

# Decimal Number System

- Base (aka *radix*) 10
  - › 10 digits = 0 through 9
  - › Positional notation number system
    - Left most digit is “most significant digit”
    - Right most digit is “least significant digit”
  - ›  $1,234.56 = 1,234.56_{10}$ 
    - $1 \times 1000 + 2 \times 100 + 3 \times 10 + 4 \times 1 + 5 \times 0.1 + 6 \times 0.01$
    - $1 \times 10^3 + 2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0 + 5 \times 10^{-1} + 6 \times 10^{-2}$
    - In general, for  $n$  digits:  $c_{n-1} \times 10^{n-1} + \dots + c_1 \times 10^1 + c_0 \times 10^0$ 
      - where  $c_i$  is the coefficient for the  $i^{\text{th}}$  power of 10
      - and  $c_i$  is a digit 0 through 9

# Binary Number System

- Base (or radix) 2
  - › Only 2 digits = 0 and 1
    - A binary digit is called a *bit*
  - › Positional notation number system
    - Left most bit is “most significant bit” or MSB
    - Right most bit is “least significant bit” or LSB
  - › In general, for  $n$  digits:  $c_{n-1}x2^{n-1} + \dots + c_1x2^1 + c_0x2^0$ 
    - where  $c_i$  is the coefficient for the  $i^{\text{th}}$  power of 2
    - and  $c_i$  is a digit 0 or 1
  - ›  $101010_2 =$ 
    - $1x2^5 + 0x2^4 + 1x2^3 + 0x2^2 + 1x2^1 + 0x2^0$
    - $1x32 + 0x16 + 1x8 + 0x4 + 1x2 + 0x1 = 42_{10}$
  - ›  $101.01_2 =$ 
    - $1x2^2 + 0x2^1 + 1x2^0 + 0x2^{-1} + 1x2^{-2}$
    - $1x4 + 0x2 + 1x1 + 0x\frac{1}{2} + 1x\frac{1}{4} = 5.25_{10}$
  - › Normally ignore leading 0s unless # of bits is specified
    - $00000101_2 = 101_2$
    - But if we are dealing with 8-bit values (for example) we keep all 8 bits

# Octal & Hexadecimal Number Systems

- Other useful number systems in digital design
  - › Base 8 (octal)
    - 8 digits: 0 through 7
    - $5310_8 =$ 
      - $5 \times 8^3 + 3 \times 8^2 + 1 \times 8^1 + 0 \times 8^0$
      - $5 \times 512 + 3 \times 64 + 1 \times 8 + 0 \times 1 = 2760_{10}$
  - › Base 16 (hexadecimal, or just ‘hex’ is the common slang)
    - 16 digits: 0 through 9 and A through F
      - Digits A - F represent decimal values 10 - 15, respectively
    - $A1C_{16} =$ 
      - $A \times 16^2 + 1 \times 16^1 + C \times 16^0$
      - $10 \times 256 + 1 \times 16 + 12 \times 1 = 2588_{10}$
  - › Leading 0s also apply to octal and hex values

# Octal & Hexadecimal

- Octal & hexadecimal are “power of 2” numbers

- ›  $8 = 2^3 \Rightarrow 1 \text{ octal digit} = 3 \text{ bits}$
  - ›  $16 = 2^4 \Rightarrow 1 \text{ hex digit} = 4 \text{ bits}$

- Easy conversion between binary and octal or hexadecimal

- › Simply group digits & substitute
    - Groups of 3 bits for octal digit
    - Groups of 4 bits for hex digit
  - ›  $011011101001_2 =$ 
    - $011\ 011\ 101\ 001 = 3351_8$
    - $0110\ 1110\ 1001 = 6E9_{16}$

binary	octal	binary	hex
		0000	0
		0001	1
		0010	2
		0011	3
000	0	0100	4
001	1	0101	5
010	2	0110	6
011	3	0111	7
100	4	1000	8
101	5	1001	9
110	6	1010	A
111	7	1011	B
		1100	C
		1101	D
		1110	E
		1111	F

# Conversion Between Systems

## Conversion of decimal to:

### > binary

- $35_{10} = 100011_2$ 
  - $35 \div 2 = 17$  remainder 1 = odd (LSB)
  - $17 \div 2 = 8$  remainder 1 = odd
  - $8 \div 2 = 4$  remainder 0 = even
  - $4 \div 2 = 2$  remainder 0 = even
  - $2 \div 2 = 1$  remainder 0 = even
  - $1 \div 2 = 0$  remainder 1 = odd (MSB)
- Divide by 2 & look at:
  - 0/1 remainder, or
  - even/odd dividend

### > octal

- $35_{10} = 43_8$ 
  - $35 \div 8 = 4$  remainder 3 (LSB)
  - $4 \div 8 = 0$  remainder 4 (MSB)

### > hexadecimal

- $35_{10} = 23_{16}$ 
  - $35 \div 16 = 2$  remainder 3 (LSB)
  - $2 \div 16 = 0$  remainder 2 (MSB)

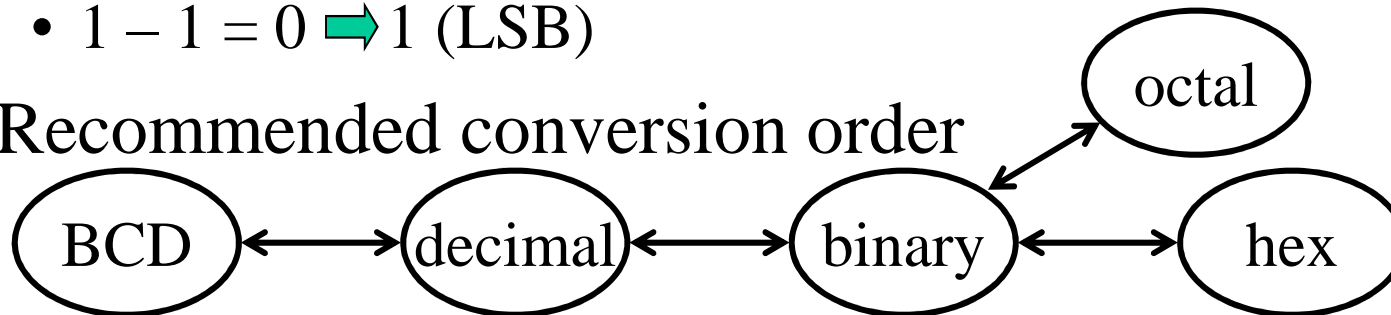
# Conversion Between Systems

Another conversion technique:

- >  $35_{10} = 0100011_2$ 
  - $35 - 32 = 3 \Rightarrow 1$  (MSB)
  - $3 - 16 < 0 \Rightarrow 0$
  - $3 - 8 < 0 \Rightarrow 0$
  - $3 - 4 < 0 \Rightarrow 0$
  - $3 - 2 = 1 \Rightarrow 1$
  - $1 - 1 = 0 \Rightarrow 1$  (LSB)

$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$
64	32	16	8	4	2	1
0	1	0	0	0	1	1

> Recommended conversion order



# Binary Arithmetic

## *Addition*

Decimal	Binary	Octal	Hex
<i>carry</i> 10	<i>carry</i> 10111100	<i>carry</i> 11	<i>carry</i> 11
190	10111110	276	BE
+141	+10001101	+215	+8D
<hr/> 331	<hr/> 101001011	<hr/> 513	<hr/> 14B

## *Subtraction*

Decimal	Binary	Octal	Hex
<i>borrow</i> 1	<i>borrow</i> 1	<i>borrow</i>	<i>borrow</i>
190	10111110	276	BE
-141	-10001101	-215	-8D
<hr/> 49	<hr/> 00110001	<hr/> 61	<hr/> 31

# Binary Arithmetic

## *Multiplication*

Decimal	Binary
190	10111110
<u>x141</u>	<u>x10001101</u>
190	10111110
760	00000000
<u>190</u>	10111110
26790	10111110
	00000000
	00000000
	00000000
	<u>10111110</u>
	110100010100110

## *Division*

Decimal	Binary
12	1100
14 $\overline{)177}$	1110 $\overline{)10110001}$
<u>14</u>	<u>1110</u>
37	10000
<u>28</u>	<u>1110</u>
9	1001

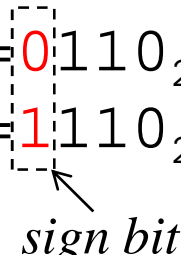
# Binary Negative Numbers

- Negative number representations for binary arithmetic operations

- › Signed magnitude

- Add a MSB to represent positive (0) or negative (1)

$$\begin{array}{l}
 6_{10} = 0110_2 \\
 -6_{10} = 1110_2
 \end{array}$$


  
 sign bit

- › 1s complement (aka *diminished radix complement*)

- Invert the binary representation for the number
      - 1s become 0s & 0s become 1s

$$\begin{array}{l}
 6_{10} = 0110_2 \\
 -6_{10} = 1001_2
 \end{array}$$

- › 2s complement (aka *radix complement*)

- Take 1s complement and add 1 (00...001)
    - Alternative method:
      - right to left: copy bits through 1<sup>st</sup> logic 1, then invert

$$\begin{array}{l}
 6_{10} = 0110_2 \\
 -6_{10} = 1010_2
 \end{array}$$

- › MSB in 1s & 2s complement is sign value (0 = +, 1 = -)

- A specified # bits & leading 0s often used in binary arithmetic
      - This will fix sign bit position

# Addition & Subtraction

- Addition (& subtraction) with 2s complement
  - › Most often used in hardware implementations of subtractors

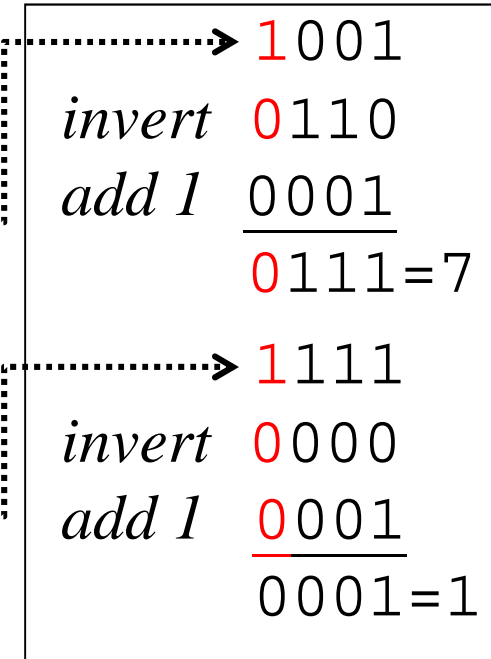
$$\begin{array}{r}
 +3_{10} = 0011_2 \\
 + +4_{10} = 0100_2 \\
 \hline
 +7_{10} = 0111_2
 \end{array}$$

$$\begin{array}{r}
 -3_{10} = 1101_2 \\
 + -4_{10} = 1100_2 \\
 \hline
 -7_{10} = 1001_2
 \end{array}$$

$$\begin{array}{r}
 -3_{10} = 1101_2 \\
 + +4_{10} = 0100_2 \\
 \hline
 +1_{10} = 0001_2
 \end{array}$$

$$\begin{array}{r}
 +3_{10} = 0011_2 \\
 + -4_{10} = 1100_2 \\
 \hline
 -1_{10} = 1111_2
 \end{array}$$

$$\begin{array}{r}
 +3_{10} = 0011_2 \\
 + -3_{10} = 1101_2 \\
 \hline
 +0_{10} = 0000_2
 \end{array}$$



*2s complement  
operation to show  
positive result*

# Addition & Subtraction

- Addition (& subtraction) with 1s complement
  - › End-around-carry is difficult to implement in hardware

$  \begin{array}{r}  +3_{10} = 0011_2 \\  + +4_{10} = 0100_2 \\  \hline  +7_{10} = 0111_2  \end{array}  $	$  \begin{array}{r}  -3_{10} = 1100_2 \\  + -4_{10} = 1011_2 \\  \hline  -7_{10} = 10111_2  \end{array}  $
	$1000_2$
<i>end-around-carry</i>	
$  \begin{array}{r}  -3_{10} = 1100_2 \\  + +4_{10} = 0100_2 \\  \hline  +1_{10} = 10000_2  \end{array}  $	$  \begin{array}{r}  +3_{10} = 0011_2 \\  + -4_{10} = 1011_2 \\  \hline  -1_{10} = 1110_2  \end{array}  $
	$1110$
<i>end-around-carry</i>	
$  \begin{array}{r}  -3_{10} = 1100_2 \\  + +3_{10} = 0011_2 \\  \hline  -0_{10} = 1111_2 = 0000_2 = +0_{10}  \end{array}  $	

$\rightarrow 1000$   
*invert 0111 = 7*

$\rightarrow 1110$   
*invert 0001 = 1*

*1s complement to see result*

# Binary Code Words

- Given  $N$  bits, we can have  $K = 2^N$  different combinations of 0s and 1s
  - ›  $N$  bits can uniquely represent  $K = 2^N$  different items
  - › Adding one bit doubles the number of items
- Given  $K$  items, we need  $N = \lceil \log_2 K \rceil$  bits to obtain a unique binary representation for each item
  - ›  $\lceil x \rceil$  = ceiling function of  $x \Rightarrow$  round-up to next higher integer
    - $\lceil 5.1 \rceil = 6$
    - $\lceil 5.0 \rceil = 5$
- A collection of  $n$  bits is often called a *data word*
  - › 8-bits called a *byte*
  - › 4-bits called a *nibble*

# ASCII Code

American Standard Code for Information Interchange – ASCII code

ASCII	MSBs - $b_6b_5b_4$							
$b_3b_2b_1b_0$	000 - 0	001 - 1	010 - 2	011 - 3	100 - 4	101 - 5	110 - 6	111 - 7
0000 - 0	NUL	DLE	SP	0	@	P	`	p
0001 - 1	SOH	DC1	!	1	A	Q	a	q
0010 - 2	STX	DC2	“	2	B	R	b	r
0011 - 3	ETX	DC3	#	3	C	S	c	s
0100 - 4	EOT	DC4	\$	4	D	T	d	t
0101 - 5	ENQ	NAK	%	5	E	U	e	u
0110 - 6	ACK	SYN	&	6	F	V	f	v
0111 - 7	BEL	ETB	‘	7	G	W	g	w
1000 - 8	BS	CAN	(	8	H	X	h	x
1001 - 9	HT	EM	)	9	I	Y	i	y
1010 - A	LF	SUB	*	:	J	Z	j	z
1011 - B	VT	ESC	+	;	K	[	k	{
1100 - C	FF	FS	,	<	L	\	l	
1101 - D	CR	GS	-	=	M	]	m	}
1110 - E	SO	RS	.	>	N	^	n	~
1111 - F	SI	US	/	?	O	_	o	DEL

# Other Binary Codes

Other codes sometimes encountered:

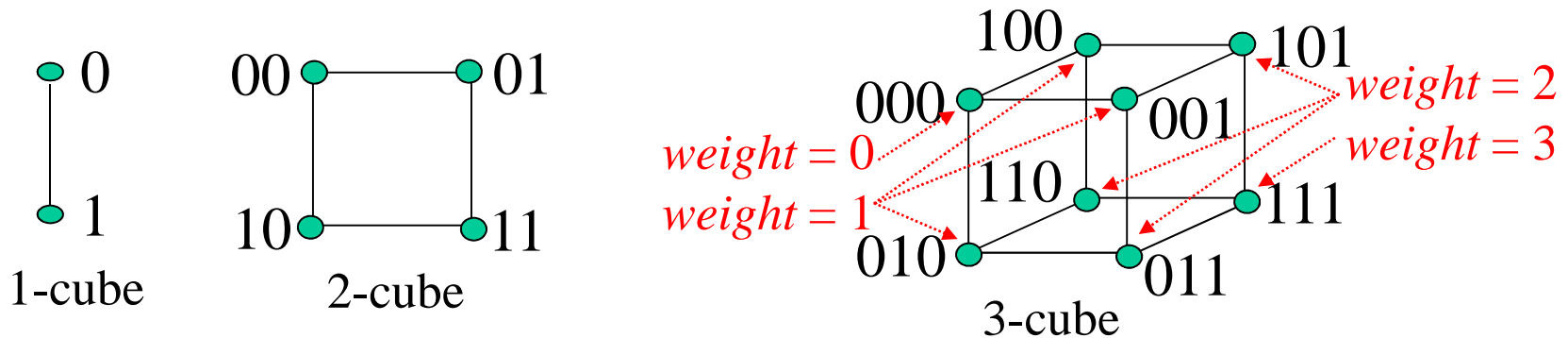
- Binary Coded Decimal (BCD)
  - › Uses binary representation for digits 0-9
  - › Binary digits grouped in groups of 4 bits
  - ›  $53_{10} = 0101\ 0011_{\text{BCD}}$
- Gray code
  - › As we move through consecutive binary values, only one bit changes
- These & other codes have limited uses
  - › Error detection in data transmission

gray	dec
000	0
001	1
011	2
010	3
110	4
111	5
101	6
100	7

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

# Cubes, Distance & Weight

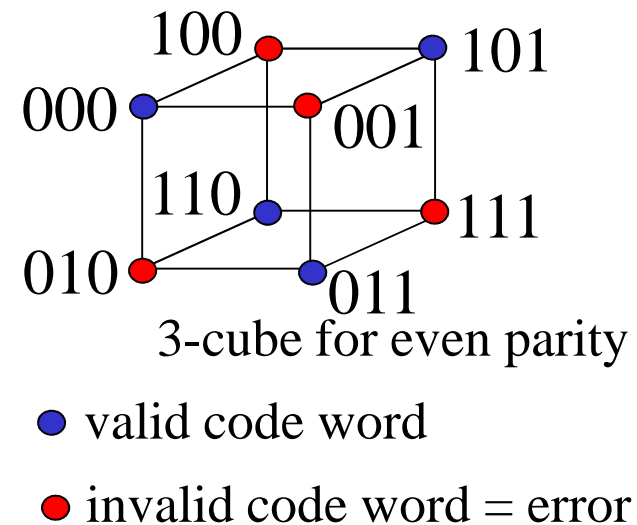
- $N$ -cubes
  - › Each binary value is a *vertex* (aka *node*) in a graph
    - An  $N$ -cube has  $2^N$  vertices
  - › Edges connect vertices that differ by 1 bit
    - Each vertex has edges to  $N$  other vertices
- **Distance** is number of edges between two vertices
  - › The number of bits different between two binary values
- **Weight** is the number of 1s in the binary value



# Error Detection Codes

- Error detection codes use distance to detect data transmission errors
  - › Add bits to create code words with distance =  $n$
  - › Example: parity (distance = 2) detects single bit errors
    - Requires 1 additional bit (parity bit)
    - Even parity has even # 1s in code word (data + parity bit)
    - Odd parity has odd # 1s

Data Word	Code Word	
	even parity	odd parity
00	00 0	00 1
01	01 1	01 0
10	10 1	10 0
11	11 0	11 1



# Error Detection Codes

- Checksum error detection codes
  - › Sum subsets of data bits
    - assuming positive binary values
  - › Some types of checksum
    - Single precision – ignore carry-out
      - $N$ -bit code word
    - Residue – end-around-carry
      - Carry added as an LSB
      - $N$ -bit code word
    - Double precision – retain all carries
      - $2N$ -bit code word
    - There are other checksums
- There are other error detection codes

```
Single precision
0110 data0
1100 data1
0101 data2
0111 checksum
```

```
Residue
0110 data0
1100 data1
0101 data2
1000 checksum
```

```
Double precision
 0110 data0
 1100 data1
 0101 data2
00010111 checksum
```

# Error Detection Codes

- Other error detection (& correction) codes:
  - › Berger code (Bose-Lin code is similar)
    - Code word based on the number of 1s in data word
      - Inverted count of the number of 1s
  - › Cyclic Redundancy Check (CRC)
    - Code word based on polynomial division
      - Remainder of the data polynomial  $\div$  CRC polynomial
  - › Hamming code
    - Based on multiple parity codes
      - Detects all single & double bit errors
      - Corrects all single bit errors