Floating Point Instructions and Stack

Topics

- Floating Point Instructions
- Shallow Stack

Announcements

Buffer Lab is due Monday Oct 27 (note extension)

- Note it is due by 8 am Monday

Recitation Exercises #3 on floating point due next Monday Oct 20 in recitation

Midterms graded, return in TA office hours & recitation next Monday

- Can pick them up in TA office hours Thursday & Friday

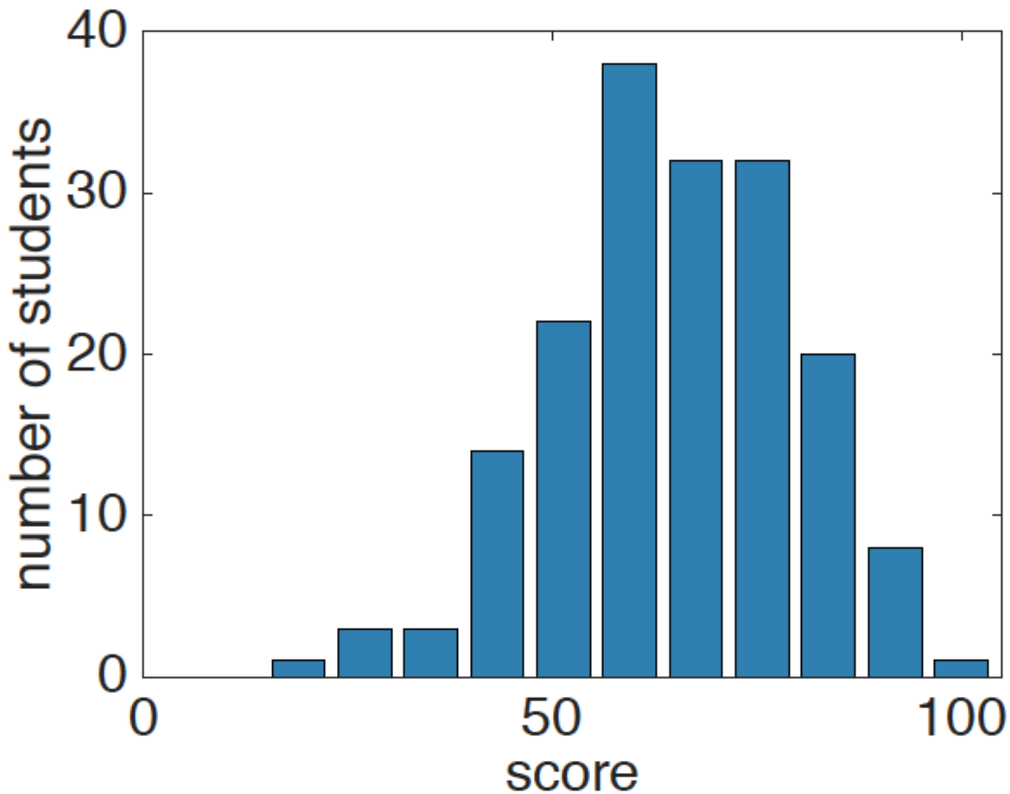
Essential that you read the textbook in detail & do the practice problems

- Chapter 2.4 Floating Point
- Then move on to Chapter 4, but Skip 4.2, 4.3.4, 4.5.9-4.5.11 (skip the PIPE implementation), 4.5.13. Overall, skipping these sections will save you about 50 pages of reading

Midterm #1 Results

The mean & median are 69, stddev is 14.3

- Hand back either in TA office hours or recitation
- Solutions available on moodle later Thursday
- Final is worth 20%, midterm is worth 12.5%



IEEE Floating Point Summary

s	exp	frac
	`e bits`	`f bits`

- MSB `s` is sign bit
- `exp` field encodes E, and is `e` bits wide
- `frac` field encodes M, and is `f` bits wide

Bias = 2^{e-1}-1, where e is # of exponent bits.

Floating point Value = $(-1)^S$ * M * 2^E, except special cases.

3 Encoding cases:

If (`exp`!=all 0's && `exp`!=all 1's): // Normalized case

E = `exp`-Bias, M = 1.`frac`, i.e. Value = $(-1)^S$ * (1.`frac`)* $2^{exp-Bias}$

Else if (`exp`==all 1's): // Special cases

if (`frac`==all 0's): Value = +/- ∞ (infinity)

else Value = NAN

Else if (`exp`==all 0's): // De-normalized case for extra precision near 0

E = 1-Bias, M = 0.`frac`, i.e. Value = $(-1)^S$ * (0.`frac`) * 2^{1-Bias}

Floating Point Arithmetic Operations

Rounding modes

- Round to zero, Round down, Round up, Round-to-nearest-even
- Needed in floating point multiplication and addition due to finite # of `frac` bits

Floating Point Multiplication

$(-1)^{s1}\ M1\ \ 2^{E1}\ \ *\ \ (-1)^{s2}\ M2\ \ 2^{E2}$

Exact Result

$(-1)^{s}\ M\ \ 2^{E}$

- **Sign** s: $s1$ ^ $s2$
- **Significand** M: $M1 * M2$
- **Exponent** E: $E1 + E2$

Floating Point Addition

$(-1)^{s1}\ M1\ \ 2^{E1}\ \ +\ \ (-1)^{s2}\ M2\ \ 2^{E2}$

- **Assume** $E1 > E2$

Exact Result

$(-1)^{s}\ M\ \ 2^{E}$

- **Sign** s, **significand** M:
  - Result of signed align & add
- **Exponent** E: $E1$

FP Addition - 8 bit example

- $9/512 + 224$
- $9/512 = 1.001 * 2^{-6} = \quad 0.000001001$
- $224 = 1.110 * 2^7 = 11100000.$
- $9/512+224 \quad = 11100000.000001001$

$= 1.1100000000001001 * 2^7$

$= 1.110 \quad * 2^7$ (3 bits `frac`)

$= 224$ (9/512 is rounded away!)

Implication of this rounding effect:

- Suppose you added 9/512 50,000 times to 224:
  Then $9/512 + 9/512 + \ldots + 9/512 + 224 > 224$

  But $9/512 + (9/512+(\ldots+(9/512+224)))))))) = 224$!!!

  So floating point addition is not associative!

FP Arithmetic and Associativity

- Floating addition is not associative:

  Example: single-precision

  $$(3.14+1e10)-1e10 \neq 3.14+(1e10-1e10)$$

  $$= 0.0 \qquad\qquad = 3.14$$

- Floating point multiplication is not associative:

  Example: single-precision

  $$(1e20*1e20)*1e\text{-}20 \neq 1e20*(1e20*1e\text{-}20)$$

  $$= +\infty \qquad\qquad = 1e20$$

  - Largest positive 32-bit single precision # is about 10^37, so 10^40 will overflow as positive infinity.

Floating Point in C

C Guarantees Two Levels

`float` single precision

`double` double precision

Conversions

- Casting between `int`, `float`, and `double` changes numeric values *and bit representations*, unlike casting between signed/unsigned `ints, shorts` and `longs`
- `Double` or `float` to `int`
  - Truncates fractional part
  - Like rounding toward zero
  - Not defined when out of range
    - » Generally saturates to TMin or TMax
- `int` to `double`
  - Exact conversion, as long as int has ≤ 53 bit word size
- `int` to `float`
  - Will round according to rounding mode

Ariane 5

- Exploded 37 seconds after liftoff
- Cargo worth $500 million

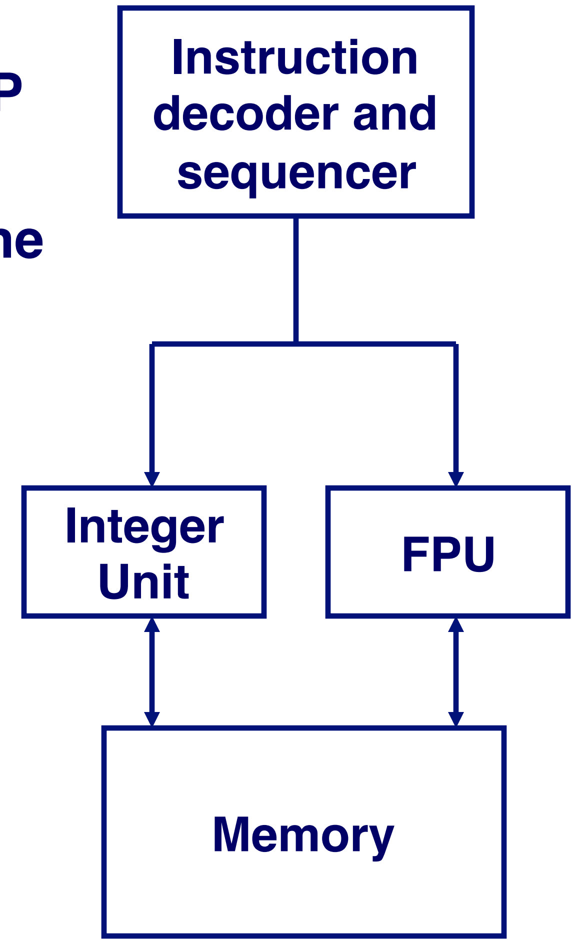
Why

- Computed horizontal velocity as floating point number
- Converted to 16-bit integer
- Worked OK for Ariane 4
- Overflowed for Ariane 5
  - Used same software, which was OK for lower velocities
  - Ariane 5 had 5X horizontal velocity of Ariane 4
- Software was written in Ada, which allows protection for overflows
  - Protections explicitly not used

IA32 Floating Point – from 3.14 too

History

- 8086: first computer to implement IEEE FP
  - separate 8087 FPU (floating point unit)
- 486: merged FPU and Integer Unit onto one chip



Summary

- Hardware to add, multiply, and divide
- Floating point data registers
- Various control & status registers

Floating Point Formats

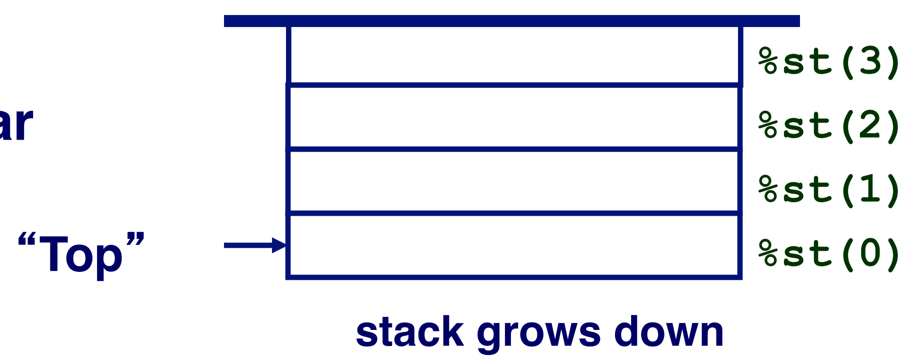
- single precision (C `float`): 32 bits
- double precision (C `double`): 64 bits
- extended precision (C `long double`): 80 bits

FPU Data Register Stack

FPU register format (extended precision)

FPU registers

- 8 registers
- Logically forms shallow stack
- Top called `%st(0)`
- When push too many, bottom values disappear



FPU instructions

Large number of floating point instructions and formats

- ~50 basic instruction types
- load, store, add, multiply
- sin, cos, tan, arctan, and log!

Sample instructions:

Instruction	Effect	Description
`fldz`	push 0.0	Load zero
`flds Addr`	push M[`Addr`]	Load single precision real
`fmuls Addr`	`%st(0) <- %st(0)`*M[`Addr`]	Multiply
`faddp`	`%st(1) <- %st(0)+%st(1); pop`	Add and pop

Floating Point Code Example

Compute Inner Product of Two Vectors

- Single precision arithmetic
- Common computation

```
float ipf (float x[],
           float y[],
           int n)
{
  int i;
  float result = 0.0;

  for (i = 0; i < n; i++) {
    result += x[i] * y[i];
  }
  return result;
}
```

```
   pushl %ebp                 # setup
   movl %esp,%ebp
   pushl %ebx

   movl 8(%ebp),%ebx          # %ebx=&x
   movl 12(%ebp),%ecx         # %ecx=&y
   movl 16(%ebp),%edx         # %edx=n
   fldz                       # push +0.0
   xorl %eax,%eax             # i=0
   cmpl %edx,%eax             # if i>=n done
   jge .L3

.L5:
   flds (%ebx,%eax,4)         # push x[i]
   fmuls (%ecx,%eax,4)        # st(0)*=y[i]
   faddp                      # st(1)+=st(0); pop
   incl %eax                  # i++
   cmpl %edx,%eax             # if i<n repeat
   jl .L5

.L3:
   movl -4(%ebp),%ebx         # finish
   movl %ebp, %esp
   popl %ebp
   ret                        # st(0) = result
```

Inner Product Stack Trace

```
...
.L5:
    flds (%ebx,%eax,4)    # push x[i]
    fmuls (%ecx,%eax,4)   # st(0)*=y[i]
    faddp                 # st(1)+=st(0); pop
    incl %eax             # i++
    cmpl %edx,%eax        # if i<n repeat
    jl .L5
...
```

Initialization

1. `fldz`

0.0	%st(0)

Iteration 0

2. `flds (%ebx,%eax,4)`

0.0	%st(1)
x[0]	%st(0)

3. `fmuls (%ecx,%eax,4)`

0.0	%st(1)
x[0]*y[0]	%st(0)

4. `faddp`

0.0+x[0]*y[0]	%st(0)

Iteration 1

5. `flds (%ebx,%eax,4)`

x[0]*y[0]	%st(1)
x[1]	%st(0)

6. `fmuls (%ecx,%eax,4)`

x[0]*y[0]	%st(1)
x[1]*y[1]	%st(0)

7. `faddp`

	%st(0)

x[0]*y[0]+x[1]*y[1]

Floating Point Summary

IEEE Floating Point Has Clear Mathematical Properties

- Represents numbers of form $M \times 2^E$
- Can reason about operations independent of implementation
  - As if computed with perfect precision and then rounded
- Not the same as real arithmetic
  - Violates associativity/distributivity
  - Makes life difficult for compilers & serious numerical applications programmers
- Conversions between float/double and int/long can cause overflow

IA32 Floating Point

- Strange “shallow stack” architecture

Major Revelations So Far…

1. Two's complement encoding and arithmetic for integers
2. Programs in high-level languages are compiled into assembly instructions and executed on the CPU
3. Assembly uses a call stack to efficiently manage function calls
4. Call stacks can be overflowed on x86 CPUs, resulting in execution of malicious code
5. Floating point representation encodes real #s as $M*2^E$, and x86 FP employs a FP register stack
6. How assembly insructions execute on a CPU, and pipelining for efficient execution

Supplementary Slides

Special Properties of Encoding

FP Zero Same as Integer Zero

- All bits = 0

Can (Almost) Use Unsigned Integer Comparison

- Must first compare sign bits
- Must consider -0 = 0
- NaNs problematic
  - Will be greater than any other values
  - What should comparison yield?
- Otherwise OK
  - Denorm vs. normalized
  - Normalized vs. infinity

Mathematical Properties of FP Add

Compare to those of Abelian Group

- Closed under addition? **YES**
  - But may generate infinity or NaN
- Commutative? **YES**
- Associative? **NO**
  - Overflow and inexactness of rounding

Example: single-precision, $(3.14+1e10)-1e10 \neq 3.14+(1e10-1e10)$

$= 0.0$ $\quad$ $= 3.14$

- 0 is additive identity? **YES**
- Every element has additive inverse **ALMOST**
  - Except for infinities & NaNs

Monotonicity

- $a \geq b \Rightarrow a+c \geq b+c$? **ALMOST**
  - Except for infinities & NaNs

Math. Properties of FP Mult

Compare to Commutative Ring

- Closed under multiplication? YES
  - But may generate infinity or NaN
- Multiplication Commutative? YES
- Multiplication is Associative? NO
  - Possibility of overflow, inexactness of rounding

  Example: single-precision, (1e20*1e20)*1e-20 ≠ 1e20*(1e20*1e-20)

  = + ∞ = 1e20
- 1 is multiplicative identity? YES
- Multiplication distributes over addition? NO — See textbook example
  - Possibility of overflow, inexactness of rounding

Monotonicity

- $a \geq b$ & $c \geq 0 \Rightarrow a * c \geq b * c$? ALMOST
  - Except for infinities & NaNs