Binary Representation Summary

- Every storage locations stores a finite sequence of bits
  - 8-bit, 16-bit, 32-bit etc.
- The same bit string can mean different things depending on how the program wants to look at it.

Address	7	6	5	4	3	2	1	0
35	1	0	0	0	0	0	0	1
36	1	0	0	0	0	1	1	1
37	1	1	1	0	0	0	0	1
38	0	1	1	0	1	1	0	1

Unsigned: +129

2C: -127

2C: 109

ASCII: 'm'

33

33

Arithmetic and Logical Operations

34

34

Arithmetic Operations…

- Addition: we've seen this
  - Same as decimal…add and propagate carry
- Subtraction: A –B
  - Negate B: compute 2's complement of B
  - Add to A
- Multiplication – we've seen this in decimal…
  - Shift and add
- Shift
  - What happens if we add a number to itself ?
  - (0011) + (0011) = ??
- Shift left once = multiply by 2

35

35

Dose of reality: Finite Width

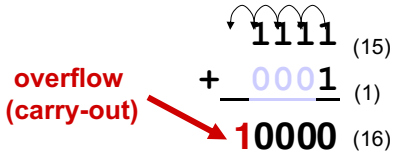
- On a real computer each memory storage location can only store a finite number of bits
  - For example we can talk about a 16 bit machine, a 32 bit machine or a 64 bit machine
  - The fact that the actual storage locations are limited caps the size of the numbers that we can store and manipulate.
- These limitations also show up in programming languages where different basic types have different sizes
  - Some basic types in C
    - char – typically 8 bits
    - short int – typically 16 bits
    - int – typically 32 bits
    - long int – typically 64 bits
  - Note these sizes are not guaranteed and can change on different architectures.

36

36

Dose of reality: Finite Width and Overflow

- Integers have infinite width
  - There are an infinite number of them
- Hardware integers have finite (architecture defined) width
  - Limited by hardware circuits themselves
  - 64- bit these days (2^{64} integers):
  - LC3 integers are 16-bit (2^{16} or ~64,000)
- Overflow: when operation result is outside type's range
  - Example: 15 + 1 with 4-bit integers (16 needs 5 bits, 10000)



37

37

Overflow

- If the numbers are two big, then we cannot represent the sum using the same number of bits.
- For 2's complement, this can only happen if both numbers are positive or both numbers are negative.

```
  01000  (8)        11000  (−8)
+ 01001  (9)      + 10111  (−9)
  10001  (−15)       01111  (+15)
```

- How to test for overflow:
  - Signs of both operands are the same, AND
  - Sign of sum is different.
- *Another test (easier to perform in hardware): Carry-in to most significant bit position is different than carry-out.*

38

38

Arithmetic Overflow - Summary

- For unsigned numbers
  - Any addition that produces an 'extra bit' is a problem
- For 2C signed numbers
  - Sometimes addition or subtraction produce an extra bit – this is not necessarily a problem.
  - Arithmetic overflow can occur when you are adding 2 positive or 2 negative numbers – in this case if the sign of the result is different from the sign of the addends you have an arithmetic overflow
    - (this is the key to determining overflow condition in 2C)
  - Note: most CPU architectures today, use 2C representation

39

39

Shifting Bit Fields

	7	6	5	4	3	2	1	0
Original Pattern x	0	1	1	0	1	0	1	1
X << 1 – Left Shift by 1	1	1	0	1	0	1	1	0
X << 2 – Left Shift by 2	1	0	1	0	1	1	0	0
Original Pattern x	1	1	1	0	1	0	1	1
X >> 1 –Shift Right (logical) by 1	0	1	1	1	0	1	0	1
X >>> 1 – Shift Right (arithmetic) by 1	1	1	1	1	0	1	0	1

- Shift Left:
  - Move all #'s to the left, fill in empty spots with a 0
- Shift Right (2 kinds):
  - shift right logical (SRL) >>
    - shift 0's in from the left
  - shift right arithmetic (SRA) >>>
    - replicate the sign bit, (very useful for sign extension!)

40

40

Shifts

- Powers of 2 are everywhere …
- … and so is multiplication by (small) powers of 2
- Another use of the $2^n = 2*2^{n-1}$ binary identity
  - Shift left by n (pushing in 0s) is the same as multiplying by 2^n
  - Use << to construct both hardware and software multipliers
  - What about shift right ?
- Think of it like multiplying by 10. Say you have 5*10, isn't that just shifting 5 to the ten's place?
  - 5*100, just shifting the 5 to the hundred's place?
- Most important use of "shifting circuits"…
  - To implement multiplication in a computer (recall shift & add?)

41

41

Sign Extension

- Suppose we have a number which is stored in a four bit register
- We wish to add this number to a number stored in a eight bit register
- We have a device which will do the addition and it is designed to add two 8 bit numbers
- What issues do we need to deal with?

42

42

Sign Extension

- To add two numbers, we must represent them with the same number of bits.
- If we just pad with zeroes on the left:

4-bit	**8-bit**
`0100` (4)	`00000100` (still 4)

4-bit	**8-bit**
`1100` (-4)	`00001100` (12, not -4)

43

43

Sign Extension

- To add two numbers, we must represent them with the same number of bits.
- If we just pad with zeroes on the left, won't work for negative integers:

- Instead, replicate the MS bit -- the sign bit:

4-bit	**8-bit**
`0100` (4)	`00000100` (still 4)
`1100` (-4)	`11111100` (still -4)

Question to think about: why does this work?

44

44

Logical Operators

45

45

Another use for bits: Logic

- *logical variables* can be *true* or *false*, *on* or *off*, etc., and so are readily represented by the binary system.
  - A logical variable A can take the values *false* = *0* or *true* = *1* only.
    - Logical Variables = Propositions in propositional logic
    - Example proposition: "Sarah is a UTA for this course" – can only be True or False
- The manipulation of logical variables is known as Boolean Algebra, and has its own set of operations
  - not to be confused with the arithmetic operations
  - Some basic operations: NOT, AND, OR, XOR
  - Boolean function: function over Boolean variables and using the Boolean operators (i.e, logic operations)

46

46

Propositional Logic – sound familiar from CS1311?

- each variable has True (T) or False (F) value
- Use logical connectives to build more complex propositions (i.e., logic statements)
  - Connectives: AND, OR, NOT, …
- (A AND B) is True if A is True and B is true….
- Build "truth table" for propositional 'formula'

A	B	A AND B
F	F	F
F	T	F
T	F	F
T	T	T

A	B	A OR B
F	F	F
F	T	T
T	F	T
T	T	T

A	NOT A
F	T
T	F

47

Boolean Logical Operations

- Represent propositions using binary representation
- Operations on logical TRUE or FALSE variables
  - Boolean variables
  - two states -- takes one bit to represent: TRUE=1, FALSE=0

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

A	NOT A
0	1
1	0

- View *n*-bit number as a collection of *n* logical values
  - operation applied to each bit independently

48

Exclusive OR

- (A XOR B) is true if exactly one of A or B is true; else false

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

49

49

Logic Operations..more examples

"compound" proposition – composition of logic operators

A	B	C	(A AND B)	(NOT C)	(A AND B) OR (NOT C)
0	0	0	0	1	1
0	0	1	0	0	0

50

Bitwise Logical Operations

- View n-bit field as a collection of n logical values
  - Apply operation to each bit independently
- Bitwise AND: useful for clearing bits
  - AND with zero = 0
  - AND with one = no change

```
  11000101
AND 00001111
  ________
  00000101
```

- Bitwise OR: useful for setting bits
  - OR with zero = no change
  - OR with one = 1
- Computers don't support individual bits as a data type
  - Just use least significant of n-bit integer
  - Integers are generally more useful

```
  11000101
OR 00001111
  ________
  11001111
```

51

51

Another dose of "reality"

- look at how some of the concepts we have studied take shape in 'real life'
  - C programming and O/S
- Will loop back to this topic in 2 weeks
  - Go through the lecture notes posted on my webpage
  - As you learn C, try out the operators discussed in the notes

52

52

Logical Operations in C

- C supports both bitwise and boolean logic operations
  - x & y bitwise logic operation
  - x && y boolean operation: output is boolean value
- What's going on here?
  - In boolean operation the result has to be TRUE (1) or FALSE (0)
  - Treats any non-zero argument as TRUE and returns only TRUE (1) or FALSE (0)
- In C: logical operators do not evaluate their second argument if result can be obtained from first
  - a && 5/a can we get divide by zero error?

53

53

Bitwise Operators in C

- Can only be applied to integral operands
- *that is,* `char`, `short`, `int` *and* `long`
- `(signed` *or* `unsigned)`

```
&       Bitwise AND
|       Bitwise OR
^       Bitwise XOR
<<      Shift Left
>>      Shift Right
~       1's Complement (Inversion)
```

54

54

Question: Bitwise operations in C

- assume 4 bit
- What is (4 & 6) : 0100 & 0110 ?
- What is (4 ^ 6): 0100 ^ 0110 ?
- What is (~4)
- What is (4 && 6): 0100 && 0110 ?

55

55

Text, Real numbers,….

56

56

Hexadecimal (Base-16) Notation

- More compact and convenient than binary (base-2)
  - Fewer digits: group four bits per hex digit → less error prone
  - Just a notation, not a different machine representation
    - Most languages (including C and LC-3) parse hex constants
  - Sometimes hex numbers preceded with x or 0x

Binary	Hex	Decimal	Binary	Hex	Decimal
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	A	10
0011	3	3	1011	B	11
0100	4	4	1100	C	12
0101	5	5	1101	D	13
0110	6	6	1110	E	14
0111	7	7	1111	F	15

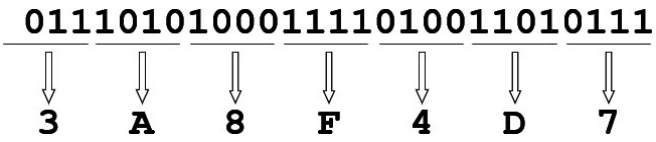
16 symbols: 0,1,2,…A,B,C,D,E,F

57

57

Converting from binary to hex

- Starting from the right, group every four bits together into a hex digit. Sign-extend as needed.



This is not a new machine representation or data type, just a convenient way to write the number.

Ex: Hex number 3D6E easier to communicate than binary 0011110101101110.
Ex: 0x1 = representation of decimal number 1
Ex: 0x14 = decimal number 16+4 = 20
Ex: 0xFFFF = 16 bit number with all 1's = decimal number 2^{16}-1

58

58

Using Binary #'s to represent any type of information

- "**Encoding**" data, simply means an agreed upon "mapping" of data from one representation to another
  - At some point, it is the choice of an engineer to define the encoding or "mapping" of data between two forms
- To represent text we use ASCII encoding
- ASCII: American Standard Code for Information Interchange
  - 7 bits needed to encode all characters
  - Represent as 8 bit number (i.e., a byte)

59

59

ASCII Codes

- Represent characters from keyboard
  - This encoding used to transfer characters between computer and all peripherals (keyboard, disk, network…)
- Typing a key on keyboard = corresponding 8-bit ASCII code is stored and sent to computer
  - The computer has to interpret the ASCII code and 'extract' the character represented by the code
    - Most programming languages have this feature built-in (ie., compiler figures it out for you)

7 bit binary	Hex	character	7 bit binary	Hex	character
011 0000	30	0	100 0101	45	E
011 0001	31	1	110 0101	65	e
010 0001	21	!	010 0000	*20*	*space*
010 0011	23	#	000 1010	*0A*	*linefeed*

60

60

Table: ASCII Codes

- ASCII: Maps 128 characters to 7-bit code.
  - both printable and non-printable (ESC, DEL, …) characters

00	nul	10	dle	20	sp	30	0	40	@	50	P	60	`	70	p
01	soh	11	dc1	21	!	31	1	41	A	51	Q	61	a	71	q
02	stx	12	dc2	22	"	32	2	42	B	52	R	62	b	72	r
03	etx	13	dc3	23	#	33	3	43	C	53	S	63	c	73	s
04	eot	14	dc4	24	$	34	4	44	D	54	T	64	d	74	t
05	enq	15	nak	25	%	35	5	45	E	55	U	65	e	75	u
06	ack	16	syn	26	&	36	6	46	F	56	V	66	f	76	v
07	bel	17	etb	27	'	37	7	47	G	57	W	67	g	77	w
08	bs	18	can	28	(	38	8	48	H	58	X	68	h	78	x
09	ht	19	em	29	)	39	9	49	I	59	Y	69	i	79	y
0a	nl	1a	sub	2a	*	3a	:	4a	J	5a	Z	6a	j	7a	z
0b	vt	1b	esc	2b	+	3b	;	4b	K	5b	[	6b	k	7b	{
0c	np	1c	fs	2c	,	3c	<	4c	L	5c	\	6c	l	7c	|
0d	cr	1d	gs	2d	-	3d	=	4d	M	5d	]	6d	m	7d	}
0e	so	1e	rs	2e	.	3e	>	4e	N	5e	^	6e	n	7e	~
0f	si	1f	us	2f	/	3f	?	4f	O	5f	_	6f	o	7f	del

how to handle more than 128 characters?...
Unicode representation

61

61

Interesting Properties of ASCII Code

- What is relationship between a decimal digit ('0', '1', …) and its ASCII code?

- What is the difference between an upper-case letter ('A', 'B', …) and its lower-case equivalent ('a', 'b', …)?

- Given two ASCII codes for characters, how do we tell which comes first in alphabetical order?

62

62

Limitations of integer representations ?.. do we need anything else?

- Most numbers are not integer!
  - Even with integers, there are other considerations
- Range:
  - The magnitude of the numbers we can represent is determined by how many bits we use:
    - e.g. with 32 bits the largest number we can represent is about +/- 2 billion, far too small for many purposes.
- Precision:
  - The exactness with which we can specify a number:
    - e.g. a 32 bit number gives us 31 bits of precision, or roughly 9 figure precision in decimal repesentation.
- How to deal with Real numbers…We need other data types!

63

63

How to deal with complicated real numbers….Some History…

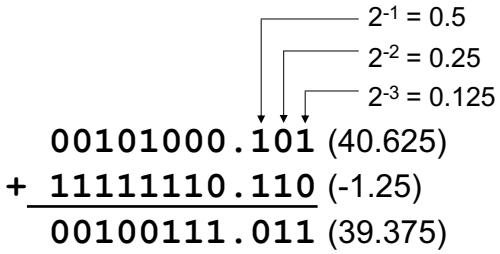
- The Indiana Legislature once introduced legislation declaring that the value of π was exactly 3.2

64

64

Fixed-Point

- How can we represent fractions?
  - "binary point" separate positive from negative powers of two
    - Analogous to "decimal point": .75 = (7/10)+(5/100)
  - 2C addition and subtraction still work
    - If binary points are aligned ("fixed-point")



65

65

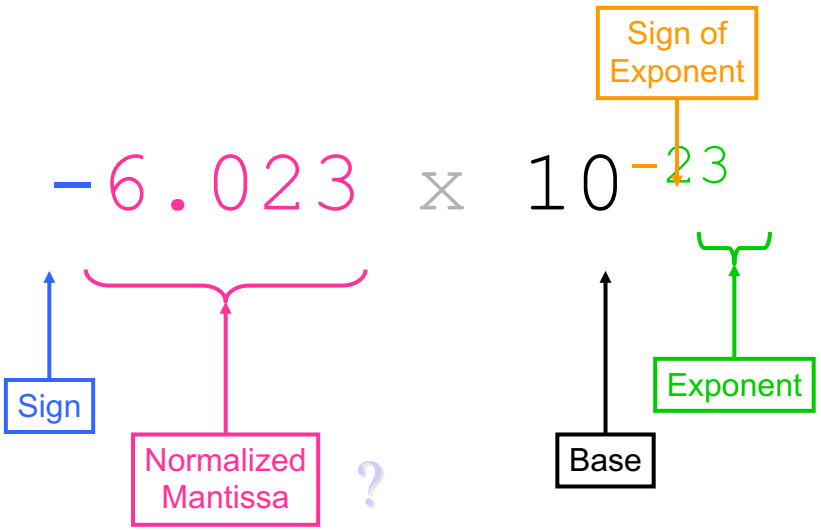
Very Large and Very Small: Floating-Point

- Problem
  - Large values: $6.022 \times 10^{23} \rightarrow$ requires 79 bits
  - Small values: $6.626 \times 10^{-34} \rightarrow$ requires >110 bits
- Solution: use equivalent of "scientific notation": $F \times 2^E$
  - Need to represent F (fraction), E (exponent), and S (sign)
- IEEE 754 Floating-Point Standard

66

66

Scientific Notation



67

67

What next..

- The hardware building blocks and their operations – Chapter 3
- Digital Logic structures
  - Basic device operations: CMOS transistor
  - Combinational Logic circuits
    - Gates (NAND, OR, NOT), Decoder, Multiplexer
    - Adders, multipliers
  - Sequential circuits– concept of memory
    - Finite state machines, memory organization
    - Basic storage elements: latches, flip-flops

68

68

Appendix

- Additional notes not covered during lecture

69

69

Generalized Weighted Positional base k (radix k) representation

- Can generalize weighted positional to any base k
- Use k symbols – also known as k-ary numbers (radix k)
  - Radix-10 (decimal) 0,1,2,…,9
  - Radix 2 (binary) 0,1
  - Radix 16 (hex) 0,1,…,9,A,B,C,D,E,F
- Weighted positional numbers – position gives "weight" of location
  - Position 0 (rightmost) has weight 1 (k^0), Position i has weight k^i
- The base k number $a_{n-1}\ldots a_1 a_0$ represents decimal value

  $\sum_{i=0}^{i=n-1} a_i \, k^i$

- How many different base k numbers of length n ?
  - Each of the n positions can have k values
  - How many different strings of length n, where each position has one of k values

70

70

Signed Magnitude

- 5-bit number
- Leading bit is the sign bit

$Y = \text{“abc”} = (-1)^a (b.2^1 + c.2^0)$

Range is:
$-2^{N-1} + 1 < i < 2^{N-1} - 1$

-4	10100
-3	10011
-2	10010
-1	10001
-0	10000
+0	00000
+1	00001
+2	00010
+3	00011
+4	00100

71

71

One’s Complement

- Invert all bits

If msb (most significant bit) is 1 then the number is negative (same as signed magnitude)

Range is:
$-2^{N-1} + 1 < i < 2^{N-1} - 1$

-4	11011
-3	11100
-2	11101
-1	11110
-0	11111
+0	00000
+1	00001
+2	00010
+3	00011
+4	00100

72

72

Two's Complement (2C) – why does it work

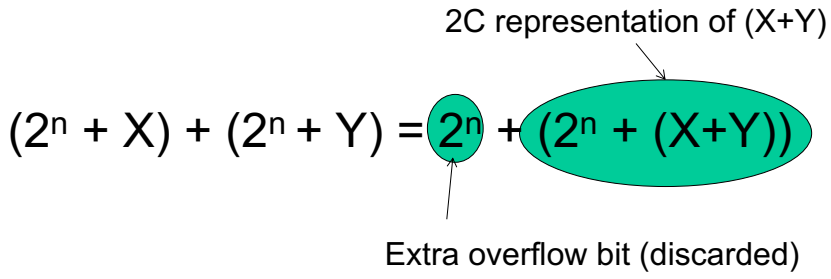
- Representation designed to allow us to store and manipulate both positive (aka: +ve) and negative (aka: –ve) numbers
- To represent a number X we actually compute and store (2^n + X)
- Recall 2^n in binary will be a 1 followed by n zeros

73

73

Why does this work?

- Consider adding two 2C numbers:



In practice:

- The Most Significant Bit (MSB) in N-bit 2C representation has a weight of $-2^{(N-1)}$

74

74

Encoding Integers: Formal Definition

Unsigned

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

Two's Complement

$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

Sign Bit

```
short int x =  15213;
short int y = -15213;
```

- C `short` 2 bytes long

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
y	-15213	C4 93	11000100 10010011

- Sign Bit
  - For 2's complement, most significant bit indicates sign
    - 0 for nonnegative
    - 1 for negative

75

Comparison

- Another useful operation is comparison
  - == (equals), != (not equals), >, <, >=, <=
- Comparison via subtraction, A – B, if result is …
  - Zero → A == B, not zero → A != B
  - Positive → A > B, not positive → A <= B
  - Negative → A < B, not negative → A >= B
- Pitfall: comparison is explicitly signed or unsigned
  - +/– are not, "result" is same either way
  - Comparison interprets numbers in a way +/– don't
  - Example, which is bigger 0110 or 1010?

76

Implementing Comparison

- How are signed and unsigned comparison implemented?
  - Let's look at 0110 (6) and 1010 (–6 or 10) in 4-bit representation
  - If this is a signed comparison, subtraction result is positive (12)
  - If unsigned, subtraction result is negative (–4)
  - Potential problem: 12 overflows 4-bit signed representation
  - What to do? Extend to 5-bit representation, check "new" MSB
    - Signed comparison? Sign extend
    - Unsigned comparison? Zero extend

```
 00110        00110
–11010       –01010
 01100        11100
```

77

Basic Logic Operations

- Equivalent Notations
  - not A = A' = $\overline{A}$
  - A and B = A.B = A$\wedge$B = A intersection B
  - A or B = A+B = A$\vee$B = A union B
- Other common logic operations:
  - NAND = NOT AND
    - Find AND and then Complement it (invert bit)
  - NOR = NOT OR
    - Find OR and then Complement it
  - XNOT = NOT XOR

78

DeMorgan's Laws 1

- There's an interesting relationship between AND and OR.
- If we NOT two values (A and B), AND them, and then NOT the result, we get the same result as an OR operation. (In the figure below, an overbar denotes the NOT operation.)

- **Here's the truth table to convince you:**

A	B	$\overline{A}$	$\overline{B}$	$\overline{A}$ AND $\overline{B}$	$\overline{\overline{A}\ \text{AND}\ \overline{B}}$
0	0	1	1	1	0
0	1	1	0	0	1
1	0	0	1	0	1
1	1	0	0	0	1

79

79

DeMorgan's Laws 2

- This means that any OR operation can be written as a combination of AND and NOT.

```
    11000101
OR  00001111
    --------
    11001111
```

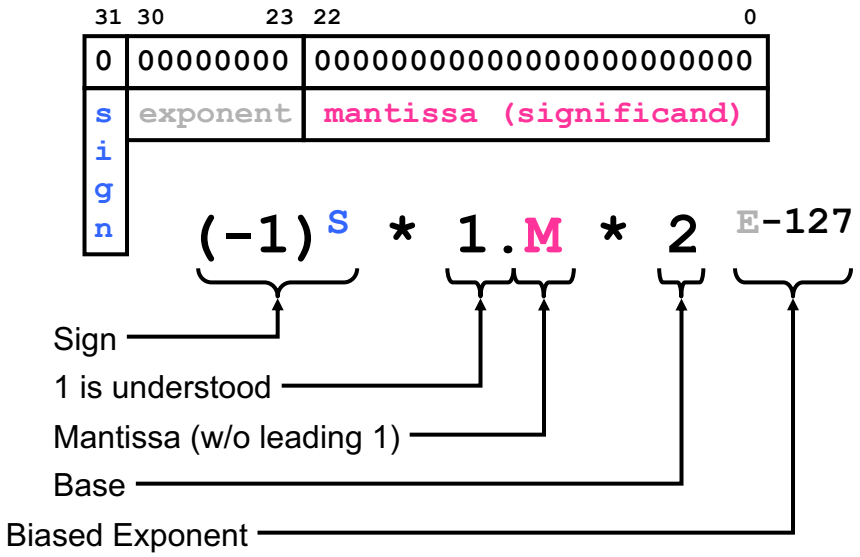
```
    00111010
AND 11110000
    --------
    00110000

NOT 00110000
    --------
    11001111
```

- This is interesting, but is it useful? We'll come back to this in later chapters...
- Also, convince yourself that a similar "trick" works to perform AND using only OR and NOT.

80

80

IEEE-754



81

81

IEEE 754 Floating-Point Standard

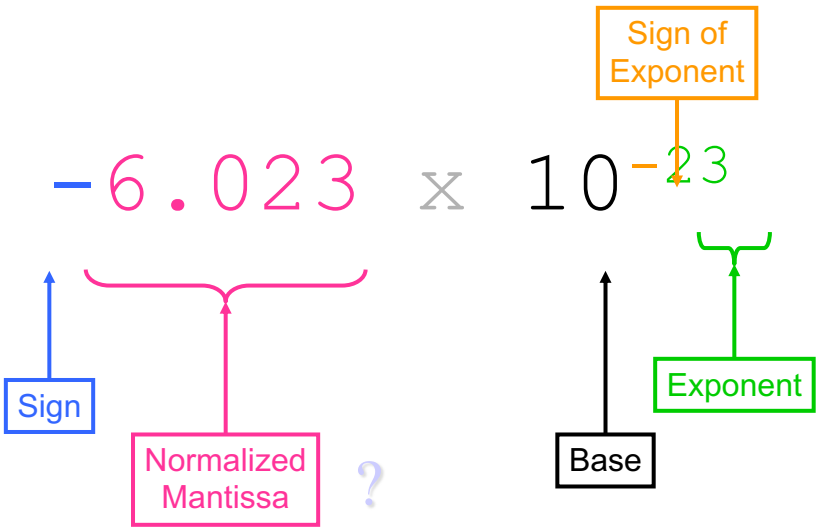
S	Exponent	Fraction

- 32-bit ("single-precision" or float)
  - 8-bit exponent, 23-bit fraction
  - X = -1^s * 1.fraction * $2^{exponent-127}$, 1 ≤ exponent ≤ 254
    - Exponent representation is called "excess notation"
- 64-bit ("double-precision" or double)
  - 11-bit exponent, 52-bit fraction
  - X = -1^s * 1.fraction * $2^{exponent-1023}$, 1 ≤ exponent ≤ 2046
- Representation must be "normalized" (just like decimal)
  - 1 ≤ Fraction < 2 (fraction to left of binary point must be 1)
    - This 1 is implicit in Fraction

82

82

Scientific Notation



83

83

Floating-Point Example

- What is this?
- 10111111010000000000000000000000

sign *exponent* *fraction*

  - Sign is 1: number is negative
  - Exponent is 01111110 = 126 (decimal)
  - Fraction is 0.100000000000… = $1/10_2$ = 0.5 (decimal)
- Value = $-1.5 * 2^{(126-127)} = -1.5 * 2^{-1} = -0.75$

84

84

More floating point

- Reading assignment – Chapter 2

85

85