Extra Lecture: Number Systems

- Objectives - To understand:
  - Base of number systems: decimal, binary, octal and hexadecimal
  - Textual information stored as ASCII
  - Binary addition/subtraction, multiplication
  - Binary logical operations
  - Unsigned and signed binary number systems
  - Fixed point binary representations
  - Floating point representations
- By the end of the lecture, you should be able to:
  - Convert between numbers represented in different bases
  - Convert between fixed point and floating point numbers
  - Perform simple binary arithmetic and logical operations
  - Read and interpret hexadecimal numbers with reasonable speed

Decimal number system

- We are familiar with decimal number representation. For example:

Hundreds	*Tens*	*Ones*		*Tenths*	*Hundredths*
10^2	10^1	10^0		10^{-1}	10^{-2}
4	**6**	**2**	**.**	**1**	**5**

- The value of this number is calculated as:

$$
\begin{aligned}
4*10^2 &= 4*100 &&= 400.\\
6*10^1 &= 6*10 &&= 60.\\
2*10^0 &= 2*1 &&= 2.\\
1*10^{-1} &= 1*.1 &&= 0.1\\
5*10^{-2} &= 5*.01 &&= \underline{+\ 0.05}\\
& & & 462.15
\end{aligned}
$$

- In general, the relationship between the contribution of a digit, its position, and the base of the system is given by:

$$DIGIT * BASE^{POSITION\ \#}$$

- Usually, we restrict $0 \le DIGIT \le BASE - 1$

The bases of a number system

- There is no reason why one should be restricted to using base-10 (decimal) numbers only.
- Digital computers use a binary number system where the base (or radix) is 2:

$$DIGIT * 2^{POSITION\ \#}$$

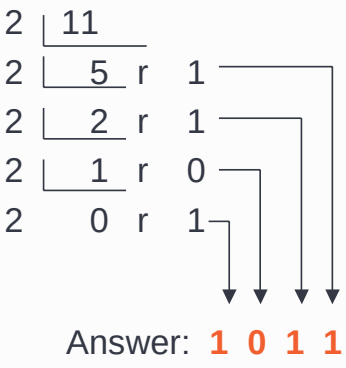
Fours	*Twos*	*Ones*		*Halves*	*Fourths*
2^2	2^1	2^0		2^{-1}	2^{-2}
1	**1**	**0**	.	**1**	**1**

- For example, the value of this binary number is:

$$\begin{aligned} 1*2^2 &= 1*4 &&= 4. \\ 1*2^1 &= 1*2 &&= 2. \\ 0*2^0 &= 0*1 &&= 0. \\ 1*2^{-1} &= 1*.5 &&= 0.5 \\ 1*2^{-2} &= 1*.25 &&= +\ 0.25 \\ & & & \quad 6.75 \end{aligned}$$

Converting decimal integers to binary

- Repeatedly divide the decimal number by 2 (the base of the binary system).
- Division by 2 will either give a remainder of 1 or 0.
- Collecting the remainders (LSB first!) gives the binary answer.
- Convert 11_{10} into binary



Answer: **1 0 1 1**

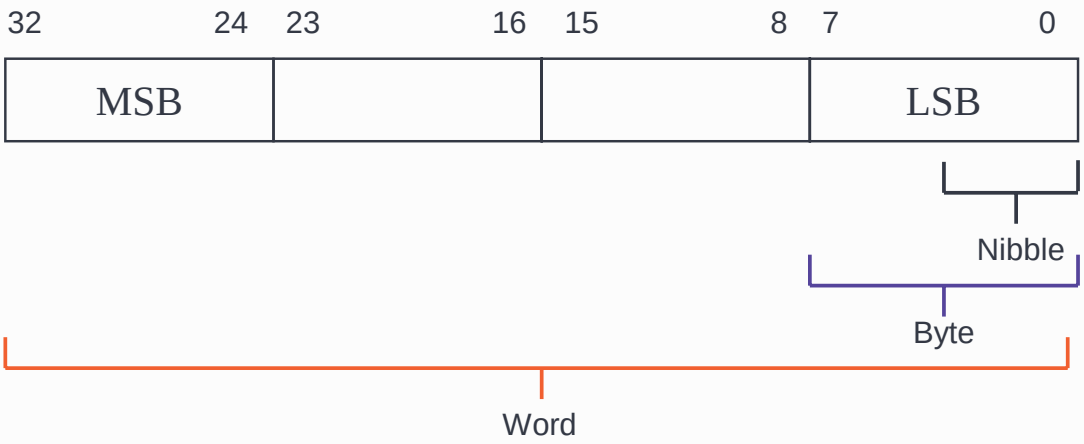
Octal and hexadecimal number systems

Binary	Octal	Decimal	Hexadecimal
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	10	8	8
1001	11	9	9
1010	12	10	A
1011	13	11	B
1100	14	12	C
1101	15	13	D
1110	16	14	E
1111	17	15	F
Base-2	*Base-8*	*Base-10*	*Base-16*

run out of "normal" digit symbols

Nibbles, Bytes, Words

- Internal datapaths inside computers could be different width - for example 4-bit, 8-bit, 16-bit or 32-bit.
- For example: ARM processor uses 32-bit internal datapath
- WORD = 32-bit for ARM (byte and nibble are architecture independent)



Hexadecimal representation

- Convenient to divide any size of binary numbers into nibbles
- Represent each nibble as hexadecimal – think of the human!
- Example:

0100 1101 0110 1011 1000 0011 0000 1111

4 D 6 B 8 3 0 F

- This is possible because 16 is a power of 2
- Converting from decimal to hexadecimal is the same as converting to binary, except, divide by 16 instead of 2:

16	237		
16	14	r	13
	0	r	14

Answer: **E D** $_h$

Representing Text in ASCII

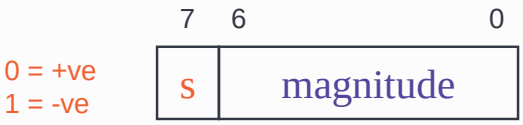
- Textual information must also be stored as binary numbers.
- Each character is represented as a 7-bit number known as ASCII codes (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange**)**
- For example, ‘A’ is represented by 41_h and ‘a’ by 61_h

b_3 - b_0

b_6 - b_4	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SPC	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Signed numbers

- So far, numbers are assumed to be unsigned (i.e. positive)
- How to represent signed numbers?
- Solution 1: **Sign-magnitude** - Use one bit to represent the **sign**, the remain bits to represent **magnitude**



0 = +ve
1 = -ve

+27 = 0001 1011$_b$
-27 = 1001 1011$_b$

  - Advantage: easy human reasoning – it's what we use
  - Problem: addition and subtraction require quite complex circuits

Two's complement

- Solution 2: **Two's complement** – represent a negative number *x* by the number 2^*N* + *x*, in an *n*-bit representation:
- Example: Encode –27 in 8 bit two's complement –

256 – 27 = 229
229_{10} = 1110 0101$_b$

- So long as we only want to represent numbers with magnitude less than 2^(N-1), the MSB is still the *sign bit*
- Example: Encode 27 in 8 bit two's complement –

27_{10} = 0001 1011$_b$

Two's complement

- Another view of two's complement is to represent a negative number by taking its magnitude, inverting all bits and adding one:

Positive number	**+27 = 0001 1011$_b$**
Invert all bits	**1110 0100$_b$**
Add 1	**-27 = 1110 0101$_b$**

- Why is this the same?
- Inverting all bits is the same as subtracting the number from 11….1:

"All ones"	**1111 1111$_b$**
Positive number	**+27 = 0001 1011$_b$**
Subtraction	**1110 0100$_b$**

- So inverting and then adding one is the same as subtracting from 11….1 + 1 = 100….0 = 2^N

"All ones"	**1111 1111$_b$**
Add 1	**0000 0001$_b$**

Two's complement

- A third (and final!) way to view two's complement is that the weight of position *i* is 2^*i except the MSB, which has negative weight*

$$x = -b_{N-1}2^{N-1} + b_{N-2}2^{N-2} + \bullet\bullet\bullet + b_1 2^1 + b_0 2^0$$

-27 = 1110 0101$_b$ = -128 + 64 + 32 + 4 + 1

- Why is this the same?
- If we interpreted this as an unsigned number, it would be

$$y = b_{N-1}2^{N-1} + b_{N-2}2^{N-2} + \bullet\bullet\bullet + b_1 2^1 + b_0 2^0$$

- If *x* is negative, sign bit b_{N-1} is 1, so difference is y – x = 2^N, i.e. $y = 2^N + x$

Why 2's complement representation?

- If we represent signed numbers in 2's complement form, subtraction is the same as addition to the (2's complemented) number (if we ignore any carry out)

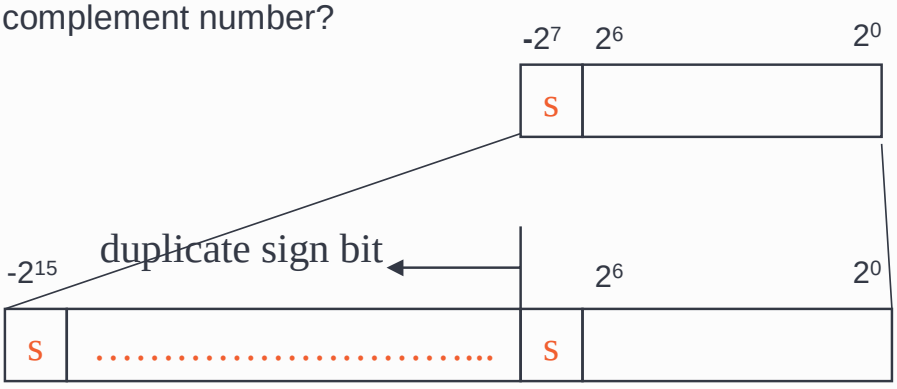
27	$0001\ 1011_b$
- 17	$0001\ 0001_b$
+ 10	$0000\ 1010_b$

+27	$0001\ 1011_b$
+ - 17	$1110\ 1111_b$
+10	$0000\ 1010_b$

- Note that the range for 8-bit unsigned and signed numbers are different.
  - 8-bit unsigned: **0 …… +255**
  - 8-bit 2's complement signed number: **-128 …… +127**

Sign Extension

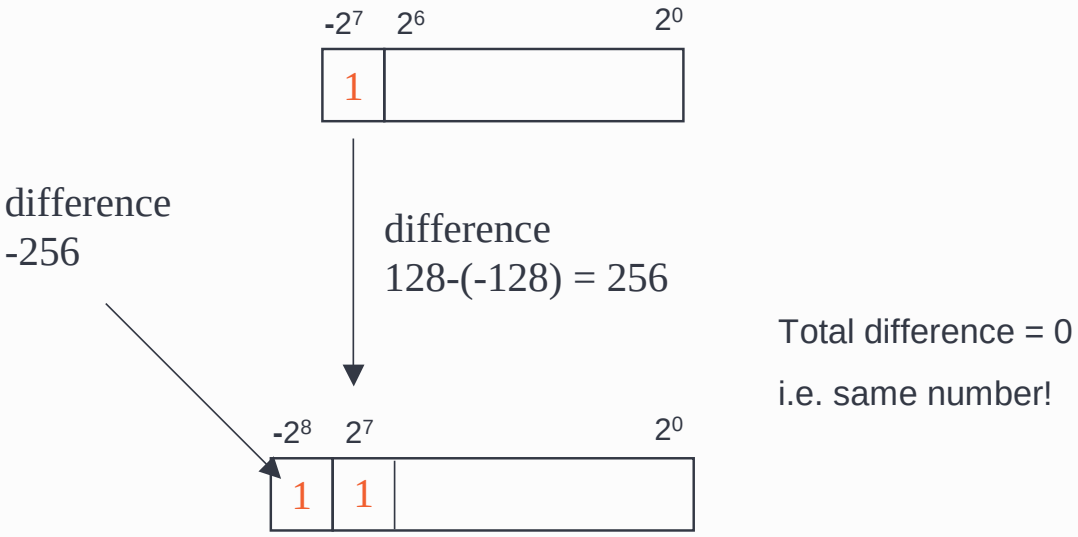
- How to translate an 8-bit 2's complement number to a 16-bit 2's complement number?



- This operation is known as **sign extension**.
- Result is the same: trivially so for positive numbers

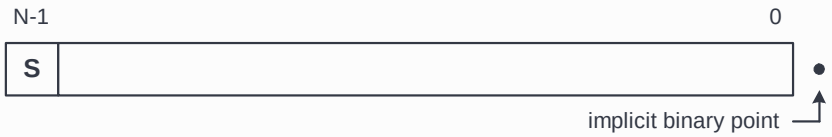
Sign Extension

- For negative numbers, consider a 1-bit sign extension

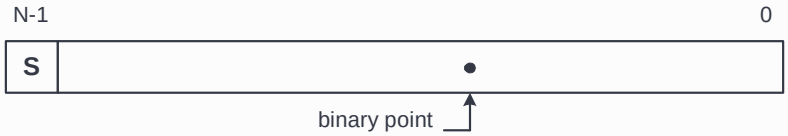


Fixed point representation

- So far, we have concentrated on **integer** representation with the **fractional** part.
- There is an implicit binary point to the right:

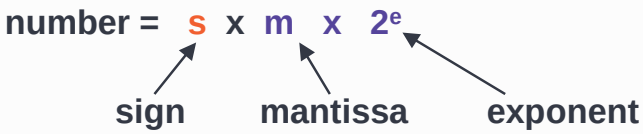


- In general, the binary point can be in the middle of the word (or off the end!)



Idea of floating point representation

- Although fixed point representation can cope with numbers with fractions, the range of values that can represented is still limited.
- Alternative: use the equivalent of “scientific notation”, but in binary:



- For example:

10.5 in fixed point 1010.1_b
Move binary point to left $1.0101_b \times 2^3$

10.5 = 1.3125 x 8

IEEE-754 standard floating point

- 32-bit single precision floating point:

$$x = -1^s \times 2^{exp-127} \times 1.frac$$
$$1.175 \times 10^{-38} < |x| < 1.7 \times 10^{38}$$

- MSB is sign-bit (same as fixed point)
- 8-bit exponent in bias-127 integer format (i.e. store 127+exponent)
- 23-bit to represent only the fractional part of the mantissa. The MSB of the mantissa is ALWAYS ‘1’, therefore it is not stored