12. Floating Point Operations, Instruction Timing

Background

One of the goals of this lab is experiment with floating point operations on the x86. Hardware support for floating point operations is included on every modern CPU as they are a necessity for many applications. NOTE: The in-class lecture may or may not have covered the details of x86 floating point operations by the time you take this lab. However, do not stress over this - the lab looks at very simple uses of the floating-point instructions

"All men are created equal" is a truth that Americans hold to be self-evident. Substitute "instructions" for "men" and "All *instructions* are created equal" is not self-evident -- in fact - the statement "All instructions are created equal, except some instructions are more equal than others[±]" is closer to the truth. In many cases, one reason for going to the trouble of writing something in assembly code is for performance -- and in order to write high performance code you need to understand which instructions or instruction sequences are fast or slow. This lab looks at a method for timing x86 instruction sequences to help you understand which instructions are slow or fast in terms of execution speed.

Objectives

Understand:

A. Simple uses of the x86 floating point instructions.

B. Differences in execution time of various instruction types.

Pre-Lab

The basics of floating point operations are covered very well in the Irvine textbook, Chapter 15. Be sure to read this chapter before attempting this lab. Also read the online lecture notes concerning the IEEE floating-point number format.

±Read George Orwell's *Animal Farm* if you get a chance.

Lab

A. x86 Floating Point Operation

The program below (*fpexam.asm*) illustrates some simple uses of the x86 floating point instructions.

```
.model small
.586
.stack 200h

.data
fres   dd  ?     ;; floating point result
op1    dd 2.789
op2    dd 12.1
op3    dd 3.141592654
op4     dd 5.0
op5    dd 1.570796327


rstring db "Float Result is: ",0
ares   db 30 dup (?)


.code
       ;; ftoa procedure found in 'float.lib'
       extrn ftoa:proc, writestring:proc, crlf:proc

       main    proc
              mov     ax,@data
              mov    ds,ax
              finit
              fld    op1    ;; load into st0
              fld    op2    ;; load into st1
       A1:    fdiv            ;; st1 = st1/st0, pop stack so st1 goes into st0
              fstp   fres   ;; save single result as single precision
                             ;; pop stack also
              call   fprint
              fld    op3
       B1:    fmul    op4     ;; st0 = st0 * operand, no stack pop
              fstp   fres
              call   fprint
              fld    op3
       C1:    fsqrt         ;; square root of pi
              fstp   fres
              call   fprint
              fld    op3
       D1:    fsin          ; sin  (pi)
              fstp   fres
              call   fprint
              fld    op3
       E1:    fcos          ; cos (pi)
              fstp   fres
              call   fprint
              ;;
              Mov    ax, 4c00h    ;exit to DOS
              Int    21h

       main   endp
```

```
        fprint  proc

                mov     dx,offset rstring
                call    writestring
                mov     ax,word ptr fres
                mov     dx, word ptr fres+2
                mov     cx, 10 ;; 10 digits of precision
                mov     di,offset ares
                call    ftoa
                mov     dx,offset ares
                call    writestring
                call    crlf
                ret
        fprint  endp
end      main
```

The program does the following floating-point operations: divide, multiply, square root, sine and cosine. The operands come for the single precision floating point values labeled as op1, op2, op3, and op4. The result of each operation is printed to the screen using the *fprint* procedure.

How does this program work?

1. The *finit* instruction initializes the Floating Point Unit (FPU). The FPU has 8 registers named ST0-ST7 and which are arranged as a stack. Register ST0 is the top of the stack.
2. The instructions *"fld op1"* and *"fld op2"* loads operand *op1* into register ST0, and *op2* into register ST1. Note that *op1* and *op2* are stored in memory as 32-bit values (single precision floating point numbers). In order for MASM to generate a floating point format instead of using an integer, you MUST use a decimal point in the number when specifying its value (ie. op1 dd 2.0 -- use a decimal point even if the fraction is zero).
3. The *fdiv* instruction (no operand format) does a divide operation ST1 = ST1/ST0, and then pops the stack. This means that ST1 goes into ST0, and that ST1 becomes empty. After the FDIV instruction, the result is sitting in register ST0.
4. The "*fstp fres*" instruction stores the register value ST0 to the memory operand *fres* as a 32-bit single precision number. After the store operation, the stack is popped so register ST0 is now 'empty'.
5. The procedure *fprint* is used to print the value in the memory location *fres* to the screen as a floating-point number. It uses an external procedure called *ftoa* that is found in the *float.lib* library to covert the 32-bit FP number to an ASCIIZ representation. The *ftoa* procedure expects CX to contain the number of digits of precision needed; DX:AX to contain the 32-bit single precision FP number, and DS:DI to point to a memory buffer where the null-terminated string is to be placed (the 'ares' buffer is used for this). After *ftoa* is called, the Irvine library procedure *writestring* is used to print the string to the console. Note that this program requires two libraries to be linked to it, so the library needs to be specified as "Irvine+float" when the linker prompts for a library name.
6. The *"fmul op4"* operation (single operand format) does a multiply operation ST0 = ST0*op4. Note ST0 is equal to *'op3*' because of the 'fld op3' instruction that immediately preceded the *fmul* operation. The result of the *fmul* operation is then stored into *fres* via 'fstp fres' and printed to the console by the *fprint* procedure.

7. The *fsqrt* instruction performs the square root operation ST0 = sqrt(ST0). The value of ST0 was set to *op3* by the previous "fld op3" instruction.
8. The *fsin* instruction performs the sine operation ST0 = sine(ST0) (ST0 value in radians). The value of ST0 was set to *op3* by the previous "fld op3" instruction.
9. The *fcos* instruction performs the cosine operation ST0 = cosine(ST0) (ST0 value in radians). The value of ST0 was set to *op3* by the previous "fld op3" instruction.

Lab Question 1: Assemble this program and execute it.

A. For each of the calculations, list the numerical arguments and verify that the correct result is obtained. Do you notice any round-off error in any of the computations?

B. Change the program such that the FSQRT function is passed a negative value. Single step through the program and record the 32-bit value that is passed back (wait until the result is stored in memory at location *fres*, then examine the value in memory - remember that it will a 32-bit value stored LSB first). . Using the information in the notes on single-precision floating point numbers, determine the values of the sign bit, the *exponent*, and the *significand*. What 'special' number is this?

C. Write a program that will compute the two real roots for the equation $ax^2 + bx + c$ using the quadratic formula $(-b \pm \sqrt{b^2 - 4 a c})/ 2a$. Write your program such that it prompts the user for the values of a, b, and c (use the Irvine *Readint* procedure). Warning -- the Irvine *Readint* procedure returns a 16-bit signed integer value. You will need to store this to a memory location, then use "FILD memloc" to read this integer value into the floating point register stack (will go into ST0 and will be converted to floating point format). Test your program with several values of a,b,c and include the assembled listing of this program in your lab report.

B. Cycle Counts for Instructions

The number of clock cycles that an instruction requires depends on the instruction and the addressing modes of its instructions. Typically, instructions of the same general class with the same addressing modes will take the same number of clock cycles (i.e, the logic instructions XOR, AND, OR using register operands take the same number of clock cycles). The program below (*inttim.asm*) can be used to estimate the number of clock cycles required for an instruction.

```
.model small
.586
.stack 200h
.data

baselo  dd  ?
basehi  dd  ?
timlo  dd  ?
timhi  dd  ?
mtest  dw  0

.code
       extrn Crlf:proc,Writestring:proc, WriteInt:proc

main    proc
       mov     ax,@data
       mov    ds,ax

       mov    si,1
outerlp1:

;; baseline loop -- everything except instruction to be measured
       mov    ecx,64
       call   timget
       mov    baselo,eax   ;; save low value
       mov    basehi,edx   ;; save high value

lp1:
       mov    ax, 0f20h
       mov    bx, 0140h   ;320
       loopd  lp1

       call   timget
       sub    eax,baselo   ;; subtract low value
       sbb    edx,basehi   ;; subtract high value
       mov    baselo,eax   ;; save
       mov    basehi,edx

;; measure loop -- has instruction to be measured

       mov    ecx,64
       call   timget
       mov    timlo,eax   ;; save low value
       mov    timhi,edx   ;; save high value

lp2:

       mov    ax, 0f20h
       mov    bx, 0140h   ;320

;uncomment one line at a time and see how cycle time changes
```

```
;       div     bl
;       mul     bx
;       add     ax,bx
;       mov     ax, word ptr mtest
;       add     word ptr mtest,ax
;       shl     ax,1
;       shl     ax,20

sk1:    loopd   lp2

        call    timget
        sub     eax,timlo    ;; subtract low value
        sbb     edx,timhi    ;; subtract high value
;; now subtract baseline value
        sub      eax,baselo
        sbb     edx,basehi    ;; EDX:EAX has delta time
        ;;
        mov     timlo,eax
        mov     timhi,edx

        dec     si
        cmp     si,0
        jnz     outerlp1     ;; loop at least once to
                             ;; make sure all code is in cache
                             ;; else may get negative times!
;; print this out
        mov     cx,4
        lea     si,timhi+2
        mov     bx,16
lp3:
        mov     ax,[si]
        call    writeint
        sub     si,2
        loop    lp3

        Mov     ax, 4c00h     ;exit to DOS
        Int     21h
main    endp

;;; procedure that returns the Pentium+ 64 bit timer value
;;;  in EDX:EAX

        timget        proc
                      rdtsc         ;; read timestamp counter
                      ret
        timget        endp

end      main
```

The above program attempts to measure the number of clock cycles for a particular instruction using the 64-bit timer available on Pentium compatible processors. The approach is to execute a loop a fixed number of times with and without the target instruction, and compute the cycle count differences between the two loops.

How does the program work?

1. The loop labeled as 'lp1' is the loop without the target instruction. The loop is executed 64 times and the resulting 64-bit timer count representing the number of clock cycles that this loop required to execute is stored in the memory value *basehi:baselo* .

2. The looped labeled as 'lp2' is the same code as 'lp1' except the target instruction should be inserted in this loop. There are several target instructions in the code that are commented out -- one of these should be uncommented and the code assembled/executed to get a timing measurement for that instruction. This loop is executed 64 times and the resulting 64-bit timer count representing the number of clock cycles that this loop required to execute is stored in the memory value *timerhi:timerlo* .
3. After the two loops are executed, the value for *timerhi:timerlo* - *basehi:baselo* is written to the console (in hex). This is the number of extra clock cycles that *lp2* took over *lp1* because of the additional instruction in *lp2*. Dividing this number by the loop count (64) should give you an *approximate* clock cycles per instruction value.

Careful examination of the *inttim* example shows the loops lp1, lp2 are actually executed twice before a value is printed. This is an attempt to make sure that all instructions are in the on-chip cache before we calculate a cycle count time (the first time through the code all instructions get copied into the cache). The loop count time of 64 was chosen somewhat arbitrarily - you want it large enough to account for any extraneous clock cycles due to cache behavior but not so large that the program gets temporarily suspended while another task executes.

Lab Question 2: For each of the target instructions in loop2 (div, mul, add registers, mov, add with memory, shift small, shift large), uncomment the instruction, assemble and execute. Record the hex value for the difference in clock cycles between the two loops, and compute the number of clock cycles per instruction. Be sure that you record the processor type that you are running the programs on, and use the SAME PC for all measurements.

A. You may get a value less than 1 clock cycle per instruction, or even see very little to no difference between the loops for some instructions. What could be the reason for this? (hint: look up the term 'superscalar' in either your Uffenbeck textbook or from WWW resources).
B. Rank the instructions in terms of SLOWEST to FASTEST. Why might some instructions take longer to execute than others?

Lab Question 3: The multiply operation in the program does AX * BX where BX is the value 320. This multiplication could also be done as:
AX*320 = AX*(256+64) = AX*256+AX*64 = shl(AX,8) + shl(AX,6).

Modify the program to see which method is faster. When we are trying to compare two dissimilar loops, you should modify the loop2 code to NOT subtract the 'basehi:baselo' value that is recorded in loop1 -- just record the absolute clock cycles for loop2 (timelo, timehi) using the MUL instruction, and loop2 rewritten to perform the operation above (note that the value of 320 does not have to be loaded into BX for the 2nd case). Report your findings in your lab report and include the assembled listing of your program.

NOTE: The answers to these questions will depend entirely upon what type of processor you are using: Pentium 2/3/4/??, Athon?? Etc. You need to perform all measurements on the same processor type.

Lab Report

A. Describing What You Learned

Include the answers to all "**Lab Questions**" in your report.

B. Applying What You Learned

Demonstrate the programs you wrote for Lab Questions 1 and 3 to the TA.

One last note - it is instructive to also try this instruction timing approach for floating point operations. However, this only works if you have Win 2000/XP. It does not work for Win98/ME/NT. The x86 Virtual Machine used under Win98/ME/NT skews the FP cycle counts.