SIMPLIFIED CORE WAR FOR INTRODUCING LOW-LEVEL CONCEPTS

Dino Schweitzer, David Gibson, Leemon Baird
Department of Computer Science
US Air Force Academy, CO 80840
dino.schweitzer@usafa.edu

ABSTRACT

Assembly language programming is often used to teach students low-level hardware components and computer architecture basics. To introduce some of the basic concepts of program execution in memory, we have developed a very simple language based on Core War, a programming game developed in the 1980's pitting competing programs against each other in a simulated memory. We have built a visual development environment that allows students to create a program, see its representation in memory, step through the program's execution, and simulate a battle of competing programs in a visual memory environment. This paper will describe the language, the environment, the competition simulator, and our experience.

INTRODUCTION

Computer architecture and organization is one of the recognized Body of Knowledge elements in the 2001 ACM Curriculum for undergraduate Computer Science [3]. Core topics within architecture include machine level representation, assembly level machine organization, and memory system organization and architecture. Most undergraduate CS programs teach at least one course in architecture, many of which use assembly language as a tool to introduce low-level concepts.

Teaching assembly language can be a challenge and different approaches have been used to make it easier for students to understand. For example, over the past decade, simulators have been widely used for teaching assembly language concepts in computer architecture [9]. The use of simulators provides many benefits over teaching assembly language programming using real processors as the target machines [8]. An example of a simplified simulator is Silverman, Ackerman, and Chesley's SC123 which has a lean 20-instruction architecture [6]. One of the most widely-used simulators in computer architecture courses is James Larus' SPIM. SPIM implements most of the MIPS32 assembler-extended instruction set [5]. While simulators such as SPIM make it easier to teach concepts of computer architecture concepts, the assembly language programming they support remains inherently tedious and can be unmotivational to many students. Even simple tasks can be long and challenging to create.

An interesting SPIM extension to help motivate students is Craig Zilles' SPIMbot which adds the capability to simulate robots in a virtual world. A survey of undergraduate students learning assembly language programming at the University of Illinois indicates that using SPIMbot and competing in a SPIMbot tournament increased student enjoyment [10].

Some CS educators even question the need to teach assembly language programming [1]. The primary reason for the change is that newer topics such as internet programming and security are crowding out some more traditional CS topics. Despite these new demands, we believe the architectural concepts learned from assembly language programming remain an important element of undergraduate computer science education.

The goal of our work has been to develop and teach assembly language concepts using a visual simulator that students can learn in a single class. We also wanted students to find the

language motivational and fun to use. Our CodeBlue language is much simpler than most simplified architecture instruction sets such as the SC123. Still it allows students to develop interesting assembly level programs that compete in tournaments, similar to the far more complex SPIMbot simulator. Most important, through CodeBlue programming, students learn fundamental computer architecture concepts such as instructions and data co-residence in memory, control structure implementation, and addressing modes.

CORE WAR

The game of Core War was developed by A. K. Dewdney and introduced in two *Scientific American* "Computer Recreations" articles in 1984 and 1985 [2]. The original Core War game was based on a simple assembly-like language called RedCode that had only ten instructions. Multiple RedCode programs can be loaded into a simulated memory and simultaneously executed (one instruction from each program at each iteration of the battle) until one of the programs executes an illegal instruction or after a set number of iterations. Programs attempt to attack each other by writing over their opponent and forcing their opponent to execute illegal instructions.

Core War became popular with programmers, and the International Core War Society (now defunct) was founded in 1985 to promote the game and host competitions [4]. Strategies for beating other RedCode programs include:

- *Replicator* – a program that makes repeated copies of itself and executes them in parallel.
- *Bomber* – blindly copies a "data bomb" at regular intervals trying to hit the opposition and force them to execute a data location.
- *Scanner* – tries to locate the enemy and force it to execute a known instruction.
- *Imp* – a single instruction program that continually copies itself and moves through memory. While hard to kill, it does not kill other programs since it never writes data values that would result in illegal execution. Thus, it can only result in a tie.

CODEBLUE

One of the motivations for using Core War to introduce basic concepts in program execution is its simplicity. The central ideas and simple instructions can be presented and understood in a single lesson. Students can write simple programs and "battle" each other with minimal instruction.

To enhance this simplicity, modifications were made to the original RedCode language. The number of instructions was reduced from ten to four (plus a DATA location). Labels were added to simplify code development and instructions were renamed to make them more self-descriptive than the original 3-character mnemonic. Another design decision was to eliminate the ability to split off separate processes. While forking may be useful in teaching process management, it was removed to maintain simplicity. The modified language is called CodeBlue, to distinguish it from the original RedCode language specification. The complete set of instructions is:

```
DATA    <value>   - <value> set at current location
COPY    A, B      - copies source A to destination B
ADD     A, B      - adds A to B, putting result in B
JUMP    A         - transfer execution to A
JUMPZ   A, B      - if B = 0, transfer to A
```

A simple looping program is shown in Figure 1.

```
; semicolons indicate comments
Start:      ADD     #5, Temp
            ADD     #-1, Counter
            JUMPZ   Done, Counter
            JUMP    Start
Done:       ;other instructions would go here
Temp:       DATA    0
Counter:    DATA    10
```

Figure 1. Simple CodeBlue looping program.

CodeBlue supports three addressing modes: literal, relative, and indirect. Literal values are contained in the instruction and are specified with the '#' character as shown in the first instruction above. Relative addressing is the default addressing mode and represents an offset from the current location. Thus, the instruction *COPY 0, 1* makes a copy of the current instruction (relative offset 0) to the next location (current location + 1). Labels get translated into relative offsets when the program is loaded into memory. Indirect addressing is used for pointers and is signified with the '@' character. The instruction *JUMP @5* would be interpreted as transfer execution to the memory address contained in the location five slots past the location of the current JUMP instruction.

The simplest CodeBlue program is the single line *COPY 0, 1*. When it executes, it copies itself to the next location and the program counter is incremented, thus pointing to the instruction just copied. A more sophisticated program that "carpet bombs" memory is shown in Figure 2.

```
Loop:       COPY    #0, @MemoryPtr
            ADD     #1, MemoryPtr
            JUMP    Loop
MemoryPtr:  DATA    0
```

Figure 2. "Carpet Bombing" program.

DEVELOPMENT ENVIRONMENT

Since students are inexperienced in writing low-level programs, CodeBlue provides a development environment that allows them to edit programs, load them into a simulated memory, and debug them using single-step execution. Figure 3 shows two screen shots of the Code Blue environment. The left screen is the development environment, the right screen is the battle environment.

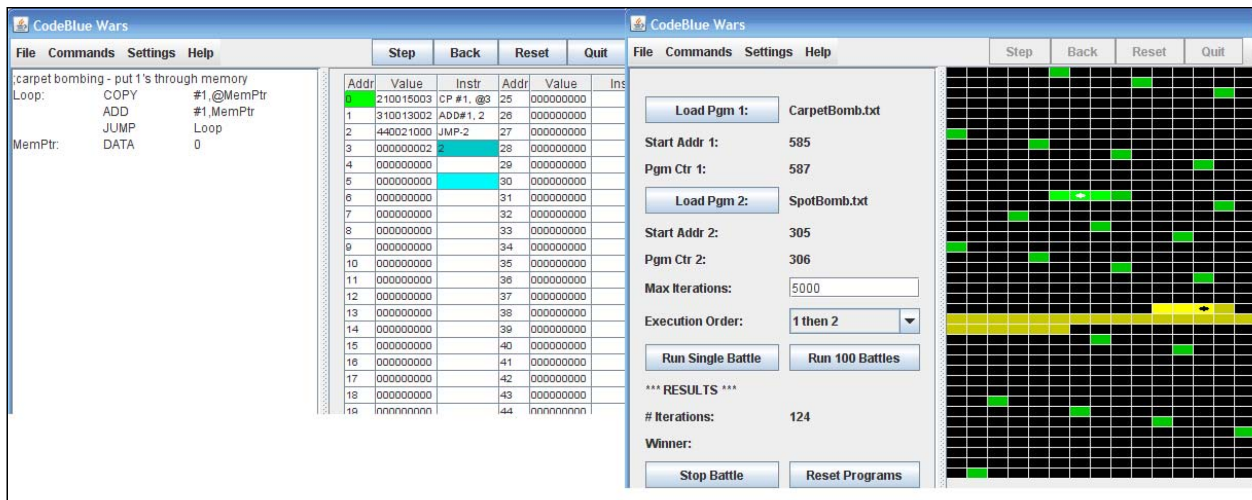

Figure 3. CodeBlue development and battle environment.

The left pane in the development environment is a simple text editor that allows CodeBlue programs to be created, saved, loaded, and edited. When the user is ready to test the program they "load" the program into the simulated memory shown to the right of the editing pane. At this point, the program is parsed, labels are resolved, and any illegal instructions are flagged in the memory with red highlighting. To emphasize to students that data and instructions co-reside in memory, the numeric value for each location is shown along with its "interpretation". Instructors can show the instruction format for the numeric value if they want to demonstrate how memory is interpreted. For DATA locations, the display will simply show the data value. For instructions, the display shows a shortened version of the instruction with the appropriate relative offset for the operands. To help students understand exactly how the addressing works, both indirect and final locations are color-coded for the current instruction. In Figure 3, the current instruction is location 0 as shown with the highlighted address. The source operand, #1, is a literal so no highlighting is shown. The destination operand, *@MemPtr* was translated to *@3* when the label MemPtr was resolved (MemPtr is three locations past the current instruction). Memory shows different highlighting for both memory location 3 (the indirect address) and location 5 (the final destination address).

To debug programs, students can execute the program one step at a time and see any changes to memory as well as the next program counter location. In the above example, a single step will result in a 1 being written in memory location 5, and the Program Counter moving to location 1. If a program tries to execute an illegal instruction (such as a DATA location), it is flagged in red and no further action occurs. At any point the user can "undo" an execution step by using the *Back* button. When the user is satisfied with their program they can save it to an external file for competition in a CodeBlue battle.

BATTLES

Another key factor for using CodeBlue is to motivate students through competition. Similar to the original Core War, two programs are loaded into a simulated memory that is circular (the first location follows the last). The CodeBlue simulator has a 1000-adddress memory. The competition is run by executing one instruction at the current program counter for each program until either a set number of iterations has passed (resulting in a tie), or one of the

programs executes an illegal instruction declaring the other program the winner. The right screen shot in Figure 3 shows a sample battle between a "CarpetBomb" program and "SpotBomb" program. When the programs are loaded, they are placed in random memory locations that guarantee they do not initially overlap. The user can set the number of iterations and which program goes first. If the user chooses to run a single battle, the battle is animated in the displayed memory. Data values and instructions written to memory are color coded to show which program they belong to. In the figure, the horizontal row of shaded memory locations shows the strategy of "carpet bombing" versus the more dispersed spot bombing strategy. The current program counter for each program is shown with the highlighted cell (white or black dot).

Single battle mode is useful to see how the different strategies compare when executed. The final winner of a competition between two programs is determined by the "Run 100 Battles" button that executes 100 different battles, reloading each program into a new random location each time and alternating between which program goes first. Animation is turned off for this mode, and only the final results are displayed.

CLASSROOM USE

CodeBlue was first demonstrated to senior CS students in Fall 2007 to determine how easily they could understand the concepts of the language and begin writing their own competing programs. The overall concepts of the competition and language were presented in a lecture format for about 30 minutes followed by 20 minutes of hands-on experimentation with the tool. Along with the environment, students were provided with simple CodeBlue programs that they could use in battles to see how the program worked. Within minutes, students were editing the sample programs to experiment with various changes to the strategies. They quickly discovered that a more random bombing strategy was much more effective than the more simple "carpet bombing". A full student competition was not held during this initial demonstration of the CodeBlue environment. Minor modifications were made based on their feedback.

In Spring 2008, CodeBlue was used in the Architecture course to introduce students to a simplified assembly language and motivate them to understand some low-level concepts. Once again, the environment was presented in a single lesson, and students were invited to submit their program into a class competition. Although only three entries were submitted, interesting non-transitive results led to further discussion and analysis. That is, program A beat program B which beat program C which beat program A.

These initial results in the classroom suggest the utility of a single feature that constitutes the largest difference between Core War and CodeBlue. In traditional Core War, the language is inherently multithreaded. A single player can have multiple small programs that take turns sharing that player's computational cycles. These small programs can easily spawn even more, with each receiving a proportionally smaller fraction of the cycles. This feature is fundamentally important to the deep strategies that are possible in the game, and is a large part of what makes Core War appealing to experienced assembly language programmers. In contrast, CodeBlue has only a single execution thread. The only way it could perform multithreading is if the player wrote specific code to do so. In that way, CodeBlue more closely resembles a real-world assembly language. The lack of multithreading reduces the depth of strategies that could be explored in the game which would make it less appealing for assembly language experts to participate in a tournament. However, the additional simplicity makes it more accessible to the

novice student. It also makes it more pedagogically useful, since it focuses student attention on those aspects of the language that resemble real-world languages.

When CodeBlue was used in the classroom with students who had very little experience in assembly language, they were immediately able to pick it up. In addition, they showed that the language was still rich enough to achieve its purpose, since they were immediately drawn into modifying and experimenting with the system, and were interested in comparing the various strategies that are possible in it. This experience suggests that a graphically-rich, conceptually-simple system designed for competition does succeed in motivating students to learn more about assembly language.

CONCLUSION

Student reaction to CodeBlue is encouraging. The fact that the language and environment can be learned rapidly and used without taking a lot of valuable lecture time indicate instructors can introduce basic computer architecture concepts quickly and easily. In addition to using it in the Computer Architecture course to introduce low-level concepts, we plan to investigate creating a variation that can be used in the Computer Security course for teaching students the concept of worms. One recommendation by Szor was to create a networked version of Core War that allowed competing programs to travel from machine to machine attacking each other [7]. An advantage to using variations of CodeBlue in other classes is that students would be familiar with it from the Architecture course which would shorten the learning curve even further.

CodeBlue is public domain and available for anyone to download and experiment with. Please contact the authors for further information.

REFERENCES

[1] Agarwal, K. K. and Agarwal, A., Do we need a separate assembly language programming course?, *Journal of Computing Sciences in Colleges,* 19, (4), 246-251, 2004.

[2] A. K. Dewdney, Computer Recreations (Core Wars), *Scientific American*, May, 1984 and March, 1985.

[3] Association of Computing Machinery, Computing Curricula 2001, www.acm.org/education/curricula.html, retrieved September 7, 2007.

[4] Core War, http://en.wikipedia.org/wiki/Core_War, retrieved September 7, 2007.

[5] Larus, J., SPIM: A MIPS32 Simulator, http://pages.cs.wisc.edu/~larus/spim.html, retrieved September 7, 2007.

[6] Silverman, R., Ackerman, A. F., and Chesley, H., A new simulator and IDE for teaching CS220: computer architecture, *Journal of Computing Sciences in Colleges,* 22, (4) 137-144, 2007.

[7] Szor, P., The Art Of Computer Virus Research and Defense, Addison-Wesley, 2005.

[8] Yurcik, W., Special Issue on General Computer Architecture Simulators (Editorial), *Journal on Educational Resources in Computing*, 1, (4), 1-3, 2001.

[9] Yurcik, W., Wolffe, G.S., and Holliday, M. A., A Survey of Simulators Used in Computer Organization/Architecture Courses, *Summer Computer Simulation Conference (SCSC),*Society for Computer Simulation, 2001.

[10] Zilles, C., SPIMbot: an engaging, problem-based approach to teaching assembly language programming, *Proceedings of the 2005 Workshop on Computer Architecture Education*, WCAE '05, ACM Press, New York, 2005.