System Hardware

Lecture 3 -Data Representation Part-1 Integers

Objectives

- Understand the fundamentals of numerical data representation and manipulation in digital computers.
- Master the skill of converting between various number systems.
- Signed Integer Computations: Addition, multiplication and division.
- Understand how errors can occur in computations because of overflow and truncation.

Introduction

- A *bit* is the most basic unit of information in a computer.
  - It is a state of “on” or “off” in a digital circuit.
  - Sometimes these states are “high” or “low” voltage instead of “on” or “off..”
- A *byte* is a group of eight bits.
  - A byte is the smallest possible ***addressable*** unit of computer storage.
  - The term, **“addressable”** means that a particular byte can be retrieved according to its **address** in memory.

Introduction

- A *word* is a contiguous group of bytes.
  - Word sizes of 16, 32, or 64 bits are most common. Note that each word has 2^n bytes.
  - In a ***word-addressable*** system, a word is the smallest addressable unit of storage.
- A group of 4 bits is called a ***nibble*** *or* ***hexit***.
  - A hexit represents a single hexadecimal digit.
  - A byte, therefore, is represented by two hexadecimal digits: a "high-order hexit," and a "low-order" hexit.

Positional Numbering Systems

- Bytes store numbers using the position of each bit to represent a power of 2.
  - The binary system is also called the base-2 system.
  - Our decimal system is the base-10 system. It uses powers of 10 for each position in a number.
  - Any integer quantity can be represented exactly using any base. We will be looking at base 2 (binary) and base 16 (hexadecimal)

Positional Numbering Systems

- The decimal number 947 in powers of 10 is:

$$9 \times 10^{2} + 4 \times 10^{1} + 7 \times 10^{0}$$

- The decimal number 5836.47 in powers of 10 is:

$$5 \times 10^{3} + 8 \times 10^{2} + 3 \times 10^{1} + 6 \times 10^{0} + 4 \times 10^{-1} + 7 \times 10^{-2}$$

Positional Numbering Systems

- The binary number 11001 in powers of 2 is:

$$1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$
$$= 16 + 8 + 0 + 0 + 1 = 25$$

- When the base of a number is something other than 10, the base is denoted by a subscript.
  - Sometimes, the subscript 10 is added for emphasis but it is not required:

$$11001_2 = 25_{10}$$

Natural Numbers

- So far, we have seen ***Natural Numbers***. They are the non negative numbers including zero.
- Using binary bit patterns to represent natural numbers (aka unsigned numbers) is called ***Natural Semantics***.
- Suppose that our registers are 32-bits long. A register would be able to hold 2^{32} distinct 32-bit patterns.
- If we interpret registers using natural semantics, then the 2^{32} bit patterns represent the (natural) numbers from 0 to $2^{32} - 1$. To generalize: 2^n bit patterns from 0 to $2^n - 1$
- **Example**: In a 3-bit register, there are 2^3 or 8 bit patterns: 000=0, 001=1, 010=2, 011=3, 100=4, 101=5, 110=6, 111=7

The Binary Numbering System

- Because binary numbers are the basis for all data representation in digital computer systems, it is important that you become proficient with this base system.
- Your knowledge of the binary numbering system will enable you to understand the operation of all computer components as well as the design of instruction set architectures.

Converting To Binary

- There are 2 methods for conversion: the ***Subtraction Remainder method*** and the ***Division Remainder method***.
- ***The Subtraction Remainder method*** is more intuitive, but cumbersome. However, it reinforce the ideas behind base mathematics and it is strongly encouraged. In this method, we represent the given decimal number as the sum of powers of two starting from the largest possible of 2 and going down.

Converting To Binary Subtraction Remainder Method

- **Suppose we want to convert the decimal number 190 to binary.**
  - We know that $2^8 = 256$ and that $2^7 = 128$ so our result will be eight digits wide.
  - The largest power of 2 that can be subtracted is therefore 2^7.
  - Subtract 128 from 190, giving 62.

$$\begin{array}{rl} 190 & \\ \underline{-128} & = 1 \text{ X } 2^7 \\ 62 & \end{array}$$

Converting To Binary Subtraction Remainder Method

Converting 190 to binary...

- The next power of 2, 2^{6} = 64 is too large, but we have to assign a placeholder of zero.
- The next power of 2, 2^{5} = 32. We'll need one of these, so subtract 32 and write down the result.
- The next power of 2, 2^{4} = 16. We can subtract one of these too.
- The next power of 2, 2^{3} = 8. We can subtract one of these too.
- Continue, until all powers of 2 are represented including place holders.

190= 1 X 2^{7} + 0 X 2^{6} + 1 X 2^{5} + 1 X 2^{4} + 1 X 2^{3} +1 X 2^{2}+1 X 2^{1} + 0 X 2^{0}

<u>Note</u>: The bit corresponding to exponent 0 is on the right ***(least significant bit)*** and the bit corresponding to largest exponent is the ***most significant bit*** *(on the right)*.

190 = 10111110_2

$$\begin{array}{r l}
190 & \\
\underline{-128} & = 1 \ X \ 2^{7} \\
62 & \\
\underline{-\ \ 0} & = 0 \ X \ 2^{6} \\
62 & \\
\underline{-\ 32} & = 1 \ X \ 2^{5} \\
30 & \\
\underline{-\ 16} & = 1 \ X \ 2^{4} \\
14 & \\
\underline{-\ \ 8} & = 1 \ X \ 2^{3} \\
6 & \\
\underline{-\ \ 4} & = 1 \ X \ 2^{2} \\
2 & \\
\underline{-\ \ 2} & = 1 \ X \ 2^{1} \\
0 & \\
\underline{-\ \ 0} & = 0 \ X \ 2^{0} \\
0 &
\end{array}$$

Converting To Binary Division Method

- The other method of converting integers from decimal to binary uses division.
- This method is mechanical and easy.
- It employs the idea that successive division by a base is equivalent to successive subtraction by powers of the base.
- Let's use the division remainder method to again convert the decimal 190 to binary.

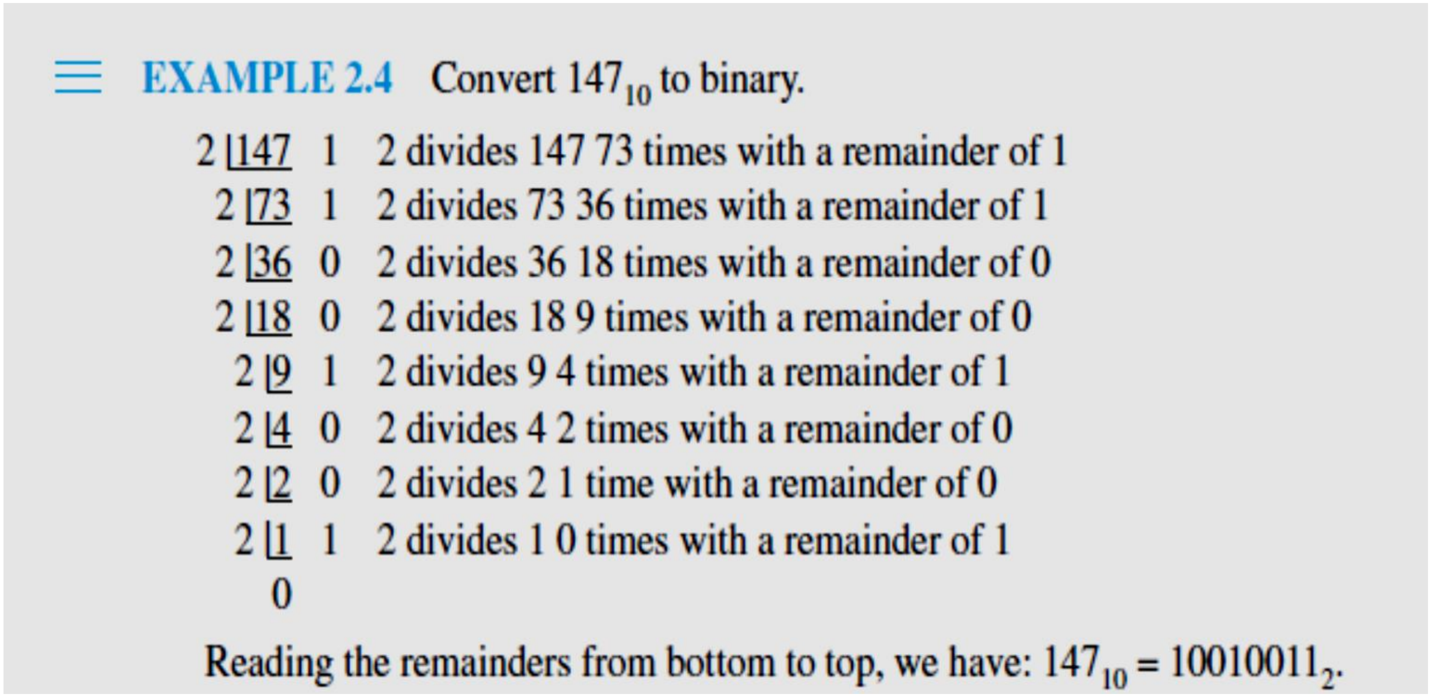
Converting To Binary Division Method

Converting 190 to binary...

- First we take the number that we wish to convert and carry out subsequent divisions by 2.
- At each step, record the quotient and the remainder.
- Read remainders from ***bottom to top*** and record them ***from most significant to least significant bit***.
- $190_{10} = 10111110_2$

2 | 190	
2 | 95	0
2 |47	1
2 |23	1
2 |11	1
2 |5	1
2 |2	1
2 |1	0
0	1

Division Method Another Example



EXAMPLE 2.4 Convert 147_{10} to binary.

```
2 |147  1   2 divides 147 73 times with a remainder of 1
 2 |73  1   2 divides 73 36 times with a remainder of 1
 2 |36  0   2 divides 36 18 times with a remainder of 0
 2 |18  0   2 divides 18 9 times with a remainder of 0
  2 |9  1   2 divides 9 4 times with a remainder of 1
  2 |4  0   2 divides 4 2 times with a remainder of 0
  2 |2  0   2 divides 2 1 time with a remainder of 0
  2 |1  1   2 divides 1 0 times with a remainder of 1
     0
```

Reading the remainders from bottom to top, we have: $147_{10} = 10010011_2$.

The Hexadecimal Numbering System

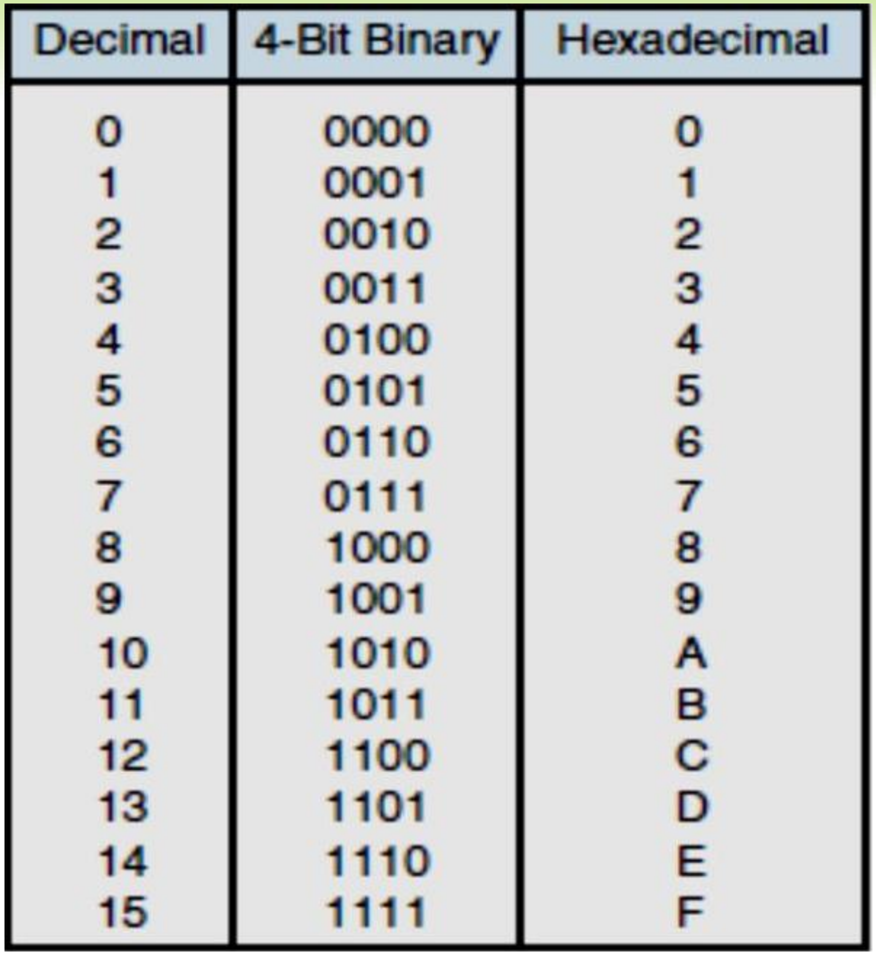
- The binary numbering system is the most important base system for digital computers.
- However, it is difficult to read long strings of binary numbers -- and even a modestly-sized decimal number becomes very long in binary.
  - For example: $11010100011011_2 = 13595_{10}$
- For compactness and ease of reading, binary values are usually expressed using the ***hexadecimal (base-16)*** numbering system.

The Hexadecimal Numbering System

- The hexadecimal numbering system uses the digits 0 through 9 and the letters A through F.
  Example: The decimal number 12 is C_{16} and 14 is E_{16}.
- It is easy to convert between base 16 and base 2, because $16 = 2^4$.
- Thus, to convert from binary to hexadecimal, all we need to do is group the binary digits **from right to left** into groups of four.

A group of four binary digits is called a hexit

Converting Between Bases
Values of Number Bases



Decimal	4-Bit Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

The Hexadecimal Numbering System

- Grouping each 4 binary digits into a hexit, 11010100011011_2 in hexadecimal is:

0011	0101	0001	1011
3	5	1	B

If the number of bits is not a multiple of 4, pad on the left with zeros to a multiple of 4.

- Octal (base 8) values are derived from binary by using groups of three bits ($8 = 2^3$):

011	010	100	011	011
3	2	4	3	3

- Octal is not used much any more.

Converting To Hexadecimal

The ***division remainder method*** can be used to convert numbers from decimal to hexadecimal.

Example: Convert 299_{10} to hex

16|<u>299</u> B divides 18 times (16x18=288) with remainder of 11 (B)

16|<u>18</u> 2 divides 1 times with a remainder of 2

16|<u>11</u> 1 divides 0 times with a remainder of 1

0

Read remainders from ***bottom to top***

Answer: $12B_{16}$

Signed Integer Representation

- Conversions presented so far involved unsigned (natural) numbers only
- Earlier systems used to implement signed integers (such as ***signed magnitude*** and ***one's complement)*** had the disadvantage of having two different representations for zero: positive and negative zero.
- Two's complement solves this problem.
- Two's complement is the base complement of the binary numbering system; The *base complement* of a non-zero number *N* in base *2* with *d* digits is $2^d - N$.
- *E.g. N=5 with 4 digits, base comp. is* $2^4-5=11=1011=-5$.

Signed Integer Representation Hexadecimal Base Complement

- Recall that the ***base complement*** of a non-zero number N in base 2 with d digits is $2^d - N$.
- In base 16 (hexadecimal), the base complement is $16^d - N$.
- *E.g. N=190. Base complement to be represented with 3 digits is* $16^3 - 190 = 4096 - 190 = 3906 = C18_{16}$
- $-190_{10} = C18_{16}$

Signed Integer Representation

- To express a value in two's complement representation:
  - If the number is positive, just convert it to binary.
  - If the number is negative, find the ***one's complement*** of the number (***flip every bit***) and then add 1.
- Example:
  - In 8-bit binary, 5 is: 00000101
  - One's complement representation is: 11111010
  - Adding 1 gives -5 in two's complement: 11111011
  - ***Base complement*** *N=5, 8 digits:* $2^8 - 5 = 251 =$ 11111011

Signed Integer Representation Example

A 4-bit register, using two's complement semantics:

Positive Values:

0000 = 0, 0001 = 1, 0010 = 2, 0011 = 3,

0100 = 4, 0101 = 5, 0110= 6, 0111= 7

Negative Values:

1000 = -8, 1001 = -7, 1010 = -6, 1011 = -5,

1100 = -4, 1101 = -3, 1110 = -2, 1111 = -1.

- Since 0 occupies a position as a positive number, ***there is one more nonzero negative number*** than ***nonzero positive numbers***. In n-bits, we can represent signed numbers in the range: (-2^{n-1})… $(2^{n-1} – 1)$. E.g. n=4 bits can represent -8 to 7.
- The most significant bit (***the sign bit***) indicates if a number is positive or negative but is not reserved for this purpose.

Signed Integer Representation Binary Addition

- Binary addition is as easy as it gets. You need to know only four rules:

```
0 + 0 =  0      0 + 1 =  1
1 + 0 =  1      1 + 1 = 10
```

- The simplicity of this system makes it possible for digital circuits to carry out arithmetic operations.
  - We will describe these circuits in future lectures.

Let's see how the addition rules work with signed numbers . . .

Signed Integer Representation Binary Addition

- With two's complement arithmetic, all we do is add our two binary numbers. Just discard any carries emitting from the high order bit.

– **Example 1:** Using two's complement binary arithmetic, find the sum of 48 and - 19.

$$\boxed{1} \longleftarrow \begin{array}{r} 1\;1\;\;\;\;\;\;\;\;\;\;\;\;\;\;\;\;\; \\ 0\,0\,1\,1\,0\,0\,0\,0 \\ +\;\;1\,1\,1\,0\,1\,1\,0\,1 \\ \hline 0\,0\,0\,1\,1\,1\,0\,1 \end{array}$$

We note that 19 in binary is:	**00010011**,
so -19 using one's complement is:	**11101100**,
and -19 using two's complement is:	**11101101**.

Signed Integer Representation Overflow

- When we use any finite number of bits to represent a number, we always run the risk of the result of our calculations becoming ***too large*** or ***too small*** to be stored in the computer.
- This situation is called ***overflow*** and is used in the context of signed numbers (Two's complement).
- While we can't always prevent overflow, we can always *detect* overflow (in complement arithmetic)
- **Note** that a carry out of the register is not in itself a sign of overflow (as seen in Example 1).

Signed Integer Representation Detecting Overflow

Example 2:

- Find the sum of 107 and 46, using two's complement binary arithmetic
- There is no carry out of the register.
- **But, We see that the nonzero carry from the seventh bit *overflows* into the sign bit, giving us the erroneous result: 107 + 46 = -103.**

$$\begin{array}{cccccccccc} & \boxed{1} & 1 & & 1 & 1 & 1 & & \\ & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 \\ + & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ \hline & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \end{array}$$

But overflow into the sign bit does not always mean that we have an error.

Signed Integer Representation Detecting Overflow

Example 3:

- Using two’s complement binary arithmetic, find the sum of 23 and -9.
- We see that there is carry into the sign bit and carry out. The final result is correct: 23 + (-9) = 14.

$$
\begin{array}{cccccccccc}
\textcircled{1} & \leftarrow & \textcircled{1} & 1 & 1 & & 1 & 1 & 1 & \\
 & & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\
 & + & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\
\hline
 & & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0
\end{array}
$$

<u>Rule for detecting overflow in signed two’s complement</u>:
Overflow occurs if the carry into the sign bit and the carry out of it are different.

Signed Integer Representation Correcting Overflow

Back to Example 2 (107+46):

- To fix overflow, we need to extend the size of the numbers (by 1-bit) before doing the addition.
- Extend each number by duplicating the value of the sign bit to the left.

```
   11  111
  001101011
+ 000101110
-----------
  010011001
```

Extending a signed integer:

1000 = -8 (4 bits). To extend to 8 bits,

replicate the sign bit on the left.

1111 1000 = -8 (8 bits)

Another example: 0100 = 4 (4 bits)

0000 0100 = 4 (8-bits)

Signed Integer Representation

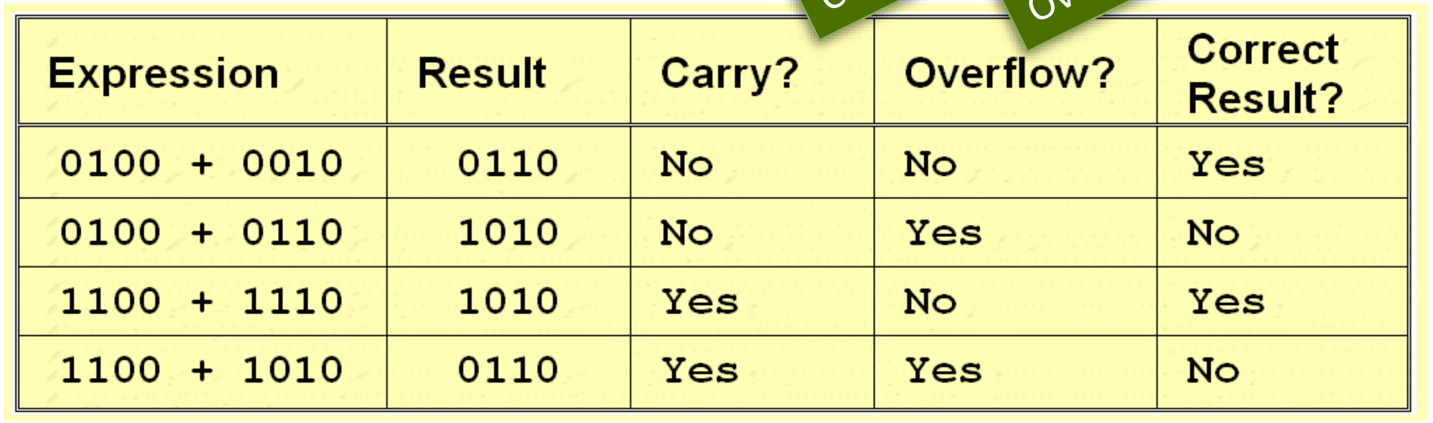
- Signed and unsigned numbers are both useful.
  - For example, memory addresses and computer instructions are always unsigned.
- Using the same number of bits, unsigned integers can express twice as many “positive” values as signed numbers.
- **In Unsigned numbers:** If a carry occurs (from the most left bit) then it ***always*** indicates a “***problem or a carry flag”***, which means the new value is too large to be stored in the given number of bits.
- **In signed numbers:** If a carry occurs (from the most left bit), it does not always indicate an overflow.

Signed Integer Representation

- In signed numbers addition, overflow occurs when two positives produce a negative result or two negatives produce a positive result. However, a carry out does not prove anything by itself.
- In unsigned numbers addition, a carry out is always a problem.

Carry out of the leftmost bit

Overflow that changes sign



Expression	Result	Carry?	Overflow?	Correct Result?
0100 + 0010	0110	No	No	Yes
0100 + 0110	1010	No	Yes	No
1100 + 1110	1010	Yes	No	Yes
1100 + 1010	0110	Yes	Yes	No

Signed Integer Representation

- We can do binary multiplication and division by 2 very easily using an *arithmetic shift* operation
- A *left arithmetic shift* inserts a 0 in for the rightmost bit and shifts everything else left one bit; in effect, it multiplies by 2
- A *right arithmetic shift* shifts everything one bit to the right, but copies the sign bit; it divides by 2
- Let's look at some examples.

Signed Integer Representation

Example:

Multiply the value 11 (expressed using 8-bit signed two's complement representation) by 2.

We start with the binary value for 11:

    00001011 (+11)

We shift left one place, resulting in:

    00010110 (+22)

The sign bit has not changed, so the value is valid.

To multiply 11 by 4, we simply perform a left shift twice.

Signed Integer Representation

Example:

Multiply the value -3 (expressed using 4-bit signed two’s complement representation) by 2.

We start with the binary value for -3:

1101 (-3)

We shift left one place, resulting in:

1010 (-6)

The sign bit has not changed, so the value is valid.

To multiply -3 by 4, we simply perform a left shift twice.

Signed Integer Representation

Example:

Divide the value 12 (expressed using 8-bit signed two's complement representation) by 2.

We start with the binary value for 12:

00001100 (+12)

We shift right one place, resulting in:

00000110 (+6)

(Remember, we carry the sign bit to the left as we shift.)

To divide 12 by 4, we right shift twice.

Signed Integer Representation

Example:

Divide the value -12 (expressed using 8-bit signed two’s complement representation) by 2.

We start with the binary value for -12:

11110100 (-12)

We shift right one place, resulting in:

11111010 (-6)

(Remember, we carry the sign bit to the left as we shift.)

To divide -12 by 4, we right shift twice.

Character Codes

- Calculations aren't useful until their results can be displayed in a manner that is meaningful to people.
- We also need to store the results of calculations, and provide a means for data input.
- Thus, human-understandable characters must be converted to computer-understandable bit patterns using some sort of character encoding scheme.

Character Codes

- As computers have evolved, character codes have evolved.
- Larger computer memories and storage devices permit richer character codes.
- The earliest computer coding systems used six bits.
- Binary-coded decimal (BCD) was one of these early codes. It was used by IBM mainframes in the 1950s and 1960s.

2.6 Character Codes

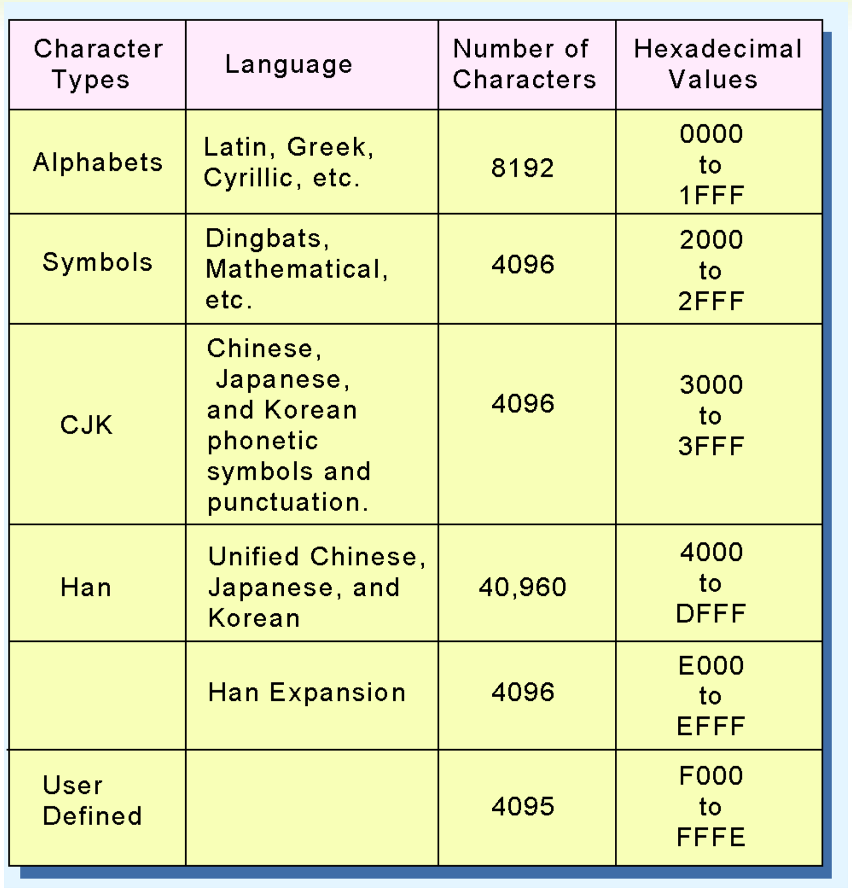
- Other computer manufacturers chose the 7-bit ASCII (American Standard Code for Information Interchange) as a replacement for 6-bit codes.
- **ASCII:** 8 bits would give us 2^8 = 256 characters
- ASCII uses the ***lower-order 7 bits only*** to distinguish characters and is able to represent 128 different characters.
- Until recently, ASCII was the dominant character code outside the IBM mainframe world.

2.6 Character Codes

- Many of today's systems embrace Unicode, a 16-bit system that can encode the characters of every language in the world.
  - The Java programming language, and some operating systems now use Unicode as their default character code.
- The Unicode code space is divided into six parts. The first part is for Western alphabet codes, including English, Greek, and Russian.

Character Codes

- The Unicode code space allocation is shown at the right.
- The lowest-numbered Unicode characters comprise the ASCII code.
- The highest provide for user-defined codes.



Character Types	Language	Number of Characters	Hexadecimal Values
Alphabets	Latin, Greek, Cyrillic, etc.	8192	0000 to 1FFF
Symbols	Dingbats, Mathematical, etc.	4096	2000 to 2FFF
CJK	Chinese, Japanese, and Korean phonetic symbols and punctuation.	4096	3000 to 3FFF
Han	Unified Chinese, Japanese, and Korean	40,960	4000 to DFFF
	Han Expansion	4096	E000 to EFFF
User Defined		4095	F000 to FFFE

Conclusion

- Computers store data in the form of bits, bytes, and words using the binary numbering system.
- Hexadecimal numbers are formed using four-bit groups called nibbles or hexits.
- Signed integers are stored in two’s complement representation.
- Character data is stored using 7-bit ASCII, and more recently 16-bit Unicode.