CS2100 COMPUTER ORGANISATION

http://www.comp.nus.edu.sg/~cs2100/

Lecture #3

Data Representation and Number Systems

NUS National University of Singapore | School of Computing

Lecture #3: Data Representation and Number Systems (1/2)

1. Data Representation
2. Decimal (base 10) Number System
3. Other Number Systems
4. Base-*R* to Decimal Conversion
5. Decimal to Binary Conversion
   - 5.1 Repeated Division-by-2
   - 5.2 Repeated Multiplication-by-2
6. Conversion Between Decimal and Other Bases
7. Conversion Between Bases
8. Binary to Octal/Hexadecimal Conversion

Lecture #3: Data Representation and Number Systems (2/2)

1. Data Representation (1/2)

Basic data types in C:

int float double char

Variants: short, long

How data is represented depends on its type:

01000110

As an ‘int’, it is 70

As a ‘char’, it is ‘F’

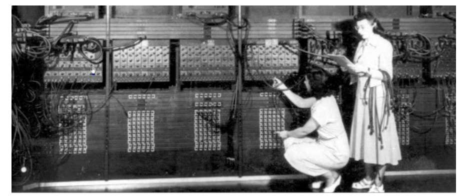
11000000110100000000000000000000

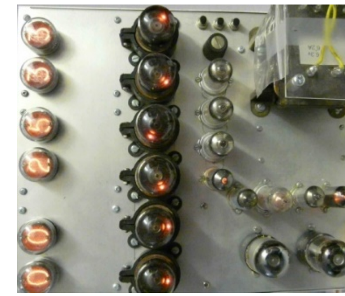
As an ‘int’, it is -1060110336

As an ‘float’, it is -6.5

1. Data Representation (2/2)

- Data are internally represented as sequence of bits (***b***inary dig***it***s). A bit is either 0 or 1.





- Other units
  - Byte: 8 bits
  - Nibble: 4 bits (rarely used now)
  - Word: Multiple of bytes (eg: 1 byte, 2 bytes, 4 bytes, etc.) depending on the computer architecture
- N bits can represent up to 2^N values
  - Eg: 2 bits represent up to 4 values (00, 01, 10, 11);
    4 bits represent up to 16 values (0000, 0001, 0010, …., 1111)
- To represent M values, $\lceil \log_2 M \rceil$ bits required
  - Eg: 32 values require 5 bits; 1000 values require 10 bits

2. Decimal (base 10) Number System

- A weighted-positional number system.
- Base (also called radix) is 10
- Symbols/digits = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }
- Each position has a weight of power of 10
  - Eg: $(7594.36)_{10} = (7 \times 10^3) + (5 \times 10^2) + (9 \times 10^1) + (4 \times 10^0) + (3 \times 10^{-1}) + (6 \times 10^{-2})$

$$
\begin{aligned}
(a_n a_{n-1} \ldots a_0 \,.\, f_1 f_2 \ldots f_m)_{10} = \; & (a_n \times 10^n) + (a_{n-1} \times 10^{n-1}) + \ldots + (a_0 \times 10^0) + \\
& (f_1 \times 10^{-1}) + (f_2 \times 10^{-2}) + \ldots + (f_m \times 10^{-m})
\end{aligned}
$$

3. Other Number Systems (1/2)

- Binary (base 2)
  - Weights in powers of 2
  - Binary digits (bits): **0, 1**
- Octal (base 8)
  - Weights in powers of 8
  - Octal digits: **0, 1, 2, 3, 4, 5, 6, 7**.
- Hexadecimal (base 16)
  - Weights in powers of 16
  - Hexadecimal digits: **0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F**.
- Base/radix *R*:
  - Weights in powers of *R*

3. Other Number Systems (2/2)

- In some programming languages/software, special notations are used to represent numbers in certain bases
  - In programming language C
    - Prefix 0 for octal. Eg: 032 represents the octal number $(32)_8$
    - Prefix 0x for hexadecimal. Eg: 0x32 represents the hexadecimal number $(32)_{16}$
  - In QTSpim (a MIPS simulator you will use)
    - Prefix 0x for hexadecimal. Eg: 0x100 represents the hexadecimal number $(100)_{16}$
  - In Verilog, the following values are the same
    - 8'b11110000: an 8-bit binary value 11110000
    - 8'hF0: an 8-bit binary value represented in hexadecimal F0
    - 8'd240: an 8-bit binary value represented in decimal 240

4. Base-*R* to Decimal Conversion

- Easy!
  - $1101.101_2 = 1\times2^3 + 1\times2^2 + 1\times2^0 + 1\times2^{-1} + 1\times2^{-3}$
    $= 8 + 4 + 1 + 0.5 + 0.125 = \mathbf{13.625_{10}}$
  - $572.6_8 = 5\times8^2 + 7\times8^1 + 2\times8^0 + 6\times8^{-1}$
    $= 320 + 56 + 2 + 0.75 = \mathbf{378.75_{10}}$
  - $2A.8_{16} = 2\times16^1 + 10\times16^0 + 8\times16^{-1}$
    $= 32 + 10 + 0.5 = \mathbf{42.5_{10}}$
  - $341.24_5 = 3\times5^2 + 4\times5^1 + 1\times5^0 + 2\times5^{-1} + 4\times5^{-2}$
    $= 75 + 20 + 1 + 0.4 + 0.16 = \mathbf{96.56_{10}}$
- DLD page 42 Quick Review Questions
  Questions 2-1 to 2-4.

5. Decimal to Binary Conversion

- For whole numbers
  - Repeated Division-by-2 Method
- For fractions
  - Repeated Multiplication-by-2 Method

5.1 Repeated Divison-by-2

- To convert a whole number to binary, use successive division by 2 until the quotient is 0. The remainders form the answer, with the first remainder as the *least significant bit (LSB)* and the last as the *most significant bit (MSB)*.

$(43)_{10} = (\ 101011\ )_2$

2	43		
2	21	rem 1	← LSB
2	10	rem 1	
2	5	rem 0	
2	2	rem 1	
2	1	rem 0	
	0	rem 1	← MSB

5.2 Repeated Multiplication-by-2

- To convert decimal fractions to binary, repeated multiplication by 2 is used, until the fractional product is 0 (or until the desired number of decimal places). The carried digits, or *carries*, produce the answer, with the first carry as the MSB, and the last as the LSB.

$(0.3125)_{10} = (.0101)_2$

	Carry	
$0.3125 \times 2 = 0.625$	0	←MSB
$0.625 \times 2 = 1.25$	1	
$0.25 \times 2 = 0.50$	0	
$0.5 \times 2 = 1.00$	1	←LSB

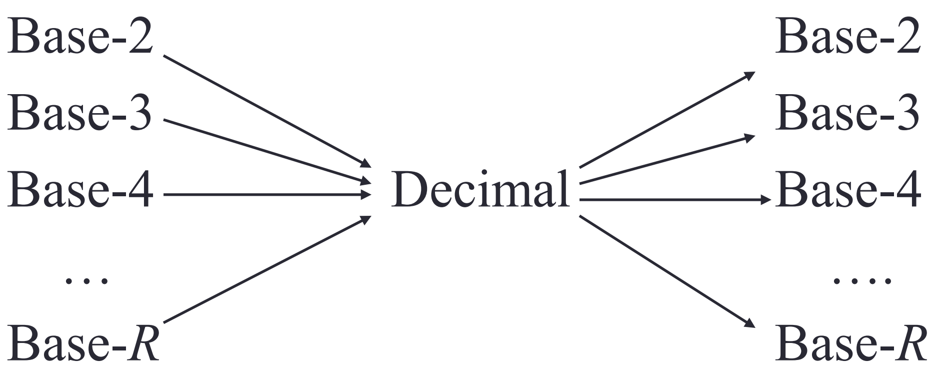
6. Conversion Between Decimal and Other Bases

- Base-*R* to decimal: multiply digits with their corresponding weights
- Decimal to binary (base 2)
  - Whole numbers: repeated division-by-2
  - Fractions: repeated multiplication-by-2
- Decimal to base-*R*
  - Whole numbers: repeated division-by-*R*
  - Fractions: repeated multiplication-by-*R*

- DLD page 42 Quick Review Questions
  Questions 2-5 to 2-8.

7. Conversion Between Bases

- In general, conversion between bases can be done via decimal:



- Shortcuts for conversion between bases 2, 4, 8, 16 (see next slide)

8. Binary to Octal/Hexadecimal Conversion

- Binary → Octal: partition in groups of 3
  - $(10\ 111\ 011\ 001\ .\ 101\ 110)_2$ = $\mathbf{(2731.56)_8}$
- Octal → Binary: reverse
  - $(2731.56)_8$ = $\mathbf{(10\ 111\ 011\ 001\ .\ 101\ 110)_2}$
- Binary → Hexadecimal: partition in groups of 4
  - $(101\ 1101\ 1001\ .\ 1011\ 1000)_2$ = $\mathbf{(5D9.B8)_{16}}$
- Hexadecimal → Binary: reverse
  - $(5D9.B8)_{16}$ = $\mathbf{(101\ 1101\ 1001\ .\ 1011\ 1000)_2}$
- DLD page 42 Quick Review Questions Questions 2-9 to 2-10.

9. ASCII Code (1/3)

- ASCII code and Unicode are used to represent characters ('a', 'C', '?', '\0', etc.)
- ASCII
  - American Standard Code for Information Interchange
  - 7 bits, plus 1 parity bit (odd or even parity)

Character	ASCII Code
0	0110000
1	0110001
. . .	. . .
9	0111001
:	0111010
A	1000001
B	1000010
. . .	. . .
Z	1011010
[	1011011
\	1011100

9. ASCII Code (2/3)

- ASCII table

‘A’: 1000001 (or 65_{10})

	MSBs							
LSBs	**000**	**001**	**010**	**011**	**100**	**101**	**110**	**111**
0000	NUL	DLE	SP	0	@	P	`	p
0001	SOH	DC_1	!	1	A	Q	a	q
0010	STX	DC_2	“	2	B	R	b	r
0011	ETX	DC_3	#	3	C	S	c	s
0100	EOT	DC_4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	‘	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	O	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

9. ASCII Code (3/3)

(Slide 4)

01000110

As an ‘int’, it is 70

As a ‘char’, it is ‘F’

- Integers (0 to 127) and characters are ‘somewhat’ interchangeable in C

CharAndInt.c

```
int num = 65;
char ch = 'F';

printf("num (in %%d) = %d\n", num);
printf("num (in %%c) = %c\n", num);
printf("\n");

printf("ch (in %%c) = %c\n", ch);
printf("ch (in %%d) = %d\n", ch);
```

```
num (in %d) = 65
num (in %c) = A

ch (in %c) = F
ch (in %d) = 70
```

Past-Year's Exam Question!

PastYearQn.c

```
int i, n = 2147483640;
for (i=1; i<=10; i++) {
    n = n + 1;
}
printf("n = %d\n", n);
```

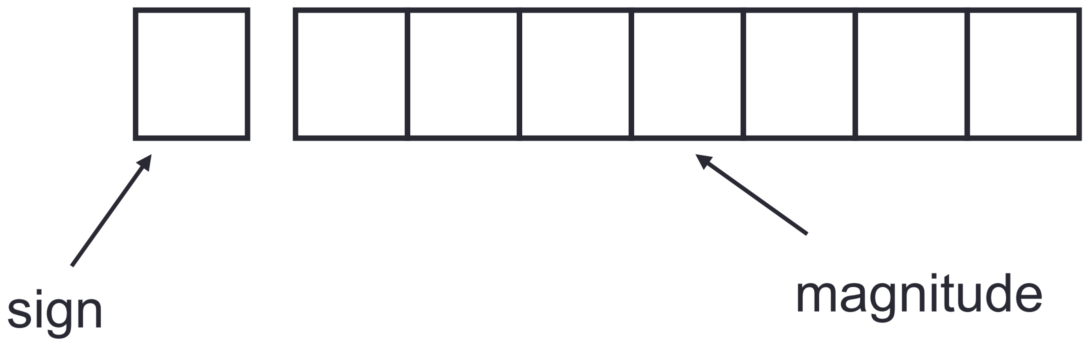
- What is the output of the above code when run on sunfire?
- Is it 2147483650?

10. Negative Numbers

- Unsigned numbers: only non-negative values
- Signed numbers: include all values (positive and negative)
- There are 3 common representations for signed binary numbers:
  - Sign-and-Magnitude
  - 1s Complement
  - 2s Complement

10.1 Sign-and-Magnitude (1/3)

- The sign is represented by a 'sign bit'
  - 0 for +
  - 1 for -
- Eg: a 1-bit sign and 7-bit magnitude format.



- 00110100 ➔ $+110100_2 = +52_{10}$
- 10010011 ➔ $-10011_2 = -19_{10}$

10.1 Sign-and-Magnitude (2/3)

- Largest value: 01111111 = $+127_{10}$
- Smallest value: 11111111 = -127_{10}
- Zeros: 00000000 = $+0_{10}$
  10000000 = -0_{10}
- Range (for 8-bit): -127_{10} to $+127_{10}$
- Question:
  - For an *n*-bit sign-and-magnitude representation, what is the range of values that can be represented?

10.1 Sign-and-Magnitude (3/3)

- To negate a number, just <u>invert the sign bit</u>.
- Examples:
  - How to negate 00100001_{sm} (decimal 33)?
    Answer: 10100001_{sm} (decimal -33)
  - How to negate 10000101_{sm} (decimal -5)?
    Answer: 00000101_{sm} (decimal +5)

10.2 1s Complement (1/3)

- Given a number ***x*** which can be expressed as an *n*-bit binary number, its negated value can be obtained in **1s-complement** representation using:

$$\mathbf{-x = 2^n - x - 1}$$

- Example: With an 8-bit number 00001100 (or 12_{10}), its negated value expressed in 1s-complement is:

$$
\begin{aligned}
-00001100_2 &= 2^8 - 12 - 1 \text{ (calculation done in decimal)} \\
&= 243 \\
&= 11110011_{1s}
\end{aligned}
$$

(This means that -12_{10} is written as 11110011 in 1s-complement representation.)

10.2 1s Complement (2/3)

- Technique to negate a value: invert all the bits.
- Largest value: 01111111 = $+127_{10}$
- Smallest value: 10000000 = -127_{10}
- Zeros: 00000000 = $+0_{10}$
  11111111 = -0_{10}
- Range (for 8 bits): -127_{10} to $+127_{10}$
- Range (for *n* bits): $-(2^{n-1} - 1)$ to $2^{n-1} - 1$
- The most significant bit (MSB) still represents the sign: 0 for positive, 1 for negative.

10.2 1s Complement (3/3)

- Examples (assuming 8-bit):

$(14)_{10} = (00001110)_2 = (00001110)_{1s}$

$-(14)_{10} = -(00001110)_2 = (11110001)_{1s}$

$-(80)_{10} = -(\ ?\ )_2 = (\ ?\ )_{1s}$

10.3 2s Complement (1/3)

- Given a number ***x*** which can be expressed as an *n*-bit binary number, its negated value can be obtained in **2s-complement** representation using:

$$-x = 2^n - x$$

- Example: With an 8-bit number 00001100 (or 12_{10}), its negated value expressed in 2s-complement is:

$$
\begin{aligned}
-00001100_2 &= 2^8 - 12 \text{ (calculation done in decimal)} \\
&= 244 \\
&= 11110100_{2s}
\end{aligned}
$$

(This means that -12_{10} is written as 11110100 in 2s-complement representation.)

10.3 2s Complement (2/3)

- Technique to negate a value: invert all the bits, then add 1.
- Largest value: 01111111 = $+127_{10}$
- Smallest value: 10000000 = -128_{10}
- Zero: 00000000 = $+0_{10}$
- Range (for 8 bits): -128_{10} to $+127_{10}$
- Range (for n bits): -2^{n-1} to $2^{n-1} - 1$
- The most significant bit (MSB) still represents the sign: 0 for positive, 1 for negative.

10.3 2s Complement (3/3)

- Examples (assuming 8-bit):

$$(14)_{10} = (00001110)_2 = (00001110)_{2s}$$

$$-(14)_{10} = -(00001110)_2 = (11110010)_{2s}$$

$$-(80)_{10} = -(\ ?\ )_2 = (\ ?\ )_{2s}$$

Compare with slide 26.

- 1s complement:

$$(14)_{10} = (00001110)_2 = (00001110)_{1s}$$

$$-(14)_{10} = -(00001110)_2 = (11110001)_{1s}$$

10.4 Comparisons

Important!

4-bit system

Positive values

Value	Sign-and-Magnitude	1s Comp.	2s Comp.
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
+0	0000	0000	0000

Negative values

Value	Sign-and-Magnitude	1s Comp.	2s Comp.
-0	1000	1111	-
-1	1001	1110	1111
-2	1010	1101	1110
-3	1011	1100	1101
-4	1100	1011	1100
-5	1101	1010	1011
-6	1110	1001	1010
-7	1111	1000	1001
-8	-	-	1000

Past-Year's Exam Question! (Answer)

PastYearQn.c

```
int i, n = 2147483640;
for (i=1; i<=10; i++) {
    n = n + 1;
}
printf("n = %d\n", n);
```

- What is the output of the above code when run on sunfire?
- Is it 2147483650? ✘

- int type in sunfire takes up 4 bytes (32 bits) and uses 2s complement
- Largest positive integer = $2^{31} - 1 = 2147483647$

1st iteration: n = 2147483641

7th iteration: n = 2147483647

```
  01111 ……. 1111111111
                   + 1
  10000…….0000000000
```

8th iteration: n = -2147483648

9th iteration: n = -2147483647

10th iteration: n = -2147483646

10.5 Complement on Fractions

- We can extend the idea of complement on fractions.
- Examples:
  - Negate 0101.01 in 1s-complement
    Answer: 1010.10
  - Negate 111000.101 in 1s-complement
    Answer: 000111.010
  - Negate 0101.01 in 2s-complement
    Answer: 1010.11

10.6 2s Complement on Addition/Subtraction (1/4)

- Algorithm for addition of integers, A + B:
  1. Perform binary addition on the two numbers.
  2. Ignore the carry out of the MSB.
  3. Check for overflow. Overflow occurs if the ‘carry in’ and ‘carry out’ of the MSB are different, or if result is opposite sign of A and B.

- Algorithm for subtraction of integers, A – B:

  A – B = A + (-B)

  1. Take 2s-complement of B.
  2. Add the 2s-complement of B to A.

10.6 Overflow (2/4)

- Signed numbers are of a fixed range.
- If the result of addition/subtraction goes beyond this range, an **overflow** occurs.
- Overflow can be easily detected:
  - *positive* add *positive* → *negative*
  - *negative* add *negative* → *positive*
- Example: 4-bit 2s-complement system
  - Range of value: -8_{10} to 7_{10}
  - $0101_{2s} + 0110_{2s} = 1011_{2s}$
    $5_{10} + 6_{10} = -5_{10}$?! (overflow!)
  - $1001_{2s} + 1101_{2s} = \underline{1}0110_{2s}$ (discard end-carry) $= 0110_{2s}$
    $-7_{10} + -3_{10} = 6_{10}$?! (overflow!)

10.6 2s Complement Addition (3/4)

- Examples: 4-bit system

```
   +3          0011
 + +4        + 0100
 ----        -------
   +7          0111
 ----        -------
```

No overflow

```
   -2          1110
 + -6        + 1010
 ----        -------
   -8         11000
 ----        -------
```

No overflow

```
   +6          0110
 + -3        + 1101
 ----        -------
   +3         10011
 ----        -------
```

No overflow

```
   +4          0100
 + -7        + 1001
 ----        -------
   -3          1101
 ----        -------
```

No overflow

```
   -3          1101
 + -6        + 1010
 ----        -------
   -9         10111
 ----        -------
```

Overflow!

```
   +5          0101
 + +6        + 0110
 ----        -------
  +11          1011
 ----        -------
```

Overflow!

- Which of the above is/are overflow(s)?

10.6 2s Complement Subtraction (4/4)

- Examples: 4-bit system
  - 4 – 7
  - Convert it to 4 + (-7)

```
  +4         0100
+ -7       + 1001
----       -------
  -3         1101
----       -------
```

No overflow

  - 6 – 1
  - Convert it to 6 + (-1)

```
  +6         0110
+ -1       + 1111
----       -------
  +5        10101
----       -------
```

No overflow

  - -5 – 4
  - Convert it to -5 + (-4)

```
  -5         1011
+ -4       + 1100
----       -------
  -9        10111
----       -------
```

Overflow!

- Which of the above is/are overflow(s)?

10.7 1s Complement on Addition/Subtraction (1/2)

- Algorithm for addition of integers, A + B:
  1. Perform binary addition on the two numbers.
  2. If there is a carry out of the MSB, add 1 to the result.
  3. Check for overflow. Overflow occurs if result is opposite sign of A and B.

- Algorithm for subtraction of integers, A – B:

  A – B = A + (-B)

  1. Take 1s-complement of B.
  2. Add the 1s-complement of B to A.

10.7 1s Complement Addition (2/2)

- Examples: 4-bit system

```
  +3         0011
+ +4       + 0100
----       -------
  +7         0111
----       -------
```

No overflow

```
  +5         0101
+ -5       + 1010
----       -------
  -0         1111
----       -------
```

No overflow

```
  -2         1101
+ -5       + 1010
----       -------
  -7        10111
----       +    1
           -------
             1000
           -------
```

No overflow

```
  -3         1100
+ -7       + 1000
----       -------
 -10        10100
----       +    1
           -------
             0101
           -------
```

Overflow!

Any overflow?

DLD page 42 – 43 Quick Review Questions
Questions 2-13 to 2-18.

10.8 Excess Representation (1/2)

- Besides sign-and-magnitude and complement schemes, the **excess representation** is another scheme.
- It allows the range of values to be distributed <u>evenly</u> between the positive and negative values, by a simple translation (addition/subtraction).
- Example: Excess-4 representation on 3-bit numbers. See table on the right.

Excess-4 Representation	*Value*
000	-4
001	-3
010	-2
011	-1
100	0
101	1
110	2
111	3

- Questions: What if we use Excess-2 on 3-bit numbers? Or Excess-7?

10.8 Excess Representation (2/2)

- Example: For 4-bit numbers, we may use excess-7 or excess-8. Excess-8 is shown below.

Excess-8 Representation	*Value*
0000	-8
0001	-7
0010	-6
0011	-5
0100	-4
0101	-3
0110	-2
0111	-1

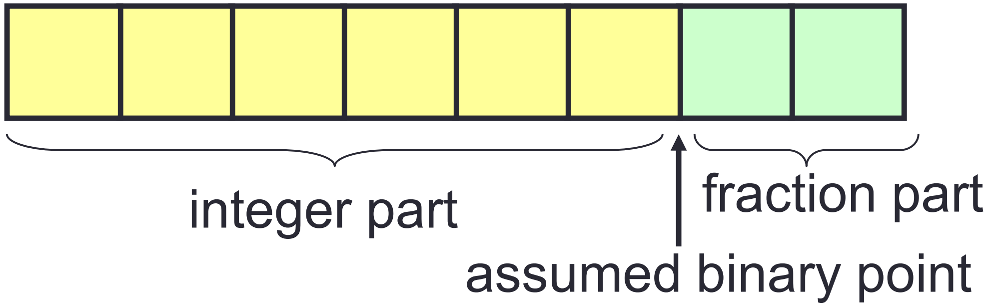
Excess-8 Representation	*Value*
1000	0
1001	1
1010	2
1011	3
1100	4
1101	5
1110	6
1111	7

11. Real Numbers

- Many applications involve computations not only on integers but also on real numbers.
- How are real numbers represented in a computer system?
- Due to the finite number of bits, real number are often represented in their approximate values.

11.1 Fixed-Point Representation

- In fixed-point representation, the number of bits allocated for the whole number part and fractional part are fixed.
- For example, given an 8-bit representation, 6 bits are for whole number part and 2 bits for fractional parts.



- If 2s complement is used, we can represent values like:

$$011010.11_{2s} = 26.75_{10}$$

$$111110.11_{2s} = -000001.01_{2} = -1.25_{10}$$

11.2 Floating-Point Representation (1/4)

- Fixed-point representation has limited range.
- Alternative: Floating point numbers allow us to represent very large or very small numbers.
- Examples:

  0.23×10^{23} (very large positive number)

  0.5×10^{-37} (very small positive number)

  -0.2397×10^{-18} (very small negative number)

11.2 IEEE 754 Floating-Point Rep. (2/4)

- 3 components: **sign**, **exponent** and **mantissa (fraction)**

sign	exponent	mantissa

- The base (radix) is assumed to be 2.
- Two formats:
  - Single-precision (32 bits): 1-bit sign, 8-bit exponent with bias 127 (excess-127), 23-bit mantissa
  - Double-precision (64 bits): 1-bit sign, 11-bit exponent with bias 1023 (excess-1023), and 52-bit mantissa
- We will focus on the single-precision format
- Reading
  - DLD pages 32 - 33
  - IEEE standard 754 floating point numbers: http://steve.hollasch.net/cgindex/coding/ieeefloat.html

11.2 IEEE 754 Floating-Point Rep. (3/4)

- 3 components: **sign**, **exponent** and **mantissa (fraction)**

sign	exponent	mantissa

- Sign bit: 0 for positive, 1 for negative.
- Mantissa is **normalised** with an implicit leading bit 1
  - 110.1_2 → normalised → $1.101_2 \times 2^2$ → only **101** is stored in the mantissa field
  - 0.00101101_2 → normalised → $1.01101_2 \times 2^{-3}$ → only **01101** is stored in the mantissa field

11.2 IEEE 754 Floating-Point Rep. (4/4)

- Example: How is -6.5_{10} represented in IEEE 754 single-precision floating-point format?

$$-6.5_{10} = -110.1_2 = -1.\boxed{101}_2 \times 2^{2}$$

Exponent = 2 + 127 = 129 = 10000001_2

sign	exponent (excess-127)	mantissa
1	10000001	10100000000000000000000

- We may write the 32-bit representation in hexadecimal:

$1\ 10000001\ 10100000000000000000000_2 = \text{C0D00000}_{16}$

(Slide 4) 11000000110100000000000000000000

As an ‘int’, it is -1060110336

As an ‘float’, it is -6.5

End of File