ECE/CS 250
Computer Architecture

"From C to Binary"

Duke University

Slides are derived from work by
Drew Hilton (Duke), Alvy Lebeck (Duke),
Benjamin Lee (Duke), Amir Roth (Penn)

Outline

- Previously:
  - Computer is machine that does what we tell it to do
- Now:
  - How do we tell computers what to do?
  - How do we represent variables with bits?

Representing High Level Things in Binary

- Computers represent everything in binary
- Instructions are specified in binary
- Instructions must be able to describe the following:
  - Operation types (add, subtract, shift, etc.)
  - Data objects (integers, decimals, characters, etc.)
  - Memory locations
- Examples:

```
int x, y;          // Where are x and y?  How to represent an int?
bool decision;     // How to represent a bool?  Where is it?
int* z;            // Where is z?  How to represent an int*?
y = x + 7;         // How to specify "add"?  How to represent 7?
decision=(y>18);   //  Etc.
```

Number Bases

- We use multiple bases for numbers in this class
  - Base-10 (decimal), base-2 (binary), base-16 (hexadecimal), etc.
  - When ambiguous, will note it explicitly
- Base-10: place values are powers of 10
  - At each place value, digit can be 0-9

X	X	X	X	.	X	X	X
10^3	10^2	10^1	10^0	.	10^{-1}	10^{-2}	10^{-3}
3	0	4	9	.	5	7	7

Number Bases: Binary (base 2)

- Base-2: place values are powers of 2
  - At each place value, **bit** can be 0 or 1

x	x	x	x	.	x	x	x
2^3	2^2	2^1	2^0	.	2^{-1}	2^{-2}	2^{-3}
1	1	0	0	.	1	0	1

  - So what is the value of 1100.101_2?
    - 8+4+1/2+1/8 = 12.625

Number Bases: Hexadecimal (base 16)

- Hexadecimal = Base-16: place values are powers of 16
  - At each place value, can be 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
  - $A=10_{10}$, $B=11_{10}$, $C=12_{10}$, $D=13_{10}$, $E=14_{10}$, $F=15_{10}$
  - Each hex "digit" corresponds to exactly 4 bits
  - Why hex? Concise way to write binary numbers
    - Usually denoted with 0x prefix, e.g., 0x2A3

X	X	X	X	.	X	X	X
16^3	16^2	16^1	16^0	.	16^{-1}	16^{-2}	16^{-3}
0	2	A	3	.	0	0	0

  - So what is the value of 0x2A3?
    - $2*16^2 + A*16^1 + 3*16^0$
    - $= 2*256 + 10*16 + 3*1 = 675_{10}$

Representing Operation Types

- Q: How do we tell computer to add? Shift? Read from memory? Etc.
- A: Arbitrarily! ☺

- Each **Instruction Set Architecture (ISA)** has its own binary encodings for each operation type
- E.g., in MIPS:
  - Integer add is: 00000 010000
  - Read from memory (load) is: 010011
  - Etc.

Representing Data Types

- How do we specify an integer? A character? A floating point number? A bool? Etc.
- Same as before: binary!
- Key Idea: the same 32 bits might mean one thing if interpreted as integer but another thing if interpreted as floating point number

Basic Data Types

Bit (bool): 0, 1

Bit String: sequence of bits of a particular length
- 4 bits is a nibble
- → **8 bits is a byte**
- 16 bits is a half-word
- 32 bits is a word
- 64 bits is a double-word
- 128 bits is a quad-word

Integers (int, long):
- "2's Complement" (32-bit or 64-bit representation)

Floating Point (float, double):
- Single Precision (32-bit representation)
- Double Precision (64-bit representation)
- Extended (Quad) Precision (128-bit representation)

Character (char):
- ASCII 8-bit code (7-bit code with 1-bit padding)

Issues for Binary Representation of Numbers

- There are many ways to represent numbers in binary
  - Binary representations are encodings → many encodings possible
  - What are the issues that we must address?
- Issue #1: Complexity of arithmetic operations
- Issue #2: Negative numbers
- Issue #3: Maximum representable number
- Choose representation that makes these issues easy for machine, even if it's not easy for humans (i.e., ECE/CS 250 students)
  - Why? Machine has to do all the work!

Let's start with integers first …

One (Bad) Option for Integers: Sign Magnitude

- Use leftmost bit for +(0) or –(1):
- 6-bit example (1 sign bit and 5 magnitude bits):
- $+17_{10} = 010001_2$
- $-17_{10} = 110001_2$
- Pros:
  - Conceptually simple
  - Easy to convert positive ←→ negative
- Cons:
  - Harder to compute (add, subtract, etc.) with
  - Positive and negative 0: 000000 and 100000

Another (Bad) Option: 1's Complement

- Use largest positive binary numbers to represent negative numbers
- To negate a number,

  invert ("not") each bit:

  0 → 1

  1 → 0
- $(-x) = 2^n - x - 1$
- Cons:
  - Still two 0s (yuck)
  - Still hard to compute with

0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	-7
1001	-6
1010	-5
1011	-4
1100	-3
1101	-2
1110	-1
1111	-0

Best Option: 2's Complement

- Use large positives to represent negatives
- $(-x) = 2^n - x$
- This is 1's complement + 1
- $(-x) = 2^n - 1 - x + \mathbf{1}$
- So, just invert bits and add 1

<u>6-bit examples:</u>

$010110_2 = 22_{10}$; $101010_2 = -22_{10}$

$1_{10} = 000001_2$; $-1_{10} = 111111_2$

$0_{10} = 000000_2$; $-0_{10} = 000000_2$ → good!

0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1

Pros and Cons of 2's Complement

- Advantages:
  - Only one representation for 0 (unlike 1's comp): 0 = 000000
  - Addition algorithm is much easier than with sign/magnitude
    - Independent of sign bits

- Disadvantage:
  - One more negative number than positive
  - Example: 6-bit 2's complement number
    100000_2 = -32_{10}; but 32_{10} could not be represented

All modern computers use 2's complement for integers

2's Complement Precision Extension

- Most computers today support 32-bit (int) or 64-bit integers
  - Specify 64-bit using gcc C compiler with long long
- To extend precision, use sign bit extension
  - Integer precision is number of bits used to represent a number

Examples

$14_{10} = 001110_2$ in 6-bit representation.

$14_{10} = 000000001110_2$ in 12-bit representation

$-14_{10} = 110010_2$ in 6-bit representation

$-14_{10} = 111111110010_2$ in 12-bit representation.

2's Complement Arithmetic : Addition

- Suppose we want to add two numbers:

```
    00011101
+   00101011
```

- How do we do this?

2's Complement Arithmetic: Addition

- Suppose we want to add two numbers:

```
   00011101          695
+  00101011        + 232
```

- How do we do this?
  - Let's revisit decimal addition
  - Think about the process as we do it

2's Complement Arithmetic: Addition

- Suppose we want to add two numbers:

```
   00011101          695
+  00101011        + 232
----------        -----
                      7
```

- First add one's digit 5+2 = 7

2's Complement Arithmetic: Addition

- Suppose we want to add two numbers:

```
                    1
    00011101      695
+   00101011    + 232
-------------   -----
                  927
```

- First add one's digit 5+2 = 7
- Next add ten's digit 9+3 = 12 (2 carry a 1)

2's Complement Arithmetic: Addition

- Suppose we want to add two numbers:

```
   00011101          695
+  00101011        + 232
----------        -----
                     927
```

- First add one's digit 5+2 = 7
- Next add ten's digit 9+3 = 12 (2 carry a 1)
- Last add hundred's digit 1+6+2 = 9

2's Complement Arithmetic: Addition

- Suppose we want to add two numbers:

```
    00011101
+   00101011
```

- Back to the binary:
- First add 1's digit 1+1 = …?

2's Complement Arithmetic: Addition

- Suppose we want to add two numbers:

```
          1
    00011101
+   00101011
-------------
          0
```

- Back to the binary:
- First add 1's digit 1+1 = 2 (0 carry a 1)

2's Complement Arithmetic: Addition

- Suppose we want to add two numbers:

```
          11
    00011101
+   00101011
-------------
          00
```

- Back to the binary:
- First add 1's digit 1+1 = 2 (0 carry a 1)
- Then 2's digit: 1+0+1 =2 (0 carry a 1)
- You all finish it out….

2's Complement Arithmetic: Addition

- Suppose we want to add two numbers:

```
     111111
    00011101    = 29
+   00101011    = 43
-------------------
    01001000    = 72
```

- Can check our work in decimal

2's Complement Arithmetic: Addition

- What about this one:

```
    01011101
+   01101011
-------------
```

2's Complement Arithmetic: Addition

- What about this one:

```
    1111111
    01011101    =  93
+   01101011    = 107
-------------------------
    11001000    = -56
```

- But… that can't be right?
  - What do you expect for the answer?
  - What is it in 8-bit signed 2's complement?

Integer Overflow

- Answer should be 200
  - Not representable in 8-bit 2s complement
  - No right answer
- Call Integer Overflow
- Real problem in real programs

2's Complement Arithmetic: Subtraction

- 2's complement makes subtraction easy:
  - Remember: A - B = A + (-B)
  - And: -B = ~B + 1

    ↑ that ~ means flip bits ("not")
  - So we just flip the bits and start with carry-in (CI) = 1
  - Later: No new circuits to subtract (re-use adder hardware!)

```
                                  1
   0110101        ->        0110101
-  1010010                + 0101101
```

What About Non-integer Numbers?

- There are infinitely many real numbers between two integers
- Many important numbers are real
  - Speed of light ~= $3x10^8$
  - π = 3.1415…
- Fixed number of bits limits range of integers
  - Can't represent some important numbers
- Humans use Scientific Notation
  - $1.3x10^4$

Borrowing from Scientific Notation

- Think about scientific notation for a second:
- For example:

  $6.82 * 10^{23}$
- Real number, but comprised of ints:
  - 6 generally only 1 digit here
  - 82 any number here
  - 10 always 10 (base we work in)
  - 23 can be positive or negative
- Can we do something like this in binary?

Floating Point

- How about:
- +/- X.YYYYYY * $2^{+/-N}$

- Big numbers: large positive N
- Small numbers (<1): negative N
- Numbers near 0: small N

- This is "floating point" : most common way

IEEE single precision floating point

- Specific format called IEEE single precision:
- +/- 1.YYYYY * $2^{(N-127)}$
- "float" in Java, C, C++,…

- Assume X is always 1 (saves us a bit)
- 1 sign bit (+ = 0, 1 = -)
- 8 bit biased exponent (do N-127) in UNSIGNED format
- Implicit 1 before *binary point*
- 23-bit *mantissa* (YYYYY) in UNSIGNED format

Binary fractions

- 1.YYYY has a binary point
  - Like a decimal point but in binary
  - After a decimal point, you have
    - Tenths
    - Hundredths
    - Thousandths
    - ….
- So after a binary point you have…

Binary fractions

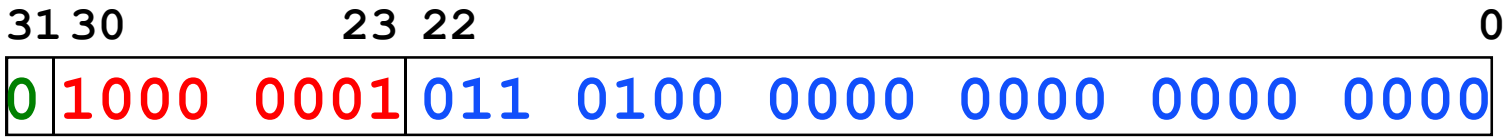
- 1.YYYY has a binary point
  - Like a decimal point but in binary
  - After a decimal point, you have
    - Tenths
    - Hundredths
    - Thousandths
    - ….
- So after a binary point you have…
  - Halves
  - Quarters
  - Eighths
  - ….

Floating point example

- Binary fraction example:

  $101.101_2 = 4 + 1 + ½ + {}^1/_8 = 5.625$
- For floating point, needs normalization:

  $1.01101_2 * 2^2$
- Sign is +, which = 0
- $\text{Exponent}_{10} = 127 + 2 = 129 = 1000\ 0001$
- Mantissa = 1.011 0100 0000 0000 0000 0000



Floating Point Representation

Example:

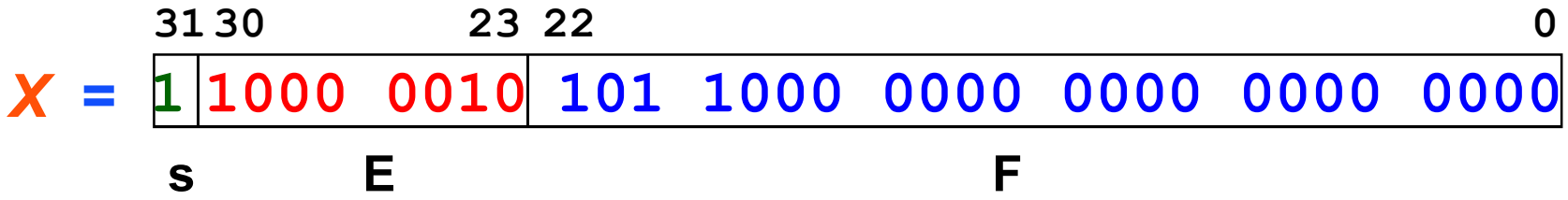
What floating-point number is:

0xC1580000?

Answer

What floating-point number is 0xC1580000?

1100 0001 0101 1000 0000 0000 0000 0000



Sign = 1 = negative

Exponent = (128+2)-127 = 3
Mantissa = 1.1011

$-1.1011 \times 2^3 = -1101.1 = -13.5$

Trick question

- How do you represent 0.0?
  - Why is this a trick question?

Trick question

- How do you represent 0.0?
  - Why is this a trick question?
  - 0.0 = 000000000
  - But need 1.XXXXX representation?

Trick question

- How do you represent 0.0?
  - Why is this a trick question?
  - 0.0 = 000000000
  - But need 1.XXXXX representation?
- Exponent of 0 is denormalized
  - Implicit 0. instead of 1. in mantissa
  - Allows 0000….0000 to be 0
  - Helps with very small numbers near 0
- Results in +/- 0 in FP (but they are "equal")

Other Weird FP numbers

- Exponent = 1111 1111 also not standard
  - All 0 mantissa: +/- ∞
  - Non zero mantissa: Not a Number (NaN)

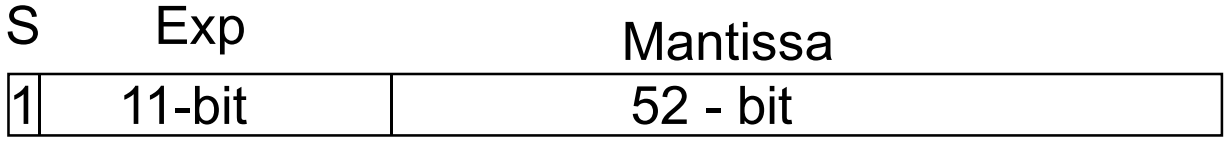
    sqrt(-42) = NaN

Floating Point Representation

- Double Precision Floating point:

  64-bit representation:
  - 1-bit sign
  - 11-bit (biased) exponent
  - 52-bit fraction (with implicit 1).

- "double" in Java, C, C++, …



What About Strings?

- Many important things stored as strings…
  - E.g., your name
- How should we store strings?
  - As a string of characters?
- How do we represent characters?

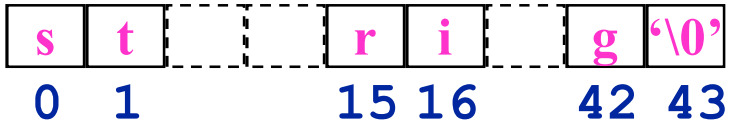
ASCII Character Representation

Oct. Char

000	nul	001	soh	002	stx	003	etx	004	eot	005	enq	006	ack	007	bel
010	bs	011	ht	012	nl	013	vt	014	np	015	cr	016	so	017	si
020	dle	021	dc1	022	dc2	023	dc3	024	dc4	025	nak	026	syn	027	etb
030	can	031	em	032	sub	033	esc	034	fs	035	gs	036	rs	037	us
040	sp	041	!	042	"	043	#	044	$	045	%	046	&	047	'
050	(	051	)	052	*	053	+	054	,	055	-	056	.	057	/
060	0	061	1	062	2	063	3	064	4	065	5	066	6	067	7
070	8	071	9	072	:	073	;	074	<	075	=	076	>	077	?
100	@	101	A	102	B	103	C	104	D	105	E	106	F	107	G
110	H	111	I	112	J	113	K	114	L	115	M	116	N	117	O
120	P	121	Q	122	R	123	S	124	T	125	U	126	V	127	W
130	X	131	Y	132	Z	133	[	134	\	135	]	136	^	137	_
140	`	141	a	142	b	143	c	144	d	145	e	146	f	147	g
150	h	151	i	250	j	153	k	154	l	155	m	156	n	157	o
160	p	161	q	162	r	163	s	164	t	165	u	166	v	167	w
170	x	171	y	172	z	173	{	174	|	175	}	176	~	177	del

- Each character represented by 7-bit ASCII code.
  - Packed into 8-bits

Strings as Arrays



- A string is an array of characters with '\0' at the end
- Each element is one byte in ASCII code
- '\0' is null (ASCII code 0)

String Length Function: strlen()

- `strlen()` returns the number of characters in a string
  - same as number elements in char array?

```
int strlen(char* s)
// s points to first char in string that ends in NULL char
 {
     int count=0;
     while (*s != NULL){
            count++;
            s = s + 1;
      }
     return count;
 }
```

Summary: From C to Binary

- Everything must be represented in binary!
  - Instructions
  - Variables
    - int, float, double, char, etc.

- Next: Instruction Sets & Assembly Programming