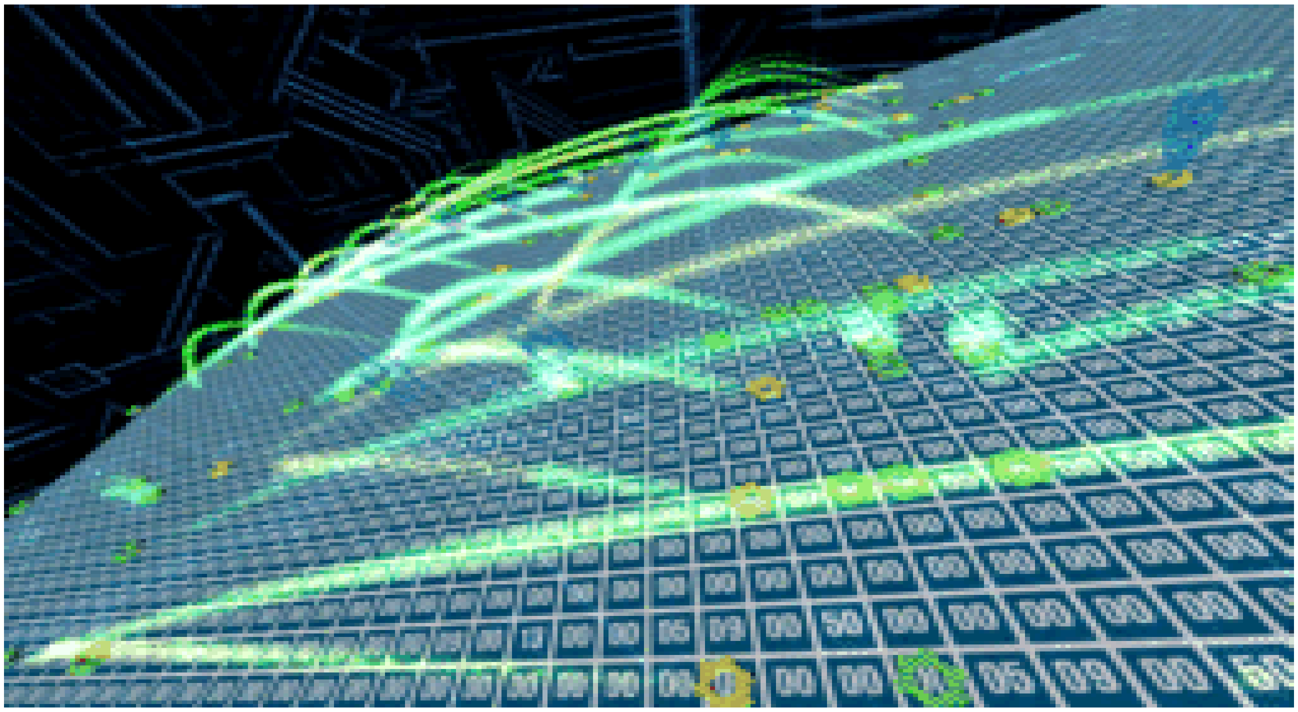
Core



War

By Walid Maalouli - November 2012

Based on the original concept by A. K. Dewdney

TI 99/4A Computer

Introduction

Core War is the brainchild of A. K. Dewdney who first described it in an issue of Scientific American in 1984.

“ Two computer programs in their native habitat – the memory chips of a digital computer – stalk each other from address to address. Sometimes they go scouting for the enemy; sometimes they lay down a barrage of numeric bombs; sometimes they copy themselves out of danger or stop to repair damage. This is the game I call Core War. It is unlike almost all other computer games in that people do not play at all! The contending programs are written by people, of course, but once a battle is under way the creator of a program can do nothing but watch helplessly as the product of hours spent in design and implementation either lives or dies on the screen. The outcome depends entirely on which program is hit first in a vulnerable area. “

There have been many enhancements made to this game over the decade following its introduction, primarily by adding novel instructions and modifiers, which have made it quite challenging for the non-programmer. This version is based on the original 1984 version which should be accessible to most anyone with an interest in this kind of game.

It has been programmed using TI FORTH.

I hope you will find it enjoyable.

Walid Maalouli

November 2012

Loading and Running Core War

The game will run either on a real TI 99/4A with 32K RAM, Editor/Assembler cartridge, and 2 disk drives, or using the following emulators:

- **Win994A**
- **V9t9**
- **MESS**
- **PC99**

While it can be played on the Classic99 emulator, you will not however be able to save your battle programs.

You will need to place the TI FORTH disk in drive #1, and the Corewars disk in drive #2.

Select LOAD AND RUN option from the Editor/Assembler menu and type DSK1.FORTH followed by a carriage return. TI FORTH will load and you will be returned to the command prompt.

Type 91 LOAD followed by a carriage return, and Core War will load. Please note that it will take a couple of minutes for the loading to be completed, and then the command prompt will come back.

To run Core War at this point, simply type COREWAR followed by a carriage return. But before doing so, I suggest you read the original Scientific American article in Appendix A at the end of the manual then get familiarized with the battle program editor in the next section.

Battle Program Editor

Core War utilizes the TI FORTH 64 integrated editor for programming the battle programs. It is a very capable full screen editor.

TI FORTH is based on so-called SCREENS, each containing up to 1024 characters. <u>Only SCREENS 119 to 179 are available for your use</u>, which should prove more than sufficient. The lower SCREENS contain the system and program files, and tampering with them will likely lead to either a system crash or program malfunction.

SCREEN 117 contains two sample battle programs, namely the IMP and the DWARF which you are free to study and modify at will. SCREEN 118 has a program called GEMINI, which is a more advanced battle robot.

To start the editor, type the SCREEN number you wish to edit, a space, then EDIT followed by a carriage return. For example:

119 EDIT

If that SCREEN appears to be full of garbage characters, it means that it has not been edited before. Press FCTN-9 to access the command prompt, type the SCREEN number, a space, then CLEAR followed by a carriage return. For example:

119 CLEAR

You can at that point return to the editor by typing ED@ followed by a carriage return at which point the SCREEN would have been cleared.

Editor commands:

- **Use the FCTN-S, FCTN-D, FCTN-E and FCTN-X (arrow) keys to move the cursor around the screen**
- **FCTN-1: delete character under cursor and shift line left**
- **FCTN-2: insert a space under the cursor and shift line right**

- **FCTN-3: delete entire line and scroll screen up. The deleted line is copied to the clipboard**
- **CTRL-8: insert a new line at the cursor position and shift the screen down**
- **FCTN-8: paste the content of the clipboard at the cursor location and scroll the screen down**
- **CTRL-V: tab forward by word**
- **FCTN-V: tab backwards by word**
- **FCTN-4: move to the next SCREEN**
- **FCTN-6: move to the previous SCREEN**
- **FCTN-5 scrolls the screen to the left**

Once done editing a SCREEN, press FCTN-9 to exit to the command prompt. Again, you can return to the editor by typing ED@.

Battle Program Instruction Set

Core War has 8 executable instructions and a single non-executable one. The battleground consists of a core memory with 832 addresses from 0 to 831. This memory is circular, meaning that address 832 becomes address 0. The battle programs can never know their absolute location within the core memory, and all the instructions refer only to memory locations <u>relative</u> to the current program's instruction pointer. For example, if an instruction refers to address 2, this means it is targeting an address two memory addresses ahead of its current instruction location.

This version of Core War works exactly the same as described in the original article although the program entry syntax differs a bit as noted below. <u>Program length cannot exceed 100 instructions</u>.

The battle programs need to be entered using the editor in a specific format so they can be properly interpreted by the game engine.

- **Every program needs to start with a : , a space, followed by the program name which cannot exceed 15 characters. For example, : TEST**

- **This should be followed by the word INIT to indicate the beginning of the battle program.**

- **ORG should be used immediately prior to the first executable opcode in your program. This is frequently the first opcode, but not necessarily.**

- **All instructions consist of 5 fields, each field separated by at least one space. The fields are:**
  - **Opcode**
  - **A field addressing mode**
  - **A field operand**
  - **B field addressing mode**
  - **B field operand**

Please note that all the fields should be present even when an opcode uses only one the fields.

- **The valid opcodes are:**
  - **MOV**
  - **ADD**
  - **SUB**
  - **JMP**
  - **JMZ**
  - **JMG**
  - **DJZ**
  - **CMP**
  - **DAT**

- **The addressing modes are:**
    - **.# immediate**
    - **.! direct**
    - **.@ indirect**

- **All programs should end with a ;**

- **<u>A space should separate every element of a program</u>**

- **Comments can be added by using a (followed by a space, the comment text, and a closing)**

- **If a program requires more than one SCREEN, place a - - > at the bottom of the SCREEN and continue program entry with the subsequent SCREEN. You may repeatedly do so as needed.**

- **Only capital letters are recognized by the interpreter.**

Program examples:

```
: IMP

INIT

ORG

MOV .! 0 .! 1 ;  ( COPIES ITS INSTRUCTION AHEAD ONE ADDRESS)
```

```
: DWARF

INIT

ORG

ADD .# 4 .! 3

MOV .! 2 .@ 2

JMP .! -2 .# 0  ( NOTE A .# 0 IS PLACED IN FIELD B BUT NOT USED)

DAT .# 0 .# 0 ;
```

IMPORTANT HINTS:

- **Remember that every instruction needs all 5 fields regardless of whether all the field are used by that instruction. It is good practice to place a .# 0 in the unused fields. Every field needs to be separated by at least a space**
- **Unused fields can be used as handy storage areas**
- **All instructions target the B Fields ONLY! This is a common source of error.**
- **All addresses are relative to the address of the current instruction being executed.**
- **With indirect addressing (.@), the contents of the target address are added to the current instruction pointer to create a new target address, which contents will in turn be added to the latter to come up with the final target address. This can be tricky, so make sure you pay particular attention to this fact.**
- **Do not forget to place an ORG just before the first instruction that will be executed when your battle program launches.**

Saving and Loading Battle Programs

Once you have finished typing in your program, press FCTN-9 to exit the editor and return to the command prompt. At this time it is best to save your program to disk by typing FLUSH followed by a carriage return. The battle program will be saved on the same disk as the Core War one and will essentially become part of the program.

But before you can test your program in battle, you need to compile it by typing the SCREEN number where it resides followed by LOAD. For example:

118 LOAD

If there are no syntax errors, the command prompt will come back with ok:0. Otherwise, a ? will be displayed to indicate that an error has occurred and that you should check your program.

Important: in order not to have multiple copies of your program in memory when you edit it and load it several times, it is good practice to do a FORGET followed by your program name before loading the program SCREEN again. For example:

FORGET TEST

Hint: If you wish for your battle programs to load when the Core War program is loaded, simply chain link their associated screens with - - > then execute a FLUSH operation at the command line. Core War is already linked to SCREEN 118 by default.

Running a Battle

Once you have a completed and loaded a battle program, type at the command prompt COREWAR followed by a carriage return. A splash screen will be displayed, and you can press any key to move on.

You will then be prompted for the names of the two battle programs that will be competing against each other. Type in the appropriate names followed by a carriage return, and then the program will initialize the core memory prior to the battle and will prompt you to press a key when it's ready.

At that point, the locations of the battle programs will be displayed in the core memory. Each instruction in the battle programs occupies a single core memory location, and is color coded as follows:

- **MOV : Light Green**
- **ADD : Dark Blue**
- **SUB : Light Blue**
- **JMP : Light Red**
- **JMZ : Medium Red**
- **JMG : Dark Red**
- **DJZ : Dark Yellow**
- **CMP : Light Yellow**
- **DAT : White**
- **Current instruction for Battle Program #1 : Magenta**
- **Current instruction for Battle Program #2 : Cyan**

Once you are ready to start the battle, press a key, and you will note that the corresponding instruction pointer for each battle program will move from instruction to instruction per the program's design. Any changes to the core memory will be displayed as well. You will therefore be able to watch the execution of the programs and their effect on core memory in real time.

The battle program that goes first is chosen randomly at the beginning of the battle, and will subsequently alternate between the two battle programs. A battle cycle consists of one instruction execution by each of the battle programs.

Battle Outcome

If one of the battle programs attempts to execute a DAT statement, it will automatically lose the battle. If at the end of 1000 cycles there is no winner, then the battle is considered a draw.

Obviously, your program's objective is to try to corrupt your opponent's program and get it to execute a DAT statement. Easier said than done...

Good Luck!

Appendix A

COMPUTER RECREATIONS

In the game called Core War hostile programs engage in a battle of bits.

by A. K. Dewdney

Two computer programs in their native habitat -- the memory chips of a digital computer -- stalk each other from address to address. Sometimes they go scouting for the enemy; sometimes they lay down a barrage of numeric bombs; sometimes they copy themselves out of danger or stop to repair damage. This is the game I call Core War. It is unlike almost all other computer games in that people do not play at all! The contending programs are written by people, of course, but once a battle is under way the creator of a program can do nothing but watch helplessly as the product of hours spent in design and implementation either lives or dies on the screen. The outcome depends entirely on which program is hit first in a vulnerable area.

The term Core War originates in an outmoded memory technology. In the 1950's and 1960's the memory system of a computer was built out of thousands of ferromagnetic cores, or rings, strung on a meshwork of fine wires. Each core could retain the value of one bit, or binary digit, the fundamental unit of information. Nowadays memory elements are fabricated on semiconductor chips, but the active part of the memory system where a program is kept while it is being executed, is still often referred to as core memory, or simply core.

Battle programs in Core War are written in a specialized language I have named Redcode, closely related to the class of programming languages called assembly languages. Most computer programs today are written in a high-level language such as Pascal, Fortran or BASIC; in these languages a single statement can specify an entire sequence of machine instructions. Moreover, the statements are easy for the programmer to read and to understand. For a program to be executed, however, it must first be translated into "machine language," where each instruction is represented by a long string of binary digits. Writing a program in this form is tedious at best.

Assembly languages occupy an intermediate position between high-level languages and machine code. In an assembly-language program each statement generally corresponds to a single instruction and hence to a particular string of binary digits. Rather than writing the binary numbers, however, the programmer represents them by short words or abbreviations called mnemonics (because they are easier to remember than numbers). The translation into machine code is done by a program called an assembler.

Comparatively little programming is done in assembly languages because the resulting programs are longer and harder to understand or modify than their high-level counterparts. There are some tasks, however, for which an assembly language is ideal. When a program must occupy as little space as possible or be made to run as fast as possible, it is generally written in assembly language. Furthermore, some things can be done in an assembly language that are all but impossible in a high-level language. For example, an assembly-language program can be made to modify its own instructions or to move itself to a new position in memory.

Core War was inspired by a story I heard some years ago about a mischievous programmer at a large corporate research laboratory I shall designate X. The programmer wrote an assembly-language program called Creeper that would duplicate itself every time it was run. It could also spread from one computer to another in the network of the X corporation. The program had no function other than to perpetuate itself. Before long there were so many copies of Creeper that more useful programs and data were being crowded out. The growing infestation was not brought under control until someone thought of fighting fire with fire. A second self-duplicating program called Reaper was written. Its purpose was to destroy copies of Creeper until it could find no more and then to destroy itself. Reaper did its job, and things were soon back to normal at the X lab.

In spite of fairly obvious holes in the story, I believed it, perhaps because I wanted to. It took some time to track down the real events that lay behind this item of folklore. (I shall give an account of them below.) For now it is sufficient to note that my desire to believe rested squarely on the intriguing idea of two programs doing battle in the dark and noiseless corridors of core.

Last year I decided that even if the story turned out not to be true, something like it could be made to happen. I set up an initial version of Core War and, assisted by David Jones, a student in my department at the University of Western Ontario, got it working. Since then we have developed the game to a fairly interesting level.

Core War has four main components: a memory array of 8,000 addresses, the assembly language Redcode, an executive program called MARS (an acronym for Memory Array Redcode Simulator) and the set of contending battle programs. Two battle programs are entered into the memory array at randomly chosen positions; neither program knows where the other one is. MARS executes the programs in a simple version of time-sharing, a technique for allocation the resources of a computer among numerous users. The two programs take turns: a single instruction of the first program is executed, then a single instruction of the second, and so on.

What a battle program does during the execution cycles allotted to it is entirely up to the programmer. The aim, of course, is to destroy the other program by ruining its instructions. A defensive strategy is also possible: a program might undertake to repair any damage it has received or to move out of the way when it comes under attack. The battle ends when MARS comes to an instruction in one of the programs that cannot be executed. The program with the faulty instruction -- which presumably is a casualty of war -- is declared the loser.

Much can be learned about a battle program merely by analyzing its actions mentally or with pencil and paper. To put the program to the test of experience, however, one needs access to a computer and a version of MARS. The programs could be made to operate on a personal computer, and Jones and I have prepared brief guidelines for those who would like to set up a Core War battlefield of their own. (For a copy of the guidelines send your name and address and $2 for postage and handling to Core War, Scientific American, 415 Madison Avenue, New York, N.Y., 10017. Delivery may take a few weeks.)

Before describing Redcode and introducing some simple battle programs, I should say more about the memory array. Although I have noted that it consists of 8,000 addresses, there is nothing magical about this number; a smaller array would work quite well. The memory array differs from most computer memories in its circular configuration; it is a sequence of addresses numbered from 0 to 7999 but it thereupon rejoins itself, so that address 8000 is equivalent to address 0. MARS always reduces an address greater than 7999 by taking the remainder after division by 8000. Thus if a battle program orders a hit at address 9378, MARS interprets the address as 1378.

Redcode is a simplified, special-purpose assembly-style language. It has instructions to move the contents of one address in memory to another address, to alter the contents arithmetically and to transfer control forward or backward within a program. Whereas the output of a real assembler consists of binary codes, the mnemonic form of a Redcode instruction is translated by MARS into a large decimal integer, which is then stored in the memory array; each address in the array can hold one such integer. It is also MARS that interprets the integers as instructions and carries out the indicated operations.

A list of the elementary Redcode instructions is given in the top illustration on page 19. With each instruction the programmer is required to supply at least one argument, or value, and most of the instructions take two arguments. For example, in the instruction JMP -7 the mnemonic JMP (for "jump") is followed by the single argument -7. The instruction tells MARS to transfer control to the memory address seven places before the current one, that is, seven places before the JMP -7 instruction itself. If the instruction happened to be at address 3715, execution of the program would jump back to address 3708.

This method of calculating a position in memory is called relative addressing, and it is the only method employed in Redcode. There is no way for a battle program to know its own absolute position in the memory array.

The instruction MOV 3 100 tells MARS to go forward three addresses, copy what it finds there and deliver it 100 addresses beyond the MOV instruction, overwriting whatever was there. The arguments in this instruction are given in "direct" mode, meaning they are to be interpreted as addresses to be acted on directly. Two other modes are allowed. Preceding an argument with an @ sign makes it "indirect." In the instruction MOV @3 100 the integer to be delivered to relative address 100 is not the one found at relative address 3 but rather the one found at the address specified by the contents of relative address 3. (The bottom illustration on page 19 gives more detail on the process of indirect addressing.) A # sign makes an argument "immediate," so that it is treated not as an address but as an integer. The instruction MOV #3 100 causes the integer 3 to be moved to relative address 100.

Most of the other instructions need no further explanation, but the data statement (DAT) requires some comment. It can serve as a work space to hold information a program may need to refer to. Strictly speaking, however, it is not an instruction; indeed, any memory location with a zero in its first decimal position can be regarded as a DAT statement and as such is not executable. If MARS should be asked to execute such an "instruction," it will not be able to and will declare that program the loser.

The decimal integer that encodes a Redcode instruction has several fields, or functional areas [see middle illustration on page 19]. The first digit represents the mnemonic itself, and two more digits identify the addressing mode (direct, indirect or immediate). In addition four digits are set aside for each argument. Negative arguments are stored in complement form: -1 would be represented as 7999, since in the circular memory array adding 7999 has the same effect as subtacting 1.

The instructions making up a simple battle program called Dwarf are listed in the illustration on page 20. Dwarf is a very stupid but very dangerous program that works its way through the memory array bombarding every fifth address with a zero. Zero is the integer signifying a nonexecutable data statement, and so a zero dropped into an enemy program can bring it to a halt.

Assume that Dwarf occupies absolute addresses 1 through 4. Address 1 initially contains DAT -1, but execution begins with the next instruction. ADD #5 -1. The effect of the instruction is to add 5 to the contents of the preceding address, namely the DAT -1 statement, thereby transforming it into DAT 4. Next Dwarf executes the instruction at absolute address 3, MOV #0 @-2. Here the integer to be moved is 0, specified as an immediate value. The target address is calculated indirectly in the following way. First MARS counts back two addresses from address 3, arriving at address 1. It then examines the data value there, namely 4, and interprets it as an address relative to the current position; in other words, it counts four places forward from address 1 and hence deposits a 0 at address 5.

The final instruction in Dwarf, JMP -2, creates an endless loop. It directs execution back to absolute address 2, which again increments the DAT statement by 5, making its new value DAT 9. In the next execution cycle a 0 is therefore delivered to absolute address 10. Subsequent 0 bombs will fall on addresses 15, 20, 25 and so on. The program itself is immobile but its artillery threatens the entire array. Eventually Dwarf works its way around to addresses 7990, 7995 and then 8000. As far as MARS is concerned, 8000 is equal to 0, and so Dwarf has narrowly avoided committing suicide. Its next missile again lands on address 5.

It is sobering to realize that no stationary battle program that has more than four instructions can avoid taking a hit from Dwarf. The opposing program has only three options: to move about and thereby elude the bombardment, to absorb hits and repair the damage or to get Dwarf first. To succeed through the last strategy the program may have to be lucky: it can have no idea where Dwarf is in the memory array, and on the average it has about 1,600 execution cycles before a hit is received. If the second program is also a Dwarf, each program wins 30 percent of the time; in 40 percent of the contests neither program scores a fatal hit.

Before taking up the other two strategies, I should like to introduce a curious one-line battle program we call Imp. Here it is:

```
MOV 0 1
```

Imp is the simplest example of a Redcode program that is able to relocate itself in the memory array. It copies the contents of relative address 0 (namely MOV 0 1) to relative address 1, the next address. As the program is executed it moves through the array at a speed of one address per cycle, leaving behind a trail of MOV 0 1 instructions.

What happens if we pit Imp against Dwarf? The barrage of zeros laid down by Dwarf moves through the memory array faster than Imp moves, but it does not necessarily follow that Dwarf has the advantage. The question is: Will Dwarf hit Imp even if the barrage does catch up?

If Imp reaches Dwarf first, Imp will in all probability plow right through Dwarf's code. When Dwarf's JMP -2 instruction transfers execution back two steps, the instruction found there will be Imp's MOV 0 1. As a result Dwarf will be subverted and become a second Imp endlessly chasing the first one around the array. Under the rules of Core War the battle is a draw. (Note that this is the outcome to be expected "in all probability." Readers are invited to analyze other possibilities and perhaps discover the bizarre result of one of them.)

Both Imp and Dwarf represent a class of programs that can be characterized as small and aggressive but not intelligent. At the next level are programs that are larger and somewhat less aggressive but smart enough to deal with programs in the lower class. The smarter programs have the ability to dodge an attack by copying themselves out of trouble. Each such program includes a segment of code somewhat like the one named Gemini, shown in the lower illustration on page 22. Gemini is not intended to be a complete battle program. Its only function is to make a copy of itself 100 addresses beyond its present position and then transfer execution to the new copy.

The Gemini program has three main parts. Two data statements at the beginning serve as pointers; they indicate the next instruction to be copied and its destination. A loop in the middle of the program does the actual copying, moving each instruction in turn to an address 100 places beyond its current position. On each transit through the loop both pointers are incremented by 1, thereby designating a new source and destination address. A compare instruction (CMP) within the loop tests the value of the first data statement; when it has been incremented nine times, the entire program has been copied, and so an exit from the loop is taken. One final adjustment remains to be made. The destination address is the second statement in the program and it has an initial value of DAT 99; by the time it is copied, however, it has already been incremented once, so that in the new version of the program it reads DAT 100. This transcription error is corrected (by the instruction MOV #99 93) and then execution is transferred to the new copy.

By modifying Gemini it is possible to create an entire class of battle programs. One of these, Juggernaut, copies itself 10 locations ahead instead of 100. Like Imp, it tries to roll through all its opposition. It wins far more often than Imp, however, and leads to fewer draws, because an overwritten program is less likely to be able to execute fragments of Juggernaut's code. Bigfoot, another program employing the Gemini mechanism, makes the interval between copies a large prime number. Bigfoot is hard to catch and has the same devastating effect on enemy code as Juggernaut does.

Neither Bigfoot nor Juggernaut is very intelligent. So far we have written only two battle programs that qualify for the second level of sophistication. They are too long to reproduce here. One of them, which we call Raidar, maintains two "pickets" surrounding the program itself [see illustration on page 18]. Each picket consists of 100 consecutive addresses filled with 1's and is separated from the program by a buffer zone of 100 empty addresses. Raidar divides its time between systematically attacking distant areas of the memory array and checking its picket addresses. If one of the pickets is found to be altered, Raidar interprets the change as evidence of an attack by Dwarf, Imp or some other unintelligent program. Raidar than copies itself to the other side of the damaged picket, restores it, constructs a new picket on its unprotected side and resumes normal operation.

In addition to copying itself a battle program can be given the ability to repair itself. Jones has written a self-repairing program that can survive some attacks, although not all of them. Called Scanner, it maintains two copies of itself but ordinarily executes only one of them. The copy that is currently running periodically scans the other copy to see if any of its instructions have been altered by an attack. Changes are detected by comparing the two copies, always assuming that the executing copy is correct. If any bad instructions are found, they are replaced and control is transferred to the other copy, which then begins to scan the first one.

So far Scanner remains a purely defensive program. It is able to survive attacks by Dwarf, Imp, Juggernaut and similar slow-moving aggressors -- at least if the attack comes from the right direction. Jones is currently working on a self-repair program that keeps three copies of itself.

I am curious to see whether readers can design other kinds of self-repairing programs. For example, one might think about maintaing two or more copies of a program even though only one copy is ever executed. The program might include a repair section that would refer to an extra copy when restoring damaged instructions. The repair section could even repair itself, but it might still be vulnerable to damage at some positions. One measure of vulnerability assumes that a single instruction has been hit; on the average, how many such instruction, if they are hit, ultimately cause the program to die? By this measure, what is the least vulnerable self-repairing program that can be written?

Only if reasonably robust programs can be developed will Core War reach the level of an exciting game, where the emphasis is shifted from defense to offense. Battle programs will then have to seek out and identify enemy code and mount an intensive attack wherever it is found.

I may have given the impression that Redcode and the entire MARS system are fixed. They are not. In spare moments we have been experimenting with new ideas and are certainly open to suggestions. Indeed, we have been experimenting so much with new programs and new features that some battles remain to be fought in our own system.

One idea we have been playing with is to include an extra instruction that would make self-repair or self-protection a little easier. The instruction PCT A would protect the instruction at address A from alteration until it is next executed. How much could the vulnerability of a program be reduced by exploiting an instruction of this kind?

In the guidelines offered above we describe not only the rules of Core War but also how to set up a memory array and write a MARS system in various high-level languages. We also suggest how to display the results of a Core War battle. For now the following rules define the game with enough precision to enable pencil-and-paper players to begin designing programs:

1. The two battle programs are loaded into the memory array at random positions but initally are no closer than 1,000 addresses.

2. MARS alternates in executing one instruction from each program until it reaches an instruction that cannot be executed. The program with the erroneous instruction loses.

3. Programs can be attacked with any available weapon. A "bomb" can be a 0 or any other integer, including a valid Redcode instruction.

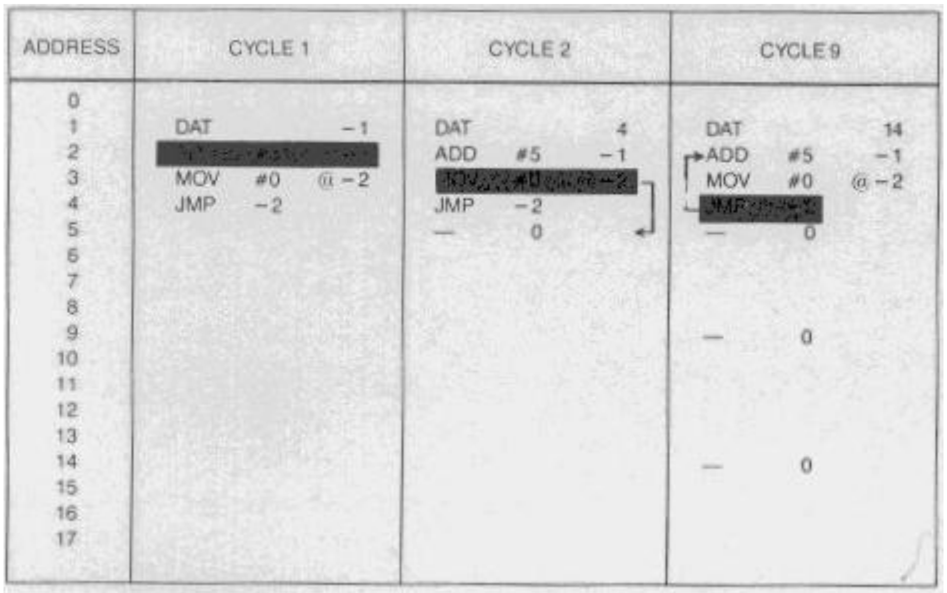
4. A time limit is put on each contest, determined by the speed of the computer. If the limit is reached and both programs are still running, the contest is a draw.

The story of Creeper and Reaper seems to be based on a compounding of two actual programs. One program was a computer game called Darwin, invented by M. Douglas McIlroy of AT&T Bell Laboratories. The other was called Worm and was written by John F. Shoch of the Xerox Palo Alto Research Center. Both programs are some years old, allowing ample time for rumors to blossom. (Darwin was described in Software: Practice and Experience, Volume 2, pages 93-96, 1972. A vague description of what appears to be the same game is also given in the 1978 edition of Computer Lib.)

In Darwin each player submits a number of assembly-language programs called organisms, which inhabit core memory along with the organisms of other players. The organisms created by one player (and thus belonging to the same "species") attempt to kill those of other species and occupy their space. The winner of the game is the player whose organisms are most abundant when time is called. McIlroy invented an unkillable organism, although it won only "a few games." It was immortal but apparently not very aggressive.

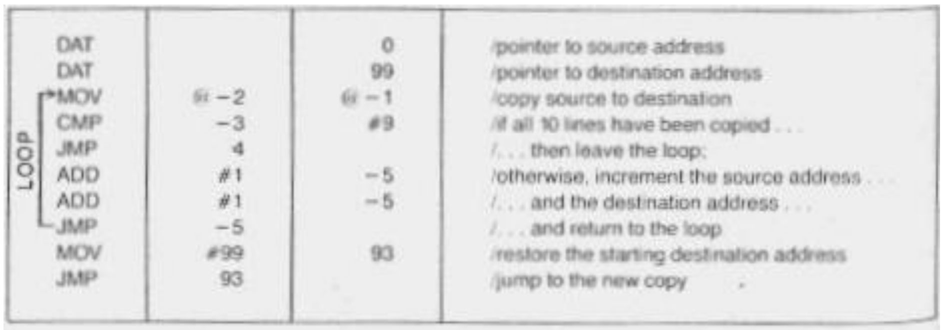
Worm was an experimental program designed to make the fullest use possible of minicomputers linked in a network at Xerox. Worm was loaded into quiescent machines by a supervisory program. Its purpose was to take control of the machine and, in coordination with Worms inhabiting other quiescent machines, run large applications programs in the resulting multiprocessor system. Worm was designed so that anyone who wanted to use one of the occupied machines could readily reclaim it without interfering with the larger job.

One can see elements of both Darwin and Worm in the story of Creeper and Reaper. In Core War, Reaper has becom reality.



ADDRESS	CYCLE 1	CYCLE 2	CYCLE 9
0			
1	DAT -1	DAT 4	DAT 14
2	ADD #5 -1	ADD #5 -1	ADD #5 -1
3	MOV #0 @-2	MOV #0 @-2	MOV #0 @-2
4	JMP -2	JMP -2	JMP -2
5		— 0	— 0
6			
7			
8			
9			— 0
10			
11			
12			
13			
14			— 0
15			
16			
17			

Dwarf, a battle program, lays down a barrage of "zero bombs"



DAT		0	/pointer to source address
DAT		99	/pointer to destination address
MOV	@-2	@-1	/copy source to destination
CMP	-3	#9	/if all 10 lines have been copied . . .
JMP	4		/. . . then leave the loop;
ADD	#1	-5	/otherwise, increment the source address . . .
ADD	#1	-5	/. . . and the destination address . . .
JMP	-5		/. . . and return to the loop
MOV	#99	93	/restore the starting destination address
JMP	93		/jump to the new copy

Gemini, a program that copies itself to a new position in the memory array

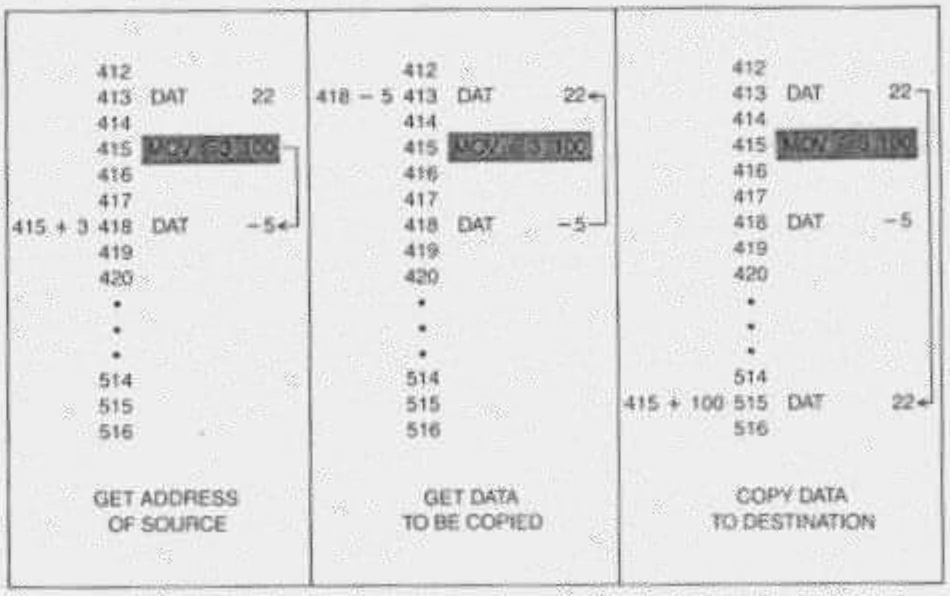
INSTRUCTION	MNEMONIC	CODE	ARGUMENTS		EXPLANATION
Move	MOV	1	A	B	Move contents of address *A* to address *B*.
Add	ADD	2	A	B	Add contents of address *A* to address *B*.
Subtract	SUB	3	A	B	Subtract contents of address *A* from address *B*.
Jump	JMP	4	A		Transfer control to address *A*.
Jump if zero	JMZ	5	A	B	Transfer control to address *A* if contents of address *B* are zero.
Jump if greater	JMG	6	A	B	Transfer control to address *A* if contents of *B* are greater than zero.
Decrement: jump if zero	DJZ	7	A	B	Subtract 1 from contents of address *B* and transfer control to address *A* if contents of address *B* are then zero.
Compare	CMP	8	A	B	Compare contents of addresses *A* and *B*; if they are unequal, skip the next instruction.
Data statement	DAT	0		B	A nonexecutable statement: *B* is the data value.

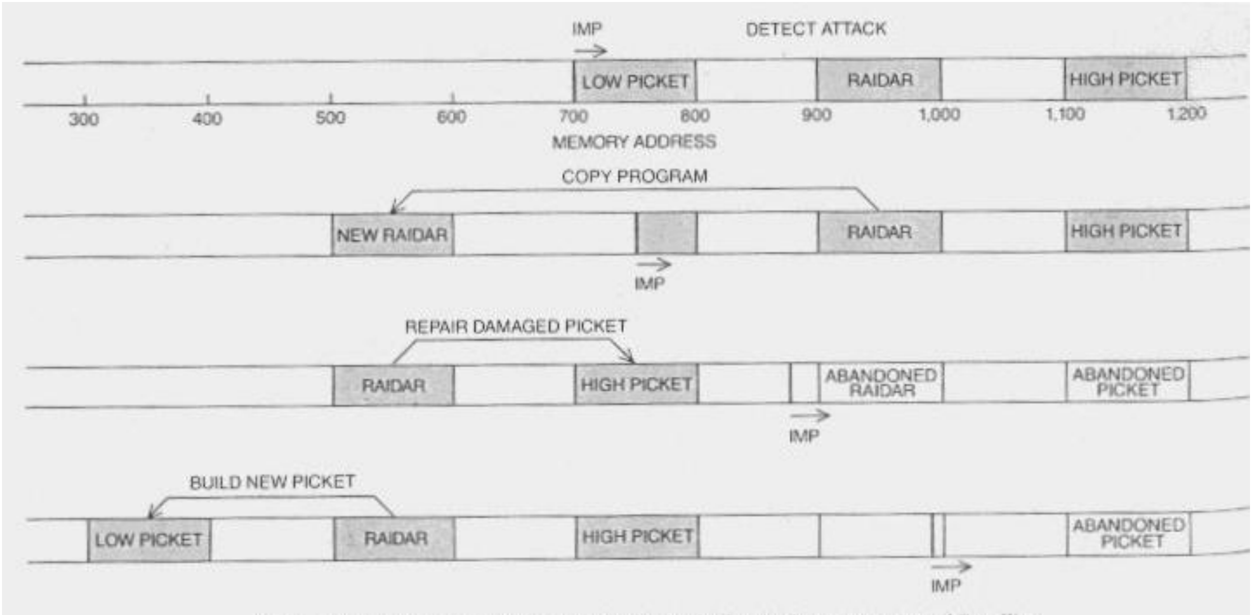
The instruction set of Redcode, an assembly language for Core War

MNEMONIC	ARGUMENT A	ARGUMENT B	OPERATION CODE	MODE DIGIT ARGUMENT A	MODE DIGIT ARGUMENT B	ARGUMENT A	ARGUMENT B
DAT		−1	0	0	0	0000	7999
ADD	#5	−1	2	0	1	0005	7999
MOV	#0	@ −2	1	0	2	0000	7998
JMP	−2		4	1	0	7998	0000

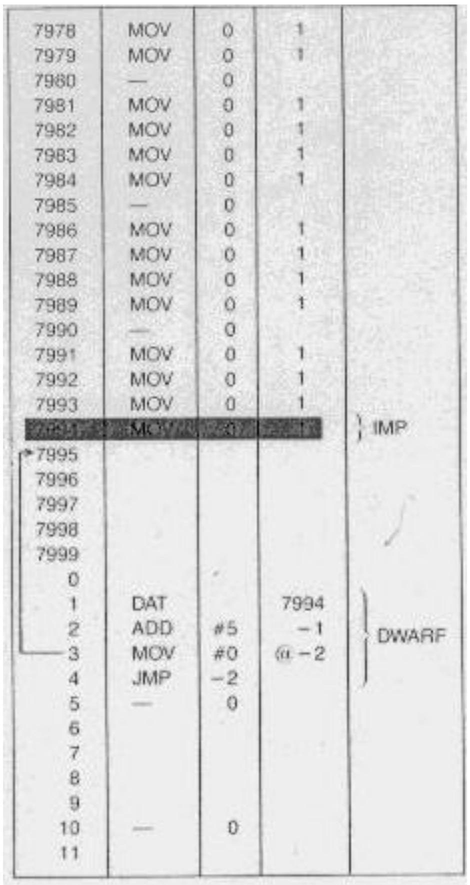
ADDRESSING MODES:			
	IMMEDIATE	#	0
	DIRECT		1
	INDIRECT	@	2

The encoding of Redcode instructions as decimal integers





Raidar, a sophisticated battle program, eludes the simpler Imp in the memory array of Core War



7978	MOV	0	1	
7979	MOV	0	1	
7980	—	0		
7981	MOV	0	1	
7982	MOV	0	1	
7983	MOV	0	1	
7984	MOV	0	1	
7985	—	0		
7986	MOV	0	1	
7987	MOV	0	1	
7988	MOV	0	1	
7989	MOV	0	1	
7990	—	0		
7991	MOV	0	1	
7992	MOV	0	1	
7993	MOV	0	1	
7994	MOV	0	1	} IMP
7995				
7996				
7997				
7998				
7999				
0				
1	DAT		7994	DWARF
2	ADD	#5	−1	
3	MOV	#0	@−2	
4	JMP	−2		
5	—	0		
6				
7				
8				
9				
10	—	0		
11				

Imp v. Dwarf: Who wins?