


CS 135: Computer Architecture I


Instructor: Prof. Bhagi Narahari
Dept. of Computer Science
Course URL: www.seas.gwu.edu/~bhagiweb/cs135/



Number Representation

- Chapter 2 of Text (P&P): Today, & next class.
- Datatypes of the machine
 - Integer Datatypes - Unsigned & Signed
 - 2's Complement Integers
 - Binary-Decimal Conversion
 - Text: ASCII, Hex
 - Floating point representation
- Arithmetic and Logic Operations on Bits
 - Sign Extension and Overflow
- C datatypes and machine representation


CS 135



Course Trivia...Important

- CS 143 “tutorial” on Saturday 10am
 - Will get you up to speed with some systems issues:
 - Using compilers
 - Unix
 - editors


CS 135



Recall: what are Computers meant to do ?

- We will be solving problems that are describable in English (or Greek or French or Hindi or Chinese or ...) and using a box filled with electrons and magnetism to accomplish the task.
 - This is accomplished using a system of well defined (sometimes) transformations that have been developed over the last 50+ years.

CS 135




Problem Transformation - levels of abstraction

The desired behavior:
the application

Natural Language
Algorithm
Program
Machine Architecture
Micro-architecture
Logic Circuits
Devices

The building blocks:
electronic devices


CS 135



Representing Information/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
 - Logical
 - sound
 - Images
 - Video
 - ...
 - We will start by examining different ways of representing *integers*, and look for a form that suits the computer.


CS 135



Datatypes

- A type is a classification of data that tells the compiler or interpreter how the programmer intends to use it.
 - the process/algorithm and result of adding two variables differs greatly according to whether they are integers, floating point numbers, or strings.
- Patt: "We say a particular representation is a datatype if there are operations in the computer that can operate on information that is encoded in that representation."
- Programming languages have a set of data types defined in the language
 - In C : float, int, long int, unsigned int


CS 135



Machine Representation

- Computers store variables (data)
- We/program-logic reason about numbers in many different ways
 - Datatype may restrict what we can do
 - Example ??
- Key is to use a representation that is amenable to hardware implementation and is "efficient"
 - What do we mean by efficient ??


CS 135



Data Representation in a Machine

- Question: How do computers store numbers
when you write in a program:
int x; float y;
x=10;
y= 0.34;
- What do the variables x and y actually hold ???
 - Later question: where are x and y stored?


CS 135



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
- Simplest number representation ?
 - What did you (as an infant) start with ?
 - Also used by Turing machine....


CS 135



Radix-k notation

- Use k symbols – also known as k-ary numbers (radix k)
 - 0,1,2,...k-1
 - 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
- How many different radix k numbers of length n
 - Recall CS 123 –Discrete Math:
 - Each of the n positions can have k values

CS 135



Weighted Positional Numbers

- “decimal” means that we have ten digits to use in our representation (the symbols 0 through 9)
 - Weighted positional: “weight” of digit depends on its position
- What is 3,546?
 -
- How about negative numbers?
 -

CS 135

Binary Representation

- Weighted positional notation using base (radix) 2
- What are the symbols in base 2 ?
- k bit number: $b_{k-1}, b_{k-2}, \dots, b_1, b_0$
 - > Each b_i is ?
- Decimal integer N for this binary number is:
N =

CS 135

Binary Numbers

- $11_{10} = (???)_2$
- $011_2 = (??)_{10}$
- How many different numbers can be represented using N bits ?
- What is the largest decimal number representable using N binary bits ?

CS 135

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 r and you have an algorithm that computes the base r representation for N

CS 135

Unsigned Binary Arithmetic

- Base-2 addition – just like base-10!
 - > add from right to left, propagating carry

$\begin{array}{r} 10010 \\ + 1001 \\ \hline 11011 \end{array}$	$\begin{array}{r} 10010 \\ + 1011 \\ \hline 11101 \end{array}$	$\begin{array}{r} 1111 \\ + 1 \\ \hline 10000 \end{array}$
--	--	--

$\begin{array}{r} 10111 \\ + 111 \\ \hline \end{array}$

Subtraction, multiplication, division,...

CS 135

Question

- Why do we bother with bits? Why not simply use a decimal system?
 - > **Base 10 Number Representation**
 - > fingers are called "digits"
 - > Natural representation for financial transactions
 - > Floating point number cannot exactly represent \$1.20
 - > Even carries through in scientific notation
 - > 1.5213×10^4

CS 135

Why Don't Computers Use Base 10?

- Implementing Electronically
 - > **Hard to store**
 - > ENIAC (First electronic computer) used 10 vacuum tubes / digit
 - > **Hard to transmit**
 - > Need high precision to encode 10 signal levels on single wire
 - > **Messy to implement digital logic functions**
 - > Addition, multiplication, etc.

CS 135

How do we represent data in a computer?

- At the lowest level, a computer is an electronic machine.
 - > works by controlling the flow of electrons
- Easy to recognize two conditions:
 1. presence of a voltage – we'll call this state "1"
 2. absence of a voltage – we'll call this state "0"
- Do you need any special hardware to test 0 or 1 in your wall socket ?
- Could base state on *value* of voltage, but control and detection circuits more complex.
 - > compare turning on a light switch to measuring or regulating voltage

CS 135

Computer is a binary digital system.

Digital system:

- finite number of symbols

Binary (base two) system:


- has two states: 0 and 1

Digital Values ▶ "0" Illegal "1"

Analog Values ▶ 0 0.5 2.4 2.9 Volts

- Basic unit of information is the *binary digit*, or *bit*.
- Values with more than two states require multiple bits.
 - > A collection of **two** bits has **four** possible states: 00, 01, 10, 11
 - > A collection of **three** bits has **eight** possible states: 000, 001, 010, 011, 100, 101, 110, 111
 - > A collection of n bits has 2^n possible states.


CS 135



Machine Data Types

- devices that make up a computer are switches that can be on or off, i.e. at high or low voltage.
 - Thus they naturally provide us with two symbols to work with: we can call them **on** & **off**, or (more usefully) **0** and **1**.


CS 135



Course Adminis-trivia

- Labs and lab exercises/assignments start next week
 - Lab exercise given out at start of lab
 - Finish it in first part of lab; if you cannot then you can turn it in within 24 hours.
- Teams assigned
 - Will change teams at least once
- office hours – go to those!
- Homework 1 and Team homework 1 – will be posted Tuesday.
- Quiz 1 will be next week- on Thursday Sept.9th at start of class 2:20pm
 - If you come late then you will miss the quiz.


CS 135



Binary Representation...more to consider

- We now have a model for representing natural numbers using binary notation
- Is this all we need for all 'applications' ?
 - Do you need to work with other types of numbers/data ?
 - Is this enough to represent all integers ?

CS 135



Unsigned Binary Integers

Y = "abc" = a.2² + b.2¹ + c.2⁰

N = number of bits	3-bits	5-bits	8-bits
Range is: 0 ≤ i < 2 ^N - 1	0 000	00000	00000000
	1 001	00001	00000001
	2 010	00010	00000010
	3 011	00011	00000011
	4 100	00100	00000100

Problem:

- How do we represent *negative* numbers?

CS 135

Signed Integers

- Signed magnitude
 - > Easy to understand
 - > Fun to be with
 - > Not so good with hardware
- 1's Complement
 - > To get negative Just flip all the bits
 - > Used in some early computers

CS 135

Signed Magnitude

- Leading bit is the sign bit

$Y = "abc" = (-1)^a (b.2^1 + c.2^0)$

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

Problems: ?

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

CS 135

Signed Magnitude

- Leading bit is the sign bit

$Y = "abc" = (-1)^a (b.2^1 + c.2^0)$

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

Problems:

- How do we do addition/subtraction?
 - What is $2 + (-1)$?
- We have two numbers for zero (+/-)!

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

CS 135

One's Complement

- Invert all bits

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

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

Problems: ?

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

CS 135

One's Complement

- Invert all bits

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

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

Problems:

- What is $2 + (-1)$?
- Same as for signed magnitude

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

CS 135

Reality Sets In

- Any arbitrary scheme of allowing different bit patterns to represent different numbers could be made to work but...
- Problems with sign-magnitude and 1's complement
 - > two representations of zero (+0 and -0)
 - > arithmetic circuits are complex
 - > How to add two sign-magnitude numbers?
 - > e.g., try $2 + (-1)$
 - > How to add to one's complement numbers?
 - > e.g., try $2 + (-1)$

CS 135

Two's Complement

- 2's complement
 - > Used in almost all computers manufactured today
 - > Allows easy to implement math operations
 - > How does it work?
- Two's complement representation developed to make circuits easy for arithmetic.
 - > for each positive number (X), assign value to its negative (-X), such that $X + (-X) = 0$ with "normal" addition, ignoring carry out

00101 (5)	01001 (9)
+ 11011 (-5)	+ _____ (-9)
00000 (0)	00000 (0)

CS 135

Two's Complement Representation

- If number is positive or zero,
 - > normal binary representation, zeroes in upper bit(s)
- If number is negative,
 - > start with positive number
 - > flip every bit (i.e., take the one's complement)
 - > then add one

00101 (5)	01001 (9)
11010 (1's comp)	11000 (1's comp)
+ _____ 1	+ _____ 1
11011 (-5)	11001 (-9)

CS 135

Two's Complement

- Transformation**
 - To transform a into -a, invert all bits in a and add 1 to the result

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

Advantages:

- Operations need not check the sign
- Only one representation for zero
- Efficient use of all the bits

-16	10000
...	...
-3	11101
-2	11110
-1	11111
0	00000
+1	00001
+2	00010
+3	00011
...	...
+15	01111

CS 135

Two's Complement Shortcut

- To take the two's complement of a number:
 - copy bits from right to left until (and including) the first "1"
 - flip remaining bits to the left

$$\begin{array}{r}
 01101000 \\
 10010111 \quad (1's \text{ comp}) \\
 + \quad \quad \quad 1 \\
 \hline
 10011000
 \end{array}$$

$$\begin{array}{r}
 01101000 \\
 \downarrow \quad \downarrow \\
 10011000
 \end{array}$$

(flip) (copy)

Question to think about: Why does this trick work ?

CS 135

N = 4				
Binary	Number Represented			
	Unsigned	Signed Mag	1's Comp	2's Comp
0000	0	0	0	0
0001	1	1	1	1
0010	2	2	2	2
0011	3	3	3	3
0100	4	4	4	4
0101	5	5	5	5
0110	6	6	6	6
0111	7	7	7	7
1000	8	-0	-7	-8
1001	9	-1	-6	-7
1010	10	-2	-5	-6
1011	11	-3	-4	-5
1100	12	-4	-3	-4
1101	13	-5	-2	-3
1110	14	-6	-1	-2
1111	15	-7	-0	-1

CS 135

Converting Binary (2's C) to Decimal

- If leading bit is one, take two's complement to get a positive number.
- Add powers of 2 that have "1" in the corresponding bit positions.
- If original number was negative, add a minus sign.

$$\begin{aligned}
 X &= 01101000_{\text{two}} \\
 &= 2^6 + 2^5 + 2^3 = 64 + 32 + 8 \\
 &= 104_{\text{ten}}
 \end{aligned}$$

n	2 ⁿ
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024

CS 135

More Examples

X = 00100111_{two}
=
=

X = 11100110_{two}
-X =
=
=
X =

n	2 ⁿ
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024

Assuming 8-bit 2's complement numbers.

CS 135

Converting Decimal to Binary (2's C)

- **First Method: Division**
- 1. Find magnitude of decimal number. (Always positive.)
- 2. Divide by two – remainder is least significant bit.
- 3. Keep dividing by two until answer is zero, writing remainders from right to left.
- 4. Append a zero as the MS bit; if original number was negative, take two's complement.

Question to think about: what is going on here?

CS 135

Converting Decimal to Binary (2's C)

X = 104 _{ten}	104/2 = 52 r0	bit 0
	52/2 = 26 r0	bit 1
	26/2 = 13 r0	bit 2
	13/2 = 6 r1	bit 3
	6/2 = 3 r0	bit 4
	3/2 = 1 r1	bit 5
X = 01101000 _{two}	1/2 = 0 r1	bit 6

CS 135

Encoding Integers

Unsigned

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

Two's Complement

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

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

↖ Sign Bit

➤ **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

CS 135

Encoding Example (Cont.)

x = 15213: 00111011 01101101
 y = -15213: 11000100 10010011

Weight	15213		-15213	
1	1	1	1	1
2	0	0	1	2
4	1	4	0	0
8	1	8	0	0
16	0	0	1	16
32	1	32	0	0
64	1	64	0	0
128	0	0	1	128
256	1	256	0	0
512	1	512	0	0
1024	0	0	1	1024
2048	1	2048	0	0
4096	1	4096	0	0
8192	1	8192	0	0
16384	0	0	1	16384
-32768	0	0	1	-32768
Sum	15213		-15213	

CS 135

Numeric Ranges

- **Unsigned Values**
 - > $UMin = 0$
000...0
 - > $UMax = 2^w - 1$
111...1

- **Two's Complement Values**
 - > $TMin = -2^{w-1}$
100...0
 - > $TMax = 2^{w-1} - 1$
011...1
- **Other Values**
 - > Minus 1

Values for $W = 16$

	Decimal	Hex	Binary
UMax	65535	FF FF	11111111 11111111
TMax	32767	7F FF	01111111 11111111
TMin	-32768	80 00	10000000 00000000
-1	-1	FF FF	11111111 11111111
0	0	00 00	00000000 00000000

CS 135

Values for Different Word Sizes

	W			
	8	16	32	64
UMax	255	65,535	4,294,967,295	18,446,744,073,709,551,615
TMax	127	32,767	2,147,483,647	9,223,372,036,854,775,807
TMin	-128	-32,768	-2,147,483,648	-9,223,372,036,854,775,808

- **Observations**
 - > $|TMin| = TMax + 1$
Asymmetric range
 - > $UMax = 2 * TMax + 1$


- **C Programming**
 - > `#include <limits.h>`
K&R App. B11
 - > **Declares constants, e.g.,**
ULONG_MAX
LONG_MAX
LONG_MIN
 - > **Values platform-specific**

CS 135

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?

CS 135




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:

<u>4-bit</u>	<u>8-bit</u>
0100 (4)	00000100 (still 4)
1100 (-4)	00001100 (12, not -4)

CS 135



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:

<u>4-bit</u>	<u>8-bit</u>
0100 (4)	00000100 (still 4)
1100 (-4)	00001100 (12, not -4)
- Instead, replicate the MS bit -- the sign bit:

<u>4-bit</u>	<u>8-bit</u>
0100 (4)	00000100 (still 4)
1100 (-4)	11111100 (still -4)

Question to think about: why does this work?


CS 135



Other Data Types

- Other numeric data types
 - e.g. Hex, BCD – binary coded decimal
- Bit vectors & masks
 - sometimes we want to deal with the individual bits themselves
- Floating Point – for real numbers
- Text representations
 - ASCII: uses 8 bits to represent main Western alphabetic characters & symbols, plus several “control codes”,
 - Unicode: 16 bit superset of ASCII providing representation of many different alphabets and specialized symbol sets.
 - EBCDIC: IBM's mainframe representation.


CS 135



Reading binary strings

- Using binary numbers is both a blessing a curse!
 - One can examine directly any particular bit
 - Reading, writing, etc. prone to error
- What is easier to read and remember
 - 01101010
 - 6A


CS 135



Solution

- Group bits and assign a single digit to represent each group
- How big should each group be?

CS 135




Hexadecimal Notation

- It is often convenient to write binary (base-2) numbers as hexadecimal (base-16) numbers instead.
 - > fewer digits -- four bits per hex digit
 - > less error prone -- easy to corrupt long string of 1's and 0's

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

CS 135



Converting from Binary to Hexadecimal


- Every four bits is a hex digit.
 - > start grouping from right-hand side

```

011101010001111010011010111
  ↓   ↓   ↓   ↓   ↓   ↓   ↓
  3   A   8   F   4   D   7
    
```

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


CS 135



Hexadecimal Notation

- Number of k length represented by string of HEX symbols
 - > $h_{k-1}, h_{k-2}, \dots, h_1, h_0$
 - > Each h_i has valued between 0 and F
 - > A=10, B=11, C=12, D=13, E=14, F=15
- Decimal equivalent N is:
 - > $N = h_{k-1} * 16^{k-1} + \dots + h_1 * 16^1 + h_0 * 16^0$
- Eg: 2B7 = $2 * 16^2 + 11 * 16^1 + 7 * 16^0$
- also denoted as x2B7


CS 135



ASCII Codes

- Represent characters from keyboard
 - This encoding used to transfer characters between computer and all peripherals (keyboard, disk, network...)
- ASCII: American Standard Code for Information Interchange
 - 7 bits needed to encode all characters
 - Represent as 8 bit number
- Typing a key on keyboard = corresponding 8-bit ASCII code is stored and sent to computer

CS 135




Text: ASCII Characters

- 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	e	50	P	60	~	70	p
01	soh	11	dcl	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

CS 135




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 characters, how do we tell which comes first in alphabetical order?
- Are 128 characters enough? (<http://www.unicode.org/>)

No new operations – integer arithmetic and logic.

CS 135



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 representation.
- How to deal with Real numbers...We need other data types!

CS 135

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

CS 135

Scientific Notation

-6.023×10^{-23}

Diagram illustrating the components of scientific notation:

- Sign:** The negative sign (-) is labeled as the sign.
- Normalized Mantissa:** The decimal part 6.023 is labeled as the normalized mantissa.
- Base:** The base 10 is labeled as the base.
- Exponent:** The power -23 is labeled as the exponent.
- Sign of Exponent:** The negative sign of the exponent is labeled as the sign of the exponent.

CS 135

Binary Floating Point Representation

- Same basic idea as scientific notation
- Modifications and improvements based on
 - Hardware architecture
 - Efficiency (Space & Time)
 - Additional requirements
 - Infinity
 - Not a number
 - Not normalized
 - etc.

CS 135


IEEE-754

Diagram illustrating the IEEE-754 floating point format:

31	30	23	22	0
0	00000000	000000000000000000000000		
s		exponent		
i		mantissa (significand)		
g				
n				


$(-1)^S * 1.M * 2^{E-127}$

Sign —————
 1 is understood —————
 Mantissa (w/o leading 1) —————
 Base —————
 Biased Exponent —————

CS 135: Computer Architecture I


Instructor: Prof. Bhagi Narahari
 Dept. of Computer Science
 Course URL: www.seas.gwu.edu/~bhagiweb/cs135/



Operations: Arithmetic and Logical

- Recall: a data type includes *representation* and *operations*.
- We now have a good representation for signed integers, so let's look at some arithmetic operations:
 - > Addition
 - > Subtraction
 - > Sign Extension
- overflow conditions for addition.
- Multiplication, division, etc., can be built from these basic operations.
- Logical operations are also useful:
 - > AND
 - > OR
 - > NOT

CS 135




Addition

- As we've discussed, 2's comp. addition is just binary addition.
 - > assume all integers have the same number of bits
 - > ignore carry out
 - > for now, assume that sum fits in n-bit 2's comp. representation

01101000 (104)	+	11110110 (-10)	+	_____ (-9)
+ 11110000 (-16)				
01011000 (98)				(-19)

Assuming 8-bit 2's complement numbers.

CS 135



What happens when we add a number to itself?

CS 135

Subtraction

- **Negate subtrahend (2nd no.) and add.**
 - > assume all integers have the same number of bits
 - > ignore carry out
 - > for now, assume that difference fits in n-bit 2's comp. representation

01101000 (104)	11110110 (-10)
- 00010000 (16)	- _____ (-9)
01101000 (104)	11110110 (-10)
+ 11110000 (-16)	+ _____ (9)
01011000 (88)	_____ (-1)

Assuming 8-bit 2's complement numbers.

CS 135

Example..

- From team hw1: (i)

CS 135

Overflow

- If operands are too big, then sum cannot be represented as an n-bit 2's comp number.

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

- **We have overflow if:**
 - > signs of both operands are the same, and
 - > sign of sum is different.

CS 135

Overflow

- If we add two positive numbers and we get a carry into the sign bit we have a problem

3	0011	4	0100
3	0011	4	0100
6	0110	8	1000

CS 135

Overflow

- If we add two positive numbers and we get a carry into the sign bit we have a problem

3	0011	4	0100
3	0011	4	0100
<u>6</u>	<u>0110</u>	<u>8</u>	<u>1000</u>

carry in 0
carry out 0

carry in 1
carry out 0

CS 135

Overflow

- If we add two negative numbers and we get a carry into the sign bit do we always have a problem

-5	1011	-4	1100
-2	1110	-5	1011
<u>-7</u>	<u>1110</u>	<u>-9</u>	<u>1011</u>

carry in 0
carry out 0

carry in 1
carry out 0

CS 135

Overflow

- If we add two negative numbers and we get a carry into the sign bit do we have a problem

-5	1011	-4	1100
-2	1110	-5	1011
<u>-7</u>	<u>11001</u>	<u>-9</u>	<u>10111</u>

carry in 1
carry out 1

carry in 0
carry out 1

CS 135

Overflow


- If we add a positive and a negative number we won't have a problem

5	0101	-4	1100
-3	1101	5	0101
<u>2</u>	<u>10010</u>	<u>1</u>	<u>10001</u>

carry in 1
carry out 1

carry in 1
carry out 1


CS 135



Overflow

- Carry out of the leading digit
- If we add two positive numbers and we get a carry into the sign bit we have a problem
- If we add two negative numbers and we get a carry into the sign bit do we have a problem (??)
- If we add a positive and a negative number we won't have a problem


CS 135



Overflow Condition

- In terms of Carry-in and Carry-out ??
- If carry in is not equal to carry out of MSB then overflow – easy to check in H/W

CS 135




Multiplication

- How do you multiply two decimal numbers:
 - > $25 \times 32 = 25 \times (3 \cdot 10^1 + 2 \cdot 10^0)$
 - > $25 + 25 \times 10^1 \times 3$
- Similar method for binary
 - > For every bit position in the multiplier, shift multiplicand left, multiply by 1 or 0 and add to result
 - > $1101 \times 1011 = (13 \times 11)$

```

1101
 1101
 0000
 1101
-----
10001111      (143)
            
```


CS 135



A note on computer arithmetic


- What actual type of additions (arith operations) are taking place ?
 - > recall: max number T_{max} represented by N bit unsigned integer is 2^N
 - > If $N=8$ then max is 255
- Modulo arithmetic: $\text{mod}(T_{max} + 1)$
 - > If $N=8$, then $(a+b) = (a+b) \text{ mod } 256$
- Even bigger problem with multiplication
 - > $2N$ bit product, but truncated to N bits!
- Is $(a+b) + c = a + (b+c)$?
- Not always!! What if b, c are large negative numbers- can have overflow

CS 135



Logical Operators


CS 135



Another use for bits: Logic

- Beyond numbers
 - > *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
 - > The manipulation of logical variables is known as **Boolean Algebra**, and has its own set of operations - which are not to be confused with the arithmetical operations of the previous section.
- Some basic operations: NOT, AND, OR, XOR

CS 135




Propositional Logic

- 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		A	B	A OR B		A	NOT A
F	F	F		F	F	F		F	T
F	T	F		F	T	T		T	F
T	F	F		T	F	T		F	F
T	T	T		T	T	T		T	F

CS 135




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		A	B	A OR B		A	NOT A
0	0	0		0	0	0		0	1
0	1	0		0	1	1		1	0
1	0	0		1	0	1		1	0
1	1	1		1	1	1		1	0


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

CS 135



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


CS 135





Logic Operations..more examples

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

CS 135

- 
- Basic Logic Operations**
- **Equivalent Notations**
 - > not A = $A' = \bar{A}$
 - > 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**
- CS 135

- 
- Bitwise Logical Operators**
- **View n -bit number as a collection of n logical values**
 - > operation applied to each bit independently
 - **Number operated on is an n -bit number**
 - **Operation being performed is logical operation on each bit**
- CS 135




Bitwise AND

0101 AND 0111 *in C: (5 & 6)*

0101
0110


CS 135



Why use bitwise operators?

- Masking operations
 - > If we are only interested in last 8 bits of a 16 bit number X, how to extract this?
 - > X & x00FF
 - > Zero out the most significant 8 bits; value of least significant 8 bits is same as the value of these in X
 - > x27A4 & x00FF = x00A4

CS 135




Bitwise OR

0101 OR 0111 *in C: (5 | 6)*

0101
0110

CS 135




Bitwise NOT (Complement)

NOT 0101 *in C: ~5*

0101

CS 135




Bitwise XOR

0101 XOR 0111 *in C: 5^6*

0101
0110

CS 135




Bitwise NAND

0101 NAND 0111 *No C Operator*
~(5 & 6)

0101
0110

CS 135




Bitwise NOR

0101 NOR 0111 *No C Operator*
~(5 | 6)

0101
0110


CS 135



Boolean Relational Operators ?

- **What is the semantics of:**
 - > If $(x==0)$ then
 - > how many outcomes for $(x==0)$?
- **Concept of boolean operators**
 - > Apply logic operators, but treat input and output as boolean variables
 - > Only 1 or 0 (True or False) values for entire variable
 - > But input strings can be n-bits long?
 - > Treat entire string as ONE boolean variable
 - > How ?
- **Boolean operators**
 - > Logical AND: $(x \text{ AND } y)=1$ if both x and y are non-zero
 - > Logical OR: $(x \text{ OR } y)=1$ if at least one of x,y are non-zero
 - > Logical NOT: $(\text{NOT } x)=1$ if $x=0$ and $(\text{NOT } x)=0$ if x is non-zero.


CS 135



LC-3 Data Types

- Some data types are supported directly by the instruction set architecture.
- For LC-3, there is only one hardware-supported data type:
 - 16-bit 2's complement signed integer
 - Operations: ADD, AND, NOT
- Other data types are supported by interpreting 16-bit values as logical, text, fixed-point, etc., in the software that we write.


CS 135



Next...a little bit of "reality"

- look at how some of the concepts we have studied take shape in 'real life'
 - C programming and O/S


CS 135



Data Representations

- **Sizes of C Objects (in Bytes)**
 - C Data Type Compaq Alpha Typical 32-bit Intel IA32
 - int 4 4 4
 - long int 8 4 4
 - char 1 1 1
 - short 2 2 2
 - float 4 4 4
 - double 8 8 8
 - long double 8 8 10/12
 - char * 8 4 4
 - Or any other pointer


CS 135



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?


CS 135



Bitwise Review

- 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)


CS 135



Shift Operations

- $x \gg y$
 - > x right shifted y bit positions, sign extended
 - > Sign bit shifted into positions vacated by shifted bits
 - > $x = 011000 \quad y = 2$
 - > $x \gg y = ?$
 - > $z = 101000 \quad y = 2$
 - > $z \gg y = ?$
- $x \ll y$
 - > x left shifted y bit positions, zero placed in positions vacated by shifted bits
 - > $x \ll y = ?$
 - > $z \ll y = ?$


CS 135



Logical Operations in C

- **!** Logical NOT
 - > $!x$
 - > $!x = 0$ if x is non-zero, $!x = 1$ if value of x is zero
- **&&** Logical AND
 - > $x \&\& y$
 - > $x \&\& y = 1$ if value of x is not zero and value of y is not zero
 - > $x \&\& y = 0$ if both x and y are zero
- **//** logical OR
 - > $x \|\| y$
 - > $x \|\| y = 1$ if at least one of x, y are not zero
 - > $x \|\| y = 0$ if both x, y are zero

CS 135



Examples

- 8 bit numbers, $f = 7, g = 8$
 - > $f = 00000111 \quad g = 00001000$
- $h = (f \& g)$ (bitwise AND)....
 - > $h = ?$
- $h = (f \&\& g)$ (logical AND)....
 - > $h = ?$
- $h = (f | g)$ (bitwise OR)... $h = ?$
- $h = (f \|\| g)$ (logical OR).... $h = ?$
- $h = (\sim f | \sim g)$... $h = ?$
- $h = (!f \&\& !g)$... $h = ?$

CS 135

Unsigned & Signed Numeric Values

X	B2U(X)	B2T(X)
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

- **Equivalence**
 - Same encodings for nonnegative values
- **Uniqueness**
 - Every bit pattern represents unique integer value
 - Each representable integer has unique bit encoding
- **⇒ Can Invert Mappings**
 - $U2B(x) = B2U^{-1}(x)$
 - Bit pattern for unsigned integer
 - $T2B(x) = B2T^{-1}(x)$
 - Bit pattern for two's comp integer

CS 135

Signed vs. Unsigned in C

- **Constants**
 - By default are considered to be signed integers
 - Unsigned if have "U" as suffix
`0U, 4294967259U`
- **Casting**
 - Explicit casting between signed & unsigned same as U2T and T2U


```
int tx, ty;
unsigned ux, uy;
tx = (int) ux;
uy = (unsigned) ty;
```
 - Implicit casting also occurs via assignments and procedure calls


```
tx = ux;
uy = ty;
```

CS 135

Casting Signed to Unsigned

- **C Allows Conversions from Signed to Unsigned**

```
short int      x = 15213;
unsigned short int ux = (unsigned short) x;
short int      y = -15213;
unsigned short int uy = (unsigned short) y;
```

- **Resulting Value**
 - No change in bit representation – only in interpretation
 - What is value of ux ?
 - What is value of uy?

CS 135

Relation Between Signed & Unsigned

Weight	-15213		50323	
1	1	1	1	1
2	1	2	1	2
4	0	0	0	0
8	0	0	0	0
16	1	16	1	16
32	0	0	0	0
64	0	0	0	0
128	1	128	1	128
256	0	0	0	0
512	0	0	0	0
1024	1	1024	1	1024
2048	0	0	0	0
4096	0	0	0	0
8192	0	0	0	0
16384	1	16384	1	16384
32768	1	-32768	1	32768
Sum		-15213		50323

➢ $uy = y + 2 * 32768 = y + 65536$

CS 135

Relation between Signed & Unsigned

Two's Complement

T2B
T2U

$x \rightarrow$
X
 $\rightarrow ux$

Unsigned

Maintain Same Bit Pattern

$w-1$
 ux + + + + • • • + + + + 0

$-x$ - + + + • • • + + + +

$+2^{w-1} - 2^{w-1} = 2 * 2^{w-1} = 2^w$

$$ux = \begin{cases} x & x \geq 0 \\ x + 2^w & x < 0 \end{cases}$$

CS 135

Casting Signed to Unsigned

- C Allows Conversions from Signed to Unsigned

```
short int      x = 15213;
unsigned short int ux = (unsigned short) x;
short int      y = -15213;
unsigned short int uy = (unsigned short) y;
```

- Resulting Value
 - > No change in bit representation
 - > Nonnegative values unchanged
 - > $ux = 15213$
 - > Negative values change into (large) positive values
 - > $uy = 50323$

CS 135

Signed vs. Unsigned in C

- Constants
 - > By default are considered to be signed integers
 - > Unsigned if have "U" as suffix
`0U, 4294967259U`
- Casting
 - > Explicit casting between signed & unsigned same as U2T and T2U


```
int tx, ty;
unsigned ux, uy;
tx = (int) ux;
uy = (unsigned) ty;
```
 - > Implicit casting also occurs via assignments and procedure calls


```
tx = ux;
uy = ty;
```

CS 135

Casting Surprises

- Expression Evaluation
 - > If mix unsigned and signed in single expression, signed values implicitly cast to unsigned
 - > Including comparison operations `<`, `>`, `==`, `<=`, `>=`
 - > Examples for $W = 32$

Constant ₁	Constant ₂	Relation	Evaluation
0	0U	==	unsigned
-1	0	<	signed
-1	0U	>	unsigned
2147483647	-2147483648	>	signed
2147483647U	-2147483648	<	unsigned
-1	-2	>	signed
(unsigned) -1	-2	>	unsigned
2147483647	2147483648U	<	unsigned
2147483647	(int) 2147483648U	>	signed

CS 135

Why Should I Use Unsigned?

- **Don't Use Just Because Number Nonzero**
 - Easy to make mistakes
 - `for (i = cnt-2; i >= 0; i--)`
 - `a[i] += a[i+1];`
- **Do Use When Performing Modular Arithmetic**
 - Multiprecision arithmetic
 - Other esoteric stuff
- **Do Use When Need Extra Bit's Worth of Range**
 - Working right up to limit of word size

CS 135

Byte-Oriented Memory Organization

- **Programs Refer to Virtual Addresses**
 - Conceptually very large array of bytes
 - Actually implemented with hierarchy of different memory types
 - SRAM, DRAM, disk
 - Only allocate for regions actually used by program
 - In Unix and Windows NT, address space private to particular "process"
 - Program being executed
 - Program can clobber its own data, but not that of others
- **Compiler + Run-Time System Control Allocation**
 - Where different program objects should be stored
 - Multiple mechanisms: static, stack, and heap
 - In any case, all allocation within single virtual address space

CS 135

Encoding Byte Values

- **Byte = 8 bits**
 - Binary 00000000_2 to 11111111_2
 - Decimal: 0_{10} to 255_{10}
 - Hexadecimal 00_{16} to FF_{16}
 - Base 16 number representation
 - Use characters '0' to '9' and 'A' to 'F'
 - Write $FA1D37B_{16}$ in C as $0xFA1D37B$
 - Or $0xfa1d37b$

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

CS 135

Machine Words

- **Machine Has "Word Size"**
 - Nominal size of integer-valued data
 - Including addresses
 - Most current machines are 32 bits (4 bytes)
 - Limits addresses to 4GB
 - Becoming too small for memory-intensive applications
 - High-end systems are 64 bits (8 bytes)
 - Potentially address $\approx 1.8 \times 10^{19}$ bytes
 - Machines support multiple data formats
 - Fractions or multiples of word size
 - Always integral number of bytes

CS 135

Representing Integers

- int A = 15213;
- int B = -15213;
- long int C = 15213;

Decimal: 15213
Binary: 0011 1011 0110 1101
Hex: 0000 3 B 6 D
Decimal: -15213
Hex: FFFF C 4 9 3

- Little endian layout for A:
 - For B
 - For C
- Big endian layout for A:
 - For B:
 - For C:

CS 135

Representing Integers

- int A = 15213;
- int B = -15213;
- long int C = 15213;

Decimal: 15213
Binary: 0011 1011 0110 1101
Hex: 3 B 6 D

Linux/Alpha A Sun A

6D	00
3B	00
00	3B
00	6D

Linux C Alpha c Sun c

6D	6D	00
3B	3B	00
00	00	3B
00	00	6D

Linux/Alpha B Sun B

93	FF
C4	FF
FF	C4
FF	93

Two's complement representation
(Covered next lecture)

CS 135

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

CS 135

What next....reading

- Read floating point representation
- Quiz 1 on Thursday
- HW1 posted
- In-lab exercises this week
- Check your team memberships
- Will be posting team discussion problems by Thursday 6pm
 - Come prepared after team discussions
- Read Chapter 3


CS 135



Limitations of integer representations

- **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 representation.
- **We need other data types!**


CS 135



Some History...

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


CS 135



Real numbers

- Our decimal system handles non-integer *real* numbers by adding yet another symbol - the decimal point (.) to make a *fixed point* notation:
 - e.g. $3,456.78 = 3 \cdot 10^3 + 5 \cdot 10^2 + 4 \cdot 10^1 + 6 \cdot 10^0 + 7 \cdot 10^{-1} + 8 \cdot 10^{-2}$
- How can we represent fractions?
 - Use a “binary point” to separate positive from negative powers of two -- just like “decimal point.”

CS 135



Real Numbers and Fractions in Binary

- How to represent $6 \frac{5}{8}$ in binary
- $6 \frac{5}{8} = 4 + 2 + \frac{1}{2} + \frac{1}{8}$
- $6 \frac{5}{8} = (1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3})$
- $6 \frac{5}{8} = 110.101$

CS 135

Fractions: Fixed-Point

- 2's comp addition and subtraction still work.
 - if binary points are aligned

00101000.101 (40.625)
 $+ 11111110.110$ (-1.25)

 00100111.011 (39.375)

No new operations -- same as integer arithmetic.

CS 135

Do we still have problems ?

- What about representing very large and very small numbers ?
 - Using N bits, we can only represent x where $-2^N \leq x \leq 2^N - 1$
- The *floating point*, or scientific, notation allows us to represent very large and very small numbers (integer or real), with as much or as little precision as needed:
 - Unit of electric charge $e = 1.602\ 176\ 462 \times 10^{-19}$ Coul.
 - Volume of universe = 1×10^{85} cm³
 - the two components of these numbers are called the mantissa and the exponent

CS 135

Real numbers in binary

- We mimic the decimal floating point notation to create a “hybrid” binary floating point number:
 - We first use a “binary point” to separate whole numbers from fractional numbers to make a fixed point notation:
 - e.g. $00011001.110 = 1.2^4 + 1.10^3 + 1.10^1 + 1.2^{-1} + 1.2^{-2} \Rightarrow 25.75$
($2^{-1} = 0.5$ and $2^{-2} = 0.25$, etc.)
 - We then “float” the binary point:
 - $00011001.110 \Rightarrow 1.1001110 \times 2^4$
mantissa = 1.1001110, exponent = 4
 - Now we have to express this without the extra symbols (x, 2, .)
 - by convention, we divide the available bits into three fields: sign, mantissa, exponent

CS 135

Scientific Notation

-6.023×10^{-23}

CS 135

Binary Floating Point Representation

- Same basic idea as scientific notation
- Modifications and improvements based on
 - > Hardware architecture
 - > Efficiency (Space & Time)
 - > Additional requirements
 - > Infinity
 - > Not a number
 - > Not normalized
 - > etc.

CS 135

IEEE-754

31	30	23	22	0
0	00000000	000000000000000000000000		
s	exponent	mantissa (significand)		

$(-1)^S * 1.M * 2^{E-127}$

Sign —————
 1 is understood —————
 Mantissa (w/o leading 1) —————
 Base —————
 Biased Exponent —————

IEEE-754

0	00000000	000000000000000000000000
s	exponent	mantissa (significand)

$(-1)^S * 1.M * 2^{E-127}$

Can any of these equal 0?

CS 135

IEEE-754 fp numbers - 2

- Example:
 - > 25.75 => 00011001.110 => 1.1001110 x 2⁴
 - > sign bit = 0 (+ve)
 - > normalized mantissa (fraction) = 100 1110 0000 0000 0000 0000
 - > biased exponent = 4 + 127 = 131 => 1000 0011
 - > so 25.75 => 0 1000 0011 100 1110 0000 0000 0000 0000 => x41CE0000
- Values represented by convention:
 - > Infinity (+ and -): exponent = 255 (1111 1111) and fraction = 0
 - > NaN (not a number): exponent = 255 and fraction ≠ 0
 - > Zero (0): exponent = 0 and fraction = 0

CS 135 note: exponent = 0 => fraction is de-normalized, i.e no hidden 1

So how can we represent 0?

0	00000000	000000000000000000000000
s i g n	exponent	mantissa (significand)
	$(-1)^S * 1.M * 2^{E-127}$	

	E == 0	0 < E < 255	E == 255
M == 0	0	Powers of Two	∞
M != 0	Non-normalized typically underflow	Ordinary Old Numbers	Not A Number

CS 135

Can be written...

0	00000000	000000000000000000000000
s i g n	exponent	mantissa (significand)
	$(-1)^S * 2^{E-127} * 1.M$	

	E == 0	0 < E < 255	E == 255
M == 0	0	Powers of Two	∞
M != 0	Non-normalized typically underflow	Ordinary Old Numbers	Not A Number

CS 135

Pop Quiz!!!

George Washington University

0	00000000	000000000000000000000000	= 0
1	00000000	000000000000000000000000	= -0

s i g n	exponent	mantissa (significand)
	$(-1)^S * 2^{E-127} * 1.M$	

	E == 0	0 < E < 255	E == 255
M == 0	0	Powers of Two	∞
M != 0	Non-normalized typically underflow	Ordinary Old Numbers	Not A Number

CS 135

0 11111111 000000000000000000000000 = Infinity
 1 11111111 000000000000000000000000 = -Infinity

0	00000000	000000000000000000000000
s i g n	exponent	mantissa (significand)
	$(-1)^S * 2^{E-127} * 1.M$	

	E == 0	0 < E < 255	E == 255
M == 0	0	Powers of Two	∞
M != 0	Non-normalized typically underflow	Ordinary Old Numbers	Not A Number

CS 135

0 11111111 000001000000000000000000 = NaN
 1 11111111 00100010001001010101010 = NaN

0	00000000	000000000000000000000000
s i g n	exponent	mantissa (significand)
	$(-1)^S * 2^{E-127} * 1.M$	

	E == 0	0 < E < 255	E == 255
M == 0	0	Powers of Two	∞
M != 0	Non-normalized typically underflow	Ordinary Old Numbers	Not A Number

CS 135

0 10000000 000000000000000000000000 = +1 * 2⁽¹²⁸⁻¹²⁷⁾ * 1.0 = 2

0	00000000	000000000000000000000000
s i g n	exponent	mantissa (significand)
	$(-1)^S * 2^{E-127} * 1.M$	

	E == 0	0 < E < 255	E == 255
M == 0	0	Powers of Two	∞
M != 0	Non-normalized typically underflow	Ordinary Old Numbers	Not A Number


CS 135

0 10000001 101000000000000000000000 = +1 * 2⁽¹²⁹⁻¹²⁷⁾ * 1.101 = 6.5
 1 10000001 101000000000000000000000 = -1 * 2⁽¹²⁹⁻¹²⁷⁾ * 1.101 = -6.5

0	00000000	000000000000000000000000
s i g n	exponent	mantissa (significand)
	$(-1)^S * 2^{E-127} * 1.M$	

	E == 0	0 < E < 255	E == 255
M == 0	0	Powers of Two	∞
M != 0	Non-normalized typically underflow	Ordinary Old Numbers	Not A Number

CS 135



 0 00000001 000000000000000000000000 = +1 * 2⁽¹⁻¹²⁷⁾ * 1.0 = 2⁽⁻¹²⁶⁾


 0 00000000 100000000000000000000000 = +1 * 2⁽⁻¹²⁶⁾ * 0.1 = 2⁽⁻¹²⁷⁾

0	00000000	000000000000000000000000
s i g n	exponent	mantissa (significand)

$(-1)^S * 2^{E-127} * 1.M$

	E == 0	0 < E < 255	E == 255
M == 0	0	Powers of Two	∞
M != 0	Non-normalized typically underflow	Ordinary Old Numbers	Not A Number

CS 135



Even Larger Numbers? IEEE-754 fp double precision

- Double precision (64 bit) floating point

s	biased exp.	fraction
s	fraction	biased exp.

- 32 bit:
 - mantissa of 23 bits + 1 => approx. 7 digits decimal
 - 2^{+/-127} => approx. 10^{+/-38}
- 64 bit:
 - mantissa of 52 bits + 1 => approx. 15 digits decimal
 - 2^{+/-1023} => approx. 10^{+/-306}

CS 135