CSE 241
Number Systems & Binary Arithmetic

Positional Number Systems

- *aka*, Radix-Weighted Positional Number System
- Consider a base r number system
  - ☞Radix point separates integer & fractional components
  - ☞Finite set of r symbols called digits
  - ☞Position of digit determines weight of digit
- A positive number, N, can be written in positional notation as:

  $N = (d_{n-1}\ d_{n-2} \ldots d_1\ d_0\ .\ d_{-1}\ d_{-2} \ldots d_{-m})_r$

  where

  . = radix point

  r = radix (base) of number system

  n = number of integer digits to the left of the radix point

  m = number of fractional digits to the right of the radix point

  d_i = integer digit i, where $n\text{-}1 \geq i \geq 0$

  d_i = integer digit i, where $-1 \geq i \geq -m$

  d_{n-1} = most significant digit

  d_{-m} = least significant digit
- General form as a power series in r
- Example
  - ☞78.526_{10}
    - ↳$7 \times 10^1 + 8 \times 10^0 + 5 \times 10^{-1} + 2 \times 10^{-2} + 6 \times 10^{-3}$
  - ☞214.03_8
    - ↳$2 \times 8^2 + 1 \times 8^1 + 4 \times 8^0 + 0 \times 8^{-1} + 3 \times 8^{-2}$

Decimal to Binary Conversion

- Procedure
  - ☞Separate the decimal number into two portions, the integer component, i, & the fractional component, f.
  - ☞First, consider the integer component.
    - ↳Choose the largest power of 2, 2^n, less than or equal to i.

      $i/2^n = q,r_n$ $\quad b_n = q$

      $r_n/2^{n-1} = q,r_{n-1}$ $\quad b_{n-1} = q$

      $r_{n-1}/2^{n-2} = q,r_{n-2}$ $\quad b_{n-2} = q$

      . . . $\quad$ . . .

      $r_1/2^0 = q$ $\quad b_0 = q$
  - ☞Alternate method for the integer component.

    $i/b = q_0,r$ $\quad b_0 = r$

    $q_0/b = q_1,r$ $\quad b_1 = r$

    $q_1/b = q_2,r$ $\quad b_2 = r$

    . . . $\quad$ . . .
    - ↳Continue until q=0

- ☞Next, let's consider the fractional component.
  - $f \times 2 = b_{-1} . f_{-1}$
  - $f_{-1} \times 2 = b_{-2} . f_{-2}$
  - . . .
  - $f_{-m+1} \times 2 = b_{-m} . f_{-m}$
  - When $f_{-m} = 0$, stop
- ☞Put the integer & fractional results together.
  - $b_n\ b_{n-1} \ldots b_1\ b_0 . b_{-1}\ b_{-2} \ldots b_{-m}$

● Example

- ☞Convert 483.75_{10} to binary
  - First, consider the integer portion (using the first method)
    - ✓483/256 = 1, 227 — b_8
    - ✓227/128 = 1, 99 — b_7
    - ✓99/64 = 1, 35 — b_6
    - ✓35/32 = 1, 3 — b_5
    - ✓3/16 = 0, 3 — b_4
    - ✓3/8 = 0, 3 — b_3
    - ✓3/4 = 0, 3 — b_2
    - ✓3/2 = 1, 1 — b_1
    - ✓1/1 = 1, 0 — b_0
  - $(b_8\ b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0)_2$

  - Consider the integer portion using the alternate method
    - ✓483/2 = 241, 1 — b_0
    - ✓241/2 = 120, 1 — b_1
    - ✓120/2 = 60, 0 — b_2
    - ✓60/2 = 30,0 — b_3
    - ✓30/2 = 15,0 — b_4
    - ✓15/2 = 7, 1 — b_5
    - ✓7/2 = 3, 1 — b_6
    - ✓3/2 = 1, 1 — b_7
    - ✓1/2 = 0, 1 — b_8
  - $(b_8\ b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0)_2$
  - Next, consider the fractional portion
    - ✓0.75 x 2 = 1.5 — b_{-1}
    - ✓0.5 x 2 = 1.0 — b_{-2}
  - $(b_{-1}\ b_{-2})_2$
    - ✓11_2
  - Finally, combine the integer & fractional portions.
    - ✓111100011.11_2

Binary to Decimal Conversion

- Procedure
  - ☞ $b_{n-1} \times r^{n-1} + b_{n-2} \times r^{n-2} + \ldots + b_0 \times r^0 + b_{-1} \times r^{-1} + b_{-m+1} \times r^{-m+1} + b_{-m} \times r^{-m}$
- Example
  - ☞ Convert 1011.101_2 to decimal
    - $(b_3\, b_2\, b_1\, b_0\, .\, b_{-1}\, b_{-2}\, b_{-3})_2$
    - $1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}$
    - $8 + 2 + 1 + .5 + .125$
    - 11.625_{10}

Storage Limitations

- A storage device is limited by the number of bits it can store
- An *n* bit storage device can hold 2^n possible values

Binary Number System

- Digits
  - ☞ 0,1
- Radix point
  - ☞ Binary point
- Example
  - ☞ 11011.101
- Notation
  - ☞ $2^{10} \equiv$ K (kilo)
  - ☞ $2^{20} \equiv$ M (mega)
  - ☞ $2^{30} \equiv$ G (giga)
- Key powers of 2

n	2^n	*n*	2^n
0	1	10	1,024
1	2	11	2,048
2	4	12	4,096
3	8	13	8,192
4	16	14	16,384
5	32	15	32,768
6	64	16	65,536
7	128	20	1,048,576
8	256	30	1,073,741,824
9	512		

- Example
  - ☞ 64 M
  - ☞ $64 \times 2^{20} = 2^6 \times 2^{20} = 2^{26}$
  - ☞ 67,108,864

Convenient Number Systems

- Commonly used number systems in digital systems
  - ☞Binary
    - ↳Base 2
    - ↳0, 1
  - ☞Octal
    - ↳Base 8
    - ↳0, 1, 2, 3, 4, 5, 6, 7
  - ☞Hexadecimal
    - ↳Base 16
    - ↳0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
    - ↳Notation
      - ✓0x
      - ✓$

Converting from Base A to Base B when B=A^k

- Base A to B when B=A^k and k is a positive integer
  - ☞Group digits of N in groups of k digits proceeding away from the radix point in both directions
  - ☞Replace each group with its equivalent digit in base B.
- Base B to A when B=A^k and k is a positive integer
  - ☞Replace each base B digit in N with equivalent k digits in base A.
- Examples:
  - ☞0xA9 → 10101001_2
  - ☞1110100_2 → 0x74
  - ☞241_8 → 10100001_2
  - ☞1011101001_2 → 1351_8

Binary Addition

- Addition Table

+	0	1
0	0	1
1	1	0 (Carry 1)

- Using the above table, proceed as with base ten.
- Example
  - ☞Consider $14_{10} + 9_{10}$ using binary addition

$$
\begin{array}{r}
{}^{1}\quad\;\; 1\,1\,1\,0 \\
+\,1\,0\,0\,1 \\
\hline
1\,0\,1\,1\,1
\end{array}
$$

  - ☞Sum = $10111_2 = 23_{10}$

Binary Subtraction

- Consider M-N
  - ☞M ≡ Minuend
  - ☞N ≡ Subtrahend
- Table

- (Minuend \ Subtrahend)	0	1
0	0	1 (Borrow 1)
1	1	0

- Using the above table, proceed as with base ten.
- Example
  - ☞Consider 37_{10}-11_{10} using binary subtraction

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

  - ☞Difference = 11010_2 = 26_{10}

Binary Multiplication

- Multiplication Table

x	0	1
0	0	0
1	0	1

- Using the above table, proceed as with base ten.
- Example
  - ☞Consider $22_{10} \times 6_{10}$ using binary multiplication

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

  - ☞Product = 10000100_2 = 132_{10}

Binary Division

- Division consists of a series of repeated multiplications & subtractions.
- Process analogous to base ten division.
- Example
  - Consider $214_{10}/5_{10}$ using binary division

```
            101010
       ___________
101 ) 11010110
      101
      ---
       011
      000
      ---
        110
        101
        ---
         011
         000
         ---
          111
          101
          ---
           100
           000
           ---
           100
```

  - Quotient = $101010_2 = 42_{10}$
  - Remainder = $100_2 = 4_{10}$

Extending Arithmetic Operations to Other Bases

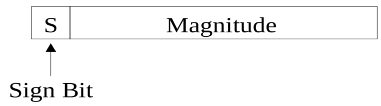
- These operations can be extended to other bases
  - Generate tables
  - Carry procedure similar to those previously described
- Often, it is easier to first convert numbers to decimal, carry out operation, and convert the result to the desired base.

Signed Numbers

- Signed numbers are represented using sign-magnitude or complement notation.
- The most significant bit represents the sign bit, indicating whether the number is positive or negative.
- Signed Number Range
  - $[-2^{n-1}, 2^{n-1} - 1]$
    - r's complement representation
  - $[-2^{n-1}+1, 2^{n-1} - 1]$
    - (r-1)'s complement & sign-magnitude representation
- Unsigned Number Range
  - $[0, 2^n - 1]$

Sign-magnitude Representation

- Consider a number, N, in base *r*:
  - $N_r = (a_{n-1}, a_{n-2}, \ldots, a_1, a_0)$
- Sign
  - $a_{n-1} = 0$ if $N_r \geq 0$
  - $a_{n-1} = r-1$ if $N_r \leq 0$



- Magnitude
  - $a_{n-2}, a_{n-3}, \ldots, a_1, a_0$
- Example
  - Represent 3 as a 16 bit number using sign-magnitude representation
    - 0000 0000 0000 0011
  - Represent -3 as a 16 bit number using sign-magnitude representation
    - 1000 0000 0000 0011

Radix Complement

- *aka*
  - r’s Complement
  - True Complement
- Procedure
  - Consider a number, N_r, in base *r*.
    - Let
      - ✓ $n \equiv$ number of integer digits in N_r
      - ✓ $m \equiv$ Number of fractional digits
  - For $N_r = 0$,
    - 0
    - This case is defined since $r^n - 0$ is an $n+1$ bit result
      - ✓ Result must have $n+m$ bits
  - For N_r<>0,
    - $r^n - N_r$
- Specific Cases
  - 10's complement (base 10)
  - 2's complement (base 2)
- Radix Complement Examples
  - Determine the 10's complement of 52520_{10}
    - $10^5 - 52520 = 47480$
  - Determine the 2’s complement of 101100_2
    - $1000000_2 - 101100_2 = 010100_2$
- Simple Algorithm
  - Start with the least significant digit & move toward the most significant (right to left)
  - Retain every digit until the first nonzero digit, a_i, is reached.
  - Replace a_i with $(r - a_i)$
  - Replace each remaining digit, a_j, (if any) with $(r-1-a_j)$

- For 2’s Complement
  - Start at the least significant bit and move toward the most significant bit.
  - Retain all zeros, until the first one, a_i, is reached.
  - Retain a_i
  - Complement all bits, a_j, more significant than a_i
- Examples
  - Determine the 10's complement of 496010_{10}
    - 503990
  - Determine the 10's complement of 367.24_{10}
    - 632.76
  - Determine the 2's complement of 1101010_2
    - 0010110_2

Diminished Radix Complement

- *aka*
  - (r-1)’s Complement
  - Radix-minus-one Complement
- Procedure
  - Consider a number, N_r, in base *r*.
    - Let
      - ✓ $n \equiv$ number of integer digits in N_r
  - For $N_r \neq 0$,
    - $r^n - r^{-m} - N_r$
      - ✓ $n \equiv$ Number of integer digits
      - ✓ $m \equiv$ Number of fractional digits
        - Note that for a number without a fractional component, $r^{-m} = 1$
- Specific Cases
  - 9's complement (base 10)
  - 1's complement (base 2)
- Examples
  - Determine the 9's complement of 52520_{10}
    - $10^5 - 52520 - 1$
    - 47479
  - Determine the 1's complement of 101100_2
    - $1000000_2 - 101100_2 - 1$
    - 010011_2
  - Determine the 1's complement of 11010.1011_2
    - $100000_2 - 101100_2 - 0.0001$
    - 00101.0100_2
- Simple Algorithm
  - Start with the least significant digit & move toward the most significant (right to left)
  - Replace every digit, a_i, with $r-1- a_i$

- For 1's Complement
  - ☞Complement every bit
- Examples
  - ☞Determine the 9's complement of 49601.83_{10}
    - ↳50398.16
  - ☞Determine the 1's complement of 110101_2
    - ↳001010_2

Notes on r's & (r-1)'s Complements

- Taking the complement of a complement returns the original number
- (r-1)'s complement notation & sign-magnitude notation have a positive & negative 0
  - ☞r's complement notation does not
- Relationship between r's & (r-1)'s complements
  - ☞r's complement = (r-1)'s complement + 1
    - ↳Exception
      - ✓1s 00…00 which represents -10^n_r in the r's complement representation

Notes on Signed Numbers

- Textbook
  - ☞Negative numbers appended with 1 at MSB position
- Class
  - ☞Negative numbers start with r-1 in MSB position

Signed Number Examples

- Examples
  - ☞Represent the following numbers in 10's complement, 9's complement, and sign-magnitude representations using 4 digits
    - ↳34
      - ✓10's complement = 0034
      - ✓9's complement = 0034
      - ✓Sign magnitude = 0034
    - ↳-178
      - ✓10's complement = 10000 - 178 = 9822
      - ✓9's complement = 10000 - 178 - 1 = 9821
      - ✓Sign magnitude = 9178
  - ☞Represent the following numbers in 2's complement, 1's complement, and sign-magnitude representations using 8 bits
    - ↳72_{10} = (1001000_2)
      - ✓2's complement = 01001000
      - ✓1's complement = 01001000
      - ✓Sign magnitude = 01001000
    - ↳-56_{10} = (111000_2)
      - ✓2's complement = 11001000
      - ✓1's complement = 11000111
      - ✓Sign magnitude = 10111000

Sign-Extension

- When an *n* digit signed number is represented by *n*+*k* bits using complement representation, the most significant *k* bits must replicate the sign bit of the *n* digit number.
- Example
  - ☞Consider the -49
    - ↳8-bit 2’s complement representation
      - ✓11001111
    - ↳Extended to 16-bits
      - ✓1111111111001111
    - ↳What happens if the sign bit is not properly extended?

Implementation of Addition & Subtraction

- Most computers use the radix complement number system to perform integer arithmetic.
- Why?
  - ☞Amount of circuitry required for these operations is minimized.
  - ☞A binary adder & complementing circuits can handle both addition & subtraction.

Subtraction with r’s Complement

- Procedure (M-N)
  - ☞Express minuend, M, and subtrahend, N, with same number of integer and fractional digits
  - ☞Add minuend, M, to r’s complement of subtrahend, N.
  - ☞If an end carry occurs, discard it.
    - ↳Indicates positive result
  - ☞If not, the result is a negative value represented in r’s complement notation.
- Example #1
  - ☞Consider 67_{10} - 15_{10} using 10's complement
    - ↳10's complement of 15 = 985
      - ✓Note 9 indicates negative
    - ↳067 + 985 = 1052
    - ↳Discard end carry
    - ↳Difference = 052_{10}
- Example #2
  - ☞Consider 21_{10} - 89_{10} using 10's complement
    - ↳10's complement of 89 = 911
    - ↳021 + 911 = 932
    - ↳No end carry
    - ↳Result (932) is in 10's complement notation
    - ↳Difference = -68_{10}

- Example #3
  - ☞Consider 81_{10} - 45_{10} using 2's complement
    - ↳1010001_2 - 00101101_2
    - ↳2's complement of 45_{10} = 11010011_2
    - ↳001010001 + 11010011 = 100100100
    - ↳End carry occurs
    - ↳Discard end carry
    - ↳Difference = 00100100_2 = 36_{10}
- Example #4
  - ☞Consider 53_{10} - 60_{10} using 2's complement
    - ↳00110101_2 - 00111100_2
    - ↳2's complement of 60_{10} = 11000100_2
    - ↳00110101 + 11000100 = 11111001
    - ↳No end carry
    - ↳Result (11111001_2) is in 2's complement notation
    - ↳Difference = 11111001_2 = -7_{10}

Subtraction with (r-1)'s Complement

- Procedure (M-N)
  - ☞Express minuend, M, and subtrahend, N, with same number of integer and fractional digits
  - ☞Add minuend, M, to (r-1)'s complement of subtrahend, N.
  - ☞If an end carry occurs, add 1 to the least significant digit.
    - ↳Referred to as an end-around carry
    - ↳Indicates positive result
  - ☞If not, the sum is a negative value represented in (r-1)'s complement notation.
- Example #1
  - ☞Consider 58_{10} - 37_{10} using 9's complement
    - ↳9's complement of 37 = 962
      - ✓Note 9 indicates negative
    - ↳058 + 962 = 1020
    - ↳End carry occurs
    - ↳Add end carry
    - ↳Difference = 21_{10}
- Example #2
  - ☞Consider 11_{10}- 53_{10} using 9's complement
    - ↳9's complement of 53 = 946
      - ✓Note 9 indicates negative
    - ↳011 + 946
    - ↳No end carry
    - ↳Result (957) is in 9's complement notation
    - ↳Difference = - 42_{10}

- Example #3
  - Consider 81_{10} - 45_{10} using 1's complement
    - 1's complement of 45 = 110100010_2
    - 01010001 + 11010010 = 100100011
    - End carry occurs
    - Add end carry
    - Difference = 36_{10}
- Example #4
  - Consider 25_{10} - 42_{10} using 1's complement
    - 1's complement of 42 = 11010101_2
    - 00011001 + 11010101 = 11101110
    - No carry occurs
    - Result is in 1’s complement notation
    - Difference = - 17_{10}

Overflow Conditions

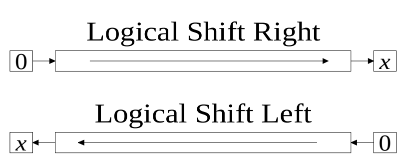
- An overflow occurs when the result of an arithmetic operation falls outside the available range that can be stored.
- Condition codes in the processor are maintained to determine if an overflow has occurred.
- Detection of overflow for addition of signed numbers
  - Carries into & out of MSB (sign bit) differ
  - Two positive numbers added & negative result is obtained
  - Two negative numbers added & a positive result is obtained
- Note that overflow cannot occur if two number of differing signs are added

Comparison of 1's & 2's Complements

- 1's complement is easier to implement by digital circuits.
- 2's complement requires only 1 arithmetic operation to carry out subtraction where 1's complement requires 2 due to the end-around carry.
- 1's complement has the disadvantage of 2 zeros.
  - Positive 0: 0...0
  - Negative 0: 1...1

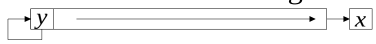
Shifts & Rotates

- Logical Shifting
  - Bits shifted left or right
  - Logical 0 shifted in
  - Shifting n positions left implements multiplication by 2^n
  - Example
    - Shift 11010110 logically left 3 places
      - ✓Result: 10110000



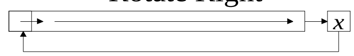
- Arithmetic Right Shifting
  - MSB shifted in
  - Maintains sign bit
  - Shifting n positions left implements division by 2^n
  - Example
    - Arithmetic shift 11010110 right 3 places
      - Result: 11111010

Arithmetic Shift Right

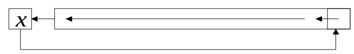


- Rotating
  - Bits rotated left or right
  - Bit(s) rotated out is (are) shifted in
  - Example
    - Rotate 11010010 left 5 places
      - Result: 01011010

Rotate Right



Rotate Left



Codes

- Code Group
  - Unique string of binary digits representing a symbol (character, digit, etc.)
- Decimal Codes
  - BCD
    - Binary Coded Decimal
  - Represents decimal digits
    - $0 \rightarrow 9$
  - Weighted Codes
    - Position of 1 indicates weight
  - 8421 Code
    - Weighted Code
    - Most common BCD code
  - 2421 Code
    - Self-complementing
      - 1’s complement of code yields 9’s complement of number
      - Example
        - 9’s complement of 61 is 38
        - 1’s complement of 11000001 is 00111110
  - 7536 Code
    - Weights of 7, 5, 3 are positive
    - Weight of 6 is negative
  - 5421 Code
  - Biquinary Code
    - 5043210 weighted code
    - Two of seven bits are 1
      - First 1 in first two bits
      - Second 1 in last 5 bits

- ☞Excess-three Code
  - ➥*aka*, XS-3 code
  - ➥Nonweighted BCD
  - ➥3 is added to each 8421 code group
  - ➥Self-complementing
  - ➥Example
    - ✓74
      - 0111 + 0011 = 1010
      - 0100 + 0011 = 0111
      - XS-3 code is 10100111
- ☞2-out-of-5 Code
  - ➥Nonweighted code
  - ➥Exactly 2 of 5 bits are 1
  - ➥Error detecting

● Unit-Distance Codes

- ☞Only a single bit changes between any two successive coded integers
- ☞Example
  - ➥Gray Code

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

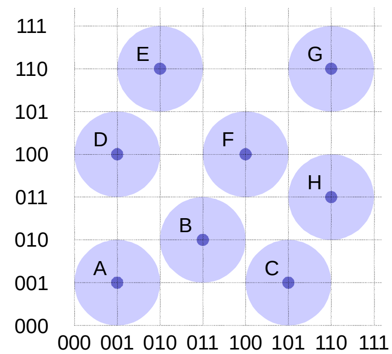
● Alphanumeric Codes

- ☞Uppercase/Lowercase letters of alphabet
- ☞Digits (0 → 9)
- ☞Punctuation
- ☞Control Operations
  - ➥Backspace
  - ➥Form Feed
  - ➥Carriage Return
  - ➥Escape
  - ➥…

- Special Characters
  - ($ # @ = + [ ] * - …)
- American Standard Code for Information Interchange
  - *aka*, ASCII
  - 7-bit
  - Examples
    - d ≡ 1100100
    - 9 ≡ 0111001
    - R ≡ 1010010
    - * ≡ 0101010
- Unicode
  - 16-bit
  - International
    - English as well as many other languages
    - Punctuation marks
    - Mathematical Symbols
    - Technical Symbols
    - Geometric Shapes
    - Dingbats

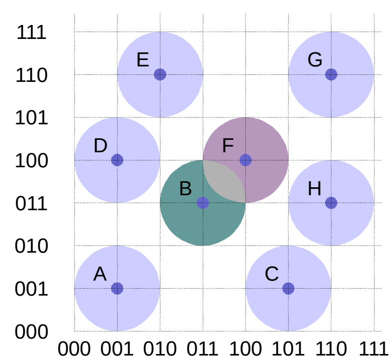
- Error Detection
  - A code is said to be *n*-error detecting if the minimum of *n* errors that *cannot* be detected is *n*+1
    - Error defined as a bit being complemented erroneously
  - Example
    - 2-out-of-5 codes
      - Single error detecting
      - Example
        - A 01010 transmitted as 01110
        - Error can be detected
    - Parity
      - A parity bit can be concatenated to a code word that does not incorporate error detection to make it a single error detecting code
        - Detects an odd number of errors
      - Even Parity
        - The code word (including the parity bit) has an even number of 1's
      - Odd Parity
        - The code word (including the parity bit) has an odd number of 1's
      - Example
        - The 7-bit ASCII code is often concatenated with a parity bit
        - H (odd parity) ≡ 11001000
  - Distance between two code groups
    - The number of bits that must change so that the first code group becomes the second
  - Minimum Distance
    - Minimum distance between any two valid code groups in a coding scheme

- ☞Maximum number of detectable errors
  - ↳D = M - 1
    - ✓D ≡ error detecting capability of code
    - ✓M ≡ minimum distance
- Error Correction
  - ☞It is possible to construct a code whereby a finite number of errors can be corrected
    - ↳C + D = M - 1 where $C \leq D$
      - ✓C ≡ Number of erroneous bits that can be corrected
      - ✓D ≡ Number of errors that can be detected
      - ✓M ≡ Minimum distance of code
- Error Detection vs. Error Correction
  - ☞Consider a 6-bit code group used to represent 8 unique codes
  - ☞Graphical Representation
    - ↳First six bits are along the x-axis, last six bits are along the y-axis



  - ☞Is the codegroup error detecting, error correcting, both, or neither?

  - ☞Code B is changed from 010010 to 011011. Does this change whether it is error detecting or correcting? If so, how?



- Error Correction
  - Hamming Code
    - Derived by R.W. Hamming
    - Consider the case of four information bits
      - ✓Three parity bits are included
        - Each calculated over a specified set of bits
      - ✓Let p_i represent parity bit i
      - ✓Let b_i represent parity information bit i

7	6	5	4	3	2	1	Position
b_4	b_3	b_2	p_3	b_1	p_2	p_1	Code Group Format

      - ✓$p_1 \equiv$ Even parity over positions 1, 3, 5, 7
      - ✓$p_2 \equiv$ Even parity over positions 2, 3, 6, 7
      - ✓$p_3 \equiv$ Even parity over positions 4, 5, 6, 7
    - Example
      - ✓Code word to be coded
        - 1101
      - ✓We need to determine p_1, p_2, p_3
        - 110_1_ _
      - ✓Hamming Code
        - 1100110
    - To detect/correct a single error, a *binary check number* is created
      - ✓$c_3^* c_2^* c_1^*$
      - ✓c_i is p_i recalculated
      - ✓The binary check number determines the position of the bit that must be complemented to obtain the error free code word
        - If the binary check number is zero, the code received is error free
      - ✓Example
        - A 1101110 is received
        - $c_3^* c_2^* c_1^* = 100$
        - Complement the 4^{th} (100_2) bit to correct the code word (1100110)
    - The Hamming Code can be generalized to any number of bits
      - ✓$m \equiv$ number of information bits
      - ✓$k \equiv$ number of parity bits
      - ✓$m \le 2^k - k - 1$
      - ✓$m+k$ bits are required for code word
      - ✓Let us number positions from right to left starting with 1 and ending at $m+k$
      - ✓Parity bit, p_i, in position 2^i, considers every other group of 2^i bits beginning with the parity bit in position 2^i
      - ✓Binary check number $c_k^* \ldots c_1^*$ determines the position that must be complemented to determine the correct code

- Single Error Correction & Double Error Detection
  - ☞Append a parity bit to the entire code group and implement even parity
    - Not used in calculation of other parity bit calculations
  - ☞Interpreting a Code Word
    - Case 1
      - ✓ $c_k^* \ldots c_1^* = 0\ldots0$ and additional parity bit is correct
        - No single or double errors
    - Case 2
      - ✓ $c_k^* \ldots c_1^* \neq 0\ldots0$ and additional parity bit is incorrect
        - Single error, corrected by complementing bit in position $c_k^* \ldots c_1^*$
    - Case 3
      - ✓ $c_k^* \ldots c_1^* \neq 0\ldots0$ and additional parity bit is correct
        - Two errors, not correctable
- Check Sum Digits for Error Correction
  - ☞Consider 5 + 4 ZIP Codes
    - 2-out-of-5 code is used to encode each digit
    - A checksum digit is appended to ZIP code so that sum is a multiple of 10
      - ✓ If a single digit is in error (number of 1's ≠ 2) the checksum can be used to correct check digit

$$\left(\begin{array}{c}\text{ZIP Digit}\\\text{Sum}\end{array} + \begin{array}{c}\text{Check Sum}\\\text{Digit}\end{array}\right)_{\text{mod } 10} = 0$$

References

- M. Morris Mano and Charles R. Kime, *Logic and Computer Design Fundamentals*, Prentice Hall, Inc., 2000
- Victor P. Nelson, H. Troy Nagle, Bill D. Carroll, and J. David Irwin, *Digital Logic Circuit Analysis and Design*, Prentice Hall, Inc., 1995
- Donald D. Givone, *Digital Principles and Design*, McGraw-Hill, 2003