EECS 373 Winter 2004

Lab 2: Introduction to PowerPC Assembly and the SingleStep Debugger

Requirements

Pre-lab: You must set up your software environment on a CAEN workstation and practice using SingleStep on a provided assembly code sample. You must demonstrate your understanding of this code sample by describing its function in pseudocode. See below for details.

In lab: You must demonstrate achievement of specific milestones to your lab instructor as noted in the lab procedure below.

Post-lab: There is no post-lab for lab 2. The lab demonstration sheet is due at the beginning of your lab section during the week of January 12, 2004.

Lab Value: 2% of total grade

Objectives

The purpose of this lab is to provide a basic introduction to the software development and debugging environment you will be using in the EECS 373 lab. Specifically, in this lab you will:

1. Gain familiarity with the PowerPC architecture and assembly language by observing the execution of a PowerPC assembly-language program using the simulator and debugger.
2. Gain familiarity with the SingleStep software environment.
3. Get some experience with debugging and modifying a simple assembly-language program.

Background

The MetaWare Assembler and Linker

The assembler takes your PowerPC assembly program and translates it into machine code, generating an object file. This object file is *relocatable*, i.e., memory locations in the file have not been assigned to specific addresses. The linker has two functions. First, it combines multiple object files and libraries into a single executable, resolving any functions or variables that are cross-referenced between files. Second, it takes the memory locations found in that executable and assigns them to specific addresses so that the program is ready for execution. In EECS 373, we will generally have small programs consisting of only a single object file, so we will be taking advantage of only the latter function.

The SingleStep Debugger

The SingleStep environment allows you to execute and debug PowerPC programs. You can observe a program as it executes in a number of ways—you can run a program without interruption, set breakpoints to stop execution at various strategic points, or step through your program instruction by instruction if you so desire. The software comes in two different flavors. The first is the SingleStep Simulator, which uses a simulated PowerPC processor to execute your code on any Windows machine. This powerful capability allows you to debug most of your code before you ever set foot in the lab. The second is the SingleStep On Chip environment, which uses an actual PowerPC processor (in this case, the MPC823 on your lab board) to execute your program. In this situation, SingleStep uses a special feature of the MPC823 called *on-chip debugging* (also known as *background debug mode*,

or *BDM*) to achieve the same breakpoint and single-stepping functions as in the simulator. With SingleStep On Chip, you can observe and control the software-visible state of your lab board, allowing you to debug your software in conjunction with the actual lab hardware.

Overview

This lab requires you to assemble and link a couple of simple PowerPC assembly programs and execute them on the SingleStep simulator. In the pre-lab, you will set up your environment, learn about basic SingleStep features, and learn the operation of a program by observing its execution under SingleStep. In the lab, you will make a series of minor changes to this program and observe the effects of those changes. You will then be given a second program will have some small errors that you will be required to find and debug.

Pre-lab assignment

Step 1. Setting up the necessary files

You will need to do this assignment on a CAEN XP machine. The files you will need for the pre-lab are in the NTFS directory \\caen-mahogany\dfscourse\perm\eecs373\lab2. In your class directory, create a subdirectory called LAB2 and copy the files LAB2.s and LAB2-lnk.txt from the perm directory to your subdirectory. Now, in that same subdirectory, create a batch file called ASM.bat which contains the following two lines:

```
asppc -c -g -l %1.s > %1.lst
ldppcl -dn -A %1-lnk.txt -o %1.elf %1.o
```

Note: The only places where the number one appears is directly after percent signs for example, %1.o. All other characters with similar appearances are lowercase L's.

You will use this batch file (or variants on this file) to assemble and link your programs throughout the semester. Note that the batch file can take arguments; the “%1” will be replaced by the first argument when the file is executed. If you wanted to include a second argument, you would denote it as “%2” within the batch file. The first line invokes the assembler (asppc) to assemble the source (.s) file into a relocatable object file (the .o file). It also generates a listing (.lst) file which SingleStep makes use of when displaying your program. The second line calls the linker (ldppcl) to fix the addresses of the relocatable object to generate a loadable .elf file—ELF, the Executable and Linking Format, is the object file format used by all of our software tools. The addresses for a given program’s code and data section are defined in the lnk.txt file. Look at the LAB2-lnk.txt file to see the addresses used in this program.

Now open a DOS command prompt window and go to your class directory. Type the following line at the prompt:

```
path s:\caen\sds75\hcppc\bin;%path%
```

Doing this adds the MetaWare tools directory—the location where the assembler and linker reside—to your search path. This allows you to run the assembler and linker and create the ELF file simply by executing your batch file by typing:

```
ASM LAB2
```

Note, if you are in the lab, the path is already set to:

```
c:\sds75\hcppc\bin;
```

Remember, you need to be in the asm.bat directory to run it.

Step 2. Setting up SingleStep

At this point, you are ready to run your code using the SingleStep simulator. The simulator can be found on a CAEN Windows machine in the Start menu under:

Engineering Applications > SingleStep 7.5 > SingleStep Simulator (PowerPC)

Once you have started the simulator, a dialog box will appear. Ensure that the following settings are correct in the tabs of the dialog:

Tab Name	**Correct Settings**
File	<your class dir>\LAB2.elf
Target Configuration	Processor: MPC821
Options	"Execute until 'main'" unchecked
Logging	Log to screen always

Select the OK button when all settings are correct. The software will now reset the simulated target board, download the ELF file into the target's memory, and initialize the PC to the first instruction of the program, which is the _start label by default.

Step 3. Simple Program Execution

The processor is now stopped at the first program instruction. The main Debug window shows your code in the left pane, with a green arrow pointing to the instruction at the current PC. The stack contents are in the upper right pane, and the registers are in the lower right pane. Re-size these panes to show more of the registers, since we don't use the stack in this lab.

Note that there are three small buttons labeled "S", "D", and "M" in the lower left corner of the Debug window. These buttons set the display mode of the left pane to source, disassembly, and mixed mode, respectively. Source mode shows only your original source file. Disassembly mode reads the actual machine code out of the target memory and converts it back to assembly code—in other words, disassembles it. You may notice that the disassembled code differs slightly from the original assembly code, even though it represents the same program. This discrepancy is due to the existence of simplified mnemonics—source mode uses them, disassembly mode doesn't—and the fact that the disassembler is incapable of converting constants back to symbols (e.g. "array" or "elsize"). Mixed mode shows both the original source and the disassembly, appropriately interleaved. This mode is useful if your source code is in a higher-level language, such as C. For this lab, we recommend using source mode.

To execute the first instruction, click on the "Step Over" button. Note that the instruction changes the value of R3, and that the new value is reflected in the register window, colored in red to denote a recent change. Execute the next instruction in the same manner, and see that R3 now has the full 32-bit address of the data array. Note that this address matches the address given in the LAB2-lnk.txt file for the top of the data section. Also note that this two-instruction sequence loads a 32-bit address of a memory location into a general-purpose register—you will need to perform this exact operation frequently throughout the semester.

At this point, you should be familiar enough with the assembly code to predict the outcome of each instruction's execution. Do so for each instruction up to the *lwz* instruction, referencing the register window to verify your predictions. In order to predict the effect of the *lwz* instruction, you must be able to examine memory. Clicking on the memory chip icon will bring up the memory display window. The small box in the upper-left corner of this window is the address entry box; entering an address in this box allows you to view

memory starting at the specified address. The box to its right is the data entry box, which allows you to set values in memory so that you can verify the operation of load and store instructions. Double-click the address entry box and enter the address in register R3—the base address of the array. The memory window should now display the contents of memory at this address.

To make the display format match the format used in the assembly file's .word declaration, you must modify the window's display properties. Right-click in the window to bring up the "Properties" dialog, and select "Word," "Decimal," and "Signed" to see the array in the desired format. Note that you can also specify the number of bytes per line displayed. Take a minute to try several different display formats and familiarize yourself with the various options.

You should now be able to predict the effect of all the instructions in the loop. Step through the *lwz* instruction, checking to see that it behaves as you expected, and continue stepping until you reach the *stw* instruction. When you step through this instruction, note that one of the values in memory changes and that, like the register window, the memory window shows changed values in red. Continue stepping through the rest of the first loop iteration and onto the next iterations.

Step 4. Using Breakpoints

Obviously, stepping through code one instruction at a time is an incredibly detailed yet painfully slow method of observing the program's execution. If you don't need quite as much detail and want to speed up the process, you can use breakpoints to specify points in the code at which you wish to stop execution and observe the program's progress.

To set a breakpoint, first make sure that you're in source mode by verifying that the "S" button in the lower left corner of the Debug window has been pressed. Then, double-click in the margin to the left of the instruction at which you wish to set the breakpoint. For the purposes of this exercise, set a breakpoint at the "bne loop2" instruction. A small red flag should appear there to indicate that your breakpoint has been set. Note that the debugger has already automatically placed one breakpoint at the exit label; it does so to ensure that the program does not enter an infinite loop when it finishes executing.

To verify the location of your breakpoint, click on the Breakpoint button—the one with the red flag—in the toolbar. The window which appears lists all current breakpoints. If there is a check in the box next to the breakpoint, then it is active and will halt the processor when execution reaches that address.

Click on the "Go" button—the one with the green traffic light. This button causes the processor to run freely from the current PC, stopping only when it reaches the breakpoint you just set. Once the program stops, note that the register and memory windows display several red values, indicating that all of these values have changed since the last time the processor was stopped. If you continue clicking on the "Go" button, you can observe the execution of the program one loop iteration at a time. Use this technique to determine what the program is doing.

There are several different ways to delete breakpoints. You can bring up the Breakpoint window, select the desired breakpoint, and press the Delete key (or right-click and select Delete). You can also place the cursor on the instruction in the Debug window and press F9, which toggles the breakpoint status of that instruction. In source display mode, you can right-click on the red flag symbol and select Toggle Breakpoint, or click on the symbol and select Delete from the dialog box. Use one of these techniques now to delete the breakpoint you set earlier. You can now run the program to completion by clicking on the Go button; when the code reaches the exit label, it should stop at the breakpoint there. If you inadvertently deleted this breakpoint earlier, the processor will loop infinitely, which you can terminate by clicking on the Stop button—the one with the red traffic light.

Step 5. Viewing Program Variables

The Watch window, which is placed at the bottom of the screen, allows you to view program variables—like the array in our LAB2 program—in symbolic form. To add a variable to the Watch window, right click in the window and select Add. In the Name field, type the name of the variable you wish to view. In the Change Type To field, list the appropriate type as you would declare it in C (e.g., for an integer array of ten elements, int[10]). Use the Format field to set the display format for the variables. Once all these properties are set, clicking OK will add the variable to the Watch window. To expand the display for a variable, click on the plus sign to the left of the entry.

At this point, you should rerun the program while observing the values of the array variables in the Watch window. To restart the debug session (and therefore the program), type CTRL+D or select Debug from the File menu in the SingleStep window. Add the variable "array" to the Watch window as described above. Step through one iteration of the program loop one instruction at a time, then run each full loop iteration one at a time by using breakpoints as before. Observe the data changes in the Watch window. Unlike the memory window, you can show each value in multiple formats simultaneously.

The following table lists the main watch variable types you will need. Enter the value in "Change type to" field.

Watch Variable Type	Description	Number of Bytes
long	word	4
short	Half word	2
char	byte	1
type[n]	Array of longs, shorts or chars of size n	n*type

Step 6. Program Functionality

Now, using SingleStep, step through the program and study the functionality. Use pseudo code to document the flow and program functions. The pseudo code can take the form of a higher level language such as C without syntactical correctness. For example, a condition statement might take the form:

```
if (x > 0) then
   a[y++] = j
else
   go to exit
```

where:

- x and y represent general purpose registers or perhaps memory locations.
- if (x>0) represents a series of test and branch assembler instructions.
- [y++] represents incrementing a register contents
- a[y++] represents indexing to a memory location with a base address of a[0].

You do not need to include assembler specifics, such as register use. The pseudo code represents program functionality only, without the specifics of assembler implementation.

Pre-lab deliverable 1: Turn in the pseudo code at the beginning of lab.

In-Lab Assignment

For the in-lab portion of this assignment you will make some minor changes to the code you used for the pre-lab and try to predict what will happen when you make the changes. You will also be given a file by your TA in lab that you will have to debug and correct.

Part I: Modifying the program operation

Using your favorite text editor, open the assembly source file LAB2.s. Find the line marked with the "<<==" in the comment. Change the opcode for this instruction from "ble" to "bge". You should be able to predict the program's operation after this change, but in order to verify that you are correct, you must first create a new ELF file be rerunning your batch file. If you get error messages, look in the listing (.lst) file for details. Start a new debugging session by pressing CTRL+D or selecting File > Debug from the SingleStep window. The target will reset and the new version of your program will be downloaded. If you still have the array displayed in your Watch window, you will have to reset the type information to "int[10]" by right-clicking on the variable and selecting Properties, because the empty type information in the ELF file overwrites the type information you entered earlier. Run the program and verify that it operated as you suspected.

Demonstration 1.1: Demonstrate to the lab instructor the program runs correctly after changing "ble" to "bge"**.**

Currently, the program treats the contents of the array as signed integers. With a few simple modifications, you can get the program to treat the array values as unsigned integers (and you don't need to change the array initialization constants to do this). Make these changes and run the program again, using the memory window to verify the program's operation. If your first attempt is unsuccessful, use the debugger to fix it.

Demonstration 1.2: Demonstrate to the lab instructor that the program correctly treats the array values as unsigned integers.

Now, modify the program to sort an array of signed halfwords. Use .half to declare initialized halfword values. Debug and verify its operation.

Demonstration 1.3: Demonstrate to the lab instructor that the program sorts the array as halfword values.

Part II: File Debugging

When you have completed the modifications section, your TA will give you a disk with some code that contains errors. You should fix the code so that it compiles and so that the assembly code properly performs what the comments say it should be doing.

Demonstration 1.4: Demonstrate the working program to the lab instructor and identify and explain each bug.

Lab 2 Demonstration Sheet

Print this page and present it to your lab instructor when demonstrating the various lab sections. Turn this sheet in with your post lab or when your in lab demonstration is due. You are required to turn in only one demonstration sheet per group.

List Partners Names

_____

_____

_____

Part 1: Modifying the Program Operation

D1.1 The program runs correctly after changing “ble” to “bge”

Lab instructor’s initials**:**

D1.2 The program correctly treats the array values as unsigned integers

Lab instructor’s initials**:**

D1.3 The program sorts the array as halfword values.

Lab instructor’s initials**:**

Part 2: Debugging

D1.4 Identify bugs and demonstrate working program.

Lab instructor’s initials**:**