Real Arithmetic

Computer Organization and Assembly Languages
Yung-Yu Chuang

Binary real numbers

- Binary real to decimal real
  \[110.011_2 = 4 + 2 + 0.25 + 0.125 = 6.375\]

- Decimal real to binary real
  \[
  \begin{align*}
  0.5625 \times 2 &= 1.125 & \text{first bit} &= 1 \\
  0.125 \times 2 &= 0.25 & \text{second bit} &= 0 \\
  0.25 \times 2 &= 0.5 & \text{third bit} &= 0 \\
  0.5 \times 2 &= 1.0 & \text{fourth bit} &= 1 \\
  \end{align*}
  \]

  \[4.5625 = 100.1001_2\]

Fractional binary numbers

- Representation
  - Bits to right of “binary point” represent fractional powers of 2
  - Represents rational number:
    \[
    \sum_{k=j}^{i} b_k \cdot 2^k
    \]

Fractional binary numbers examples

- Value | Representation
  --- | ---
  5/3 | \(101.11_2\)
  27/8 | \(10.111_2\)
  63/64 | \(0.1111111_2\)

- Value | Representation
  --- | ---
  1/3 | \(0.010101010101_2\)
  1/5 | \(0.001100110011_2\)
  1/10 | \(0.0001100110011_2\)
**Fixed-point numbers**

<table>
<thead>
<tr>
<th>sign</th>
<th>integer part</th>
<th>fractional part</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0000 0000</td>
<td>0110 0110 0000 0000 0000 = 110.011</td>
</tr>
</tbody>
</table>

- only $2^{16}$ to $2^{-16}$  
  Not flexible, not adaptive to applications  
- Fast computation, just integer operations. It is often a good way to speed up in this way if you know the working range beforehand.

**IEEE floating point**

- IEEE Standard 754  
  - Established in 1985 as uniform standard for floating point arithmetic  
  - Before that, many idiosyncratic formats  
  - Supported by all major CPUs  
- Driven by Numerical Concerns  
  - Nice standards for rounding, overflow, underflow  
  - Hard to make go fast  
  - Numerical analysts predominated over hardware types in defining standard

**IEEE floating point format**

- IEEE defines two formats with different precisions: single and double

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>23</th>
<th>22</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>e</td>
<td>f</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- s: sign bit - 0 = positive, 1 = negative  
- e: biased exponent (8-bits) = true exponent + 7F (127 decimal). The values 00 and FF have special meaning (see text).  
- f: fraction - the first 23-bits after the 1. in the significand.

23.85 = 10111.110110₂ = 1.0111110110x2⁴  
$e = 127 + 4 = 83h$

**special values**

<table>
<thead>
<tr>
<th>63</th>
<th>62</th>
<th>59</th>
<th>51</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>e</td>
<td>f</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

IEEE double precision

- $e = 0$ and $f = 0$ denotes the number zero (which cannot be normalized) Note that there is a +0 and -0.  
- $e = 0$ and $f \neq 0$ denotes a *denormalized number*. These are discussed in the next section.  
- $e = FF$ and $f = 0$ denotes infinity ($\infty$). There are both positive and negative infinities.  
- $e = FF$ and $f \neq 0$ denotes an undefined result, known as NaN (Not a Number).
Denormalized numbers

- Number smaller than $1.0 \times 2^{-126}$ can’t be presented by a single with normalized form. However, we can represent it with denormalized format.
- $1.0000..00 \times 2^{-126}$ the least “normalized” number
- $0.1111..11 \times 2^{-126}$ the largest “denormalized” number
- $1.001 \times 2^{-129} = 0.001001 \times 2^{-126}$

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Summary of Real Number Encodings

<table>
<thead>
<tr>
<th>+</th>
<th>-</th>
<th>NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-∞</td>
<td>-Normalized</td>
<td>-Denorm</td>
</tr>
<tr>
<td>+Denorm</td>
<td>+Normalized</td>
<td>+∞</td>
</tr>
</tbody>
</table>

(3.14 + 1e20) - 1e20 = 0
3.14 + (1e20 - 1e20) = 3.14

IA-32 floating point architecture

- Original 8086 only has integers. It is possible to simulate real arithmetic using software, but it is slow.
- 8087 floating-point processor (and 80287, 80387) was sold separately at early time.
- Since 80486, FPU (floating-point unit) was integrated into CPU.

FPU data types

- Three floating-point types

Single-Precision Floating-Point

Sign  | Exponent | Fraction |
51 30  | 23 22    | 0       |

Double-Precision Floating-Point

Sign  | Exponent | Fraction |
63 52  | 51 0     | 0       |

Double Extended-Precision Floating-Point

Sign  | Exponent | Fraction |
79 59  | 64 38    | 0       |
FPU data types

• Four integer types
  - Four integer types

FPU registers

• Data register
• Control register
• Status register
• Tag register

Data registers

• Load: push, TOP--
• Store: pop, TOP++
• Instructions access the stack using ST(\text{i}) relative to TOP
• If TOP=0 and push, TOP wraps to R7
• If TOP=7 and pop, TOP wraps to R0
• When overwriting occurs, generate an exception
• Real values are transferred to and from memory and stored in 10-byte temporary format. When storing, convert back to integer, long, real, long real.

Postfix expression

• (5*6) - 4 \rightarrow 5 \ 6 \ * \ 4 \ -
Special-purpose registers

- Last data pointer stores the memory address of the operand for the last non-control instruction. Last instruction pointer stored the address of the last non-control instruction. Both are 48 bits, 32 for offset, 16 for segment selector.

Rounding

- FPU attempts to round an infinitely accurate result from a floating-point calculation
  - Round to nearest even: round toward to the closest one; if both are equally close, round to the even one
  - Round down: round toward to -∞
  - Round up: round toward to +∞
  - Truncate: round toward to zero

Example

- suppose 3 fractional bits can be stored, and a calculated value equals +1.0111.
  - rounding up by adding .0001 produces 1.100
  - rounding down by subtracting .0001 produces 1.011
Rounding

<table>
<thead>
<tr>
<th>method</th>
<th>original value</th>
<th>rounded value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Round to nearest even</td>
<td>1.0111</td>
<td>1.100</td>
</tr>
<tr>
<td>Round down</td>
<td>1.0111</td>
<td>1.011</td>
</tr>
<tr>
<td>Round up</td>
<td>1.0111</td>
<td>1.100</td>
</tr>
<tr>
<td>Truncate</td>
<td>1.0111</td>
<td>1.011</td>
</tr>
</tbody>
</table>

Floating-Point Exceptions

- Six types of exception conditions
  - #I: Invalid operation
  - #Z: Divide by zero (detect before execution)
  - #D: Denormalized operand
  - #O: Numeric overflow (detect after execution)
  - #U: Numeric underflow
  - #P: Inexact precision

- Each has a corresponding mask bit
  - if set when an exception occurs, the exception is handled automatically by FPU
  - if clear when an exception occurs, a software exception handler is invoked

Status register

FPU data types

.data
bigVal REAL10 1.212342342234234243E+864
.code
fld bigVal

Table 17-11 Intrinsic Data Types.

<table>
<thead>
<tr>
<th>Type</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>QWORD</td>
<td>64-bit integer</td>
</tr>
<tr>
<td>BYTE</td>
<td>80-bit (10-byte) integer</td>
</tr>
<tr>
<td>REAL4</td>
<td>32-bit (4-byte) IEEE short real</td>
</tr>
<tr>
<td>REAL8</td>
<td>64-bit (8-byte) IEEE long real</td>
</tr>
<tr>
<td>REAL10</td>
<td>80-bit (10-byte) IEEE extended real</td>
</tr>
</tbody>
</table>
FPU instruction set

• Instruction mnemonics begin with letter F
• Second letter identifies data type of memory operand
  - B = bcd
  - I = integer
  - no letter: floating point
• Examples
  - FBLD  load binary coded decimal
  - FISTP  store integer and pop stack
  - FMUL  multiply floating-point operands

FPU instruction set

• Fop {destination}, {source}
• Operands
  - zero, one, or two
    • fadd
    • fadd [a]
    • fadd st, st(1)
  - no immediate operands
  - no general-purpose registers (EAX, EBX, ...), (FSTSW is the only exception which stores FPU status word to AX)
  - destination must be a stack register
  - integers must be loaded from memory onto the stack and converted to floating-point before being used in calculations

Classic stack (0-operand)

• ST(0) as source, ST(1) as destination. Result is stored at ST(1) and ST(0) is popped, leaving the result on the top. (with 0 operand, fadd=faddp)

| f1d op1 | ; op1 = 20.0 |
| f1d op2 | ; op2 = 100.0 |
| fadd    | |

Memory operand (1-operand)

• ST(0) as the implied destination. The second operand is from memory.

| FADD mySingle | ; ST(0) = ST(0) + mySingle |
| FADD mySingle | ; ST(0) = ST(0) + mySingle |
| FSUB mySingle | ; ST(0) = ST(0) - mySingle |
| FSUBR mySingle | ; ST(0) = mySingle - ST(0) |
| FADD myInteger | ; ST(0) = ST(0) + myInteger |
| PISUB myInteger | ; ST(0) = ST(0) - myInteger |
| PISUBR myInteger | ; ST(0) = myInteger - ST(0) |
Register operands (2-operand)

- Register: operands are FP data registers, one must be ST.
  - FADD st, st(1) ; ST(0) = ST(0) + ST(1)
  - FDIVR st, st(3) ; ST(0) = ST(3) / ST(0)
  - FMUL st(2), st ; ST(2) = ST(2) * ST(0)

- Register pop: the same as register with a ST pop afterwards.
  - FADDP at(1), st

Example: evaluating an expression

```
INCLUDE Irvine32.inc
.data
array REAL4 6.0, 2.0, 4.5, 3.2
dotProduct REAL4 ?

.code
main PROC
    ; push 6.0 onto the stack
    fld array
    ; push 4.5 onto the stack
    fld array + 8
    ; push 4.5 onto the stack
    fmul array + 12
    ; pop stack into memory operand
    fadd
    ; pop stack into memory operand
    fadd
    ; pop stack into memory operand
    fadd
    ; pop stack into memory operand
    fadd
```

Load

- `FLD source` loads a floating point number from memory onto the top of the stack. The `source` may be a single, double or extended precision number or a coprocessor register.
- `FILD source` reads an integer from memory, converts it to floating point and stores the result on top of the stack. The `source` may be either a word, double word or quad word.
- `FLD1` stores a one on the top of the stack.
- `FLD0` stores a zero on the top of the stack.
- `FLDPI` stores \( \pi \)
- `FLDL2T` stores \( \log_2(10) \)
- `FLDL2E` stores \( \log_2(e) \)
- `FLDLG2` stores \( \log_{10}(2) \)
- `FLDLN2` stores \( \ln(2) \)
load

.data
array REAL8 10 DUP(?)

.code
fld array ; direct
fld [array+16] ; direct-offset
fld REAL8 PTR[esi] ; indirect
fld array[esi] ; indexed
fld array[esi*8] ; indexed, scaled
fld REAL8 PTR[ebx+esi]; base-index
fld array[ebx+esi] ; base-index-displacement

Store

fst  dblOne ; 200.0
fst  dblTwo  ; 200.0
fstp dblThree ; 200.0
fstp  dblFour ; 32.0

<table>
<thead>
<tr>
<th>ST(0)</th>
<th>200.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST(1)</td>
<td>32.0</td>
</tr>
</tbody>
</table>

Table 17-12 Basic Floating-Point Arithmetic Instructions.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCHS</td>
<td>Change sign</td>
</tr>
<tr>
<td>FADD</td>
<td>Add source to destination</td>
</tr>
<tr>
<td>FSUB</td>
<td>Subtract source from destination</td>
</tr>
<tr>
<td>FSUBR</td>
<td>Subtract destination from source</td>
</tr>
<tr>
<td>FMUL</td>
<td>Multiply source by destination</td>
</tr>
<tr>
<td>FDIV</td>
<td>Divide destination by source</td>
</tr>
<tr>
<td>FDIVR</td>
<td>Divide source by destination</td>
</tr>
</tbody>
</table>

FCHS ; change sign of ST
FABS ; ST=|ST|
Floating-Point add

- **FADD**
  - adds source to destination
  - No-operand version pops the FPU stack after addition

- Examples:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>ST(1)</th>
<th>ST(0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>fadd st(1), st(0)</td>
<td>234.56</td>
<td>10.1</td>
</tr>
<tr>
<td>After:</td>
<td>244.66</td>
<td>10.1</td>
</tr>
</tbody>
</table>

Floating-Point subtract

- **FSUB**
  - subtracts source from destination.
  - No-operand version pops the FPU stack after subtracting

- Example:

  ```
  fsub mySingle ; ST -= mySingle
  fsub array[edi*8] ; ST -= array[edi*8]
  ```

Floating-point multiply/divide

- **FMUL**
  - Multiplies source by destination, stores product in destination

- **FDIV**
  - Divides destination by source, then pops the stack

Miscellaneous instructions

- **FCHS**
  - ST0 = - ST0 Changes the sign of ST0

- **FABS**
  - ST0 = |ST0| Takes the absolute value of ST0

- **FSQRT**
  - ST0 = √ST0 Takes the square root of ST0

- **FScale**
  - ST0 = ST0 \times 2^{[ST1]} multiples ST0 by a power of 2 quickly. ST1 is not removed from the coprocessor stack.

- **.data**

  ```
  x REAL4 2.75
  five REAL4 5.2
  ```

- **.code**

  ```
  fld five ; ST0=5.2
  fld x ; ST0=2.75, ST1=5.2
  fscale ; ST0=2.75*32=88
  ; ST1=5.2
  ```
Example: compute distance

; compute D=sqrt(x^2+y^2)
fld x ; load x
fld st(0) ; duplicate x
fmul ; x*x

fld y ; load y
fld st(0) ; duplicate y
fmul ; y*y

fadd ; x*x+y*y
fsqrt
fst D

Example: expression

; expression: valD = -valA + (valB * valC).
.data
valA REAL8 1.5
code
fld valA ; ST(0) = valA
fchs ; change sign of ST(0)
.valB REAL8 2.5
fld valB ; load valB into ST(0)
.valC REAL8 3.0
fmul valC ; ST(0) *= valC
.valD REAL8 ? ; will be +6.0
fadd ; ST(0) += ST(1)
fstp valD ; store ST(0) to valD

Example: array sum

.data
N = 20
array REAL8 N DUP(1.0)
sum REAL8 0.0
.code
mov ecx, N
mov esi, OFFSET array
fldz ; ST0 = 0
lp:
  fadd REAL8 PTR [esi]; ST0 += *(esi)
  add esi, 8 ; move to next double
  loop lp
fstp sum ; store result

Comparisons

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition Code Bits</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCOM src</td>
<td>C3 C2 C1 C0</td>
<td>ST &gt; source</td>
</tr>
<tr>
<td>FCOMP src</td>
<td>0 0 X 0</td>
<td>ST &lt; source</td>
</tr>
<tr>
<td>FCOMPP</td>
<td>0 0 X 1</td>
<td>ST = source</td>
</tr>
<tr>
<td>FICOM src</td>
<td>1 0 X 0</td>
<td>ST or source undefined</td>
</tr>
<tr>
<td>FICOMP src</td>
<td>1 1 X 1</td>
<td>ST or source undefined</td>
</tr>
</tbody>
</table>

X = Don't care
Comparisons

- The above instructions change FPU’s status register of FPU and the following instructions are used to transfer them to CPU.

  FSTSW dest Stores the coprocessor status word into either a word in memory or the AX register.

  SAHF Stores the AH register into the FLAGS register.

  LAHF Loads the AH register with the bits of the FLAGS register.

- SAHF copies C₀ into carry, C₂ into parity and C₃ to zero. Since the sign and overflow flags are not set, use conditional jumps for unsigned integers (ja, jae, jb, jbe, je, jz).

Branching after FCOM

- Required steps:
  1. Use the FSTSW instruction to move the FPU status word into AX.
  2. Use the SAHF instruction to copy AH into the EFLAGS register.
  3. Use JA, JB, etc to do the branching.

- Pentium Pro supports two new comparison instructions that directly modify CPU’s FLAGS.

  FCOMI ST(0), src ; src=STn
  FCOMIP ST(0), src

Example

```assembly
.data
x REAL8 1.0
y REAL8 2.0
.code
; if (x>y) return 1 else return 0
fld x ; ST0 = x
fcomp y ; compare ST0 and y
fstsw ax ; move C bits into FLAGS
sahf
jna else_part ; if x not above y, ...

then_part:
    mov eax, 1
    jmp end_if
else_part:
    mov eax, 0
end_if:
```

Example: comparison
Example: comparison

```
.data
x REAL8 1.0
y REAL8 2.0
.code
; if (x>y) return 1 else return 0
fld y ; ST0 = y
fld x ; ST0 = x ST1 = y
fcomi ST(0), ST(1)

jna else_part ; if x not above y, ...
then_part:
    mov eax, 1
    jmp end_if
else_part:
    mov eax, 0
end_if:
```

Comparing for equality

- Not to compare floating-point values directly because of precision limit. For example, \( \sqrt{2.0} \times \sqrt{2.0} \neq 2.0 \)

```
<table>
<thead>
<tr>
<th>instruction</th>
<th>FPU stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>fld two</td>
<td>ST(0): +2.0000000E+000</td>
</tr>
<tr>
<td>fsqrt</td>
<td>ST(0): +1.4142135+000</td>
</tr>
<tr>
<td>fmul ST(0),ST(0)</td>
<td>ST(0): +2.0000000E+000</td>
</tr>
<tr>
<td>fsub two</td>
<td>ST(0): +4.4408921E-016</td>
</tr>
</tbody>
</table>
```

Example: quadratic formula

\[
ax^2 + bx + c = 0 \quad x_1, x_2 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}
\]

```
.data
epsilon REAL8 1.0E-12 ; difference value
val2 REAL8 0.0 ; value to compare
val3 REAL8 1.001E-13 ; considered equal to val2

.code
; if( val2 == val3 ), display "Values are equal"
    fld epsilon
    fld val2
    fsub val3
    fabs
    fcomi ST(0),ST(1)
    ja skip
mWrite <"Values are equal",0dh,0ah>
skip:
```

Example: quadratic formula

```
fild MinusFour ; stack -4
fld a ; stack: a, -4
fld c ; stack: c, a, -4
fmulp st1,ST(0) ; stack: a*c, -4
fmulp st1,ST(0) ; stack: -4*a*c
fld b
fld b ; stack: b, b, -4*a*c
fmulp st1,ST(0) ; stack: b*b, -4*a*c
faddp st1,ST(0) ; stack: b*b - 4*a*c
```
Example: quadratic formula

Example: quadratic formula

Other instructions

- **F2XM1** ; ST=2^{ST(0)}-1; ST in [-1,1]
- **FYL2X** ; ST=ST(1)\log_2(ST(0))
- **FYL2XP1** ; ST=ST(1)\log_2(ST(0)+1)

- **FPTAN** ; ST(0)=1;ST(1)=\tan(ST)
- **FPATAN** ; ST=\arctan(ST(1)/ST(0))
- **FSIN** ; ST=\sin(ST) in radius
- **FCOS** ; ST=\sin(ST) in radius
- **FSINCOS** ; ST(0)=\cos(ST);ST(1)=\sin(ST)

\[
x_1, x_2 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}
\]