CS4012 Project
Core War

Glenn Strong
Glenn.Strong@cs.tcd.ie

Michaelmas Term 2017

The game Core War was invented in 1983 by A.K. Dewdney and implemented by Dewdney and David Jones at the University of Western Ontario. It was popularised by an article in the May 1984 issue of Scientific American. Since the original publication of Core War the game has become more sophisticated, but in this project we will be concerned only with the original game as described in 1984.

Core War is a game in which programs written in a specialised assembly language (called *redcode*) are executed on a simulated computer. Each programs object is to disable all other programs running on the simulator without itself becoming disabled. This can be achieved by writing illegal instructions into memory locations which a program will attempt to execute.

1 The machine

The simulator for redcode is usually called MARS (Memory Array Redcode Simulator). The MARS operates by maintaining an array which represents the memory (or "core") of the simulated computer, which is empty apart from the competing programs. Each location in the array is capable of storing one redcode instruction. The programs have no way of determining their location in the memory since all addressing in redcode is *instruction-relative* and the memory is circular (see the redcode section for explanations of the addressing scheme). The execution model is simple: each program is allowed to execute one instruction at a time in a round-robin fashion (that is, program 1 executes an instruction, then program 2, then program 3 and when all programs have had a turn then program 1 is allowed to execute a second instruction, and so on). In general each program will execute its first instruction, followed by the next instruction in memory, then the next, and so on (although some instructions can alter this order).

Each program starts at a random location in memory, not less than 1000 addresses apart. A time limit is placed on each game. If both programs are still running when the time has elapsed then the game is declared a draw.

A complication is that MARS is a multi-tasking system. This means that each program may have more than one task active at a time. Each program starts with only one task. It is possible for a program to split off a new task (see the redcode section following for the mechanism by which this happens), and each task has it's own notion of what the next instruction to be executed is.

A program can still only execute one instruction before another program gets it's turn, so the tasks are also scheduled in a round-robin fashion. For example, if program 1 has two

tasks (A and B) and program 2 has only one task, then the order of execution will be: A, Program 2, B, Program 2, A, Program 2, and so on.

A program is removed from play when all of it's tasks have been disabled.

2 Redcode

Redcode is a simple low-level language intentionally similar to a typical assembly language. It contains instructions to move one memory location to another, perform simple arithmetic and to alter the execution order of the program. Each redcode instruction has three parts: the instruction itself and two *fields* representing the data or arguments in the instruction (some instructions only permit one of the two fields to be occupied).

2.1 Fields

The two fields are designated A and B, and each is further subdivided into an addressing mode which affects the interpretation of the field, and a value. Unused fields (in instructions which only permit one of the fields to be used) will contain zero values.

Redcode instructions each take either one or two values as arguments. The values are usually *addresses* which specify which part of the machines memory the instruction find values in or place values in. Each address is prefixed by a symbol indicating the addressing mode. The mode controls the interpretation of the field. There are four addressing modes in redcode: *Direct* (no prefix, or a dollar sign), which means the address itself is the target, *Indirect* (prefixed by `@`), which means that the contents of the address specify the target address, *Immediate* (prefixed by `#`), which means that the address should be treated as an integer, and *Auto-decrement* (prefixed by `<`) which indicates that the number in the field should be decremented by one and then treated as an indirect address.

The value in a field is a simple number, optionally preceded by a sign.

2.2 Instructions

An unusual feature of redcode is that all instructions are calculated relative to the *current* instruction, so the instruction `JMP 1` means "jump to the next location" (this is the default behaviour anyway, so this instruction really means "do nothing"). Similarly, `ADD #1 -1` means "add the integer one to the contents of the previous location". Equally unusual is the property that the machines memory is *circular* (that is, all addresses are calculated modulo the memory size). This means if the last real memory location is 8000 (a common size) then attempting to address location 8000+1 will in fact get location zero. This feature allows programs to ignore the special cases that would otherwise arise when the approached the boundaries of the machines memory.

`DAT` Contains only a B-Field. The `DAT` instruction has two uses: it causes a program to halt when it is executed (thus it is the primary "weapon" in the arsenal of a redcode warrior). Additinally, the B-Field of the instruction can be used to store a numerical value (it can thus be used to store data, hence the mnemonic "DAT").

`MOV` The `MOV` instruction will copy the complete contents of the location indicated by the A field into the location indicated by the B field. If the A field indicates an *immediate* value (i.e. it is not an address) then a MOV instruction is created in the target address,

and the value of the A field is placed in the new instructions B-field (this is the primary mechanism by which warriors create DAT instruction in places where they hope their opponents will attempt to execute instructions). The B field may not be immediate.

`ADD` Takes the contents of the A-Field and adds it to the contents of the B-Field. Remember, all arithmetic in redcode is performed modulo the number of locations in memory.

`SUB` Takes the contents of the A-Field and subtracts it from the contents of the B-Field. All arithmetic in redcode is performed modulo the number of locations in memory.

`JMP` This is a jump instruction that causes the instruction referenced in the A-Field to be the next one executed.

`JMZ` This is a conditional jump instruction. If the value indicated by the B-Field is zero then it jumps to the instruction indicated by the A-Field, otherwise it does nothing.

`JMN` This is a conditional jump instruction. If the value indicated by the B-Field is non-zero then it jumps to the instruction indicated by the A-Field, otherwise it does nothing.

`DJN` The most complex of the jump instructions. Decrements the value indicated by the B-Field and then jumps to the instruction indicated by the A-Field if the number indicated by the B-Field has become non-zero.

`CMP` Compares the values indicated by the A and B-Fields and skips the next instruction if they are not equal.

`SPL` Contains only an A-Field. Splits execution into the next instruction and the instruction contained in the specified address, thus creating a new task for the program.

2.3 Input file format

Each core wars program is loaded from a source file when the MARS simulator starts up. The format of the file is described in this section.

To avoid ambiguities we assume that each non-blank line of the input file is formatted as follows:

- An opcode (typically in all-caps)
- Some whitespace (separating the opcode and the first field)
- The addressing mode for the A field (may be omitted if the mode is Direct)
- The value for the A field (this, and the previous item, are omitted if the instruction does not use the A field)
- A comma (optionally with some whitespace on either side to allow the programmer to format the instruction neatly)
- The addressing mode for the B field, and
- The value for the B field (this and the previous *two* items are omitted if the instruction does not use the B field).

- Finally, the line may conclude with a comment, which is any sequence of text begun with a semicolon and concluded by the end of the line

Blank lines are permitted, and lines which contain only comments (any text begun with a semicolon) are also permitted.

2.4 Examples

The smallest Core War warrior is the Imp:

```
MOV 0 1
```

This program copies the contents of address zero (which, because of the relative addressing used in redcode, means the current location) to address 1. In other words, it copies itself to the next location. Since this will be the next location to be executed the effect is to attempt to fill memory with copies of the `MOV 0 1` instruction.

A more sophisticated program is Dwarf:

```
ADD #5,  3
MOV #0, @2
JMP -2
DAT    -1
```

which lays down "zero bombs" through memory. Table 1 shows a partial trace of the Dwarf program being executed. Points to note:

- Each other program in the machine has executed a single step between each of the steps this one performs. This trace assumes that none of those instructions modified any of the memory visible in the table.
- using `MOV` to place a zero in a location is equivalent to placing a `DAT 0` instruction there. Any program attempting to execute this statement will be halted.
- The `@` indexing mode allows the value in a location to be used as an indirection (rather like a pointer in C).
- The final `JMP -2` statement means that the program will continue placing `DAT 0` statements every 5 locations in memory. Since the program is less than 5 instructions long it will never overwrite itself as long as the total size of the memory is a factor of 5.

Address	Step 1	Step 2	Step 3	Step 4
1000	ADD #5, 3	ADD #5, 3	ADD #5, 3	ADD #5, 3
1001	MOV #0, @2	MOV #0, @2	MOV #0, @2	MOV #0, @2
1002	JMP -2	JMP -2	JMP -2	JMP -2
1003	DAT -1	DAT 4	DAT 4	DAT 4
1004				
1005			DAT 0	DAT 0

Table 1: Trace of Dwarf being executed