Floating Point Arithmetic

ICS312 - Spring 2009
Machine-Level and Systems Programming

Henri Casanova (henric@hawaii.edu)

Floating Point Numbers

- Many computer applications manipulate floating point numbers
  - Fixed precision, as an approximation of real numbers
- Therefore, computers are built with ways to store and operate on floating point numbers
- We're going to see how floating point numbers are encoded in computers, and what x86 instructions can operate on them

Floating Point Numbers

- In base ten, we have a natural interpretation of floating point numbers:
  - $412.3126 = 4*10^2 + 1*10^1 + 2*10^0 + 3*10^{-1} + 1*10^{-2} + 2*10^{-3} + 6*10^{-4}$
- It's the same in binary!
  - $011.1011 = 0*2^2 + 1*2^1 + 1*2^0 + 1*2^{-1} + 0*2^{-2} + 1*2^{-3} + 1*2^{-4}$
    - $2^{-1} = 0.5$
    - $2^{-2} = 0.25$
    - $2^{-3} = 0.125$
    - $2^{-4} = 0.0625$
- Therefore it's easy to convert a floating point binary number into a floating point decimal number

From Decimal to Binary

- This is a two-step process:
  - Convert the integer part to binary
    - We know how to do that
      - The "divide by two and look at the remainder" technique
  - Convert the fractional part to binary
- Converting the fraction part:
  - Multiply by 2 and look at the number to the left of the decimal point
  - This is the leftmost bit of the binary representation of the fractional part (in the right order!)
  - Repeat
- Let's see this on an example

Example Conversion

- Let's convert 23.125 from decimal to binary
- Converting 23 (most to least significant: bottom to top)
  - 23 / 2 = 11 + **1**
  - 11 / 2 = 5 + **1**
  - 5 / 2 = 2 + **1**
  - 2 / 2 = 1 + **0**
  - 1 / 2 = 0 + **1**
  - Conversion: 10111
- Converting .125 (most to least significant: top to bottom)
  - .125 * 2 = **0**.250
  - .250 * 2 = **0**.500
  - .500 * 2 = **1**.000
  - Stop
  - Conversion: 1001
- End result: **10111.001**
  - $2^4+2^2+2^1+2^0+2^{-3}$

Another Example

- Let's look at 0.85
  - .85*2 = 1.7
  - .7*2 = 1.4
  - .4*2 = 0.8
  - .8*2 = 1.6
  - .6*2 = 1.2
  - .2*2 = 0.4
  - .4*2 = 0.8
  - . . .
- This never ends!
- This number has a finite representation in decimal, but an infinite representation in binary
  - Just like 1/6 in decimal (0.166666666…)
- The computer can only store an approximation of it

Storing Binary FP Numbers

- Binary FP numbers are stored in hardware using a consistent format:
  - A number encoded with some number of bits
  - An exponent encoded with some number of bits

  Number = 1.ssssssssss * $2^{eeeeeeee}$
    - ssssssssss is a binary number
    - eeeeeeeee is a binary number
- Example:
  - Earlier we saw that 23.850 is 10111.11011001100110…
  - This is not in the right format because we don't have a single 1 to the left of the point
  - Say we have a CPU in which we have 8 "s bits" and 4 "e bits"
  - Then, encoding = 1.01111101 * 2^{0100}
    - Multiply by 2^4 so that the point is moved 4 spots to the left
  - In memory the number would look like: 011111010100

IEEE Standard

- Most computers use an encoding scheme proposed by the IEEE society
  - Institute of Electrical and Electronics Engineers
- Often supported by the actual hardware
  - Could be supported in software
- The IEEE defines two formats
  - Single Precision (in C: float)
  - Double Precision (in C: double)
- In fact, Intel's math coprocessor uses a third encoding which has higher precision!
  - Could be used directly in assembly, or by some compilers
  - Typically though it's converted to IEEE format when going back and forth to and from memory

IEEE Single Precision: 32 bits

- Uses 32 bits
  - s: sign bit (no 2's complement!)
    - 0 = positive
    - 1 = negative
  - e: biased exponent (8 bits)
    - Biased exponent = true exponent + 7F
  - f: fraction (23 bits)
    - The 23 bits after the "1." (hidden one representation)
- Typically "accurate" to the 7th decimal digit

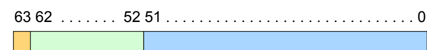
IEEE Single Precision: 32 bits

- Example: 23.850
  - Bit sign = 0
  - The exponent is 4. We store 4 + 7F = 83 (in hex) = 10000011
  - The fraction is 01111101100110011001100
  - Overall:
    - 0100 0001 1011 1110 1100 1100 1100 1100
    - 4 1 B E C C C C
  - This is an approximation of 23.850
  - If you convert back, you get: 23.849998474

Special Values of e and f

- Range ~ $10^{-44.85}$ to $10^{+38.53}$
- e = 0 and f = 0: denotes number 0
  - Because we're using sign magnitude we have a +0 and a -0
- e = FF and f = 0: denotes +inf
  - There is a +Inf and a -Inf
- e = FF and f != 0: denotes NaN

IEEE Double Precision: 8 bytes



- Typically accurate to 15 decimal digit
- Exponent is now 11 bits
  - Sum of the true exponent and 3FF
- Fraction is now 52 bits
- Range ~ $10^{-323.3}$ to $10^{+308.3}$

Floating Point Arithmetic

- The book has a section about how the computer perform FP arithmetic
- The whole point is to show that FP arithmetic isn't exact, and in fact errors can be (relatively) large
- As a result, when you write code you often have to avoid direct comparison
  - Don't do if (x == a)
  - But if (fabs(x-a)/fabs(a) < EPSILON)
    - Assumes a != 0.0

The Numeric Coprocessor

- Early processors didn't have hardware support for FP arithmetic
- FP arithmetic was performed "by hand" via _many_ non FP arithmetic operations
  - Took a LONG time
  - Was a long-standing performance bottleneck
- Then Intel provided a "math coprocessor"
  - A chip that performed hardware FP arithmetic
- This coprocessor is integrated with the CPU
  - But still programmable separately

The Numeric Coprocessor

- The coprocessor has 8 registers
- Each register holds 80 bits of data
  - Remember that internally numbers are stored using a higher precision than with the IEEE standard
- The registers are named ST0, ST1, …, ST7
- These registers are organized as a stack!
  - ST0 always stores the value at the top of the stack
- There is a status word that holds useful flags: the book only talks about C0, C1, C2, and C3

ST0
ST1
ST2
ST3
ST4
ST5
ST6
ST7

status word

FP Instructions (Pushing)

- Coprocessor instructions start with an F
- FLD xxx
  - Loads a floating point number from memory on top of the FP stack
  - xxx may be a single, double, or extended precision number or a coprocessor register
- FILD xxx
  - Reads an integer from memory, converts it to a floating point, and stores the result on top of the stack. The source can be a word, a dword, or a qword
- FLD1
  - Stores a 1 on top of the stack
- FLDZ
  - Stores a 0 on top of the stack

FP Instructions (Popping)

- FST xxx
  - Stores ST0 (the top of the stack) into memory
- FSTP xxx
  - Stores ST0 into memory and pop it
- FIST xxx
  - Store ST0 into memory, but converted as an integer
  - By default rounds off to the nearest integer
- FXCH STn
  - Exchanges ST0 and STn
- FFREE STn
  - Frees up a register by marking it as empty

FP Additions

- FADD xxx
  - ST0 += xxx
- FADD xxx, ST0
  - xxx += ST0
- FADDP xxx
  - xxx += ST0 and then pop the stack
- FIADD xxx
  - ST0 += (float)xxx, where xxx is an integer in memory

FP Subtractions

- In floating point a - b != -(b - a)
  - Due to round-off errors
- So we have more instructions
- FSUB xxx
  - ST0 -= xxx
- FSUBBR xxx
  - ST0 = xxx - ST0
- Etc. (see the book for the whole list)

FP Multiplications, Divisions

- FMUL xxx
  - ST0 *= xxx
- And others…

- FDIV xxx
  - ST0 /= xxx
- And others…

- See book for full list

FP Comparisons

- FCOM xxx
  - Compares ST0 and xxx
- FCOMP xxx
  - Compares ST0 and xxx, then pops the stack
- And a few others (see book)

- Comparisons set the C0, C1, C2, and C3 bits of the status word
- But these bits are not directly accessible!
- So one transfers the status word register to the regular FLAGS register
  - First to a regular register
  - Then from that register to the FLAGS register

FP Comparisons

- FSTSW xxx
  - Stores the status word into a word of memory or the AX register
  - Flags have the same “meaning” as when comparing unsigned integers
- SAHF
  - Stores the AH register into the FLAGS register
- LAHF
  - Loads the AH registers with the bits of the FLAGS register
- Let’s see an example

FP Comparisons Example

```
;    if (x > y) then  ...  else ...
;
         fld      qword [x]          ; ST0 = x
         fcomp    qword [y]          ; compare ST0 and y
         fstsw    ax                 ; move C bits into AX
         sahf                        ; move AH in FLAGS
         jna      else_part          ; if (x <= y) jump
then_part:
         ; then block
         jmp end_if
else_part:
         ; else block
end_if:
```

Newer FP Comparisons

- Starting with the Pentium Pro, the x86 assembly includes instructions to compare FP registers and that modify the FLAGS register directly
- FCOMI   xxx
  - Compare ST0 and xxx, which should be a FP register
- FCOMIP xxx
  - Same, but pops

Other Instructions

- FSQRT
  - ST0 = sqrt(ST0)
- FABS
  - ST0 = |ST0|
- FCHS
  - ST0 = - ST0
- FSCALE
  - ST0 = ST0 ^ ceil(ST1)

Example

- Let's write a function that computes the Euclidian Distance between two points in space
  - Sqrt $((x1 - x2)^2 + (y1 - y2)^2)$
- Let's write this as a function that's called by a C program:

```
double d, x1, x2, x3, x4;
. . .
distance(x1, x2, x3, x4, &d);
. . .
```

Distance Function

- We're going to use the NASM "%define" directive to conveniently refer to function parameters
  - Think of it as a C "#define"

```
%define x1  qword  [EBP + 8]
%define x2  qword  [EBP + 16]
%define y1  qword  [EBP + 24]
%define y2  qword  [EBP + 32]
%define d   dword  [EBP + 40]
```

Distance Function

See ics312_fp.asm on Web site

```
segment . text
        global _distance
_distance:
        push    ebp                     ; save ebp
        mov     ebp, esp                ; set ebp to esp
        push    ebx                     ; save ebx

        fld             x1              ; stack = x1
        fsub            x2              ; stack = (x1-x2)
        fmul            st0             ; stack = (x1-x2)^2
        fld             y1              ; stack = y1   (x1-x2)^2
        fsub            y2              ; stack = (y1-y2) (x1-x2)^2
        fmul            st0             ; stack = (y1-y2)^2 (x1-x2)^2
        fadd            st1             ; stack = d^2  (x1-x2)^2
        fsqrt                           ; stack = d    (x1-x2)^2
        mov             ebx, d          ; ebx = &d
        fst             qword [ebx]     ; write d to memory

        pop             ebx             ; restore ebx
        pop             ebp             ; restore ebp
        ret                             ; return
```

Solving a Quadratic Equation

- The book has a good example about solving a quadratic equation
- You should definitely study it...

FP Constants

- NASM does not allow us to do something like:

```
        fld     3.14
```

- The way to do this is declare all constants in the data segment:

```
segment .data
        pi      dd      3.14            ; float
segment .text
        . . .
        fld     dword [pi]
        . . .
```

The dump_math Macro

- asm_io.* implements a “dump_math” macro that works just like dump_regs
- It prints the FP stack
- We’ll see it in use on our next example

- This is what you should use to debug your code when doing the assignment

FP Return values

- Convention
  - The Callee: places return values of floating point types in ST0
  - The Caller: finds that the return value has been pushed onto the FP stack

- Let’s look at an example

FP Return Value Example

```
%include "asm_io.inc"
segment .data
    pi     dd     3.14
segment .bss
    tmp    resd   1
segment .text
    global  asm_main

asm_main:
    enter 0,0
    pusha

    fld    dword [pi]          ; ST0 = 3.14
    fld1                       ; ST0 = 1, ST1 = 3.14
    fld1                       ; ST0 = 1, ST1 = 1, ST2 = 3.14
    fadd   st2                 ; ST0 = 4.14, ST1 = 1, ST2 = 3.14
    fst    dword [tmp]         ; tmp = 4.14
    dump_math 0                ; print the FP stack
    push   dword [tmp]         ; put 4.14 on the stack
    call   some_func           ; call some_func
    add    esp, 4              ; clean up the stack
    dump_math 1                ; print the FP stack

    popa
    mov    eax, 0
    leave
    ret
```

```
some_func:
    push   ebp                 ; same ebp on the stack
    mov    ebp, esp            ; set ebp = esp

    fld    dword [ebp+8]       ; put the parameter on FP stack
                               ; as if it were a return value
    fld1                       ; put 1 on the stack
    dump_math 2                ; print the FP stack
    faddp  st1                 ; add and pop
    dump_math 3                ; print the FP stack

    mov    esp, ebp            ; reset esp
    pop    ebp                 ; restore ebp
    ret                        ; return
```

see ics312_fp2.asm on Web site

Let’s run it...

FP Return Values

- So, when the homework asks that you write a function that returns a FP value, make sure that you put that value in ST0 right before returning
- When you call a function that returns a FP value (e.g., one that is compiled by gcc), expect its return value in ST0

Conclusion

- FP calculations in x86 are not the most convenient thing
  - Overuse of the dump_math macro is HIGHLY recommended!!

- We’ll have a homework with some FP calculations