# CS 740, Fall 1997
## Homework Assignment 1

Assigned: Sept. 15
Due: Wed., Sept. 24, 10:30AM

**Policy**

You may work in a group of up to 3 people in solving the problems for this assignment. Turn in a single writeup per group, indicating all group members.

**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-f97/public`

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

You should hand in a hard copy document for this assignment. Formatted text is preferred to hand written.

In addition, you should make an electronic version of your multiply routines available for grading. Put these in a file `smult.c`, implementing the procedures declared in *ASSTDIR*`/smult.h`. This file should not contain any `main` function. Any functions other than the ones declared in `smult.h` should be declared `static`. Include as comments at the beginning of the file the names of all group members. Send this file by electronic mail to `Randy.Bryant@cs.cmu.edu`.

**Introduction**

The purpose of this assignment is to develop techniques for measuring code performance, to practice reasoning about bit-level arithmetic and low-level code optimization, and to better understand MIPS procedure calling conventions.

For this assignment, you will want to use a machine with a MIPS processor. Candidates include the DECSTATION workstations in the CS Department, 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.

**Using Interval Timers**

When comparing machines, or when optimizing code, it is often useful to measure the number of clock cycles required to execute a particular operation or procedure. Some machines have special facilities to assist in performance measurements. Lacking such facilities, almost all machines provide *interval timers*,

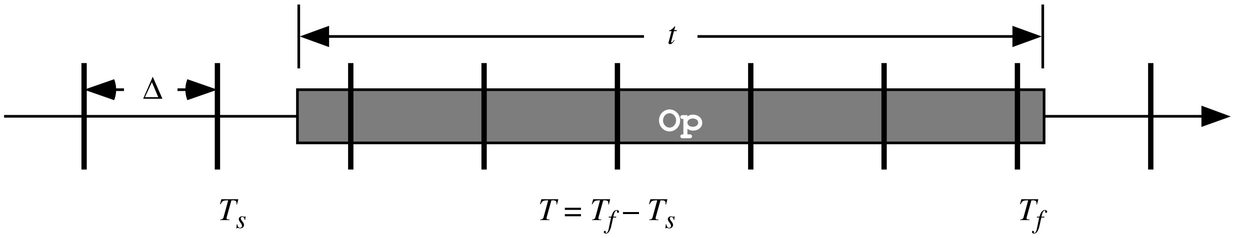


Figure 1: Time Measurement with an Interval Timer

a relatively crude method of computing elapsed times. For the first part of this assignment, we will devise a method for getting cycle-level accuracy using an interval timer.

The overall operation of an interval timer is illustrated in Figure 1. The system maintains a settable timer that is updated periodically. That is, every Δ microseconds, the counter is incremented by Δ. Using the Unix library routine `getitimer`, the user can poll the value of this counter. Thus, to measure the elapsed time of some operation `Op`, the user can poll the timer to get a starting value $T_s$, perform the operation, and poll the counter to get a final value $T_f$. The elapsed time for the operation is then approximately $T = T_f - T_s$. As the figure illustrates, however, the actual elapsed time $t$ may differ from $T$ significantly, due to the coarseness of the timer resolution. Since the value of Δ is around 10 *milliseconds* for most systems, this error can be very significant.

We have encapsulated the Unix interval timer routines for you in a handy timer package called *ASSTDIR*`/etime.c`. You should use this package for all measurements in the assignment. See *ASSTDIR*`/example.c` for a simple example of how to use the package. One notable feature is that it converts the measurements to units of seconds, expressed as a C `double`. The procedure for timing operation `Op` is then:

```
init_etime();
Ts = get_etime();
Op;
Tf = get_etime()
T = Tf - Ts;
```

**Problem 1:**

Derive an expression for the minimum and maximum values of $t$ in terms of $T$ and Δ.

---

**Problem 2:**

Write a procedure that will determine the value of Δ for your machine.

---

We can improve the accuracy of the measurements by making sure that the activity we measure has sufficient duration to overcome the imprecision of interval timers. That is, we can accurately measure the time required by `Op` by executing it $n$ times for sufficiently large $n$:

```
init_etime();
Ts = get_etime();
for (i=0; i<n; i++) {
```

```
        Op;
    }
    Tf = get_etime()
    T = Tf - Ts;
    Top = T/n;
```

### Problem 3:

Suppose we wish to compute a time $T_{op}$ for `Op` that is guaranteed to approximate the true time $t_{op}$ within some maximum relative error $\epsilon$. That is

$$\frac{|T_{op} - t_{op}|}{t_{op}} \leq \epsilon.$$

Assume that the only source of inaccuracy is the coarseness of the interval timer. Derive an expression for the minimum measured elapsed time $T$ in terms of $\Delta$ and $\epsilon$ that will guarantee this accuracy.

---

### Problem 4:

The file *ASSTDIR*/`ftime.h` contains the declaration of a procedure `ftime` that computes the time required by a procedure, passed as its first argument, with an error tolerance given by its second argument.

Implement this function. The idea is to start with $n = 1$, and keep doubling it until the overall time $T$ is large enough to guarantee sufficient accuracy.

---

### Problem 5:

Test your function timer using the program *ASSTDIR*/`freq.c`, which uses `ftime()` to estimate the clock frequency of your machine. This routine assumes that your machine executes an integer addition in one clock cycle. This is a safe assumption for most modern processors.

---

You will use your timing and clock frequency code extensively this term. You might want to put it all in a single package that can determine $\Delta$, time functions, and measure the clock frequency.

#### Saturating Multiplication

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

```
int mult_lo(int a, int b)
{
  return a * b;
}
```

generates the following MIPS code:

```
mult:
        mult    $4,$5      # Initiate multiplication
```

```
        mflo    $2          # Grab low order word
        j       $31         # Return
```

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 return value.

In general, this multiplication will yield a valid result only if $-2^{W-1} \leq a \cdot b \leq 2^{W-1} - 1$ (the range of $W$-bit, two's complement integers). For products outside this range, the result has little relation to the true product.

In some applications, we would prefer to perform *saturating* arithmetic, where products outside the range are set to one of the limiting values $-2^{W-1}$ or $2^{W-1} - 1$. This functionality is commonly used in signal processing applications, where we would rather set an amplitude to a limiting value than to have it wrap around as is done with conventional multiplication. As an example, the MMX instructions implemented by recent Intel processors include saturating arithmetic operations, as do most digital signal proccessing chips.

Getting the high order word of the product requires dropping into assembly code to get the value of the `HI` register. 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 puts it in a known register. [In this case we know that variable `a` is in register `$4`.]

```
int mult_hi(int a, int b)
{
  asm("mult $4,$5");
  asm("mfhi $4");
  return a;
}
```

When compiled, this generates code [shown using our convention that upper case branch/jump instructions denote delayed branches]:

```
mult_hi:
        mult    $4,$5      # Initiate multiplication
        mfhi    $4         # Grab high order word
        J       $31        # Return
        move    $2, $4     # with value in register 4
```

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

The file *ASSTDIR*/`saturate.h` contains prototypes for four different ways of implementing saturating multiplication. For each of these, you should describe your algorithm using a concise mathematical derivation, and then show your code. For each implementation, you should put first priority on correctness and second priority on efficiency. That is, you should attempt to be reasonably optimal in your implementation, but don't hack on it endlessly.

## Problem 6:

The first routine `mult_a` uses an assembly code directive to get the high order word of the product. Based on the low and high order words, it computes the saturating product.

**Problem 7:**

The second routine `mult_d` uses a double precision multiply to compute the product and then generates an integer representation of the saturating product. Since the IEEE double precision has a 52-bit mantissa, we are guaranteed to compute the product to sufficient accuracy.

**Hint**: You might want to run experiments on the effect of casting a `double` to an `int` to see how this conversion handles out of range values.

---

**Problem 8:**

The third routine `mult_h` splits each argument into $W/2$-bit words, forms 3 partial products (each of which fits within the $W$-bit word size, and computes the result based on the arguments and partial products.

**Hints**:

- You may find it easiest to compute the product of the absolute values of the operands and adjust the answer accordingly.
- You will want to be able to determine whether the sum of two unsigned $W$-bit words overflows. Make an argument that the following code detects this condition correctly for unsigned integers `u` and `v`:

```
unsigned t = u+v;
ovf = (t < u);
```

---

**Problem 9:**

The fourth routine `mult_s` avoids using any form of multiply instruction. Instead, it implements the standard shift-and-add method of multiplying numbers, modified to handle the saturating conditions. Follow the hints from the previous problem.

---

The file *ASSTDIR*/`smult.test` contains 941 test cases for testing your routines. Each line of this file contains two arguments plus their saturating product. Note how the first 841 cases in this file exercise your routines for typical corner cases, while the final 100 consists of random values.

**Performance Measurement**

Now, let's see what kind of performance these routines give you.

**Problem 10:**

Measure the running times of your four multiplication routines on a MIPS machine. Express your result as the average time per product, using the 941 test cases as your sample. Express your times in both microseconds and clock cycles. Document the machine type and clock freqency.

**Hints**:

- Make sure your are not attempting any file I/O while you are measuring your routines. Load your test cases into memory before beginning the evaluation.

- You can subtract off the overhead of making timer calls, accessing memory locations, and looping by writing a "dummy routine" that performs the same operations as the code you are testing, but without performing the multiplies. See the functions in *ASSTDIR*/`freq.c` as examples.
- Be sure not to write code that an optimizing compiler can elimintate via dead code elimination. The code in *ASSTDIR*/`freq.c` shows how this can be done by updating some memory location with each computed result.

---

### Problem 11:

Measure the last three implementations on some machine other than a MIPS machine. This could be your own workstation or PC, or some high performance server. If you choose to measure the performance of a 64-bit machine, e.g., a DEC Alpha, you should implement your routines for $W = 64$.

Document the machine type, its clock frequency, and your experimental results.

---

#### 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`

### Problem 12:

Document the following:

A. Show the layout of the data structure `list_ele`.

B. Explain how the program implements the returning of a structure by `sum_list`.

C. Describe the register allocation used in `sum_list`.

D. Show the layout of the stack frame used by `sum_list`.

E. Create an annotated version of the assembly code for `sum_list` using our usual formatting conventions.

---

#### Extra Credit

### Problem 13:

For the second machine, implement a version of saturating multiplication that, like `smult_a`, makes use of the machine-computed high order word of the product. Examples of machines that can compute this word include: DEC Vax, PowerPC, and DEC Alpha. Include measurements of this routine in your performance comparison.

**Hint**: The GCC compiler provides a data type `long long int` for implementing multiple precision integer arithmetic. You can use it to reverse engineer efficient ways to compute the high order word of a product.