Representing Data

15-213: Introduction to Computer Systems
Recitation 3: Monday, Sept. 9th, 2013

Ian Hartwig
Section E

Welcome to Recitation

- Recitation is a place for interaction
  - If you have questions, please ask.
  - If you want to go over an example not planned for recitation, let me know.
  - We don't spend a lot of class time on using UNIX/C. Is this needed?
- We'll cover:
  - A quick recap of topics from class, especially ones we have found students struggled with in the past
  - Example problems to reinforce those topics and prepare for exams
  - Tips for labs

News

- Tutoring available in Mudge Reading Room, Tues. 8:30-11pm.
- Office hours in WeH 5207, Sun.-Thur. 5:30-8:30pm
- Data lab due thisThursday at 11:59pm

Agenda

- How do I Data Lab?
- Integers
  - Biasing division
  - Endianness
- Floating point
  - Binary fractions
  - IEEE standard
  - Example problem

How do I Data Lab?

- Step 1: Download lab files
  - All lab files are on autolab
  - Remember to also read the lab handout (“view writeup” link)
- Step 2: Work on the right machines
  - Remember to do all your lab work on Andrew or Shark machines
    - Some later labs will restrict you to just the shark machines (bomb lab, for example)
  - This includes untaring the handout. Otherwise, you may lose some permissions bits
  - If you get a permission denied error, try “chmod +x *filename*”

How do I Data Lab?

- Step 3: Edit and test
  - Bits.c is the file you're looking for
  - Remember you have 3 ways to test your solutions. See the writeup for details.
  - driver.pl runs the same tests as autolab
- Step 4: Submit
  - Unlimited submissions, but please don't use autolab in place of driver.pl
  - Must submit via web form
  - To package/download files to your computer, use "tar -cvzf *out.tar.gz in1 in2 …*" and your favorite file transfer protocol

How do I Data Lab?

- Tips
  - Write C like it's 1989
    - Declare variable at top of function
    - Make sure closing brace ("}") is in 1st column
    - We won't be using the dlc compiler for later labs
  - Be careful of operator precedence
    - Do you know what order `~a+1+b*c<<3*2` will execute in?
    - Neither do I. Use parentheses: `(~a)+1+(b*(c<<3)*2)`
  - Take advantage of special operators and values like !, 0, and T_{min}
  - Reducing ops once you're under the threshold won't get you extra points.
  - Undefined behavior
    - Like shifting by <32. See Anita's rant.

Anita's Rant

- From the Intel x86 Reference:

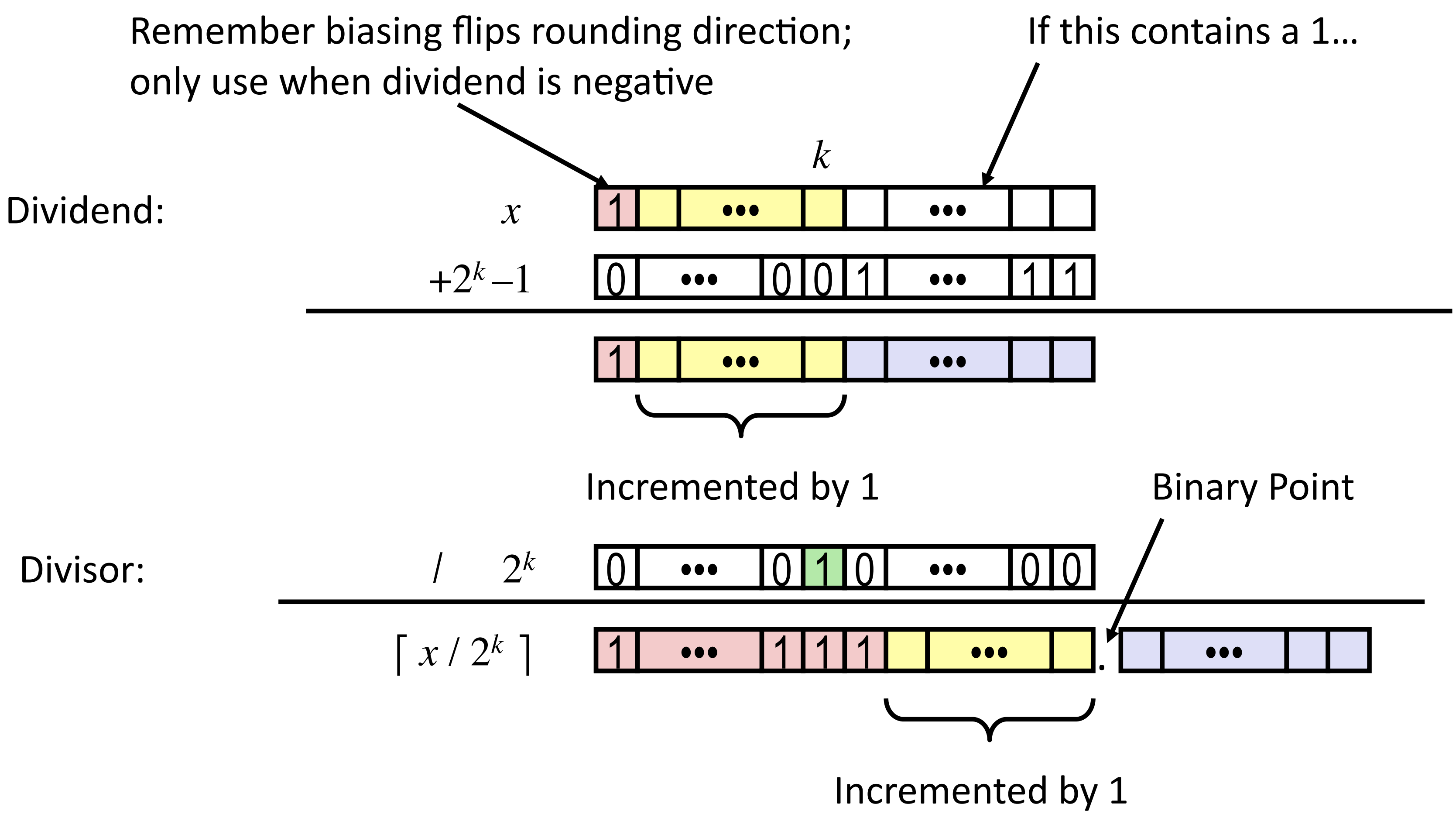
  "These instructions shift the bits in the first operand (destination operand) to the left or right by the number of bits specified in the second operand (count operand). Bits shifted beyond the destination operand boundary are first shifted into the CF flag, then discarded. At the end of the shift operation, the CF flag contains the last bit shifted out of the destination operand.

  The destination operand can be a register or a memory location. The count operand can be an immediate value or register CL. The count is masked to five bits, which limits the count range to 0 to 31. A special opcode encoding is provided for a count of 1."

Integers - Biasing

- Can multiply/divide powers of 2 with shift
  - Multiply:
    - Left shift by k to multiply by 2^k
  - Divide:
    - Right shift by k to divide by 2^k
    - … for positive numbers
    - Shifting rounds towards -inf, but we want to round to 0
    - Solution: biasing when negative

Integers - Biasing

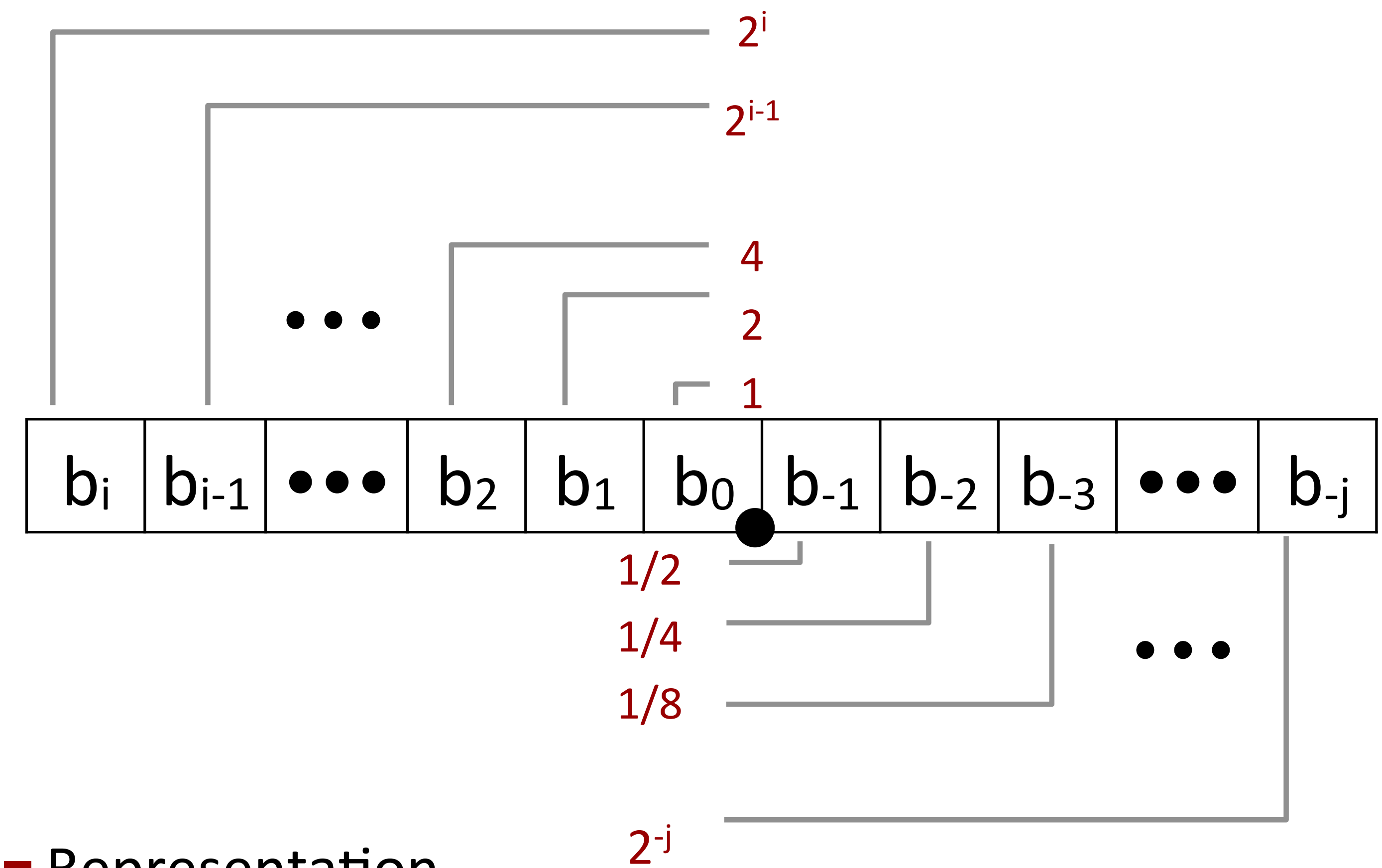


Biasing adds 1 to final result

Integers – Endianness

- Endianness describes which bit is most significant in a binary number
- You won't need to work with this until bomb lab
- Big endian:
  - First byte (lowest address) is the *most* significant
  - This is how we typically talk about binary numbers
- Little endian:
  - First byte (lowest address) is the *least* significant
  - Intel x86 (shark/andrew linux machines) implement this

Floating Point – Fractions in Binary



- Representation
  - Bits to right of "binary point" represent fractional powers of 2
  - Represents rational number: $$\sum_{k=-j}^{i} b_k \times 2^k$$

Floating Point – IEEE Standard

- Single precision: 32 bits

s	exp	frac
1	8-bits	23-bits

- Double precision: 64 bits

s	exp	frac
1	11-bits	52-bits

- Extended precision: 80 bits (Intel only)

s	exp	frac
1	15-bits	63 or 64-bits

Floating Point – IEEE Standard

- What does this mean?
  - We can think of floating point as binary scientific notation
    - IEEE format includes a few optimizations to increase range for our given number of bits
  - The number represented is *essentially* (sign * frac * 2^{exp})
    - There are a few steps I left out there
- Example:
  - Assume our floating point format has **no sign bit, k = 3 exponent bits, and n=2 fraction bits**
  - What does 0b10010 represent?

Floating Point – IEEE Standard

- What does this mean?
  - We can think of floating point as binary scientific notation
    - IEEE format includes a few optimizations to increase range for our given number of bits
  - The number represented is *essentially* (sign * frac * 2^{exp})
    - There are a few steps I left out there
- Example:
  - Assume our floating point format has **no sign bit, k = 3 exponent bits, and n=2 fraction bits**
  - What does 0b10010 represent? **3**

Floating Point – IEEE Standard

- Bias
  - exp is unsigned; needs a bias to represent negative numbers
  - Bias = 2^{k-1} - 1, where k is the number of exponent bits
  - Can also be thought of as bit pattern 0b011…111
- Normalized | Denormalized

Normalized	Denormalized
	exp = 0
Implied leading 1	
E = exp - Bias	
Denser near origin	
	Represents small numbers

  - When converting frac/int => float, assume normalized until proven otherwise

Floating Point – IEEE Standard

- Bias
  - exp is unsigned; needs a bias to represent negative numbers
  - Bias = 2^{k-1} - 1, where k is the number of exponent bits
  - Can also be thought of as bit pattern 0b011…111
- Normalized | Denormalized

Normalized	Denormalized
0 < exp < (2^k-1)	exp = 0
Implied leading 1	**Leading 0**
E = exp - Bias	**E = 1 - Bias. Why?**
Denser near origin	**Evenly spaced**
Represents large numbers	Represents small numbers

  - When converting frac/int => float, assume normalized until proven otherwise

Floating Point – IEEE Standard

- Special Cases (exp = 2^k-1)
  - Infinity
    - Result of an overflow during calculation or division by 0
    - exp = 2^k-1, frac = 0
  - Not a Number (NaN)
    - Result of illegal operation (sqrt(-1), inf – inf, inf * 0)
    - exp = 2^k-1, frac != 0
  - Keep in mind these special cases are not the same

Floating Point – IEEE Standard

- Round to even
  - Why? Avoid statistical bias of rounding up or down on half.
  - How? Like this:

1.0100_2	truncate	1.01_2
1.0101_2	below half; round down	1.01_2
1.0110_2	interesting case; round to even	1.10_2
1.0111_2	above half; round up	1.10_2
1.1000_2	truncate	1.10_2
1.1001_2	below half; round down	1.10_2
1.1010_2	Interesting case; round to even	1.10_2
1.1011_2	above half; round up	1.11_2
1.1100_2	truncate	1.11_2

Rounding

1.BBGRXXX

Guard bit: LSB of result

Sticky bit: OR of remaining bits

Round bit: 1st bit removed

- Round up conditions
  - Round = 1, Sticky = 1 → > 0.5
  - Guard = 1, Round = 1, Sticky = 0 → Round to even

Value	Fraction	GRS	Incr?	Rounded
128	1.0000000	000	N	1.000
15	1.1010000	100	N	1.101
17	1.0001000	010	N	1.000
19	1.0011000	110	Y	1.010
138	1.0001010	011	Y	1.001
63	1.1111100	111	Y	10.000

Floating Point – Example

- Consider the following 5-bit floating point representation based on the IEEE floating point format. This format does not have a sign bit – it can only represent nonnegative numbers.
  - There are k=3 exponent bits.
  - There are n=2 fraction bits.

4	3	2	1	0
exp			frac	

- What is the…
  - Bias?
  - Largest denormalized number?
  - Smallest normalized number?
  - Largest finite number it can represent?
  - Smallest non-zero value it can represent?

Floating Point – Example

- Consider the following 5-bit floating point representation based on the IEEE floating point format. This format does not have a sign bit – it can only represent nonnegative numbers.
  - There are k=3 exponent bits.
  - There are n=2 fraction bits.

4	3	2	1	0
exp			frac	

- What is the…
  - Bias? **011_2 = 3**
  - Largest denormalized number? **000 11_2 = 0.0011_2 = 3/16**
  - Smallest normalized number? **001 00_2 = 0.0100_2 = 1/4**
  - Largest finite number it can represent? **110 11_2 = 1110.0_2 = 14**
  - Smallest non-zero value it can represent? **000 01_2 = 0.0001_2 = 1/16**

Floating Point – Example

- For the same problem, complete the following table:

Value	Floating Point Bits	Rounded Value
9/32		
8		
9		
	000 10	
19		

Floating Point – Example

- For the same problem, complete the following table:

Value	**Floating Point Bits**	**Rounded Value**
9/32	**`001 00`**	**1/4**
8	**`110 00`**	**8**
9	**`110 00`**	**8**
1/8	`000 10`	
19	**`111 00`**	**inf**

Floating Point Recap

- Floating point = $(-1)^s$ M 2^E
- MSB is sign bit `s`
- Bias = $2^{(k-1)} - 1$ (k is num of `exp` bits)
- Normalized (larger numbers, denser towards 0)
  - `exp` ≠ 000…0 and `exp` ≠ 111…1
  - M = 1.`frac`
  - E = `exp` - Bias
- Denormalized (smaller numbers, evenly spread)
  - `exp` = 000….0
  - M = 0.`frac`
  - E = - Bias + 1

Floating Point Recap

- Special Cases
  - +/- Infinity: `exp` = 111…1 and `frac` = 000…0
  - +/- NaN: `exp` = 111…1 and `frac` ≠ 000…0
  - +0: `s` = 0, `exp` = 000…0 and `frac` = 000…0
  - -0: `s` = 1, `exp` = 000…0 and `frac` = 000…0
- Round towards even when half way (lsb of `frac` = 0)

Questions?