Januarie 1987 uitgawe van "Scientific American"

Moet asseblief nie verwyder uit lêer nie

COMPUTER RECREATIONS

A program called MICE nibbles its way to victory at the first Core War tournament

by A. K. Dewdney

Core War—the game in which specialized programs do their level best to destroy one another—was in the spotlight late last year at the first international Core War tournament held at the Computer Museum in Boston, Mass. Of 31 programs entered, three emerged as most robust. The ultimate victor was a program called MICE. Its author, Chip Wendell of Rochester, N.Y., received a handsome trophy that incorporated a core-memory board from an early CDC 6600 computer.

Core War has already appeared twice in this department in recent years [see "Computer Recreations," May, 1984, and March, 1985]. Written by human beings, the Core War programs are on their own as they spar in the arena of a computer's memory. The section of memory reserved for the struggle is called the core, after an obsolete form of memory constructed from miniature ferromagnetic rings known as core elements. The game has generated so much enthusiasm that it has sparked the formation of the International Core Wars Society. The game was recently modified by the society; the new version lays out the format that players will follow for now.

The basis of Core War—and the ammunition of the recent tournament—is a battle program written in a special, low-level language called Redcode. A set of 10 simple instructions enables a program to move information from one memory location to another, to add and subtract information, to alter the order in which its instructions are executed and even to have several instructions executing simultaneously [*see illustration on page 10*]. One basic instruction, for example, is the move command MOV. It consists of three parts—an instruction code and two addresses—that all occupy the same location in the core. The command is most generally written as MOV *A B*. If *A* happens to be 102 and *B* is −5, the computer will go forward 102 addresses and copy what it finds there into the location five addresses behind the MOV instruction.

The simplest Redcode program consists of just one MOV instruction: MOV 0 1. The program, which is called IMP, causes the contents at relative address 0 (namely the MOV instruction itself) to be transferred to relative address 1, just one address ahead of itself. Redcode instructions are normally executed consecutively. This means that after the MOV 0 1 instruction is executed the computer will try to execute an instruction at the next address. There is, of course, now an instruction occupying that address: the MOV 0 1 command just copied there. As a consequence IMP patters from address to address through the core, mindlessly destructive. It leaves a trail of MOV 0 1 instructions behind it.

An IMP can even steal an enemy program's very soul, its execution. To see how this can happen, imagine that a battle program is being executed in the usual manner, in the order of its instructions. An IMP enters the program from the top, overwriting the code with an endless sequence of MOV 0 1 instructions. Sooner or later the subverted program will probably transfer execution back to the overrun section. At such a point the program becomes a new IMP. It flies the same flag but is now doomed to follow in the tracks of the enemy IMP until the battle is over.

To avoid being overrun a Core War program must at the very least contain an IMP-STOMPER. The safeguard consists of two instructions executed cyclically:

```
MOV #0 −1
JMP −1
```

The first command moves the integer 0, symbolized by #0, to the relative address −1; in other words, every time the MOV command is executed the location just above it (the only direction from which IMP's can attack) is filled with a 0. The second instruction is the JMP command. When it is executed, it transfers the stream of execution, or flow of control, to the address at relative location −1, namely the address just above the JMP. Each execution cycle of the program causes a 0 to be slammed down on any IMP that may have arrived just above the IMP-STOMPER. Consequently the IMP is erased.

There are two basic rules in Core War. The first rule is that the two competing programs must take turns executing their instructions. The alternation is governed by MARS, the Memory Array Redcode Simulator. As the somewhat strained military mnemonic suggests, MARS simulates the action of a computer. It continually updates the contents of the core array in accordance with the instructions being executed. In doing so, it allows just one instruction per side to be executed per turn. The second rule is that if a program ceases to run, it loses.

As a program runs, it can have more than one stream of execution. If execution encounters the command SPL *A* in a Redcode program, it splits into two streams. One stream goes to the instruction that immediately follows SPL *A* and the other jumps to the instruction at relative address *A*. Unfortunately the MARS system cannot execute both instructions simultaneously; it executes one of the instructions on the next turn and the other instruction on the turn after that. What might be thought an incredible advantage is somewhat adumbrated; the more concurrent streams of execution a program has, the slower each stream proceeds. This is only fair, however. In the case of multiple streams of execution a battle program is declared the winner when all its opponent's streams have died out. At such a point MARS, which would still expect to find an executable instruction, can find only the computational equivalent of shell holes and bomb craters.

To illustrate the SPL command, here are the first five instructions of my own entry in the Core War tournament. It is called COMMANDO for reasons that will soon become clear.

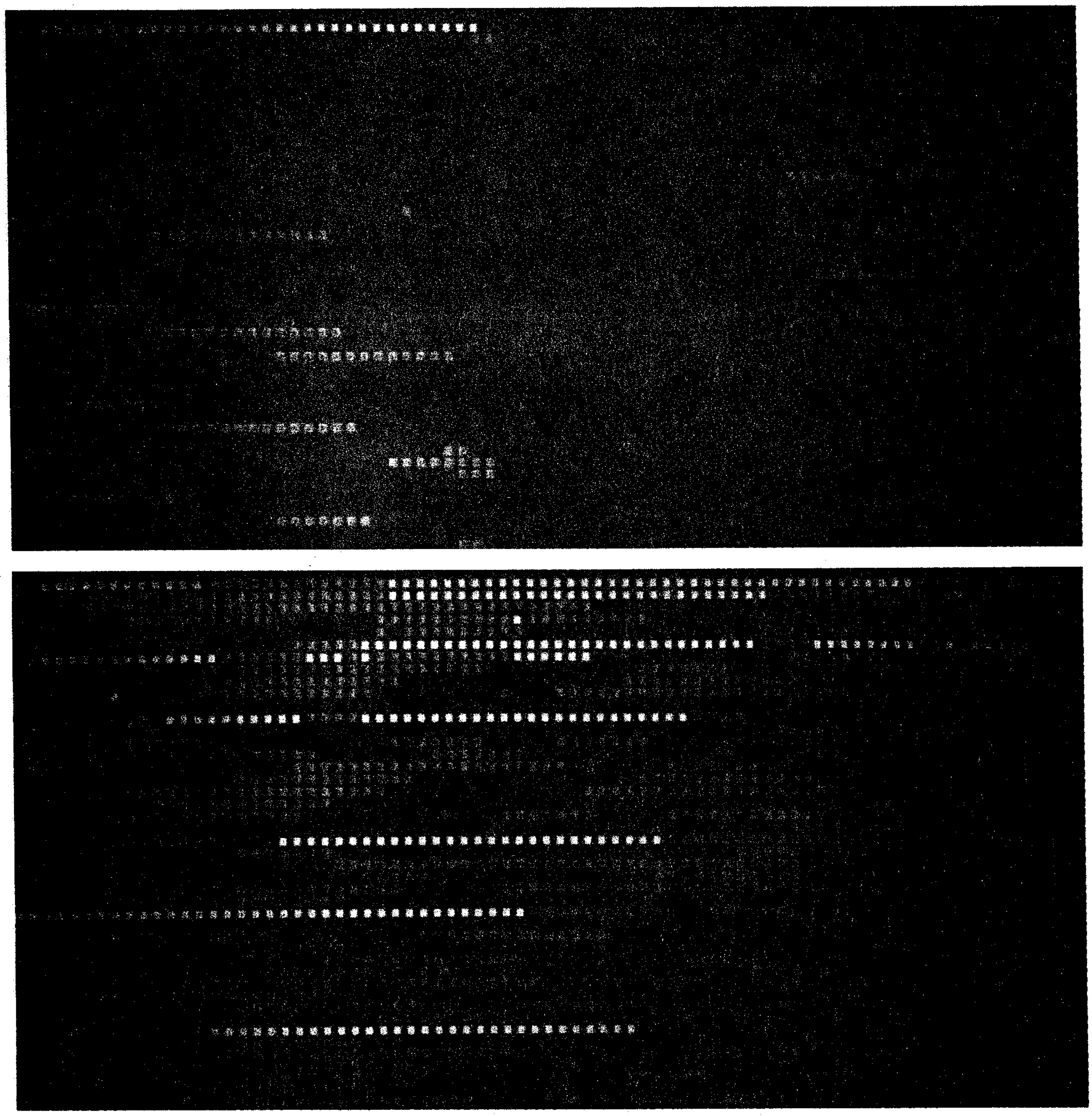
```
MOV #0 −1
JMP −1
SPL −2
MOV 10 113
SPL 112
   •
   •
   •
```

Readers will recognize an IMP-STOMPER in the first two instructions. Execution of the actual program begins at the third instruction, SPL −2. On COMMANDO's next two turns the first and fourth instructions will be executed. On the two turns after those, the second and fifth instructions will be executed. Each stream proceeds independently of the other and at half the speed, so to speak. In the code above, COMMANDO sets the IMP-STOMPER running on its own. Then it moves another IMP (patiently waiting 10 addresses beyond the second MOV instruction) to a distant location (113 addresses beyond). The second IMP is activated by the second SPL command.

COMMANDO's remaining instructions copy the entire program into a new segment of the core, 100 addresses beyond its present location. The new copy, like a commando just parachuted into enemy territory, is activated by a JMP command in the original program. The old copy of COMMANDO, except for the IMP-STOMPER, ceases to run. Then the entire cycle of copying begins again.

How would COMMANDO fare against its competitors? The tournament was organized to provide for as many engagements as possible between the 31 entries. A complete round robin, in which every program fought all others in turn, would have required 465 battles, more than time would allow. Consequently the entries were divided arbitrarily into two nearly equal groups, division I and division II. A round robin was then held within each division.

Imagine the strange mixture of emotions I felt when COMMANDO emerged as the winner of division II. On one



Early and late stages in a battle between MICE (red) *and* CHANG1 (blue) hand I was proud that my cybernetic child had done so well. At the same time I was somewhat mortified at the prospect of winning the tournament overall. Since I had consented to serve as a commentator for the finals, my objectivity (and credibility) would undoubtedly be strained.

The top four programs from each division were then entered in a new round robin. Three programs emerged victorious from the fray, CHANG1 by Morrison J. Chang of Floral Park, N.Y., and two entries by Chip Wendell, MIDGET and MICE. My COMMANDO fell by the wayside, mortally wounded. The final win by MICE came oddly; MIDGET and MICE both fought CHANG1 to a draw, but MICE captured the deciding point by beating MIDGET.

The contest between each pair of finalists consisted of four consecutive battles. The time limit on each engagement was 15,000 instructions per side, or approximately two minutes of real time. In each case the two battle programs were placed in random, nonoverlapping positions in the core and allowed to have a go at it. As it happened, each battle between a given pair of programs always had the same result. In the case of MICE versus CHANG1 the result was four draws.

It is fascinating to watch a Core War in progress. The display used at the tournament shows the core as a succession of cellular strips [*see illustration on preceding page*]. Each cell represents a single address in the core, and the last cell in the bottom row is contiguous with the first cell in the top row, in keeping with the circular structure of the core. The program that has the first move initially occupies address 0 and subsequently fills consecutive core locations. Its color is light blue. The opposition occupies a randomly selected segment of locations not overlapping those assigned to the first program. The color given to the second program is bright red. The color of a cell in the display is determined by the last program to alter the address it represents. In this way one has an engaging overview of the action.

Against a dark blue screen MICE and CHANG1 crept about, launched IMP's, hurled bombs and reproduced (parthenogenetically). One of the contests was typical: CHANG1 began as a strip of blue cells in the upper left-hand corner of the screen and the birth of MICE was heralded by a red strip that appeared less than halfway down the screen. Immediately MICE began to proliferate rapidly.

One of the shortest self-replicating programs I know, MICE has just eight instructions, two of which create a new copy of the program some 833 addresses beyond its present location in the core [*see top illustration on opposite page*]. The two instructions demonstrate a few additional features of the Redcode language:

```
loop MOV @ptr <5
     DJN loop ptr
```

The word *loop*, which is simply a label that stands for an address, makes Core War programs easier to write. The DJN (short for decrement and jump on nonzero values) command causes execution to jump to the instruction labeled *loop* if the value stored at another address (labeled *ptr*) is not yet zero. The @ sign indicates a system of reference known as indirection; when the MOV command is executed, it does not move the contents of the location labeled *ptr* but instead moves the contents of the contents, so to speak. The number stored at *ptr* is the address of the datum to be moved. In this case the datum is one of MICE's instructions.

The number stored at *ptr* continually changes owing to the decrementing function of the DJN command. The number starts at the last program address and steadily decrements to zero, at which point the copying loop is finished. In a similar manner the address at which the instructions are to be stored is also given by indirection. The relative address 5 initially holds the number 833 and the first instruction moved by MICE lands 832 addresses beyond the MOV command; as indicated by the < sign, the target address is decremented and MOV is executed. MICE copies itself tail first.

An SPL (split) command immediately following the loop transfers execution to the new copy of MICE. But following this successful birth the parent program begins anew. There is no limit to how many progeny a single program of this type may produce. And each new program does the same thing. MICE, indeed!

So it was that in a typical contest with CHANG1, MICE bred with incredible rapidity. Soon the screen was full of little red strips. In the meantime CHANG1 had activated a kind of IMP factory at its downstream end. The factory was achieved with only three instructions:

```
SPL 2
JMP -1
MOV 0 1
```

When execution arrives at the SPL command, it splits into two branches. One of them transfers execution to MOV 0 1. The other executes the JMP −1 instruction, which begins the process anew. In the meantime one IMP has already left the assembly line on a mousing mission. One problem with profligate IMP production is that a large number of independent streams of execution slows down every process executed; 1,000 IMP's move 1,000 times slower and more painfully than a single IMP. In any event, the fateful horde emerged slowly at the top of the display screen as an ever lengthening

INSTRUCTION	MNEMONIC	ARGUMENTS		EXPLANATION
Data statement	DAT		B	A nonexecutable statement; *B* is the data value
Move	MOV	A	B	Move contents of address *A* to address *B*.
Add	ADD	A	B	Add contents of address *A* to address *B*.
Subtract	SUB	A	B	Subtract contents of address *A* from address *B*.
Jump	JMP	A		Transfer control to address *A*.
Jump if zero	JMZ	A	B	Transfer control to address *A* if contents of address *B* are zero
Jump if not zero	JMN	A	B	Transfer control to address *A* if contents of address *B* are not zero
Decrement: Jump if not zero	DJN	A	B	Subtract 1 from contents of address *B* and transfer control to address *A* if contents of address *B* are not zero.
Compare	CMP	A	B	Compare contents of addresses *A* and *B*; if they are equal, skip the next instruction.
Split	SPL	A		Split execution into next instruction and the instruction at *A*.

A summary of Redcode, an assembly language for Core War

CHANG1

```
        MOV   #0     -1
        JMP   -1
        DAT          +9
start   SPL   -2
        SPL   4
        ADD   #-16   -3
        MOV   #0     @-4
        JMP   -4
        SPL   2
        JMP   -1
        MOV   0      1
```

MICE

```
ptr     DAT   #0
start   MOV   #12    ptr
loop    MOV   @ptr   <5
        DJN   loop   ptr
        SPL   @3
        ADD   #653   2
        JMZ   -5     -6
        DAT   833
```

Contenders for the Core War championship

solid blue strip. Would they be able to subvert MICE?

While the IMP's were reproducing, some of the MICE copies were killed by data bombs from CHANG1. A data bomb usually consists of a 0 that is launched by a MOV command into what one hopes is enemy territory. The key instruction in Chang's program is MOV #0 @ − 4. The 0 is moved to an address contained in a location that is four instructions above the MOV command. The location is continually incremented by 16 to ensure a well-spaced barrage.

As some MICE were dying in this manner, the IMP's began to exert their destructive influence. But each copy of the original MICE program carries with it a suicide option; it continually checks on whether its first instruction (which is a data statement consisting of 0 alone) is still zero. If it is not, MICE allows execution to proceed to a (non-executable) data statement and dies quietly rather than lose its soul to a tiny fiend.

If some copies of MICE were being killed by data bombs and others were executing their own execution, so to speak, to avoid capture, how did MICE survive? The answer surely lies in its profligate spawning of new copies. Many of these, after all, landed on enemy IMP's. Indeed, before time was called one copy of MICE had landed on CHANG1's home program and destroyed it. CHANG1, however, had created enough IMP's to tide it over until the closing buzzer sounded. The battle was a draw.

The art of Core War programming is surely still in its infancy. Progress will be incremental and cumulative. Some intrepid programmer will discover infallible remedies against IMP's and another will discover simple means of self-repair. Readers wanting to keep abreast of the latest developments may subscribe to *The Core War Newsletter* by writing to William R. Buckley at 5712 Kern Drive, Huntington Beach, Calif. 92649. Readers who want to write battle programs should probably join the International Core Wars Society. Mark Clarkson currently directs the society and would welcome new members. He lives at 8619 Wassall Street, Wichita, Kan. 67210–1934. One does not have to join the society, however, to order the all-important "Core War Standards" document from Clarkson. It precisely describes the syntax and semantics of Redcode programs; its cost is $4. One cannot be a Core Warrior without it.

Battle programs of the future will perhaps be longer than today's winners but orders of magnitude more robust. They will gather intelligence, lay false trails and strike at their opponents suddenly and with determination. Such trends may already be in evidence at the second international Core War tournament to be held at the Computer Museum this fall. In the meantime readers have ample opportunity to express their cleverness and cunning in Redcode language.

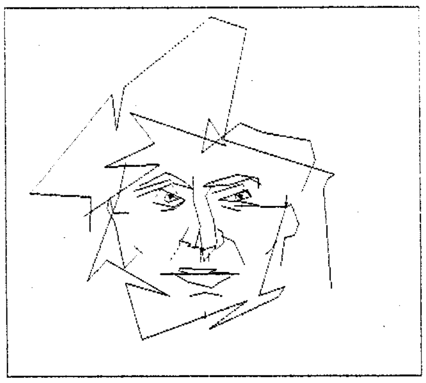
Last fall's tournament owes much of its success to Mark and Beth Clarkson as well as to Gwen Bell, president of the Computer Museum, and Oliver Strimpel, its associate director and curator. It seems worthwhile to conclude with a brief note on the museum itself.

The Computer Museum in Boston is apparently the only museum in the world devoted entirely to computers. Housed in a renovated (and now chic) warehouse on the Boston waterfront, it features old vacuum-tube monsters, PC's for personal play, walls adorned with stunning graphics, a complete NORAD SAGE computer system and a host of exhibits that entertain and educate. Readers visiting a certain famous old ship in the Boston harbor can have their computational cup of tea right next door.

In this department last October I described a program called FACEBENDER, inspired by the work of Susan E. Brennan of Hewlett-Packard Laboratories in Palo Alto, Calif. As input the program takes the digitized version of a face to be caricatured, which it then compares with an average reference face, similarly digitized, in memory. The program then distorts, or exaggerates, each feature of the input face by an amount that is proportional to its distance from the corresponding feature in the reference face; an ear that is moderately large compared with the reference ear will be enlarged still further by multiplying all the differences by an exaggeration factor *f*.

Readers who want to implement the FACEBENDER program may have been daunted by the prospect of digitizing their own face from a photograph. Pat Macaluso of White Plains, N.Y., uses the reference face as the basis of its own caricature. "The key," says Macaluso, "is to scale the range of variation to the size of each feature. Thus an ear receives more absolute variation than the chin cleft. Simply calculate the enclosing 'box' by calculating the maximum and minimum of the *x* and *y* coordinates for each feature." Within this framework the amount of distortion is governed by random numbers selected by the program. In this way an endless variety of faces can be produced by Macaluso's self-referential version of FACEBENDER. One of the caricatures so produced resembled Leonardo da Vinci. It is shown below.

A reader known only as DMI from Pasadena, Calif., has a suggestion for avoiding "facelessness," the dreaded state that occurs when the exaggeration factor is too large; all features degenerate into a wild and unrecognizable bird's nest of polygons. Imagine that the face to be caricatured is superposed on the reference face and that corresponding points are connected by springs. The distortion process now attempts to displace the points of the input face, but in doing so it encounters resistance from the springs. Small distortions are thereby hardly affected, but large ones are pulled up short of the faceless state.



Self-caricature of the average face