Core War ‘88 - A Proposed Standard[1]

Thomas Gettys

1.0 MARS Functional Units.

From the Redcode programmer’s point of view, MARS is composed of four primary functional units: a read/write memory (RAM), a computing unit (ALU), two process queues (FIFO’s), and the control unit (CU). There are no registers visible to the programmer; that is, memory is the only storage available for both instructions and data.

RAM All Redcode instructions occupy exactly one memory location. Addressing in Redcode is relative, so it is best to think of memory as being organised as a circular list; an instruction that references memory location zero is referring to itself. Every RAM location is initialised to the data pattern which corresponds to the instruct DAT 0 0 before loading any Core War programs, (elsewhere refered to as “warriors”).

FIFO A game of Core War is played by pitting two Redcode programs against each other. Each attempts to force the other to fail by causing it to execute a “halt” instruction, (the DAT). During each machine cycle one instruction from each program is executed, always in the same order. it is thus necessary to maintain a program counter for each side. This is the purpose of the process queues. As will be seen, each side may consist of more than one process. This is why a FIFO and not a single register is necessary for each side. There are no upper or lower limits to the size of the FIFO’s, save the minimum of one each, as battles cannot occur with less than two Core War programs.

ALU The computing unit performs all the arithmetic and logical operations required by the Redcode instruction set, and the Control Unit, such as adding two operands or incrementing a program counter.

CU As in any processor, the control unit has the responsibility of fetching, decoding and executing instructions. Details about the fetch and decode phase will be delayed until the instruction set has been introduced and defined, (see the section on Effective Address Calculation). The control unit must also provide an interface to the MARS supervisor function (ZEUS). The supervisor may provide various support functions such as a Redcode

1. This text is reproduced from the article “Core War ‘88 - A Proposed Standard” by Thomas Gettys which appeared in the Summer 1988 edition of The Core War Newsletter, Copyright (c) 1988 by AMRAN. It appears here by the permission of AMRAN and the International Core War Society, both of which retain the aforementioned copyright. Additional information is available by contacting the ICWS at 5712 Kern Drive, Huntington Beach, CA 92649-4535. A $1.00 donation for this document is requested.

debugger, graphics display, parameter control, etc. and is highly implementation specific.

2.0 Instruction Format

A Redcode instruction has three fields: the opcode field and two operand fields, denoted as A and B. The contents of the opcode field specify the operation to be performed, and the addressing modes to be used in evaluating the operands. For pragmatic purposes it is useful to divide the opcode into three sub-fields: the operation and the two operand modifiers, but this is not strictly necessary. The operand fields may contain any number from zero to the memory size minus one. Currently, eleven Redcode instructions are defined. They are listed in table 1 below, with a mnemonic and short operational description for each.

TABLE 1. Instruction set of the proposed Core War Standard of 1988

Opcode	Function
DAT A B	Remove executing process from process queue
MOV A B	Move A to B
ADD A B	Add A to B
SUB A B	Subtract A from B
JMP A B	Jump to A
JMZ A B	Jump to A if B is zero.
JMN A B	Jump to A if B is not zero
CMP A B	If A equals B then skip the next instruction
SLT A B	If A is less than B, then skip next instruction
DJN A B	Decrement B; If B is not zero then jump to A
SPL A B	Place A in the process queue.

3.0 Addressing Modes

There are currently four addressing modes defined: immediate, direct, indirect, and pre-decrement-indirect.

The default mode is direct. If no modifier symbol precedes an operand, the value of the operand is used as an offset from the memory locations from which it was fetched. The resulting memory location is the source and/or destination of the data to be used by the instruction, or else is the destination for branching instructions.

An octothorpe (#) is used to introduce an immediate operand. The value of the operand is not an address but rather the data to be used by the instruction; the data is “immediately” available.

The commercial at sign (@) is used to introduce an indirect operand. The value of the operand is used as an offset, as it is with direct addressing. The B operand of the resulting memory location is then used as an offset from the memory location from which it was fetched. The resulting memory location is the source and/or destination of the data to be used by the instruction, of else is the destination for branching instructions.

The less than sign (<) is used to introduce a pre-decrement indirect operand. The value of the operand is used as an offset, as it is with direct addressing. The B operand of the resulting memory location is fetched, decremented, and then restored. It is then used as an offset from the memory location from which it was fetched. The resulting memory location is the source and/or destination for the data to be used by the instruction, or else is the destination for branching instructions.

4.0 The Redcode Instruction Set

For the most part, the Redcode instruction set presents no unusual problems, and the implementation is straight-forward. Some additional definition and clarification is required however to insure that it is implemented correctly.

`DAT A B`	This instruction actually serves two functions, one utilitarian, and the other quite destructive. On the one hand, it is used for the storage of data to be used as a counter, pointer, etc. If executed however, the process is halted! This, then, is the mechanism by which one obtains victory in a Core War: cause every process associated with the opponent's program to execute a DAT instruction.
`MOV A B`	If the A operand is immediate, it is placed in the B-field of the memory location specified by the B operand, otherwise the contents of the entire memory location specified by the A operand is copied to the memory location specified by the B operand.
`ADD A B`	If the A operand is immediate it is added to the B-field of the B operand. If the A operand is not immediate both the A-field and B-field of the A operand are added respectively to the A-field and B-field of the B operand.
`SUB A B`	If the A operand is immediate it is subtracted from the B-field of the B operand. If the A operand is not immediate both the A-field and B-field of the A operand are subtracted respectively from the A-field and B-field of the B operand.
`JMP A B`	The address of the memory location specified by the A operand is placed at the back of the process queue associated with the executing program. The B operand does not necessarily participate in the execution of the instruction.
`JMZ A B`	If the B-field of the B operand is zero then the address of the memory location specified by the A operand is placed at the back of the process queue associated with the executing program.

`JMN A B` If the B-field of the B operand is not zero then the address of the memory location specifiec by the A operand is placed at the back of the process queue associated with the executing program.

`CMP A B` If the A operand is immediate it is compared to the B-field of the memory location specified by the B operand, otherwise the contents of the entire memory location specified by the A operand is compared to the contents of the memory location specified by the B operand. If the compared values are equal, the next instruction is skipped (the program counter is incremented).

`SLT A B` If the A operand is not immediate, the B-field of the memory location specified by the A operand is compared to the B-field of the B operand, otherwise the A operand itself is used in the comparison. If the A value is less than the B value, the next instruction is skipped (the program counter is incremented).

`DJN A B` If the B operand is not immediate, the B-field of the memory location specified by the B operand is fetched, decremented, and then restored, otherwise the B-field of the current instruction is used. If the value is not zero, the address of the memory location specified by the A operand is placed at the back of the process queue associated with the executing program.

`SPL A B` After a process has caused an SPL instruction to be fetched, the program counter is incremented and placed at the back of its process queue. The address ot the spawned process is then placed at the back of the same queue, providing the queue is not full. The B operand does not necessarily participate in the execution of this instruction.

5.0 Effective Address Calculations.

The fetch-decode phase of Redcode instruction execution is complicated somewhat by the predecrement-indirect addressing mode. After the instruction to be executed has been fetched, the operands must be “normalised” before the opcode is processed. That is, the operands are processed so as to remove any indirection. Because this may involve the modification of memory, the order of evaluation must be specified exactly.

First the A addressing mode is examined. If immediate, we are done. Otherwise, the program counter is added to the value in the A-field, yielding an absolute address. If direct, we are done. Otherwise, the B-field of this direct memory location is read into the A-field. If predecrement is indicated, the value just read is decremented and written back to the memory location from whence it came. Finally, the program counter is added to the value in the A-field.

At this point the A-field either contains immediate data or else it contains the absolute address of the A operand. The same procedure is now performed to normalise the B operand. With bother operands now normalised, the operation code is examined and the specified operation performed.

One final aspect of instruction execution remains to be resolved. Because of the process queue FIFO's implied by the SPL instruction, the updating of the program counter associated with the current process must be delayed until the opcode is processed. If the instruction does not require a branch to be taken and the instruction is not a DAT, the program counter is incremented and placed at the back ot the process queue associated with the executing program before the instruction is executed.

6.0 Source Code Syntax

The source code file consists of lines as generated by a typical ASCII editor. A line may be classified as either a blank line, a comment line, a Redcode instruction, or as a pseudo-instruction.

Blank lines are used solely for cosmetic purposes in the source and listing files. Blank lines enhance readability by allowing groups of related instructions to be easily separated.

All comments are introduced by a semicolon, (;). Comments may be placed anywhere within the source file. All characters following the semicolon are ignored.

A Redcode instruction consists of five components (fields) : an optional label, the instruction mnemonic, the A and B operands, and an optional comment. Fields are separated by one or more space and/or tab characters.

Labels begin in the first column and are composed of alpha-numeric characters. Only the first eight characters of a label are significant. If a label is present, it will take the current value of the program counter.

An operand consists of the addressing mode symbol followed by an expression that is composed of labels, number, and operator symbols. The valid operators are addition (+), subtraction (-), multiplication (*), and integer division (/).

A pseudo-instruction is a directive to the assembler; object code is not generated by a pseudo instruction. The mnemonic and definition of each pseudo instruction follows.

`EQU`	The EQU pseudo-instruction must be preceded by a label, and foloowed by an expression which is consistent with the rules governing operand expressions. The value of the expression is the value associated with the label. The expression may contain such other labels as a priorly defined.
`END`	The END pseudo-instruction follows the last line to be assembled. All lines following the END statement are ignored. Additionally, if an operand is provided, it will be used as the program entry point. If no operand appears, the entry point is assumed to be the first instruction line in the file.

7.0 Appendix A

The following is a complete list of all legal Redcode instructions under this proposed standard.

Opcode	Opcode	Opcode	Opcode
MOV # A B	SLT # A B	DJN A # B	SPL A # B
MOV # A @ B	SLT # A @ B	DJN A B	SPL A B
MOV # A < B	SLT # A < B	DJN A @ B	SPL A @ B
MOV A B	SLT A B	DJN A < B	SPL A < B
MOV A @ B	SLT A @ B	DJN @ A # B	SPL @ A # B
MOV A < B	SLT A < B	DJN @ A B	SPL @ A B
MOV @ A B	SLT @ A B	DJN @ A @ B	SPL @ A @ B
MOV @ A @ B	SLT @ A @ B	DJN @ A < B	SPL @ A < B
MOV @ A < B	SLT @ A < B	DJN < A # B	SPL < A # B
MOV < A B	SLT < A B	DJN < A B	SPL < A B
MOV < A @ B	SLT < A @ B	DJN < A @ B	SPL < A @ B
MOV < A < B	SLT < A < B	DJN < A < B	SPL < A < B
ADD # A B	JMP A # B	SUB # A B	JMZ A # B
ADD # A @ B	JMP A B	SUB # A @ B	JMZ A B
ADD # A < B	JMP A @ B	SUB # A < B	JMZ A @ B
ADD A B	JMP A < B	SUB A B	JMZ A < B
ADD A @ B	JMP @ A # B	SUB A @ B	JMZ @ A # B
ADD A < B	JMP @ A B	SUB A < B	JMZ @ A B
ADD @ A B	JMP @ A @ B	SUB @ A B	JMZ @ A @ B
ADD @ A @ B	JMP @ A < B	SUB @ A @ B	JMZ @ A < B
ADD @ A < B	JMP < A # B	SUB @ A < B	JMZ < A # B
ADD < A B	JMP < A B	SUB < A B	JMZ < A B
ADD < A @ B	JMP < A @ B	SUB < A @ B	JMZ < A @ B
ADD < A < B	JMP < A < B	SUB < A < B	JMZ < A < B
CMP # A B	JMN A # B		
CMP # A @ B	JMN A B		
CMP # A < B	JMN A @ B		
CMP A B	JMN A < B		
CMP A @ B	JMN @ A # B		
CMP A < B	JMN @ A B		DAT # A # B
CMP @ A B	JMN @ A @ B		DAT # A < B
CMP @ A @ B	JMN @ A < B		DAT < A # B
CMP @ A < B	JMN < A # B		DAT < A < B
CMP < A B	JMN < A B		
CMP < A @ B	JMN < A @ B		
CMP < A < B	JMN < A < B		