CO20-320241

Computer Architecture and Programming Languages

CAPL

Lecture 15

Dr. Kinga Lipskoch

Fall 2019

Real Numbers

- What if we want to encode the approximate age of the earth?
  $4,600,000,000$ or $4.6x10^9$
  or the weight in kg of one a.m.u. (atomic mass unit)
  $0.00000000000000000000000000166$ or $1.6x10^{-27}$
  There is no way we can encode either of the above in a 32-bit integer
- Floating point representation
  $(-1)^{sign}$ x *fraction* x $2^{exponent}$
  Still have to fit everything in 32 bits (single precision)

IEEE 754 Floating Point (1)

s	E (exponent)	F (fraction)
1 bit	**8 bit**	**23 bit**

- The base (2, not 10) is hardwired in the design of the FPALU
- More bits in the fraction (F) or the exponent (E) is a trade-off between precision (accuracy of the number) and range (size of the number)
  - To simplify sorting FP numbers, E comes before F in the word and E is represented in excess (biased) notation
- IEEE 754 floating point standard:
  - single precision:
    8 bit exponent, 23 bit fraction
  - double precision:
    11 bit exponent, 52 bit fraction

IEEE 754 Floating Point (2)

- Form
  - Arbitrary $363.4 * 10^{34}$
  - Normalized $3.634 * 10^{36}$
- Binary notation
  - Normalized $1.xxx_{two} * 2^{yy}$
- Standardized format IEEE 754
  - Single precision: 8 bit exp, 23 bit fraction
    $2 * 10^{-38} \ldots 2 * 10^{38}$
  - Double precision: 11 bit exp, 52 bit fraction
    $2 * 10^{-308} \ldots 2 * 10^{308}$
- Both formats are supported by MIPS

IEEE 754 Floating Point Standard

- Leading "1" bit of fraction is implicit
- Exponent is "biased" to make sorting easier
  - all 0s is smallest exponent all 1s is largest
  - bias of 127 for single precision and 1023 for double precision
  - summary: $(-1)^{sign} * (1 + fraction) * 2^{exponent-bias}$
- Example:
  - decimal: $-.75 = -3/4_{ten} = -11_{two}/2^2_{ten} = -0.11_{two}$
  - binary: $-.11 = -1.1x2^{-1}$
  - floating point: $exponent = 126 = 01111110$
  - IEEE 754 single precision:
    1 01111110 10000000000000000000000

Floating Point Complexities

- Operations are somewhat more complicated
- In addition to overflow we can have “underflow”
- Accuracy can be a big problem:
  - IEEE 754 keeps two extra bits, guard and round
  - four rounding modes
  - positive divided by zero yields “infinity”
  - zero divide by zero yields “not a number”
  - other complexities
- Implementing the standard is difficult
- Not using the standard can be even worse
- see text for description of 80x86 and Pentium bug

IEEE 754

- Most computers these days conform to the IEEE 754 floating point standard
- Some bit combinations have special meaning

Single Precision		Double Precision		Object Represented
E (8)	F (23)	E (11)	F (52)	
0	0	0	0	true zero (0)
0	nonzero	0	nonzero	± denormalized number
1-254	anything	1-2046	anything	± floating point number
255	0	2047	0	± infinity
255	nonzero	2047	nonzero	not a number (NaN)

IEEE 754 Specialties

- Denormalized numbers
  - exponent is 0, fraction non-zero
  - no implicit 1 in front of floating point
- Example:
  - 0 00000000 10000000000000000000000
  - is $5.877472e-39$
  - expands range for small numbers
  - reduces risk of underflow

Two Simple Test Programs

- C construct union shares memory for different representations

```
union ieee754 {
  float d;
  unsigned int an_integer;
};
```

- Allows to convert from one type to another
  - read float and then test each single bit of the variable an_integer
- str2float.c
  - converts a binary string to floating point number
- floating.c
  - shows binary representation of floating point number

How to Compute a Binary Float

```
Decimal fraction: 8.703125
Integral part: 8 → 1000

Fraction part: 0.703125
Bits:
 0 bit: 0.7031250 x 2 = 1.4062500 |1|
 1 bit: 0.4062500 x 2 = 0.8125000 |0|
 2 bit: 0.8125000 x 2 = 1.6250000 |1|
 3 bit: 0.6250000 x 2 = 1.2500000 |1|
 4 bit: 0.2500000 x 2 = 0.5000000 |0|
 5 bit: 0.5000000 x 2 = 1.0000000 |1|

1000.1011010000000000000000000 x 2^0
1.0001011010000000000000000000 x 2^3
```

Number now needs to be normalized and exponent needs to be determined $(127 + 3)$

Result:

0 10000010 00010110100000000000000

- Split number in integral and fractional part
- Compute bit pattern for integral part
- For the fractional part multiply by 2
- If result has bit set left of decimal point ($>= 1.0$) generate set bit, otherwise generate zero bit
- Continue with rest until result does not have rest