

    scan    ADD #const, @2
            JMZ scan, @ptr  ; hit here
    throw   MOV jmp_i, @ptr
    ptr     MOV spl_i, <init+ptr
    redo    JMN scan, scan
    spl_i   SPL 0, 0
            MOV dat_i, <-1
    jmp_i   JMP -1, 0
    dat_i   END scan</listing>
```

The *SLT* instruction has been dropped off but this program performs much better. Note that the warrior scans in *modulo 2* or one for every two instructions. Also note that the warrior structure is aligned such as the B-scanner will *scan zero B-field in its own code*. This is how it avoids winding up its own code. There is but one instruction: the second one that will be read as non-zero when it reads its own code. This instruction is the indicator for this warrior to begin its core-clear.
<h2>CMP-scanners</h2>

CMP-scanners detect the presence of opponent code by comparing (CMP) two instructions at different locations. One of ’88 rules is that at loading time, all instructions other than those of two warriors are initialized with *DAT $0, $0*. When CMP-scanner finds two non-identical instructions, it knows that it is not comparing two DAT $0, $0. At least one of these two instructions is either an opponent code or a modified code. In both cases, CMP-scanner simply throws in self-splitting instructions at the concerned locations. The tricky part is to find out which one of the two potentially belongs to the opponent. Like B-scanner, it should also avoid any unintentional self-modification.
CMP-scanners might as well fall into two smaller divisions. Their difference is in the way they handle two non-identical instructions. Their choices are based on their scanning gap. The CMP-scanner with large/medium scanning gap assumes the following: "if it is not the first instruction, then the second one is part of opponent’s". It then takes the next step (detailed below) to accomplish its duty. The other CMP-scanner (small scanning gap) assumes that they have touched the intersection of the opponent’s code. It then throws in self-splitting instructions at all locations between the two locations it is comparing.
Most CMP-scanners have the following components:

```
<listing>   ; ... CMP-scan
    ; ... handle everything to do upon two non-identical instructions.
    ; ... redo before finish
    ; ... core-clear</listing>
```

<h3>CMP-scan</h3>

The standard instructions for this component:

```
<listing>   update ADD loc_mod, scan
    scan   CMP loc, loc + gap
    avoid  SLT #num, scan
    rescan JMP update, 0</listing>
```

The first instruction updates both A-field and B-field of scanning location.

The second instruction does the scanning. The third instruction provides a
mechanism to prevent damaging its own codes. The last instruction loops
back to label update in the case of identical instructions. Most scanners
use the form *DJN update, <b-attack* as their looping
instruction.
<h3>Handle the next part after CMP-scan</h3>

One basic problem with CMP-scanners is that ’88 doesn’t have any A-field
indirect references. Since CMP-scanners use both A-field and B-field as
their scanning location, they should as well be able to inspect both pointed
locations and to take the neccessary actions based on both fields. Not until
then, their progress is incomplete.
<dl>
Some solutions to the above problem are:
<dt>Bomb in-between the two locations.

```
<dd><listing>       MOV #gap, cnt  ; the constant gap is known
        MOV spl_i, <scan
    cnt DJN -1, #cnt
        ADD #gap, scan ; re-adjust the B-field scan ptr</listing>
```

<dt>Bomb exactly at the two locations. (I)

```
<dd><listing>   MOV jmp_i, @scan   ; on B-field
    MOV spl_i, <scan
    SUB #gap-1, scan   ; now B-field scan has the same value...
                       ; ... as A-field scan
    MOV jmp_i, @scan   ; on A-field
    MOV spl_i, <scan
    ADD #gap+1, scan   ;resume to B-field scan</listing>
```

<dt>Bomb at first location and re-enter the scanning phase with B-field
now refers to A-field.

```
<dd><listing>   MOV jmp_i, @scan
    MOV spl_i, <scan
    ADD loc_mod2, scan</listing>
```

</dl>

Due to the lengthy codes, the second method is rarely used. The first
The first method is used by CMP-scanners based on
Agony type warrior.
The second method is rarely used due to its lengthy codes.
The last method is used by Crimp type
CMP-scanners.
The last method is intriguing to know. In normal scanning (instruction 1 - 4),
both location pointers are updated as from E-F to C-D to A-B ... (below).

```
<listing>   *   *   *   *   *   *
    A   B   C   D   E   F
</listing>
```

When it detects different instructions, e.g between E and F, it changes
its scanning pointers as from E-F to D-E. The purpose is to provide way
to access the A-Field.
<h3>Redo before finish</h3>

Like B-scanner, CMP-scanner intentionally bombs itself
to indicate that it has finished its scanning phase. A single instruction
does the trick:

```
<listing>   JMN update, update</listing>
```

<h3>Core-clear</h3>

A normal core-clear. The const value of *MOV const, <const* can
be used as loc_mod constant.

<h3>Overall</h3>

```
Putting up together, here are the two versions of CMP-scanners:
<listing>   ; name CMP-scanner (small)         ; name CMP-scanner (large)
    gap     EQU 12                     gap     EQU 49
    const   EQU -28                    const   EQU -98
    init    EQU update+const           init    EQU update+const2
                                       const2  EQU -49
    update  ADD loc_mod, scan          update  ADD loc_mod, scan
    scan    CMP init-gap, init         scan    CMP init-gap, init
            SLT #last-update, scan             SLT #last-update, scan
    rescan  DJN update, <6000          rescan  DJN update, <6000
            MOV spl_i, <scan                   MOV jmp_i, @scan
    cnt     DJN -1, #cnt                       MOV spl_i, <scan
            MOV #gap, cnt                      ADD mod_2, scan
            ADD #gap, scan             redo    JMN scan, scan
    redo    JMN update, update         spl_i   SPL 0
    spl_i   SPL 0                      mod_2   MOV const2, <const2+1
    loc_mod MOV const, <const          jmp_i   JMP -1
    last    END scan                   loc_mod DAT #const, #const
                                               END scan</listing>

<h1>Comparison between B-scanners and CMP-scanners

<dl>
<dt>Size
<dd>B-scanner is much smaller than CMP-scanner. The average B-scanner #lines
of codes is 8. The average CMP-scanner #lines of codes is 12.
<dt>Scanning speed
<dd>CMP-scanner is generally faster than B-scanner. CMP-scanner scans
two locations for every three instructions (67%) while B-scanner scans
one location for every two instructions (50%).
<dt>Coverage
<dd>A success B-scan can cover exactly half size of the core before entering
core-clear. A success CMP-scan can cover from half size to almost full size of
the core depending on the spread of its opponents.
<dt>Additional offense and defense
<dd>B-scanner: B-protection. CMP-scanner: DJN stream plus B-protection.
<dt>Wasting on decoys
<dd>B-scanner wastes less cycles than CMP-scanner does on decoys spreaded
by their opponents. Most CMP-scanners however avoid most decoys caused by
opponent’s DJN-stream.
<dt>Efficiency against stone
<dd>B-scanner performs better than CMP-scanner does.
<dt>Efficiency against paper
<dd>CMP-scanner performs better than B-scanner does.
</dl>

<address>Author: wangsawm@kira.csos.orst.edu</address>

</body>
```

Vampire Style (Earlier)

Vampire warriors, rather than throwing DAT bombs, they throw JMP pointers into the core. These pointers point to a placed trap. Once any enemy process steps on one of those pointers, it is immediately transfered to the trap and forced to do the slave works for vampire warriors.

Earlier incarnation of vampires worked like below:

```
   ;name Vampire 1

    const EQU 2365

    loc   MOV ptr, ptr     ; throw JMP pointer to core
          ADD #const, ptr  ; update pointer
          SUB #const, loc  ; update location
          JMP loc          ; loop back

    ptr   JMP @0, trap     ; the pointer weapon

    trap  SPL 1, -100      ; this is where the pointer points to
          MOV bomb, <-1    ; core-clear
          JMP trap
    bomb  DAT #0</listing>

This warrior throws pointers one for every 5 instructions (modulo 5).
The constant has been chosen to work well with
modulo 5. Note that the next pointer and the
next location are updated by the same constant but reversed in sign.
The trap here simply forces all processes in it to accumulate more processes
and to execute core-clear. The standard
self-splitting is not used here so that
the slavers can execute self-destruct once they finish core-clear.
<h1>Vampire Style (Modern)</h1>

Modern vampires use more effective procedure:
<listing>   ;name Vampire 2

    const EQU 2365

          SPL 0            ; self splitting
    vamp  MOV ptr, @ptr    ; throw pointer
          ADD data, ptr    ; update pointer
          DJN vamp, <2339  ; loop back + non-lethal attack

    ptr   JMP trap, ptr    ; pointer to...

    trap  SPL 1, -100      ; ...here
          MOV data, <-1
          JMP -2
    data  DAT #const, #-const</listing>

The only changes here are the main component (vampire) and the pointer
structure. Pointer is now updated at once. With this change, it permits
self-splitting mechanism for harder
shell (protection). Another improvement is that the vampire throws pointers
faster than before. The gained speed is approximately 30%.
Samples of vampire warriors are:
```

Sucker,
PitTrap,
Twilight Pits, and many others.
</body>