Terminology

- Computers store & process *data*
- Data is anything that has some interest to the user
- Thoughtfully processed data is called *information*
- Physical devices used to store & process data in computers are 2-state devices
- State is represented using;
  - 0 - OFF
  - 1 - ON

Bit & Byte

- An abbreviation for *BInary digIT*
- Bit is the smallest unit of data representation
- When 8 bits are combined together it is called a *byte*
- 1024 (2^{10}) bytes are called a *Kilobyte*
Terminology

- Computers represent numbers as a series of switches which store a pattern of ON's & OFF's, representing the binary digits 1 & 0.
- Each of these digits is called a BIT (BIinary digit)

```
  7  6  5  4  3  2  1  0
  1  1  0  1  1  0  0  1
```

Terminology cont...

```
  7  6  5  4  3  2  1  0
  1  1  0  1  1  0  0  1
```

- MSB – Most Significant Bit
- LSB – Least Significant Bit
Metrics used to measure quantities of data

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Symbol</th>
<th>Bytes</th>
<th>Power of 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit</td>
<td>Bit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Byte</td>
<td>Byte</td>
<td>8 bits</td>
<td>$2^8$</td>
</tr>
<tr>
<td>Kilobyte</td>
<td>KB</td>
<td>1024 Bytes</td>
<td>$2^{10}$</td>
</tr>
<tr>
<td>Megabyte</td>
<td>MB</td>
<td>1024 Kilobytes</td>
<td>$2^{20}$</td>
</tr>
<tr>
<td>Gigabyte</td>
<td>GB</td>
<td>1024 Megabytes</td>
<td>$2^{30}$</td>
</tr>
<tr>
<td>Terabyte</td>
<td>TB</td>
<td>1024 Gigabyte</td>
<td>$2^{40}$</td>
</tr>
<tr>
<td>Kilobit</td>
<td>Kb</td>
<td>1000 bits</td>
<td>$2^{10}$</td>
</tr>
<tr>
<td>Megabit</td>
<td>Mb</td>
<td>1000 Kilobits</td>
<td>$2^{20}$</td>
</tr>
</tbody>
</table>

© CSE
Humans vs. Computers

- It is not surprising that our number system is based on units of **TEN**, since nature provided man with five fingers on each of 2 hands.

- Computers use **binary**
  - They have only electronic or electromechanical switches.
  - These switches have only 2 states, either ON or OFF

Distinguishing Number Systems

- The characteristic which distinguish one number system from another is called the **base** (or **radix**)

- The base or radix, of a number system is:
  - the number of different digits that can occur in each position in the number system.
  - this is the number of symbols in a system
  - Example: Base 10 - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
**Number Bases**

<table>
<thead>
<tr>
<th>Number System</th>
<th>Base</th>
<th>Symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
<td>2</td>
<td>0, 1</td>
</tr>
<tr>
<td>Octal</td>
<td>8</td>
<td>0, 1, 2, 3, 4, 5, 6, 7</td>
</tr>
<tr>
<td>Decimal</td>
<td>10</td>
<td>0, 1, 2, 3, 4, 5, 6, 7, 8, 9</td>
</tr>
<tr>
<td>Hexadecimal</td>
<td>16</td>
<td>0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F</td>
</tr>
</tbody>
</table>

**Decimal Number System**

- The number 4567 in base 10 means:

  \[4567 = \text{four thousand five hundred and sixty seven} = 4000 + 500 + 60 + 7 = (4 \times 10^3) + (5 \times 10^2) + (6 \times 10^1) + (7 \times 10^0)\]
Positional notations

- In general, the relationship between a digit, its position & the base of the number system is expressed by the following formula:

\[ \text{DIGIT} \times \text{BASE}^{\text{POSITION} \#} \]

Weight Associated with Digit Position

When expressed as a series of digits, the value of a number is determined by considering the "weight" associated with the position of each digit.
The number 512.49 in base 10 means:

\[ 512.49 = 500 + 10 + 2 + 0.40 + 0.09 \]
\[ = (5 \times 100) + (1 \times 10) + (2 \times 1) + (4 \times 0.1) + (9 \times 0.001) \]
\[ = (5 \times 10^2) + (1 \times 10^1) + (2 \times 10^0) + (4 \times 10^{-1}) + (9 \times 10^{-2}) \]

Binary number system

- Base of the binary number system is 2
- Symbols are 0 & 1

The number 1101.11 in base 2 means:

\[ 1101.01 \]
\[ = (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) + (0 \times 2^{-1}) + (1 \times 2^{-2}) \]
Octal number system

- Base of the octal number system is 8
- The symbols are 0, 1, 2, 3, 4, 5, 6 & 7

- The number 456.41 in base 8 means:
  
  $475.01 = (4 \times 8^2) + (7 \times 8^1) + (5 \times 8^0) + (0 \times 8^{-1}) + (1 \times 8^{-2})$

Hexadecimal number system

- The base is 16
- The symbols are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

- The number 1FA.4C in base 16 means:
  
  $1FA.4C = (1 \times 16^2) + (F \times 16^1) + (A \times 16^0) + (4 \times 16^{-1}) + (C \times 16^{-2})$
  
  $= (1 \times 16^2) + (15 \times 16^1) + (10 \times 16^0) + (4 \times 16^{-1}) + (12 \times 16^{-2})$
Decimal to Binary conversion - Integers

- Convert $9_{10}$ into binary
  
  $\begin{align*}
  9/2 &= 4 \quad r = 1 \\
  4/2 &= 2 \quad r = 0 \\
  2/2 &= 1 \quad r = 0 \\
  1/2 &= 0 \quad r = 1 \\
\end{align*}$

- $9_{10} = 1001_2$

Decimal to Binary – Fractions

- Example 1: Represent $0.5_{10}$ in binary
  
  $0.5 \times 2 = 1.0$
  
  $0.0 \times 2$ (this ends the process)
  
  So $0.5_{10} = 0.1_2$

- Example 2: Represent $0.1_{10}$ in binary
  
  - $0.1_{10} = 0.000110011001_2$
  - $= 0.00011101$
Binary to Decimal conversion - Integers

- Example 3: Convert $11101.01_2$ to decimal

- We will calculate the position value for each binary digit & then sum these values
- We write the position values above each digit.

<table>
<thead>
<tr>
<th>Position Values</th>
<th>$2^4$</th>
<th>$2^3$</th>
<th>$2^2$</th>
<th>$2^1$</th>
<th>$2^0$</th>
<th>$2^{-1}$</th>
<th>$2^{-2}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary Digits</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0.5</td>
<td>0.25</td>
</tr>
</tbody>
</table>

Example 3: cont…

- $11101.01$
  
  $= (1 \times 2^4) + (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) + (0 \times 2^{-1}) + (1 \times 2^{-2})$
  
  $= (1 \times 16) + (1 \times 8) + (1 \times 4) + (0 \times 2) + (1 \times 1) + (0 \times 0.5) + (1 \times 0.25)$
  
  $= 16 + 8 + 4 + 0 + 1 + 0 + 0.25$
  
  $= 29.25$
Decimal to Octal conversion

- Represent $31_{10}$ in octal

\[
\begin{align*}
31/8 &= 3 \quad r = 7 \\
3/8 &= 0 \quad r = 3
\end{align*}
\]

- $31_{10} = 37_8$

Binary to Octal conversion

- Example 6: convert $11010011_2$ in to octal

11 | 010 | 011
011 | 010 | 011 - add extra bit
3 2 3

$11010011_2 = 323_8$
Binary to Hexadecimal conversion

- Convert 10011101 into hexadecimal

\[
\begin{array}{ccc}
1001 & | & 1101 \\
9 & & 13 \\
9 & & D \\
\end{array}
\]

\[10011101_2 = 9D_8\]

Character representation

- Belong to the category of qualitative data
- Represent quality or characteristics
- Also called non-numeric values
- Includes
  - Letters: a-z, A-Z
  - Digits: 0-9
  - Symbols: !, @, *, /, &, #, $
  - Control characters: <CR>, <BEL>, <ESC>, <LF>
Character representation cont …

- With a single byte (8-bits) 256 characters can be represented

- Standards
  - ASCII - American Standard Code for Information Interchange
  - EBCDIC - Extended Binary-Coded Decimal Interchange Code
  - Unicode

ASCII

- De facto world-wide standard for the code numbers used by computers to represent:
  - upper & lower-case Latin letters
  - Numbers
  - Punctuations

- There are 128 standard ASCII codes
  - Can be represented by a 7 digit binary number: 000 0000 through 111 1111
### ASCII Table

<table>
<thead>
<tr>
<th>ASCII</th>
<th>Hex</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>NUL</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>SOH</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>STX</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>ETX</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>EOT</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>ENQ</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>ACK</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>BEL</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>BS</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>TAB</td>
</tr>
<tr>
<td>10</td>
<td>A</td>
<td>LF</td>
</tr>
<tr>
<td>11</td>
<td>B</td>
<td>VT</td>
</tr>
<tr>
<td>12</td>
<td>C</td>
<td>FF</td>
</tr>
<tr>
<td>13</td>
<td>D</td>
<td>CR</td>
</tr>
<tr>
<td>14</td>
<td>E</td>
<td>SO</td>
</tr>
<tr>
<td>15</td>
<td>F</td>
<td>SI</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ASCII</th>
<th>Hex</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>10</td>
<td>ECH</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>ESC</td>
</tr>
<tr>
<td>18</td>
<td>12</td>
<td>FS</td>
</tr>
<tr>
<td>19</td>
<td>13</td>
<td>GS</td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>RS</td>
</tr>
<tr>
<td>21</td>
<td>15</td>
<td>US</td>
</tr>
<tr>
<td>22</td>
<td>22</td>
<td>XON</td>
</tr>
<tr>
<td>23</td>
<td>23</td>
<td>XOFF</td>
</tr>
<tr>
<td>24</td>
<td>24</td>
<td>CAN</td>
</tr>
<tr>
<td>25</td>
<td>25</td>
<td>STRIP</td>
</tr>
<tr>
<td>26</td>
<td>26</td>
<td>ETB</td>
</tr>
<tr>
<td>27</td>
<td>27</td>
<td>MFG</td>
</tr>
</tbody>
</table>

### ASCII Table Cont...

<table>
<thead>
<tr>
<th>ASCII</th>
<th>Hex</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>20</td>
<td>(space)</td>
</tr>
<tr>
<td>33</td>
<td>21</td>
<td>!</td>
</tr>
<tr>
<td>34</td>
<td>22</td>
<td>&quot;</td>
</tr>
<tr>
<td>35</td>
<td>23</td>
<td>#</td>
</tr>
<tr>
<td>36</td>
<td>24</td>
<td>$</td>
</tr>
<tr>
<td>37</td>
<td>25</td>
<td>%</td>
</tr>
<tr>
<td>38</td>
<td>26</td>
<td>&amp;</td>
</tr>
<tr>
<td>39</td>
<td>27</td>
<td>'</td>
</tr>
<tr>
<td>40</td>
<td>28</td>
<td>(</td>
</tr>
<tr>
<td>41</td>
<td>29</td>
<td>)</td>
</tr>
<tr>
<td>42</td>
<td>2A</td>
<td>*</td>
</tr>
<tr>
<td>43</td>
<td>2B</td>
<td>+</td>
</tr>
<tr>
<td>44</td>
<td>2C</td>
<td>,</td>
</tr>
<tr>
<td>45</td>
<td>2D</td>
<td>-</td>
</tr>
<tr>
<td>46</td>
<td>2E</td>
<td>.</td>
</tr>
<tr>
<td>47</td>
<td>2F</td>
<td>/</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ASCII</th>
<th>Hex</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>48</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>49</td>
<td>31</td>
<td>1</td>
</tr>
<tr>
<td>50</td>
<td>32</td>
<td>2</td>
</tr>
<tr>
<td>51</td>
<td>33</td>
<td>3</td>
</tr>
<tr>
<td>52</td>
<td>34</td>
<td>4</td>
</tr>
<tr>
<td>53</td>
<td>35</td>
<td>5</td>
</tr>
<tr>
<td>54</td>
<td>36</td>
<td>6</td>
</tr>
<tr>
<td>55</td>
<td>37</td>
<td>7</td>
</tr>
<tr>
<td>56</td>
<td>38</td>
<td>8</td>
</tr>
<tr>
<td>57</td>
<td>39</td>
<td>9</td>
</tr>
<tr>
<td>58</td>
<td>3A</td>
<td>:</td>
</tr>
<tr>
<td>59</td>
<td>3B</td>
<td>;</td>
</tr>
<tr>
<td>60</td>
<td>3C</td>
<td>&lt;</td>
</tr>
<tr>
<td>61</td>
<td>3D</td>
<td>=</td>
</tr>
<tr>
<td>62</td>
<td>3E</td>
<td>&gt;</td>
</tr>
<tr>
<td>63</td>
<td>3F</td>
<td>?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ASCII</th>
<th>Hex</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>40</td>
<td>@</td>
</tr>
<tr>
<td>65</td>
<td>41</td>
<td>A</td>
</tr>
<tr>
<td>66</td>
<td>42</td>
<td>B</td>
</tr>
<tr>
<td>67</td>
<td>43</td>
<td>C</td>
</tr>
<tr>
<td>68</td>
<td>44</td>
<td>D</td>
</tr>
<tr>
<td>69</td>
<td>45</td>
<td>E</td>
</tr>
<tr>
<td>70</td>
<td>46</td>
<td>F</td>
</tr>
<tr>
<td>71</td>
<td>47</td>
<td>G</td>
</tr>
<tr>
<td>72</td>
<td>48</td>
<td>H</td>
</tr>
<tr>
<td>73</td>
<td>49</td>
<td>I</td>
</tr>
<tr>
<td>74</td>
<td>4A</td>
<td>J</td>
</tr>
<tr>
<td>75</td>
<td>4B</td>
<td>K</td>
</tr>
<tr>
<td>76</td>
<td>4C</td>
<td>L</td>
</tr>
<tr>
<td>77</td>
<td>4D</td>
<td>M</td>
</tr>
<tr>
<td>78</td>
<td>4E</td>
<td>N</td>
</tr>
<tr>
<td>79</td>
<td>4F</td>
<td>O</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ASCII</th>
<th>Hex</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
<td>50</td>
<td>P</td>
</tr>
<tr>
<td>81</td>
<td>51</td>
<td>Q</td>
</tr>
<tr>
<td>82</td>
<td>52</td>
<td>R</td>
</tr>
<tr>
<td>83</td>
<td>53</td>
<td>S</td>
</tr>
<tr>
<td>84</td>
<td>54</td>
<td>T</td>
</tr>
<tr>
<td>85</td>
<td>55</td>
<td>U</td>
</tr>
<tr>
<td>86</td>
<td>56</td>
<td>V</td>
</tr>
<tr>
<td>87</td>
<td>57</td>
<td>W</td>
</tr>
<tr>
<td>88</td>
<td>58</td>
<td>X</td>
</tr>
<tr>
<td>89</td>
<td>59</td>
<td>Y</td>
</tr>
<tr>
<td>90</td>
<td>5A</td>
<td>Z</td>
</tr>
<tr>
<td>91</td>
<td>5B</td>
<td>[</td>
</tr>
<tr>
<td>92</td>
<td>5C</td>
<td>\</td>
</tr>
<tr>
<td>93</td>
<td>5D</td>
<td>]</td>
</tr>
<tr>
<td>94</td>
<td>5E</td>
<td>^</td>
</tr>
<tr>
<td>95</td>
<td>5F</td>
<td>_</td>
</tr>
<tr>
<td>96</td>
<td>60</td>
<td>`</td>
</tr>
<tr>
<td>97</td>
<td>61</td>
<td>a</td>
</tr>
<tr>
<td>98</td>
<td>62</td>
<td>b</td>
</tr>
<tr>
<td>99</td>
<td>63</td>
<td>c</td>
</tr>
<tr>
<td>100</td>
<td>64</td>
<td>d</td>
</tr>
<tr>
<td>101</td>
<td>65</td>
<td>e</td>
</tr>
<tr>
<td>102</td>
<td>66</td>
<td>f</td>
</tr>
<tr>
<td>103</td>
<td>67</td>
<td>g</td>
</tr>
<tr>
<td>104</td>
<td>68</td>
<td>h</td>
</tr>
<tr>
<td>105</td>
<td>69</td>
<td>i</td>
</tr>
<tr>
<td>106</td>
<td>6A</td>
<td>j</td>
</tr>
<tr>
<td>107</td>
<td>6B</td>
<td>k</td>
</tr>
<tr>
<td>108</td>
<td>6C</td>
<td>l</td>
</tr>
<tr>
<td>109</td>
<td>6D</td>
<td>m</td>
</tr>
<tr>
<td>110</td>
<td>6E</td>
<td>n</td>
</tr>
<tr>
<td>111</td>
<td>6F</td>
<td>o</td>
</tr>
</tbody>
</table>
ASCII - things to note

- ASCII codes for digits are not equal to the numeric value
- A simple code set & supported by almost every computer
- But, most languages need more than 128 characters

EBCDIC

- Use 8-bits
- Used by large IBM computer (main frames)
- Note
  - IBM PC uses only ASCII
CS101 - Computer Systems

By:

Dilum Bandara
Dept. of Computer Science & Engineering
University of Moratuwa
System Organization
To write even a modest 80x86 assembly language program requires considerable familiarity with the 80x86 family. To write good assembly language programs requires a strong knowledge of the underlying hardware. Unfortunately, the underlying hardware is not consistent. Techniques that are crucial for 8088 programs may not be useful on 80486 systems. Likewise, programming techniques that provide big performance boosts on an 80486 chip may not help at all on an 80286. Fortunately, some programming techniques work well whatever microprocessor you’re using. This chapter discusses the effect hardware has on the performance of computer software.

Overview
This chapter describes the basic components that make up a computer system: the CPU, memory, I/O, and the bus that connects them. Although you can write software that is ignorant of these concepts, high performance software requires a complete understanding of this material. This chapter begins by discussing bus organization and memory organization. These two hardware components will probably have a bigger performance impact on your software than the CPU’s speed. Understanding the organization of the system bus will allow you to design data structures that operate at maximum speed. Similarly, knowing about memory performance characteristics, data locality, and cache operation can help you design software that runs as fast as possible. Of course, if you’re not interested in writing code that runs as fast as possible, you can skip this discussion; however, most people do care about speed at one point or another, so learning this information is useful. Unfortunately, the 80x86 family microprocessors are a complex group and often overwhelm beginning students. Therefore, this chapter describes four hypothetical members of the 80x86 family: the 886, 8286, the 8486, and the 8686 microprocessors. These represent simplified versions of the 80x86 chips and allow a discussion of various architectural features without getting bogged down by huge CISC instruction sets. This text
uses the x86 hypothetical processors to describe the concepts of instruction encoding, addressing modes, sequential execution, the prefetch queue, pipelining, and superscalar operation. Once again, these are concepts you do not need to learn if you only want to write correct software. However, if you want to write fast software as well, especially on advanced processors like the 80486, Pentium, and beyond, you will need to learn about these concepts. Some might argue that this chapter gets too involved with computer architecture. They feel such material should appear in an architectural book, not an assembly language programming book. This couldn’t be farther from the truth! Writing good assembly language programs requires a strong knowledge of the architecture. Hence the emphasis on computer architecture in this chapter.

**The Basic System Components**
The basic operational design of a computer system is called its architecture. John Von Neumann, a pioneer in computer design, is given credit for the architecture of most computers in use today. For example, the 80x86 family uses the Von Neumann architecture (VNA). A typical Von Neumann system has three major components: the central processing unit (or CPU), memory, and input/output (or I/O). The way a system designer combines these components impacts system performance (see Figure 3.1). In VNA machines, like the 80x86 family, the CPU is where all the action takes place. All computations occur inside the CPU. Data and CPU instructions reside in memory until required by the CPU. To the CPU, most I/O devices look like memory because the
CPU can store data to an output device and read data from an input device. The major difference between memory and I/O locations is the fact that I/O locations are generally associated with external devices in the outside world.

The System Bus
The system bus connects the various components of a VNA machine. The 80x86 family has three major busses: the address bus, the data bus, and the control bus. A bus is a collection of wires on which electrical signals pass between components in the system. These busses vary from processor to processor. However, each bus carries comparable information on all processors; e.g., the data bus may have a different implementation on the 80386 than on the 8088, but both carry data between the processor, I/O, and memory. A typical 80x86 system component uses standard TTL logic levels. This means each wire on a bus uses a standard voltage level to represent zero and one. We will always specify zero and one rather than the electrical levels because these levels vary on different processors (especially laptops).

The Data Bus
The 80x86 processors use the data bus to shuffle data between the various components in a computer system. The size of this bus varies widely in the 80x86 family. Indeed, this bus defines the “size” of the processor. On typical 80x86 systems, the data bus contains eight, 16, 32, or 64 lines. The 8088 and 80188 microprocessors have an eight bit data bus (eight data lines).
The 8086, 80186, 80286, and 80386SX processors have a 16 bit data bus. The 80386DX, 80486, and Pentium Overdrive processors have a 32 bit data bus. The Pentium and Pentium Pro processors have a 64 bit data bus. Future versions of the chip (the 80686/80786?) may have a larger bus. Having an eight bit data bus does not limit the processor to eight bit data types. It simply means that the processor can only access one byte of data per memory cycle (see

1. TTL logic represents the value zero with a voltage in the range 0.0-0.8v. It represents a one with a voltage in the range 2.4-5v. If the signal on a bus line is between 0.8v and 2.4v, it’s value is indeterminate. Such a condition should only exist when a bus line is changing from one state to the other.

The “Size” of a Processor
There has been a considerable amount of disagreement among hardware and software engineers concerning the “size” of a processor like the 8088. From a hardware designer’s perspective, the 8088 is purely an eight bit processor – it has only eight data lines and is bus compatible with memory and I/O devices designed for eight bit processors. Software engineers, on the other hand, have argued that the 8088 is a 16 bit processor. From their perspective they cannot distinguish between the 8088 (with an eight-bit data bus) and the 8086 (which has a 16-bit data bus). Indeed, the only difference is the speed at which the two processors operate; the 8086 with a 16 bit data bus is faster. Eventually, the hardware designers won out. Despite the fact that software engineers cannot differentiate the 8088 and 8086 in their programs, we call the 8088 an eight bit processor and the 8086 a 16 bit processor. Likewise, the 80386SX (which has a sixteen bit data bus) is a 16 bit...
processor while the 80386DX (which has a full 32 bit data bus) is a 32 bit processor.

“The Memory Subsystem” on page 87 for a description of memory cycles). Therefore, the eight bit bus on an 8088 can only transmit half the information per unit time (memory cycle) as the 16 bit bus on the 8086. Therefore, processors with a 16 bit bus are naturally faster than processors with an eight bit bus. Likewise, processors with a 32 bit bus are faster than those with a 16 or eight bit data bus. The size of the data bus affects the performance of the system more than the size of any other bus. You’ll often hear a processor called an eight, 16, 32, or 64 bit processor. While there is a mild controversy concerning the size of a processor, most people now agree that the number of data lines on the processor determines its size. Since the 80x86 family busses are eight, 16, 32, or 64 bits wide, most data accesses are also eight, 16, 32, or 64 bits. Although it is possible to process 12 bit data with an 8088, most programmers process 16 bits since the processor will fetch and manipulate 16 bits anyway. This is because the processor always fetches eight bits. To fetch 12 bits requires two eight bit memory operations. Since the processor fetches 16 bits rather than 12, most programmers use all 16 bits. In general, manipulating data which is eight, 16, 32, or 64 bits in length is the most efficient. Although the 16, 32, and 64 bit members of the 80x86 family can process data up to the width of the bus, they can also access smaller memory units of eight, 16, or 32 bits. Therefore, anything you can do with a small data bus can be done with a larger data bus as well; the larger data bus, however, may access memory faster and can access larger chunks of data in one memory operation. You’ll read about the exact nature of these memory accesses a little later (see “The Memory Subsystem” on page 87).
### The Address Bus

The data bus on an 80x86 family processor transfers information between a particular memory location or I/O device and the CPU. The only question is, "Which memory location or I/O device?" The address bus answers that question. To differentiate memory locations and I/O devices, the system designer assigns a unique memory address to each memory element and I/O device. When the software wants to access some particular memory location or I/O device, it places the corresponding address on the address bus. Circuitry associated with the memory or I/O device recognizes this address and instructs the memory or I/O device to read the data from or place data on the data bus. In either case, all other memory locations ignore the request. Only the device whose address matches the value on the address bus responds. With a single address line, a processor could create exactly two unique addresses: zero and one. With \( n \) address lines, the processor can provide \( 2^n \) unique addresses (since there are \( 2^n \) unique values in

---

### Table 17: 80x86 Processor Data Bus Sizes

<table>
<thead>
<tr>
<th>Processor</th>
<th>Data Bus Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>8088</td>
<td>8</td>
</tr>
<tr>
<td>80188</td>
<td>8</td>
</tr>
<tr>
<td>8086</td>
<td>16</td>
</tr>
<tr>
<td>80186</td>
<td>16</td>
</tr>
<tr>
<td>80286</td>
<td>16</td>
</tr>
<tr>
<td>80386sx</td>
<td>16</td>
</tr>
<tr>
<td>80386dx</td>
<td>32</td>
</tr>
<tr>
<td>80486</td>
<td>32</td>
</tr>
<tr>
<td>80586 class/ Pentium (Pro)</td>
<td>64</td>
</tr>
</tbody>
</table>
an $n$-bit binary number). Therefore, the number of bits on the address bus will determine the maximum number of addressable memory and I/O locations. The 8088 and 8086, for example, have 20 bit address busses. Therefore, they can access up to $1,048,576$ (or $2^{20}$) memory locations. Larger address busses can access more memory. The 8088 and 8086, for example, suffer from an anemic address space – their address bus is too small. Later processors have larger address busses: Future 80x86 processors will probably support 48 bit address busses. The time is coming when most programmers will consider four gigabytes of storage to be too small, much like they consider one megabyte insufficient today. (There was a time when one megabyte was considered far more than anyone would ever need!) Fortunately, the architecture of the 80386, 80486, and later chips allow for an easy expansion to a 48 bit address bus through segmentation.

3.1.1.3 The Control Bus
The control bus is an eclectic collection of signals that control how the processor communicates with the rest of the system. Consider for a moment the data bus. The CPU sends data to memory and receives data from memory on the data bus. This prompts the question, “Is it sending or receiving?” There are two lines on the control bus, read and write, which specify the direction of data flow. Other signals include system clocks, interrupt lines, status lines, and so on. The exact make up of the control bus varies among pro-

2. The address space is the set of all addressable memory locations.
Table 18: 80x86 Family Address Bus Sizes

<table>
<thead>
<tr>
<th>Processor</th>
<th>Address Bus Size</th>
<th>Max Addressable Memory</th>
<th>In English!</th>
</tr>
</thead>
<tbody>
<tr>
<td>8088</td>
<td>20</td>
<td>1,048,576</td>
<td>One Megabyte</td>
</tr>
<tr>
<td>8086</td>
<td>20</td>
<td>1,048,576</td>
<td>One Megabyte</td>
</tr>
<tr>
<td>80188</td>
<td>20</td>
<td>1,048,576</td>
<td>One Megabyte</td>
</tr>
<tr>
<td>80186</td>
<td>20</td>
<td>1,048,576</td>
<td>One Megabyte</td>
</tr>
<tr>
<td>80286</td>
<td>24</td>
<td>16,777,216</td>
<td>Sixteen Megabytes</td>
</tr>
<tr>
<td>80386sx</td>
<td>24</td>
<td>16,777,216</td>
<td>Sixteen Megabytes</td>
</tr>
<tr>
<td>80386dx</td>
<td>32</td>
<td>4,294,976,296</td>
<td>Four Gigabytes</td>
</tr>
<tr>
<td>80486</td>
<td>32</td>
<td>4,294,976,296</td>
<td>Four Gigabytes</td>
</tr>
<tr>
<td>80586 / Pentium (Pro)</td>
<td>32</td>
<td>4,294,976,296</td>
<td>Four Gigabytes</td>
</tr>
</tbody>
</table>

Future 80x86 processors will probably support 48 bit address busses. The time is coming when most programmers will consider four gigabytes of storage to be too small, much like they consider one megabyte insufficient today. (There was a time when one megabyte was considered far more than anyone would ever need!) Fortunately, the architecture of the 80386, 80486, and later chips allow for an easy expansion to a 48 bit address bus through segmentation.

The Control Bus

The control bus is an eclectic collection of signals that control how the processor communicates with the rest of the system. Consider for a moment the data bus. The CPU sends data to memory and receives data from memory on the data bus. This prompts the question, “Is it sending or receiving?” There are two lines on the control bus, read and write, which specify the direction of data flow. Other signals include system clocks, interrupt lines, status lines, and so on. The exact make up of the control bus varies among processors in the 80x86 family. However, some control lines are common to all processors and are worth a brief mention. The read and write control lines control the direction of data on the data bus. When both contain a logic one, the CPU and memory-I/O are not communicating with one another. If the read line is low (logic zero), the CPU is reading data from memory (that is, the system is transferring
data from memory to the CPU). If the write line is low, the system transfers data from the CPU to memory. The byte enable lines are another set of important control lines. These control lines allow 16, 32, and 64 bit processors to deal with smaller chunks of data. Additional details appear in the next section.

The 80x86 family, unlike many other processors, provides two distinct address spaces: one for memory and one for I/O. While the memory address busses on various 80x86 processors vary in size, the I/O address bus on all 80x86 CPUs is 16 bits wide. This allows the processor to address up to 65,536 different I/O locations. As it turns out, most devices (like the keyboard, printer, disk drives, etc.) require more than one I/O location. Nonetheless, 65,536 I/O locations are more than sufficient for most applications. The original IBM PC design only allowed the use of 1,024 of these. Although the 80x86 family supports two address spaces, it does not have two address busses (for I/O and memory). Instead, the system shares the address bus for both I/O and memory addresses. Additional control lines decide whether the address is intended for memory or I/O. When such signals are active, the I/O devices use the address on the L.O. 16 bits of the address bus. When inactive, the I/O devices ignore the signals on the address bus (the memory subsystem takes over at that point).

The Memory Subsystem
A typical 80x86 processor addresses a maximum of $2^n$ different memory locations, where $n$ is the number of bits on the address bus. As you’ve seen already, 80x86 processors have 20, 24, and 32 bit address busses (with 48 bits on the way). Of course, the first question you should ask is, “What exactly is a memory location?” The 80x86 supports byte addressable memory. Therefore, the basic memory unit is a byte. So with 20, 24, and 32 address lines, the 80x86 processors can address one megabyte, 16 megabytes, and four gigabytes of memory, respectively. Think of memory as a linear array of bytes. The address of the first byte is zero and the address of the last byte is $2^n-1$. For an 8088 with a 20 bit address bus, the following pseudo-Pascal array declaration is a good approximation of
memory: Memory: array [0..1048575] of byte; To execute the equivalent of the Pascal statement “Memory [125] := 0;” the CPU places the value zero on the data bus, the address 125 on the address bus, and asserts the write line (since the CPU is writing data to memory, see Figure 3.2) To execute the equivalent of “CPU := Memory [125];” the CPU places the address 125 on the address bus, asserts the read line (since the CPU is reading data from memory), and then reads the resulting data from the data bus (see Figure 3.2). The above discussion applies only when accessing a single byte in memory. So what happens when the processor accesses a word or a double word? Since memory consists of an array of bytes, how can we possibly deal with values larger than eight bits? Different computer systems have different solutions to this problem. The 80x86 family deals with this problem by storing the L.O. byte of a word at the address specified and the H.O. byte at the next location. Therefore, a word consumes two consecutive memory addresses (as you would expect, since a word consists of two bytes). Similarly, a double word consumes four consecutive memory locations. The address for the double word is the
address of its L.O. byte. The remaining three bytes follow this L.O. byte, with the H.O. byte appearing at the address of the double word plus three (see Figure 3.4) Bytes, words, and double words may begin at any valid address in memory. We will soon see, however, that starting larger objects at an arbitrary address is not a good idea. Note that it is quite possible for byte, word, and double word values to overlap in memory. For example, in Figure 3.4 you could have a word variable beginning at address 193, a byte variable at address 194, and a double word value beginning at address 192. These variables would all overlap. The 8088 and 80188 microprocessors have an eight bit data bus. This means that the CPU can transfer eight bits of data at a time. Since each memory address corresponds to an eight bit byte, this turns out to be the most convenient arrangement (from the hardware perspective), see Figure 3.5. The term “byte addressable memory array” means that the CPU can address memory in chunks as small as a single byte. It also means that this is the smallest unit of memory you can access at once with the processor. That is, if the processor wants to access a four bit value, it must read eight bits and then ignore the extra four bits. Also realize that byte addressability does not imply that the CPU can access eight bits on any arbitrary bit boundary. When you specify address 125 in memory, you get the entire eight bits at that address, nothing less, nothing more. Addresses are integers; you cannot, for example, specify address 125.5 to fetch fewer than eight bits. The 8088 and 80188 can manipulate word and double word values, even with their eight bit data bus. However, this requires multiple memory operations because these processors can only move eight bits of data at once. To load a word requires two memory operations; to load a double word requires four memory operations.
The 8086, 80186, 80286, and 80386sx processors have a 16 bit data bus. This allows these processors to access twice as much memory in the same amount of time as their eight bit brethren. These processors organize memory into two banks: an “even” bank and an “odd” bank (see Figure 3.6). Figure 3.7 illustrates the connection to the CPU (D0-D7 denotes the L.O. byte of the data bus, D8-D15 denotes the H.O. byte of the data bus): The 16 bit members of the 80x86 family can load a word from any arbitrary address. As mentioned earlier, the processor fetches the L.O. byte of the value from the address specified and the H.O. byte from the next consecutive address. This creates a subtle problem if you look closely at the diagram above. What happens
when you access a word on an odd address? Suppose you want to read a word from location 125. Okay, the L.O. byte of the word comes from location 125 and the H.O. word comes from location 126. What’s the big deal? It turns out that there are two problems with this approach.

First, look again at Figure 3.7. Data bus lines eight through 15 (the H.O. byte) connect to the odd bank, and data bus lines zero through seven (the L.O. byte) connect to the even bank. Accessing memory location 125 will transfer data to the CPU on the H.O. byte of the data bus; yet we want this data in the L.O. byte! Fortunately, the 80x86 CPUs recognize this situation and automatically transfer the data on D8-D15 to the L.O. byte. The second problem is even more obscure. When accessing words, we’re really accessing two separate bytes, each of which has its own byte address. So the question arises, “What address appears on the address bus?” The 16 bit 80x86 CPUs always place even addresses on the bus. Even bytes always appear on data lines D0-D7 and the odd bytes always appear on data lines D8-D15. If you access a word at an even address, the CPU can bring in the entire 16 bit chunk in one memory operation. Likewise, if
you access a single byte, the CPU activates the appropriate bank (using a “byte enable” control line). If the byte appeared at an odd address, the CPU will automatically move it from the H.O. byte on the bus to the L.O. byte. So what happens when the CPU accesses a word at an odd address, like the example given earlier? Well, the CPU cannot place the address 125 onto the address bus and read the 16 bits from memory. There are no odd addresses coming out of a 16 bit 80x86 CPU. The addresses are always even. So if you try to put 125 on the address bus, this will put 124 on to the address bus. Were you to read the 16 bits at this address, you would get the word at addresses 124 (L.O. byte) and 125 (H.O. byte) – not what you’d expect. Accessing a word at an odd address requires two memory operations. First the CPU must read the byte at address 125, then it needs to read the byte at address 126. Finally, it needs to swap the positions of these bytes internally since both entered the CPU on the wrong half of the data bus.

Fortunately, the 16 bit 80x86 CPUs hide these details from you. Your programs can access words at any address and the CPU will properly access and swap (if necessary) the data in memory. However, to access a word at an odd address requires two memory operations (just like the 8088/80188). Therefore, accessing words at odd addresses on a 16 bit processor is slower than accessing words at even addresses. By carefully arranging
how you use memory, you can improve the speed of your program. Accessing 32 bit quantities always takes at least two memory operations on the 16 bit processors. If you access a 32 bit quantity at an odd address, the processor will require three memory operations to access the data. The 32 bit 80x86 processors (the 80386, 80486, and Pentium Overdrive) use four banks of memory connected to the 32 bit data bus (see Figure 3.8). The address placed on the address bus is always some multiple of four. Using various “byte enable” lines, the CPU can select which of the four bytes at that address the software wants to access. As with the 16 bit processor, the CPU will automatically rearrange bytes as necessary. With a 32 bit memory interface, the 80x86 CPU can access any byte with one memory operation. If (address MOD 4) does not equal three, then a 32 bit CPU can access a word at that address using a single memory operation. However, if the remainder is three, then it will take two memory operations to access that word (see Figure 3.9). This is the same problem encountered with the 16 bit processor, except it occurs half as often. A 32 bit CPU can access a double word in a single memory operation if the address of that value is evenly divisible by four. If not, the CPU will require two memory operations. Once again, the CPU handles all of this automatically. In terms of loading correct data the CPU handles everything for you. However, there is a performance benefit to proper data alignment. As a general rule you should always place word values at even addresses and double word values at addresses which are evenly divisible by four. This will speed up your program.

The I/O Subsystem

Besides the 20, 24, or 32 address lines which access memory, the 80x86 family provides a 16 bit I/O address bus. This gives the 80x86 CPUs two separate address spaces: one for memory and one for I/O operations. Lines on the control bus differentiate between memory and I/O addresses. Other than separate control lines and a smaller bus, I/O addressing behaves exactly like memory addressing. Memory and I/O devices both share the same data bus and the L.O. 16 lines on the address bus. There
are three limitations to the I/O subsystem on the IBM PC: first, the 80x86 CPUs require special instructions to access I/O devices; second, the designers of the IBM PC used the “best” I/O locations for their own purposes, forcing third party developers to use less accessible locations; third, 80x86 systems can address no more than 65,536 (216) I/O addresses. When you consider that a typical VGA display card requires over 128,000 different locations, you can see a problem with the size of I/O bus. Fortunately, hardware designers can map their I/O devices into the memory address space as easily as they can the I/O address space. So by using the appropriate circuitry, they can make their I/O devices look just like memory. This is how, for example, display adapters on the IBM PC work. Accessing I/O devices is a subject we’ll return to in later chapters. For right now you can assume that I/O and memory accesses work the same way.

**System Timing**

Although modern computers are quite fast and getting faster all the time, they still require a finite amount of time to accomplish even the smallest tasks. On Von Neumann machines, like the 80x86, most operations are **serialized**. This means that the computer executes commands in a prescribed order. It wouldn’t do, for example, to execute the statement `I:=I*5+2;` before `I:=J;` in the following sequence:

```
I := J;
I := I * 5 + 2;
```

Clearly we need some way to control which statement executes first and which executes second. Of course, on real computer systems, operations do not occur instantaneously. Moving a copy of J into I takes a certain amount of time. Likewise, multiplying I by five and then adding two and storing the result back into I takes time. As you might expect, the second Pascal statement above takes quite a bit longer to execute than the first. For those interested in writing fast software, a natural question to ask is, “How does the processor execute statements, and how do we measure how long they take to execute?” The CPU is a very complex piece of circuitry. Without going into too many
details, let us just say that operations inside the CPU must be very carefully coordinated or the CPU will produce erroneous results. To ensure that all operations occur at just the right moment, the 80x86 CPUs use an alternating signal called the system clock.

The System Clock
At the most basic level, the system clock handles all synchronization within a computer system. The system clock is an electrical signal on the control bus which alternates between zero and one at a periodic rate (see Figure 3.10). CPUs are a good example of a complex synchronous logic system (see the previous chapter). The system clock gates many of the logic gates that make up the CPU allowing them to operate in a synchronized fashion.

The frequency with which the system clock alternates between zero and one is the system clock frequency. The time it takes for the system clock to switch from zero to one and back to zero is the clock period. One full period is also called a clock cycle. On most modern systems, the system clock switches between zero and one at rates exceeding several million times per second. The clock frequency is simply the number of clock cycles which occur each second. A typical 80486 chip runs at speeds of 66 million cycles per second. “Hertz” (Hz) is the technical term meaning one cycle per second. Therefore, the aforementioned 80486 chip runs at 66 million hertz, or 66 megahertz (MHz). Typical frequencies for 80x86 parts range from 5 MHz up to 200 MHz and beyond. Note that one clock period (the amount of time for one complete clock cycle) is the reciprocal of the clock frequency. For example, a 1 MHz clock would have a clock period of one microsecond (1/1,000,000th of a
second). Likewise, a 10 MHz clock would have a clock period of 100 nanoseconds (100 billionths of a second). A CPU running at 50 MHz would have a clock period of 20 nanoseconds. Note that we usually express clock periods in millionths or billionths of a second. To ensure synchronization, most CPUs start an operation on either the *falling edge* (when the clock goes from one to zero) or the *rising edge* (when the clock goes from zero to one). The system clock spends most of its time at either zero or one and very little time switching between the two. Therefore clock edge is the perfect synchronization point. Since all CPU operations are synchronized around the clock, the CPU cannot perform tasks any faster than the clock4. However, just because a CPU is running at some clock frequency doesn’t mean that it is executing that many operations each second. Many operations take multiple clock cycles to complete so the CPU often performs operations at a significantly lower rate.

**Memory Access and the System Clock**

Memory access is probably the most common CPU activity. Memory access is definitely an operation synchronized around the system clock. That is, reading a value from memory or writing a value to memory occurs no more often than once every clock cycle5. Indeed, on many 80x86 processors, it takes several clock cycles to access a memory location. The *memory access time* is the number of clock cycles the system requires to access a memory location; this is an important value since longer memory access times result in lower performance. Different 80x86 processors have different memory access times ranging from one to four clock cycles. For example, the 8088 and 8086 CPUs require *four* clock cycles to access memory; the 80486 requires only one. Therefore, the 80486 will execute programs which access memory faster than an 8086, even when running at the same clock frequency.
Memory access time is the amount of time between a memory operation request (read or write) and the time the memory operation completes. On a 5 MHz 8088/8086 CPU the memory access time is roughly 800 ns (nanoseconds). On a 50 MHz 80486, the memory access time is slightly less than 20 ns. Note that the memory access time for the 80486 is 40 times faster than the 8088/8086. This is because the 80486’s clock frequency is ten times faster and it uses one-fourth the clock cycles to access memory. When reading from memory, the memory access time is the amount of time from the point that the CPU places an address on the address bus and the CPU takes the data off the data bus. On an 80486 CPU with a one cycle memory access time, a read looks something like shown in Figure 3.11. Writing data to memory is similar (see Figure 3.11). Note that the CPU doesn’t wait for memory. The access time is specified by the clock frequency. If the memory subsystem doesn’t work fast enough, the CPU will read garbage data on a memory read operation and will not properly store the data on a memory write operation. This will surely cause the system to fail. Memory
devices have various ratings, but the two major ones are capacity and speed (access time). Typical dynamic RAM (random access memory) devices have capacities of four (or more) megabytes and speeds of 50-100 ns. You can buy bigger or faster devices, but they are much more expensive. A typical 33 MHz 80486 system uses 70 ns memory devices. Wait just a second here! At 33 MHz the clock period is roughly 33 ns. How can a system designer get away with using 70 ns memory? The answer is wait states

![Diagram of decoding and buffering delays]

**Wait States**

A wait state is nothing more than an extra clock cycle to give some device time to complete an operation. For example, a 50 MHz 80486 system has a 20 ns clock period. This implies that you need 20 ns memory. In fact, the situation is worse than this. In most computer systems there is additional circuitry between the CPU and memory: decoding and buffering logic. This additional circuitry introduces additional delays into the system (see Figure 3.13). In this diagram, the system loses 10ns to buffering and decoding. So if the CPU needs the data back in 20 ns, the memory must respond in less than 10 ns. You can actually buy 10ns memory. However, it is very expensive, bulky, consumes a lot of power, and generates a lot of heat. These are bad attributes. Supercomputers use this type of memory. However, supercomputers also cost millions of dollars, take up entire rooms, require special cooling, and have giant
power supplies. Not the kind of stuff you want sitting on your desk. If cost-effective memory won’t work with a fast processor, how do companies manage to sell fast PCs? One part of the answer is the wait state. For example, if you have a 20 MHz processor with a memory cycle time of 50 ns and you lose 10 ns to buffering and decoding, you’ll need 40 ns memory. What if you can only afford 80 ns memory in a 20 MHz system? Adding a wait state to extend the memory cycle to 100 ns (two clock cycles) will solve this problem. Subtracting 10 ns for the decoding and buffering leaves 90 ns. Therefore, 80 ns memory will respond well before the CPU requires the data. Almost every general purpose CPU in existence provides a signal on the control bus to allow the insertion of wait states. Generally, the decoding circuitry asserts this line to delay one additional clock period, if necessary. This gives the memory sufficient access time, and the system works properly (see Figure 3.14). Sometimes a single wait state is not sufficient. Consider the 80486 running at 50 MHz. The normal memory cycle time is less than 20 ns. Therefore, less than 10 ns are available after subtracting decoding and buffering time. If you are using 60 ns memory in the system, adding a single wait state will not do the trick. Each wait state gives you 20 ns, so with a single wait state you would need 30 ns memory. To work with 60 ns memory you would need to add three wait states (zero wait states = 10 ns, one wait state = 30 ns, two wait states = 50 ns, and three wait states = 70 ns). Needless to say, from the system performance point of view, wait states are not a good thing. While the CPU is waiting for data from memory it cannot operate on that data.
Adding a single wait state to a memory cycle on an 80486 CPU *doubles* the amount of time required to access the data. This, in turn, *halves* the speed of the memory access. Running with a wait state on every memory access is almost like cutting the processor clock frequency in half. You’re going to get a lot less work done in the same amount of time. You’ve probably seen the ads. “80386DX, 33 MHz, 8 megabytes 0 wait state RAM... only $1,000!” If you look closely at the specs you’ll notice that the manufacturer is using 80 ns memory. How can they build systems which run at 33 MHz and have zero wait states? Easy. They lie. There is no way an 80386 can run at 33 MHz, executing an arbitrary program, without ever inserting a wait state. It is flat out impossible. However, it is quite possible to design a memory subsystem which *under certain, special, circumstances* manages to operate without wait states part of the time. Most marketing types figure if their system *ever* operates at zero wait states, they can make that claim in their literature. Indeed, most marketing types have no idea what a wait state is other than it’s bad and having zero wait states is something to brag about. However, we’re not doomed to slow execution because of added wait states. There are several tricks hardware designers can play to achieve zero wait states *most* of the time. The most common of these is the use of *cache* (pronounced “cash”) memory.
Cache Memory

If you look at a typical program (as many researchers have), you’ll discover that it tends to access the same memory locations repeatedly. Furthermore, you also discover that a program often accesses adjacent memory locations. The technical names given to this phenomenon are *temporal locality of reference* and *spatial locality of reference*. When exhibiting spatial locality, a program accesses neighboring memory locations. When displaying temporal locality of reference a program repeatedly accesses the same memory location during a short time period. Both forms of locality occur in the following Pascal code segment:

```pascal
for i := 0 to 10 do
  A[i] := 0;
```

There are two occurrences each of spatial and temporal locality of reference within this loop. Let’s consider the obvious ones first. In the Pascal code above, the program references the variable `i` several times. The for loop compares `i` against 10 to see if the loop is complete. It also increments `i` by one at the bottom of the loop. The assignment statement also uses `i` as an array index. This shows temporal locality of reference in action since the CPU accesses `i` at three points in a short time period. This program also exhibits spatial locality of reference. The loop itself zeros out the elements of array `A` by writing a zero to the first location in `A`, then to the second location in `A`, and so on. Assuming that Pascal stores the elements of `A` into consecutive memory locations, each loop iteration accesses adjacent memory locations. There is an additional example of temporal and spatial locality of reference in the Pascal example above, although it is not so obvious. Computer *instructions* which tell the system to do the specified task also appear in memory. These instructions appear sequentially in memory – the spatial locality part. The computer also executes these instructions repeatedly, once for each loop iteration – the temporal locality part. If you look at the execution profile of a typical program, you’d discover that the program typically executes less than half the statements. Generally, a typical program might only use 10-
20% of the memory allotted to it. At any one given time, a one megabyte program might only access four to eight kilobytes of data and code. So if you paid an outrageous sum of money for expensive zero wait state RAM, you wouldn’t be using most of it at any one given time! Wouldn’t it be nice if you could buy a small amount of fast RAM and dynamically reassign its address(es) as the program executes? This is exactly what cache memory does for you. Cache memory sits between the CPU and main memory. It is a small amount of very fast (zero wait state) memory. Unlike normal memory, the bytes appearing within a cache do not have fixed addresses. Instead, cache memory can reassign the address of a data object. This allows the system to keep recently accessed values in the cache. Addresses which the CPU has never accessed or hasn’t accessed in some time remain in main (slow) memory. Since most memory accesses are to recently accessed variables (or to locations near a recently accessed location), the data generally appears in cache memory. Cache memory is not perfect. Although a program may spend considerable time executing code in one place, eventually it will call a procedure or wander off to some section of code outside cache memory. In such an event the CPU has to go to main memory to fetch the data. Since main memory is slow, this will require the insertion of wait states. A cache hit occurs whenever the CPU accesses memory and finds the data in the cache. In such a case the CPU can usually access data with zero wait states. A cache miss occurs if the CPU accesses memory and the data is not present in cache. Then the CPU has to read the data from main memory, incurring a performance loss. To take advantage of locality of reference, the CPU copies data into the cache whenever it accesses an address not present in the cache. Since it is likely the system will access that same location shortly, the system will save wait states by having that data in the cache. As described above, cache memory handles the temporal aspects of memory access, but not the spatial aspects. Caching memory locations when you access them won’t speed up the program if you constantly access consecutive locations (spatial locality of reference). To solve this problem, most
caching systems read several consecutive bytes from memory when a cache miss occurs. The 80486, for example, reads 16 bytes at a shot upon a cache miss. If you read 16 bytes, why read them in blocks rather than as you need them? As it turns out, most memory chips available today have special modes which let you quickly access several consecutive memory locations on the chip. The cache exploits this capability to reduce the average number of wait states needed to access memory. If you write a program that randomly accesses memory, using a cache might actually slow you down. Reading 16 bytes on each cache miss is expensive if you only access a few bytes in the corresponding cache line. Nonetheless, cache memory systems work quite well. It should come as no surprise that the ratio of cache hits to misses increases with the size (in bytes) of the cache memory subsystem. The 80486 chip, for example, has 8,192 bytes of on-chip cache. Intel claims to get an 80-95% hit rate with this cache (meaning 80-95% of the time the CPU finds the data in the cache). This sounds very impressive. However, if you play around with the numbers a little bit, you’ll discover it’s not all that impressive. Suppose we pick the 80% figure. Then one out of every five memory accesses, on the average, will not be in the cache. If you have a 50 MHz processor and a 90 ns memory access time, four out of five memory accesses require only one clock cycle (since they are in the cache) and the fifth will require about 10 wait states. Altogether, the system will require 15 clock cycles to access five memory locations, or three clock cycles per access, on the
average. That’s equivalent to two wait states added to every memory access. Now do you believe that your machine runs at zero wait states? There are a couple of ways to improve the situation. First, you can add more cache memory. This improves the cache hit ratio, reducing the number of wait states. For example, increasing the hit ratio from 80% to 90% lets you access 10 memory locations in 20 cycles. This reduces the average number of wait states per memory access to one wait state – a substantial improvement. Alas, you can’t pull an 80486 chip apart and solder more cache onto the chip. However, the 80586/Pentium CPU has a significantly larger cache than the 80486 and operates with fewer wait states. Another way to improve performance is to build a two-level caching system. Many 80486 systems work in this fashion. The first level is the on-chip 8,192 byte cache. The next level, between the on-chip cache and main memory, is a secondary cache built on the computer system circuit board (see Figure 3.15). A typical secondary cache contains anywhere from 32,768 bytes to one megabyte of memory. Common sizes on PC subsystems are 65,536 and 262,144 bytes of cache. You might ask, “Why bother with a two-level cache? Why not use a 262,144 byte cache to begin with?” Well, the secondary cache generally does not operate at zero wait states. The circuitry to support 262,144 bytes of 10 ns memory (20 ns total access time) would be very expensive. So most system designers use slower memory which requires one or two wait states. This is still much faster than main memory. Combined with the on-chip cache, you can get better performance from the system. Consider the previous example with an 80% hit ratio. If the secondary cache requires two cycles for each memory access and three cycles for the first access, then a cache miss on the on-chip cache will require a total of six clock cycles. All told, the average system performance will be two clocks per memory access. Quite a bit faster than the three required by the system without the secondary cache. Furthermore, the secondary cache can update its values in parallel with the CPU. So the number of cache misses (which affect CPU performance) goes way down. You’re
probably thinking, “So far this all sounds interesting, but what does it have to do with programming?” Quite a bit, actually. By writing your program carefully to take advantage of the way the cache memory system works, you can improve your program’s performance. By collocating variables you commonly use together in the same cache line, you can force the cache system to load these variables as a group, saving extra wait states on each access. If you organize your program so that it tends to execute the same sequence of instructions repeatedly, it will have a high degree of temporal locality of reference and will, therefore, execute faster.

**CPU Registers**

CPU registers are *very* special memory locations constructed from flip-flops. They are not part of main memory; the CPU implements them on-chip. Various members of the 80x86 family have different register sizes. The 886, 8286, 8486, and 8686 (x86 from now on) CPUs have exactly four registers, all 16 bits wide. All arithmetic and location operations occur in the CPU registers. Because the x86 processor has so few registers, we’ll give each register its own name and refer to it by that name rather than its address. The names for the x86 registers are:

- **AX** – The accumulator register
- **BX** – The base address register
- **CX** – The count register
- **DX** – The data register

Besides the above registers, which are visible to the programmer, the x86 processors also have an *instruction pointer* register which contains the address of the next instruction to execute. There is also a *flags* register that holds the result of a comparison. The flags register remembers if one value was less than, equal to, or greater than another value. Because registers are on-chip and handled specially by the CPU, they are much faster than memory. Accessing a memory location requires one or more clock cycles. Accessing data in a register usually takes zero clock cycles. Therefore, you should try to keep variables in the registers. Register sets are very small and most registers
have special purposes which limit their use as variables, but they are still an excellent place to store temporary data.

The Arithmetic & Logical Unit
The arithmetic and logical unit (ALU) is where most of the action takes place inside the CPU. For example, if you want to add the value five to the AX register, the CPU:
• Copies the value from AX into the ALU,
• Sends the value five to the ALU,
• Instructs the ALU to add these two values together,
• Moves the result back into the AX register.

The Bus Interface Unit
The bus interface unit (BIU) is responsible for controlling the address and data busses when accessing main memory. If a cache is present on the CPU chip then the BIU is also responsible for accessing data in the cache.

The Control Unit and Instruction Sets
A fair question to ask at this point is “How exactly does a CPU perform assigned chores?” This is accomplished by giving the CPU a fixed set of commands, or instructions, to work on. Keep in mind that CPU designers construct these processors using logic gates to execute these instructions. To keep the number of logic gates to a reasonably small set (tens or hundreds of thousands), CPU designers must necessarily restrict the number and complexity of the commands the CPU recognizes. This small set of commands is the CPU’s instruction set.
Programs in early (pre-Von Neumann) computer systems were often “hard-wired” into the circuitry. That is, the computer’s wiring determined what problem the computer would solve. One had to rewire the circuitry in order to change the program. A very difficult task. The next advance in computer design was the *programmable* computer system, one that allowed a computer programmer to easily “rewire” the computer system using a sequence of sockets and plug wires. A computer program consisted of a set of rows of holes (sockets), each row representing one operation during the execution of the program. The programmer could select one of several instructions by plugging a wire into the particular socket for the desired instruction (see Figure 3.16). Of course, a major difficulty with this scheme is that the number of possible instructions is severely limited by the number of sockets one could physically place on each row. However, CPU designers quickly discovered that with a small amount of additional logic circuitry, they could reduce the number of sockets required from $n$ holes for $n$ instructions to $\log_2(n)$ holes for $n$ instructions. They did this by assigning a *numeric* code to each instruction and then
encode that instruction as a binary number using $\log_2(n)$ holes (see Figure 3.17). This addition requires eight logic functions to decode the A, B, and C bits from the patch panel, but the extra circuitry is well worth the cost because it reduces the number of sockets that must be repeated for each instruction. Of course, many CPU instructions are not stand-alone. For example, the move instruction is a command that moves data from one location in the computer to another (e.g., from one register to another). Therefore, the move instruction requires two operands: a source operand and a destination operand. The CPU’s designer usually encodes these source and destination operands as part of the machine instruction, certain sockets correspond to the source operand and certain sockets correspond to the destination operand. Figure 3.17 shows one possible combination of sockets to handle this. The move instruction would move data from the source register to the destination register, the add instruction would add the value of the source register to the destination register, etc. One of the primary
advances in computer design that the VNA provides is the concept of a stored program. One big problem with the patch panel programming method is that the number of program steps (machine instructions) is limited by the number of rows of sockets available on the machine. John Von Neumann and others recognized a relationship between the sockets on the patch panel and bits in memory; they figured they could store the binary equivalents of a machine program in main memory and fetch each program from memory, load it into a special decoding register that connected directly to the instruction decoding circuitry of the CPU. The trick, of course, was to add yet more circuitry to the CPU. This circuitry, the control unit (CU), fetches instruction codes (also known as operation codes or opcodes) from memory and moves them to the instruction decoding register. The control unit contains a special registers, the instruction pointer that contains the address of an executable instruction. The control unit fetches this instruction’s code from memory and places it in the decoding register for execution. After executing the instruction, the control unit increments the instruction pointer and fetches the next instruction from memory for execution, and so on. When designing an instruction set, the CPU’s designers generally choose opcodes that are a multiple of eight bits long so the CPU can easily fetch complete instructions from memory. The goal of the CPU’s designer is to assign an appropriate number of bits to the instruction class field (move, add, subtract, etc.) and to the operand fields. Choosing more bits for the instruction field lets you have more instructions, choosing additional bits for the operand fields lets you select a larger number of operands (e.g., memory locations or registers). There are additional complications. Some instructions have only one operand or, perhaps, they don’t have any operands at all. Rather than waste the bits associated with these fields, the CPU designers often reuse these fields to encode additional opcodes, once again with some additional circuitry. The Intel 80x86 CPU family takes this to an extreme with instructions ranging from one to about ten bytes long. Since this is a little too difficult to
deal with at this early stage, the x86 CPUs will use a different, much simpler, encoding scheme.
Storage

Primary storage
“Main Memory”
- Volatile storage
- Temporary storage

Secondary storage
- Nonvolatile storage
- Permanent storage

Characteristics
- Media
- Capacity
- Storage devices
- Access speed

Terminology

- **Reading**
  - The process of accessing information from secondary storage
  - “Getting” the data from disk
  - Opening a file → **reading**!

- **Writing**
  - The process of saving information to the secondary storage device
  - “Storing” data on the disk
  - Saving a file → **writing**!
Secondary Storage Devices

- Most desktop microcomputer systems have:

- floppy disks
- hard disks
- optical disk drives

Inexpensive, Removable

Secondary Storage Devices

- Most desktop microcomputer systems have:

- floppy disks
- hard disks
- optical disk drives

Large storage capacity, fast access time
Secondary Storage Devices

- Most desktop microcomputer systems have:
  - floppy disks, hard disks, or optical disk drives

Large storage capacity, durable and inexpensive

Secondary Storage Devices

- Uses:
  - Save important data (files)
  - Backup data
  - Transport data and programs
Floppy Disks

- Portable or removable storage media
- Typically used to store and transfer small word processing, spreadsheet, and other types of files
- Floppy disk drives (FDD)
  - Store data and programs
  - Retrieves data by reading electromagnetic charges
  - Also called flexible disks and floppies

Traditional Floppy Disk

- Most common type is 2HD “two-sided, high-density”
- Attributes
  - Shutter
  - Labels
  - Write-protection notch
  - Tracks
  - Sectors
High Capacity Floppy Disks

- Known as a floppy-disk cartridge
- Require special disk drives
- Most widely used is the Zip disk (Iomega)
  - 100 MB, 250 MB or 750 MB capacity
  - Used to store multimedia, database, large text, and spreadsheet files

Hard Disks

- Use thicker, metallic platters for storage
- Faster than a floppy diskette
- Large capacity
- Sensitive instruments
- There are three types of hard disks:
  - Internal Hard Disk
  - Hard-disk cartridge
  - Hard-disk pack
- Performance Enhancements
Materials that Cause a “Head Crash”

- **Head crash** is a disaster for a hard disk
  (Hard disk unit is completely sealed)

```
So Backup!
```

Internal Hard Disk

- Located inside system unit
- Designated as the C: drive
- Advantages over floppies
  - Capacity
  - Access speed
Internal Hard-Disk

Hard-Disk Cartridges

- Removable hard disks
- Used to complement internal hard disk
- Capacities of 20 to 100 GB
- Iomega is one of the most widely used
Hard-Disk Packs

- Removable hard disk
- Massive storage capacity
- Common in mainframes
- Are utilized by banks and credit card companies

Performance Enhancements

- Disk caching
- Redundant arrays of inexpensive disks (RAID)
- File compression and decompression
RAID

Redundant Arrays of Inexpensive Disks

- A group of low cost hard-disk drives
- All connected together to act as one disk
  (Specialized Hardware, Software, OS)
- Performs as a single large-capacity disk
- Faster than a single disk of comparable size
- Often used by
  Internet servers and large organizations
- Increased reliability is provided

Optical Disks

- Hold over 50 gigabytes of data

- Attributes
  - Lands
  - Pits

- Three types
  - Compact Disc (CD)
  - Digital Versatile Disc (DVD)
  - Hi-Def Disc
Compact Disc

- Optical format
- From 650 MB to 1 GB capacity
- Rotation speeds vary
- Three basic types
  - Read only: CD-ROM
  - Write once: CD-R
  - Rewriteable: CD-RW

Digital Versatile Disc

- Digital Versatile Disc or Digital Video Disc (DVD)
- Similar to CDs, but can store more data
- Three basic types
  - Read only: DVD-ROM
  - Write once: DVD+R; DVD-R
  - Rewritable: DVD+RW; DVD-RW; DVD-RAM
High-Definition Disc

- Next generation of optical disc
- Far greater capacity than DVDs
- Three basic types
  - Read only
  - Write once
  - Rewriteable
- Two competing hi def formats
  - HD DVD
  - Blu-Ray

<table>
<thead>
<tr>
<th>Format</th>
<th>Typical Capacity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CD</td>
<td>650 MB to 1.6 GB</td>
<td>Circa the standard optical disc</td>
</tr>
<tr>
<td>DVD</td>
<td>4.7 GB to 17 GB</td>
<td>Current standard</td>
</tr>
<tr>
<td>HD DVD</td>
<td>15 GB to 45 GB</td>
<td>Hi-def format, similar to HD DVD</td>
</tr>
<tr>
<td>Blu-Ray</td>
<td>25 GB to 50 GB</td>
<td>Hi-def format, large capacity</td>
</tr>
</tbody>
</table>

High-Definition DVD
The Hi-Def Optical Disk Format War

- HD DVD
  - 15 GB (30GB dual layer)
  - Now obsolete

- Blu-ray
  25 GB (50GB dual layer)
Other Types of Secondary Storage

- **Solid-state storage**
- **Internet hard drives**
- **Magnetic tape**

Solid-State Storage

- **Flash memory cards**
  - Widely used in notebook computers
  - Used to record MP3 music files

- **USB Flash Drives (“Thumb Drives”)**
  - Key ring flash memory devices or flash drives
  - Connects to a USB port
  - Up to 2GB
Inside a USB Flash Drive

USB connector

Flash memory chip

Microcontroller chip

"Thumb Drive"
Disk Tracks

- Data is written on the disk drive in concentric circles called **tracks**.
- A track is composed of blocks of data called **sectors**. Each sector has a header including address and checksum.
- An arm containing the read/write head can move closer or farther from the center of the disk.
- All of the tracks on both sides of all platters that can be read without moving the heads is called a **cylinder**.

Terminology

- **Sector** or **Block** – the smallest unit that can be read or written. Often 512 bytes.
- **Track** – all blocks that form a ring on a disk surface that can be read without moving the head.
- **Cylinder** – all tracks on all surfaces, one on top of another, that can be read without moving the head.
Disk Performance Parameters

Disk read or write involves three factors

1. **Seek time**
   - time it takes to position the head at the desired track

2. **Rotational delay** or rotational latency
   - time it takes for the beginning of the sector to reach the head

3. **Transfer time**
   - time required for the data to move under the head

---

**Writing to memory**

1. **Address** of the memory location is placed on the Address Bus
2. **Data** is placed on the Data Bus
3. Write Signal (WR) in the control bus is activated

---

**Reading from memory**

1. **Address** of the memory location is placed on the Address Bus
2. Read Signal (RD) in the control bus is activated
3. **Data** is fetched (read) from the Data Bus
Types of Memories

1. **ROM – Read only memory**
   - Can only be read, cannot be written to..
   - Contents are written at the time of manufacture
   - Stores initial start-up programs
   - Not economical to produce in small quantities

Types of Memories cont. . .

2. **PROM – Programmable Read Only Memories**
   - Same as ROMS, but contents can be written once, using special equipment.

3. **UVEPROM - UV Erasable PROM**
   - Same as PROM but contents can be erased by shining an UV light on the IC
   - Require special equipment to program
Types of Memories cont....

4. EEPROM – Electrical Erasable PROM
   - Same as UVEPROM except that contents can be erased by applying a special high voltage to some of the signals

5. FLASH ROM
   - A special type of EEPROM that can be erased or programmed while in the circuit.
   - Once programmed the contents remains unchanged – even after a power failure
   - Commonly used in modern PCs
Types of Memories cont....

6. **RWM (Read Write Memory)**
   - Traditionally known as RAM (Random Access Memory)
   - Contents are erased when power is disconnected.
   - Two major types – SRAM and DRAM

---

### Transistors vs. Capacitors

<table>
<thead>
<tr>
<th>Transistors</th>
<th>Capacitors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Developed using Silicon &amp; other semiconductors</td>
<td>Developed using Silicon &amp; other semiconductors</td>
</tr>
<tr>
<td>High speed switching</td>
<td>Slower performance</td>
</tr>
<tr>
<td>Will retain state forever</td>
<td>Automatically discharges after sometime, need refreshing</td>
</tr>
<tr>
<td>Reliable</td>
<td>Less reliable</td>
</tr>
<tr>
<td>Low density</td>
<td>High density</td>
</tr>
<tr>
<td>High power consumption</td>
<td>Low power consumption</td>
</tr>
<tr>
<td>High cost (per bit)</td>
<td>Low cost (per bit)</td>
</tr>
</tbody>
</table>
Types of Memories cont....

7. SRAM (Static RAM)
   - More reliable but expensive
   - Typically used for cache memories
   - More expensive & consumes more electrical power

8. DRAM (Dynamic RAM)
   - Contents are stored as charges in a small capacitor
   - Capacitor must be re-charged from time to type
   - Bulk of the PC memory is made out of DRAM

Memory Hierarchy

- Modern CPU’s are much faster than the speed of memory
- The memory has to be organized in such a way that its slowness does not reduce the performance of the overall system
- Some memory types are fast but expensive
  - Registers, SRAM
- Some other types are cheap but slow
  - DRAM
The objective of having a memory hierarchy is to have a memory system:
- with a sufficient speed
- with a sufficient capacity
- as cheap as possible

The main idea is to use:
- a limited capacity of fast but expensive memory types
- and a larger capacity of slow but cheap memory types

Traditional memory hierarchy
Cache Memory

- Consist of small amount of memory (few KBs in size) which is faster than the main memory
- Cache is used by the CPU to store frequently used instructions & data
- Sometimes multiple levels of cache is used
  - L1 Cache – Very fast, usually within CPU itself
  - L2 Cache – Slower than L1, but faster than main memory
Arithmetic & Logic Unit (ALU)

- is the data processing unit of the CPU
- Arithmetic unit is capable of performing arithmetic operations
- Logic unit performs logical operations

Control Unit

- Controls the operations of the CPU
Registers

- Are type of memory located inside the CPU
- Can hold a single piece of data
- This data is useful in both data processing & controlling
- Several types of CPU registers
  - Program counter (PC)
  - Instruction register (IR)
  - Accumulator
  - Flag register
  - General purpose registers

Program Counter (PC)

- Used to keep track of memory address of the next instruction to be executed
- When instructions are fetched (fetch cycle), always the instruction pointed by the program counter is fetched into the CPU
- Once the instruction is fetched, the program counter is updated to point to the next instruction
Instruction Register (IR)

- Once fetched, instructions are stored in the IR for execution
- Located closely to the control unit which decodes the instruction

Accumulator (A)

- Results of arithmetic & logical operations always go to the accumulator
- Connected directly to the output of the ALU

Flags Register (F)

- Store status of the last operation carried out by the ALU
- Usually include states such as:
  - Overflow
  - Zero results
  - Results of comparisons, etc.
- Status is updated directly by the ALU

General Purpose Registers (B,C,D,E ..)

- Used to store intermediate results of ALU operations
- Few general purpose registers are available in the CPU
Internal Structure of the CPU

Display (Video) Controller
VGA card classifications

- based on:
  - Video processor
  - Video memory

- Card type
  - PCI – Peripheral Component Interconnect
  - AGP – Accelerated Graphics port
  - PCI Express

Video cards
Display

- Produces the visible output
- Different types:
  - CRT - Cathode Ray Tube
  - TFT - Thin Film Transistor
  - LCD - Liquid Crystal Display
- Each of these types has its own advantages & disadvantages

Cathode Ray Tube
Cathode Ray Tube cont…

- Advantages
  - Better colour representation
  - Larger viewing angle

- Disadvantages
  - Higher power consumption
  - Consume more desk space
  - Not good for the eye

Thin Film Transistor

- Advantages
  - Lower power consumption
  - Consume less desk space
  - Less affect on eye

- Disadvantages
  - Lower colour quality
  - Lower viewing angle
  - High cost
Liquid Crystal Display

Liquid Crystal Display cont…

- **Advantages**
  - Lower cost
  - Lower power consumption
  - Suitable for small devices

- **Disadvantages**
  - Lower colour quality
  - Lower viewing angle
Categorization of displays

- Screen size
  - 14”, 15”, 17”, 21”

- Colour depth
  - Number of colours support

- Resolution
  - Measure of the number of horizontal & vertical pixels
  - Determines the amount of information that appears on the screen
  - 800x600, 1024x768, 1600x1200