CS/ECE 347, Spring 1998
Homework 3: Alpha Assembly Language

Assigned: Thursday, February 19
Due: Thursday, February 26, 3:00pm

Introduction and Logistics

The purpose of this assignment is to get hands-on experience with machine-level programming. Our interest is in being able to understand, measure, and optimize the machine code generated by a compiler. This is a far more useful skill than being able to churn out pages of assembly code by hand.

WARNING: There are a lot of logistical skills you'll need to master for this assignment. You will want to refer to the following instructions frequently.

The files you will need for this assignment are in the directory

```
/afs/cs.cmu.edu/academic/class/15347-s98/public/asst/h3
```

Parts of this assignment involve compiling, disassembling, and running code on one of our Alpha-based machines. For information on how to access these machines, please refer to the link labeled "Information about our Alpha systems" on the class WWW page.

The `Makefile` contains rules for generating the object code (`.o`) files that you need for this assignment. You can safely ignore the warning messages printed for the files given to you.

You can examine object code by disassembling it with the program `dis`. For example, to disassemble the file `code.o` and put the result in the file `code.dis`, you give the command:

```
dis -h code.o > code.dis
```

For the remainder of this discussion, when we ask you questions about object code, you should answer these by examining the disassembled version.

When asked to generate an "annotated" version of the disassembled code, you should follow the conventions used in lecture:

- Put comments at the top of a code segment describing register usage and initial conditions.
- Put comments along the right hand side describing what each instruction does.

NOTE: Comments of the form:

```
# I won't tell you anything about the registers.
s8addq  r1, r2, r2    # r2 = 8*r1 + r2
ldq     r3, 0(r2)     # r3 = Mem[r2]
```

are useless and will receive little (if any) credit. Instead, we would like to see comments like the following:

```
# Throughout the loop: r1 holds i, r7 holds n
# At the beginning of the loop: r2 = &v[0]
s8addq  r1, r2, r2    # r2 = 8*i + &v[0]
ldq     r3, 0(r2)     # r3 = v[i]
```

In other words, your comments should convey semantic information from the source code, and not simply reiterate what would be obvious to anyone who can read Alpha assembly code.

When you report the performance of a routine, express your answer in units of clock cycles. You will need to use the `ftime` routine you wrote for assignment H1, and you can use the program `freq` to determine the clock rate of your processor. (FYI, the clock rate of our Alphas is 433 MHz.) You will need to write your own code to run `ftime` on the given test routines.

Hand In

You should turn in a printed report. Here are some guidelines:

- Generate a single, typeset document, not a hodgepodge of code listings and handwritten scrawl.
- Start the answer to each problem on a new page.
- Clearly indicate the members of your group. (One report/group).
- Don't include massive code dumps. Just provide the segments of code relevant to the particular discussion.
- Much of the data requested would best be presented in tables.

Examining and Measuring Code

Consider the following C procedure for swapping two consecutive elements within an array:

```
void swapC(int v[], int k)
{
  int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

This code is included in the file `code2.c` (along with several other routines that we will examine later in this assignment).

The file `asm.s` contains a (somewhat naive) hand-coded implementation of the swap routine written in Alpha assembly language. This latter version of the swap routine is called "`swapA`".

Problem 1: Understanding Swap Routines

Use the `Makefile` to compile `code2.c` and `asm.s` into the object files `code2.o` and `asm.o`, respectively. Then disassemble these object files using the `dis` command.

A. Generate an annotated version of the disassembled code for `swapC`.

B. Compare the object code generated for `swapC` to that generated for `swapA`. What allowable optimizations did the C compiler take advantage of that were not exploited by the author of `swapA`?

C. How many instructions are required by the object code routines `swapA` and `swapC`? Call these values A and C, respectively.

Problem 2: Measuring Swap Routines

The file `tests.c` contains three test routines: `swapAtest`, `swapCtest`, and `swapDtest`. In conjunction with the `ftime` routine, you can use these tests to measure the performance of routines `swapA`, `swapC`, and `swapDummy`, respectively. Procedure `swapDummy` is a "dummy" used to subtracting out the overhead caused by the rest of the test code.

A. Examine the disassembled code for `swapDummy`. How many instructions does it require? Call this value D.

B. Measure and report the times taken by the three test routines, expressed as the average number of clock cycles per call to the respective swap or dummy routine. Call the three results a, c, and d, respectively. What are the values of $a - d$ and $c - d$, indicating the time difference (in cycles) between `swapA` and `swapC`?

C. If we assume each that Alpha instruction requires one clock cycle to execute (this is not quite true in reality, of course), then our measurements of $a - d$ and $c - d$ should match the instruction counts $A - D$ and $C - D$. How do your measurements compare to these estimates? [Do not be alarmed if they are somewhat off.]

Problem 3: Understanding Loop Code

The functions `bubbleUp1`, `bubbleUp2`, and `bubbleUp3` in the file `code2.c` each perform the same task. They shift the elements of an array down by one, moving the first element to the end. Procedure `bubbleUp1` does this by stepping through the array and calling the procedure `swapC`. Procedure `bubbleUp2` does the same, but with a macro-expanded version of `swapC`. Procedure `bubbleUp3` more directly implements the specified behavior, and contains low-level optimizations using pointer code.

Examine the disassembled version of `code2.o` to see how the C compiler has translated each of these routines into Alpha machine code. Our main interest is in the operation and performance of the inner loops.

A. Identify the inner loops in the disassembled code for these three routines. Generate annotated versions of these loops.

B. Describe qualitatively the savings made by using the macro-expanded version of swap compared to the procedure version.

C. Describe qualitatively the further savings made by the direct implementation (`bubble3`), versus a swap-based version (`bubble2`).

D. How many instructions are required by each of these loops?

Problem 4: Measuring Loop Code

Use the routines `b1test`, `b2test`, and `b3test` in the file `tests.c` to measure the performance of the three bubble routines. Express the performance in terms of the average number of clock cycles per execution of the inner loop of the bubble routine. How do these results compare to each other, and to the number of instructions in each loop? Express your result as a table with 6 entries.

Long Jumps

(NOTE: this last section is for extra credit only.)

In this section, you will enhance your understanding of Alpha calling and returning conventions by examining the Unix `setjmp` library, which in effect cheats on the standard procedure calling conventions.

`setjmp` provides a mechanism for nonlocal procedure exits. By using `setjmp` and `longjmp`, a procedure can exit to a function a long way up the procedure chain, bypassing its calling procedure.

This is typically used for error handling, in the following style:

```
jmp_buf env;

main()
{
   int jval;
   if ((jval = setjmp(env)) != 0) {
     /* Do error recovery and cleanup. */
     /* And possibly try to execute again. */
   } else {
     /* Do normal case. */
     ...
   }
}
```

The first time `setjmp()` is called, it sets up the buffer env with enough information to reconstruct the state of the registers and the stack. It returns 0 to the calling function.

The calling function then does other code, and deep within some complicated hierarchy of other functions, it may find some error:

```
deep_within_some_complicated_hierarchy ()
{
  if (error_detected) {
     longjmp(env, ERROR_CONSTANT);
  }
  ...
}
```

`longjmp()` restores the state that was saved in `setjmp()` and resets the stack pointer to the position it was in when `setjmp()` was called.

The file `ljdemo.c` contains a somewhat contrived example of the use of `setjmp()` and `longjmp()`. Study this code to better understand the behavior of these constructs. Try compiling and running it (it should work on any Unix machine).

Problem 5: *(For Extra Credit)* Understanding the Implementation of Long Jumps

The file `longjmp.dis` contains a disassembly of the code for ____`setjmp()` and `_longjmp()`, which are simplified versions of `setjmp()` and `longjmp()`. (Note that `__longjump_resume()`, which is included in this file, is called by `_longjmp()`) This code was extracted from the Unix library `/usr/lib/libc.a` on one of our Alpha machines. Your job is to create annotated versions of these two procedures. Show clearly what state is being saved and restored, and how longjmp subverts the Alpha calling routines to appear to 'return' to a place other than that from which it was called.

It may help you understand the code to write down what data is stored in which places in the `jmp_buf`. If you write this information down, turn that in too.

Things You Need To Know

This section covers information that you are not expected to know from previous lectures.

The `ldt` and `stt` instructions load and store 64-bit numbers into/from the *floating-point* register file. Their behavior is analogous to `ldq` and `stq` for integer data.

The value `0xacedbade` is used as a "magic token" to identify a buffer that has (probably) been set by `setjmp()` (A moment's thought should convince you that doing a `longjmp()` to a buffer that has not been set up by `setjmp()` is likely to result in errors that are very difficult to track down.) This value is generated by a combination of three instructions:

```
ldah r1, 22135(r31)
lda  r1, -8849(r1)
addq r1, r1, r1
```

The `ldah` instruction, which stands for Load Address High, multiplies a signed 16-bit constant by 65536, and then adds this value into the destination register. The `lda` instruction, which stands for Load Address, generates the effective address address in the destination register (in this case, it subtracts 8849 from `r1`).