inst.eecs.berkeley.edu/~cs61c

CS61CL : Machine Structures

Lecture #6 – Number Representation, IEEE FP

2009-07-07

Jeremy Huddleston

Review: Decimal (base 10) Numbers

Digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Example:

3271 =

$(3x10^3) + (2x10^2) + (7x10^1) + (1x10^0)$

Review: Hexadecimal (base 16) Numbers

- Hexadecimal:
  0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
  - Normal digits + 6 more from the alphabet
  - In C, written as 0x… (e.g., 0xFAB5)
- Conversion: Binary⇔Hex
  - 1 hex digit represents 16 decimal values
  - 4 binary digits represent 16 decimal values
  - ⇒1 hex digit replaces 4 binary digits
- One hex digit is a "nibble". Two is a "byte"
  - 2 bits is a "half-nibble". Shave and a haircut…
- Example:
  - 1010 1100 0011 (binary) = 0x_____ ?

What to do with representations of numbers?

- Just what we do with numbers!
  - Add them
  - Subtract them
  - Multiply them
  - Divide them
  - Compare them
- Example: 10 + 7 = 17
  - …so simple to add in binary that we can build circuits to do it!
  - subtraction just as you would in decimal
  - Comparison: How do you tell if X > Y ?

```
      1  1
      1  0  1  0
 +    0  1  1  1
------------------
   1  0  0  0  1
```


Which base do we use?

- Decimal: great for humans, especially when doing arithmetic
- Hex: if human looking at long strings of binary numbers, its much easier to convert to hex and look 4 bits/symbol
  - Terrible for arithmetic on paper
- Binary: what computers use; you will learn how computers do +, -, *, /
  - To a computer, numbers always binary
  - Regardless of how number is written:
  - 32_{ten} == 32_{10} == 0x20 == 100000_2 == 0b100000
  - Use subscripts "ten", "hex", "two" in book, slides when might be confusing

BIG IDEA: Bits can represent anything!!

- Characters?
  - 26 letters ⇒ 5 bits (2^5 = 32)
  - upper/lower case + punctuation ⇒ 7 bits (in 8) ("ASCII")
  - standard code to cover all the world's languages ⇒ 8,16,32 bits ("Unicode") `www.unicode.com`
- Logical values?
  - 0 ⇒ False, 1 ⇒ True
- colors ? Ex: Red (00) Green (01) Blue (11)
- locations / addresses? commands?
- MEMORIZE: N bits ⇔ at most 2^N things

How to Represent Negative Numbers?

- So far, unsigned numbers
- Obvious solution: define leftmost bit to be sign!
  - 0 ⇒ +, 1 ⇒ –
  - Rest of bits can be numerical value of number
- Representation called sign and magnitude
- MIPS uses 32-bit integers. $+1_{ten}$ would be:

  0000 0000 0000 0000 0000 0000 0000 0001
- And $–1_{ten}$ in sign and magnitude would be:

  1000 0000 0000 0000 0000 0000 0000 0001

Shortcomings of sign and magnitude?

- Arithmetic circuit complicated
  - Special steps depending whether signs are the same or not
- Also, two zeros
  - 0x00000000 = $+0_{ten}$
  - 0x80000000 = $–0_{ten}$
  - What would two 0s mean for programming?
- Therefore sign and magnitude abandoned

Another try: complement the bits

- Example: $7_{10} = 00111_2$ $-7_{10} = 11000_2$
- Called One's Complement
- Note: positive numbers have leading 0s, negative numbers have leadings 1s.

```
                00000   00001 ...   01111
<---------------+------------------------>
10000 ... 11110 11111
```

- What is -00000 ? Answer: 11111
- How many positive numbers in N bits?
- How many negative numbers?

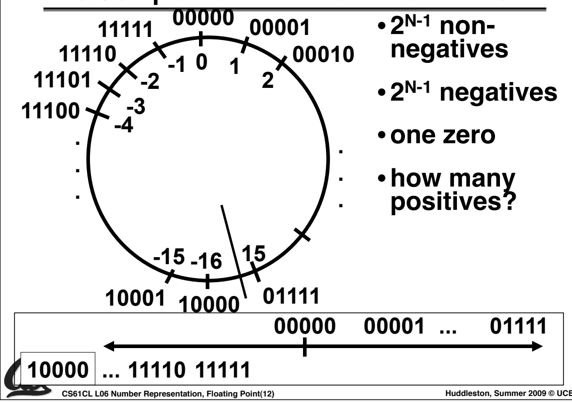
Shortcomings of One's complement?

- Arithmetic still a somewhat complicated.
- Still two zeros
  - `0x00000000` = $+0_{ten}$
  - `0xFFFFFFFF` = -0_{ten}
- Although used for awhile on some computer products, one's complement was eventually abandoned because another solution was better.

Standard Negative Number Representation

- What is result for unsigned numbers if tried to subtract large number from a small one?
  - Would try to borrow from string of leading 0s, so result would have a string of leading 1s
    - 3 - 4 ⇒ 00…0011 – 00…0100 = 11…1111
  - With no obvious better alternative, pick representation that made the hardware simple
  - As with sign and magnitude, leading 0s ⇒ positive, leading 1s ⇒ negative
    - 000000...xxx is ≥ 0, 111111...xxx is < 0
    - except 1…1111 is -1, not -0 (as in sign & mag.)
- This representation is <u>Two's Complement</u>

2's Complement Number "line": N = 5



- 2^{N-1} non-negatives
- 2^{N-1} negatives
- one zero
- how many positives?

Two's Complement Formula

- Can represent positive <u>and negative</u> numbers in terms of the bit value times a power of 2:

$$d_{31} \times -(2^{31}) + d_{30} \times 2^{30} + \ldots + d_2 \times 2^2 + d_1 \times 2^1 + d_0 \times 2^0$$

- Example: 1101_{two}

$= 1\times-(2^3) + 1\times2^2 + 0\times2^1 + 1\times2^0$

$= -2^3 + 2^2 + 0 + 2^0$

$= -8 + 4 + 0 + 1$

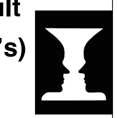
$= -8 + 5$

$= -3_{ten}$

Two's Complement shortcut: Negation

*Check out `www.cs.berkeley.edu/~dsw/twos_complement.html`

- Change every 0 to 1 and 1 to 0 (invert or complement), then add 1 to the result
- Proof*: Sum of number and its (one's) complement must be $111...111_{two}$



However, $111...111_{two} = -1_{ten}$

Let x' ⇒ one's complement representation of x

Then x + x' = -1 ⇒ x + x' + 1 = 0 ⇒ $\boxed{-x = x' + 1}$

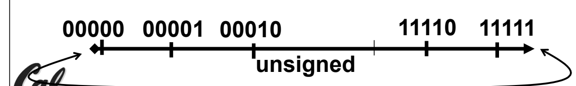
- Example: -3 to +3 to -3

x : $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two}$
x': $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two}$
+1: $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0011_{two}$
()': $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_{two}$
+1: $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two}$

You should be able to do this in your head…

What if too big?

- Binary bit patterns above are simply <u>representatives</u> of numbers. Strictly speaking they are called "numerals".
- Numbers really have an ∞ number of digits
  - with almost all being same (00…0 or 11…1) except for a few of the rightmost digits
  - Just don't normally show leading digits
- If result of add (or -, *, /) cannot be represented by these rightmost HW bits, <u>overflow</u> is said to have occurred.



What about other numbers?

1. Very large numbers? (seconds/millennium) ⇒ $31{,}556{,}926{,}000_{10}$ ($3.1556926_{10} \times 10^{10}$)
2. Very small numbers? (Bohr radius) ⇒ 0.0000000000529177_{10}m ($5.29177_{10} \times 10^{-11}$)
3. Numbers with <u>both</u> integer & fractional parts? ⇒ 1.5

First consider #3.

…our solution will also help with 1 and 2.

Representation of Fractions

"Binary Point" like decimal point signifies boundary between integer and fractional parts:

Example 6-bit representation: xx.yyyy

2^1 2^0 2^{-1} 2^{-2} 2^{-3} 2^{-4}

$10.1010_2 = 1\times2^1 + 1\times2^{-1} + 1\times2^{-3} = 2.625_{10}$

If we assume "fixed binary point", range of 6-bit representations with this format:

0 to 3.9375 (almost 4)

Fractional Powers of 2

i	2^{-i}	
0	1.0	1
1	0.5	1/2
2	0.25	1/4
3	0.125	1/8
4	0.0625	1/16
5	0.03125	1/32
6	0.015625	
7	0.0078125	
8	0.00390625	
9	0.001953125	
10	0.0009765625	
11	0.00048828125	
12	0.000244140625	
13	0.0001220703125	
14	0.00006103515625	
15	0.000030517578125	

Representation of Fractions with Fixed Pt.

What about addition and multiplication?

Addition is straightforward:

```
01.100   1.5₁₀
00.100   0.5₁₀
10.000   2.0₁₀
```

Multiplication a bit more complex:

```
      01.100   1.5₁₀
      00.100   0.5₁₀
       00 000
      000 00
     0110 0
    00000
   00000
  0000110000
   HI   LOW
```

Where's the answer, `0.11`? (need to remember where point is)

Representation of Fractions

So far, in our examples we used a "fixed" binary point what we really want is to "float" the binary point. Why?

Floating binary point most effective use of our limited bits (and thus more accuracy in our number representation):

example: put 0.1640625 into binary. Represent as in 5-bits choosing where to put the binary point.

… 000000.001010100000…

Store these bits and keep track of the binary point 2 places to the left of the MSB

Any other solution would lose accuracy!

With floating point rep., each numeral carries a exponent field recording the whereabouts of its binary point.

The binary point can be outside the stored bits, so very large and small numbers can be represented.

Scientific Notation (in Decimal)

mantissa → $6.02_{10} \times 10^{23}$ ← **exponent**

decimal point **radix (base)**

- **Normalized form: no leadings 0s (exactly one digit to left of decimal point)**
- **Alternatives to representing 1/1,000,000,000**
  - **Normalized:** 1.0×10^{-9}
  - **Not normalized:** $0.1 \times 10^{-8}, 10.0 \times 10^{-10}$

Scientific Notation (in Binary)

mantissa → $1.0_{two} \times 2^{-1}$ ← **exponent**

"binary point" **radix (base)**

- **Computer arithmetic that supports it called <u>floating point</u>, because it represents numbers where the binary point is not fixed, as it is for integers**
  - **Declare such variable in C as `float`**

Floating Point Representation (1/2)

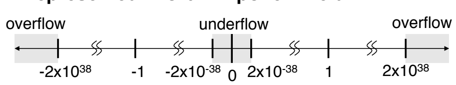
- **Normal format: $+1.\text{xxxxxxxxxx}_{two} * 2^{\text{yyyy}_{two}}$**
- **Multiple of Word Size (32 bits)**

31	30 … 23	22 … 0
S	Exponent	Significand
1 bit	8 bits	23 bits

- **S represents Sign**
  Exponent represents y's
  Significand represents x's
- **Represent numbers as small as 2.0×10^{-38} to as large as 2.0×10^{38}**

Floating Point Representation (2/2)

- **What if result too large?**
  ($> 2.0\times10^{38}$, $< -2.0\times10^{38}$)
  - **<u>Overflow!</u> ⇒ Exponent larger than represented in 8-bit Exponent field**
- **What if result too small?**
  (>0 & $< 2.0\times10^{-38}$, <0 & $> -2.0\times10^{-38}$)
  - **<u>Underflow!</u> ⇒ Negative exponent larger than represented in 8-bit Exponent field**



- **What would help reduce chances of overflow and/or underflow?**

Double Precision Fl. Pt. Representation

- **Next Multiple of Word Size (64 bits)**

31	30 … 20	19 … 0
S	Exponent	Significand
1 bit	11 bits	20 bits
Significand (cont'd)		
32 bits		

- **<u>Double Precision</u> (vs. <u>Single Precision</u>)**
  - **C variable declared as `double`**
  - **Represent numbers almost as small as 2.0×10^{-308} to almost as large as 2.0×10^{308}**
  - **But primary advantage is greater accuracy due to larger significand**

QUAD Precision Fl. Pt. Representation

- **Next Multiple of Word Size (128 bits)**
  - **Unbelievable range of numbers**
  - **Unbelievable precision (accuracy)**
- **IEEE 754-2008, Finalized Aug 2008**
  - **15 exponent bits**
  - **112 significand bits (113 precision bits)**
- **Oct-Precision?**
  - **Some have tried, no real traction so far**
- **Half-Precision?**
  - **Yep, that's for a short (16 bit)**

`en.wikipedia.org/wiki/Quad_precision`
`en.wikipedia.org/wiki/Half_precision`

IEEE 754 Floating Point Standard (1/3)

Single Precision (DP similar):

31	30 … 23	22 … 0
S	Exponent	Significand
1 bit	8 bits	23 bits

- **Sign bit:** **1 means negative**
  0 means positive
- **Significand:**
  - **To pack more bits, leading 1 implicit for normalized numbers**
  - **1 + 23 bits single, 1 + 52 bits double**
  - **always true: 0 < Significand < 1 (for normalized numbers)**
- **Note: reserve exponent value 0 to mean no implicit leading 1 (eg: 0)**

IEEE 754 Floating Point Standard (2/3)

- IEEE 754 uses “biased exponent” representation.
  - Designers wanted FP numbers to be used even if no FP hardware; e.g., sort records with FP numbers using integer compares
  - Wanted bigger (integer) exponent field to represent bigger numbers.
  - 2’s complement poses a problem (because negative numbers look bigger)
  - We’re going to see that the numbers are ordered EXACTLY as in sign-magnitude
    - I.e., counting from binary odometer 00…00 up to 11…11 goes from 0 to +MAX to -0 to -MAX to 0

IEEE 754 Floating Point Standard (3/3)

- Called Biased Notation, where bias is number subtracted to get real number
  - IEEE 754 uses bias of 127 for single prec.
  - Subtract 127 from Exponent field to get actual value for exponent
  - 1023 is bias for double precision
- Summary (single precision):

31	30 … 23	22 … 0
S	Exponent	Significand
1 bit	8 bits	23 bits

- $(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$
  - Double precision identical, except with exponent bias of 1023 (half, quad similar)

Example: Converting Binary FP to Decimal

0	0110 1000	101 0101 0100 0011 0100 0010

- Sign: 0 => positive
- Exponent:
  - $0110\ 1000_{two} = 104_{ten}$
  - Bias adjustment: 104 - 127 = -23
- Significand:

  $1 + 1x2^{-1} + 0x2^{-2} + 1x2^{-3} + 0x2^{-4} + 1x2^{-5} + \ldots$
  $= 1 + 2^{-1} + 2^{-3} + 2^{-5} + 2^{-7} + 2^{-9} + 2^{-14} + 2^{-15} + 2^{-17} + 2^{-22}$
  $= 1.0 + 0.666115$
- Represents: $1.666115_{ten} * 2^{-23} \sim 1.986 * 10^{-7}$ (about 2/10,000,000)

Example: Converting Decimal to FP

-2.340625×10^1

1. Denormalize: -23.40625
2. Convert integer part:
   $23 = 16 + (7 = 4 + (3 = 2 + (1))) = 10111_2$
3. Convert fractional part:
   $.40625 = .25 + (.15625 = .125 + (.03125)) = .01101_2$
4. Put parts together and normalize:
   $10111.01101 = 1.011101101 \times 2^4$
5. Convert exponent: $127 + 4 = 10000011_2$

1	1000 0011	011 1011 0100 0000 0000 0000

Understanding the Significand (1/2)

- Method 1 (Fractions):
  - In decimal: 0.340_{10}
    $\Rightarrow 340_{10}/1000_{10}$
    $\Rightarrow 34_{10}/100_{10}$
  - In binary: $0.110_2 \Rightarrow 110_2/1000_2 = 6_{10}/8_{10}$
    $\Rightarrow 11_2/100_2 = 3_{10}/4_{10}$
  - Advantage: less purely numerical, more thought oriented; this method usually helps people understand the meaning of the significand better

Understanding the Significand (2/2)

- Method 2 (Place Values):
  - Convert from scientific notation
  - In decimal: $1.6732 = (1x10^0) + (6x10^{-1}) + (7x10^{-2}) + (3x10^{-3}) + (2x10^{-4})$
  - In binary: $1.1001 = (1x2^0) + (1x2^{-1}) + (0x2^{-2}) + (0x2^{-3}) + (1x2^{-4})$
  - Interpretation of value in each position extends beyond the decimal/binary point
  - Advantage: good for quickly calculating significand value; use this method for translating FP numbers

Precision and Accuracy

Don’t confuse these two terms!

Precision is a count of the number bits used to represent a value.

Accuracy is a measure of the difference between the actual value of a number and its computer representation.

High precision permits high accuracy but doesn’t guarantee it. It is possible to have high precision but low accuracy.

Example: float pi = 3.14159…; pi will be represented using all 24 bits of the significant (highly precise), but is only an approximation (not accurate).

Representation for ± ∞

- In FP, divide by 0 should produce ± ∞, not overflow.
- Why?
  - OK to do further computations with ∞ E.g., X/0 > Y may be a valid comparison
  - Ask math majors
- IEEE 754 represents ± ∞
  - Most positive exponent reserved for ∞
  - Significands all zeroes

Representation for 0

- Represent 0?
  - exponent all zeroes
  - significand all zeroes
  - What about sign? Both cases valid.

```
+0: 0 00000000 00000000000000000000000
-0: 1 00000000 00000000000000000000000
```

Special Numbers

- What have we defined so far? (Single Precision)

Exponent	Significand	Object
0	0	0
0	nonzero	???
1-254	anything	+/- fl. pt. #
255	0	+/- ∞
255	nonzero	???

- "Waste not, want not"
  - We'll talk about Exp=0,255 & Sig!=0 later

Representation for Not a Number

- What do I get if I calculate `sqrt(-4.0)` or `0/0`?
  - If ∞ not an error, these shouldn't be either
  - Called Not a Number (NaN)
    - Exponent = all 1s (255),
    - Significand nonzero
- Why is this useful?
  - Hope NaNs help with debugging?
  - They contaminate: op(NaN, X) = NaN

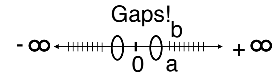
Representation for Denorms (1/2)

- Problem: There's a gap among representable FP numbers around 0
  - Smallest representable pos num:
    - $a = 1.0\ldots_2 * 2^{-126} = 2^{-126}$
  - Second smallest representable pos num:
    - $b = 1.000\ldots\ldots1_2 * 2^{-126}$
      $= (1 + 0.00\ldots1_2) * 2^{-126}$
      $= (1 + 2^{-23}) * 2^{-126}$
      $= 2^{-126} + 2^{-149}$

Normalization and implicit 1 is to blame!

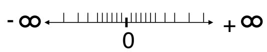
$a - 0 = 2^{-126}$

$b - a = 2^{-149}$



Representation for Denorms (2/2)

- Solution:
  - We still haven't used Exponent=0, Significand nonzero
  - Denormalized number: no (implied) leading 1, implicit exponent = -126
  - Smallest representable pos num:
    - $A = 2^{-149}$
  - Second smallest representable pos num:
    - $b = 2^{-148}$



Special Numbers Summary

- Reserve exponents, significands:

Exponent	Significand	Object
0	0	+/- 0
0	nonzero	+/- Denorm
1-254	anything	+/- Norm
255	0	+/- ∞
255	nonzero	NaN

Rounding

- When we perform math on floating point numbers, we have to worry about rounding to fit the result in the significand field.
- The FP hardware carries two extra bits of precision, and then round to get the proper value
- Rounding also occurs when converting:
  - double to a single precision value
  - floating point number to an integer
  - integer > ___ to floating point

IEEE FP Rounding Modes

Examples in decimal (but, of course, IEEE754 in binary)

- Round towards + ∞
  - ALWAYS round "up": 2.001 → 3, -2.001 → -2
- Round towards - ∞
  - ALWAYS round "down": 1.999 → 1, -1.999 → -2
- Truncate
  - Just drop the last bits (round towards 0)
- Unbiased (default mode). Midway? Round to even
  - Normal rounding, almost: 2.4 → 2, 2.6 → 3, 2.5 → 2, 3.5 → 4
  - Round like you learned in grade school (nearest int)
  - Except if the value is right on the borderline, in which case we round to the nearest EVEN number
  - Insures fairness on calculation
  - This way, half the time we round up on tie, the other half time we round down. Tends to balance out inaccuracies

"And in conclusion…"

- Floating Point lets us:
  - Represent numbers containing both integer and fractional parts; makes efficient use of available bits.
  - Store approximate values for very large and very small #s.
- IEEE 754 Floating Point Standard is most widely accepted attempt to standardize interpretation of such numbers (Every desktop or server computer sold since ~1997 follows these conventions)

- Summary (single precision):

31	30 … 23	22 … 0
S	Exponent	Significand
1 bit	8 bits	23 bits

- $(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$
  - Double precision identical, except with exponent bias of 1023 (half, quad similar)

"And in conclusion…"

- Reserve exponents, significands:

Exponent	Significand	Object
0	0	0
0	nonzero	Denorm
1-254	Anything	+/- fl. Pt #
255	0	+/- ∞
255	nonzero	NaN

- 4 Rounding modes (default: unbiased)
- MIPS Fl ops complicated, expensive

Bonus slides

- These are extra slides that used to be included in lecture notes, but have been moved to this, the “bonus” area to serve as a supplement.
- The slides will appear in the order they would have in the normal presentation

Bonus

Numbers: positional notation

- Number Base B ⇒ B symbols per digit:
  - Base 10 (Decimal): 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
    Base 2 (Binary): 0, 1
- Number representation:
  - $d_{31}d_{30} \ldots d_1d_0$ is a 32 digit number
  - value = $d_{31} \times B^{31} + d_{30} \times B^{30} + \ldots + d_1 \times B^1 + d_0 \times B^0$
- Binary: 0,1 (In binary digits called “bits”)
  - 0b11010 $= 1{\times}2^4 + 1{\times}2^3 + 0{\times}2^2 + 1{\times}2^1 + 0{\times}2^0$
    $= 16 + 8 + 2$
    $= 26$

    #s often written 0b…
  - Here 5 digit binary # turns into a 2 digit decimal #
  - Can we find a base that converts to binary easily?

Decimal vs. Hexadecimal vs. Binary

Examples:

1010 1100 0011 (binary)
= 0xAC3

10111 (binary)
= 0001 0111 (binary)
= 0x17

0x3F9
= 11 1111 1001 (binary)

How do we convert between hex and Decimal?

00	0	0000
01	1	0001
02	2	0010
03	3	0011
04	4	0100
05	5	0101
06	6	0110
07	7	0111
08	8	1000
09	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

MEMORIZE!

Two’s Complement for N=32

$0000 \ldots 0000\ 0000\ 0000\ 0000_{two} =$	0_{ten}
$0000 \ldots 0000\ 0000\ 0000\ 0001_{two} =$	1_{ten}
$0000 \ldots 0000\ 0000\ 0000\ 0010_{two} =$	2_{ten}
...	
$0111 \ldots 1111\ 1111\ 1111\ 1101_{two} =$	$2,147,483,645_{ten}$
$0111 \ldots 1111\ 1111\ 1111\ 1110_{two} =$	$2,147,483,646_{ten}$
$0111 \ldots 1111\ 1111\ 1111\ 1111_{two} =$	$2,147,483,647_{ten}$
$1000 \ldots 0000\ 0000\ 0000\ 0000_{two} =$	$-2,147,483,648_{ten}$
$1000 \ldots 0000\ 0000\ 0000\ 0001_{two} =$	$-2,147,483,647_{ten}$
$1000 \ldots 0000\ 0000\ 0000\ 0010_{two} =$	$-2,147,483,646_{ten}$
...	
$1111 \ldots 1111\ 1111\ 1111\ 1101_{two} =$	-3_{ten}
$1111 \ldots 1111\ 1111\ 1111\ 1110_{two} =$	-2_{ten}
$1111 \ldots 1111\ 1111\ 1111\ 1111_{two} =$	-1_{ten}

- One zero; 1st bit called <u>sign bit</u>
- 1 “extra” negative:no positive $2,147,483,648_{ten}$

Two’s comp. shortcut: Sign extension

- Convert 2’s complement number rep. using n bits to more than n bits
- Simply replicate the most significant bit (sign bit) of smaller to fill new bits
  - 2’s comp. positive number has infinite 0s
  - 2’s comp. negative number has infinite 1s
  - Binary representation hides leading bits; sign extension restores some of them
  - 16-bit -4_{ten} to 32-bit:

$1111\ 1111\ 1111\ 1100_{two}$

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_{two}$

Preview: Signed vs. Unsigned Variables

- Java and C declare integers `int`
  - Use two’s complement (signed integer)
- Also, C declaration `unsigned int`
  - Declares a unsigned integer
  - Treats 32-bit number as unsigned integer, so most significant bit is part of the number, not a sign bit

“Father” of the Floating point standard

IEEE Standard 754 for Binary Floating-Point Arithmetic.

1989 ACM Turing Award Winner!

Prof. Kahan

www.cs.berkeley.edu/~wkahan/
…/ieee754status/754story.html

FP Addition

- More difficult than with integers
- Can’t just add significands
- How do we do it?
  - De-normalize to match exponents
  - Add significands to get resulting one
  - Keep the same exponent
  - Normalize (possibly changing exponent)
- Note: If signs differ, just perform a subtract instead.

MIPS Floating Point Architecture (1/4)

- MIPS has special instructions for floating point operations:
  - Single Precision:
    `add.s, sub.s, mul.s, div.s`
  - Double Precision:
    `add.d, sub.d, mul.d, div.d`
- These instructions are far more complicated than their integer counterparts. They require special hardware and usually they can take much longer to compute.

MIPS Floating Point Architecture (2/4)

- Problems:
  - It's inefficient to have different instructions take vastly differing amounts of time.
  - Generally, a particular piece of data will not change from FP to int, or vice versa, within a program. So only one type of instruction will be used on it.
  - Some programs do no floating point calculations
  - It takes lots of hardware relative to integers to do Floating Point fast

MIPS Floating Point Architecture (3/4)

- 1990 Solution: Make a completely separate chip that handles only FP.
- Coprocessor 1: FP chip
  - contains 32 32-bit registers: `$f0`, `$f1`, …
  - most registers specified in `.s` and `.d` instruction refer to this set
  - separate load and store: `lwc1` and `swc1` ("load word coprocessor 1", "store …")
  - Double Precision: by convention, even/odd pair contain one DP FP number: `$f0`/`$f1`, `$f2`/`$f3`, … , `$f30`/`$f31`

MIPS Floating Point Architecture (4/4)

- 1990 Computer actually contains multiple separate chips:
  - Processor: handles all the normal stuff
  - Coprocessor 1: handles FP and only FP;
  - more coprocessors?… Yes, later
  - Today, cheap chips may leave out FP HW
- Instructions to move data between main processor and coprocessors:
  - `mfc0`, `mtc0`, `mfc1`, `mtc1`, etc.
- Appendix pages A-70 to A-74 contain many, many more FP operations.

Example: Representing 1/3 in MIPS

- 1/3
  - $= 0.33333\ldots_{10}$
  - $= 0.25 + 0.0625 + 0.015625 + 0.00390625 + \ldots$
  - $= 1/4 + 1/16 + 1/64 + 1/256 + \ldots$
  - $= 2^{-2} + 2^{-4} + 2^{-6} + 2^{-8} + \ldots$
  - $= 0.0101010101\ldots_2 * 2^0$
  - $= 1.0101010101\ldots_2 * 2^{-2}$
  - Sign: 0
  - Exponent = -2 + 127 = 125 = 01111101
  - Significand = 0101010101…

0	0111 1101	0101 0101 0101 0101 0101 010

Casting floats to ints and vice versa

`(int) floating_point_expression`

Coerces and converts it to the nearest integer (C uses truncation)

```
i = (int) (3.14159 * f);
```

`(float) integer_expression`

converts integer to nearest floating point

```
f = f + (float) i;
```


int → float → int

```
if (i == (int)((float) i)) {
 printf("true");
}
```

- Will not always print "true"
- Most large values of integers don't have exact floating point representations!
- What about `double`?

float → int → float

```
if (f == (float)((int) f)) {
 printf("true");
}
```

- Will not always print "true"
- Small floating point numbers (<1) don't have integer representations
- For other numbers, rounding errors

Floating Point Fallacy

- FP add associative: FALSE!
  - $x = -1.5 \times 10^{38}$, $y = 1.5 \times 10^{38}$, and $z = 1.0$
  - $x + (y + z) = -1.5\times10^{38} + (1.5\times10^{38} + 1.0) = -1.5\times10^{38} + (1.5\times10^{38}) = \underline{0.0}$
  - $(x + y) + z = (-1.5\times10^{38} + 1.5\times10^{38}) + 1.0 = (0.0) + 1.0 = \underline{1.0}$
- <u>Therefore, Floating Point add is not associative!</u>
  - Why? FP result <u>approximates</u> real result!
  - This example: 1.5×10^{38} is so much larger than 1.0 that $1.5 \times 10^{38} + 1.0$ in floating point representation is still 1.5×10^{38}