EE475 Lab #7 Fall 2003

Using a Software Simulator

The objective of this lab is to learn several features of the Zap simulator. Zap is an add-on to the Cosmic tools (Idea CPU12) that allows source-level debugging of embedded software programs for the HC12 processor.

Preliminaries

1. Set up a new CPU12 project. As before, use one of your previous projects as a starting point.
2. Make a temporary local folder for your work: `c:\EEClasses\EE475\tempxxx` .
3. Obtain a copy of the Lab #7 test programs from the course web site.

Exercise #1: Set Up and Use the Simulator

Launch the Idea CPU12 program with your new Lab #7 project. Start with the `lab7_1.c` file from the course web site.

a) Make sure the +debug compiler flag is set: check Tools→Compiler→Options→Miscellaneous→Generate Debug Information.

b) Build the `lab7_1.c` code.

c) You need to tell CPU12 where the Zap simulator is located. This will activate the Zap button on the user interface. Right-click on Tools→Zap Debugger and specify the path (browse to it; something like `c:\zap\zaps12.exe`).

d) Start the simulator by hitting the Zap button on the CPU12 tool bar.

e) Under the Show menu, make sure you have all the following windows open: Disassembly, Monitors, Registers, Stack, and Variables→Brief .

f) Find the "two foot steps" button on the tool bar: press this button to step one C language (high level) expression. The C source window should appear and the line indicator should be at `main()`. The stack window should also indicate `main()` as the current function.

g) Double-click on the `x` on the `x=3;` line in the C source window. A context menu will pop up: choose the `monitor…` line. This places the variable `x` into the monitor window.

h) Now click the "two foot steps" button again, and observe the various display windows.

   1) The C Source window shows the active instruction highlighted in blue (`x=3;`) .

2) Since one statement in C will generally translate into several machine instructions, the assembly instructions attributed to the current C source line are shown in the Disassembly window highlighted in yellow (current assembly instruction is blue). NOTE that you can step through each of the assembly instructions one at a time by clicking the "one foot step" button.

3) Note also that the Variables window now displays the active variables `x` and `y`, and the Monitor window still shows `x`.

i) Go ahead through the rest of the `main()` routine and the function calls using the foot steps buttons while observing what happens in each of the windows. Keep going until you reach the end of the `main()` function. You will have to hit the "stop" button (red light) since, as you may recall, the compiler inserts an infinite loop at the end of the program.

j) In the command window, type `reset`<enter> to reset the simulator. Click the "two foot steps" button to bring up the starting C source line again.

Use the simulator to go through the code carefully: *observe the function call behavior in particular*.

→ **Question #1 (for your memo report):** Comment on the difference between the "good" swap function and the "bogus" swap function. What is different about them? How much code actually was executed when you stepped through each function? Why? In this exercise does the compiler seem to be smart?

Exercise #2: Use the Simulator's Stack Bound Checker

Besides the usefulness of monitoring variables and single-stepping code, the simulator also provides features such as breakpoints, traces, and bounds checking. For this exercise you will observe the stack bounds check.

a) Return to CPU12 and alter the project to now use `lab7_2.c` .

b) Build the `lab7_2.c` code, then open it in the simulator.

c) Single step one high level instruction ("two foot steps" button). This will clear and initialize the stack pointer and get set at the start of the `main()` function.

d) Under the Events menu, click on Stack Alarm. In the pop-up window, click `Status On`, click `Break`, and set the `Stack Bounds`: the high address should be the end of memory (0x7fff) and the low address should be the end of the program and data (see the .bss section and the stack configuration in the linker map from CPU12).

e) Single step one high level instruction several times and watch the display windows. Note that the program uses a recursive function call, and stack space is used on each recursion.

f) Now click the Go button (green light) to start the program running continuously. After a moment the stack alarm should trigger and pop up a warning window.

g) If you want to run the stack check again, be sure to verify that the bounds check is re-enabled via the Events menu (status ON, set to break, and correct stack bounds).

→ **Question #2 (for your memo report):** How many recursions were made before the stack became full? Determine how much stack space is available just prior to the first function call, and how much data is pushed on the stack with each subsequent call. Does a calculation based on the available memory and required stack space match the actual execution results? Explain.

Exercise #3: Try Some Simulator Features

On your own, investigate some of the other simulator features. The documentation is available in help files and in the white lab notebooks.

→ Include a brief summary of your investigations in the memo report.

Lab Report

The lab report is to be written up in the Memo format. Be sure to put the *lab number* in the Memo header along with your name and date. For each exercise, explain what was done, how it was accomplished, and answer the given questions to demonstrate your understanding of the exercise. Include **commented** file excerpts related to each exercise, as appropriate.

→ This lab report is due the beginning of the lab period in one week.