Newsgroups: rec.games.corewar
From: DURHAM@ricevm1.rice.edu (Mark A. Durham)
Subject: Intro to Redcode Part I
Organization: Rice University, Houston, TX
Date: Thu, 14 Nov 1991 09:41:37 GMT

Introduction to Redcode

I. Preface - Reader Beware!

The name "Core War" arguably can be claimed as public domain. Thus, any program can pass itself off as an implementation of Core War. Ideally, one would like to write a Redcode program on one system and know that it will run in exactly the same manner on every other system. Alas, this is not the case.

Basically, Core War systems fall under one of four catagories: Non-ICWS, ICWS'86, ICWS'88, or Extended. Non-ICWS systems are usually a variant of Core War as described by A. K. Dewdney in his "Computer Recreations" articles appearing in Scientific American. ICWS'86 and ICWS'88 systems conform to the standards set out by the International Core War Society in their standards of 1986 and 1988, respectively. Extended systems generally support ICWS'86, ICWS'88, and proprietary extensions to those standards. I will discuss frequently common extensions as if they were available on all Extended systems (which they most certainly are not).

I will not describe Non-ICWS systems in this article. Most Non-ICWS systems will be easily understood if you understand the systems described in this article however. Although called "standards", ICWS'86 and ICWS'88 (to a lesser extent) both suffer from ambiguities and extra-standard issues which I will try to address.

This is where the reader should beware. Because almost any interpretation of the standard(s) is as valid as any other, I naturally prefer MY interpretation. I will try to point out other common interpretations when ambiguities arise though, and I will clearly indicate what is interpretation (mine or otherwise) as such. You have been warned!

II. Notation

"86:" will indicate an ICWS'86 feature. "88:" will indicate an ICWS'88 feature. "X:" will indicate an Extended feature. "Durham:" will indicate my biased interpretation. "Other:" will indicate interpretations adhered to by others. "Commentary:" is me explaining

what I am doing and why. "Editorial:" is me railing for or against certain usages. Items without colon-suffixed prefaces can be considered universal.

   Redcode consists of assembly language instructions of the form

<label>   <opcode> <A-mode><A-field>, <B-mode><B-field>   <comment>

An example Recode program:

```
; Imp
; by A. K. Dewdney
;
imp     MOV imp, imp+1      ; This program copies itself ahead one
        END                 ; instruction and moves through memory.
```

The <label> is optional.
86: <label> begins in the first column, is one to eight characters long, beginning with an alphabetic character and consisting entirely of alphanumerals. Case is ignored ("abc" is equivalent to "ABC").
88: <label> as above, except length is not limited and case is not addressed. Only the first eight characters are considered significant.
X: <label> can be preceded by any amount of whitespace (spaces, tabs, and newlines), consists of any number of significant alphanumerals but must start with an alphabetic, and case is significant ("abc" is different from "ABC").
Commentary: I will always use lowercase letters for labels to distinguish labels from opcodes and family operands.

The <opcode> is separated from the <label> (if there is one) by whitespace. Opcodes may be entered in either uppercase or lowercase. The case does not alter the instruction. DAT, MOV, ADD, SUB, JMP, JMZ, JMN, DJN, CMP, SPL, and END are acceptable opcodes.
86: SPACE is also recognized as an opcode.
88: SLT and EQU are recognized as opcodes. SPACE is not.
X: All of the above are recognized as opcodes as well as XCH and PCT, plus countless other extensions.
Commentary: END, SPACE, and EQU are known as pseudo-ops because they really indicate instructions to the assembler and do not produce executable code. I will always capitalize opcodes and pseudo-ops to distinguish them from labels and text.

The <A-mode> and <A-field> taken together are referred to as the A-operand. Similarly, the <B-mode><B-field> combination is known as the B-operand. The A-operand is optional for some opcodes. The B-operand is optional for some opcodes. Only END can go without at least one operand.
86: Operands are separated by a comma.
88: Operands are separated by whitespace.
X: Operands are separated by whitespace and/or a comma. Lack of a comma can lead to unexpected behaviour for ambiguous constructs.
Commentary: The ’88 standard forces you to write an operand without whitespace, reserving whitespace to separate the operands. I like whitespace in my expressions, therefore I prefer to separate my operands with a comma and will do so here for clarity.

<mode> is # (Immediate Addressing), @ (Indirect Addressing), or < (86: Auto-Decrement Indirect, 88: Pre-Decrement Indirect). A

    missing mode indicates Direct Addressing.
86: $ is an acceptable mode, also indicating Direct Addressing.
88: $ is not an acceptable mode.
X: $ is an acceptable mode as in 86:.
Commentary: The distinction between Auto-Decrement Indirect Addressing
    and Pre-Decrement Indirect Addressing is semantic, not syntactic.

<field> is any combination of labels and integers separated by the
    arithmetic operators + (addition) and - (subtraction).
86: Parentheses are explicitly forbidden. "*" is defined as a special
    label symbol meaning the current statement.
88: Arithmetic operators * (multiplication) and / (integer division)
    are added. "*" is NOT allowed as a special label as in 86:.
X: Parentheses and whitespace are permitted in expressions.
Commentary: The use of "*" as meaning the current statement may be
    useful in some real assemblers, but is completely superfluous in a
    Redcode assembler. The current statement can always be referred
    to as 0 in Redcode.

<comment> begins with a ; (semicolon), ends with a newline, and can
    have any number of intervening characters. A comment may appear
    on a line by itself with no instruction preceding it.
88: Blank lines are explicitly allowed.

   I will often use "A" to mean any A-operand and "B" to mean any B-operand (capitalization is important). I use "a" to mean any A-field and "b" to mean any B-field. For this reason, I never use "a" or "b" as an actual label.
   I enclose sets of operands or instructions in curly braces. Thus "A" is equivalent to "{ a, #a, @a, <a }". I use "???" to mean any opcode and "x" or "label" as an arbitrary label. Thus, the complete family of acceptable Redcode statements can be represented as

```
x    ??? A, B   ; This represents all possible Redcode statements.
```

"???" is rarely used as most often we wish to discuss the behaviour of a specific opcode. I will often use labels such as "x-1" (despite its illegality) for the instruction before the instruction labelled "x", for the logically obvious reason. "M" always stands for the integer with the same value as the MARS memory size.

III. MARS Peculiarities

   There are two things about MARS which make Redcode different from any other assembly language. The first of these is that there are no absolute addresses in MARS. The second is that memory is circular.
   Because there are no absolute addresses, all Redcode is written using relative addressing. In relative addressing, all addresses are interpreted as offsets from the currently executing instruction. Address 0 is the currently executing instruction. Address -1 was the previously executed instruction (assuming no jumps or branches). Address +1 is the next instruction to execute (again assuming no jumps or branches).
   Because memory is circular, each instruction has an infinite number of addresses. Assuming a memory size of M, the current instruction has the addresses { ..., -2M, -M, 0, M, 2M, ... }. The previous instruction is { ..., -1-2M, -1-M, -1, M-1, 2M-1, ... }. The next

instruction is { ..., 1-2M, 1-M, 1, M+1, 2M+1, ... }.

Commentary: MARS systems have historically been made to operate on object code which takes advantage of this circularity by insisting that fields be normalized to positive integers between 0 and M-1, inclusive. Since memory size is often not known at the time of assembly, a loader in the MARS system (which does know the memory size) takes care of field normalization in addition to its normal operations of code placement and task pointer initialization.

Commentary: Redcode programmers often want to know what the memory size of the MARS is ahead of time. This is not always possible. Since normalized fields can only represent integers between 0 and M-1 inclusive, we can not represent M in a normalized field. The next best thing? M-1. But how can we write M-1 when we do not know the memory size? Recall from above that -1 is equivalent to M-1. Final word of caution: -1/2 is assembled as 0 (not as M/2) since the expression is evaluated within the assembler as -0.5 and then truncated.

86: Only two assembled-Redcode programs (warriors) are loaded into MARS memory (core).
88: Core is initialized to (filled with) DAT 0, 0 before loading any warriors. Any number of warriors may be loaded into core.

Commentary: Tournaments almost always pit warrior versus warrior with only two warriors in core.

MARS is a multi-tasking system. Warriors start as just one task, but can "split" off additional tasks. When all of a warriors tasks have been killed, the warrior is declared dead. When there is a sole warrior still executing in core, that warrior is declared the winner.
86: Tasks are limited to a maximum of 64 for each warrior.
88: The task limit is not set by the standard.

Newsgroups: rec.games.corewar
From: DURHAM@ricevm1.rice.edu (Mark A. Durham)
Subject: Intro to Redcode Part II
Organization: Rice University, Houston, TX
Date: Thu, 14 Nov 1991 09:45:13 GMT

IV. Address Modes

Addressing modes subtly (sometimes not-so-subtly) alter the behaviour of instructions. A somewhat brief description of their general properties is given here. Specifics will be left to the instruction set section.

An octothorpe (#) is used to indicate an operand with an Immediate Address Mode. Immediate mode data is contained in the current instruction's field. If the A-mode is immediate, the data is in the A-field. If the B-mode is immediate, the data is in the B-field.

If no mode indicator is present (86: or the US dollar sign '$' is present), Direct Address Mode is used. Direct addresses refer to instructions relative to the current instruction. Address 0 refers to the current instruction. Direct address -1 refers to the (physically) previous instruction. Direct address +1 refers to the (physically) next instruction.

The commercial-at (@) is used to indicate Indirect Address Mode. In indirect addressing, the indirect address points to an instruction as in direct addressing, except the target is not the instruction to which the indirect address points but rather the instruction pointed to by the B-field of the instruct pointed to by the indirect address. Example:

```
x-2     DAT  #0,  #0   ; Target instruction.
x-1     DAT  #0, #-1   ; Pointer instruction.
x       MOV   0, @-1   ; Copies this instruction to location x-2.
```

The less-than (<) is used to indicate (86: Auto-, 88: Pre-) Decrement Indirect Address Mode. Its behaviour is just like that of Indirect Address Mode, except the pointer is decremented before use. Example:

```
x-2     DAT  #0,  #0   ; Target instruction
x-1     DAT  #0,  #0   ; Pointer instruction.  Compare to @ example.
x       MOV   0, <-1   ; Copies this instruction to location x-2.
```

Commentary: Although Decrement Indirect addressing appears to be a simple extension of Indirect addressing, it is really very tricky at times - especially when combined with DJN. There are sematic differences between the '86 and '88 standards, thus the change in name from Auto-Decrement to Pre-Decrement. These differences are discussed below. This discussion is non-essential for the average Redcode programmer. I suggesting skipping to the next section for the weak-stomached.

86: Durham: Instructions are fetched from memory into an instruction register. Each operand is evaluated, storing a location (into an address register) and an instruction (into a value register) for each operand. After the operands have been evaluated, the instruction is executed.

Operand Evaluation: If the mode is immediate, the address register is loaded with 0 (the current instruction's address) and the value register is loaded with the current instruction. If the mode is direct, the address register is loaded with the field value and the value register is loaded with the instruction pointed to by

the address register. If the mode is indirect, the address register is loaded with the sum of the field value and the B-field value of the instruction pointed to by the field value and the value register is loaded with the instruction pointed to by the address register. If the mode is auto-decrement, the address register is loaded with a value one less than the sum of the field value and the B-field value of the instruction pointed to by the field value and the value register is loaded with the instruction pointed to by the address register. AFTER the operands have been evaluated (but before instruction execution), if either mode was auto-decrement, the appropriate memory location is decremented. If both modes were auto-decrement and both fields pointed to the same pointer, that memory location is decremented twice. Note that this instruction in memory then points to a different instruction than either operand and also differs from any copies of it in registers.

86: Other: As above, except there are no registers. Everything is done in memory.

Commentary: ICWS'86 clearly states the use of an instruction register, but the other operand address and value registers are only implied. Ambiguities and lack of strong statements delineating what takes place in memory and what takes place in registers condemned ICWS'86 to eternal confusion and gave birth to ICWS'88.

88: As above except everything is done in memory and Pre-Decrement Indirect replaces Auto-Decrement Indirect. Pre-Decrement Indirect decrements memory as it is evaluating the operands rather than after. It evaluates operand A before evaluating operand B.

V. Instruction Set

DAT A, B

The DAT (data) instruction serves two purposes. First, it allows you to store data for use as pointers, offsets, etc. Second, any task which executes a DAT instruction is removed from the task queue. When all of warrior's tasks have been removed from the queue, that warrior has lost.

86: DAT allows only one operand - the B-operand. The A-field is left undefined (the example shows #0), but comparisons of DAT instructions with identical B-operands must yield equality.

88: DAT allows two operands but only two modes - immediate and pre-decrement.

X: DAT takes one or two operands and accepts all modes. If only one operand is present, that operand is considered to be the B-operand and the A-operand defaults to #0.

Commentary: It is important to note that any decrement(s) WILL occur before the task is removed from the queue since the instruction executes only after the operand evaluation.

MOV A, B

The MOV (move) instruction either copies a field value (if either mode is immediate) or an entire instruction (if neither mode is immediate) to another location in core (from A to B).

86: Durham: MOV #a, #b changes itself to MOV #a, #a.

Commentary: There is a clear typographical error in ICWS'86 which changes the interpretation of MOV #a, B to something non-sensical. For those with a copy of ICWS'86, delete the term "B-field" from the next-to-last line of the second column on page 4.

88: No immediate B-modes are allowed.

X: Immediate B-modes are allowed and have the same effect as a
    B-operand of 0. (See 86: Durham: above).

ADD A, B
86: The ADD instruction adds the value at the A-location to the value
    at the B-location, replacing the B-location's old contents.
88: If the A-mode is immediate, ADD is interpreted as above. If the
    A-mode is not immediate, both the A-field and the B-field of the
    instruction pointed to by the A-operand are added to the A-field
    and B-field of the instruction pointed to by the B-operand,
    respectively. The B-mode can not be immediate.
X: Immediate B-modes are allowed and have the same effect as in 86:.
    Example: ADD #2, #3 becomes ADD #2, #5 when executed once.

SUB A, B
   The SUB (subtract) instruction is interpreted as above for all
three cases, except A is subtracted from B.

JMP A, B
   The JMP (jump) instruction changes the instruction pointer to point
to the instruction pointed to by the A-operand.
86: JMP allows only one operand - the A-operand. The B-operand is
    shown as #0.
88: JMP allows both operands, but the A-mode can not be immediate.
X: JMP allows both operands and the A-mode can be immediate. An
    immediate A-mode operand is treated just like JMP 0, B when
    executed.

JMZ A, B
   The JMZ (jump if zero) instruction jumps to the instruction pointed
to by the A-operand only if the B-field of the instruction pointed to
by the B-operand is zero.
88: Immediate A-modes are not allowed.

JMN A, B
   The JMN (jump if non-zero) instruction jumps to the instruction
pointed to by the A-operand only if the B-field of the instruction
pointed to by the B-operand is non-zero.
88: Immediate A-modes are not allowed.

DJN A, B
   The DJN (decrement and jump if non-zero) instruction causes the
B-field of the instruction pointed to by the B-operand to be
decremented. If the decremented values is non-zero, a jump to the
instruction pointed to by the A-operand is taken.
88: Immediate A-modes are not allowed.

CMP A, B
   The CMP (compare, skip if equal) instruction compares two fields
(if either mode is immediate) or two entire instructions (if neither
mode is immediate) and skips the next instruction if the two are
equivalent.
Commentary: There is a clear typographical error in ICWS'86 which
    changes the interpretation of CMP #a, B to something non-sensical.
    For those with a copy of ICWS'86, delete the term "B-field" from
    the fifth line from the bottom of the second column on page 5.
    Also, the comments to the example on page 6 have been switched
    (equal is not equal and vice versa). The labels are correct
    though.
88: Immediate B-modes are not allowed.

SPL A, B

The SPL (split) instruction splits the execution between this warrior's currently running tasks and a new task. Example: A battle between two warriors, 1 and 2, where warrior 1 has two tasks (1 and 1') and warrior 2 has only one task would look like this: 1, 2, 1', 2, 1, 2, 1', 2, etc.

86: SPL allows only one operand - the B-operand. The A-operand is shown as #0. After executing the SPL, the next instruction to execute for this warrior is that of the newly added task (the new task is placed at the front of the task queue). A maximum of 64 tasks is allowed for each warrior.

88: SPL splits the A-operand, not the B-operand. After executing the SPL, the next instruction to execute for this warrior is the same instruction which would have executed had another task not been added (the new task is placed at the back of the task queue). There is no explicit task limit on warriors. Immediate A-operands are not allowed.

X: Immediate A-operands are allowed and behave as SPL 0, B when executed.

88: SLT A, B: The SLT (skip if less than) instruction skips the next instruction if A is less than B. No Immediate B-modes are allowed.

X: Immediate B-modes are allowed.

X: XCH A, B: The XCH (exchange) instructions exchanges the A-field and the B-field of the instruction pointed to by the A-operand.

X: PCT A, B: The PCT (protect) instruction protects the instruction pointed to by the A-operand until the protection is removed by an instruction attempting to copy over the protected instruction.

Pseudo-Ops: Instructions to the Assembler

END

The END pseudo-op indicates the end of the Redcode source program.

86: END takes no operands.

88: If END is followed by a label, the first instruction to be executed is that with the label following END.

X: ORG A (origin) takes over this initial instruction function from END.

Commentary: If no initial instruction is identified, the first instruction of your program will be the initial instruction. You can accomplish the same effect as "END start" or "ORG start" by merely starting your program with the instruction "JMP start".

86: SPACE A, B: The SPACE pseudo-op helps pretty-up Redcode source listings. SPACE A, B means to skip A lines, then keep B lines on the next page. Some assemblers do not support SPACE, but will treat it as a comment.

88: label EQU A: The EQU (equate) pseudo-op gives the programmer a macro-like facility by replacing every subsequent occurrence of the label "label" with the string "A".

Commentary: A normal label is a relative thing. Example:

```
x       DAT  #0,  #x   ; Here x is used in the B-field
x+1     DAT  #0,  #x   ; Each instruction's B-field gives
x+2     DAT  #0,  #x   ;    the offset to x.
```

is the same as

```
x       DAT  #0,  #0   ; Offset of zero
x+1     DAT  #0, #-1   ;    one
x+2     DAT  #0, #-2   ;    two

but

x!      EQU   0        ; Equate label like #define x! 0
        DAT  #0,  #x!  ; Exclamation points can be used
        DAT  #0,  #x!  ;    in labels (in Extended systems)
        DAT  #0,  #x!  ; I use them exclusively to indicate
                       ;    immediate equate labels.

is the same as

        DAT  #0,  #0   ; A direct text replacement
        DAT  #0,  #0   ;    appears the same on every
        DAT  #0,  #0   ;    line it is used.
```
