Master of Computer Applications

23MCAC105 –
Advanced Computer Architecture

Credits: 3

L: T: P: E – 3-0-0-3

Prepared by
Dr. A. Rengarajan, Professor

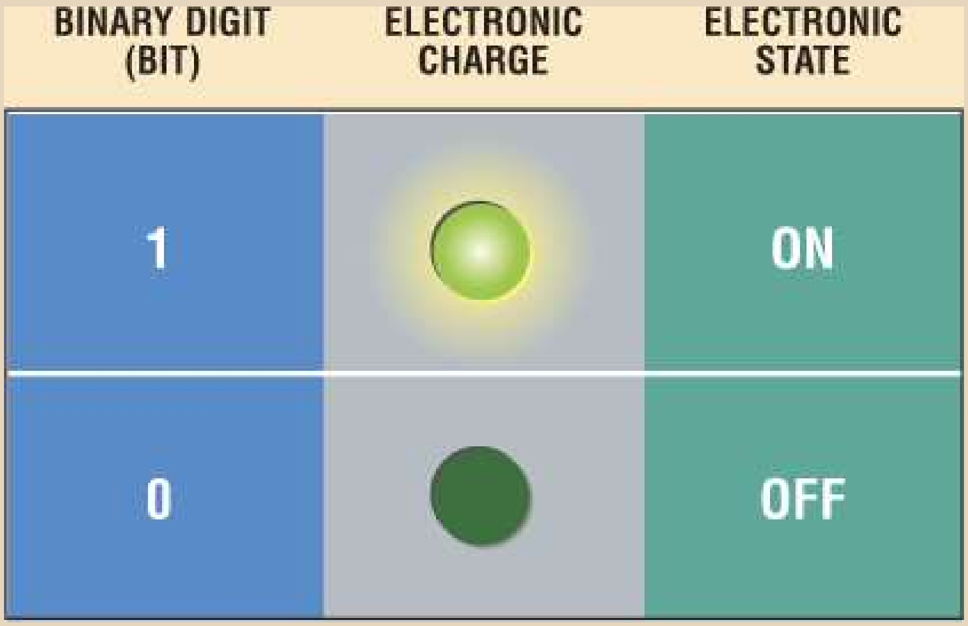
Module – 2

DATA REPRESENTATION

- Data types : Number systems
- Code conversions
- Complements
- Fixed point representation
- Addition
- Subtraction
- Multiplication
- Division
- Floating point representation
- Other Binary codes
- Error Detection Codes

Data Types – Number Systems Data Representation

- How do computers represent data?
  - **Most computers are digital**



- Recognize only two discrete states: on or off
- Use a **binary system** to recognize two states
- Use number system with two unique digits: 0 and 1, called **bits** (short for binary digits)
  - Smallest unit of data computer can process

Data Representation

- How is a letter converted to binary form and back?

Step 1.
The user presses the capital letter **D** (shift+D key) on the keyboard.

Step 2.
An electronic signal for the capital letter **D** is sent to the system unit.

Step 3.
The signal for the capital letter **D** is converted to its ASCII binary code (01000100) and is stored in memory for processing.

Step 4.
After processing, the binary code for the capital letter **D** is converted to an image, and displayed on the output device.

Name	Abbr.	Size
Kilo	K	2^10 = 1,024
Mega	M	2^20 = 1,048,576
Giga	G	2^30 = 1,073,741,824
Tera	T	2^40 = 1,099,511,627,776
Peta	P	2^50 = 1,125,899,906,842,624
Exa	E	2^60 = 1,152,921,504,606,846,976
Zetta	Z	2^70 = 1,180,591,620,717,411,303,424
Yotta	Y	2^80 = 1,208,925,819,614,629,174,706,176

MEMORY AND STORAGE SIZES

Term	Abbreviation	Approximate Size	Exact Amount	Approximate Number of Pages of Text
Kilobyte	KB or K	1 thousand bytes	1,024 bytes	1/2
Megabyte	MB	1 million bytes	1,048,576 bytes	500
Gigabyte	GB	1 billion bytes	1,073,741,824 bytes	500,000
Terabyte	TB	1 trillion bytes	1,099,511,627,776 bytes	500,000,000

Data Representation

- Computer technology is based on integrated circuits (IC), which themselves are based on transistors
  - The idea is that ICs store circuits that operate on electrical current by either letting current pass through, or blocking the current
    - So, we consider every circuit in the computer to be in a state of on or off (1 or 0)
    - Because of this, the computer can store information in binary form
    - But we want to store information from the real world, information that consists of numbers, names, sounds, pictures, instructions
  - So we need to create a representation for these types of information using nothing but 0s and 1s
    - That’s the topic for this chapter

Numbering Systems

- In mathematics, there are many different numbering systems, we are used to decimal (base 10)
  - In a numbering system base k (also known as radix k), the digits available are 0..k-1 (so in base 10, we use digits 0-9)
  - Each digit represents a different power of the base
    - For instance, 572 in decimal has 5 100s, 7 10s and 2 1s, so we have 100s column, 10s column 1s column
    - If the value was in octal (base 8), it would instead have 5 64s (8^2), 7 8s and 2 1s
    - In base 2, our digits are only 1 and 0 and our powers are all powers of 2 (1, 2, 4, 8, 16, 32, etc)
      - NOTE: we denote a value's numbering system by placing a subscript of the base after the number as in 572_8 or 572_{10}, however, if a number is omitted, it is assumed to be base 10
        - » For this course, we will also omit the 2 for base 2 numbers for convenience
  - So we can store decimal numbers in any numbering system

Conversions

- There is a simple formula to convert a number from a given base into base 10:
  - $abcd_e = a * e^3 + b * e^2 + c * e^1 + d * e^0$
  - Note that $e^0 = 1$, so the rightmost column will always be the 1s column no matter what the base is
    - Example: $7163_8 = 7 * 8^3 + 1 * 8^2 + 6 * 8^1 + 3 * 8^0 = 3699_{10}$
- To convert from base 10 to another base, e:

```
// assume value is the value to be converted
sum = 0;
for(j=numColumns(value) – 1;  j>= 0; j--)
{
        divisor = e^j;
        temp = value / divisor;
        sum = sum * 10 + temp;
        value = value – temp * divisor;
}
```

Example: 3699 to base 8

```
sum = 0 * 10 + 3699 / 8^3 = 7
value = 3699 – 7 * 8^3 = 115
sum = 7 * 10 + 115 / 8^2 = 71
value = 115 – 1 * 8^2 = 51
sum = 71 * 10 + 51 / 8^1 = 716
value = 51 – 6 * 8^1 = 3
sum = 716 * 10 + 3 / 8^0 = 7163
value = 3 – 3 * 8^0 = 0
```

Binary

- In CS we concentrate on binary (base 2)
  - Why?
    - Because digital components (from which the computer is built) can be in one of two states
      - on or off
    - We use 1 and 0 to represent these two states
      - we want to store/manipulate bits (*b*inary dig*its*)
  - We want to develop a method for representing information in binary
    - numbers (positive, negative, integer, floating point, fraction), strings of characters, booleans, images, sounds, programming instructions
  - For unsigned integer values, we can store them directly using binary
    - we convert from one to the other using the conversion algorithms on the previous slide where base = 2

Powers of 2
$2^{-2} = \frac{1}{4} = 0.25$
$2^{-1} = \frac{1}{2} = 0.5$
$2^0 = 1$
$2^1 = 2$
$2^2 = 4$
$2^3 = 8$
$2^4 = 16$
$2^5 = 32$
$2^6 = 64$
$2^7 = 128$
$2^8 = 256$
$2^9 = 512$
$2^{10} = 1{,}024$
$2^{15} = 32{,}768$
$2^{16} = 65{,}536$

Some useful powers of 2 – these illustrate the values of each column (1, 2, 4, 8,…)

Binary Conversions

- We can simplify the more general base e conversions when dealing with decimal and binary
  - Convert from binary to decimal
    - For each 1 in the binary number, add that column's power of 2
      - e.g., 11001101 = 128 + 64 + 8 + 4 + 2 = 206
  - Convert from decimal to binary
  - Two approaches (see next slide)
    - Approach 1: divide number by 2, recording quotient and remainder, continue to divide quotient by 2 until you reach 0, binary value is the remainder written backward
    - Approach 2: find all powers of 2 that make up the number, for each power of 2, place a 1 in that corresponding column, otherwise 0

Decimal → Binary

Convert 91 to binary:

91 / 2 = 45, remainder 1
45 / 2 = 22, remainder 1
22 / 2 = 11, remainder 0
11 / 2 = 5, remainder 1
5 / 2 = 2, remainder 1
2 / 2 = 1, remainder 0
1 / 2 = 0, remainder 1

91 = 1011011_2

We can test this:
1011011 = 64 + 16 + 8 + 2 + 1 = 91

Convert 91 to binary:

Largest power of 2: 64
91 – 64 = 27
Largest power of 2: 16
27 – 16 = 11
Largest power of 2: 8
11 – 8 = 3
Largest power of 2: 2
3 – 2 = 1
Largest power of 2: 1
1 – 1 = 0

64 + 16 + 8 + 2 + 1 = 1011011_2

Abbreviations

- We can't store much in 1 bit (0 or 1) so we group bits together into larger units
  - 1 byte = 8 bits
  - 1 word = 4 bytes (usually)
  - But we can't store much in 1-4 bytes either, so we refer to larger storage capacities using abbreviations as shown below
    - notice how each unit is $2^{\text{power of 10}}$
    - on the right we see similar abbreviations for time units

Prefix	Symbol	Power of 10	Power of 2	Prefix	Symbol	Power of 10	Power of 2
Kilo	K	1 thousand = 10^3	$2^{10} = 1024$	Milli	m	1 thousandth = 10^{-3}	2^{-10}
Mega	M	1 million = 10^6	2^{20}	Micro	μ	1 millionth = 10^{-6}	2^{-20}
Giga	G	1 billion = 10^9	2^{30}	Nano	n	1 billionth = 10^{-9}	2^{-30}
Tera	T	1 trillion = 10^{12}	2^{40}	Pico	p	1 trillionth = 10^{-12}	2^{-40}
Peta	P	1 quadrillion = 10^{15}	2^{50}	Femto	f	1 quadrillionth = 10^{-15}	2^{-50}
Exa	E	1 quintillion = 10^{18}	2^{60}	Atto	a	1 quintillionth = 10^{-18}	2^{-60}
Zetta	Z	1 sextillion = 10^{21}	2^{70}	Zepto	z	1 sextillionth = 10^{-21}	2^{-70}
Yotta	Y	1 septillion = 10^{24}	2^{80}	Yocto	y	1 septillionth = 10^{-24}	2^{-80}

Some Decimal and Binary Numbers

Decimal	4-Bit Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

- The table to the left shows for the first 16 decimal values, what the corresponding binary values are
  - Can you expand this table to the first 32 decimal values using 5 bits?
  - For convenience
    - we often group bits into 3s and write the value in octal
    - or in groups of 4 bits and write the value in hexadecimal (base 16)
  - In hexadecimal, since we cannot write numbers 10-16 as single digits, we use the letters A – F
    - The number $FA3_{16}$ is F (15) in the 16^2 column, A (10) in the 16^1 column and 3 in the 16^0 column

Binary Coded Decimal (BCD)

Digit	BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
Zones	
1111	Unsigned
1100	Positive
1101	Negative

- Some programming languages provide the decimal type (COBOL for one)
  - This type stores a decimal digit in ½ a byte using the binary equivalent
  - Since a digit is 0-9, this leaves 6 codes unused
  - Some architectures will use these 6 extra codes for special characters such as ‘$’, ‘.’, etc or to reference whether the decimal value is unsigned, positive or negative

Fractions

- We can extend our unsigned representational system of binary to include a decimal point
  - After the decimal point, the i exponent, in 2^i, becomes negative
    - So, we now have the ½ column, the ¼ column, etc
      - 1011.1001 =
      - $1*2^3 + 0*2^2 + 1*2^1 + 1*2^0 + 1*2^{-1} + 0*2^{-2} + 0*2^{-3} + 1*2^{-4}$ =
      - 8 + 2 + 1 + ½ + 1/16 =
      - 11 9/16 = 11.5625
    - What is .4304? Use 8-bits with 4 fraction bits
      - .4304 has a .25, .125, .03125, .015625, and more fractions, but this exceeds the number of fraction bits so the number is 0000.0110
      - But 0000.0110 = .125 + 0.3125 = .375, we have a loss in precision!
    - In the fraction representation, our decimal point is typically fixed, so this is often known as *fixed point representation*
    - We will cover a floating point representation later

Signed Integers

- So far we have treated all of our numbers as unsigned (or positive only)
  - To implement signed integers (or signed fractions), we need a mechanism to denote the sign itself (positive or negative)
    - Unfortunately, this introduces new problems, so we will see 3 different approaches, all of which add a special bit known as the *sign bit*
  - If the sign bit is 0, the number is positive
  - If the sign bit is 1, the number is negative
    - If we have an 8 bit number, does this mean that we now need 9 bits to store it with one bit used exclusively for the sign?

Signed Magnitude

- The first signed integer format is signed magnitude where we add a bit to the front of our numbers that represents the sign
  - In 4 bits, 3 = 0011 and –3 = 1011
    - Notice in 4 bits, we can store 16 numbers in unsigned magnitude (0000 to 1111, or decimal 0 to 15) but in signed magnitude we can only store 15 numbers (between –7, or 1111, and +7, 0111), so we lose a number
  - Two problems:
    - 0 is now represented in two ways: 0000, 1000, so we lose the ability to store an extra number since we have two 0s
    - We cannot do ordinary arithmetic operations using signed magnitude
      - we have to “strip” off the sign bit, perform the operation, and insert the sign bit on the new answer – this requires extra hardware

Complement

Complements are used in the digital computers in order to simplify the subtraction operation and for the logical manipulations. For each radix-r system (radix r represents base of number system) there are two types of complements.

S.N.	Complement	Description
1	Radix Complement	The radix complement is referred to as the r's complement
2	Diminished Radix Complement	The diminished radix complement is referred to as the (r-1)'s complement

Binary system complements

As the binary system has base r = 2. So the two types of complements for the binary system are 2's complement and 1's complement.

One's Complement

- An alternative approach to signed magnitude is one's complement where the first bit is again a sign bit
- But negative numbers are stored differently from positive numbers
  - Positive number stored as usual
  - Negative number – all bits are inverted
    - 0s become 1s, 1s become 0s
  - Example: +19 in 6 bits = 010011, -19 = 101100
    - The first bit is not only the sign bit, but also part of the number
  - Notice that we still have two ways to represent 0, 000000 and 111111
    - So, we won't use one's complement

Two's Complement

- Positive numbers remain the same
- Negative numbers: derived by flipping each bit and then adding 1 to the result
  - +19 in 6 bits = 010011,
  - -19 in 6 bits = 101101
    - 010011 ➔ 101100 +1 ➔ 101101
  - To convert back, flip all bits and add 1
    - 101101 ➔ 010010 + 1 ➔ 010011
  - While this is harder, it has two advantages
    - Only 1 way to represent 0 (000000) so we can store 1 extra value that we lost when we tried signed magnitude and one's complement
    - Arithmetic operations do not require "peeling" off the sign bit

4-bit Two's Complement

Binary	Decimal
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7

Some Examples

Represent 83 and –83 using 8 bits in all 3 signed representations:

+83 = 64 + 16 + 2 + 1 = 01010011 (in all 3 representations)

-83:

Signed magnitude = 11010011 (sign bit is 1 for negative)

One's complement = 10101100 (flip all bits from +83)

Two's complement = 10101101 (flip all bits from +83 and add 1)

Convert 11110010 into a decimal integer in all 4 representations

Unsigned magnitude = 128 + 64 + 32 + 16 + 2 = 242

Signed magnitude = -114 (negative, 1110010 = 114)

One's complement = -13 (leading bit = 1, the number is negative, flip all bits ➔00001101 = 13)

Two's complement = -14 (negative, so flip all bits and add 1 ➔ 00001101 + 1 = 00001110 = 14)

Addition

- This operation is much like decimal addition except that you are only adding 1s and 0s
  - Add each column as you would in decimal, write down the sum and if the sum > 1, carry a 1 to the next column
  - Four possibilities:
    - Sum of the two digits (and any carry in) = 0, write 0, carry 0
    - Sum = 1, write 1, carry 0
    - Sum = 2, write 0, carry 1 (this represents 10 = 2)
    - Sum = 3, write 1, carry 1 (this represents 11 = 3)

Examples:

1 + 1 = 2, write 0, carry 1

```
  01000101
+ 00001111
  01010100
```

```
  11111111
+ 10101010
 110101001
```

The carry out of this last bit causes *overflow*

Subtraction

- There are two ways we could perform subtraction
  - As normal, we subtract from right to left with borrows now being 2 instead of 10 as we move from one column to the next
  - Or, we can negate the second number and add them together (36 – 19 = 36 + -19)
    - We will use the latter approach when implementing a subtraction circuit as it uses the same circuit as addition

Examples:

borrow 2 from the previous column

```
  11010100                      11010100 →  11010100
- 00110011                    - 00110011  + 11001101
  10100001                                 110100001
```

Notice the overflow in this case too, but it differs from the last example because we are using two's complement

Overflow Rules

- In unsigned magnitude addition
  - a carry out of the left-most bit is also an overflow
- In unsigned magnitude subtraction
  - overflow will occur in subtraction if we must borrow prior to the left-most bit
- In two's complement addition/subtraction
  - if the two numbers have the same sign bit and the sum/difference has a different sign bit, then overflow

Below we see examples of four *signed* additions

Expression	Result	Carry?	Overflow?	Correct Result?
0100(+4)+0010(+2)	0110(+6)	No	No	Yes
0100(+4)+0110(+6)	1010(-6)	No	Yes	No
1100(-4)+1110(-2)	1010(-6)	Yes	No	Yes
1100(-4)+1010(-6)	0110(+6)	Yes	Yes	No

Multiplication

- Multiplication is much like as you do it in decimal
  - Line up the numbers and multiply the multiplicand by one digit of the multiplier, aligning it to the right column, and then adding all products together
    - but in this case, all values are either going to be multiplied by 0 or 1
  - So in fact, multiplication becomes a series of shifts and adds:

```
      110011
    * 101001
      110011
     000000
    000000
   110011
  000000
 110011
Add these values
```

This is the same as:

110011 * 101001 =
110011 * 100000 + 110011 * 00000 +
110011 * 1000 + 110011 * 000 +
110011 * 00 + 110011 * 1
= 110011 * 100000 + 110011 * 1000 + 110011 * 1

We will use a tabular approach for simplicity (see next slides)

Multiplication Algorithm

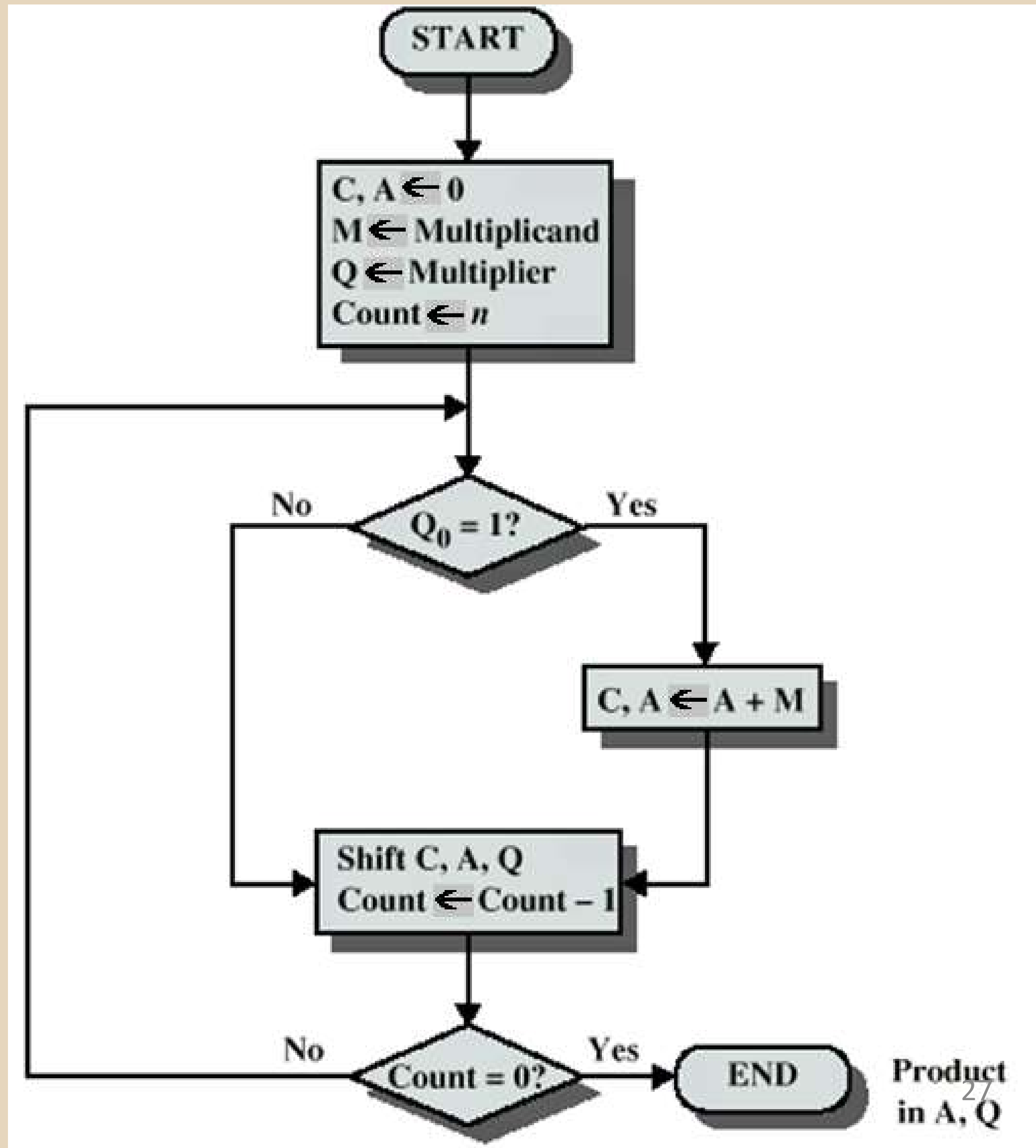
NOTE: this algorithm wor numbers are positive. If we have negative values in two’s complement, we will use a different algorithm

A is the accumulator

M and Q are temporary registers

C is a single bit storing the carry out of the addition of A and M

The result is stored in the combination of registers A and Q (A storing the upper half of the product, Q the lower half)



Example

C	A	Q	M		
0	0000	1101	1011	Initial Values	
0	1011	1101	1011	Add	First Cycle
0	0101	1110	1011	Shift	
0	0010	1111	1011	Shift	Second Cycle
0	1101	1111	1011	Add	Third Cycle
0	0110	1111	1011	Shift	
1	0001	1111	1011	Add	Fourth Cycle
0	1000	1111	1011	Shift	

Need 8 bit location to store result of two 4 bit multiplications

First, load the multiplicand in M and the multiplier in Q

A is an accumulator along with the left side of Q

As we shift C/A/Q, we begin to write over part of Q (but it's a part that we've already used in the multiplication)

For each bit in Q, if 0 then merely shift C/A/Q, otherwise add M to C/A

Notice that A/Q stores the resulting product, not just A

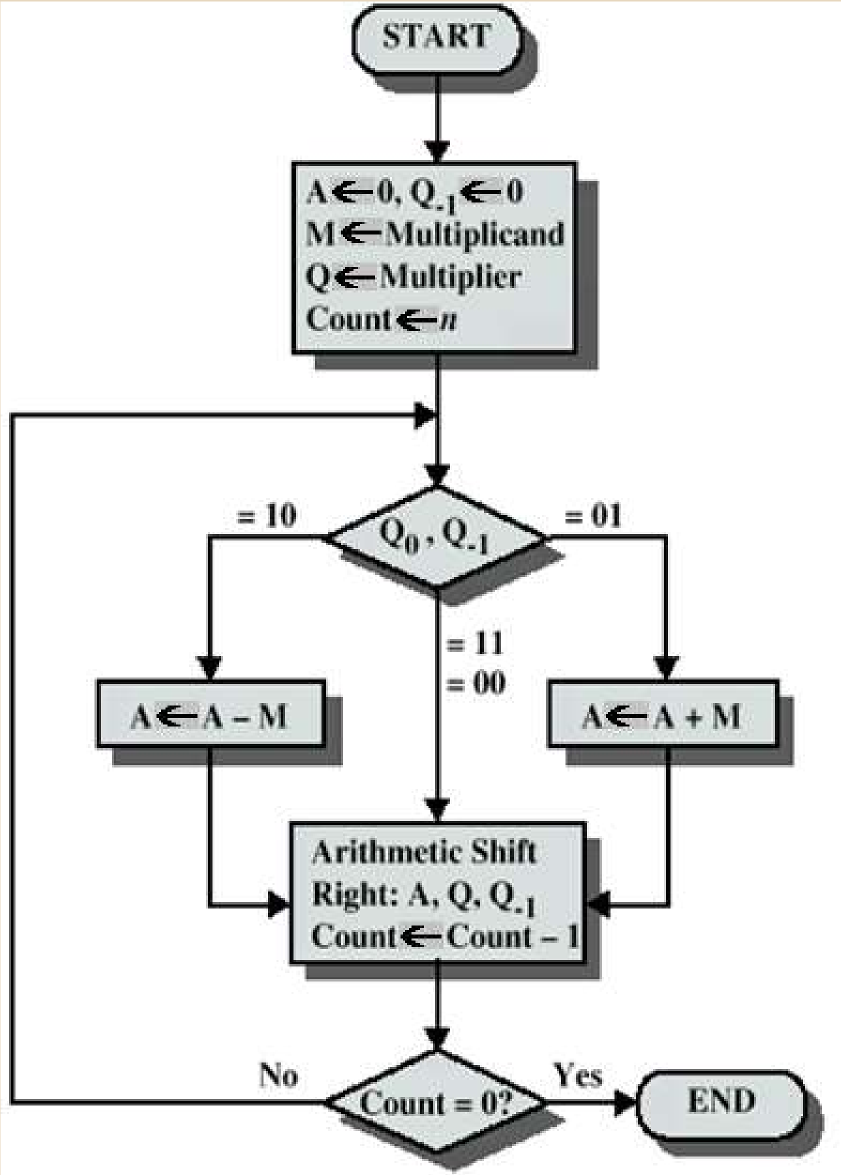
Booth's Algorithm

We will use Booth's algorithm if either or both numbers are negative

The idea is based on this observation:

0011110 =
0100000 –
0000010

So, in Booth's, we look for transitions of 01 and 10, and ignore 00 and 11 sequences in our multiplier



Compare rightmost bit of Q (that is, Q_0) with the previous rightmost bit from Q (which is stored in a single bit Q_{-1})

Q_{-1} is initialized to 0

If this sequence is 0 – 1 then add M to A

If this sequence is 1 – 0 then sub M from A

If this sequence is 0 – 0 or 1 – 1 then don't add

After each iteration, shift A >> Q >> Q_{-1}

Example of Using Booth

A	Q	Q_{-1}	M		
0000	0011	0	0111	Initial Values	
1001	0011	0	0111	$A \leftarrow A - M$	First Cycle
1100	1001	1	0111	Shift	
1110	0100	1	0111	Shift	Second Cycle
0101	0100	1	0111	$A \leftarrow A + M$	Third Cycle
0010	1010	0	0111	Shift	
0001	0101	0	0111	Shift	Fourth Cycle

Initialize A to 0
Initialize Q to 0011
Initialize M to 0111
Initialize Q_{-1} to 0

1) Q/Q_{-1}=10, A←A–M, Shift
2) Q/Q_{-1}=11, Shift
3) Q/Q_{-1}=01,A←A+M, Shift
4) Q/Q_{-1}=00, Shift

Done, Answer = 00010101

Division

- Just as multiplication is a series of additions and shifts, division is a series of shifts and subtractions
  - The basic idea is this:
    - how many times can we subtract the denominator from the numerator?

Consider 110011 / 000111

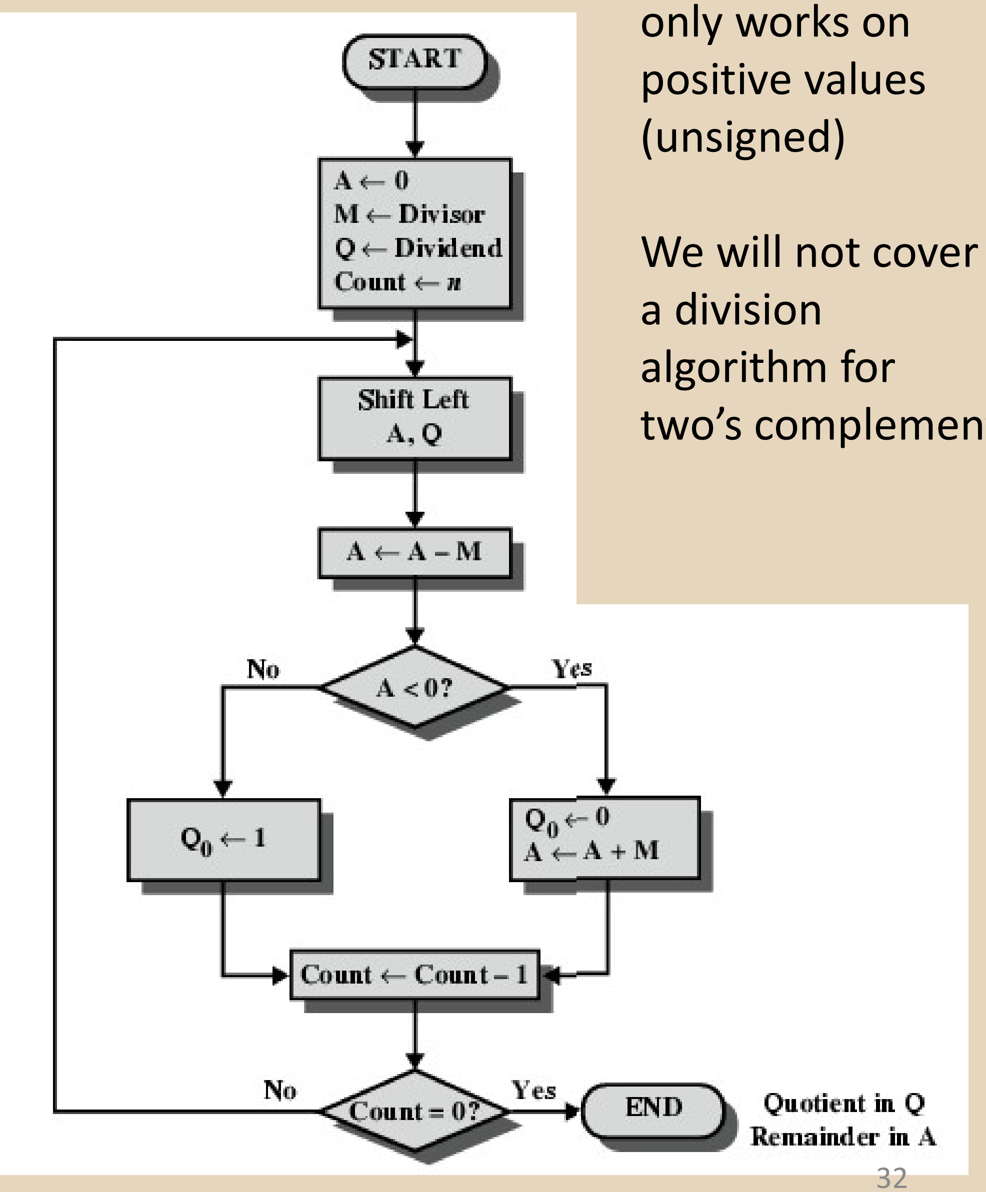
We cannot subtract 000111 from 000001
We cannot subtract 000111 from 000011
We cannot subtract 000111 from 000110
We can subtract 000111 from 001100
leaving 000101
We can subtract 000111 from 001010
leaving 000101
We can subtract 000111 from 001010
leaving 000101
Giving the answer 000111 with a remainder of 000101

Our divisor is 0, shift 000001
Our divisor is 00, shift 00011
Our divisor is 000, shift 000110
Now, our divisor is 0001,
shift 000101
Now our divisor is 00011,
shift 000101
Our divisor is now 000111

We are done after 6 iterations (6 bits)

Division Algorithm

- Dividend is expressed using 2*n bits and loaded into the combined A/Q registers
  - upper half in A, lower half in Q
- Notice that we subtract M from A and then determine if the result is negative – if so, we restore A,
- An easier approach is:
  - Remove A ← A – M
  - Replace A < 0? With A< M?
  - If No, then A ← A – M, Q_n ← 1
  - If Yes, then Q_n ← 0
  - Now we don't need to worry about restoring A
- At the conclusion of the operation
  - the quotient is in Q
  - and any remainder is in A



This algorithm only works on positive values (unsigned)

We will not cover a division algorithm for two's complement

Division Example: 7 / 3

A	Q	M	
0000	0111	0011	Initial Values
0000	1110		Shift A/Q left 1 bit
			Since A < M, insert 0 into Q_0
0001	1100		Shift A/Q left 1 bit
			Since A < M, insert 0 into Q_0
0011	1000		Shift A/Q left 1 bit
0000	1001		Since A >= M, A $\leftarrow$ A-M, insert 1 into Q_0
0001	0010		Shift A/Q left 1 bit
			Since A < M, insert 0 into Q_0

Done (4 shifts)

Result: Q = 0010, A = 0001

A = remainder (1) and Q = quotient (2) or 7 / 3 = 2 1 / 3

Bias Representation

- We can use unsigned magnitude to represent both positive and negative numbers by using a bias, or excess, representation
  - The entire numbering system is shifted up some positive amount
    - To get a value, subtract it from the excess
      - For instance, in excess-16, we subtract 16 from the number to get the real value (11001 in excess-16 is 11001 – 10000 in binary = 01001 = +9)
  - To use the representation
    - numbers have to be shifted, then stored, and then shifted back when retrieved
      - this seems like a disadvantage, so we won't use it to represent ordinary integer values
      - but we will use it to represent exponents in our floating point representation (shown next)

Excess-8 Notation

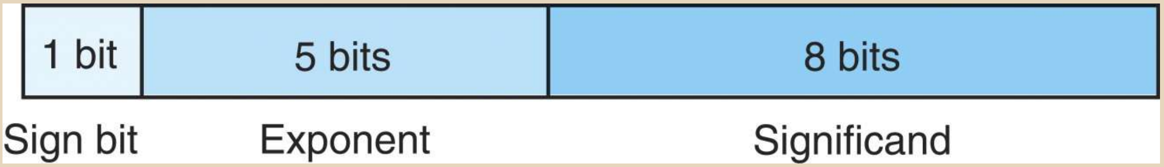
0000	-8
0001	-7
0010	-6
0011	-5
0100	-4
0101	-3
0110	-2
0111	-1
1000	0
1001	1
1010	2
1011	3
1100	4
1101	5
1110	6
1111	7

Floating Point Representation

- Floating point numbers have a *floating* decimal point
  - Recall the fraction notation used a fixed decimal point
  - Floating point is based on scientific notation
    - $3518.76 = .351876 * 10^4$
    - We represent the floating point number using 2 integer values called the significand and the exponent, along with a sign bit
      - The integers are 351876 and 4 for our example above
  - For a binary version of floating point, we use base 2 instead of 10 as the radix for our exponent
  - We store the 2 integer values plus the sign bit all in binary
    - We *normalize* the floating point number so that the decimal is implied to be before the first 1 bit, and in shifting the decimal point, we determine the exponent
    - The exponent is stored in a bias representation to permit both positive and negative exponents
    - The significand is stored in unsigned magnitude

Examples

- Here, we use the following 14-bit representation:



Exponents will be stored using excess-16

0 | 10101 | 10001000

Sign bit = 0 (positive)
Exponent = 5 (10101 – 10000 = 5)
Significand = .10001000
We shift the decimal point 5 positions giving us 10001.0 = +17

0 | 01110 | 10000000

Sign bit = 0 (positive)
Exponent = -2 (01110 – 10000 = -2)
Significand = .10000000
We shift the decimal point 2 positions to the left, giving us 0.001 = +.125

1 | 10011 | 11010100

Sign bit = 1 (negative)
Exponent = 3 (10011 – 10000 = 3)
Significand = .11010100
We shift the decimal point 3 positions to the right, giving 110.101= -6.625

Floating Point Formats and Problems

- To provide a standard for all architectures, IEEE provides the following formats:
  - Single precision
    - 32-bits: 1-bit sign, 8-bit exponent using excess-127, 23-bit significand
  - Double precision
    - 64-bits: 1-bit sign, 11-bit exponent using excess-1023, 52-bit significand
  - IEEE also provides NAN for errors when a value is not a real number
    - NAN = *not a number*
- Problems
  - there are numerous ways to represent the same number, but because we normalize the numbers, there will ultimately be a single representation for the number
  - Errors arise from
    - overflow (too great a positive number or too great a negative number) – overflowing the signficand
    - underflow (too small a fraction) – overflowing the exponent

Other Binary Codes

- We use a code to represent characters
  - EBCDIC – developed for the IBM 360 and used in all IBM mainframes since then
    - An 8-bit representation for 256 characters
  - ASCII – used in just about every other computer
    - A 7-bit representation plus the high-order bit used for parity
  - Unicode – newer representation to include non-Latin based alphabetic characters
    - 16 bits allow for 65000+ characters
    - It is downward compatible with ASCII, so the first 128 characters are the same as ASCII

ASCII Table

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	`
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(	72	48	110	H	104	68	150	h
9	9	11		41	29	51	)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	;	91	5B	133	[	123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	|
29	1D	35		61	3D	75	=	93	5D	135	]	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37		63	3F	77	?	95	5F	137	_	127	7F	177	

ASCII	EBCDIC	Graphic	ASCII	EBCDIC	Graphic	ASCII	EBCDIC	Graphic	ASCII	EBCDIC	Graphic	ASCII	EBCDIC	Graphic	ASCII	EBCDIC	Graphic	ASCII	EBCDIC	Graphic	ASCII	EBCDIC	Graphic
00	00	NUL	20	40	SP	40	7C	@	60	79	`	80	00	NUL	A0	40	SP	C0	7C	@	E0	79	`
01	01	SOH	21	5A	!	41	C1	A	61	81	a	81	01	SOH	A1	5A	!	C1	C1	A	E1	81	a
02	02	STX	22	7F	"	42	C2	B	62	82	b	82	02	STX	A2	7F	"	C2	C2	B	E2	82	b
03	03	ETX	23	7B	#	43	C3	C	63	83	c	83	03	ETX	A3	7B	#	C3	C3	C	E3	83	c
04	37	EOT	24	5B	$	44	C4	D	64	84	d	84	37	EOT	A4	5B	$	C4	C4	D	E4	84	d
05	2D	ENQ	25	6C	%	45	C5	E	65	85	e	85	2D	ENQ	A5	6C	%	C5	C5	E	E5	85	e
06	2E	ACK	26	50	&	46	C6	F	66	86	f	86	2E	ACK	A6	50	&	C6	C6	F	E6	86	f
07	2F	BEL	27	7D	'	47	C7	G	67	87	g	87	2F	BEL	A7	7D	'	C7	C7	G	E7	87	g
08	16	BS	28	4D	(	48	C8	H	68	88	h	88	16	BS	A8	4D	(	C8	C8	H	E8	88	h
09	05	HT	29	5D	)	49	C9	I	69	89	i	89	05	HT	A9	5D	)	C9	C9	I	E9	89	i
0A	25	LF	2A	5C	*	4A	D1	J	6A	91	j	8A	25	LF	AA	5C	*	CA	D1	J	EA	91	j
0B	0B	VT	2B	4E	+	4B	D2	K	6B	92	k	8B	0B	VT	AB	4E	+	CB	D2	K	EB	92	k
0C	0C	FF	2C	6B	,	4C	D3	L	6C	93	l	8C	0C	FF	AC	6B	,	CC	D3	L	EC	93	l
0D	0D	CR	2D	60	-	4D	D4	M	6D	94	m	8D	0D	CR	AD	60	-	CD	D4	M	ED	94	m
0E	0E	SO	2E	4B	.	4E	D5	N	6E	95	n	8E	0E	SO	AE	4B	.	CE	D5	N	EE	95	n
0F	0F	SI	2F	61	/	4F	D6	O	6F	96	o	8F	0F	SI	AF	61	/	CF	D6	O	EF	96	o
10	10	DLE	30	F0	0	50	D7	P	70	97	p	90	10	DLE	B0	F0	0	D0	D7	P	F0	97	p
11	11	DC1	31	F1	1	51	D8	Q	71	98	q	91	11	DC1	B1	F1	1	D1	D8	Q	F1	98	q
12	12	DC2	32	F2	2	52	D9	R	72	99	r	92	12	DC2	B2	F2	2	D2	D9	R	F2	99	r
13	13	DC3	33	F3	3	53	E2	S	73	A2	s	93	13	DC3	B3	F3	3	D3	E2	S	F3	A2	s
14	3C	DC4	34	F4	4	54	E3	T	74	A3	t	94	3C	DC4	B4	F4	4	D4	E3	T	F4	A3	t
15	3D	NAK	35	F5	5	55	E4	U	75	A4	u	95	3D	NAK	B5	F5	5	D5	E4	U	F5	A4	u
16	32	SYN	36	F6	6	56	E5	V	76	A5	v	96	32	SYN	B6	F6	6	D6	E5	V	F6	A5	v
17	26	ETB	37	F7	7	57	E6	W	77	A6	w	97	26	ETB	B7	F7	7	D7	E6	W	F7	A6	w
18	18	CAN	38	F8	8	58	E7	X	78	A7	x	98	18	CAN	B8	F8	8	D8	E7	X	F8	A7	x
19	19	EM	39	F9	9	59	E8	Y	79	A8	y	99	19	EM	B9	F9	9	D9	E8	Y	F9	A8	y
1A	3F	SUB	3A	7A	:	5A	E9	Z	7A	A9	z	9A	3F	SUB	BA	7A	:	DA	E9	Z	FA	A9	z
1B	27	ESC	3B	5E	;	5B	AD	[	7B	C0	{	9B	27	ESC	BB	5E	;	DB	AD	[	FB	C0	{
1C	1C	FS	3C	4C	<	5C	E0	\	7C	4F	|	9C	1C	FS	BC	4C	<	DC	E0	\	FC	4F	|
1D	1D	GS	3D	7E	=	5D	BD	]	7D	D0	}	9D	1D	GS	BD	7E	=	DD	BD	]	FD	D0	}
1E	1E	RS	3E	6E	>	5E	5F	^	7E	A1	~	9E	1E	RS	BE	6E	>	DE	5F	^	FE	A1	~
1F	1F	US	3F	6F	?	5F	6D	_	7F	07	DEL	9F	1F	US	BF	6F	?	DF	6D	_	FF	07	DEL

Error Detection

- Errors will still arise, so we should also provide error detection and correction mechanisms
  - One method is to add a checksum to each block of data
    - Bits are appended to every block that somehow encode the information
    - One common form is the CRC
      - Cyclic redundancy check
- Simpler approach is to use a parity bit to every byte of information
  - Add up the number of 1 bits in the byte, add a bit so that the number of total 1s is even
    - 00101011 has a parity bit of 0, 11100011 has a parity bit of 1
  - With more parity bits, we can not only detect an error, but correct it, or detect 2 errors
- Hamming Codes are a common way to provide high redundancy on error checking

Error-Detecting and Error-Correcting Codes

- Motivation
  - Computers make errors occasionally (data gets corrupted) due to
    - Voltage spikes
    - Cosmic particles
  - Corrupt data causes incorrect behavior
- Fix
  - Use some bits to hold redundant information
  - Data + Redundancy → Code Words
  - Depending of amount of redundancy (and exact properties of the codes) we can
    - Detect errors
    - Correct errors (automatically)