UNIVERSITY OF BIRMINGHAM

Computer Systems
Week 1 - Numbers

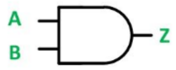
Lecture Objectives

To introduce the fundamentals of number representations, their conversion and how overflow can create problems in computer programs.

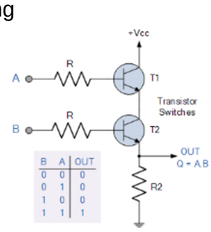
Numbers – Lecture Outline

- Why do we use binary?
- Representing Text
- Decimal, Binary, Octal and Hexadecimal Notations
- How to convert from Decimal to Binary & vice versa?
- Conversion between other notations?
- Overflow - and avoiding it
- Signed Integers - 2's complement
- Illustration of Overflow using Factorial

What are Computers made of?



- A collection of "switches" called transistors.
- Can either be "on" or "off", corresponding to a particular electrical state (conducting or not).
- Forces "binary" representation
  - "Off" → 0
  - "On" → 1
- All data must be represented as sequences of ones and zeros – binary digits – **bits**
  - As must the computer's "instructions"



Implications?

- For practical engineering reasons it is better (cheaper, more reliable, faster …) to stick with basic components that only have two states – binary
  - But, in principle, we could build other types of computer – and they have been built
- This applies within the processor, memory and all other components
- As an aside, all the data is just a pattern of bits. The interpretation put on it is down to the program:
  - You can add two character strings or instructions
  - But, there are safeguards in most operating systems and programming languages that restrict this.
- Historically, memory and processing was very limited.
  - It still is, but not to the same extent.

Representing Text

- Every text character is represented as a "binary string"
  - A: 1000001
  - B: 1000010
  - C: 1000011
- Text Encoding Schemes like ASCII, Unicode
  - These usually use 8, 16 or 32 bits to encode characters
  - Support internationalisation
    - https://en.wikipedia.org/wiki/Character_encoding
- Strings (words, names, addresses etc.) are usually represented by a block of characters. See how Java/Python does it.
- Numbers are more subtle
  - Need to worry about arithmetic

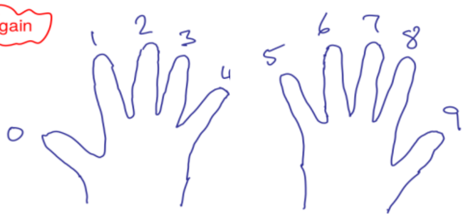
Decimal Notation (Base = 10)

The key thing is that we need to do arithmetic

Latin: by tens

Digit = finger

Latin again



Different Notations

System	Base	Symbols	Used by humans?	Used in computers?
Decimal	10	0, 1, … 9	Yes	No
Binary	2	0, 1	No	Yes
Octal	8	0, 1, … 7	No	No
Hexa-decimal	16	0, 1, … 9, A, B, … F	No	No

Counting in Different Number Systems (1 of 3)

Decimal	Binary	Octal	Hexa-decimal
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7

Counting in Different Number Systems (2 of 3)

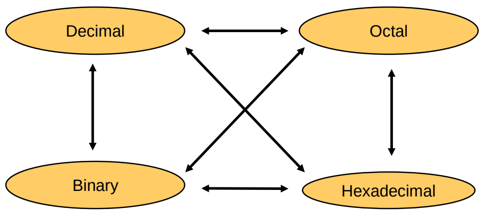
Decimal	Binary	Octal	Hexa-decimal
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Counting in Different Number Systems (3 of 3)

Decimal	Binary	Octal	Hexa-decimal
16	10000	20	10
17	10001	21	11
18	10010	22	12
19	10011	23	13
20	10100	24	14
21	10101	25	15
22	10110	26	16
23	10111	27	17

Etc.

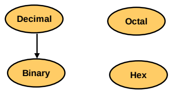
Conversion among Different Notations



$$25_{10} = 11001_2 = 31_8 = 19_{16}$$

Base

Conversion – Decimal to Binary



- Technique
  - Divide by two, keep track of the remainder
  - First remainder is bit 0 (LSB, Least-Significant Bit)
  - Second remainder is bit 1
  - Etc.

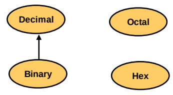
Example

$125_{10} = ?_2$

2	125	
2	62	1
2	31	0
2	15	1
2	7	1
2	3	1
2	1	1
	0	1

$125_{10} = 1111101_2$

Conversion – Binary to Decimal



- Technique
  - Multiply each bit by 2^n, where n is the "weight" ("positional value") of the bit.
  - The weight is the position of the bit, starting from 0 on the right
  - Add the results

Example

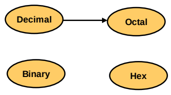
101011_2 =>

Bit "0"

$1 \times 2^0 = 1$
$1 \times 2^1 = 2$
$0 \times 2^2 = 0$
$1 \times 2^3 = 8$
$0 \times 2^4 = 0$
$1 \times 2^5 = 32$

43_{10}

Conversion – Decimal to Octal



- Technique
  - Divide by 8
  - Keep track of the remainder

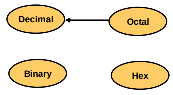
Example

$1234_{10} = ?_8$

8	1234	
8	154	2
8	19	2
8	2	3
	0	2

$1234_{10} = 2322_8$

Conversion – Octal to Decimal



- Technique
  - Multiply each digit by 8^n, where n is the "weight" of the digit
  - The weight is the position of the digit, starting from 0 on the right
  - Add the results

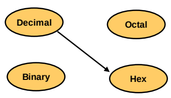
Example

724_8 =>

$4 \times 8^0 = 4$
$2 \times 8^1 = 16$
$7 \times 8^2 = 448$

468_{10}

Conversion – Decimal to Hexadecimal



- Technique
  - Divide by 16
  - Keep track of the remainder

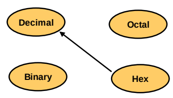
Example

$1234_{10} = ?_{16}$

16	1234	
16	77	2
16	4	13 = D
	0	4

$1234_{10} = 4D2_{16}$

Conversion – Hexadecimal to Decimal



- Technique
  - Multiply each digit by 16^n, where n is the "weight" of the digit
  - The weight is the position of the digit, starting from 0 on the right
  - Add the results

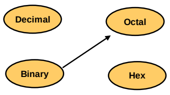
Example

ABC_{16} =>

$C \times 16^0 = 12 \times 1 = 12$
$B \times 16^1 = 11 \times 16 = 176$
$A \times 16^2 = 10 \times 256 = 2560$

2748_{10}

Conversion – Binary to Octal



- Technique
  - Group bits in threes, starting on right
  - Convert to octal digits

Example

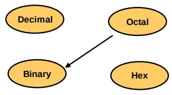
$1011010111_2 = ?_8$

1 011 010 111

1 3 2 7

$1011010111_2 = 1327_8$

Conversion – Octal to Binary



- Technique
  - Convert each octal digit to a 3-bit equivalent binary representation
- **Note**: Every digit will be converted to 3 bits, even if all bits are zeros

Example

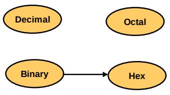
$705_8 = ?_2$

7 0 5

111 000 101

$705_8 = 111000101_2$

Conversion – Binary to Hexadecimal



- Technique
  - Group bits in fours, starting on right
  - Convert to hexadecimal digits

Example

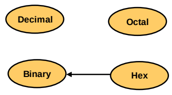
$1010111011_2 = ?_{16}$

10 1011 1011

2 B B

$1010111011_2 = 2BB_{16}$

Conversion – Hexadecimal to Binary



- Technique
  - Convert each hexadecimal digit to a 4-bit equivalent binary representation
- **Note**: Every digit will be converted to 4 bits, even if all bits are zeros

Example

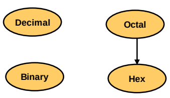
$10AF_{16} = ?_2$

1 0 A F

0001 0000 1010 1111

$10AF_{16} = 0001000010101111_2$

Conversion – Octal to Hexadecimal



- Technique
  - Use binary notation as an intermediate representation.

Example

$1076_8 = ?_{16}$

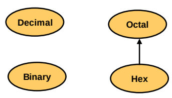
1 0 7 6

001 000 111 110

2 3 E

$1076_8 = 23E_{16}$

Conversion – Hexadecimal to Octal



- Technique
  - Use binary notation as an intermediate representation.

Example

$1F0C_{16} = ?_8$

1 F 0 C

0001 1111 0000 1100

1 7 4 1 4

$1F0C_{16} = 17414_8$

Exercise – Conversions (Home Work)

Decimal	Binary	Octal	Hexa-decimal
33			
	1110101		
		703	
			1AF

Don't use a calculator!

Exercise – Conversions (Answers)

Decimal	Binary	Octal	Hexa-decimal
33	100001	41	21
117	1110101	165	75
451	111000011	703	1C3
431	110101111	657	1AF

Bits and Numbers (Unsigned)

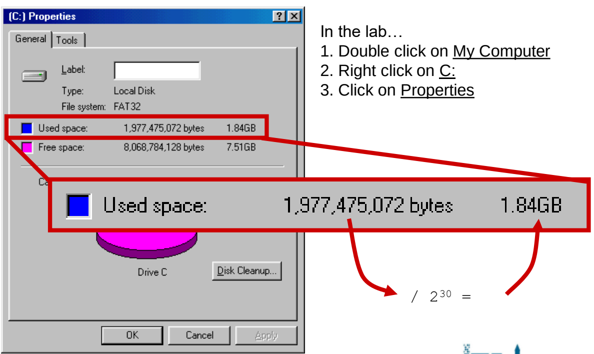
No of Bits	Min (Binary)	Max (Binary)	Max (Decimal)
1	0	1	1
2	00	11	3
3	000	111	7
4	0000	1111	15
8	00000000	11111111	255
n	0…0 (n bits)	1…1 (n bits)	**2^n - 1**

Approximating Big Powers of 2

Binary Value	Value (Exact)	Value (Approx)	Equivalent (Decimal)
2^{10}	1024	1000 (one thousand)	10^3 (Kilo k)
2^{20}	1048576	1000000 (one million)	10^6 (Mega M)
2^{30}	1073741824	1000000000 (one billion)	10^9 (Giga G)
2^{40}	1099511628000	1000000000000 (one trillion)	10^{12} (Tera T)

- What is the value of "K", "M", and "G"?
- In computing, particularly w.r.t. **memory**, the base-2 interpretation generally applies (Column 2)

Example – Memory Space



In the lab…
1. Double click on My Computer
2. Right click on C:
3. Click on Properties

What is Overflow?

- What is the highest value that can be stored in 8-bits? And what happens when arithmetic goes outside the storable range?
- Example: 255 + 1 in 8 bits ?

1111 1111	=	255
0000 0001	=	1
1 0000 0000		0

Overflow bit gets lost

Answer seems like Zero

What is Overflow?

- In a given type of computer, the size of integers is a fixed number of bits.
- 32 or 64 bits are popular choices these days.
- It is possible that addition of two n bit numbers yields a result requiring n+1 bits.
- Overflow is the term for an operation whose results exceed the space allocated for a number.

Negative Numbers

- In 8-bit arithmetic 255 + 1 looks like 0, 255 is behaving like -1
- 11111111 is the two's complement representation of -1
- More generally, for 2's complement representation (in 8 bits) of a -ve number -X; we can use ordinary binary representation of 256 – X
- Even more generally, the 2's complement of -X, we can use the following formula: $\mathbf{2^n - X}$
  - Where n is the number of bits

Finding 2's Complement – General Method

- Flip all of the bits
- Add 1

256 - X	=	255 – X + 1 (still 8 bits)
255 - X	=	Flip All Bits of X

- **Example**: How to represent -20 in 2's complement?

20 in binary	00010100
Flip All Bits	11101011
Add 1	11101100

Works in any number of bits

8-bit Signed Integers

- Integers in the range -128 … +127

127	01111111
...	. . .
1	00000001
0	00000000
-1	11111111
-2	11111110
...	...
-128	10000000

2's complement

MSB shows sign:
0 for non-negative (zero or positive)
1 for negative numbers

Range for n-bit Signed Integers

In general for signed integers: $\mathbf{-2^{n-1} \ldots 2^{n-1}-1}$

- For Example:

Bits	Minimum	Maximum
8	-128	+127
16	-32768	+32767
32	-2147483648	+2147483647

For un-signed integers: $\mathbf{0 \ldots 2^n-1}$

Bits	Minimum	Maximum
8	0	255
16	0	65535
32	0	4294967295

Binary Numbers: How they should be treated?

- For much of the arithmetic (+, -, *) it does not matter
- For comparisons, need to know whether the numbers are signed or unsigned:

For Example:

- signed -1 < 0 TRUE
- unsigned 255 < 0 FALSE

Integer Numbers – Behavior in Java

- For integer (whole number) arithmetic:
  - values are treated as signed
  - bits beyond storable range are lost
  - overflow ignored - no errors flagged!

Data Types	byte	short	int	long
Bytes	1	2	4	8
Bits	8	16	32	64

A boolean value (true or false) is 1 bit (1 or 0)

Integer Numbers – Sign Extension

- Sometimes you extend a number by giving it more space.
  - For unsigned arithmetic: just provide extra 0s
  - e.g. extending 8 to 16
  - unsigned: provide 0s

11111111

00000000 11111111

- For signed arithmetic: use sign extension
  - MSB is repeated

11111111

11111111 11111111

- Java uses sign extension when converting byte to short (and short to int, int to long)

Selecting Appropriate Data Types

Make sure your datatype is big enough for the numbers you want to compute with!

e.g. int for money in pence may be too short

max int = $2^{31} – 1$
$= 2 \times (2^{10})^3 – 1$
$\approx 2 \times 10^9$ pence
$= £\ 2 \times 10^7 = £\ 20$ million

int = 4 bytes = 32 bits
unsigned, max = $2^{32} – 1$
but int is signed

$2^{10} = 1024 \approx 10^3$

If amounts go above that, can use long - or BigInteger

How quickly overflow can occur?

n factorial (n!) = n * (n – 1) * (n – 2) * … * 1

```
public static void main(String[] args) {
        for (int i = 6; i < 20; i++){
        System.out.println(""+i+", "+fact(i));
        }
}
/**
* Calculate factorial.
* requires: 0 <= n
* @param n number whose factorial is to be calculated
* @return factorial of n
*/
public static int fact(int n){
        int a = 1;
        for (int i = 1; i <= n; i++){
                a = a * i;
        }
        return a;
}
```

"requires" comment says
nothing is guaranteed if n < 0

Do you see any problems?

n, n!
6, 720
7, 5040
8, 40320
9, 362880
10, 3628800
11, 39916800
12, 479001600
13, 1932053504
14, 1278945280
15, 2004310016
16, 2004189184
17, -288522240
18, -898433024
19, 109641728

How quickly overflow can occur?

n, n!
6, 720
7, 5040
8, 40320
9, 362880
10, 3628800
11, 39916800
12, 479001600
13, 1932053504
14, 1278945280
15, 2004310016
16, 2004189184
17, -288522240
18, -898433024
19, 109641728

Obviously wrong for n = 17 or 18 - negative answers

Actually, all answers from n = 13 must be wrong
n! = n * (n-1) * ... * 10 * ... * 5 * ... * 2 * 1

10x5x2 = 100
- correct answer must end 00 for n = 10 or more

At n = 13 get overflow.
After that all the answers are wrong.
A rough calculation explains why n = 13 is where it goes wrong.

12! = 479,001,600 ≈ 4.79 x 10^8
13! = 6,227,020,800 ≈ 6.23 x 10^9
Max int = 2^{31} – 1 = 2,147,483,647 ≈ 2.15 x 10^9
13! is too big for int (signed 32-bit integer)

A Slightly Better Version!

```
public static void main(String[] args) {
        for (int i = 6; i < 20; i++){
        System.out.println(""+i+", "+fact(i));
        }
}
/**
* Calculate factorial.
* requires: 0 <= n <= 12
* @param n number whose factorial is to be calculated
* @return factorial of n
*/
public static int fact(int n){
        int a = 1;
        for (int i = 1; i <= n; i++){
                a = a * i;
        }
        return a;
}
```

Alternatively:

Use long (but still overflow)

Use BigInteger

(or Python)

Long Version!

```
/**
* Calculate factorial.
* requires: 0 <= n <= 20
* @param n number whose factorial is to be calculated
* @return factorial of n
*/
public static long lfact(int n){
        long a = 1;
        for (int i = 1; i <= n; i++){
                a = a * i;
        }
        return a;
}
```

Works up to n = 20

n, n!
6, 720
7, 5040
8, 40320
9, 362880
10, 3628800
11, 39916800
12, 479001600
13, 6227020800
14, 87178291200
15, 1307674368000
16, 20922789888000
17, 355687428096000
18, 6402373705728000
19, 121645100408832000
20, 2432902008176640000
21, -4249290049419214848

Summary

Representing numbers in the computer

- Whole numbers in binary, octal, decimal and hexadecimal notations
- Converting from one representation to another
- Representing negative numbers
- Overflow and its consequences.