CMSC 313 Lecture 03

- Multiple-byte data
  - big-endian vs little-endian
  - sign extension
- Multiplication and division
- Floating point formats
- Character Codes
Exercise 1. Convert the following numbers.

a. $62347_{10}$ to unsigned binary
b. $8DF6_{16}$ to base 2
c. $41.375_{10}$ to base 4
d. $10011101.0101_2$ to base 10

Exercise 2. Convert each of the following numbers to 8-bit signed magnitude, 8-bit one’s complement, 8-bit two’s complement and 8-bit excess 128 formats.

a. $(−122)_{10}$
b. $(−31)_{10}$
c. $(−16)_{10}$
d. $127_{10}$

Exercise 3. Find the decimal equivalents for the following 8-bit two’s complement numbers.

a. $1000 0001$
b. $0111 1011$
c. $1111 0001$
d. $0010 1010$

Exercise 4. Perform two’s complement addition on the following pairs of numbers. In each case, indicate whether an overflow has occurred.

a. $1110 1011 + 0110 1001$
b. $1110 1011 + 1111 1111$
c. $1000 1100 + 1100 0001$
d. $0111 1001 + 0000 1001$
Common Sizes for Data Types

- A byte is composed of 8 bits. Two nibbles make up a byte.
- Halfwords, words, doublewords, and quadwords are composed of bytes as shown below:

<table>
<thead>
<tr>
<th>Bit</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nibble</td>
<td>0110</td>
</tr>
<tr>
<td>Byte</td>
<td>10110000</td>
</tr>
<tr>
<td>16-bit word (halfword)</td>
<td>11001001 01000110</td>
</tr>
<tr>
<td>32-bit word</td>
<td>10110100 00110101 10011001 01011000</td>
</tr>
<tr>
<td>64-bit word (double)</td>
<td>01011000 01010101 10110000 11110011 11001110 11101110 01111000 00110101</td>
</tr>
<tr>
<td>128-bit word (quad)</td>
<td>01011000 01010101 10110000 11110011 11001110 11101110 01111000 00110101 00001011 10100110 11110010 11100110 10100100 01000100 10100101 01010001</td>
</tr>
</tbody>
</table>
Big-Endian and Little-Endian Formats

- In a byte-addressable machine, the smallest datum that can be referenced in memory is the byte. Multi-byte words are stored as a sequence of bytes, in which the address of the multi-byte word is the same as the byte of the word that has the lowest address.

- When multi-byte words are used, two choices for the order in which the bytes are stored in memory are: most significant byte at lowest address, referred to as big-endian, or least significant byte stored at lowest address, referred to as little-endian.

Word address is $x$ for both big-endian and little-endian formats.
Two’s Complement Sign Extension

<table>
<thead>
<tr>
<th>Decimal</th>
<th>8-bit</th>
<th>16-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>+5</td>
<td>0000 0101</td>
<td>0000 0000 0000 0101</td>
</tr>
<tr>
<td>-5</td>
<td>1111 1011</td>
<td>1111 1111 1111 1011</td>
</tr>
</tbody>
</table>

**Why does sign extension work?**

- $-x$ is represented as $2^8 - x$ in 8-bit
- $-x$ is represented as $2^{16} - x$ in 16-bit

$2^8 - x + ??? = 2^{16} - x$

$??? = 2^{16} - 2^8$

\[
\begin{align*}
1 & 0000 0000 0000 0000 = 65536 \\
- 1 & 0000 0000 = 256 \\
1111 1111 0000 0000 = 65280
\end{align*}
\]
Multiplication Example

- Multiplication of two 4-bit unsigned binary integers produces an 8-bit result.

\[
\begin{array}{c}
1101 \\
\times 1011 \\
\hline
1101 \\
1101 \\
1101 \\
0000 \\
\hline
10011111 \quad (143)_{10} \quad \text{Product P}
\end{array}
\]

- Multiplicand M: \((13)_{10}\)
- Multiplier Q: \((11)_{10}\)

- Partial products

- Multiplication of two 4-bit signed binary integers produces only a 7-bit result (each operand reduces to a sign bit and a 3-bit magnitude for each operand, producing a sign-bit and a 6-bit result).
Multiplication of Signed Integers

- Sign extension to the target word size is needed for the negative operand(s).
- A target word size of 8 bits is used here for two 4-bit signed operands, but only a 7-bit target word size is needed for the result.

\[
\begin{array}{cccc}
1 & 1 & 1 & 1 \\
\times & 0 & 0 & 0 & 1 \\
\hline
1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
\hline
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1
\end{array}
\]

\((+1)_{10} \times (-1)_{10} = (-15)_{10}\)  \(\text{(Incorrect; result should be } -1)\)

\[
\begin{array}{cccc}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\times & 0 & 0 & 0 & 1 \\
\hline
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & (-1)_{10}
\end{array}
\]

\((+1)_{10} \times (-1)_{10} = (-1)_{10}\)
Example of Base 2 Division

- \((7 / 3 = 2)_{10}\) with a remainder \(R\) of 1.
- Equivalently, \((0111/ 11 = 10)_{2}\) with a remainder \(R\) of 1.

\[
\begin{array}{c}
0 & 0 & 1 & 0 & R & 1 \\
\hline
1 & 1 & 0 & 1 & 1 & 1 \\
1 & 1 & 0 & 1 & 1 & \\
\hline
0 & 1 & \\
\end{array}
\]
Base 10 Floating Point Numbers

- Floating point numbers allow very large and very small numbers to be represented using only a few digits, at the expense of precision. The precision is primarily determined by the number of digits in the fraction (or significand, which has integer and fractional parts), and the range is primarily determined by the number of digits in the exponent.

- Example (+6.023 \times 10^{23}):

```
+ 2 3 6 0 2 3
```

<table>
<thead>
<tr>
<th>Sign</th>
<th>Exponent (two digits)</th>
<th>Significand (four digits)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2 3</td>
<td>6 0 2 3</td>
</tr>
</tbody>
</table>
Normalization

- The base 10 number 254 can be represented in floating point form as $254 \times 10^0$, or equivalently as:
  - $25.4 \times 10^1$, or
  - $2.54 \times 10^2$, or
  - $.254 \times 10^3$, or
  - $.0254 \times 10^4$, or

  infinitely many other ways, which creates problems when making comparisons, with so many representations of the same number.

- Floating point numbers are usually normalized, in which the radix point is located in only one possible position for a given number.

- Usually, but not always, the normalized representation places the radix point immediately to the left of the leftmost, nonzero digit in the fraction, as in: $.254 \times 10^3$. 
Floating Point Example

- Represent \(0.254 \times 10^3\) in a normalized base 8 floating point format with a sign bit, followed by a 3-bit excess 4 exponent, followed by four base 8 digits.

- Step #1: Convert to the target base.
  \[ 0.254 \times 10^3 = 254_{10}. \] Using the remainder method, we find that \(254_{10} = 376 \times 8^0:\)
  \[
  \begin{align*}
  254/8 &= 31 \text{ R } 6 \\
  31/8 &= 3 \text{ R } 7 \\
  3/8 &= 0 \text{ R } 3
  \end{align*}
  \]

- Step #2: Normalize: \(376 \times 8^0 = 0.376 \times 8^3.\)

- Step #3: Fill in the bit fields, with a positive sign (sign bit = 0), an exponent of \(3 + 4 = 7\) (excess 4), and 4-digit fraction = \(0.3760: \)
  
  \[
  0 \ 111 \ . \ 011 \ 111 \ 110 \ 000
  \]
Error, Range, and Precision

• In the previous example, we have the base $b = 8$, the number of significant digits (not bits!) in the fraction $s = 4$, the largest exponent value (not bit pattern) $M = 3$, and the smallest exponent value $m = -4$.

• In the previous example, there is no explicit representation of 0, but there needs to be a special bit pattern reserved for 0 otherwise there would be no way to represent 0 without violating the normalization rule. We will assume a bit pattern of $0 000 000 000 000 000$ represents 0.

• Using $b$, $s$, $M$, and $m$, we would like to characterize this floating point representation in terms of the largest positive representable number, the smallest (nonzero) positive representable number, the smallest gap between two successive numbers, the largest gap between two successive numbers, and the total number of numbers that can be represented.
Error, Range, and Precision (cont’)

- Largest representable number: $b^M \times (1 - b^{-s}) = 8^3 \times (1 - 8^{-4})$

- Smallest representable number: $b^m \times b^{-1} = 8^{-4} \times 1 = 8^{-5}$

- Largest gap: $b^M \times b^{-s} = 8^3 \times 8^{-4} = 8^{-1}$

- Smallest gap: $b^m \times b^{-s} = 8^{-4} \times 8^{-4} = 8^{-8}$
Error, Range, and Precision (cont’)

- Number of representable numbers: There are 5 components: (A) sign bit; for each number except 0 for this case, there is both a positive and negative version; (B) \((M - m) + 1\) exponents; (C) \(b - 1\) values for the first digit (0 is disallowed for the first normalized digit); (D) \(b^{s-1}\) values for each of the \(s-1\) remaining digits, plus (E) a special representation for 0. For this example, the 5 components result in: \(2 \times ((3 - 4) + 1) \times (8 - 1) \times 8^{4-1} + 1\) numbers that can be represented. Notice this number must be no greater than the number of possible bit patterns that can be generated, which is \(2^{16}\).
Example Floating Point Format

- Smallest number is $1/8$
- Largest number is $7/4$
- Smallest gap is $1/32$
- Largest gap is $1/4$
- Number of representable numbers is 33.
Gap Size Follows Exponent Size

- The relative error is approximately the same for all numbers.
- If we take the ratio of a large gap to a large number, and compare that to the ratio of a small gap to a small number, then the ratios are the same:

\[
\frac{b^{M-s}}{b^M \times (1 - b^{-s})} = \frac{b^{-s}}{1 - b^{-s}} = \frac{1}{b^{s-1}}
\]

\[
\frac{b^{m-s}}{b^m \times (1 - b^{-s})} = \frac{b^{-s}}{1 - b^{-s}} = \frac{1}{b^{s-1}}
\]
Conversion Example

- **Example:** Convert \((9.375 \times 10^{-2})_{10}\) to base 2 scientific notation.

- Start by converting from base 10 floating point to base 10 fixed point by moving the decimal point two positions to the left, which corresponds to the -2 exponent: \(.09375\).

- Next, convert from base 10 fixed point to base 2 fixed point:
  
  - \(.09375 \times 2 = 0.1875\)
  - \(.1875 \times 2 = 0.375\)
  - \(.375 \times 2 = 0.75\)
  - \(.75 \times 2 = 1.5\)
  - \(.5 \times 2 = 1.0\)

- Thus, \((.09375)_{10} = (.00011)_{2}\).

- Finally, convert to normalized base 2 floating point:
  
  \[
  .00011 = .00011 \times 2^0 = 1.1 \times 2^{-4}
  \]
IEEE-754 32-bit Floating Point Format

- sign bit, 8-bit exponent, 23-bit mantissa
- normalized as 1.xxxxx
- leading 1 is hidden
- 8-bit exponent in excess 127 format
  - NOT excess 128
  - 0000 0000 and 1111 1111 are reserved
- +0 and -0 is zero exponent and zero mantissa
- 1111 1111 exponent and zero mantissa is infinity
IEEE-754 Floating Point Formats

Single precision
- Sign (1 bit)
- Exponent: 8 bits
- Fraction: 23 bits
- 32 bits total

Double precision
- Sign (1 bit)
- Exponent: 11 bits
- Fraction: 52 bits
- 64 bits total
## IEEE-754 Examples

<table>
<thead>
<tr>
<th>Value</th>
<th>Sign</th>
<th>Exponent</th>
<th>Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) $+1.101 \times 2^5$</td>
<td>0</td>
<td>1000 0100</td>
<td>101 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>(b) $-1.01011 \times 2^{-126}$</td>
<td>1</td>
<td>0000 0001</td>
<td>010 1100 0000 0000 0000 0000</td>
</tr>
<tr>
<td>(c) $+1.0 \times 2^{127}$</td>
<td>0</td>
<td>1111 1110</td>
<td>000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>(d) +0</td>
<td>0</td>
<td>0000 0000</td>
<td>000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>(e) −0</td>
<td>1</td>
<td>0000 0000</td>
<td>000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>(f) +∞</td>
<td>0</td>
<td>1111 1111</td>
<td>000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>(g) $+2^{-128}$</td>
<td>0</td>
<td>0000 0000</td>
<td>010 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>(h) +NaN</td>
<td>0</td>
<td>1111 1111</td>
<td>011 0111 0000 0000 0000 0000</td>
</tr>
<tr>
<td>(i) $+2^{-128}$</td>
<td>0</td>
<td>011 0111 1111</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
</tbody>
</table>
IEEE-754 Conversion Example

• Represent \(-12.625_{10}\) in single precision IEEE-754 format.

• Step #1: Convert to target base. \(-12.625_{10} = -1100.101_2\)

• Step #2: Normalize. \(-1100.101_2 = -1.100101_2 \times 2^3\)

• Step #3: Fill in bit fields. Sign is negative, so sign bit is 1. Exponent is in excess 127 (not excess 128!), so exponent is represented as the unsigned integer \(3 + 127 = 130\). Leading 1 of significand is hidden, so final bit pattern is:

\[
1\ 1000\ 0010\ .\ 1001\ 0100\ 0000\ 0000\ 0000\ 0000
\]
Floating Point Arithmetic

Floating point arithmetic differs from integer arithmetic in that exponents must be handled as well as the magnitudes of the operands.

The exponents of the operands must be made equal for addition and subtraction. The fractions are then added or subtracted as appropriate, and the result is normalized.

Ex: Perform the floating point operation: \((.101 \times 2^3 + .111 \times 2^4)_2\)

Start by adjusting the smaller exponent to be equal to the larger exponent, and adjust the fraction accordingly. Thus we have \(.101 \times 2^3 = .010 \times 2^4\), losing \(.001 \times 2^3\) of precision in the process.

The resulting sum is \((.010 + .111) \times 2^4 = 1.001 \times 2^4 = .1001 \times 2^5\), and rounding to three significant digits, \(.100 \times 2^5\), and we have lost another \(.001 \times 2^4\) in the rounding process.
Floating Point Multiplication/Division

- Floating point multiplication/division are performed in a manner similar to floating point addition/subtraction, except that the sign, exponent, and fraction of the result can be computed separately.

- Like/unlike signs produce positive/negative results, respectively. Exponent of result is obtained by adding exponents for multiplication, or by subtracting exponents for division. Fractions are multiplied or divided according to the operation, and then normalized.

- Ex: Perform the floating point operation: \((+.110 \times 2^5) / (+.100 \times 2^4)\)

  - The source operand signs are the same, which means that the result will have a positive sign. We subtract exponents for division, and so the exponent of the result is \(5 - 4 = 1\).

  - We divide fractions, producing the result: \(110/100 = 1.10\).

  - Putting it all together, the result of dividing \((+.110 \times 2^5)\) by \((+.100 \times 2^4)\) produces \((+1.10 \times 2^1)\). After normalization, the final result is \((+.110 \times 2^2)\).
ASCII Character Code

- ASCII is a 7-bit code, commonly stored in 8-bit bytes.
- “A” is at 41<sub>16</sub>. To convert upper case letters to lower case letters, add 20<sub>16</sub>. Thus “a” is at 41<sub>16</sub> + 20<sub>16</sub> = 61<sub>16</sub>.
- The character “5” at position 35<sub>16</sub> is different than the number 5. To convert character-numbers into number-numbers, subtract 30<sub>16</sub>: 35<sub>16</sub> - 30<sub>16</sub> = 5.

<table>
<thead>
<tr>
<th>ASCII Code</th>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 - 31</td>
<td>NUL - SOH</td>
<td>Control codes</td>
</tr>
<tr>
<td>32</td>
<td>SP</td>
<td>Space</td>
</tr>
<tr>
<td>33 - 126</td>
<td>! - ?</td>
<td>Supplemental</td>
</tr>
<tr>
<td>127 - 255</td>
<td>` - DEL</td>
<td>Supplemental</td>
</tr>
</tbody>
</table>

**ASCII Character Codes:**
- **NUL**: Null
- **SOH**: Start of heading
- **STX**: Start of text
- **ETX**: End of text
- **EOT**: End of transmission
- **ENQ**: Enquiry
- **ACK**: Acknowledge
- **BEL**: Bell
- **BS**: Backspace
- **HT**: Horizontal tab
- **LF**: Line feed
- **VT**: Vertical tab
- **FF**: Form feed
- **CR**: Carriage return
- **SO**: Shift out
- **SI**: Shift in
- **DLE**: Data link escape
- **DC1**: Device control 1
- **DC2**: Device control 2
- **DC3**: Device control 3
- **DC4**: Device control 4
- **NAK**: Negative acknowledge
- **SYN**: Synchronous idle
- **ETB**: End of transmission block
### EBCDIC Character Code

- **EBCDIC** is an 8-bit code.

<table>
<thead>
<tr>
<th>Character Code</th>
<th>ASCII Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 NUL</td>
<td>00</td>
<td>\n</td>
</tr>
<tr>
<td>01 SOH</td>
<td>01</td>
<td>Start of text</td>
</tr>
<tr>
<td>02 STX</td>
<td>02</td>
<td>Reader Stop</td>
</tr>
<tr>
<td>03 ETX</td>
<td>03</td>
<td>Device Control 1</td>
</tr>
<tr>
<td>04 EON</td>
<td>04</td>
<td>Bell</td>
</tr>
<tr>
<td>05 OMC</td>
<td>05</td>
<td>Device Control 2</td>
</tr>
<tr>
<td>06 LCC</td>
<td>06</td>
<td>Space</td>
</tr>
<tr>
<td>07 BAC</td>
<td>07</td>
<td>Backspace</td>
</tr>
<tr>
<td>08 CCL</td>
<td>08</td>
<td>Device Control 4</td>
</tr>
<tr>
<td>09 NUL</td>
<td>09</td>
<td>Idle</td>
</tr>
<tr>
<td>A0 ACK</td>
<td>A0</td>
<td>Acknowledge</td>
</tr>
<tr>
<td>A1 BEL</td>
<td>A1</td>
<td>Punch On</td>
</tr>
<tr>
<td>A2 BS</td>
<td>A2</td>
<td>Customer Use 1</td>
</tr>
<tr>
<td>A3 BS</td>
<td>A3</td>
<td>Customer Use 2</td>
</tr>
<tr>
<td>A4 BS</td>
<td>A4</td>
<td>Customer Use 3</td>
</tr>
<tr>
<td>A5 BS</td>
<td>A5</td>
<td>Customer Use 4</td>
</tr>
<tr>
<td>A6 BS</td>
<td>A6</td>
<td>Customer Use 5</td>
</tr>
<tr>
<td>A7 BS</td>
<td>A7</td>
<td>Customer Use 6</td>
</tr>
<tr>
<td>A8 BS</td>
<td>A8</td>
<td>Customer Use 7</td>
</tr>
<tr>
<td>A9 BS</td>
<td>A9</td>
<td>Customer Use 8</td>
</tr>
<tr>
<td>AA BS</td>
<td>AA</td>
<td>Customer Use 9</td>
</tr>
<tr>
<td>AB BS</td>
<td>AB</td>
<td>Customer Use 10</td>
</tr>
<tr>
<td>AC BS</td>
<td>AC</td>
<td>Customer Use 11</td>
</tr>
<tr>
<td>AD BS</td>
<td>AD</td>
<td>Customer Use 12</td>
</tr>
<tr>
<td>AE BS</td>
<td>AE</td>
<td>Customer Use 13</td>
</tr>
<tr>
<td>AF BS</td>
<td>AF</td>
<td>Customer Use 14</td>
</tr>
<tr>
<td>BI BS</td>
<td>BI</td>
<td>Customer Use 15</td>
</tr>
<tr>
<td>BJ BS</td>
<td>BJ</td>
<td>Customer Use 16</td>
</tr>
<tr>
<td>BK BS</td>
<td>BK</td>
<td>Customer Use 17</td>
</tr>
<tr>
<td>BL BS</td>
<td>BL</td>
<td>Customer Use 18</td>
</tr>
<tr>
<td>BM BS</td>
<td>BM</td>
<td>Customer Use 19</td>
</tr>
<tr>
<td>BN BS</td>
<td>BN</td>
<td>Customer Use 20</td>
</tr>
<tr>
<td>BO BS</td>
<td>BO</td>
<td>Customer Use 21</td>
</tr>
<tr>
<td>BP BS</td>
<td>BP</td>
<td>Customer Use 22</td>
</tr>
<tr>
<td>BQ BS</td>
<td>BQ</td>
<td>Customer Use 23</td>
</tr>
<tr>
<td>BR BS</td>
<td>BR</td>
<td>Customer Use 24</td>
</tr>
<tr>
<td>BS BS</td>
<td>BS</td>
<td>Customer Use 25</td>
</tr>
<tr>
<td>BT BS</td>
<td>BT</td>
<td>Customer Use 26</td>
</tr>
<tr>
<td>BU BS</td>
<td>BU</td>
<td>Customer Use 27</td>
</tr>
<tr>
<td>BV BS</td>
<td>BV</td>
<td>Customer Use 28</td>
</tr>
<tr>
<td>BW BS</td>
<td>BW</td>
<td>Customer Use 29</td>
</tr>
<tr>
<td>BX BS</td>
<td>BX</td>
<td>Customer Use 30</td>
</tr>
<tr>
<td>BY BS</td>
<td>BY</td>
<td>Customer Use 31</td>
</tr>
<tr>
<td>BZ BS</td>
<td>BZ</td>
<td>Customer Use 32</td>
</tr>
<tr>
<td>CA BS</td>
<td>CA</td>
<td>Customer Use 33</td>
</tr>
<tr>
<td>CB BS</td>
<td>CB</td>
<td>Customer Use 34</td>
</tr>
<tr>
<td>CC BS</td>
<td>CC</td>
<td>Customer Use 35</td>
</tr>
<tr>
<td>CD BS</td>
<td>CD</td>
<td>Customer Use 36</td>
</tr>
<tr>
<td>CE BS</td>
<td>CE</td>
<td>Customer Use 37</td>
</tr>
<tr>
<td>CF BS</td>
<td>CF</td>
<td>Customer Use 38</td>
</tr>
<tr>
<td>D0 BS</td>
<td>D0</td>
<td>Customer Use 39</td>
</tr>
<tr>
<td>D1 BS</td>
<td>D1</td>
<td>Customer Use 40</td>
</tr>
<tr>
<td>D2 BS</td>
<td>D2</td>
<td>Customer Use 41</td>
</tr>
<tr>
<td>D3 BS</td>
<td>D3</td>
<td>Customer Use 42</td>
</tr>
<tr>
<td>D4 BS</td>
<td>D4</td>
<td>Customer Use 43</td>
</tr>
<tr>
<td>D5 BS</td>
<td>D5</td>
<td>Customer Use 44</td>
</tr>
<tr>
<td>D6 BS</td>
<td>D6</td>
<td>Customer Use 45</td>
</tr>
<tr>
<td>D7 BS</td>
<td>D7</td>
<td>Customer Use 46</td>
</tr>
<tr>
<td>D8 BS</td>
<td>D8</td>
<td>Customer Use 47</td>
</tr>
<tr>
<td>D9 BS</td>
<td>D9</td>
<td>Customer Use 48</td>
</tr>
<tr>
<td>DA BS</td>
<td>DA</td>
<td>Customer Use 49</td>
</tr>
<tr>
<td>DB BS</td>
<td>DB</td>
<td>Customer Use 50</td>
</tr>
<tr>
<td>DC BS</td>
<td>DC</td>
<td>Customer Use 51</td>
</tr>
<tr>
<td>DD BS</td>
<td>DD</td>
<td>Customer Use 52</td>
</tr>
<tr>
<td>DE BS</td>
<td>DE</td>
<td>Customer Use 53</td>
</tr>
<tr>
<td>DF BS</td>
<td>DF</td>
<td>Customer Use 54</td>
</tr>
<tr>
<td>E0 BS</td>
<td>E0</td>
<td>Customer Use 55</td>
</tr>
<tr>
<td>E1 BS</td>
<td>E1</td>
<td>Customer Use 56</td>
</tr>
<tr>
<td>E2 BS</td>
<td>E2</td>
<td>Customer Use 57</td>
</tr>
<tr>
<td>E3 BS</td>
<td>E3</td>
<td>Customer Use 58</td>
</tr>
<tr>
<td>E4 BS</td>
<td>E4</td>
<td>Customer Use 59</td>
</tr>
<tr>
<td>E5 BS</td>
<td>E5</td>
<td>Customer Use 60</td>
</tr>
<tr>
<td>E6 BS</td>
<td>E6</td>
<td>Customer Use 61</td>
</tr>
<tr>
<td>E7 BS</td>
<td>E7</td>
<td>Customer Use 62</td>
</tr>
<tr>
<td>E8 BS</td>
<td>E8</td>
<td>Customer Use 63</td>
</tr>
<tr>
<td>E9 BS</td>
<td>E9</td>
<td>Customer Use 64</td>
</tr>
</tbody>
</table>

---

Principles of Computer Architecture by M. Murdocca and V. Heuring © 1999 M. Murdocca and V. Heuring
Unicode Character Code

- Unicode is a 16-bit code.
Next Time

- Basic Architecture of Intel Pentium Chip
- “Hello World” in Linux Assembly
- Addressing Modes