CS 740, Fall 1995
Homework Assignment 1

Assigned: Sept. 18
Due: Thurs., Sept. 28, 1:00PM

Policy

You may work in a group of up to 3 people in solving the problems for this assignment. However, you must prepare your own writeup, and you must identify your collaborators in your writeup.

Logistics

Any clarifications and revisions to the assignment will be posted on Web page `assigns.html` in the class WWW directory.

In the following, *HOMEDIR* refers to the directory:

`/afs/cs.cmu.edu/academic/class/15740-f95/public`

and *ASSTDIR* refers to the directory *HOMEDIR*`/asst/asst1`.

You only need to hand in a hard copy document for this assignment. Formatted text is preferred to hand written. If some question arises during grading regarding the correctness of your code, you may be asked to submit an electronic version.

Introduction

The purpose of this assignment is to practice reasoning about bit-level arithmetic, low-level code optimization, and procedure calling conventions. Some important techniques for writing efficient code include looking at the machine code generated by an optimizing compiler, disassembling library routines, and timing different implementations of a function.

In the following discussion, the value W standards for the word size of the machine you are using. For most machines, including the MIPS architecture, $W = 32$. A notable exception are DEC Alpha's, for which $W = 64$. The most straightforward way to do this assignment is to use a MIPS machine, e.g., a DECSTATION. If you want to be adventurous, go ahead and try another machine, but be sure to indicate what kind of machine you used.

1. Warm Up

Unsigned arithmetic in C is really performed as modular arithmetic, with a modulus of 2^W. In some applications, we want to compute the correct (nonmodular) results for all possible arguments.

Suppose we wish to add two unsigned integers a and b, for a computer with word size W. Their sum may require $W + 1$ bits. We want to efficiently detect this case.

Task 1A:

Suppose that for values a, b, and K, such that $0 \leq a, b < K$, we let $s = a + b \bmod K$. Prove that one of the two must hold:

1. Both $s \geq a$ and $s \geq b$, in which case $s = a + b$.
2. Both $s < a$ and $s < b$, in which case $s = a + b - K$.

Task 1B:

Write an efficient C Macro `ADD(Carry, Sum, a, b)` where `Carry` and `Sum` denote locations (lvalues) for two unsigned integers. We want to set these values such that $2^W \cdot$`Carry`$+$`Sum` $=$ `a`$+$`b`.

Using the result of Task 1A, this can be done by code that translates into two machine code instructions, assuming all of the operands are in registers. You may want to write some C code using your macro and generate the assembly code with `gcc -O -S` to see how efficient your implementation is.

2. Exact Multiplication

The product of two W-bit unsigned integers may require $2W$ bits. When you program in C, the high order W bits are thrown away. For this problem you will implement multiply routines having the following declaration:

```
void mult_XXX(unsigned a, unsigned b, unsigned *hip, unsigned *lop)
```

which have the effect of storing the two word product in locations `*hip`, and `*lop`. These routines must work for all possible operands, and they should be efficient.

You can test out your code using the data in the file *ASSTDIR*`/m100.test`. Each line of this file has four numbers, written in hexadecimal, giving the two arguments and the two product words of a multiplication.

Task 2A:

Implement a multiplication routine `mult_shift` based on shifting and adding.

Task 2B:

Implement a multiplication routine `mult_half` that splits the arguments into half words, performs 4 half word multiplications, and combines the results into a single product. Watch carefully for all of the overflows!

Task 2C:

The MIPS architecture performs unsigned multiplication using the instruction `multu`. It stores the results in two special registers, `HI` and `LO`. There are special instructions `mfhi` and `mflo` that copy these register values into ordinary registers.

If you're willing to do a little assembly code hacking, you can therefore compute the two word product with a single multiply. The easiest way to do this is to write some prototype C code, compile it with `gcc -O -S` to get an assembly code file (with extension "`.s`)", and then manually edit this assembly code. When you run `gcc` on the editted `.s`, it will assemble your version of the routine. Implement a multiplication routine `mult_asm` based on this approach.

If you wish to use a machine other than a MIPS, you will need to find what hardware support your machine provides for multiplication and use it to generate the most efficient multiplication routine you can. Thus far, we have been able to implement versions for MIPS, SPARC, and VAX architectures using a combination of looking at assembly code generated from prototypes, using reference manuals, and some guesswork.

3. Performance Evaluation

Compare the running times of your different implementations. You will find the "ftime" code useful for this task. This file is documented in the file *HOMEDIR*`/include/ftime.h` and the object code is available as part of the library file: *HOMEDIR*`/lib/lib740.a`.

Thus far, the code has been compiled for several CPU types. To see whether it has been installed for your CPU, do the following:

1. Connect (`cd`) to directory *HOMEDIR*.
2. Execute the command: `ls .bin/`` `sys` `` (note the backquotes).

Either the command will reply with a listing of files, or with an error message saying the subdirectory for your machine does not exist. If you cannot find the object code, feel free to add your CPU to the installation. Consult the comments in *HOMEDIR*`/Makefile` to see how to do this.

This timing code computes the running time for a function in microseconds with a fairly high degree of accuracy. It can also deduct the function call overhead by subtracting the time taken by a supplied "dummy" function. An example program using this timing facility can be found in *HOMEDIR*`/src/clock-rate.c`

Task 3

Use this code to measure the performance of your functions running on a representative sample of different test cases, e.g., the first ten test cases from the list supplied to you.

Describe your findings for the different implementations. Be sure to document which CPU (architecture, model number, clock rate) you use. Express CPU times both in absolute terms (e.g., microseconds) and in clock cycles.

To find the clock speed of your processor, try running the program *HOMEDIR*`/bin/clock-rate`. Again, you may need to compile this program for your CPU type. This program computes the approximate rate of your processor, measured in how many additions it can perform per microsecond. Run it several times to avoid hitting an anomalous case.

4. Procedure Calling and Returning

The MIPS architecture defines a set of conventions for procedure calling/returning including register usage, argument passing, and stack maintenance. These are documented in Appendix D of Kane and Heinrich, *MIPS RISC Architecture*, which is on reserve in the Engineering Library.

An interesting extension of the conventional procedure call/return mechanism is provided by the Unix "setjmp" library. This library provides a mechanism for nonlocal procedure exits, useful for such tasks as error handling. This library is documented in the Unix man page setjmp.3, a postscript version of which is provided in the file *ASSTDIR*`/setjmp.3.ps`. The file *ASSTDIR*`/longjmp.c` shows a (very contrived) example of how long jumps work. Study both the man page and this code—it may seem very nonintuitive at first. Note also that this program uses the simplified versions `_setjmp` and `_longjmp`, which avoid dealing with Unix signals.

Task 4

The file *ASSTDIR*`/longjmp.mips.s` shows the MIPS assembly code for this program. As can be seen, `_setjmp` and `_longjmp` are implemented as procedures (some architectures define them as macros). The file *ASSTDIR*`/longjmp.dmp` shows diassembled versions of the two procedures.

Annotate these disassembled procedures to describe how long jumps are implemented in MIPS. You should describe how and what state is being saved and restored, how control is passed, and how the program stack is affected.