FCPS Packets

Computer Number Systems

**Based upon material developed by
Sally Bellacqua, Mary Johnson, and Janet Mulloy
Revised by Charles Brewer, August 2004
Revised by Shane Torbert, July 2006
last revision July 2019**

**Fairfax County Public Schools
Fairfax, Virginia**

0. INTRODUCTION

All instructions and data in computers and calculators are performed on sequences of on-off switches, represented as sequences of 0's and 1's. It is convenient to think of sequences of 0's and 1's as binary (or base-2) numbers. Since the binary number system easily translates into the octal (base-8) and the hexadecimal (base-16) system, programmers need to think in all three number systems. While high-level languages, such as Java, do not require the programmer to input instructions and data in binary form, knowledge of binary arithmetic is useful in understanding how computers operate.

1. BINARY NUMBER SYSTEM

Our number system is a place-value system, meaning that the symbol's value depends on its place in the number. The "3" in "300" means something different from the "3" in "30", and the "3" in "30,000". As you know, our place-value system is grouped by groups of 10, starting from the right with the ones-place.

___	____	3	0	0
ten-thousands	thousands	hundreds	tens	ones

The base-2, base-8, and base-16 number systems also use place value, but the grouping is by groups of 2, 8, or 16, respectively. Here is "5" in base-2:

___	____	1	0	1
sixteens	eights	fours	twos	ones

The table to the right shows how numbers are represented in different bases.

Different bases group the digits differently. That's the reason for place value.

Try counting in base 3, 6, and 12.

Base 10 (decimal)	Base 2 (binary)	Base16 (hexadecimal)	Base 3	Base 6	Base 12
0	0	0			
1	1	1			
2	10	2			
3	11	3			
4	100	4			
5	101	5			
6	110	6			
7	111	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			
16	10000	10			
17	10001	11			
18	10010	12			
19	10011	13			
20	10100	14			

Our system of counting time has different bases. What is the base of seconds? ______
Of minutes? _____ Of hours? ______ Of months? ______

As you know from elementary school, a number can be expressed as the sum of each digit times its place value. For example,

$$\begin{aligned} 1394.2_{10} &= 1(10)^3 + 3(10)^2 + 9(10)^1 + 4(10)^0 + 2(10)^{-1} \\ &= 1000 + 300 + 90 + 4 + .2 \end{aligned}$$

The place value is the base raised to a power. The units place has a power of 0. Fractional place values have negative powers. It is customary to put a dot (called a "decimal point") to indicate where the whole number part ends and the fraction part begins. In general in a place value system,

$$b^8 b^7 b^6 b^5 b^4 b^3 b^2 b^1 b^0 . b^{-1} b^{-2} b^{-3} b^{-4}$$

Any number in base *b* can be converted to base 10 by summing the products of each digit and its place value. For example, given a number in binary,

$$\begin{aligned} 110101_2 &= 1(2)^5 + 1(2)^4 + 0(2)^3 + 1(2)^2 + 0(2)^1 + 1(2)^0 \\ &= 1(32) + 1(16) + 0(8) + 1(4) + 0(2) + 1(1) \\ &= 32 + 16 + 0 + 4 + 0 + 1 \\ &= 53_{10} \end{aligned}$$

Examples:

1) Convert $13F_{16}$ to base ten.

$$\begin{aligned} 13F_{16} &= 1(16)^2 + 3(16)^1 + F(16)^0 \\ &= 1(256) + 3(16) + 15(1) \\ &= 256 + 48 + 15 \\ &= 319_{10} \end{aligned}$$

2) Convert 1.02_3 to base 10

the place value of the "2" is 3^{-2} or $\frac{1}{9}$

therefore, $1.02_3 = 1\frac{2}{9} = 1.222222_{10}$

<u>Exercise 1</u>

1. How many digits are there in base 8 (octal)? ______ digits, from ____ through ____

2. Count to eighteen in octal. __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __

In problems 3 – 10, convert the base *b* number to its equivalent in base 10.

3. 159_{16} = _________ $_{10}$

4. 01101100_2 = _________ $_{10}$

5. $3A_{16}$ = __________ $_{10}$

6. 10110_2 = __________ $_{10}$

7. DC_{16} = ___________ $_{10}$

8. 1010.101_2 = ___________ $_{10}$

9. 5723_8 = __________ $_{10}$

10. 201_3 = _____________ $_{10}$

2. ADDING IN DIFFERENT BASES

Adding in different bases follows the familiar addition rules, including when to "carry" a digit. You just need to pay attention to the base. For example, $7_8 + 3_8$ has a carry operation, namely, $7_8 + 3_8 \rightarrow 10_{10} \rightarrow$ (1 group of 8 and 2 left over) $= 12_8$

carry sum

As another example,

$$B_{16} + 6_{16} \rightarrow 11_{10} + 6_{10} = 17_{10} \rightarrow \text{(1 group of 16 and 1 left over)} = 11_{16}$$

Addition in base 2 is so simple that you only have to memorize three rules:

$0_2 + 0_2 = 0_2$ and $1_2 + 0_2 = 1_2$ and $1_2 + 1_2 = 10_2$

carry sum

Also, memorize that $1_2 + 1_2 + 1_2 = 11_2$

carry sum

Exercise 2

Add in the indicated base. Then check your answers for 1 – 5 by doing the work in the decimal base (base 10).

1. $\begin{array}{r} 1_2 \\ \underline{+\ 1_2} \end{array}$ 2. $\begin{array}{r} 11_2 \\ \underline{+\ 01_2} \end{array}$ 3. $\begin{array}{r} 1\,1_2 \\ \underline{+\,1\,1_2} \end{array}$ 4. $\begin{array}{r} 1\,1\,1_2 \\ \underline{+\,0\,1\,1_2} \end{array}$

5. $\begin{array}{r} 0010\ 1110_2 \\ \underline{+\ 0011\ 1011_2} \end{array}$ 6. $\begin{array}{r} 342_8 \\ +\,\underline{517}_8 \end{array}$ 7. $\begin{array}{r} 3A9_{16} \\ +\ \underline{21C}_{16} \end{array}$ 8. $\begin{array}{r} 432_5 \\ +\ \underline{123}_5 \end{array}$

3. CONVERTING BETWEEN BINARY, OCTAL, and HEXADECIMAL

Binary numbers need lots of digits. Programmers noticed that binary can easily be converted to octal. Just group the binary number, e.g. 11001101, into groups of three digits (from the right side), then write the corresponding octal digits.

Example: 11 001 101 = 315_8

Similarly, given the same binary number, group it in groups of four digits, and substitute the hexadecimal digits.

Example: 1100 1101 $_2$ = CD_{16}.

Reversing the process also works.

Example: 62_8 = $110\ 010_2$

Example: 3B9 $_{16}$ = 0011 1011 1001 $_2$

Binary	Octal	Hex
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000		8
1001		9
1010		A
1011		B
1100		C
1101		D
1110		E
1111		F

Exercise 3

Convert between binary, octal, and hexadecimal.

1. 01011001_2 = _______$_8$ = ______ $_{16}$
2. 54_{16} = _________ $_2$ = _______$_8$
3. 01000001_2 = _______$_8$ = ______ $_{16}$
4. $3E_{16}$ = _________ $_2$ = _______$_8$
5. 00110100.0110_2 = ______ $_{16}$
6. $6A.5_{16}$ = _________________ $_2$

4. CONVERTING BASE-10 NUMBERS TO ANOTHER BASE

While we are primarily interested in being able to convert base-10 numbers to their binary (or hexadecimal) equivalents, there is a convenient algorithm for the conversion that works for all bases. The "Divide and Save" algorithm is simple to use and easy to implement in a computer program.

The technique is to use successive integer division by the desired base until a zero quotient is obtained. The remainders from each division (bottom to top) are written down (from right to left) as the result.

Example:

Convert the base-10 number 37 to binary. On paper, you start by dividing from the bottom:

$$
\begin{array}{lll}
0 & R\,1 & (one\ \ 2^5\ digit) \\
2\overline{)1} & R\,0 & (zero\ 2^4\ digit) \\
2\overline{)2} & R\,0 & (zero\ 2^3\ digit) \\
2\overline{)4} & R\,1 & (one\ \ 2^2\ digit) \\
2\overline{)9} & R\,0 & (zero\ 2^1\ digit) \\
2\overline{)18} & R\,1 & (one\ \ 2^0\ digit) \\
2\overline{)37} & &
\end{array}
$$

Therefore, $37_{10} = 100101_2$

Check the answer by converting back.

100101 =

Now you try. Convert 25_{10} to binary.

$2\overline{)25}$

Therefore, 25_{10} = __________ $_2$

Check by converting back.

Example:

Convert the base-10 number 110 to hexadecimal.

```
        0        R 6
16 | 6           R 14 → E
16 | 110
```

Therefore, $110_{10} = 6E_{16}$

Exercise 4

Convert the following base-10 numbers to the indicated base number.

1. 25_{10} = ____________ $_2$
2. 73_{10} = ____________ $_{16}$
3. 65_{10} = ____________ $_2$
4. 92_{10} = ____________ $_{16}$
5. 32767_{10} = ________________________ $_2$ = ________________________ $_{16}$
6. 7_{10} = ______________ $_3$

5. REPRESENTATION OF NEGATIVE INTEGERS—Two's Complement System

So far, we have only talked about positive integers. How should we represent negative integers, so that the binary integers are *signed*?

Computer scientists have invented several methods. The Intel family of CPUs uses the **two's-complement** method to represent negative integers, mostly because it makes addition extremely easy. If you add 2 and -2, you want to get 0. In the two's-complement system, 0000 0010 and 1111 1110 add by the standard binary addition rules directly to get 0.

```
  2      0000 0010
+-2    + 1111 1110
  0     10000 0000
discard
```

Somehow, 1111 1110 (in hex, FE) ought to represent -2. How can we see this?

One way to see this is to look at the table. 0 through 127 count up by ones in the standard binary counting system. If we add 1 more, 1000 0000 "wraps around" to become -128. If we keep counting up by ones, -2 turns out to be 1111 1110, which is what we wanted.

127	0111 1111	7F
126	0111 1110	7E
125	0111 1101	7D
. . .	. . .	
4	0000 0100	04
3	0000 0011	03
2	0000 0010	02
1	0000 0001	01
0	0000 0000	00
-1	1111 1111	FF
-2	1111 1110	FE
-3	1111 1101	FD
-4	1111 1100	FC
. . .	. . .	
-126	1000 0010	82
-127	1000 0001	81
-128	1000 0000	80

Thus, an 8-bit register can store 256 numbers, from -128 to 127 (in hex, from 80 to 7F). If you want to store bigger or smaller numbers, you need to get a bigger register. Notice that in the two's-complement system, all positive integers begin with 0 and all negative integers begin with 1.

In Sections 1-4, we were considering binary numbers and place value. We started at 0 and just kept counting. In Section 5, we are considering 8-bit registers and how to store both positive and negative numbers. This means that some binary sequences, e.g., 1111 1010, could represent two different numbers, either 250 or -6 (or characters, or colors, or sounds, or anything that can be digitized), depending on the situation.

From now on, let's specify we are using *signed binary integers in the two's complement system.* Binary numbers that begin with "0" are converted to base 10 by the place-value algorithm that was taught in Section 1. Binary numbers that begin with "1" are converted to their two's-complement form by the following algorithm.

Converting from signed binary (in the two's complement system) to base-10:

1) What base-10 integer does the *signed binary number*, 10111010, represent?

   Since the leading bit is a 1, the number must be negative; therefore, the two's-complement method is used to find the absolute value.

   Given – a signed binary number 1011 1010
   Step 1 – Form *one's complement* by "flipping" 0s and 1s 0100 0101

   Step 2 – Add 1 to form the *two's complement* + 1
   0100 0110

   Step 3 – Convert the two's complement to the base-10 absolute value $0100\ 0110_2 = 70_{10}$

   Step 4 – Insert the negative sign. Therefore, $1011\ 1010_2 = -70_{10}$.

2) What base 10 integer does the *signed binary number*, 0100 1011, represent? ____
   Since the leading bit is 0, we convert it using place values, as usual.

Converting a negative integer to the two's complement bit-pattern

To represent –6 as a two's-complement signed binary number:
First, change the absolute value of the number to binary $|-6_{10}| = 0000\ 0110$

Second, form the *one's complement* by "flipping" 0s and 1s 1111 1001
Third, add 1 + 1
to form the *two's complement* 1111 1010

Try it:

Convert -128 to its two's-complement representation:
Step 1 – Change | -128 | to binary _ _ _ _ _ _ _ _

Step 2 – Form the one's complement _ _ _ _ _ _ _ _
Step 3 – Form the two's complement + 1
_ _ _ _ _ _ _ _

Historical note: In this unit, we will use 8-bits instead of the 32-, or 64-bits that today's computers actually use. The Apple II and other personal computers of the 1970's were 8-bit computers.

<u>EXERCISE 5</u>

1 – 4. Form the one’s and two’s complements of the following bit strings.

BIT STRING	ONE’S COMPLEMENT	TWO’S COMPLEMENT
1010 1011		
0111 0000		
0000 0001		
0000 0000		

5 – 8. Represent each negative integer in 8-bit two’s complement form, then in hex.

DECIMAL INTEGER	ABSOLUTE VALUE IN BINARY	ONE’S COMPLEMENT	TWO’S COMPLEMENT	HEX
-1				
-2				
-30				
-8				

6. ADDING SIGNED BINARY NUMBERS

Any time you enter, e.g. -2, we have seen that two's complement system translates that into 1111 1110. This may seem complicated, but in this system, *all* arithmetic operations (+ - * / % √) become easier. Addition is just the addition of bits. Subtraction is the addition of two’s complement. (Computers and calculators don’t actually subtract; they just add the two's complement.) Furthermore, multiplication is repeated addition. Division is repeated two’s complement addition. No wonder programmers like the two’s complement system.

Example:
What is the answer to 10_{10} + (– 10_{10}) ?

The 8-bit binary representation of 10 is 0000 1010

The 8-bit binary representation of –10 is the two’s-complement

0000 1010
1111 0101
$\underline{\quad\quad + 1}$
1111 0110

Add the binary numbers and discard the overflow digit, the result being $0000\ 0000_2$ or 0_{10}.

0000 1010
$\underline{1111\ 0110}$
1 0000 0000

Example:
Add these two 8-bit signed binary numbers. Check your work by converting all three numbers to base ten.

1110 1100
+ $\underline{1101\ 0110}$

Check – You check your binary addition by converting all three numbers to base ten. Since all three are negative in this example, the two's complement method (see Section 5) will be used to find their absolute values. Here is how you should have done it:

ORIGINAL BINARY → (1's comp + 1 = 2's comp) → DECIMAL

$$\begin{aligned}
&1110\,1100 \rightarrow 0001\,0011 \;+\; 1 \;=\; 0001\,0100 \rightarrow -20 \\
&\underline{1101\,0110} \rightarrow 0010\,1001 \;+\; 1 = 0010\,1010 \rightarrow -\underline{42} \\
&1100\,0010 \rightarrow 0011\,1101 \;+\; 1 = 0011\,1110 \rightarrow -62
\end{aligned}$$

<u>Exercise 6</u>

Add these 8-bit signed binary numbers (using two's complement). Check your work by converting all three numbers to base-10. (The check is more work than the original addition problem.)

1. 0000 1001
+ <u>1111 1100</u>

2. 0001 1100
+ <u>1011 0101</u>

Represent these decimal numbers as 8-bit signed binary numbers. Add the binary number and then convert each answer back to base-10 as a check.

3. - 7
+ <u>9</u>

4. -17
+ <u>-13</u>

7. REPRESENTATION OF REAL NUMBERS

Real numbers are represented using one of the standard formats, either *float* or *double*, as defined by The Institute of Electrical and Electronic Engineers (IEEE). Like scientific notation, these formats use a mantissa (or fractional part) and an exponent part, but of course expressed as binary numbers. For most numbers, the mantissa is *normalized*, i.e., adjusted to be of the form 1.xxxxxxxx, and then the actual mantissa drops the leading 1. In order to handle negative exponents, 127_{10} (or 01111111) is added the exponent. Thus, 10000000 represents an exponent of 1.

Some examples of 32-bit floats:

Binary | Normalized Mantissa | Exponent | Sign 1 bit | Exponent +127 8 bits | Actual Normalized Mantissa 23 bits

$3_{10} = 11_2 = 1.1_2 * 2^1 \rightarrow$ 0 1000 0000 100 0000 0000 0000 0000 0000

$3/4_{10} = 0.11_2 = 1.1_2 * 2^{-1} \rightarrow$ 0 0111 1110 100 0000 0000 0000 0000 0000

$-10_{10} = -1010_2 = -1.010_2 * 2^3 \rightarrow$ 1 1000 0010 010 0000 0000 0000 0000 0000

$7.5_{10} = 0111.1_2 = 1.111_2 * 2^2 \rightarrow$ 0 1000 0010 111 0000 0000 0000 0000 0000

$0.1_{10} = .110011001100110011001100110011001100\ldots_2 =$
$1.10011001100110011100 * 2^1 \rightarrow$ 0 1000 0000 100 1100 1100 1100 1100

As you can see, some decimal numbers have an infinite expansion when expressed in binary form. Thus, 0.1 is approximated as 0.100000001490116119384765625 in the computer. This is an example of a *round-off error*.

The number line is infinite and infinitely dense, but not so with floats and doubles, which are dropped on the number line like breadcrumbs. Since the number of distinct values between consecutive powers of two is constant, floats and doubles are denser closer to zero. Also, very large values have very large gaps between them. All this has real-world consequences for computer scientists, including *overflow*, *underflow*, and *swamping*.

In a 4-bit system, using 2 bits for the exponent and 2 bits for the mantissa, unnormalized and without signs, you can represent 16 values, distributed as shown:

0000	0001	0010	0011	0100	0101	0110	0111	**1000**		1001		1010		1011		**1100**				1101				**1110**				1111				
0	.125	.25	.375	.5	.675	.75	.875	**1**		1.25		1.5		1.75		**2**				2.5				**3**				3.5				

Here is an example of *swamping*: 2.000 + 0.125 = 2.000 because 2.000 is the closest value to 2.125 that we can represent (the next-closest value is 2.500).

Special Values	**float**	**double**
	32 bits: 1 for sign 8 for exponent (E) 23 for mantissa (F)	64 bits: 1 for sign 11 for exponent (E) 52 for mantissa (F)
NaN (Not a Number)	E is 255 and F is nonzero 01111111100000100000000000000000 11111111100000100000000000000000	E is 2047 and F is nonzero
Infinity or -Infinity	E is 255 and F is 0 01111111100000000000000000000000 11111111100000000000000000000000	E is 2047 and F is 0
0 or -0	E is 0 and F is 0 00000000000000000000000000000000 10000000000000000000000000000000	E is 0 and F is 0
largest positive value	E is 254 and F is all 1's. i.e. 2^127 ≈ 10^38 01111111011111111111111111111111	E is 2047. 2^1037≈10^307
smallest positive value.	E is 0, F is 1. i.e. 2^-149 ≈ 10^-45 00000000000000000000000000000001	E is 0, F is -52 i.e. 2^-1074 ≈ 10^-324
epsilon	F is -23. i.e. 2^-24 ≈ 10^-8	F is -52. i.e. 2^-52 ≈10^-16

Notice that NaN, Infinity, and Negative Infinity are actually stored in `double` variables.

One important value is given the special name *machine epsilon*. Epsilon is the smallest number that can be added to 1 and yield a result that is different from 1. That is, for doubles, 1.0 + something less than 10^-16 is indistinguishable from 1.0. The epsilon for the toy 4-bit representation is 0.125, half-way between the two breadcrumbs 1.000 and 1.250.

Note that machine epsilon is not the smallest positive number that can be represented, which is, for floats, 2^-149 ≈ 10^-45 and, for doubles, 2^-1074 or 10^-324.

Code Peculiarities in Java

input	**output**	**explanation**
double x = Math.sqrt(-1); System.out.println(x +" " + (x == x));	NaN false	// Evidently not-a-number is // not equal to itself.
double y = 1.0 / 0.0; System.out.println(y +" " + (y == y+ 1));	Infinity true	// infinity equals infinity-plus- // one.
double z = -1.0 / 0.0; System.out.println(y +" " + (y == y*2));	-Infinity true	// negative infinity equals // a doubled negative infinity

double a, b;
if(a == b)
 	
if(Math.abs(a – b) < 0.00000001)
		// not safe! Never compare
		// doubles for equality.
		// Always compare within a
		// tolerance.

8. MULTIPLICATION IN BINARY

The Arithmetic Logic Unit (ALU) in the CPU multiplies binary numbers using a “shift left” and “shift right” logic. The following function uses a multiplying algorithm that illustrates the operation of this shift logic.

```
int result (int x, int y)
{
  int z = 0;
  while ( y != 0 )
   {
    if (y % 2 == 1)              // If y is odd add x to interim result (z)
      z = z + x;
    x = 2 * x;                   // Shifts binary number left
    y = y / 2 ;                  // Shifts binary number right
   }
  return z;                      // When y is 0, return interim result
}
```

A call to this function with x = 7 and y =13 would return 91 in this way:

	Decimal			Binary		
Iteration	z	x	y	z	x	y
0	0	7	13	0000 0000	0000 0111	0000 1101
1	7	14	6	0000 0111	0000 1110	0000 0110
2	7	28	3	0000 0111	0001 1100	0000 0011
3	35	56	1	0010 0011	0011 1000	0000 0001
4	**91**	112	0	**0101 1011**	0111 0000	0000 0000

9. REPRESENTATIONS OF CHARACTERS

Characters are normally represented on PCs using the 8-bit ASCII (American Standard Code for Information Interchange). The ASCII code defines 128 characters from 0 to 127 using the first seven of the eight bits in a byte. The codes for letters start at 65 (or 01000001 or 41_{16}) for ‘A’ and end with 122 (or 01111010 or $7A_{16}$) for ‘z’. Look at the partial ASCII chart on the next page, or at http://www.ascii-code.com/ . Complete this table:

DEC	OCT	HEX	BIN	Symbol
			01000001	
97				
	132			
		30		
			00100000	<space>

ASCII handles Latin characters. To include the letters of other languages, a 16-bit representation of characters called Unicode was created. Unicode is used by the Java compiler. Still another code is used to represent oriental languages.

Some mainframe computers use still another format for character representation called EBCDIC (Extended Binary Coded Decimal Interchange Code).

DEC	OCT	HEX	BIN	Symbol
48	060	30	00110000	0
49	061	31	00110001	1
50	062	32	00110010	2
51	063	33	00110011	3
52	064	34	00110100	4
53	065	35	00110101	5
54	066	36	00110110	6
55	067	37	00110111	7
56	070	38	00111000	8
57	071	39	00111001	9
58	072	3A	00111010	:
59	073	3B	00111011	;
60	074	3C	00111100	<
61	075	3D	00111101	=
62	076	3E	00111110	>
63	077	3F	00111111	?
64	100	40	01000000	@
65	101	41	01000001	A
66	102	42	01000010	B
67	103	43	01000011	C
68	104	44	01000100	D
69	105	45	01000101	E
70	106	46	01000110	F
71	107	47	01000111	G
72	110	48	01001000	H
73	111	49	01001001	I
74	112	4A	01001010	J
75	113	4B	01001011	K
76	114	4C	01001100	L
77	115	4D	01001101	M
78	116	4E	01001110	N
79	117	4F	01001111	O
80	120	50	01010000	P
81	121	51	01010001	Q
82	122	52	01010010	R
83	123	53	01010011	S
84	124	54	01010100	T
85	125	55	01010101	U
86	126	56	01010110	V

DEC	OCT	HEX	BIN	Symbol
87	127	57	01010111	W
88	130	58	01011000	X
89	131	59	01011001	Y
90	132	5A	01011010	Z
91	133	5B	01011011	[
92	134	5C	01011100	\
93	135	5D	01011101	]
94	136	5E	01011110	^
95	137	5F	01011111	_
96	140	60	01100000	`
97	141	61	01100001	a
98	142	62	01100010	b
99	143	63	01100011	c
100	144	64	01100100	d
101	145	65	01100101	e
102	146	66	01100110	f
103	147	67	01100111	g
104	150	68	01101000	h
105	151	69	01101001	i
106	152	6A	01101010	j
107	153	6B	01101011	k
108	154	6C	01101100	l
109	155	6D	01101101	m
110	156	6E	01101110	n
111	157	6F	01101111	o
112	160	70	01110000	p
113	161	71	01110001	q
114	162	72	01110010	r
115	163	73	01110011	s
116	164	74	01110100	t
117	165	75	01110101	u
118	166	76	01110110	v
119	167	77	01110111	w
120	170	78	01111000	x
121	171	79	01111001	y
122	172	7A	01111010	z

The Primitive Data Types in Java

	Keyword	Size	Range	Intel Format	Typical Use
Integers	`byte`	8-bit	-128 to 127	two's complement	where space is a concern
	`short`	16-bit	-32768 to 32767	two's complement	
	`int`	32-bit	-2^{31} to $2^{31}-1$	two's complement	for integers
	`long`	64-bit		two's complement	elapsed time in milliseconds, large Fibonacci numbers financial calculations
Real	`float`	32-bit	approx 4.28 billion	IEEE 754	
	`double`	64-bit	approx 1.84 * 10^19	IEEE 754	scientific uses
	`char`	16-bit		Unicode character	single letters or characters
	`boolean`	1 bit		0, 1	true or false

10. REVIEW EXERCISES

1. What is the largest digit in base 16? ____________

2. What is the base-ten value of the underlined bit in the binary number 0 1 1 0 <u>1</u> 1 1 0? ____________

3. Convert 124_5 to base 10. ____________

4. Express the hexadecimal number FF in base ten. ____________

5. Express $12A_{16}$ in binary, then in octal. ____________ ____________

6. Halloween = Christmas because 31 OCT = ___ DEC

7. Find the two's complement of 1101 0111. ____________

8. Add these hexadecimal numbers. Leave your answer in hexadecimal.

$$\begin{array}{r} 4F \\ + \underline{68} \end{array}$$

____

9. 1100 1111 is the signed 8-bit representation of what base ten number? ____________

10. Convert 37_{10} to binary. ____________

11. Express the binary 0101 1011.0110 in hex and in octal. __________ _________

12. Express 87_{10} in base sixteen. ____________

13. Convert 23_{10} to its signed 8-bit representation. ____________

14. Perform the following addition of two 8-bit signed binary numbers. Check your work by converting each binary number into base ten.

$$\begin{array}{r} 1111\ 0011 \\ + \underline{0000\ 1101} \end{array}$$

15. Convert the following base ten numbers to binary and add. Check your result by converting the answer back to base ten.

$$\begin{array}{r} -37 \\ + \underline{\ \ 23} \\ -14 \end{array}$$