CodeBlue Wars

Introduction

In the late 1960's, Robert Morris, the future National Security Agency (NSA) Chief Scientist, along with Dennis Ritchie and Victor Vyssotsky, created a simple computer game called *Darwin*. Later, *Darwin*, evolved and became *Core War*, a game in which two or more opponents write computer programs in a simple assembly language that execute in memory and try to "kill" their opponents by forcing them to execute an illegal instruction. The programs replicate themselves to move around memory, and try to disrupt the memory space of the other programs. The last program left alive is the winner. If after a set number of iterations, multiple programs remain, it is a tie. We will be writing programs in a simplified version of the language and competing against each other in a visual simulation of memory.

Environment

The "playing field" consists of a linear array of memory locations that can be considered circular. That is, the first memory location follows the last. Each memory location is either a data value or a single instruction. A program consists of a sequence of instructions and data values that reside somewhere in the memory. Multiple programs exist simultaneously in memory. Each competing program has an associated Program Counter (PC) that keeps track of the next instruction to execute for that program.

The game is played by loading at least two of the competing programs into memory at random starting locations such that they do not initially overlap. The game loops by executing one instruction from each program at each instruction cycle, always in the same order. If a program tries to execute an illegal instruction (such as a DATA value), it terminates and is out of the game.

The Language

The original *Core War* used an assembly language called *Redcode* which consisted of eleven simple instructions. We have created an even simpler version, called *CodeBlue*, that only has eight instructions and a modified format to make for easier coding. The basic format of a *CodeBlue* instruction is:

<label>: <instruction> <parameter1>, <parameter2> ; <comment>

where:

a) <label>: and ;<comment> are optional
b) the number of parameters (one or two) depends on the instruction
c) the parts of the instruction are separated with one or more spaces or tabs
d) comments are ignored
e) everything is case-insensitive
f) execution of an illegal instruction (such as a DATA location) halts execution

The six legal instructions are:

Instruction	Parameter(s)	Description
DATA	<value>	Puts <value> at the current memory location
COPY	A, B	Copies the value at location A to location B
ADD	A, B	Adds the value at location A to the value at B and puts the result at B
COMPARE	A, B	Compare values at A and B, if they are unequal, skip the next instruction
JUMP	A	Transfers execution to location A
JUMPZ	A, B	Transfers execution to location A if the value at location B is zero

Thus, a simple looping program might look like:

```
Start:  ADD     #5,Temp     ; increment Temp by 5
        ADD     #-1,Count   ; decrement Count by 1
        JUMPZ   New, Count  ; go somewhere (not shown) if Count is zero
        JUMP    Start       ; else go back to Start
Count:  DATA    10          ; used as a counter control loop variable
Temp:   DATA    0           ; temporary storage location
```

Parameters

Parameters in *CodeBlue* can refer to a literal value (such as the number 5) or a memory location. Syntactically, literals are written with the # sign as shown in the above example. For DATA statements, the parameter is always a literal and the # sign is optional.

References to memory locations are always relative, and may be either *direct* or *indirect*. *Direct* addressing means that the final location is computed by adding the positive or negative offset from the current location. Thus, the following instruction:

```
COPY  3, -2
```

copies the memory location that is three locations past the current location to the memory location that is two previous. Labels are used to make writing programs easier as shown in the code snippet above. Labels get translated to relative offsets when the program is loaded into memory. In the earlier looping example, the “JUMP Start” instruction could have been written as “JUMP -3”. Relative addressing is important because it does not matter where a program is loaded (or moved to) in memory. Using direct relative addressing, the simplest program that “moves itself” through memory is:

```
COPY  0, 1
```

When it executes, the memory location at relative offset 0 (the instruction itself) is copied to relative offset 1 (the next location). Since the Program Counter is incremented after each instruction is executed, the next instruction to be executed will be the one just copied.

Indirect addressing is used to reference memory that another location points to, and is indicated with an "@". The instruction JUMP 7 (direct addressing) is interpreted as transfer control to the memory location 7 past the current location, while the instruction JUMP @7 (indirect addressing) means to transfer control to the relative address *contained in* the memory location 7 past the current location. That is, the data value in the memory location 7 past the current instruction will be added to its location in memory to get the final address. Note that if the indirect offset points to an instruction, versus a DATA value, an error will occur. The following program snippet shows how indirect addressing can be used to "carpet bomb" all of memory by placing 1's in it:

```
Loop:    COPY    #1, @MemPtr ;put a 1 in the memory address at MemPtr
         ADD     #1, MemPtr  ;increment MemPtr to next address
         JUMP    Loop        ;repeat
MemPtr:  DATA    0           ;indirect pointer to memory
```

In the above example, the first time through the loop a one is written to relative offset 0 from MemPtr. The next time through the loop, a one is written to relative offset 1, or the location after MemPtr, etc. If a one is written overtop of a competing program, and it tries to execute it, it will terminate. Eventually, this program will overwrite and terminate itself.

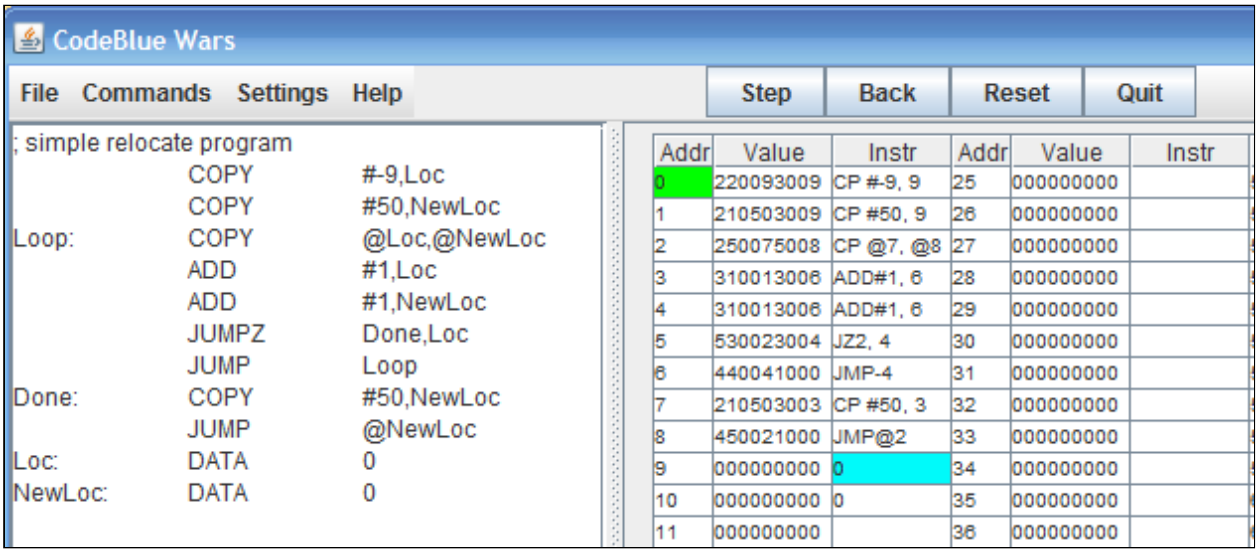
Replication

To terminate other programs, it is necessary to overwrite a portion of them. To keep yourself from being overwritten, you might want to move your program around memory, or replicate it. The following example shows how you can move your entire program to a memory address 50 locations past the end of it. Because memory is circular, this will keep moving until either it gets bombed and terminates or the game ends.

```
         COPY    #-9,Loc       ;first instruction to move is -9 from Loc
         COPY    #50,NewLoc    ;move instructions to NewLoc + 50
Loop:    COPY    @Loc,@NewLoc  ;move one instruction
         ADD     #1,Loc        ;point to next instruction to move
         ADD     #1,NewLoc     ;increment address to move it to
         JUMPZ   Done,Loc      ;if Loc is zero, all instructions moved
         JUMP    Loop          ;if have not moved all, loop back
Done:    COPY    #50,NewLoc    ;reset new location pointer to jump to
         JUMP    @NewLoc       ;transfer control to copy
Loc:     DATA    0             ;pointer to instructions to move
NewLoc:  DATA    0             ;pointer to new location
```

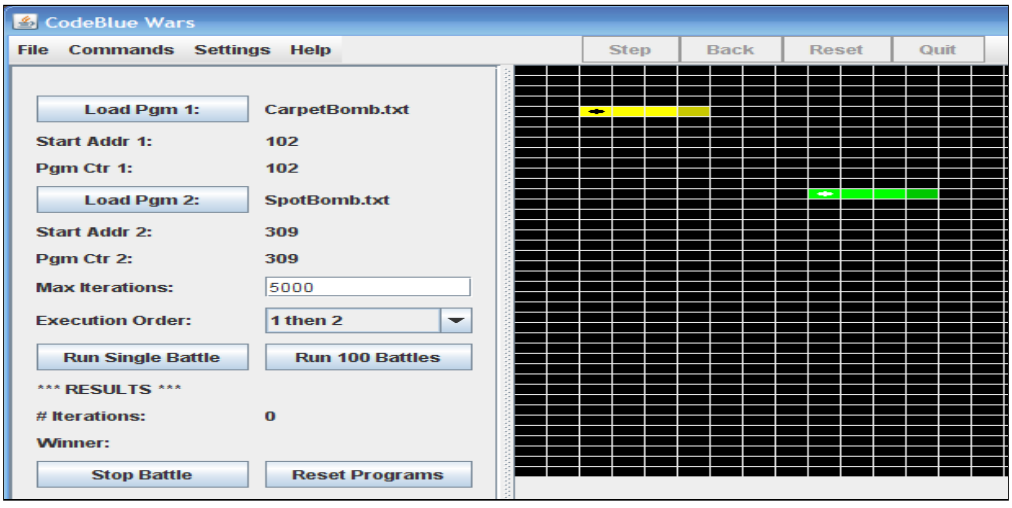
CodeBlue Development Environment / Battles

To help you create and debug your CodeBlue programs, there is a visual environment available, CodeBlue.jar. When you start it, there is a simple text editor pane and a memory pane as shown below.



CodeBlue program files are simple text files that can be loaded, saved, or edited on the screen. To test them, select Commands / Load Memory to put them into the simulated memory on the right. To execute them, use the Commands / Execute option. This activates the Step/Back… buttons and allows you to watch the program execute. The color coding in memory shows the current PC address (green), as well as the source and destination operands for the current instruction. Stepping execution will update memory if anything was changed and move to the next instruction.

Once the program is ready for battle, save it to a file and select Commands / Run Battle. From there you can load two programs into a simulated memory and execute a battle between them as shown below.



Any two programs can be loaded, and are placed in random non-overlapping memory locations. Color coding is used to show both the instructions and data for each program. Each program counter is shown as a black or white dot. The number of iterations for the battle and who gets to go first can be set by the user. When ready, the Run Single Battle button will animate the memory as each instruction executes. When one of the program executes an illegal instruction, the game is over and the other program is declared the winner. If no program quits after the set executions, the game is declared a tie. To play another single battle, simply select the Reset Programs followed by the Run Single Battle. To see the results of 100 battles (without animation), select the Run 100 Battles button. Each program is reloaded at a random location before the start of each battle, and they alternate which program executes first. If one program consistently has more wins than its opponent, it is declared the overall winner between the two.