COMP 10280
Programming I (Conversion)

John Dunnion

School of Computer Science
University College Dublin

COMP 10280 Programming I (Conversion)/Lecture 19

Outline

Representing Numbers

- We have seen previously how to represent numbers
- In everyday life, we normally (but not always!) use denary or decimal (Base 10) digits to represent numbers: 0–9
- In computer systems, we use binary (Base 2) digits to represent numbers: 0 and 1
- For convenience, humans often interpret a binary number as an octal (Base 8) or hexadecimal (Base 16) number
- To regard a binary number in octal, we take the number in groups of three bits
- To regard a binary number in hexadecimal, we take the number in groups of four bits

Representing decimal integers (1)

- In everyday life, we use a positional number system
- In a positional number system, the value of a digit is determined by its position in the number
- Consider the number 4592:
  - The 4 is worth 4000 (it appears in the 1000s position)
  - The 5 is worth 500 (it appears in the 100s position)
  - The 9 is worth 90 (it appears in the 10s position)
  - The 2 is worth 2 (it appears in the 1s (units) position)
- Formally, the digits in a decimal number are weighted by increasing powers of 10: they use Base 10
- Thus 4592 can be written as

$$4 \times 10^3 + 5 \times 10^2 + 9 \times 10^1 + 2 \times 10^0$$

- (Recall that $10^0 = 1$)

Representing decimal integers (2)

- The Least Significant Digit is the rightmost one
- This is the 2 in 4592
- It has the lowest power of 10 weighting
- Digits towards the right-hand side are called the “low-order digits” (lower powers of 10)
- The Most Significant Digit is the leftmost one
- This is the 4 in 4592
- It has the highest power of 10 weighting
- Digits towards the left-hand side are called the “high-order digits” (higher powers of 10)

The largest *n*-digit decimal integer

- What is the largest *n*-digit number?
- It is made up of *n* successive 9s
- The largest four-digit decimal number is 9999
- 9999 is $10^4 - 1$

Representing binary integers (1)

- Binary numbers are numbers in Base 2
- There are only two digits: 0 and 1
- Binary numbers also use a positional number system
- Formally, the digits in a decimal number are weighted by increasing powers of 2: they use Base 2
- Consider the number 1011
- 1011 can be written as

$$1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$

- (Recall that $2^0 = 1$)
- In the number 1011:
  - The first 1 is worth 2^3 (or 8) (it appears in the 2^3 position)
  - The second 1 is worth 2^1 (or 2) (it appears in the 2^1 position)
  - The third 1 is worth 2^0 (or 1) (it appears in the 2^0 position)

Representing binary integers (2)

- The Least Significant Digit is the rightmost one
- In binary numbers, we refer to the Least Significant Bit: “bit” = “binary digit”
- This is the rightmost 1 in 1011
- It has the lowest power of 2 weighting
- Bits towards the right-hand side are called the “low-order bits” (lower powers of 2)
- The Most Significant Bit is the leftmost one
- This is the leftmost 1 in 1011
- It has the highest power of 2 weighting
- Bits towards the left-hand side are called the “high-order bits” (higher powers of 2)

The largest *n*-digit binary integer

- What is the largest *n*-digit binary number?
- It is made up of *n* successive 1s
- The largest four-digit number is 1111
- 1111 is $2^4 - 1$

Converting from binary to decimal

- Convert the following binary numbers to decimal:
- 1011
- 1011 = $1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 8 + 2 + 1 = 11$
- 0111 1111
- 0111 1111 = $0 \times 2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 = 0 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 127$
- 1000 1000
- 1000 1000 = $2^7 + 2^3 = 128 + 8 = 136$

Converting from decimal to binary (1)

- To convert a number from decimal to another base, we successively divide the number by the new base
- The remainder after each division becomes a digit in the new base until the result of the division is 0
- The result is read upwards
- Example: Convert 39 to binary

Division	Quotient	Remainder
39 / 2	19	1
19 / 2	9	1
9 / 2	4	1
4 / 2	2	0
2 / 2	1	0
1 / 2	0	1
0 / 2	STOP	

- Thus $39_{10} = 100111_2$

Converting from decimal to binary (2)

- Example: Convert 4592 to binary

Division	Quotient	Remainder
4592 / 2	2296	0
2296 / 2	1148	0
1148 / 2	574	0
574 / 2	287	0
287 / 2	143	1
143 / 2	71	1
71 / 2	35	1
35 / 2	17	1
17 / 2	8	1
8 / 2	4	0
4 / 2	2	0
2 / 2	1	0
1 / 2	0	1
0 / 2	STOP	

- Thus $4592_{10} = 1\ 0001\ 1111\ 0000_2$

Representing octal integers (1)

- Octal numbers are numbers in Base 8
- There are eight digits: 0–7
- Octal numbers also use a positional number system
- Formally, the digits in an octal number are weighted by increasing powers of 8: they use Base 8
- Consider the number 573_8
- 573_8 can be written as

$$5 \times 8^2 + 7 \times 8^1 + 3 \times 8^0 = 379_{10}$$

- (Recall that $8^0 = 1$)
- In the number 573_8:
  - The 5 is worth 5×8^2 (or 320) (it appears in the 8^2 position)
  - The 7 is worth 7×8^1 (or 56) (it appears in the 8^1 position)
  - The 3 is worth 3×8^0 (or 3) (it appears in the 8^0 position)

Converting from decimal to octal (1)

- Example: Convert 379 to octal

Division	Quotient	Remainder
379 / 8	47	3
47 / 8	5	7
5 / 8	0	5
0 / 2	STOP	

- Thus $379_{10} = 573_8$

Converting from decimal to octal (2)

- Example: Convert 4592 to octal

Division	Quotient	Remainder
4592 / 8	574	0
574 / 8	71	6
71 / 8	8	7
8 / 8	1	0
1 / 8	0	1
0 / 8	STOP	

- Thus $4592_{10} = 10760_8$

Interpreting a binary number as octal

- We can interpret a binary number as an octal number by taking the binary digits in groups of three from the right to left (lowest-significant digit to highest-significant digit)
- $4592_{10} = 1\ 0001\ 1111\ 0000_2$
- Re-arranging the binary number in groups of three bits:
- 1 000 111 110 000
- Convert each group of three bits into an octal digit:
- 1 0 7 6 0
- 10760_8

Representing hexadecimal integers (1)

- Hexadecimal numbers are numbers in Base 16
- There are 16 digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F
- A represents 10, B represents 11, . . . , F represents 15
- Hexadecimal numbers use a positional number system
- Formally, the digits in a hexadecimal number are weighted by increasing powers of 16: they use Base 16
- Consider the number $9B3_{16}$
- $9B3_{16}$ can be written as

$$9 \times 16^2 + 11 \times 16^1 + 3 \times 16^0 = 2304 + 176 + 3 = 2483_{10}$$

- In the number $9B3_8$:
    - The 9 is worth 9×16^2 (or 2304) (it appears in the 16^2 position)
    - The B is worth 11×16^1 (or 176) (it appears in the 16^1 position)
    - The 3 is worth 3×16^0 (or 3) (it appears in the 16^0 position)

Converting from decimal to hexadecimal (1)

- Example: Convert 379 to hexadecimal

Division	Quotient	Remainder
2483 / 16	155	3
155 / 16	9	11
9 / 16	0	9
0 / 16	STOP	

- Thus $2483_{10} = 9B3_{16}$

Converting from decimal to hexadecimal (2)

- Example: Convert 4592 to hexadecimal

Division	Quotient	Remainder
4592 / 16	287	0
287 / 16	17	15
17 / 16	1	1
1 / 16	0	1
0 / 16	STOP	

- Thus $4592_{10} = 11F0_{16}$

Interpreting a binary number as hexadecimal

- We can interpret a binary number as a hexadecimal number by taking the binary digits in groups of four from the right to left (lowest-significant digit to highest-significant digit)
- $4592_{10} = 1\ 0001\ 1111\ 0000_2$
- Convert each group of four bits into a hexadecimal digit:
- 1 1 F 0
- $11F0_{16}$

Signed numbers

- Humans use a symbol, + or - to represent whether a number is positive (> 0) or negative (< 0)
- Computers don't have such symbols
- Computers have to use a 1 or a 0 to represent sign
- There are three common techniques for representing signed numbers:
    - Signed magnitude
    - One's complement
    - Two's complement

Signed magnitude representation

- Sometimes called sign-magnitude or sign and magnitude representation
- In signed magnitude representation, we designate the leftmost bit (most significant bit) as a sign bit
- The sign bit indicates whether a number is positive or negative:
    - 0 sign bit: positive number
    - 1 sign bit: negative number
- The remaining bits give the magnitude of the number
- +13 would be represented as 0000 1101
- -13 would be represented as 1000 1101

Issues with signed magnitude representation

- The range of signed magnitude numbers is -2^{n-1} to $+2^{n-1}$
- Advantages:
  - Conceptually simple
  - Symmetric range
- Disadvantages:
  - Difficult to do arithmetic with negative values
  - Two representations of zero
  - +0: 0000 0000
  - -0: 1000 0000

One’s complement representation

- In one’s complement representation, a negative number is represented by applying the bitwise NOT operator to a number
- “Flip the bits”
- Example:
- +13 would be represented as 0000 1101
- -13 would be represented as 1111 0010

Representing numbers in one's complement representation

Number	Bits
+0	0000 0000
+1	0000 0001
+2	0000 0010
+126	0111 1110
+127	0111 1111
-0	1111 1111
-1	1111 1110
-2	1111 1101
-126	1000 0001
-127	1000 0000

Issues with one's complement representation

- The range of signed magnitude numbers is $-2^{n-1} - 1$ to $+2^{n-1} - 1$
- The most significant digit acts as a sign bit
- The most significant digit represents $-2^{n-1} - 1$
- Advantages:
  - Easier to do arithmetic with negative values
  - Symmetric range
- Disadvantages:
  - More difficult for humans to understand
  - Two representations of zero
  - +0: 0000 0000
  - -0: 1111 1111

Two's complement representation

- In two's complement representation, a negative number is represented by applying the bitwise NOT operator to a number and adding one
- "Flip the bits and add one"
- Examples:
- +13 would be represented as 0000 1101
- -13 would be represented as 1111 0011
- +30 would be represented as 0001 1110
- -30 would be represented as 1110 0010

Representing numbers in two's complement representation

Number	Bits
+0	0000 0000
+1	0000 0001
+2	0000 0010
+126	0111 1110
+127	0111 1111
-1	1111 1111
-2	1111 1110
-3	1111 1101
-127	1000 0001
-128	1000 0000

Issues with two's complement representation

- The range of two's complement numbers is -2^{n-1} to $+2^{n-1} - 1$
- The most significant digit acts as a sign bit
- The most significant digit represents -2^{n-1}
- Advantages:
  - Arithmetic with negative values is identical to arithmetic with positive values
  - Single representation of zero
- Disadvantages:
  - More difficult for humans to understand
  - Asymmetric range

Floating-point numbers

- A representation of real numbers
- An approximation
- A trade-off between precision (accuracy) and range
- In general, a real number is represented approximately by a floating-point number:
  - A fixed number of significant digits: the mantissa (or the coefficient, or the significand, according to the IEEE Standard)
  - The exponent (the scale that is applied to the mantissa)

Representing floating-point numbers (1)

- Consider the number 123.456
- This can be represented by 123456×10^{-3}
- It can also be represented by $1.23456 \times 10^{+2}$
- This is the normalised form. In Base 2, the 1.23456 is called a normalised significand
- It can also be represented by $0.123456 \times 10^{+3}$
- In Base 2, the .123456 is called a normed significand

Representing floating-point numbers (2)

- What do we need to store?
- The normalised/normed significand
- The exponent
- The sign of the number
- We do not need to store:
- The position of the decimal/binary point
- The base of the exponent

Single-precision floating-point

- Usually used to represent the `float` type in the C language family
- 32 bits (Four 8-bit bytes)
- One sign bit
- 23 bits for the significand (about seven decimal digits)
- 7 bits for the exponent
- Largest number: 3.4×10^{38}
- Smallest number: 1.2×10^{-38}

Double-precision floating-point

- Usually used to represent the `double` type in the C language family
- 64 bits (eight 8-bit bytes)
- One sign bit
- 52 bits for the significand (about 16 decimal digits)
- 11 bits for the exponent
- Largest number: 1.8×10^{308}
- Smallest number: 5.0×10^{-324}

Issues with floating-point representation

- Only an approximation
- Often (usually?) converting from decimal to binary
- Overflow: The number is too large to be represented in the exponent field
- Underflow: The number is too small to be represented in the exponent field
- To reduce the chances of underflow/overflow, we can use 64-bit double-precision representation
- Two representations of zero. These must be considered to be equal when compared
- There are special positive and negative infinity values, `+INF` and `-INF`
- There are special Not a Number values (`NaN`). These represent the result of various undefined calculations (eg multiplying 0 and infinity)

Floating-point program example (1)

- Consider the following Python 2.x program:

```
# Showing the imprecision of floating-point arit

x = 0.0
for i in range(10):
    x += 0.1
    print 'x is', x

if x == 1.0:
    print x, 'is equal to 1.0.'
else:
    print x, 'is not equal to 1.0.'
```

Floating-point program example (2)

- This produces the following output:

```
x is 0.1
x is 0.2
x is 0.3
x is 0.4
x is 0.5
x is 0.6
x is 0.7
x is 0.8
x is 0.9
x is 1.0
1.0 is not equal to 1.0.
```

Floating-point program example (3)

- Here is the equivalent Python 3 program:

```
# Showing the imprecision of floating-point arit

x = 0.0
for i in range(10):
    x += 0.1
    print('x is', x)

if x == 1.0:
    print(x, 'is equal to 1.0.')
else:
    print(x, 'is not equal to 1.0.')
```

Floating-point program example (4)

- This produces the following output:

```
x is 0.1
x is 0.2
x is 0.30000000000000004
x is 0.4
x is 0.5
x is 0.6
x is 0.7
x is 0.7999999999999999
x is 0.8999999999999999
x is 0.9999999999999999
0.9999999999999999 is not equal to 1.0.
```