Addressing Modes

Chapter 5
S. Dandamudi

Outline

• Addressing modes
• Simple addressing modes
  * Register addressing mode
  * Immediate addressing mode
• Memory addressing modes
  * 16-bit and 32-bit addressing
    » Operand and address size override prefixes
  * Direct addressing
  * Indirect addressing
  * Based addressing
  * Indexed addressing
  * Based-indexed addressing
• Examples
  * Sorting (insertion sort)
  * Binary search
• Arrays
  * One-dimensional arrays
  * Multidimensional arrays
  * Examples
    » Sum of 1-d array
    » Sum of a column in a 2-d array
• Performance: Usefulness of addressing modes
Addressing Modes

- Addressing mode refers to the specification of the location of data required by an operation.
- Pentium supports three fundamental addressing modes:
  * Register mode
  * Immediate mode
  * Memory mode
- Specification of operands located in memory can be done in a variety of ways
  * Mainly to support high-level language constructs and data structures

Pentium Addressing Modes (32-bit Addresses)

- Register
- Immediate
- Memory
  - Direct [disp]
  - Indirect
    - Register Indirect [Base]
    - Based [Base + disp]
    - Indexed [(Index * scale) + disp]
    - Based-Indexed
      - Based-Indexed with no scale factor [Base + Index + disp]
      - Based-Indexed with scale factor [Base + (Index * scale) + disp]
# Memory Addressing Modes (16-bit Addresses)

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Direct</strong></td>
<td>[disp]</td>
</tr>
<tr>
<td><strong>Indirect</strong></td>
<td>[BX]</td>
</tr>
<tr>
<td><strong>Register Indirect</strong></td>
<td>[BX], [BP], [SI], [DI]</td>
</tr>
<tr>
<td><strong>Based</strong></td>
<td>[BX + disp]</td>
</tr>
<tr>
<td><strong>Indexed</strong></td>
<td>[SI + disp]</td>
</tr>
<tr>
<td><strong>Based-Indexed</strong></td>
<td>[BX + SI + disp], [BP + SI + disp], [BX + DI + disp], [BP + DI + disp]</td>
</tr>
</tbody>
</table>

## Simple Addressing Modes

### Register Addressing Mode

- Operands are located in registers
- It is the most efficient addressing mode
  - No memory access is required
  - Instructions tend to be shorter
    - Only 3 bits are needed to specify a register as opposed to at least 16 bits for a memory address
- An optimization technique:
  - Place the frequently accesses data (e.g., index variable of a big loop) in registers
Simple Addressing Modes (cont’d)

**Immediate Addressing Mode**
- Operand is stored as part of the instruction
- This mode is used mostly for constants
- It imposes several restrictions:
  * Typically used in instructions that require at least two operands (exceptions like `push` exist)
  * Can be used to specify only the source operands (not the destination operand)
  * Another addressing mode is required for specifying the destination operand
- Efficient as the data comes with the instructions (instructions are generally prefetched)

Memory Addressing Modes
- Pentium offers several addressing modes to access operands located in memory
  - Primary reason: To efficiently support high-level language constructs and data structures.
- Available addressing modes depend on the address size used
  * 16-bit modes (shown before)
    » same as those supported by 8086
  * 32-bit modes (shown before)
    » supported by Pentium
    » more flexible set
### 32-Bit Addressing Modes

- These addressing modes use 32-bit registers

**Segment + Base + (Index * Scale) + displacement**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>Scale</th>
<th>Displacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS</td>
<td>EAX</td>
<td>EAX</td>
<td>1</td>
<td>no displacement</td>
</tr>
<tr>
<td>SS</td>
<td>EBX</td>
<td>EBX</td>
<td>2</td>
<td>8-bit displacement</td>
</tr>
<tr>
<td>DS</td>
<td>ECX</td>
<td>ECX</td>
<td>4</td>
<td>32-bit displacement</td>
</tr>
<tr>
<td>ES</td>
<td>EDX</td>
<td>EDX</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>FS</td>
<td>ESI</td>
<td>ESI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GS</td>
<td>EDI</td>
<td>EDI</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>EBP</td>
<td>EBP</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ESP</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Differences between 16- and 32-bit Modes

<table>
<thead>
<tr>
<th></th>
<th>16-bit addressing</th>
<th>32-bit addressing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base register</td>
<td>BX, BP</td>
<td>EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP</td>
</tr>
<tr>
<td>Index register</td>
<td>SI, DI</td>
<td>EAX, EBX, ECX, EDX, ESI, EDI, EBP</td>
</tr>
<tr>
<td>Scale factor</td>
<td>None</td>
<td>1, 2, 4, 8</td>
</tr>
<tr>
<td>Displacement</td>
<td>0, 8, 16 bits</td>
<td>0, 8, 32 bits</td>
</tr>
</tbody>
</table>
16-bit or 32-bit Addressing Mode?

- How does the processor know?
- Uses the D bit in the CS segment descriptor
  - \(D = 0\)
    - default size of operands and addresses is 16 bits
  - \(D = 1\)
    - default size of operands and addresses is 32 bits
- We can override these defaults
  * Pentium provides two size override prefixes
    - \(66H\) operand size override prefix
    - \(67H\) address size override prefix
- Using these prefixes, we can mix 16- and 32-bit data and addresses

---

Examples: Override Prefixes

- Our default mode is 16-bit data and addresses

**Example 1:** Data size override

\[
\begin{align*}
\text{mov } AX, 123 & \Rightarrow B8 007B \\
\text{mov } EAX, 123 & \Rightarrow 66 | B8 000007B
\end{align*}
\]

**Example 2:** Address size override

\[
\begin{align*}
\text{mov } AX, [EBX*ESI+2] & \Rightarrow 67 | 8B0473
\end{align*}
\]

**Example 3:** Address and data size override

\[
\begin{align*}
\text{mov } EAX, [EBX*ESI+2] & \Rightarrow 66 | 67 | 8B0473
\end{align*}
\]
Direct Addressing

- Offset (i.e., effective address) is specified as part of the instruction
  - The assembler replaces variable names by their offset values during the assembly process
  - Useful to access only simple variables

Example

```plaintext
total_marks =
  assign_marks + test_marks + exam_marks
```

Translated into

```plaintext
mov    EAX,assign_marks
add    EAX,test_marks
add    EAX,exam_marks
mov    total_marks,EAX
```

Register Indirect Addressing

- Effective address is placed in a general-purpose register
- In 16-bit segments
  * only BX, SI, and DI are allowed to hold an effective address
    ```plaintext
    add    AX,[BX]  \text{ is valid}
    add    AX,[CX]  \text{ is NOT allowed}
    ```
- In 32-bit segments
  * any of the eight 32-bit registers can hold an effective address
    ```plaintext
    add    AX,[ECX]  \text{ is valid}
    ```
Register Indirect Addressing (cont’d)

Default Segments

• 16-bit addresses
  * BX, SI, DI : data segment
  * BP, SP : stack segment

• 32-bit addresses
  * EAX, EBX, ECX, EDX, ESI, EDI: data segment
  * EBP, ESP : stack segment

• Possible to override these defaults
  * Pentium provides segment override prefixes

Overriding Default Segments

• Use CS, SS, DS, ES, FS, or GS as in
  \[
  \text{add AX, SS:[BX]; uses stack segment} \\
  \text{add AX, DS:[BP]; uses data segment}
  \]

• You \textit{cannot use} these segment override prefixes to affect the default segment association in the following cases:
  * Destination of string instructions: always ES
  * Stack push and pop operations: always SS
  * Instruction fetch: always CS
Based Addressing

- Effective address is computed as:
  \[ \text{base} + \text{signed displacement} \]
  
  * Displacement:
    - 16-bit addresses: 8- or 16-bit number
    - 32-bit addresses: 8- or 32-bit number
- Useful to access fields of a structure or record:
  - Base register \( \rightarrow \) points to the base address of the structure
  - Displacement \( \rightarrow \) relative offset within the structure
- Useful to access arrays whose element size is not 2, 4, or 8 bytes:
  - Displacement \( \rightarrow \) points to the beginning of the array
  - Base register \( \rightarrow \) relative offset of an element within the array

Based Addressing (cont’d)

| SSA + 100 | Enrollment | 2 |
| SSA + 50  | # registered | 2 |
| displacement 48 bytes | Room # | 5 |
| SSA Structure Starting Address | Term | 1 |
| | Title | 38 |
| | Course # | 2 |
| | Enrollment | 2 |
| | # registered | 2 |
| | Room # | 5 |
| | Term | 1 |
| | Title | 38 |
| | Course # | 2 |

First course record (50 bytes)

Second course record (50 bytes)
### Indexed Addressing

- **Effective address is computed as**
  
  \[(\text{Index} \times \text{scale factor}) + \text{signed displacement}\]

- **16-bit addresses:**
  - displacement: 8- or 16-bit number
  - scale factor: none (i.e., 1)

- **32-bit addresses:**
  - displacement: 8- or 32-bit number
  - scale factor: 2, 4, or 8

- **Useful to access elements of an array (particularly if the element size is 2, 4, or 8 bytes)**
  - Displacement ==> points to the beginning of the array
  - Index register ==> selects an element of the array (array index)
  - Scaling factor ==> size of the array element

#### Examples

- **add AX, [DI+20]**
  - We have seen similar usage to access parameters off the stack (in Chapter 4)

- **add AX, marks_table[ESI*4]**
  - Assembler replaces `marks_table` by a constant (i.e., supplies the displacement)
  - Each element of `marks_table` takes 4 bytes (the scale factor value)
  - ESI needs to hold the element subscript value

- **add AX, table1[SI]**
  - SI needs to hold the element offset in bytes
  - When we use the scale factor we avoid such byte counting
Based-Indexed Addressing

Based-indexed addressing with no scale factor

- Effective address is computed as
  \[ \text{Base} + \text{Index} + \text{signed displacement} \]
- Useful in accessing two-dimensional arrays
  » Displacement \(\Rightarrow\) points to the beginning of the array
  » Base and index registers point to a row and an element within that row
- Useful in accessing arrays of records
  » Displacement \(\Rightarrow\) represents the offset of a field in a record
  » Base and index registers hold a pointer to the base of the array and the offset of an element relative to the base of the array

Based-Indexed Addressing (cont’d)

- Useful in accessing arrays passed on to a procedure
  » Base register \(\Rightarrow\) points to the beginning of the array
  » Index register \(\Rightarrow\) represents the offset of an element relative to the base of the array

Example

Assuming BX points to `table1`

```
mov AX, [BX+SI]
cmp AX, [BX+SI+2]
```

compares two successive elements of `table1`
Based-Indexed Addressing (cont’d)

Based-indexed addressing with scale factor

- Effective address is computed as
  \[ \text{Base} + (\text{Index} \times \text{scale factor}) + \text{signed displacement} \]
- Useful in accessing two-dimensional arrays when the element size is 2, 4, or 8 bytes
  - Displacement ==> points to the beginning of the array
  - Base register ==> holds offset to a row (relative to start of array)
  - Index register ==> selects an element of the row
  - Scaling factor ==> size of the array element

Illustrative Examples

- Insertion sort
  * ins_sort.asm
  * Sorts an integer array using insertion sort algorithm
    - Inserts a new number into the sorted array in its right place

- Binary search
  * bin_srch.asm
  * Uses binary search to locate a data item in a sorted array
    - Efficient search algorithm
Arrays

One-Dimensional Arrays

- Array declaration in HLL (such as C)

```c
int test_marks[10];
```

specifies a lot of information about the array:

- Name of the array (`test_marks`)
- Number of elements (10)
- Element size (2 bytes)
- Interpretation of each element (`int` i.e., signed integer)
- Index range (0 to 9 in C)

- You get very little help in assembly language!

Arrays (cont’d)

- In assembly language, declaration such as

```assembly
test_marks    DW  10 DUP (?)
```

only assigns name and allocates storage space.

- You, as the assembly language programmer, have to “properly” access the array elements by taking element size and the range of subscripts.

- Accessing an array element requires its displacement or offset relative to the start of the array in bytes.
Arrays (cont’d)

• To compute displacement, we need to know how the array is laid out
  » Simple for 1-D arrays
• Assuming C style subscripts (i.e., subscript starts at zero)
  displacement = subscript * element size in bytes
• If the element size is 2, 4, or 8 bytes, a scale factor can be used to avoid counting displacement in bytes

Multidimensional Arrays

• We focus on two-dimensional arrays
  » Our discussion can be generalized to higher dimensions
• A 5×3 array can be declared in C as
  int class_marks[5][3];
• Two dimensional arrays can be stored in one of two ways:
  * Row-major order
    – Array is stored row by row
    – Most HLL including C and Pascal use this method
  * Column-major order
    – Array is stored column by column
    – FORTRAN uses this method
Multidimensional Arrays (cont’d)

- Why do we need to know the underlying storage representation?
  - In a HLL, we really don’t need to know
  - In assembly language, we need this information as we have to calculate displacement of element to be accessed

- In assembly language,
  \[ \text{class_marks \ DW} \ 5*3 \text{ DUP (?)} \]
  allocates 30 bytes of storage

- There is no support for using row and column subscripts
  - Need to translate these subscripts into a displacement value
Multidimensional Arrays (cont’d)

- Assuming C language subscript convention, we can express displacement of an element in a 2-D array at row \( i \) and column \( j \) as

\[
displacement = (i \times \text{COLUMNS} + j) \times \text{ELEMENT\_SIZE}
\]

where

- \( \text{COLUMNS} \) = number of columns in the array
- \( \text{ELEMENT\_SIZE} \) = element size in bytes

**Example:** Displacement of

\[
\text{class\_marks}[3,1]
\]

element is \((3 \times 3 + 1) \times 2 = 20\)

Examples of Arrays

**Example 1**

- One-dimensional array
  - Computes array sum (each element is 4 bytes long e.g., long integers)
  - Uses scale factor 4 to access elements of the array by using a 32-bit addressing mode (uses ESI rather than SI)
  - Also illustrates the use of predefined location counter $\$

**Example 2**

- Two-dimensional array
  - Finds sum of a column
  - Uses “based-indexed addressing with scale factor” to access elements of a column
Performance: Usefulness of Addressing Modes

**Experiment 1**
- 16-bit addressing modes
  - Performance impact on insertion sort:
    - Only indirect mode vs. all addressing modes
  - Shows the usefulness of providing more flexible addressing modes than the basic indirect addressing mode

**Experiment 2**
- Impact of mixing 16- and 32-bit addressing modes
  - Brings out the overheads involved with using mixed addressing modes (size override prefix takes a clock cycle)
  - Tradeoff: convenience vs. performance
  - Try not to use mixed addressing modes

---

**Experiment 1**

```
<table>
<thead>
<tr>
<th>Number of elements</th>
<th>Sort time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0</td>
</tr>
<tr>
<td>2000</td>
<td>2</td>
</tr>
<tr>
<td>3000</td>
<td>4</td>
</tr>
<tr>
<td>4000</td>
<td>6</td>
</tr>
<tr>
<td>5000</td>
<td>8</td>
</tr>
<tr>
<td>6000</td>
<td>10</td>
</tr>
<tr>
<td>7000</td>
<td></td>
</tr>
<tr>
<td>8000</td>
<td></td>
</tr>
</tbody>
</table>
```

- **only indirect mode**
- **all 16-bit modes**

---

To be used with S. Dandamudi, "Introduction to Assembly Language Programming," Springer-Verlag, 1998.
Experiment 2

![Graph showing sort time vs. number of elements for 16- and 32-bit modes compared to only 16-bit modes.](image)

- **Sort time (seconds)**
- **Number of elements**

**Legend:**
- 16- and 32-bit modes
- only 16-bit modes

1998 © S. Dandamudi Addressing modes: Page 35

To be used with S. Dandamudi, "Introduction to Assembly Language Programming," Springer-Verlag, 1998.