“Module 1: Assignment 2 - Introduction to Information Technology and System Architecture”

Harsh Siddhapura

“Department of Information Technology, Arizona State University”

“IFT 510: Principles of Computer Information and Technology”

“Dr. Dinesh Sthapit”

“August 26, 2023”

"Chapter 2: Foundations of Computer Systems and Data Representation, concept of data representation and its significance in computer systems. Provide examples of why data representation is necessary"

1. **"Perform conversions between binary, hexadecimal, and octal number systems. Show step-by-step calculations and provide examples."**

   The conversion between binary, hexadecimal, and octal number system is as follows:

- **Binary to Hexadecimal Conversion:**

  To convert a binary number to hexadecimal, divide the binary digits into four groups of four, beginning on the right. Then, for each group, get the hexadecimal equivalent.

  Example: Convert the binary number 110101101011 to hexadecimal.

  ➔ Step 1: Group binary digits into sets of four:

  1101 0110 1011

  ➔ Step 2: Convert each group to its hexadecimal equivalent:

  D 6 B

  Result: The binary number 110101101011 is equivalent to the hexadecimal number D6B.

- **Binary to Octal Conversion:**

  To convert a binary number to an octal number, organise the binary digits into groups of three, beginning on the right. Then, for each group, get the octal equivalent.

  Example: Convert the binary number 101011110101 to octal.

  ➔ Step 1: Group binary digits into sets of three:

  101 011 110 101

➔ Step 2: Convert each group to its octal equivalent:

5 3 6 5

Result: The binary number 101011110101 is equivalent to the octal number 5365.

- **Hexadecimal to Binary Conversion:**

To convert a hexadecimal number to binary, replace each hexadecimal digit with its four-digit binary equivalent.

Example: Convert the hexadecimal number 2F6 to binary.

Hexadecimal to Binary Conversion Table:

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

- ➔ Step 1: Replace each hexadecimal digit with its binary equivalent:

2	F	6
0010	1111	0110

Result: The hexadecimal number 2F6 is equivalent to the binary number 001011110110.

- **Hexadecimal to Octal Conversion:**

Because hexadecimal numbers include binary digits, we may group these binary numbers into a pair and compare them to octal numbers. Here are the steps:

- ➔ Step 1: For each given hexadecimal number digit, write the equivalent binary number. If any of the binary equivalents are less than 4 digits, add 0's to the left side.
- ➔ Step 2: Combine and make the groups of binary digits from right to left, each containing 3 digits. Add 0's to the left if there are less than 3 digits in the last group.
- ➔ Step 3: Find the octal equivalent of each binary group.

Example: Convert (1BD)16 into an octal number.

Solution: Given, (1BD)16 is a hexadecimal number.

1 → 0001, B → 1011, D →1101

Now group them from right to left, each having 3 digits.

000, 110, 111, 101

000→0, 110 →6, 111→7, 101→5

Hence, (1BC)16 = (675)8

- **Octal to Binary Conversion:**

To convert an octal number to binary, replace each octal digit with its three-digit binary equivalent.

Example: Convert the octal number 753 to binary.

Octal to Binary Conversion Table:

Octal	Binary
0	**000**
1	**001**
2	**010**
3	**011**
4	**100**
5	**101**
6	**110**
7	**111**

➔ Step 1: Replace each octal digit with its binary equivalent:

```
  7     5     3

111   101   011
```

Result: The octal number 753 is equivalent to the binary number 111101011.

- **Octal to Hexadecimal Conversion:**

In order to convert the octal number into hexadecimal, we need to express every octal value using 3 binary bits. Here are the steps:

➔ Step 1 : Convert (57)8 into Binary:To convert an octal number to binary, we must represent each octal value using three binary bits.

Binary equivalent of 5 is (101)2.

Binary equivalent of 7 is (111)2.

= (57)8

= (101)(111)

= (101111)2

➔ Step 2 : Convert (101111)2 into Hexadecimal: To convert a binary number to hexadecimal, group every four binary bits and compute the value [From left to right].

(101111)2 in hexadecimal

= (101111)2

= (10)(1111)

= (2)(15)

= (27)16

Result: The octal number (57)8 is equivalent to the hexadecimal number (27)16.

2. **"Discuss the differences between bits, bytes, and words. Explain their relationship and significance in data storage."**

The differences between bits, bytes, and words can be discussed as follows:

- Bits:
  - Definition: A bit (short for binary digit) is the smallest data unit in computing. It has two possible values: 0 or 1.

    - Significance: Bits are the building blocks of all digital data. They are used to encapsulate information in its simplest form, laying the groundwork for more complicated data representations.
    - Relationship: A byte is formed by a collection of bits, and bytes are joined to generate bigger data structures such as words.
- Bytes:
    - Definition: A byte is an 8-bit unit of measurement. It is the most basic unit for storing and expressing letters, integers, and other data.
    - Significance: Bytes are a more practical unit for expressing characters, which are the building blocks of text and data. Bytes also support a variety of encoding systems, including ASCII and Unicode.
    - Relationship: Bits are the building blocks of bytes. A single ASCII character, for example, is represented by one byte (8 bits).
- Words:
    - Definition:A word is a fixed-size unit of data used for processing and storing in the central processor unit (CPU) or memory of a computer.
    - Significance: The size of a word is dictated by the computer's architecture. It is the quantity of data that the CPU can handle at the same time. Word sizes that are often used include 16, 32, and 64 bits.
    - Relationship: A word is larger than a byte and is often composed of many bytes. The connection between bits, bytes, and words is frequently altered by the architecture of the computer.

➔ Relationship and Significance in Data Storage:

- ◆ Bit-Level Storage: Individual bits are the smallest storage units, and they are used to represent binary data. They are essential in digital communication and data encoding.
- ◆ Byte-Level Storage: Bytes are the basic units of storage in computers. They are used to represent letters, numbers, and other information. Files, documents, photos, and other data are all saved in

byte sequences. Different encodings, such as UTF-8, which can employ variable-length bytes, enable the representation of diverse characters from various languages.

- ◆ Word-Level Storage: Words have precise widths that are tailored for the computer's CPU architecture. A 32-bit CPU, for example, processes data in 32-bit chunks, whereas a 64-bit CPU operates in 64-bit chunks. Word-level storage is important for efficient data processing and memory allocation.

In conclusion, bits, bytes, and words are essential units that serve various functions in data representation, storage, and processing. They are linked in the sense that bits are formed from bytes, and bytes are concatenated to make words. Understanding their interdependence and importance aids in the design of effective data storage and processing systems.

3. **“Convert between different storage units, such as kilobytes, megabytes, gigabytes, and terabytes. Provide examples of real-world scenarios where these storage units are used.”**

The following conversions can be used to convert between different storage units:

- ➔ 1 kilobyte (KB) is equal to 1024 bytes.
- ➔ 1 megabyte (MB) is equal to 1024 kilobytes.
- ➔ 1 gigabyte (GB) is equal to 1024 megabytes.
- ➔ 1 terabyte (TB) is equal to 1024 gigabytes.

If we have 2048 kilobytes, we can convert them to megabytes by dividing 2048 by 1024, which gives us 2 megabytes. Similarly, we may convert 512 gigabytes to terabytes by dividing 512 by 1024, which gives us 0.5 terabytes (Wiley, 5th edition, p. 41)..

- Real-World Scenarios: These storage units are commonly used in various real-world scenarios. Here are a few examples:

- Personal Computers: Storage capacity for personal computers typically range from a few gigabytes to many terabytes. Users may store vast volumes of data, such as papers, images, movies, and software applications, using this method.
- Cloud Storage: Cloud storage providers provide storage plans in the gigabyte or terabyte range. Users may store their files and data on distant servers and access them from any location with an internet connection.
- Multimedia Content: Large files, such as high-definition films and high-resolution photos, need a lot of storage space. A one-minute video clip in full HD definition, for example, might require hundreds of megabytes or even gigabytes of storage.
- Data Centres: Data centres sometimes have storage capabilities estimated in petabytes (1 petabyte = 1024 terabytes) or even exabytes (1 exabyte = 1024 petabytes), and they store and handle massive quantities of data for organisations.
- Backup and archiving: Backup and archival systems are designed to store massive amounts of data for extended periods of time. To maintain data integrity and long-term preservation, these systems frequently use terabytes or even petabytes of storage space.

It should be noted that storage needs might vary substantially based on the exact use case and the quantity of data processed or saved (Wiley, 5th edition, p. 136).

4. "Describe the concepts of signed and unsigned integer representation. Perform conversions between the two representations and provide examples."

Signed and unsigned integer representations are methods used to store and manipulate integer values in a computer. Here are the detailed explanations:

- **Unsigned Integer Representation:** In the context of representing unsigned integers, the binary form of a number is directly employed to denote positive whole numbers. The span of values that can be depicted hinges on the quantity of available bits. As an illustration, an 8-bit unsigned integer can portray values spanning from 0 to 255 (Wiley, 5th edition, p. 167).

- **Signed Integer Representation:** In the context of representing signed integers, a technique is employed to encode both positive and negative whole numbers. One prevalent approach is the sign-and-magnitude representation, where a designated sign bit signifies the number's sign (0 for positive, 1 for negative), while the remaining bits convey the magnitude. Another approach is the two's complement representation, wherein the highest-order bit denotes the sign, and the remaining bits denote the magnitude.
- **Converting between Signed and Unsigned Representations:** Converting a signed integer into an unsigned representation is straightforward: we interpret the bit pattern as an unsigned value. For instance, the signed presentation of -5 in an 8-bit two's complement format is 11111011, which translates to the unsigned value 251.

  On the flip side, converting an unsigned integer to a signed representation requires considering the permissible value range for representation. Take, for instance, an 8-bit two's complement format where the range spans from -128 to 127. If the unsigned value falls within this range, we can directly interpret the bits as a signed value. Conversely, if the unsigned value exceeds the highest attainable positive value, bit adjustments are essential to signify a negative value (Wiley, 5th edition, p. 169).

  Example:

  - ➔ Let's examine the conversion of the unsigned value 200 to a signed representation using an 8-bit two's complement format. Given that 200 lies within the scope of positive values, we can readily construe the bits as a signed value. As a result, the representation becomes 11001000, denoting -56.
  - ➔ Let's consider the conversion of a signed integer to its unsigned representation. Suppose we have the signed value -85 in an 8-bit two's complement format. We'll convert this to its unsigned equivalent.
    - ◆ Signed Value: -85
    - ◆ Representation: In 8-bit two's complement, -85 is represented as 10101011.

    Now, let's convert this signed value to its unsigned representation:

- Unsigned Representation: To convert a signed value to unsigned, we interpret the bit pattern as if it were an unsigned value. So, for the bit pattern 10101011, we treat it as an unsigned binary number.
- Converting the binary value 10101011 to decimal:

10101011 = 1 * 2^7 + 0 * 2^6 + 1 * 2^5 + 0 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0

= 128 + 0 + 32 + 0 + 8 + 0 + 2 + 1

= 171

- So, the signed value -85 in 8-bit two's complement form is equivalent to the unsigned value 171.

5. **“Explain the IEEE 754 standard for floating-point representation. Perform conversions between binary and decimal representations using IEEE 754 standard and provide examples.”**

The IEEE 754 standard provides computer formats for expressing floating-point integers. It defines the bit arrangement for a floating-point number's sign, exponent, and mantissa. The standard specifies formats for floating-point values of various sizes, including 16-bit, 32-bit, 64-bit, and 128-bit.

A binary exponent and a binary mantissa are used in the standard. Excess-N notation is used to record the exponent, where N is a bias number that allows for both positive and negative exponents. The mantissa reflects the number's important digits (Wiley, 5th edition, p. 191).

To convert a decimal number to binary using the IEEE 754 standard, follow these steps:

- Determine if the number is positive or negative.
- Convert the number's absolute value to binary.
- Shift the binary point to the left or right to normalise the binary representation.
- Count the number of shifts done in step 3 to find the exponent.
- Use the IEEE 754 format to encode the sign, exponent, and mantissa.

To convert a binary number to a decimal number using the IEEE 754 standard, follow these steps:

- Based on the sign bit, determine the number's sign.
- Using the binary form, calculate the exponent.
- Determine the mantissa value.
- Adjust the mantissa value based on the exponent.
- Multiply the modified mantissa by the power of two that corresponds to the exponent and sign.

Examples:

- ➔ Example 1: Convert the decimal number 10.75 to binary using IEEE 754 standard.
    - ◆ Sign: Positive
    - ◆ Absolute value: 10.75
    - ◆ Binary representation: 1010.11
    - ◆ Normalise: 1.01011
    - ◆ Exponent: 3 (shifted 3 places to the left)
    - ◆ Encode: Sign = 0, Exponent = 127 + 3 = 130 (in excess-127 notation), Mantissa = 01011000000000000000000
    - ◆ IEEE 754 representation: 0 10000010 01011000000000000000000

- ➔ Example 2: Convert the binary number 1101.0011 to decimal using IEEE 754 standard.
    - ◆ Sign: Negative
    - ◆ Exponent: 4 (from the binary representation)
    - ◆ Mantissa: 1.0011
    - ◆ Adjusted mantissa: 1.0011 * 2^4 = 10011
    - ◆ Decimal value: -19

These examples demonstrate the process of converting decimal numbers to binary and vice versa using the IEEE 754 standard.

6. **"Describe the binary-coded decimal (BCD) representation and its uses. Provide examples of how BCD is used in practical applications."**

Binary-Coded Decimal (BCD) is a technique of numerical representation that uses binary digits to represent decimal quantities. Each decimal digit is transformed to a 4-bit binary code in BCD. In BCD, the decimal value 68 would be represented as 01101000. BCD is frequently used in commercial applications where decimal rounding and accuracy are required (Wiley, 5th edition, p. 164).

Uses of Binary-Coded Decimal (BCD) in Practical Applications are listed as follows:

- BCD is often used in commercial applications that demand precise digit-for-digit decimal representation. It is effective in situations when there is a lot of input and output but just a little amount of calculation. This area includes a wide range of commercial applications. BCD is also favoured for translating between BCD and character form since the last four bits of ASCII, EBCDIC, and Unicode numeric character forms match perfectly to the BCD representation of that digit.
- One practical application of BCD is in financial systems, where accurate decimal computations for monetary values are critical. BCD provides for accurate financial computations by precisely representing and manipulating decimal numbers.
- Another application of BCD is in data entry systems, where it helps ease the conversion process from alphabetical to numerical BCD representation. By removing all except the rightmost four bits of a character, the BCD value may be obtained, making data entry and conversion more efficient (Wiley, 5th edition, p. 167).

Overall, BCD represents decimal values in binary form, enabling for accurate decimal arithmetic while preserving decimal rounding and accuracy. It is especially beneficial in corporate applications and settings that need precise decimal computations.

7. **"Discuss character encoding systems such as ASCII and Unicode. Explain the differences between them and provide examples of their applications."**

ASCII (American Standard Code for Information Interchange) and Unicode are two widely used character encoding systems.

ASCII is a 7-bit code that uses numeric codes to represent characters. ANSI created it, and it incorporates the Latin alphabet, Arabic numbers, and standard punctuation marks used in English. Because of its 128-entry table, ASCII has restrictions in terms of the amount of available characters.

In contrast, Unicode is a more extensive and current character encoding scheme. It can store nearly a million characters and use a mix of 8-bit, 16-bit, and 32-bit words. Characters from multiple character-based alphabets, ideographs for Chinese, Japanese, and Korean languages, punctuation, symbols, and control characters are all included in Unicode (Wiley, 5th edition, p. 131).

- Differences and Applications:
  - Character Set: ASCII provides a restricted character set, which is typically used for English-based systems. Unicode, on the other hand, provides a significantly greater set of characters, making it appropriate for multilingual applications.
  - Code Length: ASCII employs 7 bits to represent characters, allowing for 128 different codes. Unicode employs variable-length encoding techniques such as UTF-8, UTF-16, and UTF-32. UTF-8 makes use of 8-bit code units, UTF-16 makes use of 16-bit code units, and UTF-32 makes use of 32-bit code units.
  - Compatibility: Unicode is meant to be backwards compatible with ASCII. If an end delimiter is applied, a Unicode text file confined to the Latin-1 character set can be read successfully on computers that do not support Unicode. However, due to its restricted character set, ASCII is incompatible with Unicode.
  - Applications: It includes legacy systems, telephony, and early computer systems. Unicode is the de facto standard for current operating systems such as Windows, Linux, OS X, iOS, and Android. It's used in web development, internationalisation, and multilingual software.

In summary, ASCII is a basic character encoding system with limited character support, whereas Unicode is a more complete and adaptable encoding system with extensive character support for multilingual applications. Unicode has evolved into the industry standard for current operating systems and is extensively utilised in a variety of sectors (Wiley, 5th edition, p. 132).

8. **"Interpret an ASCII code table and explain the concept of extended ASCII. Provide examples of extended ASCII characters and their uses."**

ASCII (American system Code for Information Interchange) is a popular character encoding system that provides unique numeric codes to characters such as letters, numerals, punctuation marks, and control characters. ASCII encoded 128 distinct characters using 7 bits.

An ASCII code table maps characters to their corresponding numerical codes. Here is a small portion of the ASCII table:

Decimal	Character
65	A
66	B
67	C
…	…
97	a
98	b
99	c
…	…
48	0
49	1
50	2
…	…

Each character is associated with a unique decimal value in the ASCII table. For example, the uppercase letter 'A' is represented by the decimal value 65.

Extended ASCII: Extended ASCII, also known as high ASCII, refers to variations of the ASCII character encoding that use 8 bits instead of the standard 7 bits. This extended encoding allows for the representation of additional characters beyond the original 128 ASCII characters. Various extended ASCII sets have been developed, each adding extra characters tailored to specific languages and needs.

- Examples of Extended ASCII Characters and Uses:
  - Latin-1 (ISO 8859-1): This extended ASCII set includes Western European language characters. Accent letters (é,), unusual punctuation (, €), and symbols used in several languages are examples.
  - Windows-1252: A Latin-1 extension with extra characters such as curly quotes, en dashes, em dashes, and more.
  - CP437 (IBM Extended ASCII): This set, which was used by some DOS programmes and early PCs, contains line-drawing characters, symbols, and some special characters.
  - CP850 (Multilingual MS-DOS): A CP437 expansion, CP850 adds extra characters for languages such as French, German, and Portuguese, among others.
- Uses of Extended ASCII Characters:
  - Language-Specific Characters: Beyond the standard ASCII set, Extended ASCII enables for the depiction of characters required for diverse languages.
  - Extended ASCII has symbols for mathematics (,), currency (, £), and other specific functions.
  - Extended ASCII sets include curly quotes (" ",''), em dashes (—), and en dashes (-).
  - Graphics and Drawing: Some extended ASCII sets include block letters, line-drawing symbols, and graphical components that are utilised in text-based interfaces.

It's crucial to remember that using extended ASCII sets might cause compatibility problems because various systems may utilise different sets. Unicode, a more complete character encoding standard, promises to offer a common encoding for characters across many languages and symbols, addressing many of ASCII's shortcomings (Wiley, 5th edition, p. 132).

9. **"Explain the encoding schemes in the Unicode standard, including UTF-8, UTF-16, and UTF-32. Discuss the advantages and applications of each encoding scheme."**

Unicode, the international standard for character encoding, defines three encoding schemes: UTF-8, UTF-16, and UTF-32.

- UTF-8: UTF-8 is an 8-bit code unit variable-length encoding system. It is backwards compatible with ASCII and can represent all Unicode characters. Because of its effectiveness in expressing ASCII characters and compatibility with current ASCII-based systems, UTF-8 is extensively used on the web and in email communication.
- UTF-16: Another variable-length encoding technique that uses 16-bit code units is UTF-16. All Unicode characters, even those beyond the Basic Multilingual Plane (BMP), may be represented using it. UTF-16 is widely used in operating systems and programming languages that store and process text as 16-bit code units internally.
- UTF-32: It is a fixed-length encoding method that employs 32-bit code units. It is also known as UCS-4. It can represent any Unicode characters, even those that are not part of the BMP. UTF-32 provides a simple mapping between code points and code units, making Unicode text processing and manipulation easier.

Advantages and Applications of Each Encoding Scheme are as follows:

- UTF-8: provides a number of benefits. It is backward compatible with ASCII and can represent ASCII letters with a single byte. This makes it suitable for use with English and other ASCII-based languages. UTF-8 is commonly used on the internet because it reduces the size of text files while remaining compatible with existing ASCII-based systems. Many computer languages and operating systems use it as the default encoding.

- UTF-16: The ability to represent all Unicode characters, including those beyond the BMP, is the major advantage of UTF-16. It is widely used in operating systems and programming languages that store and process text as 16-bit code units internally. However, because it takes two bytes for ASCII characters, UTF-16 may be less efficient than UTF-8 for ASCII-based content.
- UTF-32: UTF-32 provides a simple mapping between code points and code units, making Unicode text easy to process and handle. It can represent any Unicode characters, even those that are not part of the BMP. UTF-32, on the other hand, demands more storage space than UTF-8 and UTF-16 since it takes four bytes for each code unit.

In conclusion, UTF-8 is extensively utilised for online and email communication, UTF-16 is widely used in operating systems and computer languages, and UTF-32 provides a simple mapping for processing Unicode text. The encoding method used is determined by variables such as compatibility requirements, storage efficiency, and the necessity to represent characters other than BMP.

References

The Architecture of Computer Hardware, Systems Software, and Networking: An Information Technology Approach, 6th Edition

Publisher: Wiley, Authors: Irv Englander, Wilson Wong