Digital Logic Circuits
'*Number Systems*'

ELEC2200

Summer 2009

David J. Broderick
brodedj@auburn.edu
http://www.auburn.edu/~brodedj
Office: Broun 360

Decimal – our 'goto' representation

- Whole Numbers
  - $13_{10}=1*(10^{1})+3*(10^{0})$
- Real Numbers
  - $143.79_{10}=1*(10^{2})+4*(10^{1})+3*(10^{0})+7*(10^{-1})+9*(10^{-2})$
- Base (or radix) of 10
- Difficult to implement in hardware
- Susceptible to noise

Binary – A Computer's Representation

- Devices have two states, on and off
- Easier to implement and less prone to noise
- A base of 2
- Whole
  - $1101_2=1*(2^3)+1*(2^2)+0*(2^1)+1*(2^0)=13_{10}$
- Real
  - $10.11_2=1*(2^1)+0*(2^0)+1*(2^{-1})+1*(2^{-2})=2.75_2$

Why Hex/Octal?

- Decimal is our natural number system
- Binary works well for electronic representation
- Where do Hexadecimal and Octal fit in?
  - Trade off between ease of interpretation and efficient use of storage

Octal and Hexadecimal

- Octal (Base 8)
  - $127.1_8=1*(8^2)+2*(8^1)+7*(8^0)+1*(8^{-1})=87.125_{10}$
- Hexadecimal (Base 16)
  - 16 digits, 0 through F
  - $1A6.4_{16}=1*(16^2)+10*(16^1)+6*(16^0)+4*(16^{-1})=422.25_{10}$

Number Bases

- Counting:

Dec	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Oct	0	1	2	3	4	5	6	7	10	11	12	13	14	15	16	17	20
Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10

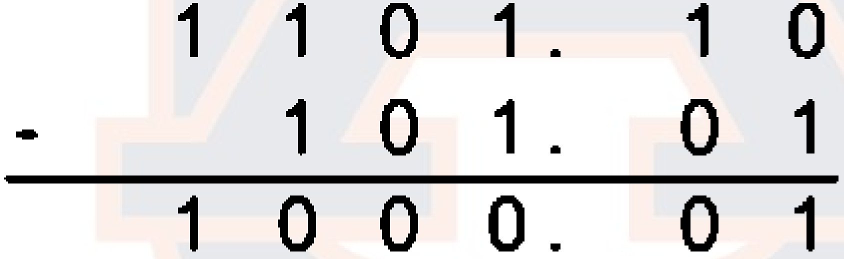
Addition

- We can perform arithmetic in any base just like we do in base 10
- $13.5_{10} + 5.25_{10} = 18.75_{10}$

$$\begin{array}{r}
1101.10 \\
+\quad 101.01 \\
\hline
10010.11
\end{array}$$

Subtraction

- We can perform arithmetic in any base just like we do in base 10
- $13.5_{10}-5.25_{10}=8.25_{10}$



```
    1 1 0 1.  1 0
-     1 0 1.  0 1
-----------------
    1 0 0 0.  0 1
```

Octal/Hex Math

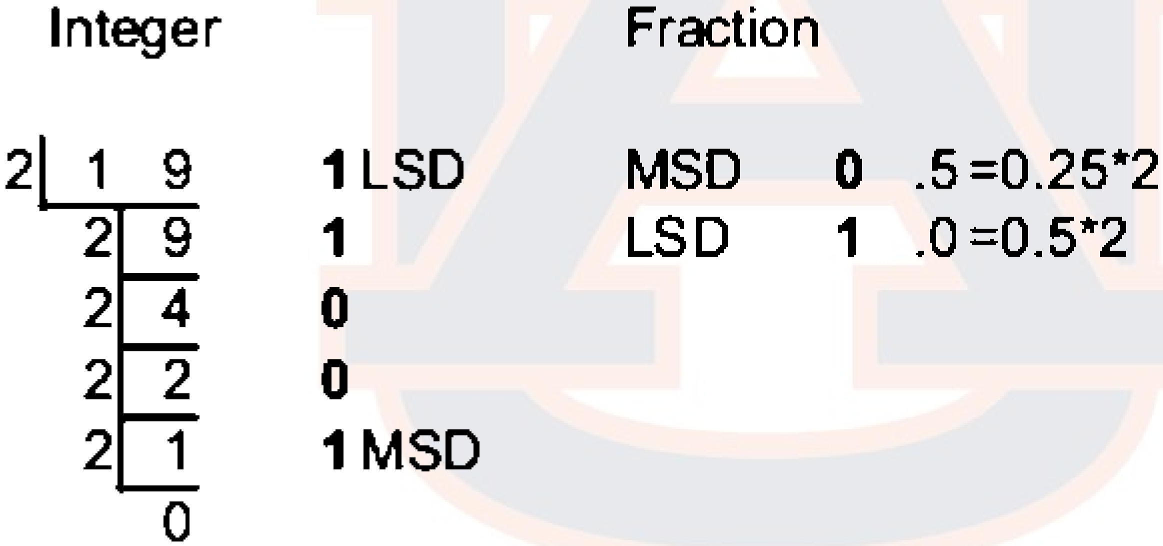
- Add octal/hex add/sub examples
-

Base Conversion

- Binary-to-Decimal
  - $1001.01_2=1*(2^3)+0*(2^2)+0*(2^1)+1*(2^0)+0*(2^{-1})+1*(2^{-2})=8_{10}+1_{10}+.25_{10}=9.25_{10}$
- But were not limited to these two base values
- Octal – Base 8
  - $1001.01_2=1*(2^3)+0*(2^2)+0*(2^1)+1*(2^0)+0*(2^{-1})+1*(2^{-2})=10_8+1_8+.2_8=11.2_8$
- 16 digits written as 0 to F
- Hexadecimal(Hex) – Base 16
  - $1001.01_2=1*(2^3)+0*(2^2)+0*(2^1)+1*(2^0)+0*(2^{-1})+1*(2^{-2})=8_{16}+1_{16}+.4_{16}=9.4_{16}$

Base Conversion

- Decimal-to-Binary (Divide/Multiply by Radix)
  - 19.25_{10}

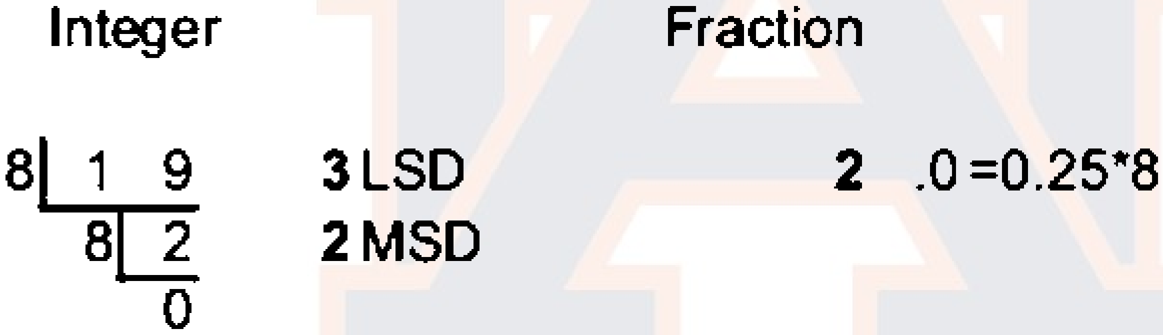


```
        Integer                           Fraction

2| 1  9       1 LSD            MSD     0   .5 =0.25*2
    2| 9      1                LSD     1   .0 =0.5*2
    2| 4      0
    2| 2      0
    2| 1      1 MSD
       0
```

Base Conversion

- Decimal-to-Octal (Divide/Multiply by Radix)
  - 19.25_{10}



Integer

Fraction

8 | 1 9 — 3 LSD

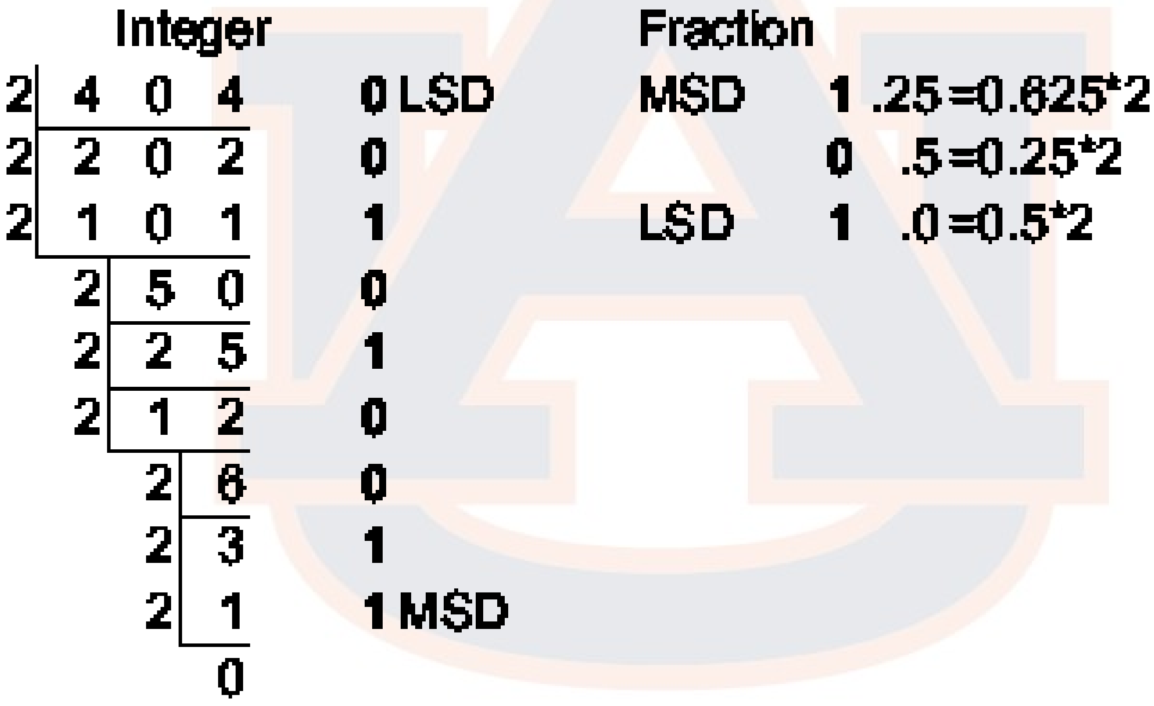
8 | 2 — 2 MSD

0

2 .0 =0.25*8

Decimal to Binary

- 404.625_{10}



Integer

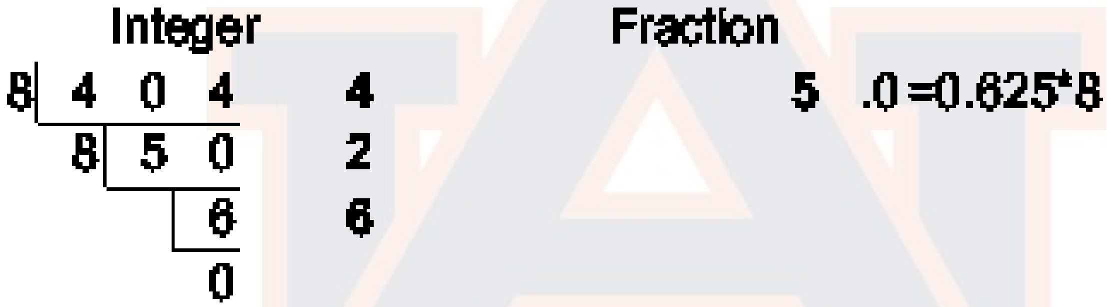
2	404	0	LSD
2	202	0	
2	101	1	
2	50	0	
2	25	1	
2	12	0	
2	6	0	
2	3	1	
2	1	1	MSD
	0		

Fraction

MSD	1	.25	=0.625*2
	0	.5	=0.25*2
LSD	1	.0	=0.5*2

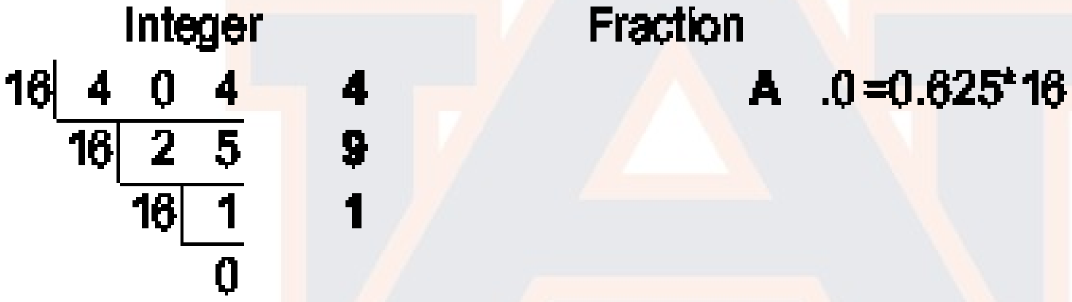
Decimal to Octal

- 404.625_{10}



Decimal to Hex

- 404.625_{10}



```
          Integer                              Fraction
16| 4  0  4          4                   A   .0 =0.625*16
   -------
   16| 2  5          9
      ----
      16| 1          1
         ---
            0
```

Uses For Hex/Octal

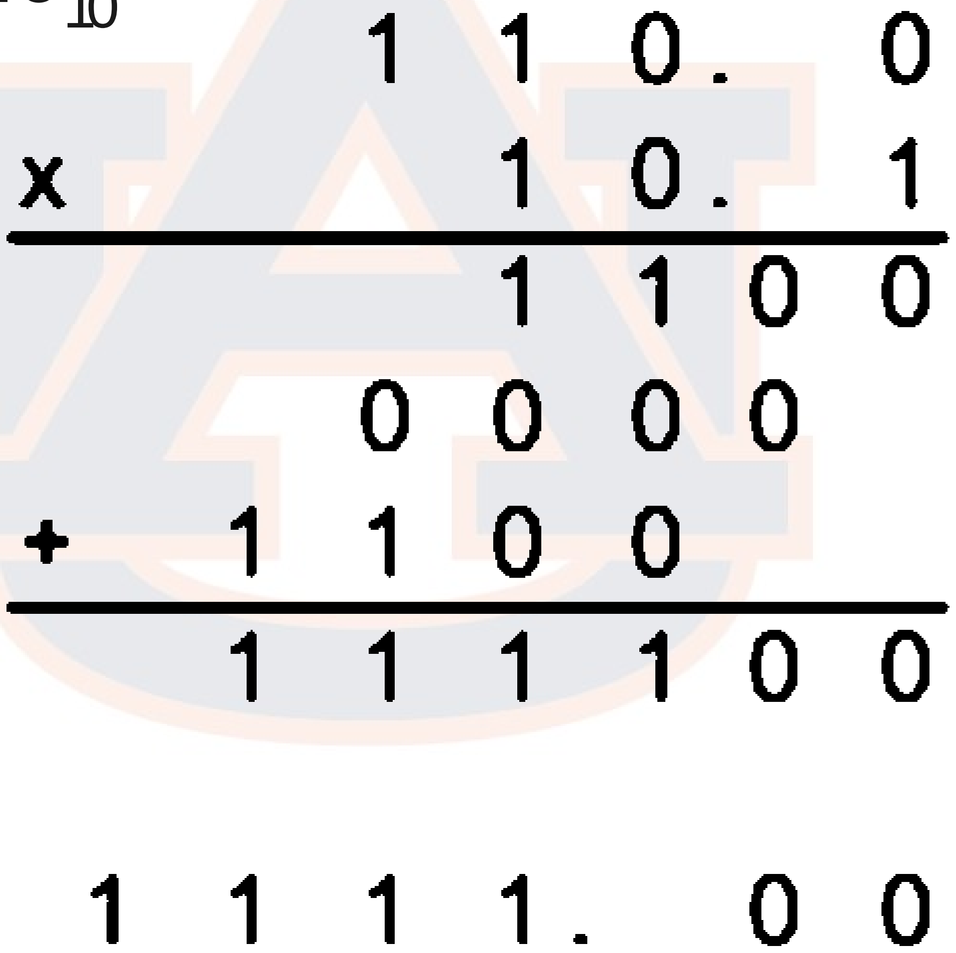
- Both of these base values allow compact representation of binary values
- 404.625_{10} $=110010100.101_2$
- Octal – Group binary digits into threes
  - $110\ 010\ 100.101_2$=$6\ 2\ 4.5_8$
- Hex – Group binary digits into fours
  - $0001\ 1001\ 0100.1010_2$=$1\ 9\ 4.A_{16}$

Why does this work?

- Base values of 8 and 16 are powers of the original 2
- 8 = 2^3, Octal groups binary digits by 3
- 16 = 2^4, Hex groups binary digits by 4
- Works for other values as long as one base is multiple of another
- However, these are the most commonly used

Multiplication

- $6_{10}*2.5_{10}=15.0_{10}$

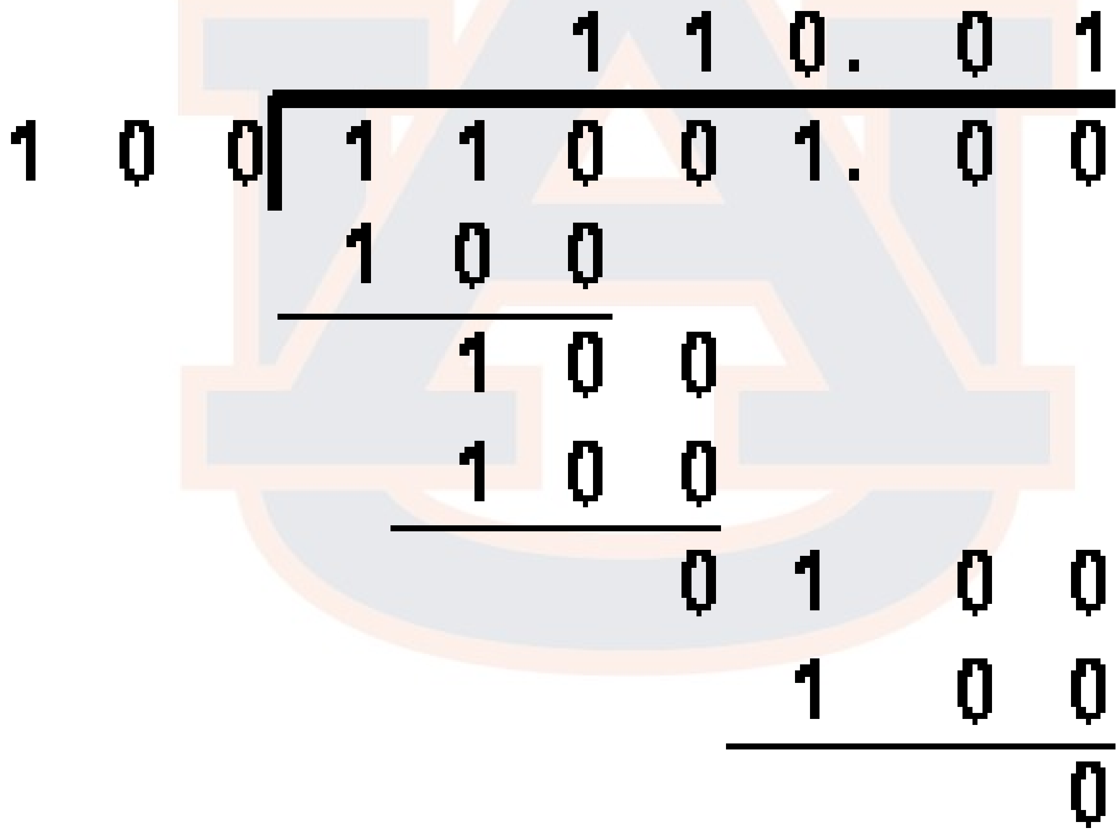


```
      1 1 0.  0
x       1 0.  1
---------------
        1 1 0 0
      0 0 0 0
+   1 1 0 0
---------------
    1 1 1 1 0 0

  1 1 1 1.  0 0
```

Division

- $25.0_{10} / 4_{10} = 6.25_{10}$



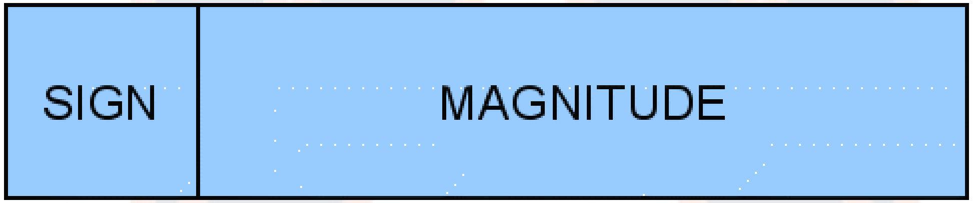
```
                 1  1  0. 0  1
          ____________________
 1  0  0 | 1  1  0  0  1. 0  0
           1  0  0
          --------
              1  0  0
              1  0  0
             ---------
                    0  1  0  0
                       1  0  0
                      --------
                             0
```

Negative Numbers

- Whole numbers are only half of the number line
- How do we represent negative numbers?
  - Sign/Magnitude
  - 1's Compliment
  - 2's Compliment
  - Floating Point

Sign Magnitude

- An additional bit is used to indicate sign
- Using small values:


SIGN
MAGNITUDE

- 0101=+5
- 1101=-5
- For this example: Maximum value,7
- Minimum value of -7
- 15 values total, two ways to represent zero

1's Complement

- Negative Number is the inverse (compliment) of the positive value
- $0101_2=+5$
- $1010_2=-5$
- For this example: Maximum value,7
- Minimum value of -7
- 15 values total, two ways to represent zero

1's Comp Arithmetic

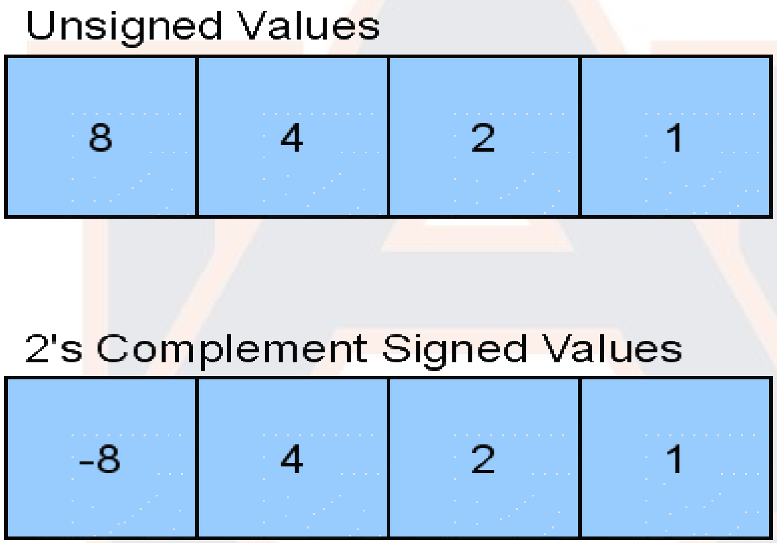
- End around carry is cumbersome to implement
- Show example here

2's Complement

- Overflow/Carry
- Negative value is inverse of positive, plus 1
- 0101_2=+5
- 1011_2=-5
- For this example: Maximum value,7
- Minimum value of -8
- Only 1 way to represent zero

2's Complement

- MSB still indicates sign (0=positive,1=negative)



Unsigned Values

8	4	2	1

2's Complement Signed Values

-8	4	2	1

- Smallest value: 1000_2=-8_{10}
- Largest value: 0111_2=$+7_{10}$
- +0 = -0

2's Comp. Addition

+3		0	0	1	1
+4	+	0	1	0	0
+7		0	1	1	1

+3		0	0	1	1
-4	+	1	1	0	0
-1		1	1	1	1

-3		1	1	0	1
+4	+	0	1	0	0
+1		0	0	0	1

-3		1	1	0	1
-4	+	1	1	0	0
-7		1	0	0	1

Negative # Summary Sign/Magnitude

- Pros
  - A reasonable first choice
  - Easiest for human interpretation

- Cons
  - Requires two separate circuits (add/subtract)
  - Must add logic to decide which to use
  - 1 less value than comparable unsigned number
  - 2 zeros

Negative # Summary One's Complement

- Pros
  - Add and Subtract now implemented in one circuit
- Cons
  - End-around carry still difficult to implement
  - Not as easily interpreted by humans (But do we care?)
  - 1 less value than comparable unsigned number
  - 2 zeros

Negative # Summary
Two's Complement

- Pros
  - Add and Subtract implemented in one circuit
  - No end-around carry
  - No wasted values
  - One zero representation

- Cons
  - Still hard to interpret (But do we care?)

Other uses for Binary

- Real values are useful but binary values are not limited to numbers
- Can be used to encode symbols
  - BCD: Assign decimal digits to binary equivalent
  - ASCII: A representation of text
  - Gray Code: A variety of uses

Binary Coded Decimal (BCD)

- If we only store values in binary, in what format can they be interpreted most readily?
- Represent each decimal digit as a collection of binary values.
- How many binary bits does it take to represent decimal digits 0 to 9?
- Is this an efficient use of storage?

BCD Table

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

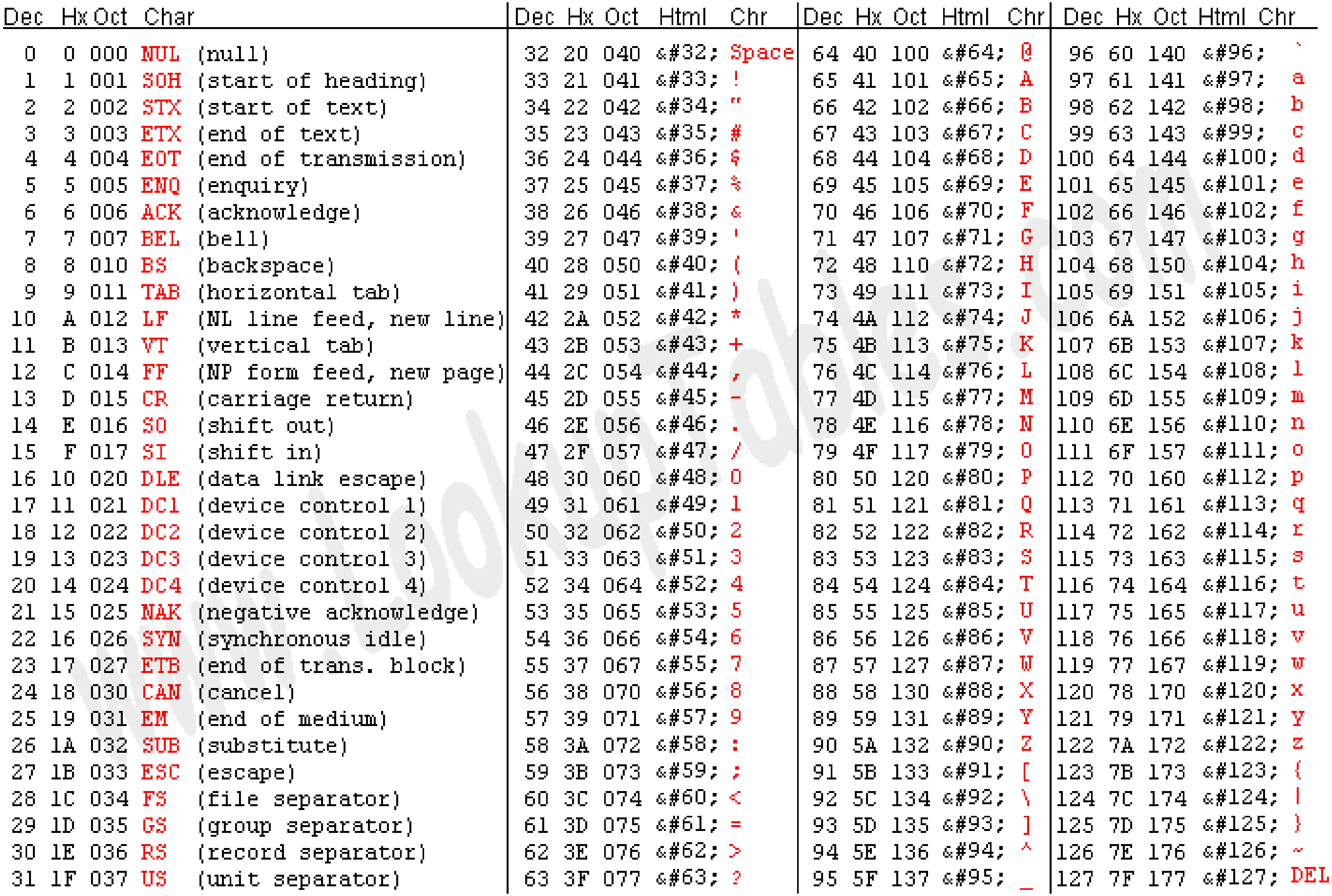
American Standard Code for Information Inter change (ASCII)

- What if we want to transmit a letter?
- 26 letters (English at least), how many bits do we need to encode?
  - $K=2^N$, K - # of codes, N - # of bits
- K=26 here, solving for N:
  - $N=LOG_2 K$
  - $LOG_2 26=4.70043...$
- Which way do we round?
  - $K=2^4=16$, $K=2^5=32$
  - UP!

American Standard Code for Information Exchange (ASCII)

- ASCII defines an 8 bit (1 byte) code word
- $K=2^8=256$ code words

American Standard Code for Information Exchange (ASCII)



Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	&#32;	Space	64	40	100	&#64;	@	96	60	140	&#96;	`
1	1	001	SOH (start of heading)	33	21	041	&#33;	!	65	41	101	&#65;	A	97	61	141	&#97;	a
2	2	002	STX (start of text)	34	22	042	&#34;	"	66	42	102	&#66;	B	98	62	142	&#98;	b
3	3	003	ETX (end of text)	35	23	043	&#35;	#	67	43	103	&#67;	C	99	63	143	&#99;	c
4	4	004	EOT (end of transmission)	36	24	044	&#36;	$	68	44	104	&#68;	D	100	64	144	&#100;	d
5	5	005	ENQ (enquiry)	37	25	045	&#37;	%	69	45	105	&#69;	E	101	65	145	&#101;	e
6	6	006	ACK (acknowledge)	38	26	046	&#38;	&	70	46	106	&#70;	F	102	66	146	&#102;	f
7	7	007	BEL (bell)	39	27	047	&#39;	'	71	47	107	&#71;	G	103	67	147	&#103;	g
8	8	010	BS (backspace)	40	28	050	&#40;	(	72	48	110	&#72;	H	104	68	150	&#104;	h
9	9	011	TAB (horizontal tab)	41	29	051	&#41;	)	73	49	111	&#73;	I	105	69	151	&#105;	i
10	A	012	LF (NL line feed, new line)	42	2A	052	&#42;	*	74	4A	112	&#74;	J	106	6A	152	&#106;	j
11	B	013	VT (vertical tab)	43	2B	053	&#43;	+	75	4B	113	&#75;	K	107	6B	153	&#107;	k
12	C	014	FF (NP form feed, new page)	44	2C	054	&#44;	,	76	4C	114	&#76;	L	108	6C	154	&#108;	l
13	D	015	CR (carriage return)	45	2D	055	&#45;	-	77	4D	115	&#77;	M	109	6D	155	&#109;	m
14	E	016	SO (shift out)	46	2E	056	&#46;	.	78	4E	116	&#78;	N	110	6E	156	&#110;	n
15	F	017	SI (shift in)	47	2F	057	&#47;	/	79	4F	117	&#79;	O	111	6F	157	&#111;	o
16	10	020	DLE (data link escape)	48	30	060	&#48;	0	80	50	120	&#80;	P	112	70	160	&#112;	p
17	11	021	DC1 (device control 1)	49	31	061	&#49;	1	81	51	121	&#81;	Q	113	71	161	&#113;	q
18	12	022	DC2 (device control 2)	50	32	062	&#50;	2	82	52	122	&#82;	R	114	72	162	&#114;	r
19	13	023	DC3 (device control 3)	51	33	063	&#51;	3	83	53	123	&#83;	S	115	73	163	&#115;	s
20	14	024	DC4 (device control 4)	52	34	064	&#52;	4	84	54	124	&#84;	T	116	74	164	&#116;	t
21	15	025	NAK (negative acknowledge)	53	35	065	&#53;	5	85	55	125	&#85;	U	117	75	165	&#117;	u
22	16	026	SYN (synchronous idle)	54	36	066	&#54;	6	86	56	126	&#86;	V	118	76	166	&#118;	v
23	17	027	ETB (end of trans. block)	55	37	067	&#55;	7	87	57	127	&#87;	W	119	77	167	&#119;	w
24	18	030	CAN (cancel)	56	38	070	&#56;	8	88	58	130	&#88;	X	120	78	170	&#120;	x
25	19	031	EM (end of medium)	57	39	071	&#57;	9	89	59	131	&#89;	Y	121	79	171	&#121;	y
26	1A	032	SUB (substitute)	58	3A	072	&#58;	:	90	5A	132	&#90;	Z	122	7A	172	&#122;	z
27	1B	033	ESC (escape)	59	3B	073	&#59;	;	91	5B	133	&#91;	[	123	7B	173	&#123;	{
28	1C	034	FS (file separator)	60	3C	074	&#60;	<	92	5C	134	&#92;	\	124	7C	174	&#124;	|
29	1D	035	GS (group separator)	61	3D	075	&#61;	=	93	5D	135	&#93;	]	125	7D	175	&#125;	}
30	1E	036	RS (record separator)	62	3E	076	&#62;	>	94	5E	136	&#94;	^	126	7E	176	&#126;	~
31	1F	037	US (unit separator)	63	3F	077	&#63;	?	95	5F	137	&#95;	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Gray Code

- Frank Gray – Bell Labs (1947)
- Used to reduce the effects of noise on analog values
- Used in circuit minimization (Karnaugh Maps)
- Discrete Optimization
- Not limited to these uses

Gray Code

- Hamming Distance: The minimum number of substitutions to change one value to another.

1	0	1	1
0	0	1	0

- Hamming distance equals 2
- Code is a reordering of values so that similar values are separated by a hamming distance of one

Gray Code

- A 2 bit Gray Code:

Decimal	Binary	Gray
0	00	00
1	01	01
2	10	11
3	11	10

- Notice the Gray Code has a hamming distance of one when wrapping from 3 to 0

Gray Code

- If transmitted value is corrupted, the interpreted value is only changed by one.
- 01_2 become 11_2
- Binary interpretation: 1_{10} becomes 3_{10}
- Gray Code: 1_{10} becomes 2_{10}

Gray Code

- We can extend this to larger values
- A 3 bit Gray Code
- 010_2 becomes 110_2
- Binary:2_{10} becomes 6_{10}
- Gray: 3_{10} becomes 4_{10}

Decimal	Binary	Gray
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

Error Detection/Correction

- Parity: an additional bit can be used to detect errors
- Given a byte (8 bits) a parity bit is added

d1	d2	d3	d4	d5	d6	d7	d8	P

- The weight of the value: The number of 1's in the binary value
- Even parity: P must be chosen to make the weight an even number
- Odd parity: P must be chosen to make the weight an odd number
- For every N bits sent, N+1 bits required
- Can't detect location of error

Parity Example

- Even Parity Example (valid transmission):

d1	d2	d3	d4	P
1	1	0	1	1

- Which transmissions are valid (even parity)?

d1	d2	d3	d4	P
1	0	0	0	1
0	1	1	1	0
0	0	1	0	1
0	1	1	0	1
1	1	1	1	1

Parity Example

- Even Parity Example (valid transmission):

d1	d2	d3	d4	P
1	1	0	1	1

- Which transmissions are valid (even parity)?

d1	d2	d3	d4	P
1	0	0	0	1
0	1	1	1	0
0	0	1	0	1
0	1	1	0	1
1	1	1	1	1

Checksum

- Lineup subsets of data and a checksum value of equal length
- Nibbles here (4 bits)
- Detect errors in columns, we'll use even parity here

N1	1	0	1	1
N2	0	1	1	1
N3	1	0	0	1
Checksum	0	1	0	1

Parity & Checksum

- Using these together we can detect the position of single bit errors
- How many bits in subsets?
- How many subsets to use?

N1	1	0	1	1	1
N2	0	1	1	1	1
N3	1	0	0	1	0
Checksum	0	1	0	1	

Parity & Checksum Example

- Which bit is wrong?

N1	1	0	0	0	1
N2	0	1	1	1	0
N3	0	0	0	0	0
Checksum	1	1	1	0	1

- What is the corrected set of values?

Parity & Checksum Example

- Which bit is wrong?

N1	1	0	0	0	1
N2	0	1	1	1	0
N3	0	0	0	0	0
Checksum	1	1	1	0	

- What is the corrected set of values?
  - $8_{10},6_{10},0_{10}$

Hamming Code

- Another error detection/correction scheme
- 7 bits used to transmit 4 bits of data

1	2	3	4	5	6	7
p1	p2	d1	p4	d2	d3	d4

- Apply even parity to:
  - p1,3,5,7
  - p2,3,6,7
  - p4,5,6,7

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

Hamming Code

- Find the 'Error Syndrome'

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

- Find errors in each parity group and write as a 3 bit value: p4 p2 p1 (1=error,0=no error)
- If no errors found, syndrome will equal 000_2

Hamming Code

- When there is an error:

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

- p4 p2 p1=010_2
- Error syndrome indicates position of error

Hamming Code

- Another error:

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

- p4 p2 p1=110_2
- Error syndrome indicates position of error
- Now we can detect error and position
- However, at the cost of transmitting additional bits for an equal amount of data