Digital Design and Binary Numbers

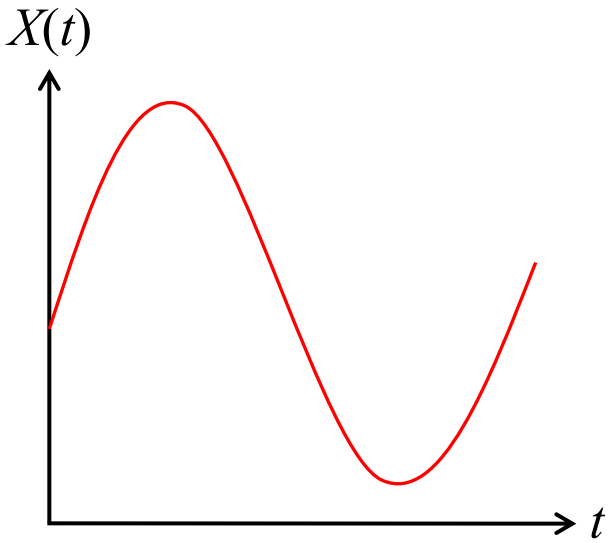
- UNIT-1

Digital Systems and Binary Numbers

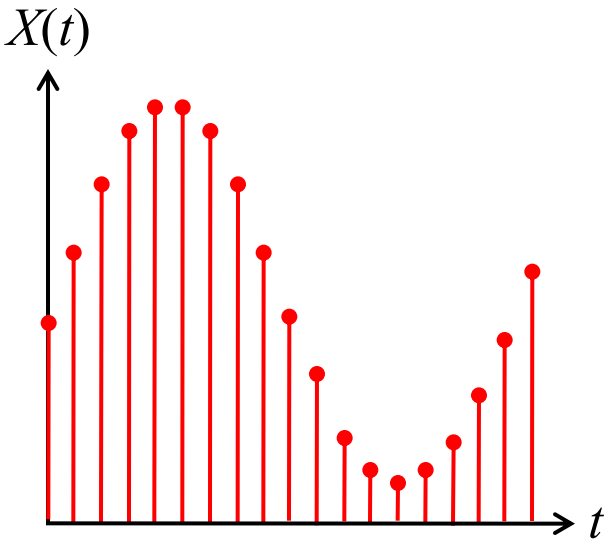
- Digital age and information age
- Digital computers
  - General purposes
  - Many scientific, industrial and commercial applications
- Digital systems
  - Telephone switching exchanges
  - Digital camera
  - Electronic calculators, PDA's
  - Digital TV
- Discrete information-processing systems
  - Manipulate discrete elements of information
  - For example, {1, 2, 3, …} and {A, B, C, …}…

Analog and Digital Signal

- Analog system
  - The physical quantities or signals may vary continuously over a specified range.
- Digital system
  - The physical quantities or signals can assume only discrete values.
  - Greater accuracy



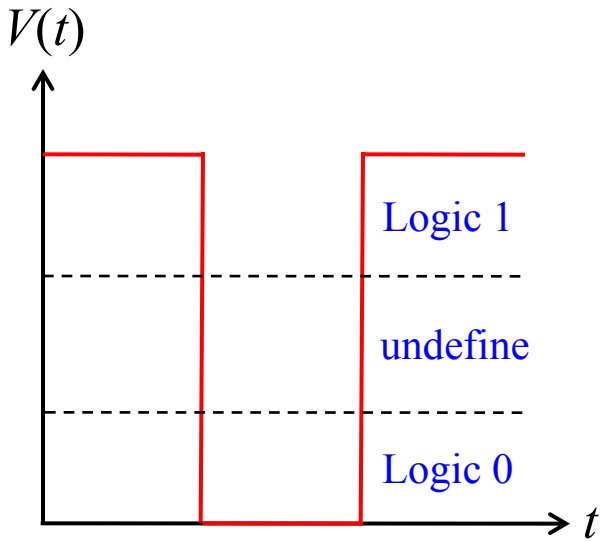
Analog signal



Digital signal

Binary Digital Signal

- An information variable represented by physical quantity.
- For digital systems, the variable takes on discrete values.
  - Two level, or binary values are the most prevalent values.
- Binary values are represented abstractly by:
  - Digits 0 and 1
  - Words (symbols) False (F) and True (T)
  - Words (symbols) Low (L) and High (H)
  - And words On and Off
- Binary values are represented by values or ranges of values of physical quantities.



Binary digital signal

Decimal Number System

- Base (also called radix) = 10
  - 10 digits { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }
- Digit Position
  - Integer & fraction
- Digit Weight
  - Weight = (*Base*) ^{Position}
- Magnitude
  - Sum of "*Digit* x *Weight*"
- Formal Notation

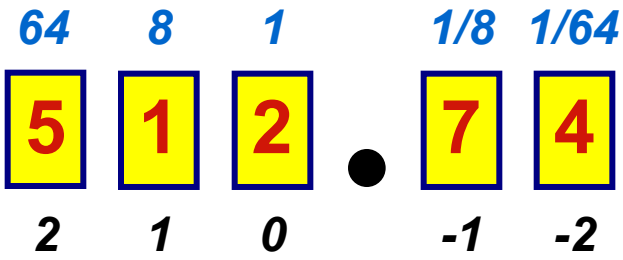
2	1	0		-1	-2
5	1	2	.	7	4
100	10	1		0.1	0.01
500	10	2		0.7	0.04

$$d_2*B^2+d_1*B^1+d_0*B^0+d_{-1}*B^{-1}+d_{-2}*B^{-2}$$

$$(512.74)_{10}$$

Octal Number System

- Base = 8
  - 8 digits { 0, 1, 2, 3, 4, 5, 6, 7 }
- Weights
  - Weight = (*Base*) ^{Position}
- Magnitude
  - Sum of "*Digit* x *Weight*"
- Formal Notation



$5*8^{2}+1*8^{1}+2*8^{0}+7*8^{-1}+4*8^{-2}$

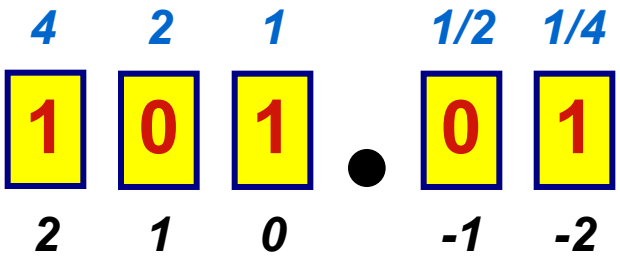
$=(330.9375)_{10}$

$(512.74)_{8}$

Binary Number System

- Base = 2
  - 2 digits { 0, 1 }, called *b*inary dig*its* or "*bits*"
- Weights
  - Weight = (*Base*) ^{Position}
- Magnitude
  - Sum of "*Bit* x *Weight*"
- Formal Notation
- Groups of bits 4 bits = *Nibble*

  8 bits = *Byte*



$1_2 * 2^2 + 0 * 2^1 + 1 * 2^0 + 0 * 2^{-1} + 1 * 2^{-}$

$= (5.25)_{10}$

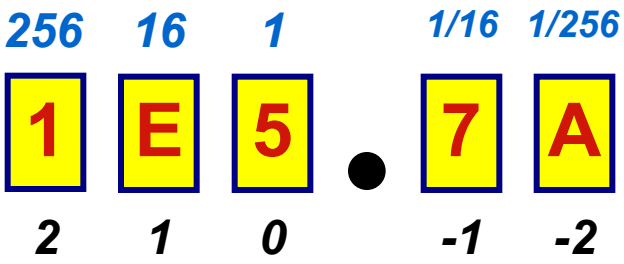
$(101.01)_2$

1 0 1 1

1 1 0 0 0 1 0 1

Hexadecimal Number System

- Base = 16
  - 16 digits { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F }
- Weights
  - Weight = (*Base*) ^{Position}
- Magnitude
  - Sum of "*Digit* x *Weight*"
- Formal Notation



$$1*16^{2}+14*16^{1}+5*16^{0}+7*16^{-1}+10*16^{-2}$$

$$=(485.4765625)_{10}$$

$$(1E5.7A)_{16}$$

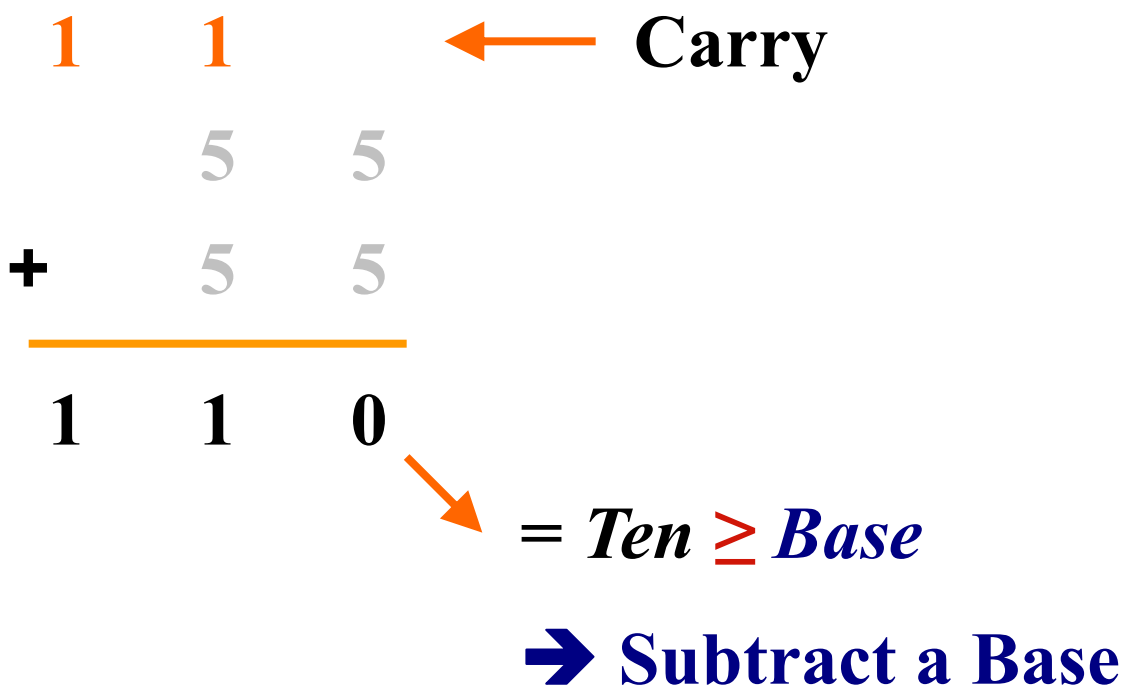
The Power of 2

n	2^n
0	$2^0=1$
1	$2^1=2$
2	$2^2=4$
3	$2^3=8$
4	$2^4=16$
5	$2^5=32$
6	$2^6=64$
7	$2^7=128$

n	2^n	
8	$2^8=256$	
9	$2^9=512$	
10	$2^{10}=1024$	Kilo
11	$2^{11}=2048$	
12	$2^{12}=4096$	
20	$2^{20}=1M$	Mega
30	$2^{30}=1G$	Giga
40	$2^{40}=1T$	Tera

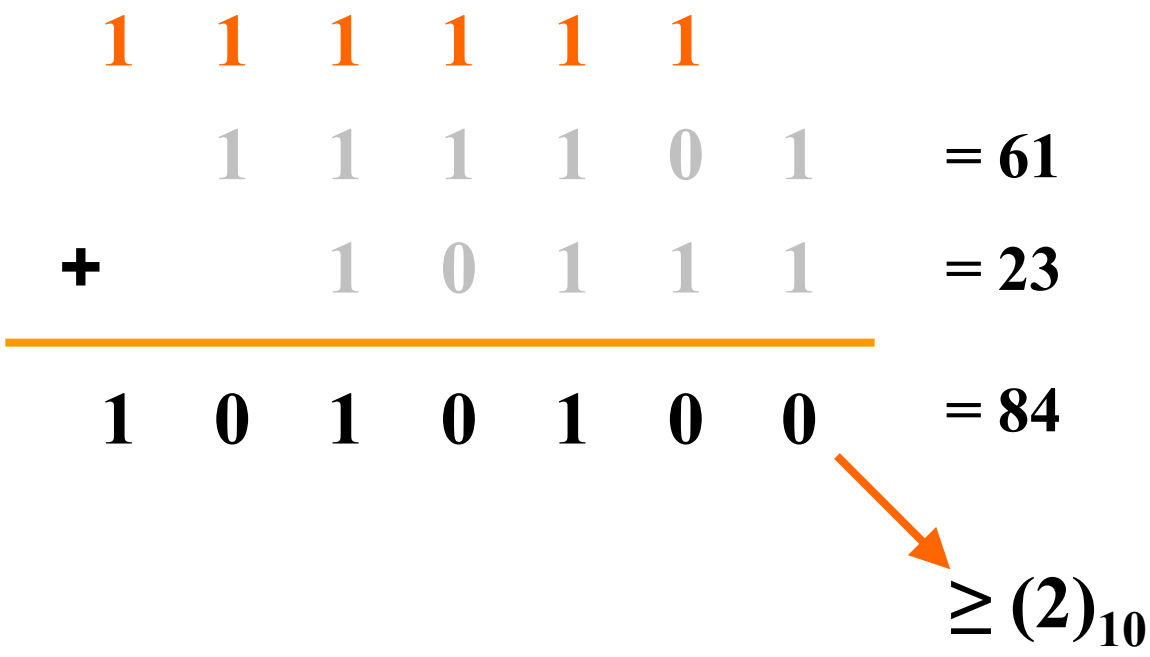
Addition

- Decimal Addition



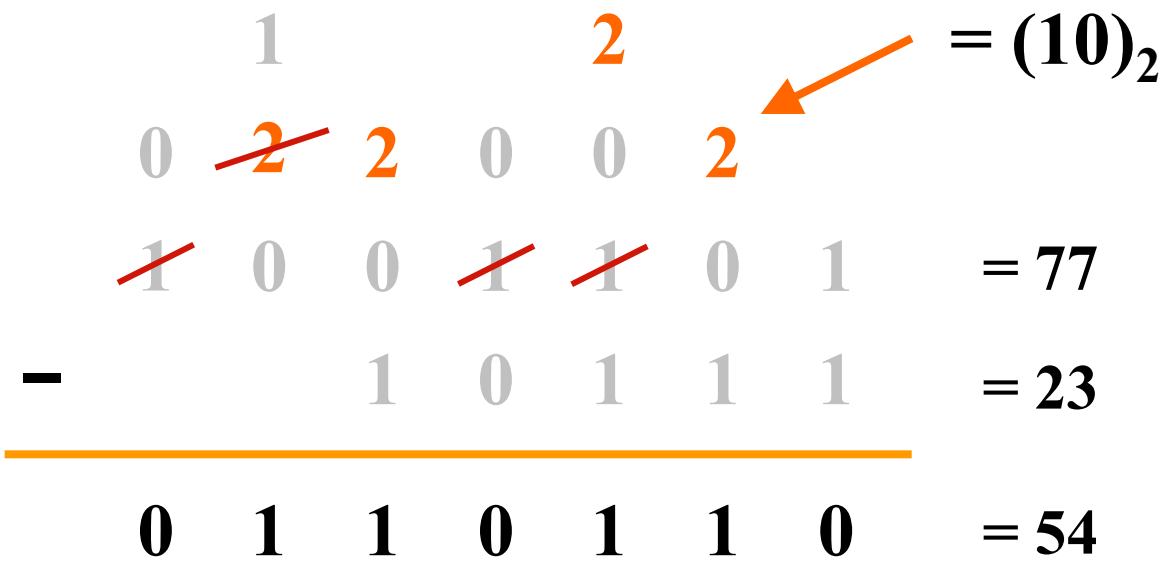
Binary Addition

- Column Addition



Binary Subtraction

- Borrow a "Base" when needed

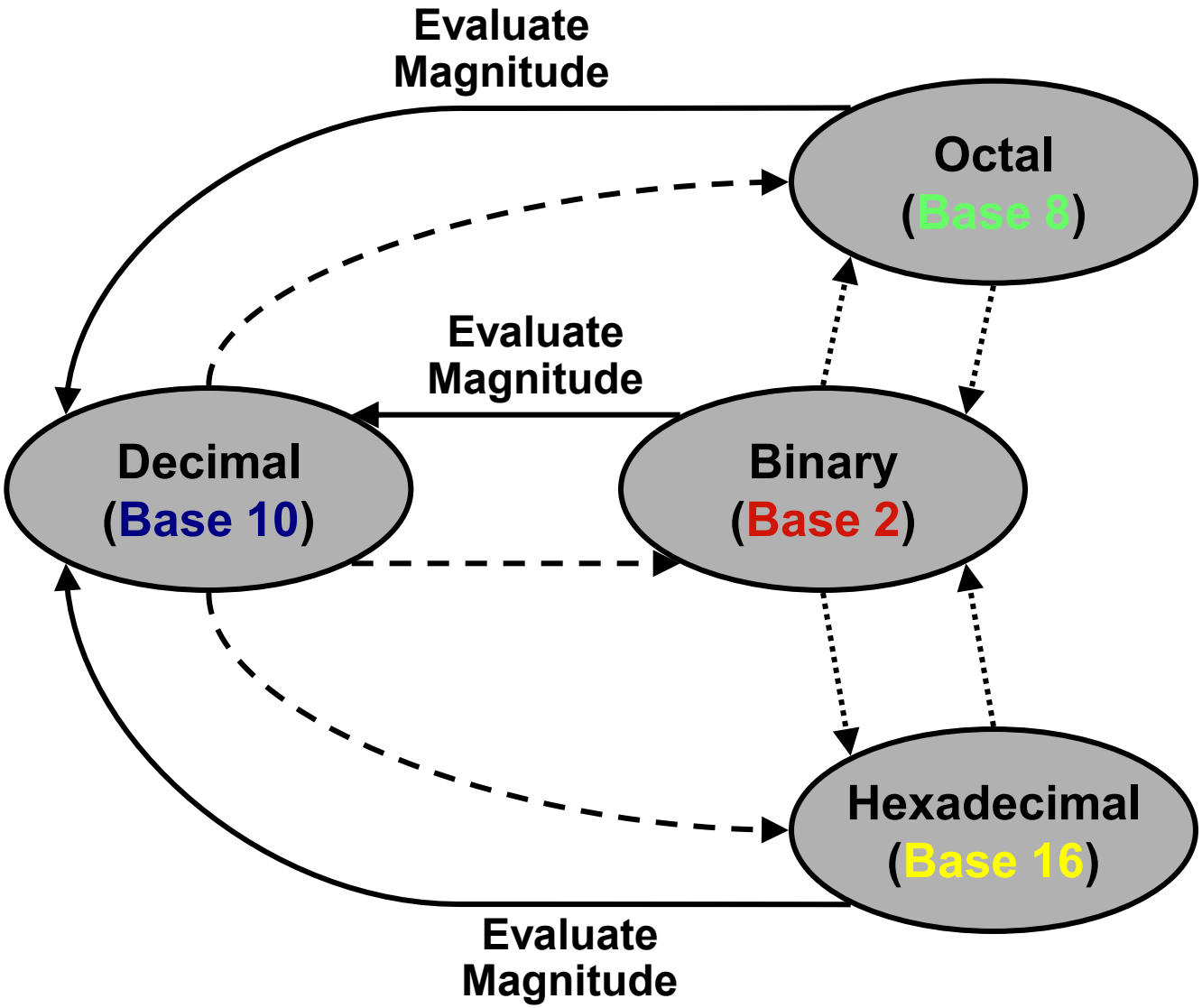


Binary Multiplication

- Bit by bit

			1	**0**	**1**	**1**	**1**
x				**1**	**0**	**1**	**0**
			0	**0**	**0**	**0**	**0**
		1	**0**	**1**	**1**	**1**	
	0	**0**	**0**	**0**	**0**		
1	**0**	**1**	**1**	**1**			
1	**1**	**1**	**0**	**0**	**1**	**1**	**0**

Number Base Conversions



Decimal (*Integer*) to Binary Conversion

- Divide the number by the 'Base' (=2)
- Take the remainder (either 0 or 1) as a coefficient
- Take the quotient and repeat the division

Example: $(13)_{10}$

	Quotient	Remainder	Coefficient
13 / 2 =	6	1	$a_0 = 1$
6 / 2 =	3	0	$a_1 = 0$
3 / 2 =	1	1	$a_2 = 1$
1 / 2 =	0	1	$a_3 = 1$

Answer: $(13)_{10} = (a_3\, a_2\, a_1\, a_0)_2 = (1101)_2$

MSB LSB

Decimal (*Fraction*) to Binary Conversion

- Multiply the number by the 'Base' (=2)
- Take the integer (either 0 or 1) as a coefficient
- Take the resultant fraction and repeat the division

Example: $(0.625)_{10}$

		Integer		Fraction	Coefficient
0.625	* 2 =	1	.	25	$a_{-1} = 1$
0.25	* 2 =	0	.	5	$a_{-2} = 0$
0.5	* 2 =	1	.	0	$a_{-3} = 1$

Answer: $(0.625)_{10} = (0.a_{-1}\, a_{-2}\, a_{-3})_2 = (0.101)_2$

MSB LSB

Decimal to Octal Conversion

Example: $(175)_{10}$

	Quotient	Remainder	Coefficient
175 / 8 =	21	7	$a_0 = 7$
21 / 8 =	2	5	$a_1 = 5$
2 / 8 =	0	2	$a_2 = 2$

Answer: $(175)_{10} = (a_2\, a_1\, a_0)_8 = (257)_8$

Example: $(0.3125)_{10}$

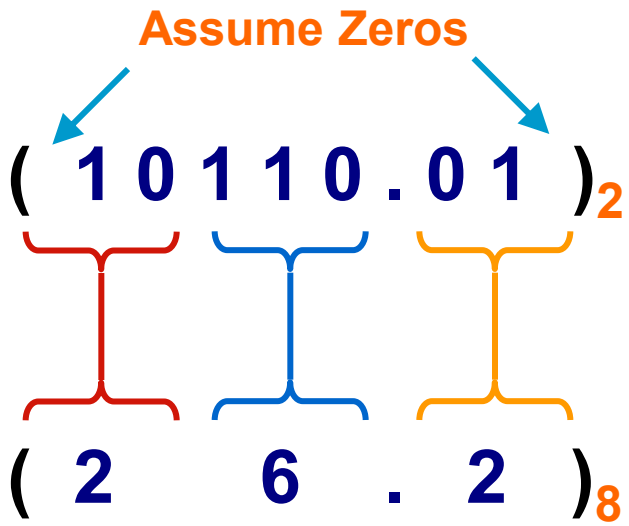
	Integer		Fraction	Coefficient
0.3125 * 8 =	2	.	5	$a_{-1} = 2$
0.5 * 8 =	4	.	0	$a_{-2} = 4$

Answer: $(0.3125)_{10} = (0.a_{-1}\, a_{-2}\, a_{-3})_8 = (0.24)_8$

Binary – Octal Conversion

- $8 = 2^3$
- Each group of 3 bits represents an octal digit

Example:



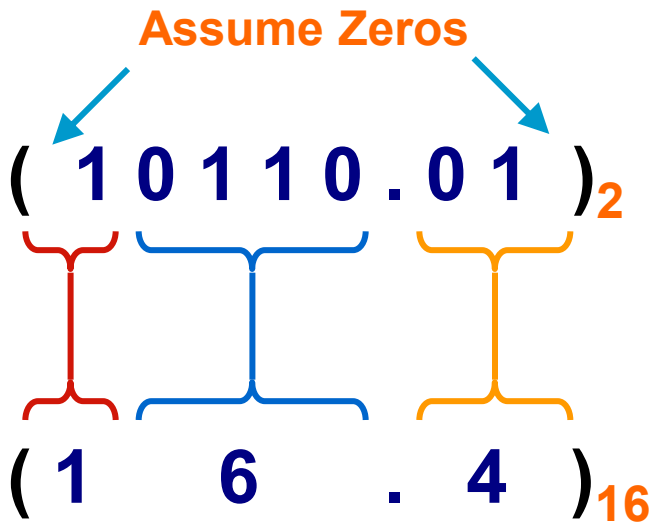
Octal	Binary
0	0 0 0
1	0 0 1
2	0 1 0
3	0 1 1
4	1 0 0
5	1 0 1
6	1 1 0
7	1 1 1

Works both ways (*Binary* to *Octal* & *Octal* to *Binary*)

Binary – Hexadecimal Conversion

- $16 = 2^4$
- Each group of 4 bits represents a hexadecimal digit

Example:



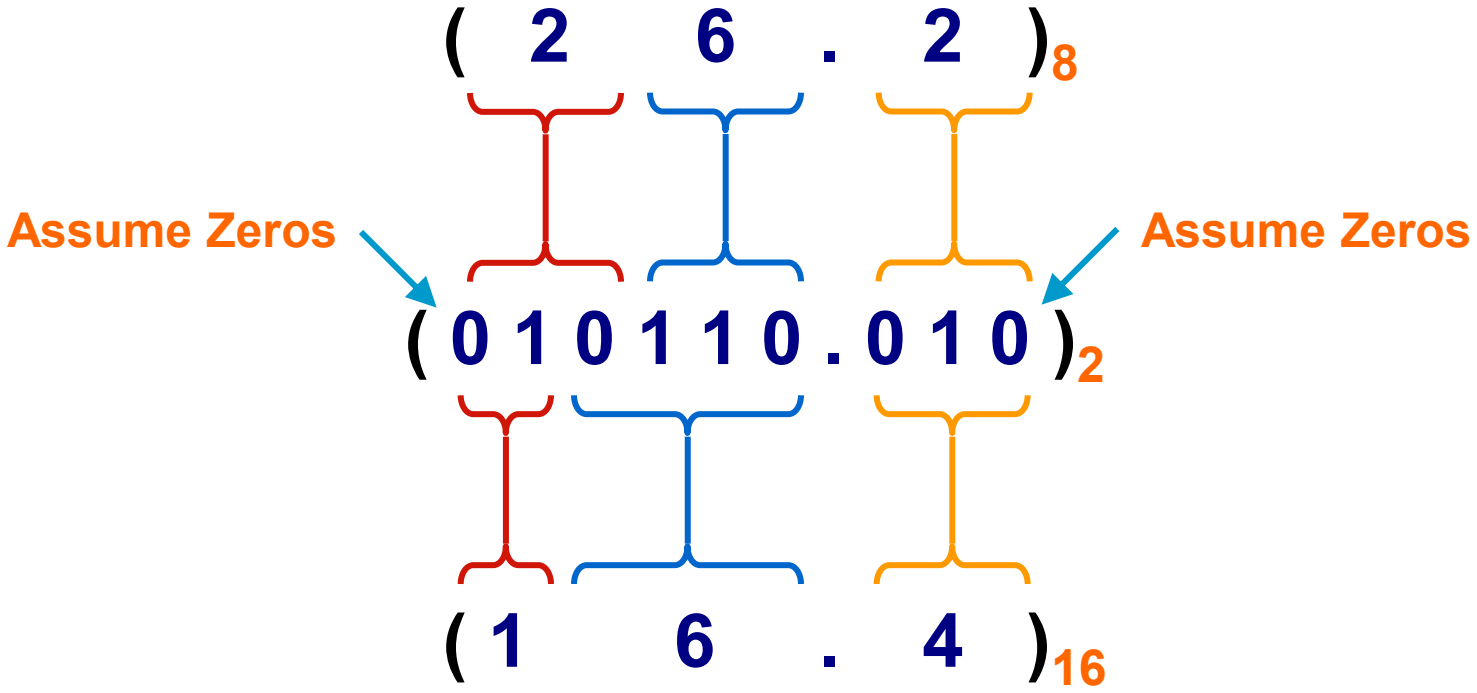
Hex	Binary
0	0 0 0 0
1	0 0 0 1
2	0 0 1 0
3	0 0 1 1
4	0 1 0 0
5	0 1 0 1
6	0 1 1 0
7	0 1 1 1
8	1 0 0 0
9	1 0 0 1
A	1 0 1 0
B	1 0 1 1
C	1 1 0 0
D	1 1 0 1
E	1 1 1 0
F	1 1 1 1

Works both ways (*Binary* to *Hex* & *Hex* to *Binary*)

Octal – Hexadecimal Conversion

- Convert to Binary as an intermediate step

Example:



Works both ways (*Octal* to *Hex* & *Hex* to *Octal*)

Decimal, Binary, Octal and Hexadecimal

Decimal	Binary	Octal	Hex
00	0000	00	0
01	0001	01	1
02	0010	02	2
03	0011	03	3
04	0100	04	4
05	0101	05	5
06	0110	06	6
07	0111	07	7
08	1000	10	8
09	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Complements

- There are two types of complements for each base-*r* system: the radix complement and diminished radix complement.
- **Diminished Radix Complement - (r-1)'s Complement**
  - Given a number *N* in base *r* having *n* digits, the (*r–1*)'s complement *of N* is defined as:

$$(r^n - 1) - N$$

- **Example for 6-digit <u>decimal</u> numbers**:
  - 9's complement is $(r^n - 1) - N = (10^6 - 1) - N = 999999 - N$
  - 9's complement of 546700 is 999999–546700 = 453299
- **Example for 7-digit <u>binary</u> numbers:**
  - 1's complement is $(r^n - 1) - N = (2^7 - 1) - N = 1111111 - N$
  - 1's complement of 1011000 is 1111111–1011000 = 0100111
- **Observation:**
  - Subtraction from $(r^n - 1)$ will never require a borrow
  - Diminished radix complement can be computed digit-by-digit
  - For binary: 1 – 0 = 1 and 1 – 1 = 0

Complements

- 1's Complement (*Diminished Radix* Complement)
  - All '0's become '1's
  - All '1's become '0's

  Example $(10110000)_2$

  ⇨ $(01001111)_2$

  If you add a number and its 1's complement …

$$
\begin{array}{r}
1\ 0\ 1\ 1\ 0\ 0\ 0\ 0 \\
+\ 0\ 1\ 0\ 0\ 1\ 1\ 1\ 1 \\
\hline
1\ 1\ 1\ 1\ 1\ 1\ 1\ 1
\end{array}
$$

Complements

- Radix Complement

The r's complement of an n-digit number N in base r is defined as $r^n - N$ for $N \neq 0$ and as 0 for $N = 0$. Comparing with the $(r - 1)$'s complement, we note that the r's complement is obtained by adding 1 to the $(r - 1)$'s complement, since $r^n - N = [(r^n - 1) - N] + 1$.

- Example: Base-10

The 10's complement of 012398 is 987602
The 10's complement of 246700 is 753300

- Example: Base-2

The 2's complement of 1101100 is 0010100
The 2's complement of 0110111 is 1001001

Complements

- 2's Complement (*Radix* Complement)
  - Take 1's complement then add 1

OR

  - Toggle all bits to the left of the first '1' from the right

Example:

Number:

1's Comp.:

```
  1 0 1 1 0 0 0 0          1 0 1 1 0 0 0 0
  0 1 0 0 1 1 1 1
+               1
-----------------
  0 1 0 1 0 0 0 0          0 1 0 1 0 0 0 0
```

Complements

- Subtraction with Complements
  - The subtraction of two n-digit unsigned numbers $M - N$ in base r can be done as follows:

1. Add the minuend M to the r's complement of the subtrahend N. Mathematically, $M + (r^n - N) = M - N + r^n$.
2. If $M \geqq N$, the sum will produce and end carry r^n, which can be discarded; what is left is the result $M - N$.
3. If $M < N$, the sum does not produce an end carry and is equal to $r^n - (N - M)$, which is the r's complement of $(N - M)$. To obtain the answer in a familiar form, take the r's complement of the sum and place a negative sign in front.

Complements

- Example 1.5
  - Using 10's complement, subtract 72532 – 3250.

	$M =$	72532
10's complement of	$N =$	+96750
	Sum =	169282
	Discard end carry 10^5 =	–100000
	Answer =	69282

- Example 1.6
  - Using 10's complement, subtract 3250 – 72532.

	M =	03250
10's complement of	N =	+27468
	Sum =	30718

→ There is no end carry.

→ Therefore, the answer is – (10's complement of 30718) = – 69282.

Complements

- Example
  - Given the two binary numbers $X = 1010100$ and $Y = 1000011$, perform the subtraction (a) $X - Y$; and (b) $Y - X$, by using 2's complement.

(a)		
	X =	1010100
	2's complement of Y =	+0111101
	Sum =	10010001
	Discard end carry 2^7 =	−10000000
	Answer. X – Y =	0010001

(b)		
	Y =	1000011
	2's complement of X =	+ 0101100
	Sum =	1101111

There is no end carry. Therefore, the answer is Y – X = – (2's complement of 1101111) = – 0010001.

Complements

- Subtraction of unsigned numbers can also be done by means of the $(r-1)$'s complement. Remember that the $(r-1)$'s complement is one less then the r's complement.
- Example
  - Repeat Example, but this time using 1's complement.

(a) $X - Y = 1010100 - 1000011$

$X =$	1010100
1's complement of $Y =$	± 0111100
Sum =	10010000
End-around carry =	+ 1
Answer. $X - Y =$	0010001

(b) $Y - X = 1000011 - 1010100$

$Y =$	1000011
1's complement of $X =$	+0101011
Sum =	1101110

There is no end carry, Therefore, the answer is Y – X = – (1's complement of 1101110) = – 0010001.

Signed Binary Numbers

- To represent negative integers, we need a notation for negative values.
- It is customary to represent the sign with a bit placed in the leftmost position of the number since binary digits.
- The convention is to make the sign bit 0 for positive and 1 for negative.
- Example:

Signed-magnitude representation:	10001001
Signed-1's-complement representation:	11110110
Signed-2's-complement representation:	11110111

- Table 1.3 lists all possible four-bit signed binary numbers in the three representations.

Signed Binary Numbers

Table 1.3
Signed Binary Numbers

Decimal	Signed-2's Complement	Signed-1's Complement	Signed Magnitude
+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
−0	—	1111	1000
−1	1111	1110	1001
−2	1110	1101	1010
−3	1101	1100	1011
−4	1100	1011	1100
−5	1011	1010	1101
−6	1010	1001	1110
−7	1001	1000	1111
−8	1000	—	—

Signed Binary Numbers

- Arithmetic addition
  - The addition of two numbers in the signed-magnitude system follows the rules of ordinary arithmetic. **If the signs are the same**, we add the two magnitudes and give the sum the common sign. **If the signs are different**, we subtract the smaller magnitude from the larger and give the difference the sign if the larger magnitude.
  - The addition of two signed binary numbers with negative numbers represented in signed-2's-complement form is obtained from the addition of the two numbers, including their sign bits.
  - A carry out of the sign-bit position is discarded.
- Example:

+ 6	00000110	− 6	11111010
+13	00001101	+13	00001101
+ 19	00010011	+ 7	00000111
+ 6	00000110	− 6	11111010
−13	11110011	−13	11110011
− 7	11111001	− 19	11101101

Signed Binary Numbers

- Arithmetic Subtraction
  - In 2's-complement form:

1. Take the 2's complement of the subtrahend (including the sign bit) and add it to the minuend (including sign bit).
2. A carry out of sign-bit position is discarded.

$$(\pm A)-(+B)=(\pm A)+(-B)$$
$$(\pm A)-(-B)=(\pm A)+(+B)$$

- Example:

$(-6)-(-13)$ → $(11111010-11110011)$

→ $(11111010+00001101)$

→ $00000111\ (+7)$

Binary Codes

- BCD Code
  - A number with k decimal digits will require 4k bits in BCD.
  - Decimal 396 is represented in BCD with 12bits as 0011 1001 0110, with each group of 4 bits representing one decimal digit.
  - A decimal number in BCD is the same as its equivalent binary number only when the number is between 0 and 9.
  - The binary combinations 1010 through 1111 are not used and have no meaning in BCD.

Table 1.4
Binary-Coded Decimal (BCD)

Decimal Symbol	BCD Digit
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Binary Code

- Example:
  - Consider decimal 185 and its corresponding value in BCD and binary:

$$(185)_{10} = (0001\ 1000\ 0101)_{BCD} = (10111001)_2$$

- BCD addition

4	0100	4	0100	8	1000
+5	+0101	+8	+1000	+9	+1001
9	1001	12	1100	17	10001
			+0110		+0110
			10010		10111

Binary Code

- Example:
  - Consider the addition of 184 + 576 = 760 in BCD:

BCD	1	1		
	0001	1000	0100	184
	+0101	0111	0110	+576
Binary sum	0111	10000	1010	
Add 6		0110	0110	
BCD sum	0111	0110	0000	760

- Decimal Arithmetic: (+375) + (-240) = +135

0	375
+9	760
0	135

Hint 6: using 10's of BCD

Binary Codes

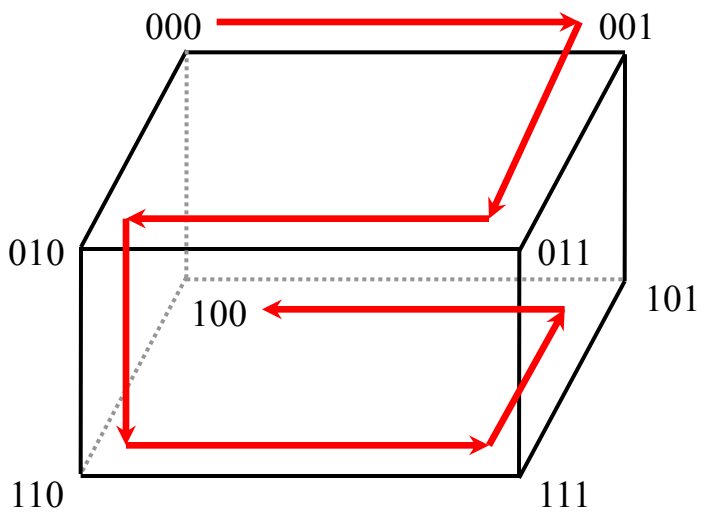
- Other Decimal Codes

Table 1.5
Four Different Binary Codes for the Decimal Digits

Decimal Digit	BCD 8421	2421	Excess-3	8, 4, −2, −1
0	0000	0000	0011	0000
1	0001	0001	0100	0111
2	0010	0010	0101	0110
3	0011	0011	0110	0101
4	0100	0100	0111	0100
5	0101	1011	1000	1011
6	0110	1100	1001	1010
7	0111	1101	1010	1001
8	1000	1110	1011	1000
9	1001	1111	1100	1111
	1010	0101	0000	0001
Unused	1011	0110	0001	0010
bit	1100	0111	0010	0011
combi-	1101	1000	1101	1100
nations	1110	1001	1110	1101
	1111	1010	1111	1110

Binary Codes

- Gray Code
  - The advantage is that only bit in the code group changes in going from one number to the next.
    - » Error detection.
    - » Representation of analog data.
    - » Low power design.



1-1 and onto!!

Table 1.6
Gray Code

Gray Code	Decimal Equivalent
0000	0
0001	1
0011	2
0010	3
0110	4
0111	5
0101	6
0100	7
1100	8
1101	9
1111	10
1110	11
1010	12
1011	13
1001	14
1000	15

Binary Codes

- American Standard Code for Information Interchange (ASCII) Character Code

Table 1.7
American Standard Code for Information Interchange (ASCII)

	$b_7b_6b_5$							
$b_4b_3b_2b_1$	000	001	010	011	100	101	110	111
0000	NUL	DLE	SP	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	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	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	−	o	DEL

Binary Codes

- ASCII Character Code

Control characters

NUL	Null	DLE	Data-link escape
SOH	Start of heading	DC1	Device control 1
STX	Start of text	DC2	Device control 2
ETX	End of text	DC3	Device control 3
EOT	End of transmission	DC4	Device control 4
ENQ	Enquiry	NAK	Negative acknowledge
ACK	Acknowledge	SYN	Synchronous idle
BEL	Bell	ETB	End-of-transmission block
BS	Backspace	CAN	Cancel
HT	Horizontal tab	EM	End of medium
LF	Line feed	SUB	Substitute
VT	Vertical tab	ESC	Escape
FF	Form feed	FS	File separator
CR	Carriage return	GS	Group separator
SO	Shift out	RS	Record separator
SI	Shift in	US	Unit separator
SP	Space	DEL	Delete

ASCII Character Codes

- American Standard Code for Information Interchange (Refer to Table 1.7)
- A popular code used to represent information sent as character-based data.
- It uses 7-bits to represent:
  - 94 Graphic printing characters.
  - 34 Non-printing characters.
- Some non-printing characters are used for text format (e.g. BS = Backspace, CR = carriage return).
- Other non-printing characters are used for record marking and flow control (e.g. STX and ETX start and end text areas).

ASCII Properties

- ASCII has some interesting properties:
  - Digits 0 to 9 span Hexadecimal values 30_{16} to 39_{16}
  - Upper case A-Z span 41_{16} to $5A_{16}$
  - Lower case a-z span 61_{16} to $7A_{16}$
    - » Lower to upper case translation (and vice versa) occurs by flipping bit 6.

Binary Codes

- Error-Detecting Code
  - To detect errors in data communication and processing, an eighth bit is sometimes added to the ASCII character to indicate its parity.
  - A parity bit is an extra bit included with a message to make the total number of 1's either even or odd.
- Example:
  - Consider the following two characters and their even and odd parity:

	With even parity	With odd parity
ASCII A = 1000001	01000001	11000001
ASCII T = 1010100	11010100	01010100

Binary Codes

- Error-Detecting Code
  - Redundancy (e.g. extra information), in the form of extra bits, can be incorporated into binary code words to detect and correct errors.
  - A simple form of redundancy is parity, an extra bit appended onto the code word to make the number of 1's odd or even. Parity can detect all single-bit errors and some multiple-bit errors.
  - A code word has even parity if the number of 1's in the code word is even.
  - A code word has odd parity if the number of 1's in the code word is odd.
  - Example:

Message A: 100010011 (even parity)

Message B: 100010010 (odd parity)

Floating-Point Representation

- If we are clever programmers, we can perform floating-point calculations using any integer format.
- This is called *floating-point emulation*, because floating point values aren't stored as such; we just create programs that make it seem as if floating-point values are being used.
- Most of today's computers are equipped with specialized hardware that performs floating-point arithmetic with no special programming required.
  - Not embedded processors!

Floating-Point Representation

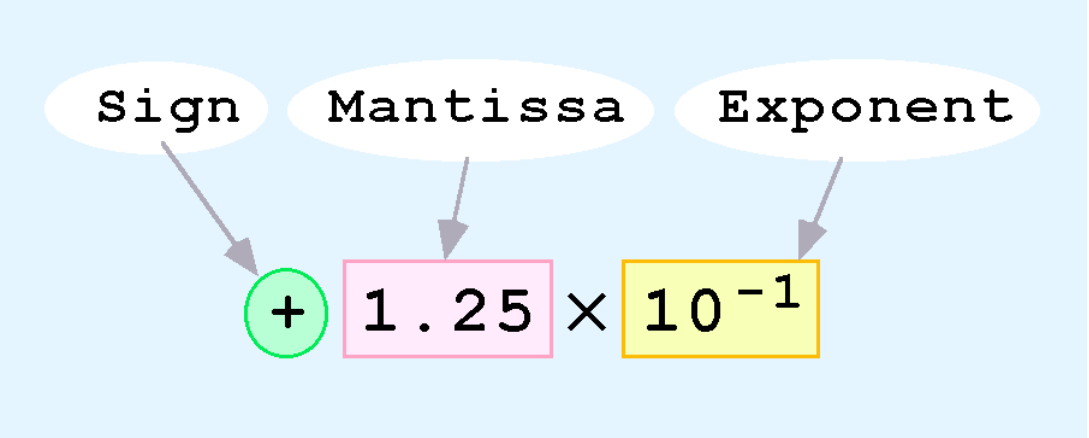
- Floating-point numbers allow an arbitrary number of decimal places to the right of the decimal point.
  - For example: $0.5 \times 0.25 = 0.125$
- They are often expressed in scientific notation.
  - For example:

    $0.125 = 1.25 \times 10^{-1}$

    $5{,}000{,}000 = 5.0 \times 10^{6}$

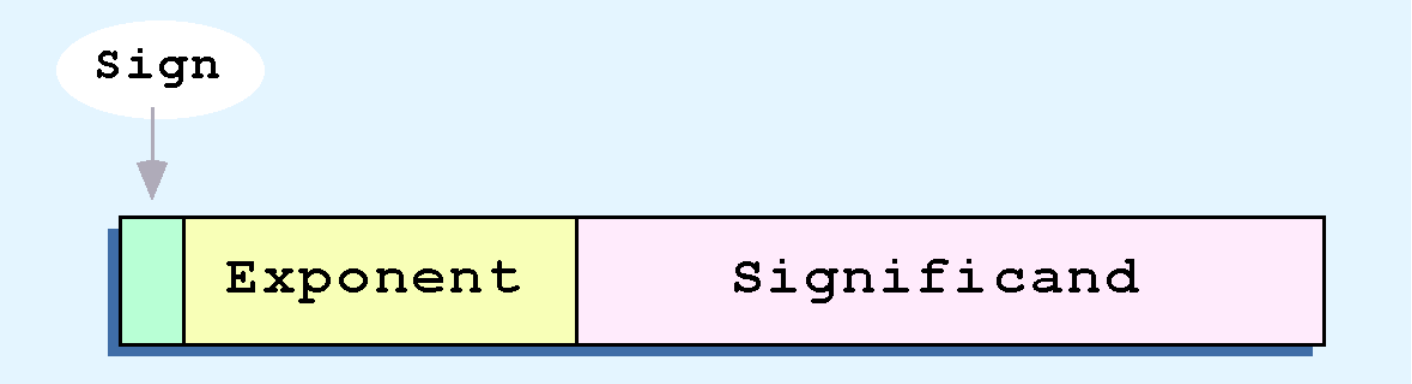
Floating-Point Representation

- Computers use a form of scientific notation for floating-point representation
- Numbers written in scientific notation have three components:



Floating-Point Representation

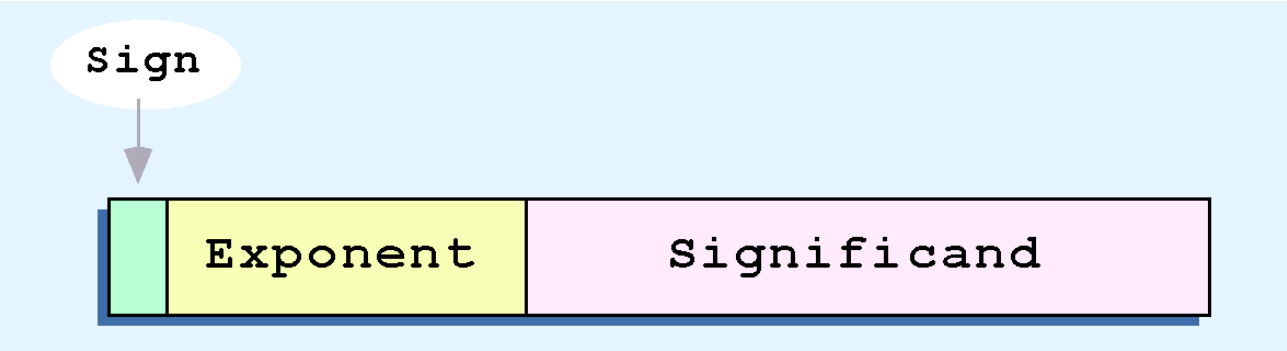
- Computer representation of a floating-point number consists of three fixed-size fields:



- This is the standard arrangement of these fields.

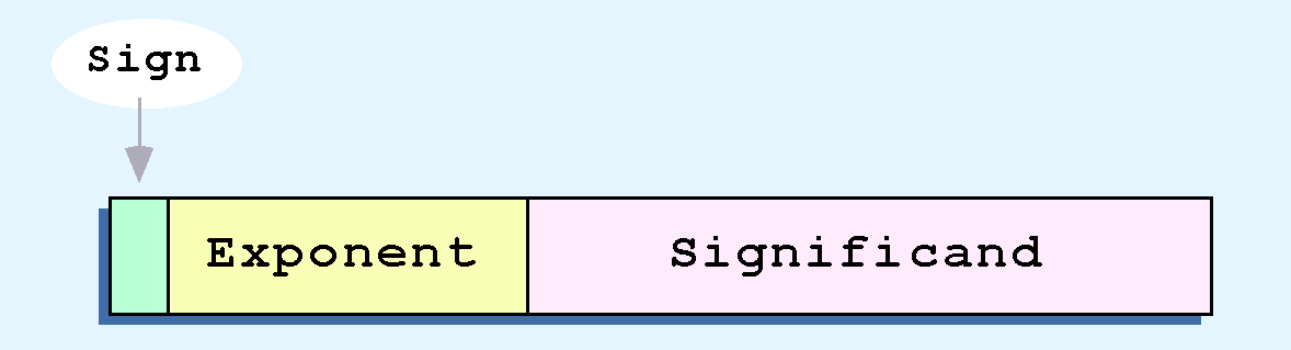
Note: Although "significand" and "mantissa" do not technically mean the same thing, many people use these terms interchangeably. We use the term "significand" to refer to the fractional part of a floating point number.

Floating-Point Representation



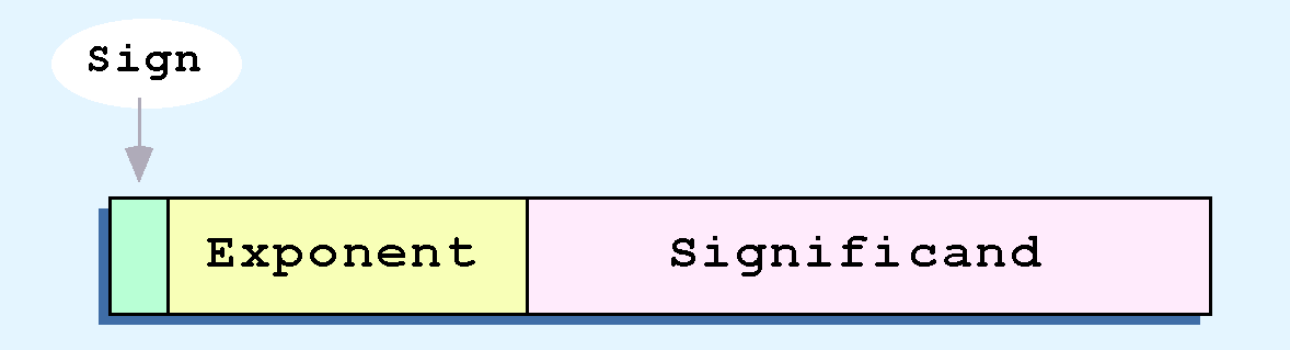
- The one-bit sign field is the sign of the stored value.
- The size of the exponent field determines the range of values that can be represented.
- The size of the significand determines the precision of the representation.

Floating-Point Representation



- We introduce a hypothetical “Simple Model” to explain the concepts
- In this model:
  - A floating-point number is 14 bits in length
  - The exponent field is 5 bits
  - The significand field is 8 bits

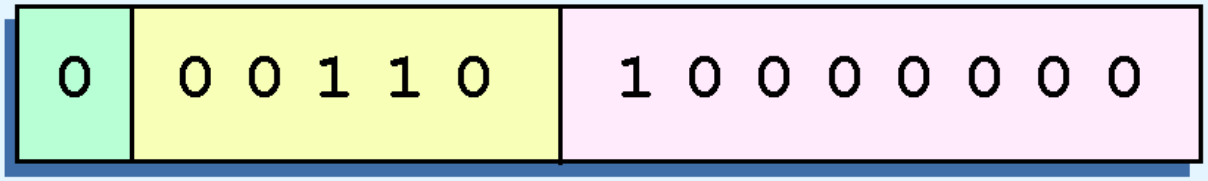
Floating-Point Representation



- The significand is always preceded by an implied binary point.
- Thus, the significand always contains a fractional binary value.
- The exponent indicates the power of 2 by which the significand is multiplied.

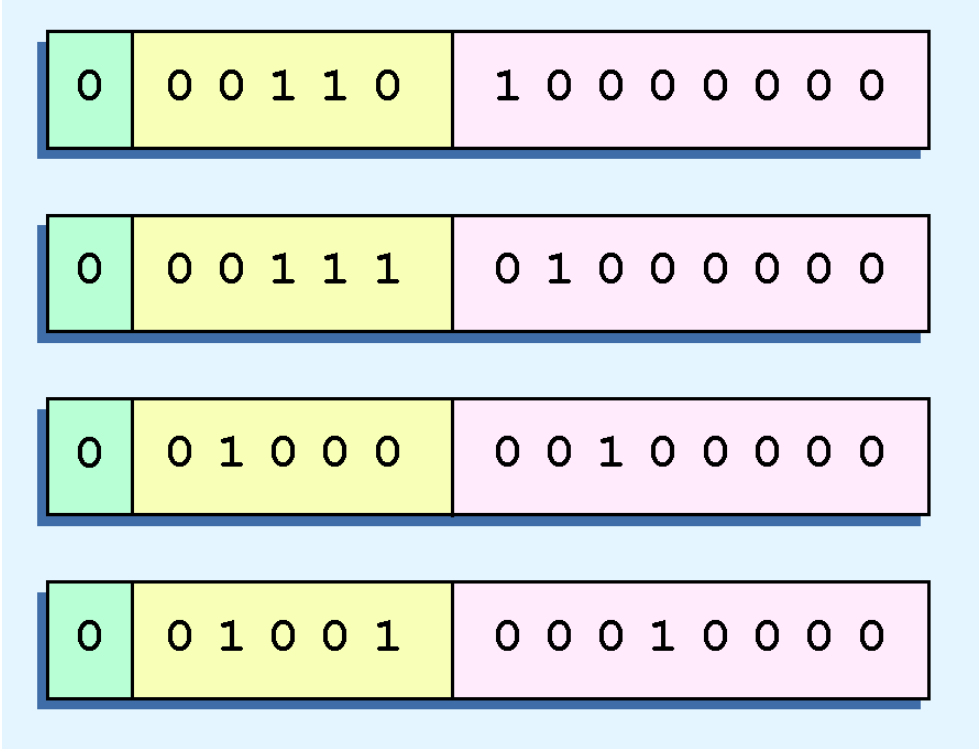
Floating-Point Representation

- Example:
  - Express 32_{10} in the simplified 14-bit floating-point model.
- We know that 32 is 2^5. So in (binary) scientific notation $32 = 1.0 \times 2^5 = 0.1 \times 2^6$.
  - In a moment, we'll explain why we prefer the second notation versus the first.
- Using this information, we put 110 (= 6_{10}) in the exponent field and 1 in the significand as shown.

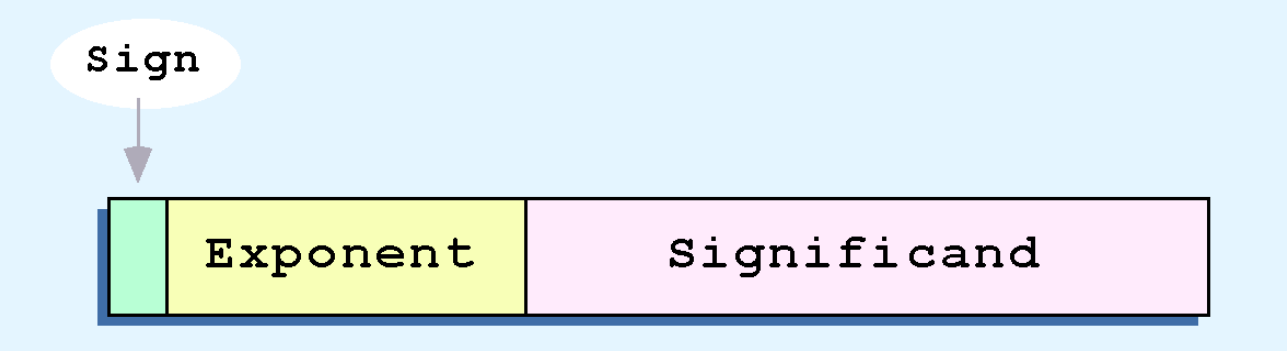


Floating-Point Representation

- The illustrations shown at the right are *all* equivalent representations for 32 using our simplified model.
- Not only do these synonymous representations waste space, but they can also cause confusion.



Floating-Point Representation



- Another problem with our system is that we have made no allowances for negative exponents. We have no way to express 0.5 ($=2^{-1}$)! (Notice that there is no sign in the exponent field.)

All of these problems can be fixed with no changes to our basic model.

Floating-Point Representation

- To resolve the problem of synonymous forms, we establish a rule that the first digit of the significand must be 1, with no ones to the left of the radix point.
- This process, called *normalization*, results in a unique pattern for each floating-point number.
  - In our simple model, all significands must have the form 0.1xxxxxxxx
  - For example, $4.5 = 100.1 \times 2^0 = 1.001 \times 2^2 = 0.1001 \times 2^3$. The last expression is correctly normalized.

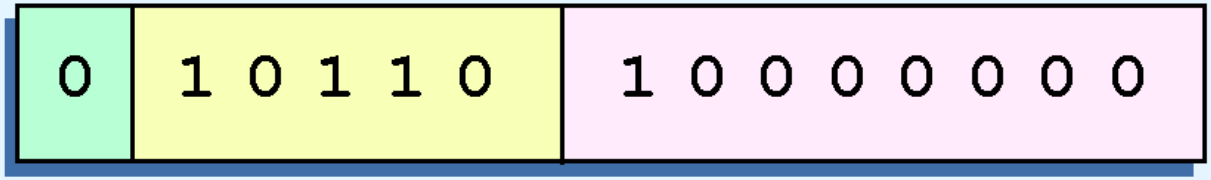
In our simple instructional model, we use no implied bits.

Floating-Point Representation

- To provide for negative exponents, we will use a *biased exponent*.
- A bias is a number that is approximately midway in the range of values expressible by the exponent. We subtract the bias from the value in the exponent to determine its true value.
  - In our case, we have a 5-bit exponent. We will use 16 for our bias. This is called *excess-16* representation.
- In our model, exponent values less than 16 are negative, representing fractional numbers.

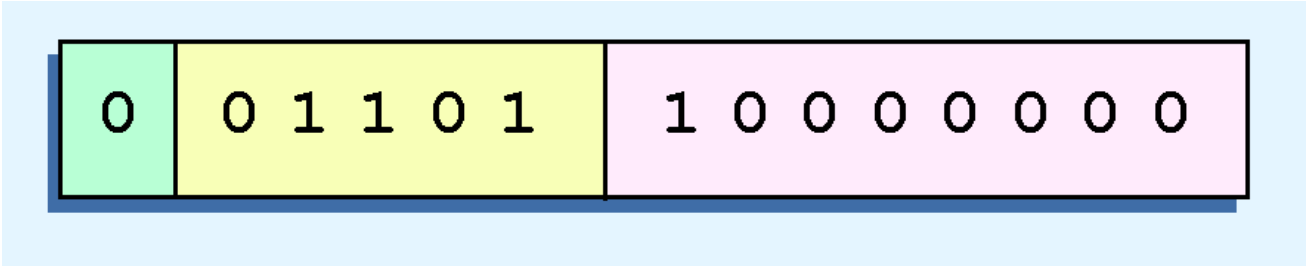
Example 1

- Example:
  - Express 32_{10} in the revised 14-bit floating-point model.
- We know that $32 = 1.0 \times 2^5 = 0.1 \times 2^6$.
- To use our excess 16 biased exponent, we add 16 to 6, giving 22_{10} (=10110_2).
- So we have:



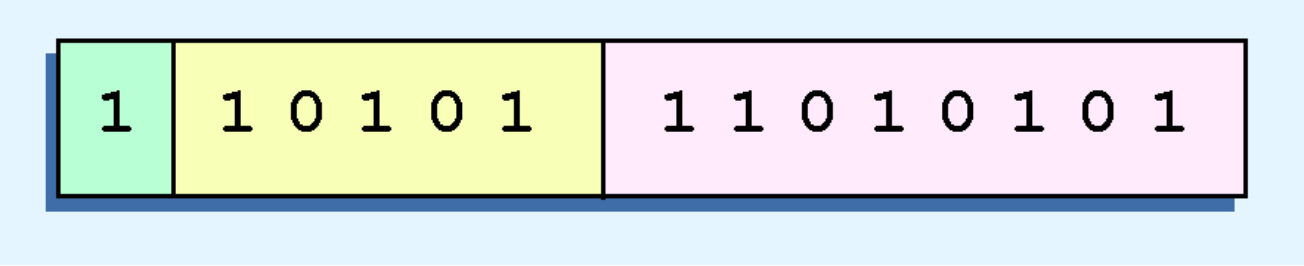
Example 2

- Example:
  - Express 0.0625_{10} in the revised 14-bit floating-point model.
- We know that 0.0625 is 2^{-4}. So in (binary) scientific notation $0.0625 = 1.0 \times 2^{-4} = 0.1 \times 2^{-3}$.
- To use our excess 16 biased exponent, we add 16 to -3, giving 13_{10} (=01101_2).



Example 3

- Example:
  - Express -26.625_{10} in the revised 14-bit floating-point model.
- We find $26.625_{10} = 11010.101_2$. Normalizing, we have: $26.625_{10} = 0.11010101 \times 2^5$.
- To use our excess 16 biased exponent, we add 16 to 5, giving 21_{10} ($=10101_2$). We also need a 1 in the sign bit.



Floating-Point Standards

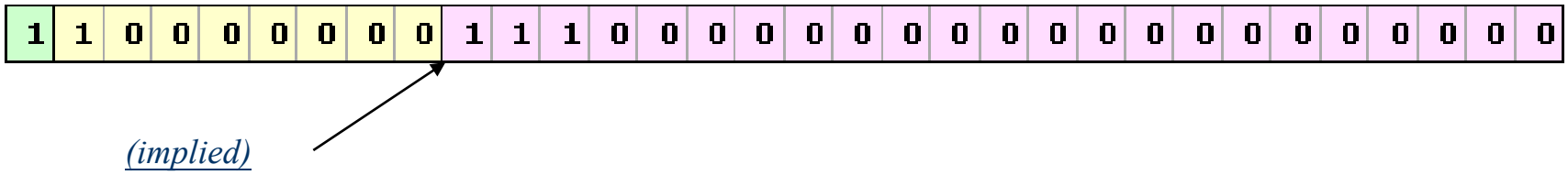
- The IEEE has established a standard for floating-point numbers
- The IEEE-754 *single precision* floating point standard uses an 8-bit exponent (with a bias of 127) and a 23-bit significand.
- The IEEE-754 *double precision* standard uses an 11-bit exponent (with a bias of 1023) and a 52-bit significand.

Floating-Point Representation

- In both the IEEE single-precision and double-precision floating-point standard, the significant has an implied 1 to the LEFT of the radix point.
  - The format for a significand using the IEEE format is: 1.xxx…
  - For example, $4.5 = .1001 \times 2^3$ in IEEE format is $4.5 = 1.001 \times 2^2$. The 1 is implied, which means is does not need to be listed in the significand (the significand would include only 001).

Floating-Point Representation

- Example: Express -3.75 as a floating point number using IEEE single precision.
- First, let's normalize according to IEEE rules:
  - $3.75 = -11.11_2 = -1.111 \times 2^1$
  - The bias is 127, so we add 127 + 1 = 128 (this is our exponent)
  - The first 1 in the significand is implied, so we have:



  - Since we have an implied 1 in the significand, this equates to

$$-(1).111_2 \times 2^{(128-127)} = -1.111_2 \times 2^1 = -11.11_2 = -3.75.$$

Linear Block Code

- Hamming Code is a Linear Block Code. Linear Block Code means that the codeword is generated by multiplying the message vector with the generator matrix.
- Minimum weight as large as possible. If minimum weight is *2t+1*, capable of detecting *2t* error bits and correcting t error bits.

Cyclic Codes

- Hamming code is useful but there exist codes that offers same (if not larger) error control capabilities while can be implemented much simpler.
- Cyclic code is a linear code that **any cyclic shift of a codeword is still a codeword**.
- Makes encoding/decoding much simpler, no need of matrix multiplication.

Cyclic code

- Polynomial representation of cyclic codes.
  $C(x) = C_{n-1}x^{n-1} + C_{n-2}x^{n-2} + \ldots + C_1x^1 + C_0x^0$,
  where, **in this course**, the coefficients belong to the binary field *{0,1}*.
- That is, if the codeword is *(1010011)* (*c6* first, *c0* last), we **write it as** $x^6 + x^4 + x + 1$.
- Addition and subtraction of polynomials – Done by doing binary addition or subtraction on each bit *individually*, no carry and no borrow.
- Division and multiplication of polynomials. Try divide $x^3 + x^2 + x + 1$ by $x + 1$.

Cyclic Code

- A *(n,k)* cyclic code can be generated by a polynomial *g(x)* which has
  - degree *n-k* and
  - is a factor of $x^n - 1$.

  Call it the **generator polynomial**.
- Given message bits, $(m_{k-1}...m_1m_0)$, the code is generated simply as:

$$C(x) = \sum_{j=0}^{k-1} m_j x^j g(x),$$

- *In other words, C(x) can be considered as the product of m(x) and g(x).*

Example

- A (7,4) cyclic code with $g(x) = x^3 + x + 1$.
- If $m(x) = x^3 + 1$, $C(x) = x^6 + x^4 + x + 1$.

Error Detection with Cyclic Code

- A (7,4) cyclic code with $g(x) = x^3 + x + 1$.
- If the received polynomial is $x^6 + x^5 + x^2 + 1$, are there any errors? Or, is this a code polynomial?

Error Detection with Cyclic Code

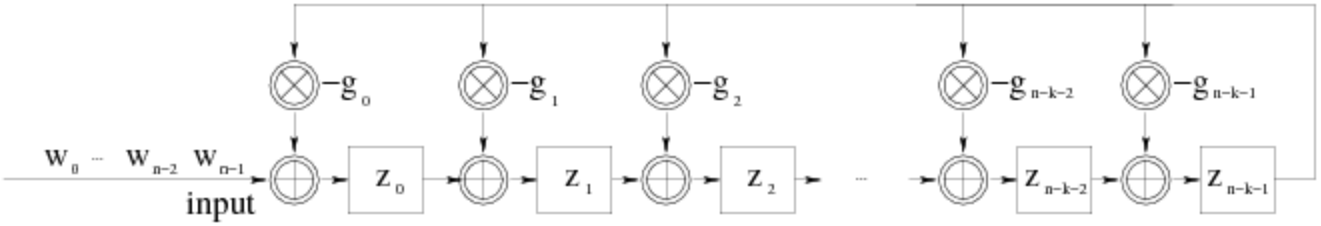
- A (7,4) cyclic code with $g(x) = x^3 + x + 1$.
- If the received polynomial is $x^6 + x^5 + x^2 + 1$, are there any errors?
- We divide $x^6 + x^5 + x^2 + 1$ by $x^3 + x + 1$, and the remainder is $x^3 + 1$. **The point is that the remainder is not 0**. So it is not a code polynomial, so there are errors.

Cyclic code used in IEEE 802

- **g(x) =** $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$
  - all single and double bit errors
  - all errors with an odd number of bits
  - all burst errors of length 32 or less

Division Circuit

- You probably would ask that we can also detect errors with the Hamming code. However it needs matrix multiplication. The division can actually be done very efficiently, even with hardware.
- Division of polynomials can be done efficiently by the division circuit. (just to know there exists such a thing, no need to understand it)



Cyclic Code

- One way of thinking it is to write it out as the generator matrix

$$G = \begin{pmatrix} g_{n-k} & g_{n-k-1} & \cdots & g_1 & g_0 & 0 & 0 & 0 & \ldots & 0 \\ 0 & g_{n-k} & g_{n-k-1} & \cdots & g_1 & g_0 & 0 & 0 & \ldots & 0 \\ & & & \vdots & & & & & & \\ 0 & 0 & \ldots & 0 & g_{n-k} & g_{n-k-1} & \cdots & g_1 & g_0 & 0 \\ 0 & 0 & 0 & \ldots & 0 & g_{n-k} & g_{n-k-1} & \cdots & g_1 & g_0 \end{pmatrix}$$

- *So, clearly, it is a linear code. Each row of the generator matrix is just a shifted version of the first row. Unlike Hamming Code.*
- *Why is it a cyclic code?*

Example

- The cyclic shift of $C(x) = x^6 + x^4 + x + 1$ is $C^1(x) = x^5 + x^2 + x + 1$.
- It is still a code polynomial, because the code polynomial is $m(x) = x^2 + 1$.

Cyclic Code

- Given a code polynomial

$$C(x) = C_{n-1}x^{n-1} + C_{n-2}x^{n-2} + \ldots + C_1 x + C_0$$

- *We have*

$$xC(x) = C_{n-1}(x^n - 1) + C_{n-2}x^{n-1} + \ldots + C_1 x^2 + C_0 x + C_{n-1} = C_{n-1}(x^n - 1) + C^1(x).$$

- *Therefore, $C^1(x)$ is the cyclic shift of $C(x)$ and*
  - *has a degree of no more than n-1*
  - *divides $g(x)$ (why?) hence is a code polynomial.*

Cyclic Code

- To generate a cyclic code is to find a polynomial that
  - has degree n-k
  - is a factor of $x^n - 1$.

Generating Systematic Cyclic Code

- A systematic code means that the first *k* bits are the data bits and the rest *n-k* bits are parity checking bits.
- To generate it, we let

$$C(x) = m(x)x^{n-k} - r(x),$$

where

$$r(x) = m(x)x^{n-k} \bmod g(x).$$

- *The claim is that C(x) must divide g(x) hence is a code polynomial.*
  - *33 mod 7 = 5. Hence 33-5=28 can be divided by 7.*

Example

- A (7,4) cyclic code with $g(x) = x^3 + x + 1$.
- If $m(x) = x^3 + 1$, the non-systematic code is $C(x) = x^6 + x^4 + x + 1$.
- What is the systematic code?

Binary Function Expression

- So far have seen to possible ways
  - Binary equations
  - Truth tables
- What other ways are there?

Standard Forms

- Facilitate simplification
- Result in more desirable implementations
- Standard Forms rely on two type of terms
  - *Product Terms* – Terms that are ANDed together
    - » XYZ
    - » (A+B)(C+D)(A+D)
  - *Sum Terms* – Terms that are ORed together
    - » X+Y+Z
    - » XYZ + VX

Minterms

- Boolean Functions can be defined by truth tables. In a Boolean function, a product term in which all the variables appear is called a minterm of the function.
- Minterms specify the function as an OR of the minterms (product terms).

Minterm for 3 variables

- Table 2-6 from text

TABLE 2-6
Minterms for Three Variables

X	Y	Z	Product Term	Symbol	m_0	m_1	m_2	m_3	m_4	m_5	m_6	m_7
0	0	0	$\bar{X}\bar{Y}\bar{Z}$	m_0	1	0	0	0	0	0	0	0
0	0	1	$\bar{X}\bar{Y}Z$	m_1	0	1	0	0	0	0	0	0
0	1	0	$\bar{X}Y\bar{Z}$	m_2	0	0	1	0	0	0	0	0
0	1	1	$\bar{X}YZ$	m_3	0	0	0	1	0	0	0	0
1	0	0	$X\bar{Y}\bar{Z}$	m_4	0	0	0	0	1	0	0	0
1	0	1	$X\bar{Y}Z$	m_5	0	0	0	0	0	1	0	0
1	1	0	$XY\bar{Z}$	m_6	0	0	0	0	0	0	1	0
1	1	1	XYZ	m_7	0	0	0	0	0	0	0	1

Minterms for n variable functions

- For 2 variables have 4 minterms
  - X'Y' X'Y XY' XY
- For 3 variables have 8 minterms
  - X'Y'Z' X'Y'Z … XYZ
- In general, if a function has n variables there are 2^n minterms
- The subscript on the minterm is the decimal of the binary value represented

Maxterms

- A sum term that contains all the variables in complemented or un-complemented form is called a maxterm.
- As before, if there a n variables then there are 2^n maxterms.

Maxterm table

- The maxterms

TABLE 2-7
Maxterms for Three Variables

X	Y	Z	Sum Term	Symbol	M_0	M_1	M_2	M_3	M_4	M_5	M_6	M_7
0	0	0	$X+Y+Z$	M_0	0	1	1	1	1	1	1	1
0	0	1	$X+Y+\overline{Z}$	M_1	1	0	1	1	1	1	1	1
0	1	0	$X+\overline{Y}+Z$	M_2	1	1	0	1	1	1	1	1
0	1	1	$X+\overline{Y}+\overline{Z}$	M_3	1	1	1	0	1	1	1	1
1	0	0	$\overline{X}+Y+Z$	M_4	1	1	1	1	0	1	1	1
1	0	1	$\overline{X}+Y+\overline{Z}$	M_5	1	1	1	1	1	0	1	1
1	1	0	$\overline{X}+\overline{Y}+Z$	M_6	1	1	1	1	1	1	0	1
1	1	1	$\overline{X}+\overline{Y}+\overline{Z}$	M_7	1	1	1	1	1	1	1	0

Specifying functions

- Functions can be specified in minterm or maxterm notation
- Minterm
  - $F(X,Y,Z) = \sum m(0,2,5,7)$
  - $= X'Y'Z' + X'YZ' + XY'Z + XYZ$
  - And then you can work on simplifying this
  - Or could have also had
  - $F(X,Y,Z) = m_0 + m_2 + m_5 + m_7$

More examples

- From text
  - $F(X,Y,Z)' = \sum m(1,3,4,6)$
  - $= (m_1+m_3+m_4+m_6)$
  - Or complementing both sides of the equation
  - $F(X,Y,Z) = (m_1+m_3+m_4+m_6)'$
  - By DeMorgan's
  - $= m_1' \cdot m_3' \cdot m_4' \cdot m_6'$
  - $= M_1 \cdot M_3 \cdot M_4 \cdot M_6$
  - As $m_j' = M_j$

And to continue

- Then have:
- $F(X,Y,Z) = M_1 \cdot M_3 \cdot M_4 \cdot M_6$
- $=(X+Y+Z')(X+Y'+Z')(X'+Y+Z)(X'+Y'+Z)$
- Another expression form for the function as a product of maxterms
- $F(X,Y,Z) = \prod M\,(1,3,4,6)$

Another example

- Express the function F(A,B,C) = AB+A'C in minterm notation
- First expand to where each term has all three variables in it.
- AND term with 1 to expand. For the 1st term 1= (C+C') and for the 2nd the 1 is (B+B')
- Now have
  - F(A,B,C)=AB(C+C') + A'C(B+B')
  - F(A,B,C)=ABC + ABC' + A'BC + A'B'C
  - $= m_7 + m_6 + m_3 + m_1$

Summary of important properties

- Most important properties of minterms:
  - There are 2^n minterms for n Boolean variables. These minterms can be generated from the binary numbers from 0 to 2^n -1
  - Any Boolean function can be expressed as a logical sum of minterms.
  - The complement of a function contains those minterms not included in the original function.
  - A function that contains all 2^n minterms is equal to a logical 1.

Expansion of another function

- Express E = Y' + X'Z' in minterm notation.
  - E = (X+X')Y' + X'Z'(Y+Y')
  - = XY'(Z+Z') + X'Y'(Z+Z')+X'YZ'+X'Y'Z'
  - = XY'Z+XY'Z'+X'Y'Z+X'Y'Z'+X'YZ'
  - = $m_5 + m_4 + m_1 + m_0 + m_2$
  - = $m_5 + m_4 + m_2 + m_1 + m_0$
- Text shows how to find the minterm expression using a truth table.

Sum-of-Products

- Starting with the minterm specification of a function
  - $F(X,Y,Z) = \sum m(2,3,4,7)$
  - $= (m_2+m_3+m_4+m_7)$
  - $= X'YZ' + X'YZ + XY'Z' + XYZ$
- Each minterm represents a product term and then we sum them to generate the function.
- This form is called *sum-of-products*.
- Even when in minimal form it is still the sum-of-products.

Producing sum-of-products

- Form other form of the function
  - F = AB + C(D + E)
  - Can distribute the C
  - F = AB + CD + CE
  - And now have the function in sum-of-products form.
  - The sum-of-products form is a 2 level implementation of the function in gates