Measuring Execution Times

The code example you provided shows how to use the GetMseconds procedure in the Irvine32 library to measure the execution time of a program.

The **GetMseconds procedure** returns the number of system milliseconds that have elapsed since midnight.

To measure the execution time of a program, you would first call the GetMseconds procedure to record the start time.

Then, you would call the program whose execution time you wish to measure. Finally, you would call the GetMseconds procedure again to record the end time.

The difference between the end time and the start time is the execution time of the program.

The following code example shows how to use the GetMseconds procedure to measure the execution time of a simple program:

```
.data
    startTime DWORD ?
    procTime DWORD ?

.code
    call GetMseconds
    ; get start time
    mov startTime, eax

    ; call the program whose execution time you wish to measure
    ; ...

    call GetMseconds
    ; get end time
    sub eax, startTime
    ; calculate the elapsed time
    mov procTime, eax
    ; save the elapsed time
```

The **variable procTime** will now contain the execution time of the program, in milliseconds.

You can use this technique to measure the execution time of any program, regardless of its complexity.

However, it is important to note that the overhead of calling the GetMseconds procedure twice is insignificant when compared to the execution time of most programs.

Relative Performance

You can also use the GetMseconds procedure to measure the relative performance of two different code implementations.

To do this, you would measure the execution time of each implementation and then divide the execution time of the first implementation by the execution time of the second implementation.

The result will be a number that indicates the relative performance of the two implementations.

For example, the following code example shows how to measure the relative performance of two different sorting algorithms:

```
.data
    startTime1 DWORD ?
    procTime1 DWORD ?
    startTime2 DWORD ?
    procTime2 DWORD ?
.code
    ;measure the execution time of the first sorting algorithm
    call GetMseconds    ;get start time
    mov startTime1, eax
    ; call the first sorting algorithm ...
    call GetMseconds
    ; get end time
    sub eax, startTime1
    ; calculate the elapsed time
    mov procTime1, eax
    ; save the elapsed time
    ; measure the execution time of the second sorting algorithm
    call GetMseconds
    ; get start time
    mov startTime2, eax
    ; call the second sorting algorithm
    ; ...
    call GetMseconds
    ; get end time
    sub eax, startTime2
    ; calculate the elapsed time
    mov procTime2, eax
    ; save the elapsed time
    ; calculate the relative performance of the two sorting algorithms
    div procTime1, procTime2
    ; the result is now in EAX
```

The EAX register will now contain the relative performance of the two sorting algorithms. A value of 1.0 indicates that the two sorting algorithms have the same performance.

A value greater than 1.0 indicates that the first sorting algorithm is faster than the second sorting algorithm. A value less than 1.0 indicates that the first sorting algorithm is slower than the second sorting algorithm.

You can use this technique to measure the relative performance of any two code implementations, regardless of their complexity.

Comparing MUL and IMUL to Bit Shifting in Depth

In older x86 processors, there was a significant difference in performance between multiplication by bit shifting and multiplication using the MUL and IMUL instructions.

However, in recent processors, Intel has managed to greatly optimize the MUL and IMUL instructions, so that they now have the same performance as bit shifting for multiplication by powers of two.

The following code shows two procedures for multiplying a number by 36 using bit shifting and the MUL instruction:

```
;Multiplies EAX by 36 using SHL, LOOP_COUNT times.
mult_by_shifting PROC
mov ecx, LOOP_COUNT
L1: push eax
; save original EAX
mov ebx, eax
shl eax, 5
shl ebx, 2
add eax, ebx
pop eax
; restore EAX
loop L1
ret
mult_by_shifting ENDP

; Multiplies EAX by 36 using MUL, LOOP_COUNT times.
mult_by_MUL PROC
mov ecx, LOOP_COUNT
L1:
push eax
; save original EAX
mov ebx, 36
mul ebx
pop eax
; restore EAX
loop L1
ret
mult_by_MUL ENDP
```

The following code calls the mult_by_shifting procedure and displays the timing results:

```
.data
    LOOP_COUNT = 0FFFFFFFFh
.data
    intval DWORD 5
.code
    call
    GetMseconds
    ; get start time
    mov
    startTime,eax
    mov
    eax,intval
    ; multiply now
    call
    mult_by_shifting
    call
    GetMseconds
    ; get stop time
    sub
    eax,startTime
    call WriteDec
    ; display elapsed time
```

The code above, is a simple example of how to measure the execution time of a program using the GetMseconds procedure in the Irvine32 library. The program multiplies the integer 5 by 36 using the mult_by_shifting procedure, and then displays the execution time.

The two .data segments in the program are used to define two variables: LOOP_COUNT and intval. LOOP_COUNT is a constant that specifies the number of times to repeat the multiplication operation. intval is the integer that is multiplied by 36.

The reason for having two .data segments is not entirely clear. It is possible that the original author of the code was simply trying to organize the data in a logical way.

However, it is also possible that the author was trying to take advantage of some optimization in the Irvine32 library.

Regardless of the reason, it is not necessary to have two .data segments in this program. The two variables could be defined in the same .data segment without any problems.

Here is a revised version of the program with the two .data segments combined into one:

```
.data
    LOOP_COUNT = 0FFFFFFFFh
    intval DWORD 5
.code
    call
    GetMseconds
    ; get start time
    mov
    startTime,eax
    mov
    eax,intval
    ; multiply now
    call
    mult_by_shifting
    call
    GetMseconds
    ; get stop time
    sub
    eax,startTime
    call WriteDec
    ; display elapsed time
```

This revised version of the program works just as well as the original version, and it is more concise and easier to read.

- You can have as many segments for .data, .code, .bss/text.
- Use segments wisely, grouping related data and code.
- Avoid excessive segments for clarity and performance.

On a legacy 4-GHz Pentium 4 processor, the mult_by_shifting procedure executed in 6.078 seconds, while the mult_by_MUL procedure executed in 20.718 seconds.

This means that using the MUL instruction was 241 percent slower. However, when running the same program on a more recent processor, the timings of both function calls were exactly the same.

This example shows that Intel has managed to greatly optimize the MUL and IMUL instructions in recent processors.

Therefore, there is no longer any need to use bit shifting for multiplication by powers of two.

In fact, using the MUL and IMUL instructions is generally preferred, as they are more readable and easier to maintain.