Design of Digital Systems II

Number Systems and Codes

Moslem Amiri, Václav Přenosil

Embedded Systems Laboratory
Faculty of Informatics, Masaryk University
Brno, Czech Republic

amiri@mail.muni.cz
prenosil@fi.muni.cz

September, 2012

Introduction

- Digital systems are built from circuits that process binary digits
  - A digital system designer must establish some correspondence between binary digits processed by digital circuits and real-life numbers, events, and conditions

Positional Number Systems

- **Positional number system**
  - In such a system, a number is represented by a string of digits, where each digit position has an associated **weight**
  - Value of a number is a weighted sum of digits

$$d_1 d_0 . d_{-1} d_{-2} = d_1 \cdot 10^1 + d_0 \cdot 10^0 + d_{-1} \cdot 10^{-1} + d_{-2} \cdot 10^{-2}$$
$$5185.68 = 5 \cdot 1000 + 1 \cdot 100 + 8 \cdot 10 + 5 \cdot 1 + 6 \cdot 0.1 + 8 \cdot 0.01$$

  Here, 10 is **base** or **radix** of number system

- Radix may be any integer $r \geq 2$
  - A digit in position i has weight r^i

$$d_{p-1} d_{p-2} \cdots d_1 d_0 \underset{\substack{\uparrow \\ \text{radix point}}}{.} d_{-1} d_{-2} \cdots d_{-n} = \sum_{i=-n}^{p-1} d_i \cdot r^i$$

  - **High-order** or **most significant digit** = the leftmost digit
  - **Low-order** or **least significant digit** = the rightmost digit

Positional Number Systems

- **Binary radix** is normally used to represent numbers in a digital system

$$\underset{\substack{\uparrow\\ \text{binary point}}}{b_{p-1}b_{p-2}\cdots b_1 b_0 . b_{-1}b_{-2}\cdots b_{-n}} = \sum_{i=-n}^{p-1} b_i \cdot 2^i$$

  - **High-order** or **most significant bit (MSB)** $=$ the leftmost bit
  - **Low-order** or **least significant bit (LSB)** $=$ the rightmost digit

- Example

  $101.001_2 = 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 + 0 \cdot 0.5 + 0 \cdot 0.25 + 1 \cdot 0.125 = 5.125_{10}$

Octal and Hexadecimal Numbers

- Radices 8 and 16 provide convenient shorthand representations for multibit numbers in a digital system
    - Used for documentation or other purposes
- **Octal number system**
    - Uses radix 8
    - Needs 8 digits, so it uses digits $0 - 7$ of decimal system
- **Hexadecimal number system**
    - Uses radix 16
    - Needs 16 digits, so it supplements decimal digits $0 - 9$ with letters $A - F$

Octal and Hexadecimal Numbers

Table 1: Binary, decimal, octal, and hexadecimal numbers.

Binary	***Decimal***	***Octal***	***3-Bit String***	***Hexadecimal***	***4-Bit String***
0	0	0	000	0	0000
1	1	1	001	1	0001
10	2	2	010	2	0010
11	3	3	011	3	0011
100	4	4	100	4	0100
101	5	5	101	5	0101
110	6	6	110	6	0110
111	7	7	111	7	0111
1000	8	10	—	8	1000
1001	9	11	—	9	1001
1010	10	12	—	A	1010
1011	11	13	—	B	1011
1100	12	14	—	C	1100
1101	13	15	—	D	1101
1110	14	16	—	E	1110
1111	15	17	—	F	1111

Octal and Hexadecimal Numbers

- **Binary-to-octal conversion**
    - Starting at binary point and working left, separate bits into groups of three and replace each group with corresponding octal digit
    - Add zeros on left to make total number of bits a multiple of 3 if required

$$11101101110101001_2 = 011\ 101\ 101\ 110\ 101\ 001_2 = 355651_8$$

- **Binary-to-hexadecimal conversion**
    - Similar to binary-to-octal conversion, except groups of four bits are used
    - Add zeros on left to make total number of bits a multiple of 4 if required

$$11101101110101001_2 = 0001\ 1101\ 1011\ 1010\ 1001_2 = 1DBA9_{16}$$

- If a binary number contains digits to right of binary point, convert to octal or hexadecimal by starting at binary point and working right
    - Both lefthand and righthand can be padded with zeros to get multiples of three or four bits

$$\begin{aligned} 10.1011001011_2 &= 010\ .\ 101\ 100\ 101\ 100_2 = 2.5454_8 \\ &= 0010\ .\ 1011\ 0010\ 1100_2 = 2.B2C_{16} \end{aligned}$$

Octal and Hexadecimal Numbers

- **Octal- or hexadecimal-to-binary conversion**
    - Replace each octal or hexadecimal digit with corresponding 3- or 4-bit string

$$2046.17_8 = 010\ 000\ 100\ 110\ .\ 001\ 111_2$$
$$9F.46C_{16} = 1001\ 1111\ .\ 0100\ 0110\ 1100_2$$

- Today, majority of machines process 8-bit **bytes**
    - Octal number system is not used much because it is difficult to extract individual byte values in multibyte quantities in octal representation
    - In hexadecimal system, two digits represent an 8-bit byte, and $2n$ digits represent an n-byte word
    - A 4-bit hexadecimal digit is called a **nibble**
    - Hexadecimal numbers are often used to describe a computer's memory address space

General Positional-Number-System Conversions

- **Radix-r-to-decimal conversion**
  - Value of a number in any radix r

$$d_{p-1}d_{p-2}\cdots d_1d_0.d_{-1}d_{-2}\cdots d_{-n} = \sum_{i=-n}^{p-1} d_i \cdot r^i$$

  - Value of number can be found by converting each digit to its radix-10 equivalent and expanding formula using radix-10 arithmetic

$$F1A3_{16} = 15\cdot 16^3 + 1\cdot 16^2 + 10\cdot 16^1 + 3\cdot 16^0 = 61859_{10}$$
$$436.5_8 = 4\cdot 8^2 + 3\cdot 8^1 + 6\cdot 8^0 + 5\cdot 8^{-1} = 286.625_{10}$$
$$132.3_4 = 1\cdot 4^2 + 3\cdot 4^1 + 2\cdot 4^0 + 3\cdot 4^{-1} = 30.75_{10}$$

  - A shortcut for converting whole numbers to radix 10

$$\begin{aligned} D = d_{p-1}d_{p-2}\cdots d_1d_0 &= \sum_{i=0}^{p-1} d_i \cdot r^i \\ &= ((\cdots((d_{p-1})\cdot r + d_{p-2})\cdot r + \cdots)\cdot r + d_1)\cdot r + d_0 \qquad (1) \end{aligned}$$

General Positional-Number-System Conversions

- **Decimal-to-radix-r conversion**
  - Dividing formula (1) by r, quotient will be
$$Q = (\cdots((d_{p-1} \cdot r + d_{p-2}) \cdot r + \cdots) \cdot r + d_1$$
    and **remainder** $= \mathbf{d_0}$
    - Furthermore, quotient Q has the same form as (1). Therefore, successive divisions by r yield successive digits of D from right to left

```
179 ÷ 2 = 89 remainder 1    (LSB)
          ÷2 = 44 remainder 1
               ÷2 = 22 remainder 0
                    ÷2 = 11 remainder 0
                         ÷2 = 5 remainder 1
                             ÷2 = 2 remainder 1
                                 ÷2 = 1 remainder 0
                                     ÷2 = 0 remainder 1    (MSB)
```
$179_{10} = 10110011_2$

```
467 ÷ 8 = 58 remainder 3    (least significant digit)
          ÷8 = 7 remainder 2
             ÷ 8 = 0 remainder 7    (most significant digit)
```
$467_{10} = 723_8$

```
3417 ÷ 16 = 213 remainder 9    (least significant digit)
            ÷ 16 = 13 remainder 5
                 ÷ 16 = 0 remainder 13    (most significant digit)
```
$3417_{10} = D59_{16}$

General Positional-Number-System Conversions

Table 2: Conversion methods for common radices.

Conversion	***Method***	***Example***
Binary to		
Octal	Substitution	$10111011001_2 = 10\ 111\ 011\ 001_2 = 2731_8$
Hexadecimal	Substitution	$10111011001_2 = 101\ 1101\ 1001_2 = 5D9_{16}$
Decimal	Summation	$10111011001_2 = 1 \cdot 1024 + 0 \cdot 512 + 1 \cdot 256 + 1 \cdot 128 + 1 \cdot 64$ $+ 0 \cdot 32 + 1 \cdot 16 + 1 \cdot 8 + 0 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 = 1497_{10}$
Octal to		
Binary	Substitution	$1234_8 = 001\ 010\ 011\ 100_2$
Hexadecimal	Substitution	$1234_8 = 001\ 010\ 011\ 100_2 = 0010\ 1001\ 1100_2 = 29C_{16}$
Decimal	Summation	$1234_8 = 1 \cdot 512 + 2 \cdot 64 + 3 \cdot 8 + 4 \cdot 1 = 668_{10}$
Hexadecimal to		
Binary	Substitution	$C0DE_{16} = 1100\ 0000\ 1101\ 1110_2$
Octal	Substitution	$C0DE_{16} = 1100\ 0000\ 1101\ 1110_2 = 1\ 100\ 000\ 011\ 011\ 110_2 = 140336_8$
Decimal	Summation	$C0DE_{16} = 12 \cdot 4096 + 0 \cdot 256 + 13 \cdot 16 + 14 \cdot 1 = 49374_{10}$
Decimal to		
Binary	Division	$108_{10} \div 2 = 54$ remainder 0 (LSB) $\div 2 = 27$ remainder 0 $\div 2 = 13$ remainder 1 $\div 2 = 6$ remainder 1 $\div 2 = 3$ remainder 0 $\div 2 = 1$ remainder 1 $\div 2 = 0$ remainder 1 (MSB) $108_{10} = 1101100_2$
Octal	Division	$108_{10} \div 8 = 13$ remainder 4 (least significant digit) $\div 8 = 1$ remainder 5 $\div 8 = 0$ remainder 1 (most significant digit) $108_{10} = 154_8$
Hexadecimal	Division	$108_{10} \div 16 = 6$ remainder 12 (least significant digit) $\div 16 = 0$ remainder 6 (most significant digit) $108_{10} = 6C_{16}$

Addition and Subtraction of Nondecimal Numbers

Table 3: Binary addition and subtraction table.

c_{in} **or** b_{in}	x	y	c_{out}	s	b_{out}	d
0	0	0	0	0	0	0
0	0	1	0	1	1	1
0	1	0	0	1	0	1
0	1	1	1	0	0	0
1	0	0	0	1	1	1
1	0	1	1	0	1	0
1	1	0	1	0	0	0
1	1	1	1	1	1	1

- **Binary addition**
  - To add two binary numbers X and Y, add together LSBs with an initial carry (c_{in}) of 0, producing carry (c_{out}) and sum (s) bits
  - Continue processing bits from right to left, adding carry out of each column into next column's sum

Addition and Subtraction of Nondecimal Numbers

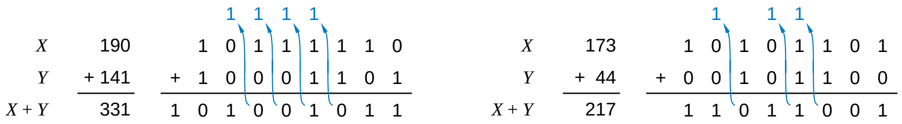

Figure 1: Examples of decimal and corresponding binary additions.

$$\begin{array}{ccr}
C & & 101111000 \\
X & 190 & 10111110 \\
\mathrm{Y} & +141 & + \quad 10001101 \\
\hline
X+Y & 331 & 101001011
\end{array}
\qquad
\begin{array}{ccr}
C & & 001011000 \\
X & 173 & 10101101 \\
Y & +\ 44 & +\ 00101100 \\
\hline
X+Y & 217 & 11011001
\end{array}$$

$$\begin{array}{ccr}
C & & 011111110 \\
X & 127 & 01111111 \\
Y & +\ 63 & + \quad 00111111 \\
\hline
X+Y & 190 & 10111110
\end{array}
\qquad
\begin{array}{ccr}
C & & 000000000 \\
X & 170 & 10101010 \\
Y & +\ 85 & +\ 01010101 \\
\hline
X+Y & 255 & 11111111
\end{array}$$

Addition and Subtraction of Nondecimal Numbers

- **Binary subtraction**
  - Is performed similar to binary addition, using borrows (b_{in} and b_{out}) instead of carries between steps, and producing a difference bit d
  - A common use of subtraction is to compare two numbers
    - If $X - Y$ produces a borrow out of MSB position, $X < Y$; otherwise, $X \geq Y$

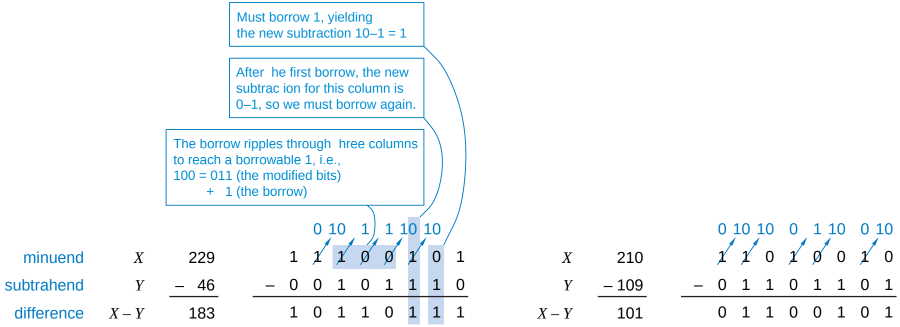
Must borrow 1, yielding the new subtraction 10–1 = 1
After he first borrow, the new subtrac ion for this column is 0–1, so we must borrow again.
The borrow ripples through hree columns to reach a borrowable 1, i.e., 100 = 011 (the modified bits) + 1 (the borrow)
minuend
subtrahend
difference
X 229 – Y 46 X – Y 183
0 10 1 1 10 10
1 1 1 0 0 1 0 1
– 0 0 1 0 1 1 1 0
1 0 1 1 0 1 1 1
X 210 Y – 109 X – Y 101
0 10 10 0 1 10 0 10
1 1 0 1 0 0 1 0
– 0 1 1 0 1 1 0 1
0 1 1 0 0 1 0 1

Figure 2: Examples of decimal and corresponding binary subtractions.

Addition and Subtraction of Nondecimal Numbers

B		001111100
X	229	11100101
Y	− 46	− 00101110
$X-Y$	183	10110111

B		011011010
X	210	11010010
Y	−109	− 01101101
$X-Y$	101	01100101

B		010101010
X	170	10101010
Y	− 85	− 01010101
$X-Y$	85	01010101

B		000000000
X	221	11011101
Y	− 76	− 01001100
$X-Y$	145	10010001

Addition and Subtraction of Nondecimal Numbers

- Two methods for octal and hexadecimal addition and subtraction
  1. Convert number to decimal, calculate results, and convert back
  2. Convert each column digits to decimal, do operation in decimal, and convert result to corresponding sum and carry digits in non-decimal radix
     - A carry is produced whenever column sum equals or exceeds radix

C	1 1 0 0	1	1	0	0
X	1 9 B 9 $_{16}$	1	9	11	9
Y	+ C 7 E 6 $_{16}$	+12	7	14	6
$X+Y$	E 1 9 F $_{16}$	14	17	25	15
		14	16+1	16+9	15
		E	1	9	F

Representation of Negative Numbers: Signed-Magnitude

- **Signed-magnitude system**
  - In this system, a number consists of a magnitude and a symbol indicating whether it is positive or negative
  - In everyday business we use this system
  - To apply to binary numbers, an extra bit position is used to represent sign called **sign bit**
    - Traditionally, MSB of a bit string is used as sign bit
    - $0 =$ plus, $1 =$ minus
  - Several 8-bit signed-magnitude integers and their decimal equivalents
    - $01010101_2 = +85_{10}$
    - $01111111_2 = +127_{10}$
    - $00000000_2 = +0_{10}$
    - $11010101_2 = -85_{10}$
    - $11111111_2 = -127_{10}$
    - $10000000_2 = -0_{10}$
  - There are two possible representations for zero
  - This system has an equal number of positive and negative integers

$$-(2^{n-1}-1) \leq \text{An } n\text{-bit integer} \leq +(2^{n-1}-1)$$

Representation of Negative Numbers: Signed-Magnitude

- **Signed-magnitude adder**
  - The circuit must examine signs of addends to determine what to do with magnitudes
  - If signs are the same, it must add magnitudes and give the result the same sign
  - If signs are different, it must compare magnitudes, subtract the smaller from the larger, and give the result the sign of the larger
  - "Ifs," "adds," "subtracts," and "compares" translate into a lot of logic-circuit complexity
- **Signed-magnitude subtractor**
  - It need only change sign of subtrahend and pass it along with minuend to an adder

Complement Number Systems

- A **complement number system** negates a number by taking its complement as defined by the system
- Two numbers in a complement number system can be added or subtracted directly without sign and magnitude checks required by signed-magnitude system
- In any complement number system, we deal with a fixed number of digits, say n
- We assume the numbers are integers
- If an operation produces a result that requires more than n digits, we throw away the extra high-order digit(s)
- If a number D is complemented twice, result is D

Radix-Complement Representation

- **Radix-complement system**
  - Radix-complement of an n-digit number D (in radix r) $= r^n - D$
  - In decimal number system, it is called **10's complement**
  - For an n-digit number D

$$1 \leq D \leq r^n - 1 \longrightarrow 1 \leq r^n - D \leq r^n - 1$$

$$D = 0 \longrightarrow r^n - D = \underbrace{r^n}_{100\cdots00} \xrightarrow{n+1 \text{ digits}} \text{remove extra high-order digit} \longrightarrow 0$$

  - Thus, there is only one representation of zero in this system

Table 4: Examples of 10's and 9s' complements using 4-digit decimal numbers.

Number	***10's complement***	***9s' complement***
1849	8151	8150
2067	7933	7932
100	9900	9899
7	9993	9992
8151	1849	1848
0	10000 (= 0)	9999

Radix-Complement Representation

- To avoid subtraction $r^n - D$ in computing radix complement, rewrite

$$r^n \longrightarrow (r^n - 1) + 1$$
$$r^n - D \longrightarrow ((r^n - 1) - D) + 1$$

  - $r^n - 1$ has the form $mm \cdots mm$, where $m = r - 1$ and there are n m's
    E.g., $10,000 = 9,999 + 1$
  - If we define

    Complement of a digit $d = r - 1 - d \xrightarrow{\text{complementing digits of } D} (r^n - 1) - D$

    Radix complement of D = (complementing individual digits of D)+1

Radix-Complement Representation

Table 5: Digit complements.

	Complement			
Digit	***Binary***	***Octal***	***Decimal***	***Hexadecimal***
0	1	7	9	F
1	0	6	8	E
2	–	5	7	D
3	–	4	6	C
4	–	3	5	B
5	–	2	4	A
6	–	1	3	9
7	–	0	2	8
8	–	–	1	7
9	–	–	0	6
A	–	–	–	5
B	–	–	–	4
C	–	–	–	3
D	–	–	–	2
E	–	–	–	1
F	–	–	–	0

Two's-Complement Representation

- For binary numbers, radix complement is called **two's complement**
  - MSB of a number serves as sign bit

$$\text{Negative number} \longleftrightarrow MSB = 1$$

  - Decimal equivalent for a two's-complement binary number
    - Weight of $MSB = -2^{n-1}$ instead of $+2^{n-1}$
    - Weight of the rest of bits is computed the same way as for an unsigned number
  - Range of representable numbers

$$-(2^{n-1}) \leq \text{Range} \leq +(2^{n-1} - 1)$$

Two's-Complement Representation

$$\begin{array}{rcrl}
17_{10} & = & 00010001^2 & \\
 & & \Downarrow & \text{complement bits} \\
 & & 11101110 & \\
 & & +1 & \\
\hline
 & & 11101111^2 & = -17_{10}
\end{array}
\qquad
\begin{array}{rcrl}
-99_{10} & = & 10011101^2 & \\
 & & \Downarrow & \text{complement bits} \\
 & & 01100010 & \\
 & & +1 & \\
\hline
 & & 01100011^2 & = 99_{10}
\end{array}$$

$$\begin{array}{rcrl}
119_{10} & = & 01110111 & \\
 & & \Downarrow & \text{complement bits} \\
 & & 10001000 & \\
 & & +1 & \\
\hline
 & & 10001001^2 & = -119_{10}
\end{array}
\qquad
\begin{array}{rcrl}
-127_{10} & = & 10000001 & \\
 & & \Downarrow & \text{complement bits} \\
 & & 01111110 & \\
 & & +1 & \\
\hline
 & & 01111111^2 & = 127_{10}
\end{array}$$

$$\begin{array}{rcrl}
0_{10} & = & 00000000^2 & \\
 & & \Downarrow & \text{complement bits} \\
 & & 11111111 & \\
 & & +1 & \\
\hline
 & & 1\ 00000000^2 & = 0_{10}
\end{array}
\qquad
\begin{array}{rcrl}
-128_{10} & = & 10000000^2 & \\
 & & \Downarrow & \text{complement bits} \\
 & & 01111111 & \\
 & & +1 & \\
\hline
 & & 10000000^2 & = -128_{10}
\end{array}$$

Two's-Complement Representation

- In two's complement system, zero is considered positive because its sign bit is 0
    - Since there is only one representation of zero, we end up with one extra negative number, -2^{n-1}, with no positive counterpart
- To convert an n-bit two's-complement number X into an m-bit one
    - If $m > n \longrightarrow$ **sign extension**: append $m - n$ copies of X's sign bit to left of X
    - If $m < n$, discard X's $n - m$ leftmost bits
        - Result is valid if all of discarded bits are same as sign bit of the result

Diminished Radix-Complement Representation

- **Diminished radix-complement system**
    - Complement of an *n*-digit number $D = (r^n - 1) - D$
        - Can be obtained by complementing individual digits of D, without adding 1
    - In decimal, this is called **9s' complement**
        - Examples in Tab. 4

Ones'-Complement Representation

- Diminished radix-complement system for binary numbers is called **Ones' complement**
  - MSB is sign, 0 if positive and 1 if negative
  - Two representations of zero, positive zero $(00\cdots00)$ and negative zero $(11\cdots11)$
  - Positive-number representations are the same for both ones' and two's complements
  - Negative-number representations differ by 1
  - Weight of MSB for decimal equivalent computing $= -(2^{n-1}-1)$ rather than -2^{n-1}
  - Range of representable numbers

$$-(2^{n-1}-1) \leq \text{Range} \leq +(2^{n-1}-1)$$

  - Advantages
    - Symmetry
    - Ease of complementation
  - Disadvantages
    - Adder design is trickier than a two's-complement adder
    - Zero-detecting circuits either must check for both representations of zero, or must always convert $11\cdots11$ to $00\cdots00$

Ones'-Complement Representation

$$17_{10} = 00010001_2 \;\Downarrow\; 11101110_2 = -17_{10}$$

$$-99_{10} = 10011100_2 \;\Downarrow\; 01100011_2 = 99_{10}$$

$$119_{10} = 01110111_2 \;\Downarrow\; 10001000_2 = -119_{10}$$

$$-127_{10} = 10000000_2 \;\Downarrow\; 01111111_2 = 127_{10}$$

$$0_{10} = 00000000_2 \text{ (positive zero)} \;\Downarrow\; 11111111_2 = 0_{10} \text{ (negative zero)}$$

Excess Representations

- **Excessive-B representation**
  - An m-bit string whose unsigned integer value is M ($0 \leq M < 2^m$) represents signed integer $M - B$
  - B is called **bias** of number system
  - E.g., **excess-$\mathbf{2^{m-1}}$ system** represents any number X in range
    $$-2^{m-1} \leq X \leq +2^{m-1} - 1$$
    by m-bit binary representation of $X + 2^{m-1}$ which is always nonnegative and less than 2^m
  - excess-2^{m-1} system vs. m-bit two's complement
    - Their range of representations are the same
    - Representations of any number in the two systems are identical except for sign bits, which are opposite
  - Most common use is in floating-point number systems

Two's-Complement Addition

Table 6: Decimal and 4-bit numbers.

Decimal	***Two's Complement***	***Ones' Complement***	***Signed Magnitude***	***Excess 2^{m-1}***
–8	1000	—	—	0000
–7	1001	1000	1111	0001
–6	1010	1001	1110	0010
–5	1011	1010	1101	0011
–4	1100	1011	1100	0100
–3	1101	1100	1011	0101
–2	1110	1101	1010	0110
–1	1111	1110	1001	0111
0	0000	1111 or 0000	1000 or 0000	1000
1	0001	0001	0001	1001
2	0010	0010	0010	1010
3	0011	0011	0011	1011
4	0100	0100	0100	1100
5	0101	0101	0101	1101
6	0110	0110	0110	1110
7	0111	0111	0111	1111

Two's-Complement Addition

- Tab. 6 reveals why two's complement is preferred
  - Starting with 1000_2 (-8_{10}) and counting up, each successive number up to 0111_2 $(+7_{10})$ can be obtained by adding 1 to previous one
  - Because ordinary addition is just an extension of counting, two's-complement numbers can thus be added by ordinary binary addition, ignoring any carries beyond MSB
    - Result is correct if range of number system is not exceeded

$$\begin{array}{r@{\quad}r}
+3 & 0011 \\
+\ +4 & +\ 0100 \\
\hline
+7 & 0111
\end{array}
\qquad\qquad
\begin{array}{r@{\quad}r}
-2 & 1110 \\
+\ -6 & +\ 1010 \\
\hline
-8 & 1\,1000
\end{array}$$

$$\begin{array}{r@{\quad}r}
+6 & 0110 \\
+\ -3 & +\ 1101 \\
\hline
+3 & 1\,0011
\end{array}
\qquad\qquad
\begin{array}{r@{\quad}r}
+4 & 0100 \\
+\ -7 & +\ 1001 \\
\hline
-3 & 1101
\end{array}$$

Two's-Complement Addition

- Another way to view two's-complement system uses 4-bit counter shown in Fig. 3
  - Starting with arrow pointing to any number, add $+n$ to (subtract $+n$ from) it by counting up n times (counting down n times), that is, by moving arrow n positions clockwise (counterclockwise)
  - n must be small enough that discontinuity between -8 and $+7$ is not crossed

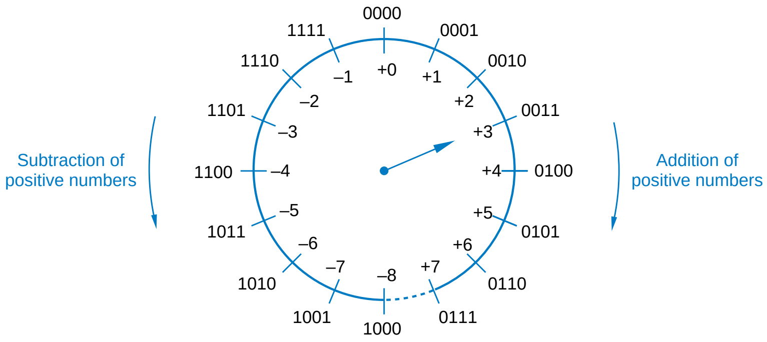

Figure 3: Modular counting representation of 4-bit two's-complement numbers.

Two's-Complement Addition

- In Fig. 3
  - We can subtract n (or add $-n$) by moving arrow $16 - n$ positions clockwise
  - $16 - n =$ 4-bit two's complement of $n =$ two's complement representation of $-n$
  - Thus, a negative number in two's-complement representation may be added to another number by adding 4-bit representations using ordinary binary addition

Two's-Complement Addition: Overflow

- **Overflow** occurs if an addition (of numbers of like sign) produces a result that exceeds the range of number system
  - In Fig. 3, overflow occurs during addition of positive (negative) numbers when we count past $+7$ (-8)

$$\begin{array}{r} -3 \\ +\ -6 \\ \hline -9 \end{array} \quad \begin{array}{r} 1101 \\ +\ 1010 \\ \hline 10111 \end{array} = +7 \qquad \begin{array}{r} +5 \\ +\ +6 \\ \hline +11 \end{array} \quad \begin{array}{r} 0101 \\ +\ 0110 \\ \hline 1011 \end{array} = -5$$

$$\begin{array}{r} -8 \\ +\ -8 \\ \hline -16 \end{array} \quad \begin{array}{r} 1000 \\ +\ 1000 \\ \hline 10000 \end{array} = +0 \qquad \begin{array}{r} +7 \\ +\ +7 \\ \hline +14 \end{array} \quad \begin{array}{r} 0111 \\ +\ 0111 \\ \hline 1110 \end{array} = -2$$

- **Overflow detection in addition**
  - An addition overflows if addends' signs are the same but sum's sign is different from addends'
  - Or equivalently, an addition overflows if c_{in} into and c_{out} out of sign position are different
    - Rows 4 and 5 in Tab. 3

Two's-Complement Subtraction

- Two's-complement numbers may be subtracted as if they were ordinary unsigned binary numbers
- But most circuits negate subtrahend by taking its two's complement, and then add it to minuend using normal rules for addition
  - Perform a bit-by-bit complement of subtrahend
  - Add complemented subtrahend to minuend with an initial carry (c_{in}) of 1 instead of 0
    - Using this method, only one addition is needed

$$\begin{array}{rrrl} & & 1 & — \; c_{\text{in}} \\ +4 & 0100 & 0100 & \\ -\;\;+3 & -\;\;0011 & +\;\;1100 & \\ \cline{1-3} +3 & & 10001 & \end{array} \qquad \begin{array}{rrrl} & & 1 & — \; c_{\text{in}} \\ +3 & 0011 & 0011 & \\ -\;\;+4 & -\;\;0100 & +\;\;1011 & \\ \cline{1-3} -1 & & 1111 & \end{array}$$

$$\begin{array}{rrrl} & & 1 & — \; c_{\text{in}} \\ +3 & 0011 & 0011 & \\ -\;\;-4 & -\;\;1100 & +\;\;0011 & \\ \cline{1-3} +7 & & 0111 & \end{array} \qquad \begin{array}{rrrl} & & 1 & — \; c_{\text{in}} \\ -3 & 1101 & 1101 & \\ -\;\;-4 & -\;\;1100 & +\;\;0011 & \\ \cline{1-3} +1 & & 10001 & \end{array}$$

Two's-Complement Subtraction

- **Overflow detection in subtraction**
  - Examine signs of minuend and *complemented* subtrahend, using the same rule as in addition
  - Or, carries into and out of sign position can be observed and overflow detected, using the same rule as in addition
  - Negating "extra" negative number results in overflow, when we add 1 in complementation process

$$-(-8) = -1000 = \begin{array}{r} 0111 \\ +\ 0001 \\ \hline 1000 \end{array} = -8$$

However, this number can still be used in additions and subtractions as long as final result does not exceed number range

$$\begin{array}{r} +4 \\ +\ -8 \\ \hline -4 \end{array} \quad \begin{array}{r} 0100 \\ +\ 1000 \\ \hline 1100 \end{array} \qquad\qquad \begin{array}{r} -3 \\ -\ -8 \\ \hline +5 \end{array} \quad \begin{array}{r} 1101 \\ -\ 1000 \\ \hline \end{array} \quad \begin{array}{r} 1 \text{ — } c_{\text{in}} \\ 1101 \\ +\ 0111 \\ \hline 10101 \end{array}$$

Ones'-Complement Addition and Subtraction

- In Tab. 6 for ones'-complement numbers
  - Starting at 1000_2 (-7_{10}) and counting up, we obtain each successive ones'-complement number by adding 1 to previous one, *except* at transition from 1111_2 (-0) to 0001_2 ($+1_{10}$)
  - We must add 2 instead of 1 whenever we count past 1111_2
  - Counting past 1111_2 can be detected by observing carry out of sign bit
- **End-around carry** rule for adding ones'-complement numbers
  - Perform a standard binary addition; if there is a carry out of sign position, add 1 to result

$$\begin{array}{rr} & +3 \\ + & +4 \\ \hline & +7 \end{array} \quad \begin{array}{rr} & 0011 \\ + & 0100 \\ \hline & 0111 \end{array} \qquad \begin{array}{rr} & +4 \\ + & -7 \\ \hline & -3 \end{array} \quad \begin{array}{rr} & 0100 \\ + & 1000 \\ \hline & 1100 \end{array} \qquad \begin{array}{rr} & +5 \\ + & -5 \\ \hline & -0 \end{array} \quad \begin{array}{rr} & 0101 \\ + & 1010 \\ \hline & 1111 \end{array}$$

$$\begin{array}{rr} & -2 \\ + & -5 \\ \hline & -7 \\ & \\ & \end{array} \quad \begin{array}{rr} & 1101 \\ + & 1010 \\ \hline & 10111 \\ + & 1 \\ \hline & 1000 \end{array} \qquad \begin{array}{rr} & +6 \\ + & -3 \\ \hline & +3 \\ & \\ & \end{array} \quad \begin{array}{rr} & 0110 \\ + & 1100 \\ \hline & 10010 \\ + & 1 \\ \hline & 0011 \end{array} \qquad \begin{array}{rr} & -0 \\ + & -0 \\ \hline & -0 \\ & \\ & \end{array} \quad \begin{array}{rr} & 1111 \\ + & 1111 \\ \hline & 11110 \\ + & 1 \\ \hline & 1111 \end{array}$$

Ones'-Complement Addition and Subtraction

- Following end-around carry rule, addition of a number and its ones' complement produces negative 0
  - In fact, an addition operation using this rule can never produce positive 0 unless both addends are positive 0
- **Ones'-complement subtraction** is done by complementing subtrahend and then adding
- Overflow rules for ones'-complement addition and subtraction are the same as for two's-complement

One's-Complement Addition and Subtraction

Table 7: Summary of addition and subtraction rules for binary numbers.

Number System	***Addition Rules***	***Negation Rules***	***Subtraction Rules***
Unsigned	Add the numbers. Result is out of range if a carry out of the MSB occurs.	Not applicable	Subtract the subtrahend from the minuend. Result is out of range if a borrow out of the MSB occurs.
Signed magnitude	(same sign) Add the magnitudes; overflow occurs if a carry out of MSB occurs; result has the same sign. (opposite sign) Subtract the smaller magnitude from the larger; overflow is impossible; result has the sign of the larger.	Change the number's sign bit.	Change the sign bit of the subtrahend and proceed as in addition.
Two's complement	Add, ignoring any carry out of the MSB. Overflow occurs if the carries into and out of MSB are different.	Complement all bits of the number; add 1 to the result.	Complement all bits of the subtrahend and add to the minuend with an initial carry of 1.
Ones' complement	Add; if there is a carry out of the MSB, add 1 to the result. Overflow if carries into and out of MSB are different.	Complement all bits of the number.	Complement all bits of the subtrahend and proceed as in addition.

Binary Multiplication

- **Unsigned binary multiplication**
  - Add a list of shifted multiplicands computed according to digits of multiplier

$$\begin{array}{r} 11 \\ \times\ 13 \\ \hline 33 \\ 11 \\ \hline 143 \end{array} \qquad\qquad \begin{array}{rl} 1011 & \text{multiplicand} \\ \times\quad 1101 & \text{multiplier} \\ \hline 1011 & \\ 0000 & \\ 1011 & \\ 1011 & \\ \hline 10001111 & \text{product} \end{array} \left.\begin{array}{l} \\ \\ \\ \\ \end{array}\right\} \text{shifted multiplicands}$$

  - This method lists all shifted multiplicands and then adds
    - Difficult to implement in a digital system

Binary Multiplication

- In a digital system, it is more convenient to add each shifted multiplicand as it is created to a *partial product*

$$\begin{array}{r} 11 \\ \times\ \ 13 \\ \hline \end{array} \qquad\qquad \begin{array}{rl} 1011 & \text{multiplicand} \\ \times\qquad 1101 & \text{multiplier} \\ \hline 0000 & \text{partial product} \\ 1011 & \text{shifted multiplicand} \\ \hline 01011 & \text{partial product} \\ 0000{\downarrow} & \text{shifted multiplicand} \\ \hline 001011 & \text{partial product} \\ 1011{\downarrow}{\downarrow} & \text{shifted multiplicand} \\ \hline 0110111 & \text{partial product} \\ 1011{\downarrow}{\downarrow}{\downarrow} & \text{shifted multiplicand} \\ \hline 10001111 & \text{product} \end{array}$$

- n-bit number $\times$ m-bit number
  - Resulting product requires at most $n + m$ bits
  - Shift-and-add algorithm requires m partial products and additions

Binary Multiplication

- **Signed multiplication**
  - Perform an unsigned multiplication of magnitudes
  - Make product positive if operands had same sign, negative if different signs
  - Convenient in signed-magnitude systems
  - In two's-complement system, obtaining magnitude of a negative number and negating unsigned product are nontrivial operations
- **Two's-complement multiplication**
  - Can be performed by a sequence of two's-complement additions of shifted multiplicands, except for the last step
  - MSB in a two's-complement number has a negative weight
  - The shifted multiplicand corresponding to MSB of multiplier must be negated before it is added to partial product

Binary Multiplication

-5	1011	multiplicand
$\times\ -3$	$\times$ 1101	multiplier
	00000	partial product
	11011	shifted multiplicand
	111011	partial product
	00000$\downarrow$	shifted multiplicand
	1111011	partial product
	11011$\downarrow\downarrow$	shifted multiplicand
	11100111	partial product
	00101$\downarrow\downarrow\downarrow$	shifted and negated multiplicand
	00001111	product

- One significant bit is gained at each step, and numbers are signed
    - Before adding each shifted multiplicand and k-bit partial product, change them to $k + 1$ significant bits by sign extension, as shown in color above
    - Each resulting sum has $k + 1$ bits; any carry out of MSB of $k + 1$-bit sum is ignored

Binary Division

- The simplest binary division is based on shift-and-subtract method
  - Mentally compare reduced dividend with multiples of divisor to determine which multiple of shifted divisor to subtract

Table 8: Example of long division for unsigned decimal and binary numbers.

	19		10011	quotient
11	)217	1011	)11011001	dividend
	11		1011	shifted divisor
	107		0101	reduced dividend
	99		0000	shifted divisor
	8		1010	reduced dividend
			0000	shifted divisor
			10100	reduced dividend
			1011	shifted divisor
			10011	reduced dividend
			1011	shifted divisor
			1000	remainder

Binary Division

- In a typical division algorithm

$$\begin{aligned}\text{Dividend: } & (n+m) \text{ bits}\\ \text{Divisor: } & n \text{ bits}\\ \text{Quotient: } & m \text{ bits}\\ \text{Remainder: } & n \text{ bits}\end{aligned}$$

- A division **overflows** if
  - Divisor is zero
  - Or quotient would take more than m bits to express
    - In most division circuits, $n = m$
- **Signed division**
  - Perform an unsigned division of magnitudes
  - Make quotient positive if operands had the same sign, negative if different signs
  - Remainder should be given the same sign as dividend
  - There are special techniques for performing division directly on two's-complement numbers (as in multiplication)

Binary Codes for Decimal Numbers

- Binary numbers are appropriate for internal computations of a digital system
- External interfaces of a digital system may read or display decimal numbers
  - People prefer to deal with decimal numbers
  - A decimal number is represented in a digital system by a string of bits
  - Some digital devices actually process decimal numbers directly
- **Code**
  - A set of *n*-bit strings in which different bit strings represent different numbers
  - A particular combination of *n* bit-values is called a **code word**
  - There may or may not be an arithmetic relationship between bit values in a code word and what it represents
  - At least four bits are needed to represent ten decimal digits

Binary Codes for Decimal Numbers

Table 9: Decimal codes.

Decimal digit	***BCD (8421)***	***2421***	***Excess-3***	***Biquinary***	***1-out-of-10***
0	0000	0000	0011	0100001	1000000000
1	0001	0001	0100	0100010	0100000000
2	0010	0010	0101	0100100	0010000000
3	0011	0011	0110	0101000	0001000000
4	0100	0100	0111	0110000	0000100000
5	0101	1011	1000	1000001	0000010000
6	0110	1100	1001	1000010	0000001000
7	0111	1101	1010	1000100	0000000100
8	1000	1110	1011	1001000	0000000010
9	1001	1111	1100	1010000	0000000001
			Unused code words		
	1010	0101	0000	0000000	0000000000
	1011	0110	0001	0000001	0000000011
	1100	0111	0010	0000010	0000000101
	1101	1000	1101	0000011	0000000110
	1110	1001	1110	0000101	0000000111
	1111	1010	1111	. . .	. . .

Binary Codes for Decimal Numbers

- **Binary-coded decimal (BCD)**
  - Encodes 0 through 9 by their 4-bit unsigned binary representations, 0000 through 1001
  - Code words 1010 through 1111 are not used
  - Conversion between BCD and decimal representations are a direct substitution of four bits for each decimal digit
  - **Packed-BCD representation**
    - Two BCD digits placed in one 8-bit byte
    - Thus, one byte may represent 0 to 99 as opposed to 0 to 255 for a normal unsigned 8-bit binary number
  - Signed BCD numbers have one extra digit position for sign
  - **Signed-magnitude** representation
    - Encoding of sign bit string is arbitrary
  - **10's-complement** representation
    - 0000 = plus, 1001 = minus

Binary Codes for Decimal Numbers

- Addition of BCD digits
  - Similar to adding 4-bit unsigned binary numbers
  - But if a result exceeds 1001, it is corrected by adding 6
  - Carry is produced into next digit position if either initial binary addition or correction-factor addition produces a carry

$$\begin{array}{rrl} 5 & 0101 & \\ +\;\;9 & +\;1001 & \\ \hline 14 & 1110 & \\ & +\;0110 & \text{— correction} \\ \cline{2-2} 10{+}4 & 1\;0100 & \end{array} \qquad \begin{array}{rr} 4 & 0100 \\ +\;\;5 & +\;0101 \\ \hline 9 & 1001 \end{array}$$

$$\begin{array}{rrl} 8 & 1000 & \\ +\;\;8 & +\;1000 & \\ \hline -16 & 1\;0000 & \\ & +\;0110 & \text{— correction} \\ \cline{2-2} 10{+}6 & 1\;0110 & \end{array} \qquad \begin{array}{rrl} 9 & 1001 & \\ +\;9 & +\;1001 & \\ \hline 18 & 1\;0010 & \\ & +\;0110 & \text{— correction} \\ \cline{2-2} 10{+}8 & 1\;1000 & \end{array}$$

Binary Codes for Decimal Numbers

- **Weighted code**
  - In a weighted code, each decimal digit can be obtained from its code word by assigning a fixed weight to each code-word bit
  - E.g., BCD (= *8421 code*) in which the weights for bits are $8, 4, 2, 1$
  - **2421 code** is *self-complementing*
    - Code word for 9s' complement of any digit may be obtained by complementing individual bits of digit's code word
- **Excess-3 code**
  - A self-complementing code
  - Code word for each decimal digit is corresponding BCD code word plus 0011_2
  - Because code words follow a standard binary counting sequence, standard binary counters can be made to count in excess-3 code

Binary Codes for Decimal Numbers

- One advantage of using more than minimum number of bits in a code is an error-detecting property
- **Biquinary code**
  - Uses seven bits
  - First two bits indicate range of number, 0-4 or 5-9
  - Last five bits indicate number in the selected range
  - Has error-detecting property
    - If any one bit is accidentally flipped, result is not a decimal digit
    - Of 128 possible 7-bit code words, only 10 are valid; the rest can be flagged as errors if appear
- **1-out-of-10 code**
  - The sparsest encoding for decimal digits
  - Uses 10 out of 1024 possible 10-bit code words

Gray Code

- In electromechanical applications of digital systems, sometimes an input sensor should produce a digital value that indicates a mechanical position

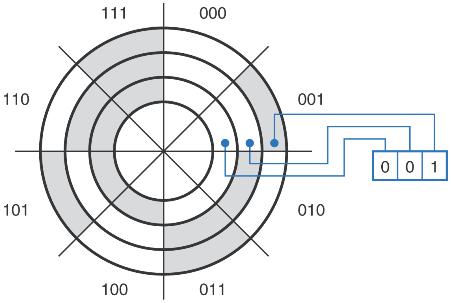

Figure 4: A mechanical encoding disk using a 3-bit binary code.

- In Fig. 4, dark areas of disk are connected to a signal source corresponding to logic 1, and light areas are unconnected (logic 0)

Gray Code

- In Fig. 4
    - Problem when disk is positioned at certain boundaries between regions
    - E.g., if disk is positioned right on boundary between 001 and 010 regions
        - Both 001 and 010 are acceptable
        - But because mechanical assembly is not perfect, incorrect reading of 000 or 011 is possible
    - This sort of problem can occur at any boundary where more than one bit changes
- Encoding-disk problem can be solved by **Gray code**
    - A digital code in which only one bit changes between each pair of successive code words

Gray Code

Table 10: A comparison of 3-bit binary code and Gray code.

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

Gray Code

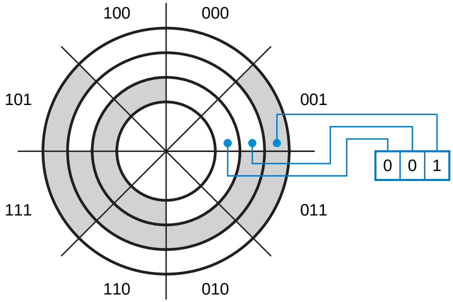

Figure 5: A mechanical encoding disk using a 3-bit Gray code.

Gray Code

- Gray code is a *reflected code*; it can be constructed recursively (with any number of bits)
  1. A 1-bit Gray code has two code words, 0 and 1
  2. First 2^n code words of an $(n+1)$-bit Gray code equal code words of an n-bit Gray code, written in order with a leading 0 appended
  3. Last 2^n code words of an $(n+1)$-bit Gray code equal code words of an n-bit Gray code, but written in reverse order with a leading 1 appended
- A method to derive an n-bit Gray-code code word directly from corresponding n-bit binary code word
  1. Bits of an n-bit binary or Gray-code code word are numbered from right to left, from 0 to $n-1$
  2. Bit i of a Gray-code code word is 0 if bits i and $i+1$ of corresponding binary code word are the same, else bit i is 1
     - When $i+1=n$, bit n of binary code word is considered to be 0

Character Codes

- Most of information processed by computers is nonnumeric
- **Text** is the most common type of nonnumeric data
    - Strings of characters from some character set
    - Each character is represented by a bit string according to an established convention
- **ASCII** (American Standard Code for Information Interchange)
    - The most commonly used character code
    - Each character is represented with a 7-bit string
    - A total of 128 different characters

Character Codes

Table 11: ASCII, Standard No. X3.4-1968 of the American National Standards Institute.

		$b_6b_5b_4$ (column)							
$b_3b_2b_1b_0$	Row (hex)	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

Character Codes

Table 12: ASCII, Standard No. X3.4-1968 of the American National Standards Institute.

Control codes

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	Synchronize
BEL	Bell	ETB	End transmitted 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 or rubout

Codes for Actions, Conditions, and States

- Numbers, positions, and characters are "data"
- In digital system design, we often encounter nondata applications
  - A string of bits must be used to control an action, to flag a condition, or to represent current state of hardware
  - The most commonly used type of code for such an application is a binary code
- If there are n different actions, conditions, or states, represent them with a b-bit **binary code** with $b = \lceil \log_2 n \rceil$
- Consider a traffic-light controller
  - N-S: north-south street
  - E-W: east-west street
  - Signals at intersection of N-S and E-W street might be in any of six states listed in Tab. 13

Codes for Actions, Conditions, and States

Table 13: States in a traffic-light controller.

	Lights						
State	***N-S green***	***N-S yellow***	***N-S red***	***E-W green***	***E-W yellow***	***E-W red***	***Code word***
N-S go	ON	off	off	off	off	ON	000
N-S wait	off	ON	off	off	off	ON	001
N-S delay	off	off	ON	off	off	ON	010
E-W go	off	off	ON	ON	off	off	100
E-W wait	off	off	ON	off	ON	off	101
E-W delay	off	off	ON	off	off	ON	110

- In Tab. 13
  - Six states can be encoded in three bits
  - Only six of eight possible 3-bit code words are used, and assignment of them to states is arbitrary, so many other encodings possible
    - An encoding which minimizes circuit cost or optimizes some other parameter (like design time) should be chosen

Codes for Actions, Conditions, and States

- Consider a system with *n* devices, each can perform a certain action
  - Devices may be enabled to operate only one at a time
  - **Binary code**
    - Shown in Fig. 6 (a)
    - Control unit produces a binary-coded "device-select" word with $\lceil \log_2 n \rceil$ bits to indicate which device is enabled at any time
    - "Device-select" code word is applied to each device, which compares it with its own "device ID" to determine whether it is enabled
    - Binary code has fewest bits, but is not always the best choice
  - **1-out-of-n code**
    - An *n*-bit code in which valid code words have one bit equal to 1 and the rest of bits equal to 0
    - Shown in Fig. 6 (b)
    - Each bit of code word is connected directly to enable input of a corresponding device
    - Simplifies design of devices, since they no longer have device IDs
  - **Inverted 1-out-of-n code**
    - Valid code words have one 0 bit and the rest of bits equal to 1
- In complex systems, a combination of coding techniques may be used

Codes for Actions, Conditions, and States

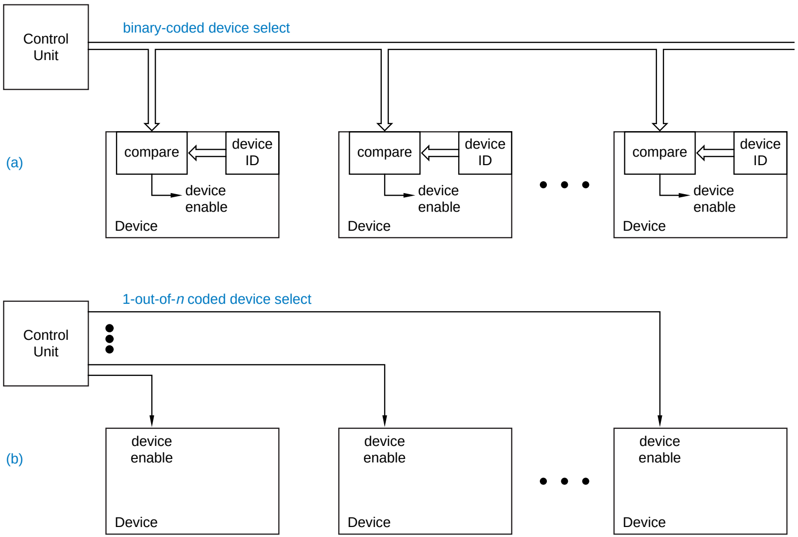

Figure 6: Control structure for a digital system with n devices: (a) using a binary code; (b) using a 1-out-of-n code.

Codes for Actions, Conditions, and States

- **m-out-of-n code**
  - A generalization of 1-out-of-*n* code
  - Valid code words have *m* bits equal to 1 and the rest of bits equal to 0
  - Can be detected with an *m*-input AND gate
  - Total number of code words $= \binom{n}{m}$
- **8B10B code**
  - A variation of an *m*-out-of-*n* code
  - Used in 802.3z Gigabit Ethernet standard
  - Uses 10 bits to represent 256 valid code words, or 8 bits worth of data
  - Most code words use a 5-out-of-10 coding
  - Since $\binom{10}{5} = 252$, some 4- and 6-out-of-10 words are also used

Codes for Serial Data Transmission and Storage

- **Parallel data**
  - Most digital systems transmit and store data in a parallel format
  - In parallel data transmission, a separate signal line is provided for each bit of a data word
  - In parallel data storage, all of bits of a data word can be written or read simultaneously
- **Serial data**
  - Serial formats allow data to be transmitted or stored one bit at a time
  - Serial formats can reduce cost and simplify certain design problems

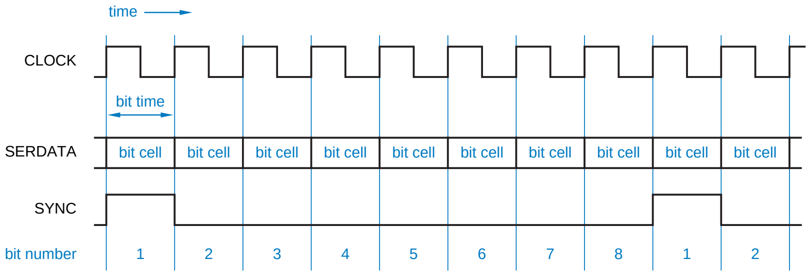

Figure 7: Basic concepts of serial data transmission.

Codes for Serial Data Transmission and Storage

- **Clock** signal defines the rate at which bits are transmitted, one bit per clock cycle
- **Bit rate** in bits per second (bps) equals clock frequency in cycles per second (Hz)
  - Reciprocal of bit rate is called **bit time** and equals clock period in seconds (s)
- The time occupied by each bit is called a **bit cell**
- Format of actual signal that appears on line during each bit cell depends on **line code**
- **Non-Return-to-Zero (NRZ)**
  - The simplest line code
  - A 1 is transmitted by placing a 1 on line for entire bit cell, and a 0 is transmitted as a 0
- **Synchronization signal**
  - A serial data-transmission or storage system needs some way of identifying significance of each bit in serial stream
  - In Fig. 7, SYNC is 1 for the first bit of each byte

Codes for Serial Data Transmission and Storage

- A minimum of three signals are needed to recover a serial data stream
  - a clock, a synchronization signal, and serial data itself
  - In some applications, a separate wire is used for each of these signals
    - Reducing number of wires from *n* to three is savings enough
  - But in many applications, cost of having three separate signals is still too high
    - Such systems combine all three signals into a single serial data stream
    - They use sophisticated analog and digital circuits to recover clock and synchronization information from data stream

Serial Line Codes

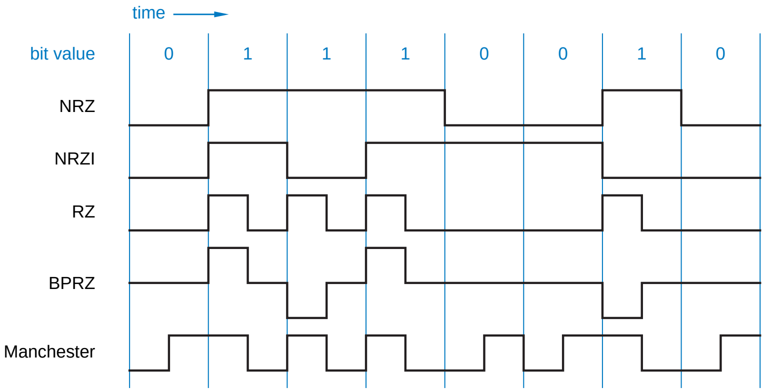

Figure 8: Commonly used line codes for serial data.

Serial Line Codes

- **NRZ code**
  - Each bit value is sent on line for the entire bit cell
  - The simplest and most reliable coding scheme for short-distance transmission
  - It requires a clock signal to be sent along with data to define bit cells
  - E.g., without a clock signal, NRZ waveform in Fig. 8 might be interpreted as 01010
- **Digital phase-locked loop (DPLL)**
  - An analog/digital circuit used to recover a clock signal from a serial data stream
  - DPLL works only if serial data stream contains enough 0-to-1 and 1-to-0 transitions
  - With NRZ-coded data, DPLL works only if data does not contain any long, continuous streams of 1s or 0s

Serial Line Codes

- **Transition-sensitive media**
  - They cannot transmit or store absolute 0 or 1 levels, only transitions between two discrete levels
  - E.g., a magnetic disc or tape stores information by changing polarity of medium's magnetization in regions corresponding to the stored bits
  - NRZ format cannot be used on transition-sensitive media
    - Data in Fig. 8 might be interpreted as 01110010 or 10001101
- **Non-Return-to-Zero Invert-on-1s (NRZI)**
  - Can be used on transition-sensitive media
  - Sends a 1 as opposite of the level that was sent during previous bit cell, and a 0 as same level
  - A DPLL can recover clock from NRZI-coded data as long as data does not contain any long, continuous streams of 0s
- **Return-to-Zero (RZ)**
  - Similar to NRZ except that, for a 1 bit, 1 level is transmitted only for a fraction of bit time, usually 1/2
  - Data with a lot of 1s create lots of transitions for a DPLL to use to recover clock
  - A long string of 0s makes clock recovery impossible

Serial Line Codes

- **DC balance**
  - DC balanced serial data stream has an equal number of 1s and 0s
  - Required by some transmission media, e.g. high-speed fiber-optic links
  - NRZ, NRZI or RZ data have no guarantee of DC balance
    - User data streams usually have more 1s than 0s or vice versa
- **Balanced code**
  - Each code word has an equal number of 1s and 0s
  - Can be achieved by using a few extra bits to code user data
  - E.g., 8B10B code
    - Codes 8 bits into 10 bits in a mostly 5-out-of-10 code
    - Only $\binom{10}{5} = 252$ balanced, but $\binom{10}{4} = \binom{10}{6} = 210$ unbalanced
    - 8B10B associates with each extra 8-bit value a *pair* of unbalanced code words, one 4-out-of-10 ("light") and the other 6-out-of-10 ("heavy")
    - Coder keeps track of *running disparity*, a bit of information indicating whether the last unbalanced code word transmitted was heavy or light
    - When transmitting another unbalanced code word, coder selects the one of the pair with opposite weight
    - $252 + 210 = 462$ code words to encode 8 bits
    - Not all unbalanced code words are used

Serial Line Codes

- A DPLL can recover a clock signal, but not byte synchronization
  - Byte synchronization is achieved by embedding special patterns into long-term serial data stream, recognizing them digitally, and then "locking" onto them
  - E.g., if IDLE = 1011011000 which is sent continuously at system startup, then beginning of code word can be recognized as the bit after three 0s in a row
  - Successive code words, even if not IDLE, can be expected to begin at every tenth bit time thereafter
- **Bipolar Return-to-Zero (BPRZ)** code
  - Transmits three signals levels: $+1, 0, -1$
  - Is like RZ except 1s are alternatively transmitted as $+1$ and -1
  - Is DC balanced, hence possible to send BPRZ streams over transmission media that cannot tolerate a DC component
    - BPRZ code has been used in T1 digital telephone links for decades
  - Possible to recover a clock signal if there are not too many 0s in a row
  - **Zero-code suppression**
    - If one of bytes is all 0s, The Phone Company changes second-LSB to 1
    - In data applications of T1 links, LSB of each byte is set to 1
      64-Kbps channel $\longrightarrow$ 56-Kbps

Serial Line Codes

- **Manchester** or **diphase** code
  - Provides at least one transition per bit cell, regardless of transmitted data pattern
  - Makes it very easy to recover clock
  - A 0 is encoded as a 0-to-1 transition in the middle of bit cell, and a 1 as a 1-to-0 transition
  - Its weakness is that it has more transitions per bit cell than other codes
    - Requires more media bandwidth to transmit a given bit rate
  - Is DC balanced

Serial Line Codes

- **UART** (Universal Asynchronous Receiver/Transmitter)
  - A circuit that takes bytes of data and transmits individual bits in a sequential fashion
  - At destination, a second UART re-assembles bits into complete bytes
  - Serial line is high during IDLE state
  - Serial bit stream uses following sequence
    1. Start bit (a low bit): For synchronization
    2. Data bits (LSB first): No. of data bits per frame configurable
    3. Parity bit (if enabled): Type of parity configurable
    4. Stop bits (at least one high bit): Indicates end of a frame - returns serial line to IDLE state - length of bits configurable

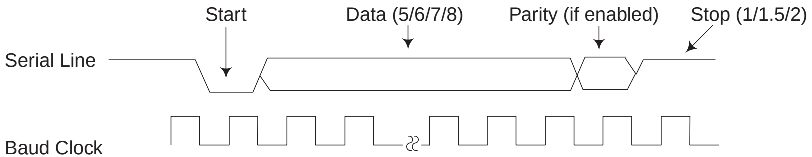

Figure 9: Standard serial data format.

Serial Line Codes

- UART baud generator
  - Programmable
    - User-defined value in read-write divisor register
  - Operates at system clock frequency
  - Creates a baud rate clock

    Divisor register value = (system clock frequency) / (baud rate $\times$ 16)

    - Transmitter shifts data out at baud rate
  - Creates a receiver reference clock
    - Sent along with serial data to receiver
    - Baud rate $\times$ 16 clock output
    - Each data bit is as long as 16 clock pulses
    - Receiver tests state of incoming signal on each clock pulse, looking for beginning of start bit
    - If apparent start bit lasts at least one-half of bit time, it is valid, if not, the pulse is ignored
    - After waiting a further bit time, state of line is again sampled and resulting level clocked into a shift register
    - After required number of bit periods (5 to 8 bits) have elapsed, contents of shift register is made available (in parallel fashion) to receiving system

References

John F. Wakerly, *Digital Design: Principles and Practices (4th Edition)*, Prentice Hall, 2005.