Supplement Notes on Data Representation

1.1 Introduction

In most of our work with numbers and computers we must be very careful to distinguish between a ***value*** and its ***representation***. There is usually little distinction between a value, say 13, and its representation in decimal. Consider however MCMXCVIII, which is a different way of representing 1998 (and MCMXCVIII+II = MM — this should be obvious to you!).

As far as we are concerned (but not in Roman numbers!) values are always represented as a sequence of digits ($x_{n-1}, x_{n-2}, \ldots, x_1, x_0$) and a base b. A value N with base b and n digits is given by

$$N = x_{n-1} b^{n-1} + x_{n-2} b^{n-2} + \ldots + x_1 b^1 + x_0$$

The value is represented by a polynomial in the *base*, with the *digits* of the representation being the *coefficients* of the polynomial. Each coefficient x_i is in the range $0 <= x_i < b$. If the base is 10, things aren't very interesting. A number such as **56432** means

$$5 \times 10^4 + 6 \times 10^3 + 4 \times 10^2 + 3 \times 10 + 2$$

1.1.1 Decimal

If you see a price tag of $123 on a book, do you know how much it costs? Your answer will clearly be 'one hundred and twenty three dollars', because you are used to the **decimal system**, also called **base ten**. Each digit is given a place value according to its position in the number, or weighting.

Decimal
A system of counting in tens, also called denary or base ten.

Weighting
The quantity you multiply by to find the true value.

For example, the number 123_{10} is worked out like this:

$$123_{10} = (1 \times 10^2) + (2 \times 10^1) + (3 \times 10^0)$$

Weight = 10^2 Weight = 10^1 Weight = 10^0

But why do the weights go by 1, 10, and 10^2, and so on? The probable reason is that we have ten fingers, so we invented only the digits 1, 2, 3, 4, 5, 6, 7, 8, 9 and 0.

1.1.2 Binary

How does a computer code number? How many 'fingers' does a computer have? An electric plus can be on or off. A magnetic pole can be north or south. In other words, the simplest part of a computer has only two states (represented by 0 and 1).

The most natural coding system for computers is therefore the **binary** system or **base two**. Binary is a system of counting in twos. Each Binary digit is usually called a **bit**.

For example, the number 1010_2 represents:

$$1010_2 = (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (0 \times 2^0)$$

Weight = 2^3 Weight = 2^2 Weight = 2^1 Weight = 2^0

Note:

- Computer memory is based on the electrical representation of data.
- Each memory position is represented by a bit which can be either 'on' or 'off'. This makes it easier to represent computer memory using a base 2 number system rather than base 10 decimal system.
- 1 represents an 'on' value, 0 represents an 'off' value.
- The left-most bit is called the high-order bit (most-significant) and the right-most bit is called the low-order bit (least-significant).

1.1.3 Octal

An octopus with eight arms would probably have invented only the digits 1, 2, 3, 4, 5, 6, 7 and 0! How would an octopus interpret the price $123?

The weights which it would assign to digits are 1, 8, 8^2 and so on. Hence the number represented would be:

$$123 = (1 \times 8^2) + (2 \times 8^1) + (3 \times 8^0)$$

Weight = 8^2 Weight = 8^1 Weight = 8^0

The coding is known as **octal** or **base eight**.

When we are talking about more than one number system at the same time, it becomes rather confusing unless we state clearly the base of each number. For example, we must write 123_{10} to represent the decimal 123 and 123_8 to represent the octal 123.

1.1.4 Hexadecimal

Hexadecimal is a system of counting in sixteens, also called base **sixteen**. The digits which it uses are as follows:

Decimal Digit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Hexadecimal Digits	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

For example, the number $12B_{16}$ represents:

$$12B = (1 \times 16^2) + (2 \times 16^1) + (11 \times 16^0)$$

Weight = 16^2 Weight = 16^1 Weight = 16^0

Example 1:

(a) Base 10 - Decimal $138_{10} = 1 \times 10^2 + 3 \times 10^1 + 8 \times 10^0$ $5729_{10} = 5 \times 10^3 + 7 \times 10^2 + 2 \times 10^1 + 9 \times 10^0$
(b) Base 2 – Binary $10101_2 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 16 + 0 + 4 + 0 + 1 = 21_{10}$ $1010_2 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 8 + 0 + 2 + 0 = 10_{10}$
(c) Base 8 – Octal $172_8 = 1 \times 8^2 + 7 \times 8^1 + 2 \times 8^0 = 64 + 56 + 2 = 122_{10}$ $26_8 = 2 \times 8^1 + 6 \times 8^0 = 16 + 6 = 22_{10}$
(d) Base 16- Hexadecimal $260_{16} = 2 \times 16^2 + 6 \times 16^1 + 0 \times 16^0 = 512 + 96 + 0 = 608_{10}$ $4b_{16} = 4 \times 16^1 + 11 \times 16^0 = 64 + 11 = 75_{10}$
(e) Base b $N = X_{n-1}b^{n-1} + X_{n-2}b^{n-2} + \ldots + X_1b^1 + X_0$

1.2 Conversion between Different Bases

In converting numbers we must perform arithmetic in some base (usually base 2 on computers, base 10 for humans) — call this the *native base*.

1.2.1 Converting from a base into internal form

There are two ways of converting, based on different ways of evaluating the polynomial.

1) The first assumes that we know the powers of the base. We multiply each of the powers by its appropriate digit and add the values, as was done in the example

$$1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2 + 1 = 16 + 4 + 1 = 21$$

2) The second writes the polynomial in a better form for computer evaluation.

$$ax^4 + bx^3 + cx^2 + dx + e = e + x(d + x(c + x(b + xa\ )))$$

The number here would appear as *abcde*. Assume a "value so far" (V), which is initially set to 0. Working from the **left-most** (most significant) digit, multiply V by the base and add in the next digit.

Example 2:

(a) Convert Binary to Decimal

(1) 01 011 111_2 Method 1: $0 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$ Answer: 1 x 64 + 0 x 32 + 1 x 16 + 1 x 8 + 1 x 4 + 1 x 2 + 1 x 1 = 95
(2) 11 010 001_2 Method 1: $1 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$ Answer: 1 x 128 + 1 x 64 + 0 x 32 + 1 x 16 + 0 x 8 + 0 x 4 + 0 x 2 + 1 x 1 = 209
Exercise **01 101 001_2** Answer:

(b) Convert Octal to Decimal

(1) 127_8 Method 1: $1 \times 8^2 + 2 \times 8^1 + 7 \times 8^0$ Answer: 64 + 16 + 7 = 87
(2) 235_8 Method 2: (((2 x 8) + 3) x 8) + 5 Answer: ((19) x 8) + 5 = 152 + 5 = 157
Exercise **151_8** Answer:

(c) Convert Hexadecimal to Decimal

(1) $1A7_{16}$ Method 1: $1 \times 16^2 + 10 \times 16^1 + 7 \times 16^0$ Answer: 256 + 160 + 7 = 423
(2) $23E_{16}$ Method 2: $2 \times 16^2 + 3 \times 16^1 + 14 \times 16^0$ Answer: 512 + 48 + 14 = 574
Exercise **$15B_{16}$** Answer:

1.2.2 Converting from internal form into a base

Set the working value V to the number to convert. Then calculate

$$d = V \,\%\, base; \text{ and}$$
$$V = V / base$$
$$\text{until } (V = 0).$$

The successive values of *d* are the digits in order, from **least-significant** (right most) **to most-significant** (left most). [Note: V % base returns the *remainder* on division, and V/base returns the *quotient*.]

Example 3:

To convert a decimal number to binary, octal or hexadecimal number, divide the number by the required base, the resulting remainder, in reverse order represent the required value.

(a) Convert Decimal to Binary

(1) 21_{10} 2 |21 remainder = 1 2 |10 remainder = 0 2 |5 remainder = 1 2 |2 remainder = 0 1 Answer: 10101_2	**(2) 10_{10}** 2 |10 remainder = 0 2 |5 remainder = 1 2 |2 remainder = 0 1 Answer: 1010_2	Exercise: **156_{10}** Answer:

(b) Convert Decimal to Octal

(1) 122_{10} 8 |122 remainder = 2 8 |15 remainder = 7 1 Answer: 172_8	**(2) 220_{10}** 8 |220 remainder = 4 8 |27 remainder = 3 3 Answer: 334_8	Exercise **1234_{10}** Answer:

(c) Convert Decimal to Hexadecimal

(1) 1940_{10}	**(2) 175_{10}**	Exercise

16 |1940 remainder = 4 16 |121 remainder = 9 7 Answer: 794_{16}	16 |175 remainder = F 10 Answer: AF_{16}	**689_{10}** Answer:

1.2.3 Converting between Octal, Binary and Hexadecimal

Pure binary numbers have so many digits that they are often difficult to handle. We usually collect bits in groups of 3 (base–8 or octal) or 4 (base–16, or hexadecimal, *NOT "hexidecimal"*) to get numbers with fewer digits that are easier for people to handle. (*Note that the bits must be **grouped from the right**, and **high-order zeros** inserted if necessary to fill out the left-most digit.*)

Octal

Bits	Digit
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7

Hexadecimal

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

Example 4:

(a) Convert Binary to Octal

To convert from Binary to Octal just replace the binary pattern with the corresponding octal digit.

(1) 001 011 111_2

$0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$ $0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$ $1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$

Answer: 137_8

(2) 011 010 001_2

$0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$ $0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$ $0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$

Answer: 321_8

101 101 001_2

(b) Convert Binary to Hexadecimal

To convert from Binary to Hex just replace the binary pattern with the corresponding hex digit.

(1) 0101 1111_2

$0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$ $1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$ **Answer: $5F_{16}$**

(2) 1101 0001_2

$1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$ $0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$ **Answer: $D1_{16}$**

101 001 101_2

(c) Convert Octal to Binary

To convert from Octal to Binary just replace the octal digit with the corresponding binary pattern.

(1) 363_8 ↙ ↓ ↘ 011 110 011_2 Answer = 011110011	**(2) 247_8** ↙ ↓ ↘ 010 100 111_2 Answer = 010100111	Exercise: **123_8** Answer =

(d) Convert Octal to Hexadecimal

To convert from Octal to Hexadecimal just replace the octal digit with the corresponding binary pattern first and then regroup binary numbers in a group of four and replace with the corresponding hexadecimal pattern.

(1) 363_8 -> 011 110 011 -> 1111 0011 Answer: $F3_{16}$	**(2) 247_8** -> 010 100 111 -> 1010 0111 Answer: $A7_{16}$	Exercise: **123_8** Answer:

(e) Convert Hexadecimal to Binary

To convert from Hex to Binary just replace the hex digit with the corresponding binary pattern.

(1) $EA3_{16}$ ↙ ↓ ↘ 1110 1010 0011_2 Answer = 111010100011	**(2) $2A7_{16}$** ↙ ↓ ↘ 0010 1010 0111_2 Answer = 001010100111	Exercise: **$1F3_{16}$** Answer =

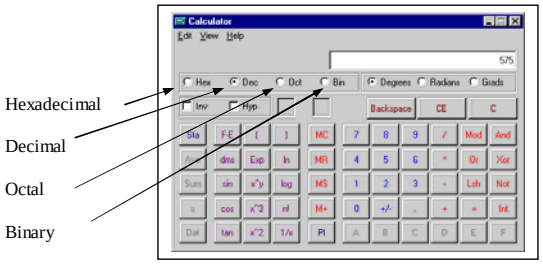
(f) Convert Hexadecimal to Octal

To convert from Hexadecimal to Octal just replace the hexadecimal digit with the corresponding binary pattern first and then regroup binary numbers in a group of three and replace with the corresponding octal pattern.

(1) $2A7_{16}$ -> 0010 1010 0111 ->\ 001 010 100 111 Answer: 1247_8	**(2) $EA3_{16}$** ->\ 1110 1010 0011 ->\ 111 010 100 011 Answer: 7243_8	Exercise **$1F3_{16}$** Answer:

1.2.4 Calculator

1. You can use the Calculator in the **scientific** mode to check results of decimal to hex, binary, and octal conversions.



1.2.5 Java Programming

- **Octal** numbers are always beginning with a **zero**, so 61 would be written as 061.
- **Hex** numbers are always preceded by **0x** so 31 would be written as 0x0031.

Example 5:

OctalNum: $\mathbf{325_8} = 3 \times 8^2 + 2 \times 8^1 + 5 \times 8^0 = 192 + 16 + 5 = 213_{10}$

HexNum: $\mathbf{12_{16}} = 1 \times 16^1 + 2 \times 16^0 = 16 + 2 = 18_{10}$

```
public class Representation {
  public static void main(String[] args) {
    int octalNum = 0325;
    System.out.print("Number in Octal: ");
    System.out.println(Integer.toOctalString(octalNum));
    System.out.println("Number in Decimal: " + octalNum);
    int hexNum = 0x12;
    System.out.print("Number in Hex: ");
    System.out.println(Integer.toHexString(hexNum));
```



```
    System.out.println("Number in Decimal: " + hexNum);
  }
}
Answer:
Number in Octal: 325
Number in Decimal: 213
Number in Hex: 12
Number in Decimal: 18
```

Note: You can also use the Java Integer Wrapper class to output binary, hex and octal Strings.

- Integer.toBinaryString(octalNum); // output = 11010101
- Integer.toOctalString(octalNum); // output = 325
- Integer.toHexString(octalNum); // output = D5

Example 6:

You can use Integer.parseInt(String s, int base) method to parse the string argument as a signed integer in the base specified by the second argument.

```
try {
      // read the number from Keyboard
      BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
      System.out.print("Enter the number in decimal:");
      String numberInStr = in.readLine();
      int num = Integer.parseInt(numberInStr);
      System.out.println();
      System.out.println("Number in Binary: " + Integer.toBinaryString(num));
      System.out.println("Number in Octal: " +  Integer.toOctalString(num));
      System.out.println("Number in Hexadecimal: " + Integer.toHexString(num));
   } catch (Exception e) {
      System.err.println("Error");
}
```

Example 1:

```
Enter the number in decimal:20
Number in Binary: 10100
Number in Octal: 24
Number in Hexadecimal: 14
```

Example 2:

```
Enter the number in decimal:10000
Number in Binary: 10011100010000
Number in Octal: 23420
Number in Hexadecimal: 2710
```

Note:

- To read a Binary number (base 2), we use
  `int n = Integer.parseInt("1010", 2) // n = 10 in decimal`
- To read an Octal number (base 8), we use
  `int n = Integer.parseInt("15", 8) // n = 13 in decimal`
- To read a Hexadecimal number (base 16), we use
  `int n = Integer.parseInt("1F", 16) // n = 31 in decimal`
- More examples:

```
Integer.parseInt("-0", 10); // returns 0
Integer.parseInt("-FF", 16); // returns -255
Integer.parseInt("2147483647", 10); // returns 2147483647
Integer.parseInt("99", 8); // throws a NumberFormatException (invalid digit)
Integer.parseInt("Kona", 10); // throws a NumberFormatException (invalid)
```

Exercise 1:

What is the output from the above program if user enters "46".

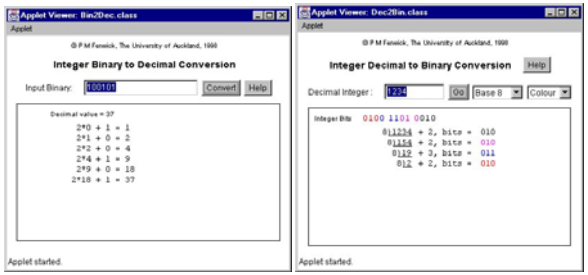
```
Enter the number in decimal:46
```

Exercise 2:
What is the output of the following program? Why?

```
public class Ex1 {
  public static void main(String[] args) {
    int n1 = 10;
    int n2 = 010;
    int n3 = 0x10;
    System.out.println(n1);
    System.out.println(n2);
    System.out.println(n3);
  }
}
```

Output:

1.2.6 Applet for Converting numbers between Binary, Octal and Hexadecimal



1.3 Arithmetic Operations

1.3.1 Adding numbers

The rules for adding numbers are very similar in all number bases, provided that we can add pairs of digits. We can use our familiar decimal addition with appropriate changes. Starting from the right (least significant) digit,

- add each pair of digits.
- if the sum is more than the base, subtract the base and generate a "carry" to include in the next addition to the left.

Example 7:

```
  192
+ 125
   100 carries
  317
```

For each position, proceeding from right to left, we add the digits and the incoming carry. If the sum is not less than the base, enter a 1 as the carry-in to the next position to the left and subtract the base from the sum; enter the difference as the sum digit. If we are adding two numbers x and y in base b, with the digits $x_{N-1}\ldots x_3\, x_2\, x_1\, x_0$ and $y_{N-1}\ldots y_3\, y_2\, y_1\, y_0$ to give a sum $z_{N-1}\ldots z_3\, z_2\, z_1\, z_0$ we can hold each of the sets of digits in an integer array and add them with the program. The operation follows exactly from the description above.

Algorithm

```
Carry = 0;
for (i = 0; i < N; i++)  {          // right to left scan
      z[i] = x[i] + y[i] + Carry;    // the add
      if (z[i] >= base)   {          // digit overflow!!
            Carry = 1;               // carry to next stage
            z[i] = z[i] - base;      // correct overflow
      } else
            Carry = 0;               // no carry to next stage
}
```

Example 8:

(1) Base 2:	(2) Base 8:	(3) Base 16:
1010	237	13a
+ 0011	+ 162	+ 1b9
0100 carries	110 carries	010 carries
1101	421	2f3

Exercises:

Base 2:	Base 8:	Base 16:
01011111	363	F3
+ 00010001	+ 247	+ a7

1.3.2 Subtracting numbers

Subtraction is similar to addition. We work in the same direction (right to left), but now must *borrow* if the subtraction "cannot be done" rather than *carry*. Again we can just use our familiar decimal subtraction with appropriate changes. Starting from the right (least significant) digit,

- subtract each pair of digits.
- if the result is less than zero, add on the base to the result and generate a "borrow" to include in the next subtraction to the left.

Before examining binary subtraction it is best to consider decimal subtraction and especially the action of the "borrow". ("Borrowing" is an aspect which is seldom well-explained.) As an example, take 416 – 263.

Example 9:

```
  416
- 263
  100 borrows
  153
```

- Remember always that any generated digit d of the difference must be such that $0 <= d < 10$.
- The first subtraction, of the unit digits, is $6 – 3 = 3$ with no problem.
- The next, tens digits subtraction yields, $1 – 6 = –5$, which is outside the valid range.
- To correct this "overdraw", *add 10* (the number base) to the tens digit of the minuend and compensate by *subtracting 1* from the hundreds digit of the minuend. (Both correspond to an adjustment of 1 in the hundreds digit and have no overall effect.) The tens digit subtraction is now $11 – 6 = 5$, which is a valid result.
- With the borrow of 1, the hundreds digit subtraction is no longer $4 – 2$ but $(4 – 1) – 2 = 3 – 2 = 1$.
- The difference is then 153.

The actions in binary subtraction are identical; if the subtraction "won't go", add the base (10_2) to the minuend digit and decrement the next most-significant minuend digit by 1. (A better way of binary subtraction is given later!) If we are subtracting two numbers x and y in base b, with the digits $x_{N-1}\ldots x_3\, x_2\, x_1\, x_0$ and $y_{N-1}\ldots y_3\, y_2\, y_1\, y_0$ to give a result $z_{N-1}\ldots z_3\, z_2\, z_1\, z_0$ ($x – y \rightarrow z$), we can hold each of the sets of digits in an integer array and subtract them with the following algorithm.

Algorithm

```
Borrow = 0;
for (i = 0; i < N; i++) {
      z[i] = x[i] - y[i] - Borrow;
      if (z[i] < 0) {
            Borrow = 1;
            z[i] = z[i] + base;
      } else
            Borrow = 0;
}
```

Example 10:

(1) Base 2:	(2) Base 8:	(3) Base 16:
0101	237	139
- 0011	- 160	- ba
0100 borrows	100 borrows	110 borrows
0010	57	7f

Exercises:

Base 2:	Base 8:	Base 16:
01011111 - 00010001	363 - 247	F3 - a7

1.3.3 Applet for Arithmetic Operations

1.4 Negative Number Representation

When we were discussing binary numbers above, we only consider **unsigned** (by default) positive integers. Although positive integers (the "natural" numbers) are important, they are inadequate for practical arithmetic. To allow negative values as well we require a ***signed number*** representation. In the following section we will be considering how binary numbers can represent both positive and negative values, i.e. signed integers. In a later section, we will see "floating point" numbers to represent real numbers.

2.4.1 Representation

We now introduce ways of *representing* positive and negative *values*.

- There are several ways of representing signed binary numbers.
- All representations are based on the unsigned (positive) number representation.
- All use the left-most bit as the sign – usually 0 (positive), and 1 (negative).
- Most represent positive integers exactly as with an unsigned representation.

An important operation is that of *complementing* a value, or changing its sign. The complement of +3 is –3, and of –46 is +46. We can complement a positive value (to get a negative value) or a negative value (to get a positive value).

There are three important representations for signed binary numbers. All represent positive integers as for unsigned integers and all reserve the most-significant bit as the sign bit 0 (+ve), and 1 (–ve).

1. **Sign and Magnitude**. We need one bit to represent whether a number is positive or negative. Only the sign bit changes when complementing (changing the sign of) the number. In 8 bits, +5 is 00000101, and –5 is 10000101. Sign and magnitude representation is used only in the significands of floating point numbers and is otherwise unimportant.

Sign bit → 10000101 = -5
Sign bit → 00000101 = +5

Sign bit → 11111010= -5
Sign bit → 00000101 = +5

2. **Excess" or "biased" representation.** The one disadvantage of Two's complement is that you can't sort the numbers. Suppose you had some hardware that could sort unsigned numbers or compare two unsigned int numbers and tell you which number was larger or if they were equal, this leads to another idea for representation, "Excess" or "biased" representation.
The following table shows 3-bit binary numbers using unsigned representation in order.

Representation	Value
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Suppose we wanted to represent negative numbers, but wanted to keep the same ordering where 000 represents the smallest value and 111 represents the largest value. This is the idea behind excess representation or biased representation: the bitstring with N 0's maps to the smallest value and the bitstring with N 1's maps to the largest value.

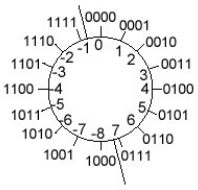
Since 3 bits allows for 8 different representations, then half that is 4. So, the smallest value is -4. When we shift by 4, this is called excess 4 representations. The following table shows 3 bits in excess 4 representation (with N bits we have excess 2^{N-1}).

Representation	Value
000	-4
001	-3
010	-2
011	-1
100	0 (4 – 4)
101	1 (5 – 4)
110	2 (6 – 4)
111	3 (7 – 4)

In this a "bias" is added to the value so that all legitimate values appear, after adjustment, as positive, unsigned, integers. This unsigned integer is used as the representation of the value. Excess representations are used mainly for the exponents of floating point numbers.

3. **Two's complement**. This is now the only way used to represent signed binary integers. To form the twos complement, take the ones complement and then add 1. Thus, if +5 is 00000101, then –5 is 11111011, and complementing –5 gives 00000101. There is only one representation for zero, but the most negative number has no complement. With 8 bits, the range is –128 <= *V* <= 127.

Sign bit → 00000101 = +5
Sign bit → 11111011 = -5



Twos complement numbers may be represented as points around a circle, shown here for 4-bit values.

Addition is performed by stepping clockwise around the circle and subtraction anticlockwise. A special axis, just anticlockwise from vertical, marks sign changes. Stepping across it near zero is a normal sign change. Stepping across remote from zero corresponds to an overflow (between -8 and +7). The symmetry of the diagram indicates why there is a -8 but not a +8.

We often say "twos (or ones) complement a value". This means "change the sign of the value according to twos (or ones) complement rules". If the original value was positive it will end up negative; if it was negative the result will be positive (usually).

1.4.2 Sign Extension

A positive or unsigned integer is always extended by prefixing it with zeros. (As in decimal, 1234 is identical to 00001234, but we usually suppress leading zeros.) Just as positive numbers extend to the left with 0 bits, so do negative numbers extend to the left with 1 bit (except for sign and magnitude). The operation of sign extension is important if we have said a signed 8-bit or 16-bit value and must extend to a 32-bit signed value. The sign bit of the old value is essentially "propagated through" the unused bits of the new value. For example, 8-bit to 16-bit extensions are

```
        0011 0101 -> 0000 0000 0011 0101
and,    1101 1001 -> 1111 1111 1101 1001
```

A longer value can be converted to a shorter value by discarding the high-order or left-hand bits, provided that the discard bits are all equal to the sign bit of the new, shorter, value.

1.4.3 Range

If we consider eight binary digits using unsigned values we can represent the range of values from 0 to 255. However if we allow the left-most bit in a binary number (the Sign bit) to represent the sign of the number (i.e. 0 means a positive number and 1 means a negative number), the other seven digits represent the magnitude of the number.

The following table shows the range of different representations.

Range	Unsigned	Sign & Magnitude	Excess (biased)	Two's Complement
255	11111111	-	-	-
254	11111110	-	-	-
…	…	-	-	-
128	10000000	-	-	-
127	01111111	01111111	11111111	01111111
126	01111110	01111110		01111110
…	…	…	…	…
0	00000000	00000000	10000000	00000000
-0	-	10000000	-	-
-1	-	10000001	01111111	11111111
…	-	…	…	…
-126	-	11111110	00000010	10000010
-127	-	11111111	00000001	10000001
-128	-	-	00000000	10000000

Summary:

- **Range**
  - Unsigned number (8-bit) : 0 <= value <=255 (2^8-1)
  - Sign and Magnitude (8-bit) : -127 <= value <=127 (2^7-1)
  - Excess (biased) (8-bit) : -128 <= value <=127 (2^7-1)
  - Two's complement (8-bit) : -128 (-2^7) <= value <=127 (2^7-1)

Exercise:

What is the range of :
a) 4-bit unsigned number?
b) 4-bit Excess (biased)?
c) 4-bit Two's complement?

1.4.4 Conversion (binary <-> decimal)

Excess (biased Representation

Excess (biased) represent numbers from -128 to 127 for an 8-bit binary number. You can convert the binary number by adding the total of all bits and subtract 2^8.

Example 11:

(1) **0**0000101 Answer = 5 – 128 = -123	**(2)** **1**0000001 Answer = 129 -128 = 1
(3) **0**0101010 Answer = 42 -128 = -86	**(4)** **1**0101010 Answer = 170-128 = 42

Suppose we have 8 bits. A positive number x is represented as 10000000 + x and a negative number –x is represented as 01111111 –x + 1.

Example 12:

(1) 30 Answer = 10000000 + 00011110 = 10011110	**(2)** -30 Answer = 01111111 – 00011110 + 1 = 01100001 + 1 = 01100010

Two's Complement Representation

First, "two's complement" is used to describe numbers from –128 to 127 for an 8-bit binary number. In this system if the value of the sign bit is:

- **Zero** – then the rest of the seven bits represent normal binary numbers, i.e. 0 to 127 are possible;
- **One** – then this represents negative number. You can calculate the value by complement the value (invert all bits) and add 1.

Example 13:

(1) **0**0000101 Sign bit = 0 => Positive number Answer = 5	**(2)** **1**0000001 Sign bit =1 => Negative number Invert all bits = 01111110 Add 1= 01111111 = 127 Answer = -127
(3) **0**0101010 Sign bit = 0 => Positive number Answer = 42	**(4)** **1**0101010 Sign bit = 1 => Negative number Invert all bits = 01010101 Add 1 = 01010110 Answer = -86

Suppose we have 8 bits. A positive number x is represented as x and a negative number –x is represented as 11111111 –x + 1.

Example 14:

(1) 30 Answer = 00011110 = 00011110	**(2)** -30 Answer = 11111111 – 00011110 + 1 = 11100001 + 1 = 11100010

Example 15:

(1) Convert -23_{10} to Binary using

- 8-bit, Excess:
  Convert 23 into Binary: 00010111
  Then invert the last seven bits, add 1: 01101000 + 1, answer = 01101001
- 8-bit, Two's complement:
  Convert 23 into Binary: 00010111
  Then invert all bits: 11101000
  Add 1: 11101001
  Therefore $-23_{10} = 11101001_2$

(2) Convert 01010101_2 to Decimal if the number is represented as

- Unsigned 8-bit binary numbers
- 01010101 = 85
- Signed 8-bit binary numbers in Excess
  01010101 -> 85 -128
  = -43
- Signed 8-bit binary numbers in two's complement
  01010101 -> Positive numbers
  = 85

(3) Convert 10101010_{10} to Decimal if the number is represented as

- Unsigned 8-bit binary numbers
- 10101010 = 170
- Signed 8-bit binary numbers in Excess
  10101010 -> 170 -128
  = 42
- Signed 8-bit binary numbers in two's complement
  10101010 -> Negative numbers
  Invert all bits = 01010101 = 85
  Add 1: 01010110 = 86
  Answer = -86

Exercises:

(1) Convert 10110011 to Decimal if the number is represented as signed 8-bit binary in Excess representation
(2) Convert 10110011 to Decimal if the number is represented as signed 8-bit binary in two's complement
(3) Convert 10110011 to Decimal if the number is represented as unsigned 8-bit binary
(4) Convert Decimal –17 to 8-bit Binary using Excess representation
(5) Convert Decimal –17 to 8-bit Binary using Two's complement representation

1.4.5 Overflow

As computers always represent integers to some small number of bits, usually 8, 16 or 32, only a limited range of values can be represented.

Example 16:
If we are using 8 bits, the ranges are from –128 to 127. If we are using 32 bits (An int is 32-bit in Java), then the ranges are from –2147483648 to 2147483647.

```
try {
  BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
  System.out.print("Enter the number in decimal:");
  String numberInStr = in.readLine();
  int num = Integer.parseInt(numberInStr);
  System.out.println();
  System.out.println("Number in Binary: " + Integer.toBinaryString(num));
  System.out.println("Number in Octal: " + Integer.toOctalString(num));
  System.out.println("Number in Hexadecimal: " + Integer.toHexString(num));
} catch (Exception e) {
  System.err.println("Error");
}
```

```
Example:
C:\>java Converting
Enter the number in decimal: 2147483647
Number in Binary 11111111111111111111111111111111
Number in Octal 17777777777
Number in Hexadecimal 7fffffff
```

```
C:\>java Converting
Enter the number in decimal: -2147483648
Number in Binary 10000000000000000000000000000000
Number in Octal 20000000000
Number in Hexadecimal 80000000
```

```
C:\>java Converting
Enter the number in decimal: 2147483648
Error //It throws an exception, since it can't convert to int.
```

If two large positive values are added the result may exceed the maximum value, leading to an ***overflow***. Overflows can also occur when adding two large negative values, but never when adding numbers of opposite sign.

Example 17:

```
   try {
      int num = 2000000000;
      System.out.println( num * 2);
      long longNum = ((long) num) * 2;
      System.out.println("Number in 32-bit: " + Integer.toBinaryString(num * 2));
      System.out.println("Number in 64-bit: " + Long.toBinaryString(longNum));
      System.out.println(longNum);
    } catch (Exception e) {
      System.err.println("Error");
    }
```

```
Answer:
-294967296
Number in 32-bit: 11101110011010110010100000000000
Number in 64-bit: 11101110011010110010100000000000
4000000000
```

An int is 32-bit, the result of n * 2 is 4, 000, 000, 000 which needs >32-bit to store the value. If we use 64 bits to store the value, then the answer is:

00000000 00000000 00000000 00000000 1110 1110 0110 1011 0010 1000 0000 0000
or 00000000EE6B2800 (hexadecimal)

However, if we use 32-bit to store the value, then the answer is

11101110 01101011 00101000 00000000 or EE6B2800

The left most bit is a "1", the sign value is negative. Then the result is equal to –294967296.

Note: Overflow conditions never throw a runtime exception; instead the sign of the result may not be the same as that expected in the mathematical result.

Example 18:
An overflow shows itself as a number of the wrong apparent sign. For example, with 4 bits and two's complement, the range is –8 <= V <= 7. Adding 6 + 7 gives

```
   0110
+  0111
   1100 carries
   1101
```

The result (13) is outside the valid range and appears as a negative sum of two positive values. Similarly, –6 + –7 gives an apparently positive result

```
  1010
+ 1001
 10000 carries
 10011 = 0011
```

Both results would be correct with **more digits available**. In general an overflow may be detected as a result of unexpected sign (+ve + +ve -> -ve, or –ve + –ve -> +ve). A better way to detect overflow is to look at the **carries into** and **out of the sign bit**. If these are not equal, there is an overflow. When adding *N*-bit twos complement numbers, always discard the carry coming out of the sign bit (truncate the result at *N* bits).

Example 19:
For example, with 4 bits and twos complement, the range is –8 <= V <= 7.

(1)

```
  0110
+ 0111
 01100 carries
 01101

Carry into the sign bit only
No carry out from the sign bit
= Invalid (Overflow) 6 + 7 = 13(>7)
```

(2)

```
  1010
+ 1001
 10000 carries
 10011

No carry into the sign bit
Carry out from sign bit only
= Invalid (Overflow) -6 + -7 = -13
```

(3)

```
  1110
+ 0011
 11100 carries
 10001       Answer: 0001

Carry into the sign bit
Carry out from sign bit
= Valid
Discard the carry coming out of the sign bit
Checking: -2 + 3 = 1
```

(4)

```
  0010
+ 0011
 00100 carries
  0101

No carry into the sign bit
No carry out from sign bit
= valid
Checking: 2 + 3 = 5
```

Exercises:

(1)
  0111
+ 0101
	(2)
	  1001
	+ 1100

(3)
  1100
+ 0111
	(4)
	  1001
	+ 0011

1.5 Subtraction by complement addition

Computers usually perform subtraction by adding the complement of the subtrahend – use X–Y = X+(–Y). To calculate 0100 1011 0110 – 0011 0111 1001 ($1{,}206_{10}$–889_{10}) using 2's complement arithmetic.

The answer has a carry out of the high order bit. Inspection shows that there is also a carry *into* the high-order bit. A correct 2's complement addition requires that the carry *into the sign bit* must be equal to the carry *out of the sign bit*.

```
   Take subtrahend         0011 0111 1001
   1's complement it             1100 1000 0110
   with a carry-in for the +1           1

                                 1100 1000 0111
   add minuend                   0100 1011 0110
   carries                     1 1001 0000 1100

   then add, to get answer  1 0001 0011 1101 = 0001 0011 1101 = 317
```

Example 20:

(1) 18 –11
18 is represented as 00010010 (8-bit), 11 is represented as 00001011
Then, one's complement of 11 is 11110100, Two's complement of 11 is 11110101
Now if we add 18 to the two's complement of 11, we get

```
  00010010
+ 11110101
 11100000 carries
 100000111
```

We got carry into and out from the sign bit, therefore the answer is valid, = 00000111 = 7_{10}

(2) 01110011 – 00001110
One's complement of 00001110 is 11110001, Two's complement of 00001110 is 11110010
Now if we add 01110011 to the two's complement of 00001110, we get

```
  01110011
+ 11110010
 11100100 carries
 101100101
```

We got carry into and out from the sign bit, therefore the answer is valid. The final answer is 01100101 = 101_{10}
Checking: 115_{10} – 14_{10} = 101_{10}

(3) 00001010 – 10000011
One's complement of 10000011 is 01111100, Two's complement of 10000011 is 01111101
Now if we add 00001010 to the two's complement of 10000011, we get

```
  01111101
+ 00001010
 01110000 carries
  10000111
```

We got carry into the sign bit only, therefore the answer is invalid.
Checking: 10_{10} – (-125_{10})= 135_{10} (overflow)

(4) 00001010 – 01010101
One's complement of 01010101 is 10101010, Two's complement of 01010101 is 10101011
Now if we add 00001010 to the two's complement of 01010101, we get

```
  10101011
+ 00001010
 00010100 carries
  10110101
```

We got no carry into and no carry out from the sign bit, therefore the answer is valid and is 10110101 = -75_{10}
Checking: 10_{10} – (85_{10})= -75_{10}

Exercises: Perform the following binary subtractions by adding the 2's complement of the subtrahend.

(1) 01101011 – 00010110

(2) 00101100 – 01101001

Summary of signed addition and subtraction:
The basic rules given for addition and subtraction really apply only to unsigned (positive) numbers, although negative values have been mentioned at times. When we have genuinely signed numbers, the rules depend on the representation. (you need to know only the part for two's complement).

- **Sign and magnitude** numbers are usually converted to one of the other representations and the result converted back if necessary. This means that S&M addition and subtraction is relatively complex and is one good reason for avoiding this representation.
- **Excess (biased) numbers** have the advantage that the ordering is the same as for unsigned numbers, so unsigned comparisons works. Excess representation is often used in the representation of the exponent for floating point numbers.
- **Two's complement** numbers are always added as unsigned values, with subtraction performed by complement addition. In other words there is virtually no special treatment needed, and that is one good reason for using two's complement.

1.6 Bitwise Logical Operations

1.6.1 Basic

Often we want to work with individual bits within a word and must use logical operations. The ones we need involve only one or two operands and is possible to write a "*truth table*" listing all possible inputs and the corresponding results.

Operators	Meaning
! (NOT)	Inverts its 1–bit argument
& (AND)	Yields 1 (TRUE) if both inputs are
| (OR)	Yields 1 if either input is true (1)
^ (EOR, exclusive or)	Yields 1 if one input = 1, but not both

NOT

X	NOT
0	1
1	0

Example 21:

(1) !0001 1111 1110 0000 (0xE0)	**(2)** !0101 0101 1010 1010 (0xAA)

AND

X	Y	AND
0	0	0
0	1	0
1	0	0
1	1	1

Example 22:

(1) 0011 1100 &0001 1111 0001 1100 (0x1c)	**(2)** 0011 1100 &0101 0101 0001 0100 (0x14)

OR

X	Y	OR
0	0	0
0	1	1
1	0	1
1	1	1

Example 23:

(1)	**(2)**
0011 1100 |0001 1111 0011 1111 (0x3f)	0011 1100 |0101 0101 0111 1101 (0x7D)

EOR

		EOR
X	Y	
0	0	0
0	1	1
1	0	1
1	1	0

Example 24:

(1)	**(2)**
0011 1100 ^0001 1111 0010 0011 (0x23)	0001 1100 ^0101 0101 0100 1001 (0x49)

Example25:

(1) 63 & 252
= 60

63 is represented as 00000000 00000000 00000000 00111111
252 is represented as 00000000 00000000 00000000 11111100
& _____
00000000 00000000 00000000 00111100 = 60

(2) 63 | 252
= 255

63 is represented as 00000000 00000000 00000000 00111111
252 is represented as 00000000 00000000 00000000 11111100
| _____
00000000 00000000 00000000 11111111 = 255

(3) 63 ^ 252
= 195

63 is represented as 00000000 00000000 00000000 00111111
252 is represented as 00000000 00000000 00000000 11111100
^ _____
00000000 00000000 00000000 11000011 = 195

Exercises:

(1)	**(2)**	**(3)**
1010 0001 &0101 1111	1010 0001 |0101 1111	1010 0001 ^0101 1111

1.6.2 Masking

Masking is another low level operation on bits. Masking essentially wipes certain bits of a value. For example, let's take the value 10101010 and mask the first 4 bits. To do this we use the Bitwise AND operator "&" with the value 00001111. What this operator is doing is every place that both values (the mask and the value we are masking) are 1, it keeps the one. Where either of the values are 0, it keeps 0. So if we perform the mask we just described (10101010 & 00001111) we will get the value 00001010.

Therefore, masking is normally used to set or extract desired bits for a variable or expression.

- use | with mask to set bits

- use & with mask to extract bits

Example 26:

(1)	**(2)**
1010 1010 &0000 **1111** 0000 1010 get the lower 4 bits	1010 1010 |0000 **1111** 1010 1111 Set all the lower 4 bits to **1**

1.7 Shift Operations

1.7.1 << (left shift)

- Bits are shifted to the left based on the value of the right operand.
- New right hand bits are zero filled.
- Equivalent to left-operand times two to the power of the right-operand if no overflow occurs.

Example 27:

(1) 8-bit binary signed number:	**(2)** 8-bit binary signed number:
00000101 << 3 = 00101**000** Note: 5 * 2^3 = 40	00010101 << 2 = 01010100 Note: 21 * 2^2 = 84
(3) 8-bit binary signed number: 10100110 (-90_{10}) << 3 = 00110000 (48_{10}) Note: overflow	**(4)** 8-bit binary signed number: 11000101 (-59_{10})<< 1 = 10001010 (-118_{10}) Note: -59 * 2^1 = -118
(5) 32-bit binary signed number: int x = 61 << 2; 61_{10} => 00000000000000000000000000111101$_2$ Answer: 00000000000000000000000011110100$_2$ = 244_{10}	

Exercises:

(1) 8-bit binary signed number: 00001100 << 3	**(2)** 32-bit binary signed number: int x = 31 << 2;

1.7.2 >> Right Shift with Sign Fill

- Bits are shifted to the right based on the value of the right operand.
- New left hand bits are filled with the value of the left-operand high order bit; therefore, the sign of the left-hand operator is always retained.
- For non-negative integers, a right-shift is equivalent to dividing the left-hand operator by two to the power of the right-hand operator.

Example 28:

(1) 8-bit binary signed number:	**(2)** 8-bit binary signed number:
00010110 >> 2 = 00000101 Note: 22 / 2^2 = 5	01010101 >> 1 = 00101010 Note: 85 / 2^1 = 42
(3) 8-bit binary signed number: 10010011 (-109_{10}) >> 2 = 11100100 (-28_{10})	**(4)** 8-bit binary signed number: 10101010 (-86_{10}) >> 1 = 11010101 (-43_{10})
(5) 32-bit binary signed number: int x = 61 >> 2; 61_{10} => 00000000000000000000000000111101$_2$ Answer: 00000000000000000000000000001111$_2$ = 15_{10}	

(6) 32-bit binary signed number:

```
int x = -2147483646 >> 2;
-2147483646₁₀ =>     10000000000000000000000000000010₂
Answer:              11100000000000000000000000000000₂
=-536870912₁₀
```

Exercises:

(1) 8-bit binary signed number: 10100001 >> 3	**(2)** 32-bit binary signed number: int x = 21 >> 2;

1.7.3 >>> Right Shift with Zero Fill

- Identical to the right-shift operator, only the left bits are zero filled.
- Because the left-operand high-order bit is not retained, the sign value can change.
- If the left-hand operand is positive, the result is the same as a right-shift with sign fill.

Example 29:

(1) 8-bit binary signed number: 00010110 >>> 2 = 00000101 Note: 22 / 2^2 = 5	**(2)** 8-bit binary signed number: 01010101 >>> 1 = 00101010 Note: 85 / 2^1 = 42
(3) 8-bit binary signed number: 10010011 (-109_{10}) >>> 2 = 00100100 (36_{10})	**(4)** 8-bit binary signed number: 10101010 (-86_{10}) >>> 1 = 01010101 (85_{10})
(5) 32-bit binary signed number: int x = 61 >>> 2; 61_{10} => 00000000000000000000000000111101₂ Answer: 00000000000000000000000000001111₂ = 15_{10}	
(6) 32-bit binary signed number: int x = -2147483646 >>> 2; -2147483646₁₀ => 10000000000000000000000000000010₂ Answer: 00100000000000000000000000000000₂ =536870912₁₀	

Exercises:

(1) 8-bit binary signed number: 10100001 >>> 3	**(2)** 32-bit binary signed number: int x = -1 >>> 2;(answer in Hex)

NOTE: Shifting is much faster than actual multiplication (*) or division (/) by 2. So if you want fast multiplications or division by 2 use shifts.

1.7.4 Applet:

1.7.5 Java Example:

Example 30:

```
public class BitOperators {
  public static void main(String[] args) {
    int x = 61, y = 31;
    System.out.println("x="+x);
    System.out.println("y="+y);
    System.out.println("x="+Integer.toBinaryString(x));
    System.out.println("y="+Integer.toBinaryString(y));
    System.out.println("x & y=" + (x & y));
    System.out.println("x | y=" + (x | y));
    System.out.println("x << 2=" + (x << 2));
    System.out.println("x >> 2=" + (x  >> 2));
    System.out.println("x >>> 2=" + (x >>> 2));
    System.out.println("x ^ y=" + (x ^ y));
  }
}
```

```
Answer:
x=61
y=31
x=111101
y=11111
x & y=29
x | y=63
x << 2=244
x >> 2=15
x >>> 2=15
x ^ y=34
```

Exercise:

What is the output if x = 21, y = 56?