The need for Data Representation

• Digital systems are built from circuits that process binary digits- 0s and 1s

• But very few real-life problems are based on binary numbers or any numbers at all.

➔ **Correspondence between the binary digits processed by digital circuits and real-life numbers and conditions must be established.**

Representation of Characters

• American Standard Code for Information Interchange (acronym: ASCII; pronounced /æski/, ASS-kee) is a character-encoding scheme based on the ordering of the English.

- •Uses 7 bits to encode each character (2^7 = 128 values)
- •Extended ASCII uses 8 bits

•**Unicode** is a computing industry standard allowing computers to consistently represent and manipulate text expressed in most of the world's writing systems (including Asian languages)

Uses 2 bytes to encode each character (2^16 = 65,536 values)
Extended ASCII uses 8 bits

Preliminaries

1. A number system consists of an order set of symbols (digits) with relations defined for +,-,*, /
2. The radix (or base) of the number system is the total number of digits allowed in the the number system.
   - Example, in decimal numbers, radix = 10,

     digits allowed = 0,1, 2,…,9

3. Common base include:
   - Binary: 0, 1
   - Octal: 0,1,..,7
   - Hexadecimal: 0,1,…,9,A,B,C, D,E,F

Outline

Number Systems

- Positional Notations
- Number Systems & Base-R to Decimal Conversion
- Conversion between bases
- Binary Arithmetic Operations
- How to represent Negative Numbers
- Comparison of Sign-and-Magnitude and Complements

Positional Notations

- Weighted-positional notation
  - Decimal number system, symbols = { 0, 1, 2, 3, …, 9 }
  - Position is important
  - Example:$(7594)_{10} = (7x10^3) + (5x10^2) + (9x10^1) + (4x10^0)$
  - The value of each symbol is dependent on its type and its position in the number
  - In general, $(a_n a_{n-1} \ldots a_0)_{10} = (a_n \times 10^n) + (a_{n-1} \times 10^{n-1}) + \ldots + (a_0 \times 10^0)$

Positional Notations

- Fractions are written in decimal numbers after the *decimal point*.
  - $(2.75)_{10} = (2 \times 10^{0}) + (7 \times 10^{-1}) + (5 \times 10^{-2})$
  - In general, $(a_n a_{n-1} \ldots a_0 \,.\, f_1 f_2 \ldots f_m)_{10} = (a_n \times 10^{n}) + (a_{n-1} \times 10^{n-1}) + \ldots + (a_0 \times 10^{0}) + (f_1 \times 10^{-1}) + (f_2 \times 10^{-2}) + \ldots + (f_m \times 10^{-m})$

Decimal (base 10) Number System

- Weighting factors (or weights) are in powers-of-10:

  $\ldots\ 10^{3}\ 10^{2}\ 10^{1}\ 10^{0}.10^{-1}\ 10^{-2}\ 10^{-3}\ 10^{-4}\ \ldots$

- To evaluate the decimal number 593.68, the digit in each position is multiplied by the corresponding weight:

  $5\times10^{2} + 9\times10^{1} + 3\times10^{0} + 6\times10^{-1} + 8\times10^{-2}$

  $= (593.68)_{10}$

Base-R to Decimal Conversion

- Binary (base 2): weights in powers-of-2.
  - Binary digits (bits): ***0,1.***
- Octal (base 8): weights in powers -of-8.
  - Octal digits: ***0,1,2,3,4,5,6,7.***
- Hexadecimal (base 16): weights in powers-of-16.
  - Hexadecimal digits:
    0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F.
- Base *R*: weights in powers-of-R.

Base-R to Decimal Conversion

- $(1101.101)_2 = 1\times2^3 + 1\times2^2 + 1\times2^0 + 1\times2^{-1} + 1\times2^{-3}$
  $= 8 + 4 + 1 + 0.5 + 0.125 = (13.625)_{10}$

- $(572.6)_8 = 5\times8^2 + 7\times8^1 + 2\times8^0 + 6\times8^{-1}$
  $= 320 + 56 + 16 + 0.75 = (392.75)_{10}$

- $(2A.8)_{16} = 2\times16^1 + 10\times16^0 + 8\times16^{-1}$
  $= 32 + 10 + 0.5 = (42.5)_{10}$

- $(341.24)_5 = 3\times5^2 + 4\times5^1 + 1\times5^0 + 2\times5^{-1} + 4\times5^{-2}$
  $= 75 + 20 + 1 + 0.4 + 0.16 = (96.56)_{10}$

Decimal-to-Base R Conversion

Algorithm 1

Step 1: Break the number in two parts: Whole number and fraction part.

Step 2: Repeated Division-by-R Method (for whole numbers)

Step 3: Repeated Multiplication-by-R Method (for fractions)

Algorithm 2: Sum-of-Weights Method

Repeated Division-by-R Method

- To convert a whole number to base R, use successive division by R until the quotient is 0. The remainders form the answer, with the first remainder as the *least significant bit (LSB)* and the last as the *most significant bit (MSB)*.

$(43)_{10} = (101011)_2$

2	43		
2	21	rem 1	LSB
2	10	rem 1	
2	5	rem 0	
2	2	rem 1	
2	1	rem 0	
	0	rem 1	MSB

Repeated Multiplication-by-R Method

- To convert decimal fractions to binary, repeated multiplication by R is used, until the fractional product is 0 (or until the desired number of decimal places). The carried digits, or *carries*, produce the answer, with the first carry as the MSB, and the last as the LSB.

E.g. $(0.3125)_{10} = (.0101)_2$

	Carry	
$0.3125 \times 2 = 0.625$	0	MSB
$0.625 \times 2 = 1.25$	1	
$0.25 \times 2 = 0.50$	0	
$0.5 \times 2 = 1.00$	1	LSB

Sum-of-Weights Method

- Determine the set of binary weights whose sum is equal to the decimal number.

$(9)_{10} = 8 + 1 = 2^3 + 2^0 = (1001)_2$

$(18)_{10} = 16 + 2 = 2^4 + 2^1 = (10010)_2$

$(58)_{10} = 32 + 16 + 8 + 2 = 2^5 + 2^4 + 2^3 + 2^1 = (111010)_2$

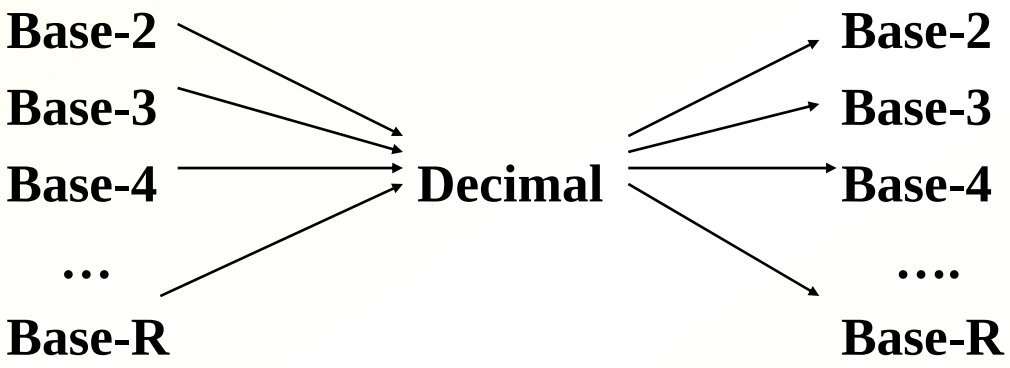
$(0.625)_{10} = 0.5 + 0.125 = 2^{-1} + 2^{-3} = (0.101)_2$

Summary: Conversion between Bases

- Base-R to decimal: multiply digits with their corresponding weights
- Decimal to binary (base 2)
  - whole numbers: repeated division-by-2
  - fractions: repeated multiplication-by-2
- Decimal to base-R
  - whole numbers: repeated division-by-R
  - fractions: repeated multiplication-by-R

Conversion between Bases

- In general, conversion between bases can be done via decimal:



Shortcuts are available for conversion between bases 2, 4, 8, 16.

Binary-Octal/Hexadecimal Conversion

- Binary $\rightarrow$ Octal: Partition in groups of 3

  $(10\ 111\ 011\ 001\ .\ 101\ 110)_2 = (2731.56)_8$

- Octal $\rightarrow$ Binary: reverse

  $(2731.56)_8 = (10\ 111\ 011\ 001\ .\ 101\ 110)_2$

- Binary $\rightarrow$ Hexadecimal: Partition in groups of 4

  $(101\ 1101\ 1001\ .\ 1011\ 1000)_2 = (5D9.B8)_{16} = 0x5D9.B8$

- Hexadecimal $\rightarrow$ Binary: reverse

  $(5D9.B8)_{16} = (101\ 1101\ 1001\ .\ 1011\ 1000)_2$

Binary Arithmetic operations

Addition

0+0 = 0

0+1 = 1

1+0 = 1

1+1 = 10 (carry)

Subtraction

0-0 = 0

0-1 = 1 (after borrowing)

1-0 = 1

1-1 = 0

Multiplication

0*0 = 0

0*1 = 0

1*0 = 0

1*1 = 1

Binary Arithmetic Operations (I)

ADDITION

- Like decimal numbers, two numbers can be added by adding each pair of digits together with carry propagation.
  - Start at the right-most bit
  - Carry is carried to one position left.

$$\begin{array}{r} (11011)_2 \\ +\ (10011)_2 \\ \hline (101110)_2 \\ \hline \end{array} \qquad \begin{array}{r} (647)_{10} \\ +\ (537)_{10} \\ \hline (1184)_{10} \\ \hline \end{array}$$

Binary Arithmetic Operations (II)

$$
\begin{array}{r}
(11011)_2 \\
+\ (10011)_2 \\
\hline
(101110)_2 \\
\hline
\end{array}
$$

Carries	1	0	0	1	1	0
	0	1	1	0	1	1
	0	1	0	0	1	1
	1	0	1	1	1	0

Binary Arithmetic Operations (III)

SUBTRACTION

- Two numbers can be subtracted by subtracting each pair of digits together with borrowing, where needed.

$(11001)_2$	$(627)_{10}$
$-\ (10011)_2$	$-\ (537)_{10}$
$(00110)_2$	$(090)_{10}$

Binary Arithmetic Operations (IV)

$$
\begin{array}{r}
(11001)_2 \\
-\ (10011)_2 \\
\hline
(00110)_2 \\
\hline
\end{array}
$$

Borrows	0	0	1	1	0	0
	0	1	1	0	0	1
	0	1	0	0	1	1
	0	0	0	1	1	0

Binary Arithmetic Operations (V)

- MULTIPLICATION
- To multiply two numbers, take each digit of the multiplier and multiply it with the multiplicand. This produces a number of partial products which are then added.

$(11001)_2$	$(214)_{10}$	Multiplicand
x $(10101)_2$	x $(152)_{10}$	Multiplier
$(11001)_2$	$(428)_{10}$	
$(11001)_2$	$(1070)_{10}$	Partial
$+(11001)_2$	$+(214)_{10}$	products
$(1000001101)_2$	$(32528)_{10}$	Result

Negative numbers representation

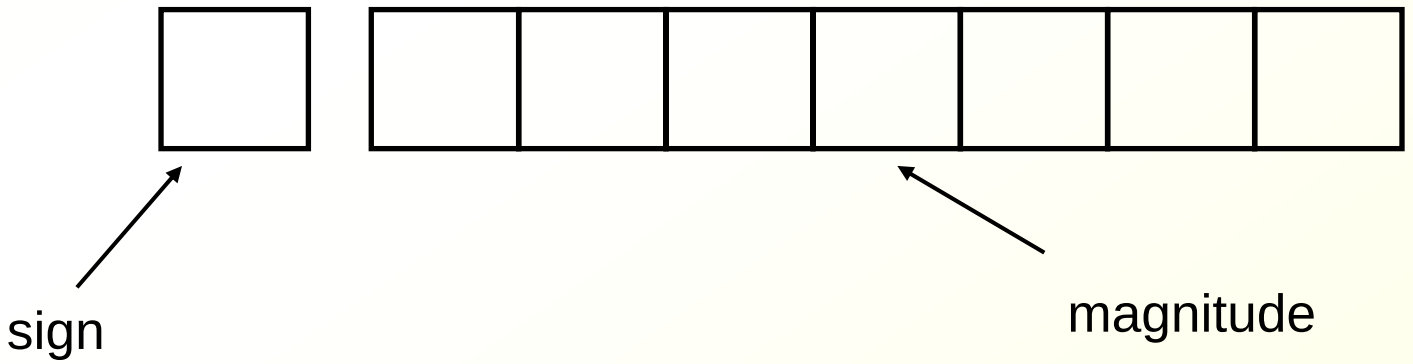
Till now, we have only considered how unsigned numbers can be represented. There are four common ways of representing signed numbers:

- Sign-and-Magnitude
- Diminished radix (*R*-1's) complement (e.g. 1s complement)
- Radix (*R*'s) complement (e.g. 2s complement)
- Excess *k*

With signed magnitude representation, one has to look at both the signs and magnitude of operands in an arithmetic operation. Hence, different procedures (circuits) would have to be used for each case. Complement representation takes care of this problem.

Negative Numbers: Sign-and-Magnitude (I)

- Negative numbers are usually written by pre-pending a minus sign in front.
  - Example:
    $-(12)_{10}$, $-(1100)_2$
- In computer memory of fixed width, this sign is usually represented by a bit:
  0 for + and 1 for -
- The left-most bit is the sign bit and the remaining bits hold the absolute magnitude. memory of fixed
- Example: an 8-bit number can have 1-bit sign and 7-bits magnitude.



Negative Numbers: Sign-and-Magnitude (II)

- Largest Positive Number: 0 1111111 $+(127)_{10}$
- Largest Negative Number: 1 1111111 $-(127)_{10}$
- Zeroes: 0 0000000 $+(0)_{10}$
  1 0000000 $-(0)_{10}$
- Range: $-(127)_{10}$ to $+(127)_{10}$
- To negate a number, just invert the sign bit.
- Examples:

$-(0\ 0100001)_{sm} = (1\ 0100001)_{sm}$

$-(1\ 0000101)_{sm} = (0\ 0000101)_{sm}$

Complement representation

- In these representation, a positive number is represented as it is (like an unsigned positive number)
- A negative number, however, is represented by taking the complement of unsigned number

Complements

- Complement numbers can help perform subtraction.
  With complements, subtraction can be performed by addition.
- In general for Base-*r* number, there are:
  (i) Diminished Radix (or *r*-1's) Complement
  (ii) Radix (or *r*'s) Complement
- For Base-2 number, we have:
  (i) 1s Complement
  (ii) 2s Complement

1s Complement (I)

- 1s complement of an unsigned number is obtained by inverting all the bits of the number

  Examples: 1s complement of $(00000001)_2$ is $(11111110)_{1s}$

  1s complement of $(01111111)_2 = (10000000)_{1s}$

- 1s representation of the number in 8-bits

Example:

1. $(+14)_{10} = (00001110)_2 = (00001110)_{1s}$
2. $(-14)_{10} = -(00001110)_2 = (11110001)_{1s}$
3. $(-80)_{10} = (?)_2 = (?)_{1s}$

1s Complement (II)

For 8-bits number system:

- Largest Positive Number: 0 1111111 $+(127)_{10}$
- Largest Negative Number: 1 0000000 $-(127)_{10}$
- Zeroes: 0 0000000
  1 1111111
- Range: $-(127)_{10}$ to $+(127)_{10}$
- The most significant bit still represents the sign:
  0 = +ve; 1 = -ve.

1s Complement (III)

- Given a number x which can be expressed as an n-bit binary number, its negative value can be obtained in 1s-complement representation using:

$$-x = 2^n - x - 1$$

Example: With an 8-bit number 00001100, its negative value, expressed in 1s complement, is obtained as follows:

$$\begin{aligned} -(00001100)_2 &= -(12)_{10} \\ &= (2^8 - 12 - 1)_{10} \\ &= (243)_{10} \\ &= (11110011)_{1s} \end{aligned}$$

2s Complement (I)

- 2s complement of an unsigned number is obtained by inverting all the bits and adding 1.

  Examples:

  1. 2s complement of $(00000001)_2$ = $(11111110)_{1s}$ (invert)

     = $(11111111)_{2s}$ (add 1)

  2. 2s complement of $(01111110)_2$ = $(10000001)_{1s}$ (invert)

     = $(10000010)_{2s}$ (add 1)

2s Complement (I)

- 2s complement representation for 8 bit numbers:

Example:

1. $(+14)_{10} = (00001110)_2 = (00001110)_{2s}$
2. $(-14)_{10} = -(00001110)_2 = (11110010)_{2s}$
3. $(-80)_{10} = (?)_2 = (?)_{2s}$

2s Complement (II)

- Given a number x which can be expressed as an n-bit binary number, its negative number can be obtained in 2s-complement representation using:

$$-x = 2^n - x$$

Example: With an 8-bit number 00001100, its negative value in 2s complement is thus:

$$-(00001100)_2 = -(12)_{10}$$
$$= (2^8 - 12)_{10}$$
$$= (244)_{10}$$
$$= (11110100)_{2s}$$

2s Complement (III)

- Largest Positive Number: 0 1111111 $+(127)_{10}$
- Largest Negative Number: 1 0000000 $-(128)_{10}$
- Zero: 0 0000000
- Range: $-(128)_{10}$ to $+(127)_{10}$
- The most significant bit still represents the sign:

  0 = +ve; 1 = -ve.

Comparisons of Sign-and-Magnitude & Complements (I)

Example: 4-bit signed number (positive values)

Value	Sign-and-Magnitude	1s Comp.	2s Comp.
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
+0	0000	0000	0000

Comparisons of Sign-and-Magnitude and Complements (II)

Example: 4-bit signed number (negative values)

Value	Sign-and-Magnitude	1s Comp.	2s Comp.
-0	1000	1111	-
-1	1001	1110	1111
-2	1010	1101	1110
-3	1011	1100	1101
-4	1100	1011	1100
-5	1101	1010	1011
-6	1110	1001	1010
-7	1111	1000	1001
-8	-	-	1000

MSB = 1 indicates a negative number in either notation

Use of complements

• Complement number system is used to minimize the amount of circuitry needed to perform integer arithmetic.

• For example, A-B can be performed by computing A + (-B), where (-B) is represented in 2s complement of B.

• Hence, the computer needs only binary adder and complementing circuit to handle both addition and subtraction

Overflow (I)

- Signed binary numbers are of a fixed range.
- If the result of addition/subtraction goes beyond this range, overflow occurs.
- Two conditions under which overflow can occur are:

  (i) *positive add positive* gives negative

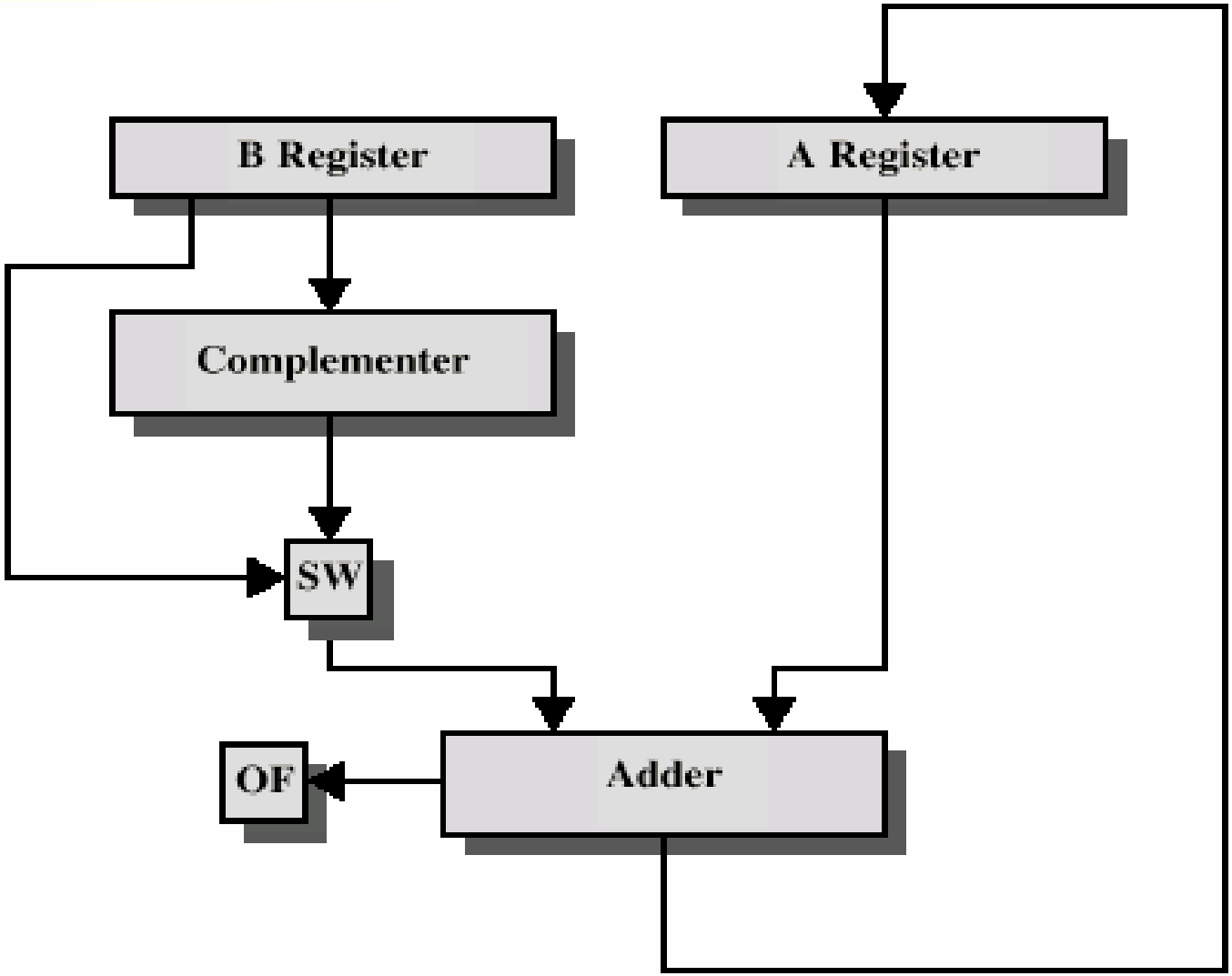
  (ii) *negative add negative* gives positive

Addition and Subtraction

- Normal binary addition
- Monitor sign bit for overflow

- Take twos compliment of substahend and add to minuend
  - i.e. a - b = a + (-b)
- So we only need addition and complement circuits

Hardware for Addition and Subtraction



OF = overflow bit
SW = Switch (select addition or subtraction)

2s complement addition

Algorithm:

1. Perform binary addition on the two numbers.
2. Ignore the carry out of the MSB.
3. Check for overflow: Overflow occurs if the carrier into and out of the MSB are different.

2s complement subtraction

Algorithm for performing A - B:

A-B = A + (-B)

1. Take 2s complement of B by inverting all the bits and adding 1
2. Add the 2s complement of B to A

Examples: 2s addition/Subtraction

4-bits system

+3	0011
+ +4	+ 0100
----	-------
+7	0111
----	-------

-2	1110
+ -6	+ 1010
----	-------
-8	**1**1000
----	-------

+6	0110
+ -3	+ 1101
----	-------
+3	**1**0011
----	-------

+4	0100
+ -7	+1001
----	-------
-3	1101
----	-------

Examples: Overflow in 2s addition/Subtraction

4-bits system

-3	1101
+ -6	+ 1010
---	-------
-9	10111
---	-------

+7

+5	0101
+ +6	+ 0110
---	-------
+11	1011
---	-------

-5

1s complement Addition/Subtraction rules

Algorithm C=A+B:

1. Perform binary addition on the two numbers
2. If there is a carry out of the MSB, add 1 to the result (to get C)
3. Check for overflow: if carried into and out of of MSB are different and C is opposite sign of A and B

Algorithm A-B

1. Complement all bits of B
2. Proceed as addition

Examples: 1s addition/subtraction

```
   +3      0011
+  +4   +  0100
  ----    -------
   +7      0111
  ----    -------
```

```
   +5      0101
+  -5   +  1010
  ----    -------
   -0      1111
  ----    -------
```

```
   -2      1101
+  -5   +  1010
  ----    -------
   -7     10111
  ----       +   1
            -------
             1000
```

```
   -3      1100
+  -7   +  1000
  ----    -------
  -10     10100
  ----       +   1
            -------
             0101
```

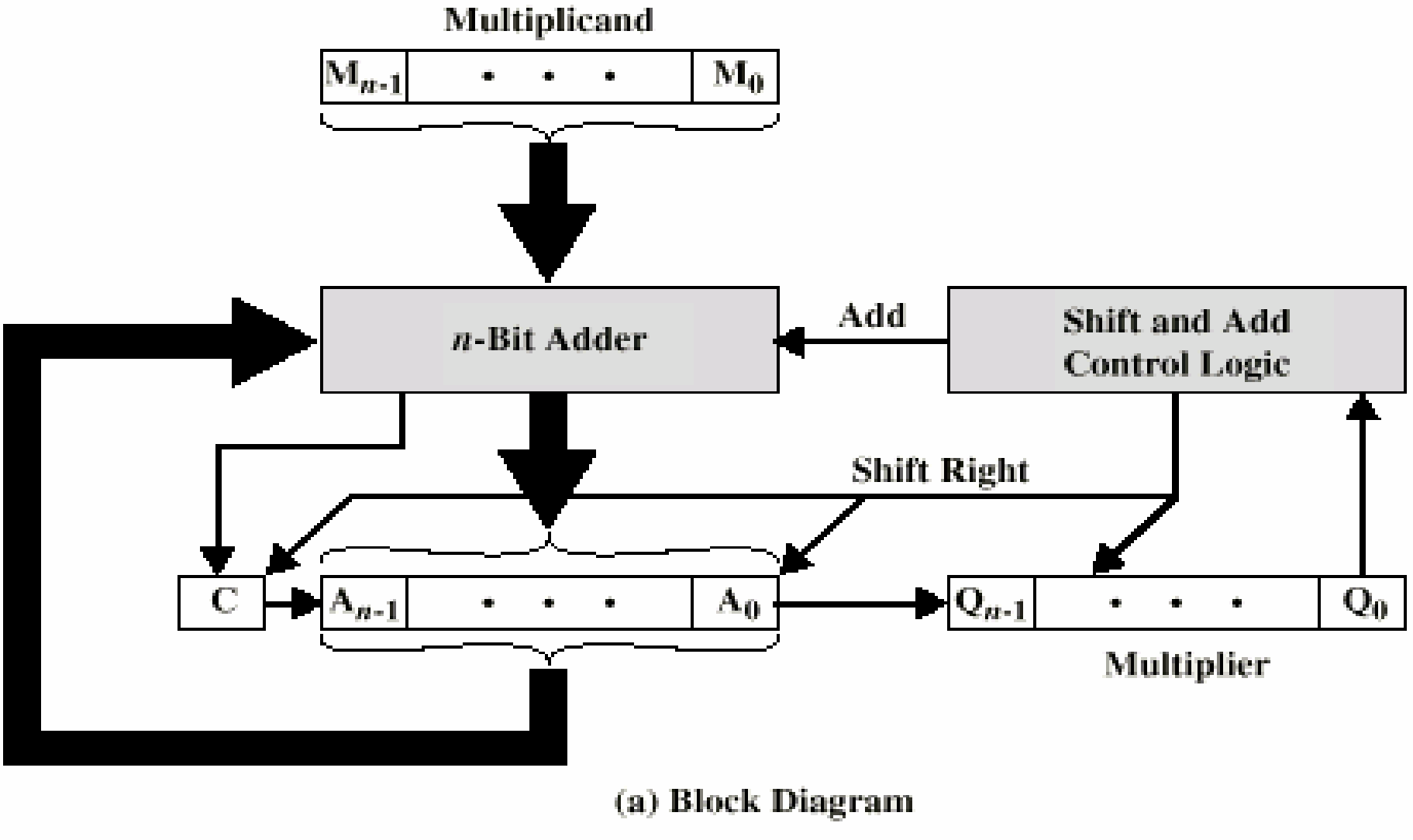
Multiplication

- Complex
- Work out partial product for each digit
- Take care with place value (column)
- Add partial products

Multiplication Example

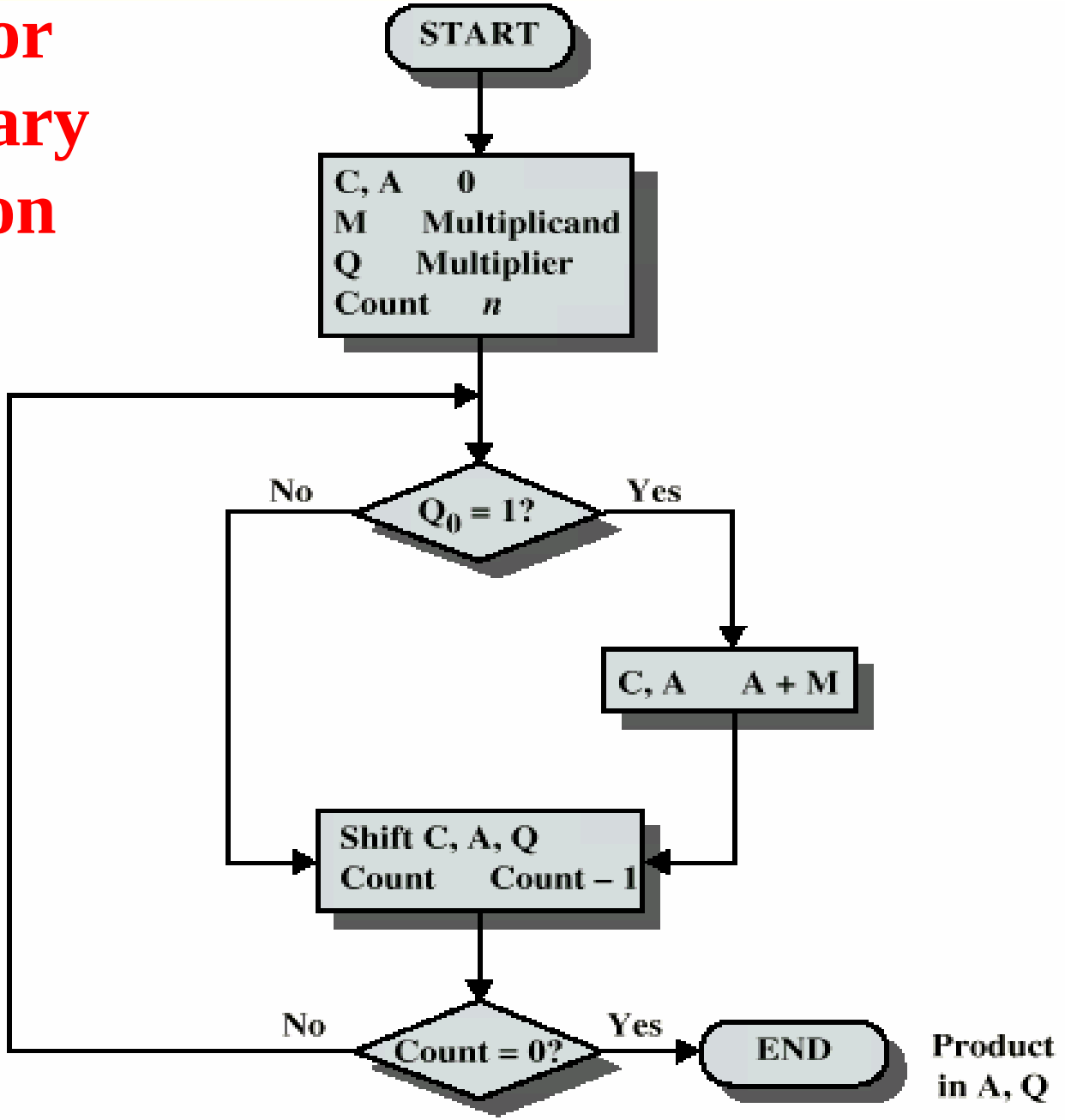
- 1011 Multiplicand (11 decimal)
- x 1101 Multiplier (13 decimal)
- 1011 Partial products
- 0000 Note: if multiplier bit is 1 copy
- 1011 multiplicand (place value)
- 1011 otherwise zero
- 10001111 Product (143 dec)
- Note: need double length result

Unsigned Binary Multiplication



(a) Block Diagram

Flowchart for unsigned binary multiplication



Execution of 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	

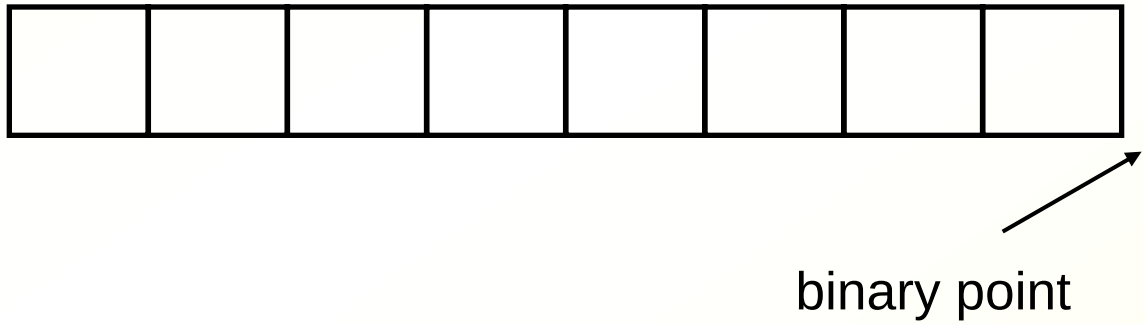
Multiplying Negative Numbers

- This does not work!
- Solution 1
  - Convert to positive if required
  - Multiply as above
  - If signs were different, negate answer
- Solution 2
  - Booth's algorithm (not covered: Refer to William Stallings' book "Computer Organization and Architecture"

Fixed Point Numbers (I)

- The signed and unsigned numbers representation given are fixed point numbers.

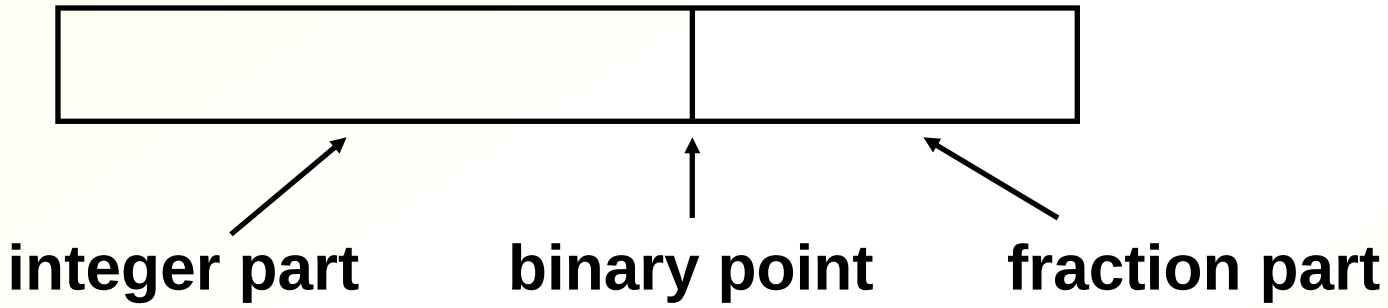
- The binary point is assumed to be at a fixed location, say, at the end of the number:



can represent all integers between -128 to 127 (for 8 bits).

Fixed Point Numbers (II)

- In general, other locations for binary points possible.



Examples: If two fractional bits are used, we can represent:

$(001010.11)_{2s} = (10.75)_{10}$

$(111110.11)_{2s} = -(000001.01)_2$

$= -(1.25)_{10}$

Floating Point Numbers (I)

- Fixed point numbers have limited range.
- To represent very large or very small numbers, we use floating point numbers (cf. scientific numbers).
  Examples:

0.23×10^{23} (very large positive number)

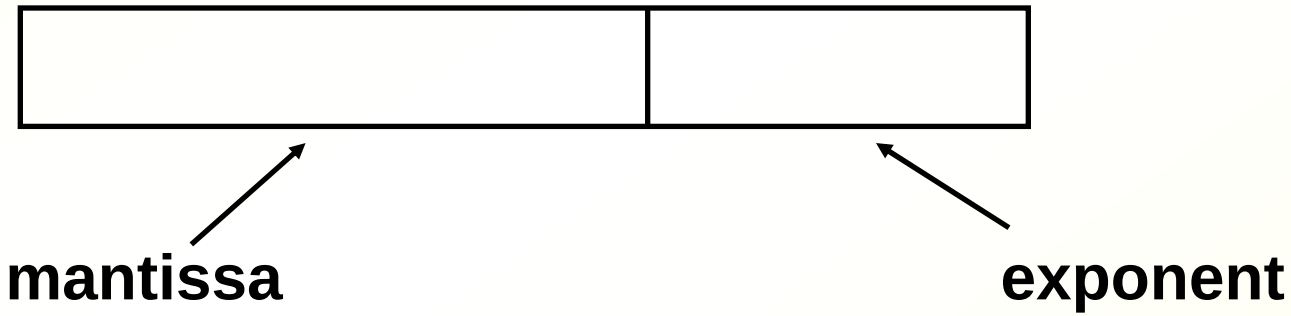
-0.1239×10^{-10} (very small negative number)

Floating Point Numbers (II)

- Floating point numbers have three parts:

  mantissa, base, and exponent

- *Base* is usually fixed for each number system.
- Therefore, needs only *mantissa* and *exponent*.



Floating Point Numbers (III)

- Mantissa is usually in normalised form:

  (base 10) 23×10^{21} normalised to 0.23×10^{23}

  (base 10) -0.0017×10^{21} normalised to -0.17×10^{19}

  (base 2) 0.01101×2^{3} normalised to 0.1101×2^{2}

- A 16-bit floating point number may have 10-bit mantissa and 6-bit exponent.
- More bits in exponent gives larger range.
- More bits for mantissa gives better precision.

FP Arithmetic +/-

- Check for zeros
- Align significands (adjusting exponents)
- Add or subtract significands
- Normalize result

FP Arithmetic x/÷

- Check for zero
- Add/subtract exponents
- Multiply/divide significands (watch sign)
- Normalize
- Round
- All intermediate results should be in double length storage

Arithmetic with Floating Point Numbers (I)

- Arithmetic is more difficult for floating point numbers.
- MULTIPLICATION

  Steps: (i) multiply the mantissa
  (ii) add-up the exponents
  (iii) normalise

  ◦ **Example:**

  $(0.12 \times 10^{2})_{10} \times (0.2 \times 10^{30})_{10}$

  $= (0.12 \times 0.2)_{10} \times 10^{2+30}$

  $= (0.024)_{10} \times 10^{32}$ **(normalise)**

  $= (0.24 \times 10^{31})_{10}$

Arithmetic with Floating Point Numbers (II)

- ADDITION

  Steps: (i) equalise the exponents

  (ii) add-up the mantissa

  (iii) normalise

  - **Example:**

    $(0.12 \times 10^3)_{10} + (0.2 \times 10^2)_{10}$
    $= (0.12 \times 10^3)_{10} + (0.02 \times 10^3)_{10}$ (equalise exponents)
    $= (0.12 + 0.02)_{10} \times 10^3$ (add mantissa)
    $= (0.14 \times 10^3)_{10}$

  - **Can you figure out how to do perform SUBTRACTION and DIVISION for (binary/decimal) floating-point numbers? (Hint: remember sometimes, the mantissa might need normalizing)**