Data Representaton:
Bits, Data Types, Operations
(Chapter 2)

1

How do you represent data ?

- Our first requirement is to find a way to represent information (data) in a form that is mutually comprehensible by human and machine.
  - What kinds of data ?
    - Integers
    - Reals
    - Text
    - …what else
    - …

2

2

Data Type

- In a computer system, we need a representation of data and operations that can be performed on the data by the machine instructions or the computer language.
- This combination of *representation + operations* is known as a data type.
  - The type tells the compiler how the programmer intends to use it
- Prog. Languages have a set of data types defined in lang
  - In C: int, float, char, unsigned int, …

Type	Representation	Operations
Unsigned integers	binary	add, multiply, etc.
Signed integers	2's complement binary	add, multiply, etc.
Real numbers	IEEE floating-point	add, multiply, etc.
Text characters	ASCII	input, output, compare

3

3

Number systems

- A number is a mathematical concept
  - Natural numbers, Integers, Reals, Rationals,..
- Many ways to represent a number…..
  - Symbols used to create a representation
  - Example: Decimal representation uses the symbols (digits) 0,1,2…9
    - Binary uses the symbols 0,1
  - Roman numerals: I, II, V, X, etc.

4

4

Example: Decimal number system

- What are you used to ? Decimal representation
- Symbols/digits = 0,1,2,…9 (ten of them, hence "decimal")
- How is a number encoded
  - how to represent three hundred twenty nine ?
- Decimal **Weighted positional representation**
  - *Position gives the weight of the location*
  - decimal number "329" (three hundred twenty nine)
  - "3" is worth 300, because of its position (most significant)
  - "9" is only worth 9 (least significant)

base-10 (decimal)

329

10^2 10^1 10^0

3x100 + 2x10 + 9x1 = 329

Value= Three hundreds,
Two tens, and
Nine ones.

5

5

Your first counting numbers experience ? How did you learn to count? How did you express a number ?

The Unary system is also used by Turing Machines …Why ?

6

6

In the CS world…..

- **There are 10 kinds of people in the world…**

Those who know binary, and those who don't

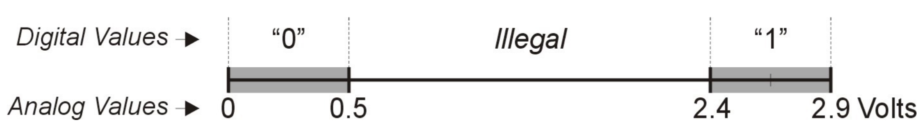
?

7

7

Computer is a Binary Digital System

- Digital = finite number of values (compared to 'analog'= infinite values)
- Binary = only two values: 0 and 1
  - Unit of information = binary digit or "bit"



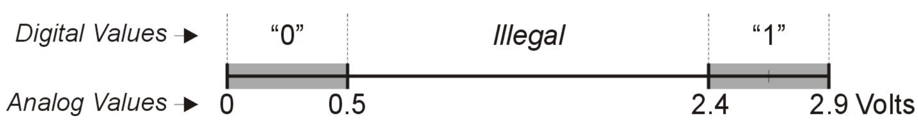
- Why binary: need only check two voltage values
  - Voltage exists or Voltage is zero
    - Circuits (Chap 3) will pull voltage down to 0 or pull up to highest voltage
  - Grey areas represent noise margin – allowable deviation due to electrical properties (resistance, capacitance, interference,..)
  - More reliable than analog
- Alternative: can define multiple discrete values in voltage range
  - Problem: circuits would become much more complex

8

8

Why Binary …

- Computers are electrical devices….
- Why binary: need only check two voltage values
  - Voltage exists or Voltage is zero
    - Circuits (Chap 3) will pull voltage down to 0 or pull up to highest voltage
  - Grey areas represent noise margin – allowable deviation due to electrical properties (resistance, capacitance, interference,..)
  - More reliable than analog
- Alternative: can define multiple discrete values in voltage range
  - Problem: circuits would become much more complex



9

9

If we have more than two values to represent…

- Basic unit of information = binary digit or *bit*
- Each "wire" in a logic circuit represents one bit = 0 or 1
- Values with more than 2 states require multiple wires (bits)
- With 2 bits → 4 possible values (states/strings): 00, 01, 10, 11
- 3 bits → 8 values: 000, 001, 010, 011, 100, 101, 110, 111

- **In general: with n bits can represent 2^n different values**
- Question: what is the minimum number of bits needed to represent 21 different values?
- at least $\log_2 21$ = 5 bits

10

10

Bits – the universal data representation

- everything that is stored or manipulated on the computer is ultimately expressed as a group of bits.
  - Text – characters, strings,
  - Numbers – integer, fraction, real,…
  - Video, Audio, Images (using pixels…pixel can be 8 bits)
  - Logical – True (1) or False (0)
  - Instructions (program) are just 0's and 1's = programs are just another kind of data!

11

11

Data Representation: Encoding

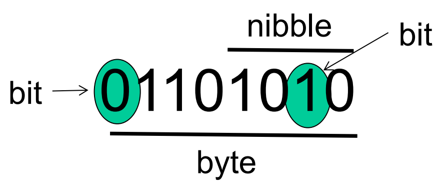
- We next encode a value by assigning a bit pattern to represent that value
  - Encoding determines *how to interpret* the value of an n-bit binary 'string'
  - *Weighted positional encoding is one type of encoding*
- we perform operations (transformations) on bits, and we interpret the results according to how the data is encoded
- How to represent different types of data:
  - **Start with Integers**
    - Unsigned (non-negative)
    - Negative
  - Text …ASCII codes
  - Real numbers – floating point

13

13

Terminology

- A single binary digit is referred to as a bit
- A collection of 8 bits is referred to as a byte
- A collection of 4 bits is referred to as a nibble
  - Also a *Hex digit*
- In a computer memory each storage location can only hold a finite number of bits



14

14

(Unsigned) Integer Representation

- Non-positional notation (unary): 5 represented as 11111
- What are you used to ? Decimal representation (0..9) and…
- Decimal **Weighted positional representation**
  - *Position gives the weight of the location*
- Extend to any base, including binary…..
  - Weights in decimal are 10^0, 10^1 , 10^2 , 10^3, …
  - Weights in binary are 2^0, 2^1, 2^2, 2^3,….

15

15

Integer Representation

- Weighted positional representation in Binary

base-10 (decimal)

329

10^2 10^1 10^0

most significant

101

least significant

2^2 2^1 2^0

base-2 (binary)

3x100 + 2x10 + 9x1 = 329

1x4 + 0x2 + 1x1 = 5

Notations: the bit position i has weight of 2^i

n bit binary number $a_{n-1}a_{n-2},\ldots,a_1,a_0$

represents the decimal value/number

$$\sum_{i=0}^{i=n-1} a_i\, 2^i$$

16

16

Unsigned Integers

- An n-bit unsigned integer represents 2^n values
  - Values from 0 to 2^n-1
- 3-bit represents 2^3=8 values
- 4-bit represents 2^4
- Max integer value 2^n-1

2^2	2^1	2^0	val
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	4
1	0	1	5
1	1	0	6
1	1	1	7

17

17

Questions

-
- what decimal number does the binary string 1011 represent
- What decimal number does 00011 represent ?

18

18

Decimal to Binary Conversion:

1. What is the binary representation of decimal number 19
   - Express 19 as a sum of numbers each a power of 2
   - Algorithm to convert decimal (base 10) to binary (base 2)
     - Generalize to convert from base k to base m

k bit number: $b_{k-1}, b_{k-2}, \ldots, b_1, b_0$
Decimal integer N represented by this binary number is:
$b_{k-1} 2^{k-1} + b_{k-2} 2^{k-2} + \ldots + b_1 2^1 + b_0 2^0$

$19 = 1.16 + 0.8 + 0.4 + 1.2 + 1.1$
$= \mathbf{1}.2^4 + \mathbf{0}.2^3 + \mathbf{0}.2^2 + \mathbf{1}.2^1 + \mathbf{1}.2^0$
10011

19

19

Conversion from Decimal to Binary

//input is Decimal number N, output is list of bits b_i //
i=0;
while N > 0 do
  b_i = N % 2; // b_i = remainder; N mod 2
  N = N / 2; // N becomes quotient of division
  i++;
end while /* replace 2 by k and your algo can convert to any base k */

- Iteration i=0: b_0 = 19%2 =1 and N= 19/2= 9
- Iteration 1: b_1 = 9%2 = 1 and N=4
- Iteration 2: b_2 = 4%2 = 0 and N=2
- Iteration 3: b_3 = 2%2 =0 and N=1
- Iteration 4: b_4 = 1%2 =1 and N=0 so loop terminates
- Binary representation of 19 = 10011

20

20

Recap: Binary representation of integers

- We saw how Natural numbers can be represented in binary using weighted positional system
- In general, base-K (radix-K) representation of numbers using weighted positional system
  - Decimal is base 10
  - Binary is base 2
  - Hex is base 16
- Next..how about negative integers ?
  - Text ?
  - Real numbers?
- Operations..arithmetic, logical

21

21

Next: Negative Integers, Text, Operations (Arithmetic and Logical), Real Numbers

- Representation of negative numbers
- Representing text…ASCII
- Arithmetic operations

- Read notes for Logical operations – this is review from your Discrete math class CSCI 1311 (truth tables for propositional logic operators – AND, OR, …)
  - Later next week: Review discussion of Karnaugh maps

22

22

Arithmetic Operations on Unsigned Integers

- Recall: Data type is representation and operations

23

23

Unsigned Binary Arithmetic

- Base 2 addition – just like base 10
- Add from right to left, propagate carry
  - 0+0 =0, 1+0 =1 , 1+1 =0 and carry =1, 1+1+1 = 1 and carry =1

```
  10010            carry            
+ 01001          10010         1111
  11011        + 01011       +    1
                 11101        10000

                 00111
               + 00111
                 11110
```

24

24

Questions for (table) groups:

- Write down answers on a page of paper…you will submit at end of class…Write your names on the page
- 1. What is the 6-bit unsigned binary representation for the decimal number 23 ?
- 2. What is the result of adding two 4 bit numbers: 0100 and 1100
  - What is the 4-bit result ?

25

25

What About Negative Integers?

- Negative numbers have rights too
  - No negation without representation!!
- How do we represent negative integers in decimal:
  - sign followed by value
  - - 269
  - +169 is usually written as 169 (drop the + sign)
- Question: Is this a valid (as per math definition) base 10 (decimal) representation ?

26

26

Negative Integers in Binary?

- One option: sign-magnitude concept
  - What do we do with paper-and-pencil: put a '-' in front
  - No '–' in binary, just use a 1 in most significant bit to denote sign (0= positive, 1= negative)

    00101 = 5

    10101 = –5
- Another option: 1's Complement
  - Simply complement bits
  - 00101 = 5
  - 11010 = -5
- Note: in both these representations, we are using an extra bit to denote the sign – most significant bit=1 if -ve

27

27

Examples

- 4 bit representation of -2 in
  - Signed magnitude binary
    - First represent 2 in binary: 0010
    - Since negative, the most significant bit (leftmost) should be=1
    - Therefore -2 in signed magnitude binary is: 1010
  - 1's complement binary – first represent 2 in binary= 0010
    - Complement all the bits to get 1101

28

28

Examples: Question – Will addition "algorithm" work?

- A and B are signed magnitude binary nos.
  - A=1010 (-2) and B= 0011 (+3)
  - What is A+B = ? must interpret result as signed magnitude rep.

```
  1010
+ 0011
  ????
```

- A and B are 1's complement binary nos.
  - A= 0101 (5) and B=1100 (-3)
  - A+B= ? Must interpret result as 1's complement representation

```
  0101
+ 1100
  ????
```

29

29

Examples: Question – will addition algorithm work?

- NO! Problems with both representations…..
- A and B are signed magnitude binary nos.
  - A=1010 (-2) and B= 0011 (+3)
  - What is A+B = -5

```
  1010
+ 0011
  1101 (-5)
```

- A and B are 1's complement binary nos.
  - A= 0101 (5) and B=1100 (-3)
  - A+B= 2

```
  0101
+ 1100
  0001 (1)
```


What type of representation do we want ?

- We would like the same arithmetic 'algorithms' work for negative numbers
  - Keeps hardware circuits simple
- We want the same addition algorithm
  - Add starting with rightmost (least significant) bit and propagate the carry bit to the left
- Oops…Problem with signed magnitude and 1's Comp
  - Same addition algorithm does not work!!
- Furthermore two representations for zero
  - In signed magnitude both 1000 and 0000 represent 0
  - In 1's Complement both 1111 and 0000 represent 0
- *Using Signed magnitude or 1C to represent negative integers is a bad idea!*

Two's Complement Representation (2C)

- This is the standard representation for (signed) integers
- viewed as weighted position: but weight of most significant bit is (-2^{N-1})
- If number is positive or zero,
  - normal binary representation*, zero in most significant bit*
- If number is negative,
  - start with positive number
  - flip every bit (i.e., take the one's complement)
  - then add one

```
  00101  (5)
  11010  (1's comp)
+     1
  11011  (-5)
```

What's really happening:
negative number x is represented as $2^n - x$

32

More 2C examples

- Find/compute 2C representation of -9
- Find/compute 2C representation of -(-6) (= 6)

```
  01001  (9)
  10110  (1's comp)
+     1
  10111  (-9)
```

```
  11010  (-6)
  00101  (1's comp)
+     1
  00110  (6)
```

33

Addition

- Two 2's Complement numbers
- A = 1010 what is its decimal equivalent:
  - = negative, therefore flip bits and add 1 to get 0101+1=0110
  - A = -6
- B = 0011 what is its decimal equivalent:
  - = positive, therefore B =3
- What is A+B

```
   1010 ( -6)
   0011 ( 3)
 +______
   1101 ( -3)
```

34

34

2C Summary

- If you have the binary representation for a number, to find the negative in 2C representation, simply:
  - Flip all the bits and add 1
    - OR ....a shortcut:
  - Copy bits from right to left up to and including the first '1'
    - Flip remaining bits
  - Techniques work in reverse as well!
- To find decimal value of a 2's complement representation
  - If MSB=0 then weighted position representation
  - If MSB=1 then number is negative, and to find its magnitude Flip all bits and add 1 (note: this turns it into a positive number so we can get the magnitude/value).

35

35

Bits – the universal data representation

- It is important to realize that everything that is stored or manipulated on the computer is ultimately expressed as a group of numbers and, hence, as a sequence of bits.
  - **Text** – individual samples represented as binary numbers/codes
  - Audio – Sounds represented as a sequence of audio samples
  - Pictures – Represented as arrays of intensity values, intensity values are stored as numbers
    - Monochrome images – 8 bits per pixel
    - Color images – 3 channels Red, Green and Blue 8 bits per channel

36

36

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

37

37

Converting from binary to hex

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

```
 011 1010 1000 1111 0100 1101 0111
  ↓    ↓    ↓    ↓    ↓    ↓    ↓
  3    A    8    F    4    D    7
```

This is not a new machine representation, 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

38

38

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)

39

39

ASCII Codes

- Represent characters from keyboard
  - This encoding used to transfer characters between the 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*

40

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

41

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….based on data representation used

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’

If string at address 38 is defined as type `int` in C then value = 109
If defined as `char` then value=m

42

42

Exercises…at the tables….write on the page and submit before leaving class.

3. What is the 6-bit 2’s complement representation of 13 ?
4. What is the 6-bit 2’s complement representation of -13 ?
5. What is the decimal equivalent of the 6 bit 2’s complement number 111110 ?

43

43

Arithmetic and Logical Operations

44

44

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 – use same algo we use for decimal?
  - Shift and add
- Shift
  - What happens if we add a number to itself ?
  - (0011) + (0011) = ??
- Shift left once = multiply by 2

45

45

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!)

46

46

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?
- important use of "shifting circuits"…
  - To implement multiplication in a computer (recall shift & add?)

47

47

Multiplication

```
 235
* 24
--------
  940
 470-
---------
5640
```

- 235 * 24
- = $235*4*10^0 + 235*2*10^1$
- 235*4 = 235 + 235 + 235 + 235 = 940
  - i.e., repeated addition
  - = 940
- 235*2 = 235 + 235
  - = 470
- $235*2*10^1$ = 4700 i.e, shift left once (one digit position)
- 235*24 = (235*4) shift zero times + (235*2) shifted once left
- Can use same algorithm "shift+add" for binary nos…
  - Repeated addition not needed since mult by 0 or mult by 1 !!

48

48

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.

49

49

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)

```
                      1111   (15)
overflow           +  0001   (1)
(carry-out)          10000   (16)
```

Problem: using 4-bit representation the sum is 0!!

50

50

Overflow

- If the numbers are too large, 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.*

51

51

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

52

52

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?

53

53

Sign Extension

- To add two numbers, we must represent them with the same number of bits
  - Because "Adder hardware" takes two inputs of same length (type)
    - SW Analogy: calling a function with the correct arguments
- 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) ❌

54

54

Sign Extension

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

55

55

Logical Operators

Review your Discrete Math course!

56

56

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: “Lauren is an LA 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 and laws
  - not to be confused with the arithmetic operations
  - Some basic operations: NOT, AND, OR, XOR
- What is a Boolean function = function over Boolean variables and using the Boolean operators (i.e, logic operations)

57

57

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

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

58

58

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

59

59

Logic Operations..more examples

"compound" proposition = composition of logic operators
(A AND B) OR (NOT C)

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
0	1	0	?	?	?
…					
1	1	1	?	?	?

3 'binary' variables A, B, C: therefore 8 rows

60

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
- 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 bit of n-bit integer
  - Integers are generally more useful

```
    11000101
AND 00001111
    00000101

    11000101
OR  00001111
    11001111
```

61

61

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

62

62

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?
- Logical operators (output is 0 or 1) in C:
  - && logical AND (both must be non-zero)
  - || Logical OR (at least one must be non-zero)
  - ! Logical NOT (!x=1 if x=0 else !x=0)

63

63

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)

64

64

Exercises…at the tables….write on the page and submit before leaving class.

6. Bitwise operations - assume 4 bit
   a) What is (4 & 6) in binary: 0100 & 0110 ? */* & is Bitwise AND */*
   b) What is (4 ^ 6): 0100 ^ 0110 ? */* ^ is Bitwise XOR */*
   c) What is (~4)+1 ? What is its decimal value ? */* ~ = complement */*
   d) What is (4 && 6): 0100 && 0110 ? */* && is logical AND */*
7. Bitwise…assume 32 bit signed *integers…an exercise in abstraction!*
   a) What is (737 & 1) :?
   b) What is (546 & 1): ?

65

65

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!

66

66

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

67

67

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")

$2^{-1} = 0.5$
$2^{-2} = 0.25$
$2^{-3} = 0.125$

```
  00101000.101 (40.625)
+ 11111110.110 (-1.25)
  00100111.011 (39.375)
```

68

68

Very Large and Very Small: Floating-Point

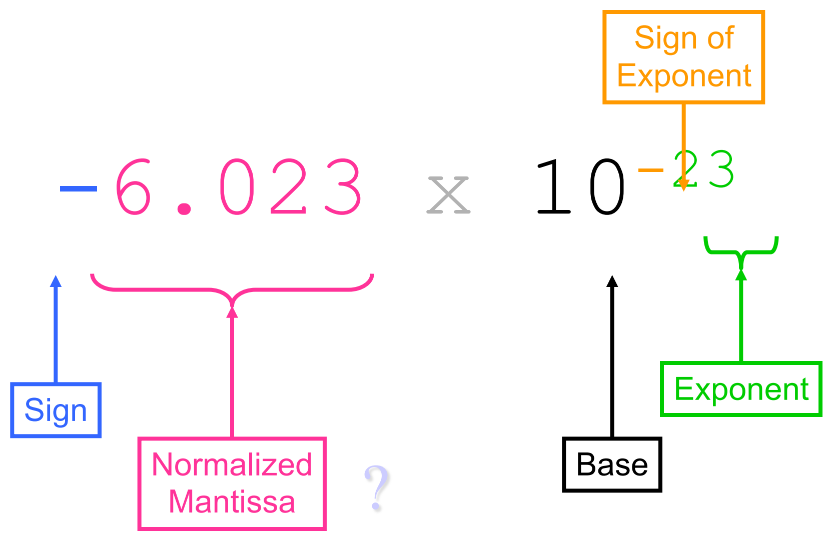
- Problem
  - Large values: $6.022 \times 10^{23} \rightarrow$ requires 79 bits
  - Small values: $6.626 \times 10^{-34} \rightarrow$ requires >110 bits
- Solution: use equivalent of "scientific notation": $F \times 2^E$
  - Need to represent F (fraction), E (exponent), and S (sign)
- IEEE 754 Floating-Point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

S	Exponent	Fraction

69

69

Scientific Notation



70

70

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

71

71

Appendix

- Additional notes not covered during lecture

72

72

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

73

73

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

74

74

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

75

75

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

76

76

Why does this work?

- Consider adding two 2C numbers:

2C representation of (X+Y)

$$(2^n + X) + (2^n + Y) = 2^n + (2^n + (X+Y))$$

Extra overflow bit (discarded)

In practice:

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

77

77

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

78

78

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?

79

79

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

80

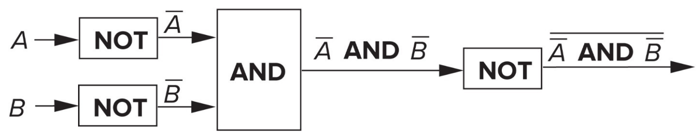
Basic Logic Operations

- Equivalent Notations
  - not A = A' = $\overline{A}$
  - A and B = A.B = A∧B = A intersection B
  - A or B = A+B = A∨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

81

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

82

82

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
```

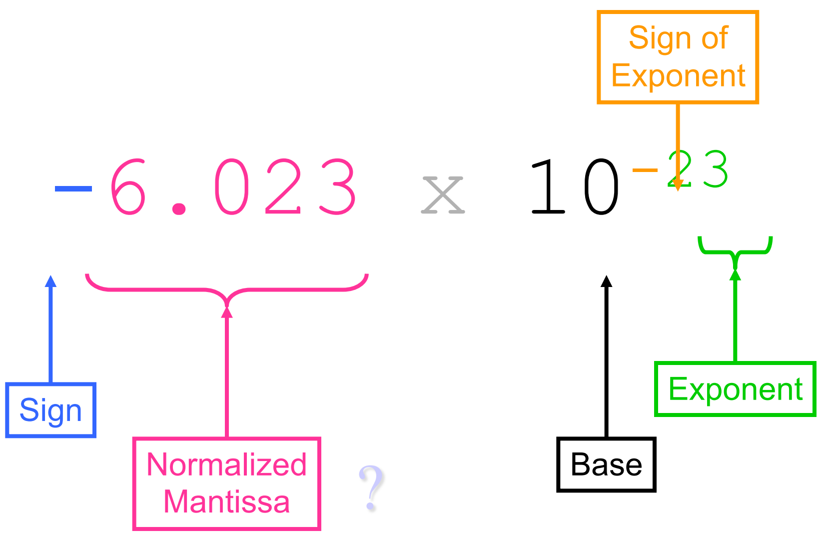
- This is interesting, but is it useful? We'll come back to this in later chapters...
- Also, convince yourself that a similar "trick" works to perform AND using only OR and NOT.

✧Access the text alternative for slide images.

83

83

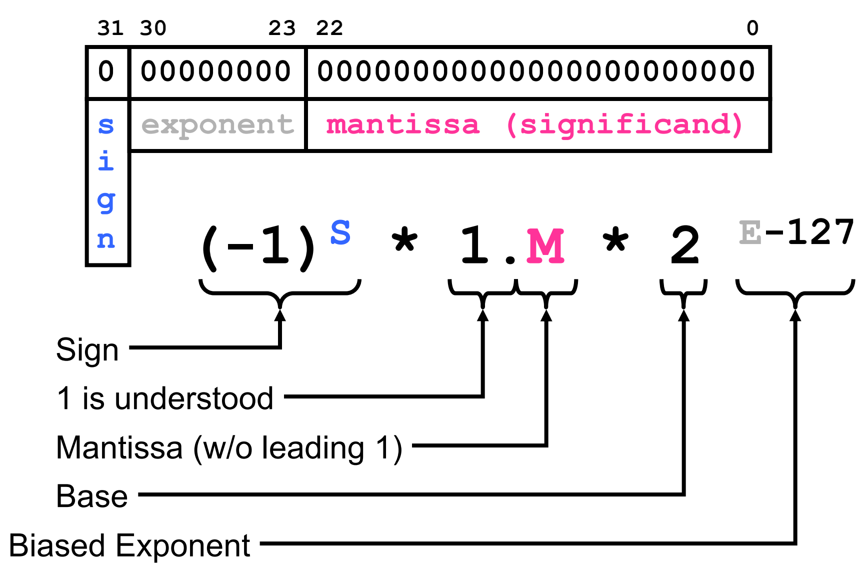
Scientific Notation



84

84

IEEE-754



85

85

IEEE 754 Floating-Point Standard

S	Exponent	Fraction

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

86

86

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 = $-1.5 * 2^{(126-127)} = -1.5 * 2^{-1} = -0.75$

87

87