CS 740, Fall 1996
Homework Assignment 1

Assigned: Sept. 17
Due: Thurs., Sept. 26, 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-f96/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.

For this assignment, you will want to use a machine with a MIPS processor. Candidates include the DECSTATION workstations in the CS Department and in the Andrew clusters, some of the general purpose machines in CS, as well as the Andrew servers `unix1.andrew`, etc. You can tell if it's the right kind of machine by executing the command '`sys`'. This will respond with '`pmax_mach`' in CS, or '`pmax_ul4`' on Andrew. Alternatively, you could use an SGI graphics workstation.

Understanding Multiplication

Signed Integers

Let W denote the word size of the computer, e.g., $W = 32$ for a MIPS machine. The product of two W-bit integers may require $2W$ bits. When you program in C, the high order W bits are thrown away. For example, the C function

```
void mult_lo(int a, int b, int *out)
{
  *out = a * b;
}
```

generates the following MIPS code [shown using our convention that upper case branch/jump instructions denote delayed branches]:

```
mult_lo:
        mult    $4,$5       # Initiate multiplication
        mflo    $4          # Grab low order word of product
        J       $31         # Return
        sw      $4,0($6)    # Storing word
```

Observe that this code uses the MIPS `mult` instruction to initiate a (signed) multiplication of the two arguments. This instruction deposits its results in two special registers: `LO` and `HI`. The `mflo` instruction then moves the low order word into the specified register which is used as the source of the final store instruction.

Suppose we wish to get the full, double word product from multiplication, e.g., to implement a function

```
void mult_full(int a, int b, int *out)
```

which stores the product in locations `out[0]` (low order), and `out[1]` (high order).

The GCC compiler lets you insert arbitrary lines of assembly code into a program using the `asm` directive. So, we can just modify the above code to include a line that grabs the high order word and put it in a known register:

```
void mult_full(int a, int b, int *out)
{
  int lo = a * b;
  asm ("mfhi $5");
  out[1] = b;
  out[0] = lo;
}
```

When compiled, this generates code:

```
mult_full:
        mult    $4,$5    # Initiate multiplication
        mflo    $4       # Grab low order word
        mfhi    $5       # Grab high order word
        sw      $5,4($6) # Store high word
        J       $31      # Return
        sw      $4,0($6) # Storing low word
```

which works just fine. [Note that this code will only compile correctly on a MIPS machine, using GCC with the -O flag set.]

Task 1

Under what conditions is it safe to ignore the high order word of a signed multiply? Write (efficient) code for a routine:

```
int test_mult(int a, int b, int *out)
```

which returns 1 if the single integer word `*out` is indeed the product of integers `a` and `b`.

Unsigned Integers

Now suppose we want to do the same thing, but for *unsigned* values, e.g., with the code

```
void try_multu_full(unsigned a, unsigned b, unsigned *out)
{
  unsigned lo = a * b;
  asm ("mfhi $5");
  out[1] = b;
  out[0] = lo;
}
```

When we compile this code, we discover that GCC generates the exact same code as for the signed case! Even though the MIPS architecture includes a special instruction `multu` to perform unsigned multiplication, the compiler writers chose not to use it. This makes one wonder how the two forms of multiplication are related, and why GCC can safely substitute one for the other.

Here's a useful way to reason about signed vs. unsigned arithmetic. Suppose we split a word into its high order bit s and the remaining bits having an (unsigned) value v. Then this word has an unsigned value $u = 2^{W-1}s + v$, and a signed value (on a two's complement machine) $t = -2^{W-1}s + v$. From this, we can see that $t = u - 2^W s$.

Task 2A

Suppose we use the `mult` instruction to multiply two unsigned words u_1 and u_2. Derive an expression showing the unsigned value the resulting `HI` and `LO` registers would represent in terms of u_1, u_2, and their high order bits s_1 and s_2.

Task 2B

Based on this expression, show that the low order word will be the same for either choice of multiply instruction.

Task 2C

Describe how you could correct the high order word resulting from a `mult` instruction to match what would result from a `multu` instruction.

Task 3

Write two routines to perform full precision, unsigned multiplication. The first should use the `multu` instruction, while the second should use the `mult` instruction plus corrections.

You can test 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 (high and low) of a multiplication.

Pipeline Behavior

Interestingly, it is possible to implement unsigned multiply using the `mult` instruction with about the same performance as using the `multu` instruction. The trick is to exploit the rather lengthy latency of the multiply instructions, executing most of the correction code within the delay between the start of `mult` and the completion of `mflo`.

First, we need to get a better understanding of instruction timings and pipeline behavior.

Task 4A

Determine how many clock cycles are required to execute the function `mult_full` shown earlier. You should count all of the cycles taken by instructions in the compiled function, including the final return and its delay slot. You should not include the cycles used to set up the arguments or make the call.

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`. 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.

Task 4B

Every instruction except the `mult` in this code should take one clock cycle. From this, one can see that integer multiply is fairly slow. The CPU designers anticipated this by making it possible to execute instructions after `mult` but before either `mflo` or `mfhi` while the product is being computed.

Determine how many cycles could be used for other computations without penalizing the performance of the overall function. You can do this by inserting different numbers of instructions (e.g., `nop`) in the code and timing the function.

Since the MIPS assembler has its own ideas of how to arrange and optimize the code, make sure you inspect your object code with a disassembler (e.g., within GDB, or using the program DIS) to ensure it is what you really want.

Document your experiments and your findings. Be prepared for some surprising results!

Task 4C

Rewrite your code from Task 3 to perform unsigned multiplication using the `mult` instruction, but yielding the same performance as the function `mult_full`.

Stack Frames and Procedure Calling

The file *ASSTDIR*/`structure.c` shows the C code for a function that demonstrates many interesting features of implementing C on a MIPS machine. The code generated by GCC with the `-O` flag is shown in the file *ASSTDIR*/`structure.s`

Task 5

Document the following:

- Show the layout of the data structure `list_ele`.
- Explain how the program implements the returning of a structure by `sum_list`.
- Describe the register allocation used in `sum_list`.
- Show the layout of the stack frame used by `sum_list`.
- Create an annotated version of the assembly code for `sum_list` using our usual formatting conventions.