Chapter Overview

• **Data Transfer Instructions**
• Addition and Subtraction
• Data-Related Operators and Directives
• Indirect Addressing
• JMP and LOOP Instructions
Data Transfer Instructions

• Operand Types
• Instruction Operand Notation
• Direct Memory Operands
• MOV Instruction
• Zero & Sign Extension
• XCHG Instruction
• Direct-Offset Instructions
Operand Types

• Three basic types of operands:
  – Immediate – a constant integer (8, 16, or 32 bits)
    • value is encoded within the instruction
  – Register – the name of a register
    • register name is converted to a number and encoded within the instruction
  – Memory – reference to a location in memory
    • memory address is encoded within the instruction, or a register holds the address of a memory location
## Instruction Operand Notation

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r8</td>
<td>8-bit general-purpose register: AH, AL, BH, BL, CH, CL, DH, DL</td>
</tr>
<tr>
<td>r16</td>
<td>16-bit general-purpose register: AX, BX, CX, DX, SI, DI, SP, BP</td>
</tr>
<tr>
<td>r32</td>
<td>32-bit general-purpose register: EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP</td>
</tr>
<tr>
<td>reg</td>
<td>any general-purpose register</td>
</tr>
<tr>
<td>sreg</td>
<td>16-bit segment register: CS, DS, SS, ES, FS, GS</td>
</tr>
<tr>
<td>imm</td>
<td>8-, 16-, or 32-bit immediate value</td>
</tr>
<tr>
<td>imm8</td>
<td>8-bit immediate byte value</td>
</tr>
<tr>
<td>imm16</td>
<td>16-bit immediate word value</td>
</tr>
<tr>
<td>imm32</td>
<td>32-bit immediate doubleword value</td>
</tr>
<tr>
<td>r/m8</td>
<td>8-bit operand which can be an 8-bit general register or memory byte</td>
</tr>
<tr>
<td>r/m16</td>
<td>16-bit operand which can be a 16-bit general register or memory word</td>
</tr>
<tr>
<td>r/m32</td>
<td>32-bit operand which can be a 32-bit general register or memory doubleword</td>
</tr>
<tr>
<td>mem</td>
<td>an 8-, 16-, or 32-bit memory operand</td>
</tr>
</tbody>
</table>
Direct Memory Operands

• A direct memory operand is a named reference to storage in memory
• The named reference (label) is automatically dereferenced by the assembler

```
.data
var1 BYTE 10h
.code
mov al,var1 ; AL = 10h
mov al,[var1] ; AL = 10h
```
MOV Instruction

- Move from source to destination. Syntax:
  \[ \text{MOV destination,source} \]
- Both operands must be the same size
- Cannot allow both operands be memory operands
- CS, EIP, and IP cannot be the destination
- Cannot move an immediate value to a segment register

```
.data
  count BYTE 100
  wVal WORD 2
.code
  mov bl, count
  mov ax, wVal
  mov count, al
  mov al, wVal ; error, size mismatch
  mov ax, count ; error, size mismatch
  mov eax, count ; error, size mismatch
```
Your turn . . .

Explain why each of the following MOV statements are invalid:

```
.data
bVal  BYTE   100
bVal2 BYTE   ?
wVal  WORD   2
dVal  DWORD  5
.code
    mov ds,45 ; why invalid?
    mov esi,wVal ; why invalid?
    mov eip,dVal ; why invalid?
    mov 25,bVal ; why invalid?
    mov bVal2,bVal ; why invalid?
```
Zero Extension

When you copy a smaller value into a larger destination, the MOVZX instruction fills (extends) the upper half of the destination with zeros.

```
mov bl,10001111b
movzx ax,bl ; zero-extension
```

The destination must be a register.
Sign Extension

The MOVSX instruction fills the upper half of the destination with a copy of the source operand's sign bit.

\[
\begin{array}{c}
\text{Source} \\
\hline
10001111 \\
\hline
\text{Destination} \\
\hline
11111111 10001111
\end{array}
\]

\texttt{mov bl,10001111b}
\texttt{movsx ax,bl} ; sign extension

The destination must be a register.
XCHG Instruction

XCHG exchanges the values of two operands. At least one operand must be a register. No immediate operands are permitted.

```assembly
.data
var1 WORD 1000h
var2 WORD 2000h
.code
xchg ax,bx ; exchange 16-bit regs
xchg ah,al ; exchange 8-bit regs
xchg var1,bx ; exchange mem, reg
xchg eax,ebx ; exchange 32-bit regs
xchg var1,var2 ; error: two memory operands
```

?
Direct-Offset Operands

A constant offset is added to a data label to produce an effective address (EA). The address is dereferenced to get the value inside its memory location.

```assembly
.data
arrayB BYTE 10h,20h,30h,40h
.code
Mov al,arrayB       ; AL = 10h
mov al,arrayB+1     ; AL = ?
mov al,[arrayB+1]   ; alternative notation
```

Q: Why does arrayB+1 produce 20h, rather than 11h?
Direct-Offset Operands (cont)

A constant offset is added to a data label to produce an effective address (EA). The address is dereferenced to get the value inside its memory location.

```
.data
arrayW  WORD 1000h,2000h,3000h
arrayD  DWORD 1,2,3,4
.code
mov ax,[arrayW+2]   ; AX = 2000h
mov ax,[arrayW+4]   ; AX = 3000h
mov eax,[arrayD+4]  ; EAX = 00000002h
```

; Will the following statements assemble?
mov ax,[arrayW-2]   ; ??
mov eax,[arrayD+16]  ; ??

What will happen when they run?
Your turn. . .

Write a program that rearranges the values of three doubleword values in the following array as: 3, 1, 2.

```
.data
arrayD DWORD 1,2,3
```

- Step1: copy the first value into EAX and exchange it with the value in the second position.

- Step 2: Exchange EAX with the third array value and copy the value in EAX to the first array position.
Evaluate this . . .

• We want to write a program that adds the following three bytes:
  ```assembly
  .data
  myBytes BYTE 80h, 66h, 0A5h
  ```

• What is your evaluation of the following code?
  ```assembly
  mov al, myBytes
  add al, [myBytes+1]
  add al, [myBytes+2]
  ```

• What is your evaluation of the following code?
  ```assembly
  mov ax, myBytes
  add ax, [myBytes+1]
  add ax, [myBytes+2]
  ```

• Any other possibilities?
Evaluate this . . . (cont)

.data
myBytes BYTE 80h, 66h, 0A5h

• How about the following code. Is anything missing?

```assembly
movzx ax, myBytes
mov  bl, [myBytes+1]
add  ax, bx
add  bl, [myBytes+2]
add  ax, bx ; AX = sum
```
What's Next

• Data Transfer Instructions
• **Addition and Subtraction**
• Data-Related Operators and Directives
• Indirect Addressing
• JMP and LOOP Instructions
Addition and Subtraction

- INC and DEC Instructions
- ADD and SUB Instructions
- NEG Instruction
- Implementing Arithmetic Expressions
- Flags Affected by Arithmetic
  - Zero
  - Sign
  - Carry
  - Overflow
INC and DEC Instructions

• Add 1, subtract 1 from destination operand
  – operand may be register or memory
• INC destination
  • Logic: destination ← destination + 1
• DEC destination
  • Logic: destination ← destination – 1
INC and DEC Examples

.data
myWord   WORD  1000h
myDword  DWORD 10000000h
.code
    inc myWord        ;  1001h
    dec myWord        ;  1000h
    inc myDword       ;  10000001h
    mov ax,00FFh
    inc ax            ; AX = 0100h
    mov ax,00FFh
    inc al            ; AX = 0000h
Your turn...

Show the value of the destination operand after each of the following instructions executes:

```assembly
.data
myByte BYTE 0FFh, 0
.code
    mov al,myByte  ; AL =
    mov ah,[myByte+1]  ; AH =
    dec ah  ; AH =
    inc al  ; AL =
    dec ax  ; AX =
```
ADD and SUB Instructions

• ADD destination, source
  • Logic: destination ← destination + source

• SUB destination, source
  • Logic: destination ← destination – source

• Same operand rules as for the MOV instruction
ADD and SUB Examples

```assembly
.data
var1 DWORD 10000h
var2 DWORD 20000h
.code
    ; ---EAX---
    mov eax, var1
    ; 00010000h
    add eax, var2
    ; 00030000h
    add ax, 0FFFFh
    ; 0003FFFFh
    add eax, 1
    ; 00040000h
    sub ax, 1
    ; 0004FFFFh
```
NEG (negate) Instruction

Reverses the sign of an operand. Operand can be a register or memory operand.

```assembly
.data
valB BYTE -1
valW WORD +32767
.code
  mov al, valB ; AL = -1
  neg al ; AL = +1
  neg valW ; valW = -32767
```

Suppose AX contains –32,768 and we apply NEG to it. Will the result be valid?
NEG Instruction and the Flags

The processor implements NEG using the following internal operation:

```
SUB 0, operand
```

Any nonzero operand causes the Carry flag to be set.

```
data
valB BYTE 1, 0
valC SBYTE -128
.code
    neg valB ; CF = 1, OF = 0
    neg [valB + 1] ; CF = 0, OF = 0
    neg valC ; CF = 1, OF = 1
```
Implementing Arithmetic Expressions

HLL compilers translate mathematical expressions into assembly language. You can do it also. For example:

\[ Rval = -Xval + (Yval - Zval) \]

```assembly
Rval DWORD ?
Xval DWORD 26
Yval DWORD 30
Zval DWORD 40
.code
    mov eax, Xval
    neg eax          ; EAX = -26
    mov ebx, Yval
    sub ebx, Zval   ; EBX = -10
    add eax, ebx
    mov Rval, eax   ; -36
```
Your turn...

Translate the following expression into assembly language. Do not permit Xval, Yval, or Zval to be modified:

\[ Rval = Xval - (\neg Yval + Zval) \]

Assume that all values are signed doublewords.
Flags Affected by Arithmetic

• The ALU has a number of status flags that reflect the outcome of arithmetic (and bitwise) operations based on the contents of the destination operand

• Essential flags:
  – Zero flag – set when destination equals zero
  – Sign flag – set when destination is negative
  – Carry flag – set when unsigned value is out of range
  – Overflow flag – set when signed value is out of range

• The MOV instruction never affects the flags.
You can use diagrams such as these to express the relationships between assembly language concepts.
Zero Flag (ZF)

The Zero flag is set when the result of an operation produces zero in the destination operand.

```
mov cx,1
sub cx,1  ; CX = 0, ZF = 1
mov ax,0FFFFh
inc ax    ; AX = 0, ZF = 1
inc ax    ; AX = 1, ZF = 0
```

Remember...
- A flag is **set** when it equals 1.
- A flag is **clear** when it equals 0.
Sign Flag (SF)

The Sign flag is set when the destination operand is negative. The flag is clear when the destination is positive.

```
  mov cx, 0
  sub cx, 1 ; CX = -1, SF = 1
  add cx, 2 ; CX = 1, SF = 0
```

The sign flag is a copy of the destination's highest bit:

```
  mov al, 0
  sub al, 1 ; AL = 11111111b, SF = 1
  add al, 2 ; AL = 00000001b, SF = 0
```
Signed and Unsigned Integers
A Hardware Viewpoint

- All CPU instructions operate exactly the same on signed and unsigned integers
- The CPU cannot distinguish between signed and unsigned integers
- YOU, the programmer, are solely responsible for using the correct data type with each instruction
Carry Flag (CF)

The Carry flag is set when the result of an operation generates an unsigned value that is out of range (too big or too small for the destination operand).

```
mov al,0FFh
add al,1 ; CF = 1, AL = 00

; Try to go below zero:

mov al,0
sub al,1 ; CF = 1, AL = FF
```
Your turn . . .

For each of the following marked entries, show the values of the destination operand and the Sign, Zero, and Carry flags:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>AX=</th>
<th>SF=</th>
<th>ZF=</th>
<th>CF=</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mov ax,00FFh</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>add ax,1</code></td>
<td>AX=</td>
<td>SF=</td>
<td>ZF=</td>
<td>CF=</td>
</tr>
<tr>
<td><code>sub ax,1</code></td>
<td>AX=</td>
<td>SF=</td>
<td>ZF=</td>
<td>CF=</td>
</tr>
<tr>
<td><code>add al,1</code></td>
<td>AL=</td>
<td>SF=</td>
<td>ZF=</td>
<td>CF=</td>
</tr>
<tr>
<td><code>mov bh,6Ch</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>add bh,95h</code></td>
<td>BH=</td>
<td>SF=</td>
<td>ZF=</td>
<td>CF=</td>
</tr>
<tr>
<td><code>mov al,2</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>sub al,3</code></td>
<td>AL=</td>
<td>SF=</td>
<td>ZF=</td>
<td>CF=</td>
</tr>
</tbody>
</table>
Overflow Flag (OF)

The Overflow flag is set when the signed result of an operation is invalid or out of range.

; Example 1
mov al,+127
add al,1 ; OF = 1, AL = ??

; Example 2
mov al,7Fh ; OF = 1, AL = 80h
add al,1

The two examples are identical at the binary level because 7Fh equals +127. To determine the value of the destination operand, it is often easier to calculate in hexadecimal.
A Rule of Thumb

• When adding two integers, remember that the Overflow flag is only set when . . .
  – Two positive operands are added and their sum is negative
  – Two negative operands are added and their sum is positive

What will be the values of the Overflow flag?

```assembly
mov al, 80h
add al, 92h ; OF = 1

mov al, -2
add al, +127 ; OF = 0
```
Your turn . . .

What will be the values of the given flags after each operation?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>CF</th>
<th>OF</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mov al, -128</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>neg al</code></td>
<td>CF =</td>
<td>OF =</td>
</tr>
<tr>
<td><code>mov ax, 8000h</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>add ax, 2</code></td>
<td>CF =</td>
<td>OF =</td>
</tr>
<tr>
<td><code>mov ax, 0</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>sub ax, 2</code></td>
<td>CF =</td>
<td>OF =</td>
</tr>
<tr>
<td><code>mov al, -5</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>sub al, +125</code></td>
<td>OF =</td>
<td></td>
</tr>
</tbody>
</table>
What's Next

• Data Transfer Instructions
• Addition and Subtraction
• **Data-Related Operators and Directives**
• Indirect Addressing
• JMP and LOOP Instructions
Data-Related Operators and Directives

- OFFSET Operator
- PTR Operator
- TYPE Operator
- LENGTHOF Operator
- SIZEOF Operator
- LABEL Directive
OFFSET Operator

- OFFSET returns the distance in bytes, of a label from the beginning of its enclosing segment
  - Protected mode: 32 bits
  - Real mode: 16 bits

The Protected-mode programs we write only have a single segment (we use the flat memory model).
OFFSET Examples

Let's assume that bVal is located at offset 00404000h:

```asm
.data
bVal BYTE ?
wVal WORD ?
dVal DWORD ?
dVal2 DWORD ?

.code
mov esi,OFFSET bVal ; ESI = 00404000
mov esi,OFFSET wVal ; ESI = 00404001
mov esi,OFFSET dVal ; ESI = 00404003
mov esi,OFFSET dVal2 ; ESI = 00404007
```
Relating to C/C++

The value returned by OFFSET is a pointer. Compare the following code written for both C++ and assembly language:

```c++
; C++ version:
char array[1000];
char * p = array;
```

```assembly
.data
array BYTE 1000 DUP(?)
.code
mov    esi,OFFSET array    ; ESI is p
```
PTR Operator

Overrides the default type of a label (variable). Provides the flexibility to access part of a variable.

```
.data
myDouble DWORD 12345678h
.code
mov ax,myDouble ; error- size mismatch

mov ax,WORD PTR myDouble ; loads 5678h
mov WORD PTR myDouble,4321h ; saves 4321h
```

Recall that little endian order is used when storing data in memory (see Section 3.4.9).
Little Endian Order

- Little endian order refers to the way Intel stores integers in memory.
- Multi-byte integers are stored in reverse order, with the least significant byte stored at the lowest address.
- For example, the doubleword 12345678h would be stored as:

<table>
<thead>
<tr>
<th>byte</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>78</td>
<td>0000</td>
</tr>
<tr>
<td>56</td>
<td>0001</td>
</tr>
<tr>
<td>34</td>
<td>0002</td>
</tr>
<tr>
<td>12</td>
<td>0003</td>
</tr>
</tbody>
</table>

When integers are loaded from memory into registers, the bytes are automatically re-reversed into their correct positions.
PTR Operator Examples

.data
myDouble DWORD 12345678h

<table>
<thead>
<tr>
<th>offset</th>
<th>doubleword</th>
<th>word</th>
<th>byte</th>
<th>myDouble</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>12345678</td>
<td>5678</td>
<td>78</td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td></td>
<td>56</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0002</td>
<td>1234</td>
<td>34</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0003</td>
<td></td>
<td>12</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

mov al, BYTE PTR myDouble ; AL = 78h
mov al, BYTE PTR [myDouble+1] ; AL = 56h
mov al, BYTE PTR [myDouble+2] ; AL = 34h
mov ax, WORD PTR myDouble ; AX = 5678h
mov ax, WORD PTR [myDouble+2] ; AX = 1234h
PTR Operator (cont)

PTR can also be used to combine elements of a smaller data type and move them into a larger operand. The CPU will automatically reverse the bytes.

```assembly
.data
myBytes BYTE 12h,34h,56h,78h

.code
mov ax,WORD PTR [myBytes] ; AX = 3412h
mov ax,WORD PTR [myBytes+2] ; AX = 7856h
mov eax,DWORD PTR myBytes ; EAX = 78563412h
```
Your turn . . .

Write down the value of each destination operand:

```assembly
.data
varB BYTE 65h,31h,02h,05h
varW WORD 6543h,1202h
varD DWORD 12345678h

.code
mov bl,BYTE PTR varD ; b.
mov bl,BYTE PTR [varW+2] ; c.
mov ax,WORD PTR [varD+2] ; d.
mov eax,DWORD PTR varW ; e.
```
The TYPE operator returns the size, in bytes, of a single element of a data declaration.

```
.data
var1 BYTE ?
var2 WORD ?
var3 DWORD ?
var4 QWORD ?

.code
mov eax,TYPE var1    ; 1
mov eax,TYPE var2    ; 2
mov eax,TYPE var3    ; 4
mov eax,TYPE var4    ; 8
```
LENGTHOF Operator

The LENGTHOF operator counts the number of elements in a single data declaration.

```assembly
.data
byte1  BYTE 10,20,30 ; 3
array1 WORD 30 DUP(?),0,0 ; 32
array2 WORD 5 DUP(3 DUP(?)) ; 15
array3 DWORD 1,2,3,4 ; 4
digitStr BYTE "12345678",0 ; 9

.code
mov ecx,LENGTHOF array1 ; 32
```
SIZEOF Operator

The SIZEOF operator returns a value that is equivalent to multiplying LENGTHOF by TYPE.

```
.data
byte1 BYTE 10,20,30 ; 3
array1 WORD 30 DUP(?),0,0 ; 64
array2 WORD 5 DUP(3 DUP(?)) ; 30
array3 DWORD 1,2,3,4 ; 16
digitStr BYTE "12345678",0 ; 9

.code
mov ecx,SIZEOF array1 ; 64
```
Spanning Multiple Lines (1 of 2)

A data declaration spans multiple lines if each line (except the last) ends with a comma. The LENGTHOF and SIZEOF operators include all lines belonging to the declaration:

```
.data
array WORD 10,20,
    30,40,
    50,60

.code
mov eax,LENGTHOF array ; 6
mov ebx,SIZEOF array    ; 12
```
Spanning Multiple Lines (2 of 2)

In the following example, array identifies only the first WORD declaration. Compare the values returned by LENGTHOF and SIZEOF here to those in the previous slide:

```
.data
array WORD 10,20
    WORD 30,40
    WORD 50,60

.code
mov eax,LENGTHOF array ; 2
mov ebx,SIZEOF array   ; 4
```
What's Next

• Data Transfer Instructions
• Addition and Subtraction
• Data-Related Operators and Directives
• **Indirect Addressing**
• JMP and LOOP Instructions
Indirect Addressing

• Indirect Operands
• Array Sum Example
• Indexed Operands
• Pointers
Indirect Operands (1 of 2)

An indirect operand holds the address of a variable, usually an array or string. It can be dereferenced (just like a pointer).

```
.data
val1 BYTE 10h,20h,30h
.code
mov esi,OFFSET val1
mov al,[esi]      ; dereference ESI (AL = 10h)
inc esi
mov al,[esi]      ; AL = 20h
inc esi
mov al,[esi]      ; AL = 30h
```
Indirect Operands (2 of 2)

Use PTR to clarify the size attribute of a memory operand.

```
.data
myCount WORD 0

.code
mov esi,OFFSET myCount
inc [esi] ; error: ambiguous
inc WORD PTR [esi] ; ok
```

Should PTR be used here?

```
add [esi],20
```

yes, because [esi] could point to a byte, word, or doubleword
Array Sum Example

Indirect operands are ideal for traversing an array. Note that the register in brackets must be incremented by a value that matches the array type.

```
.data
arrayW  WORD 1000h,2000h,3000h
.code
    mov esi,OFFSET arrayW
    mov ax,[esi]
    add esi,2 ; or: add esi,TYPE arrayW
    add ax,[esi]
    add esi,2
    add ax,[esi] ; AX = sum of the array
```

ToDo: Modify this example for an array of doublewords.
Indexed Operands

An indexed operand adds a constant to a register to generate an effective address. There are two notational forms:

\[ [\text{label} + \text{reg}] \quad \text{label}[\text{reg}] \]

```
.data
arrayW WORD 1000h,2000h,3000h
.code
  mov esi,0
  mov ax,[arrayW + esi] ; AX = 1000h
  mov ax,arrayW[esi]   ; alternate format
  add esi,2
  add ax,[arrayW + esi]
  etc.
```

ToDo: Modify this example for an array of doublewords.
Index Scaling*

You can scale an indirect or indexed operand to the offset of an array element. This is done by multiplying the index by the array's TYPE:

```assembly
.data
arrayB BYTE 0,1,2,3,4,5
arrayW WORD 0,1,2,3,4,5
arrayD DWORD 0,1,2,3,4,5

.code
mov esi,4
mov al,arrayB[esi*TYPE arrayB] ; 04
mov bx,arrayW[esi*TYPE arrayW] ; 0004
mov edx,arrayD[esi*TYPE arrayD] ; 00000004
```

* Not in the book
What's Next

• Data Transfer Instructions
• Addition and Subtraction
• Data-Related Operators and Directives
• Indirect Addressing
• JMP and LOOP Instructions
JMP and LOOP Instructions

• JMP Instruction
• LOOP Instruction
• LOOP Example
• Summing an Integer Array
• Copying a String
JMP Instruction

• JMP is an unconditional jump to a label that is usually within the same procedure.

• Syntax: **JMP target**

• Logic: EIP ← target

• Example:

```assembly
top:
  .
  .
  jmp top
```

A jump outside the current procedure must be to a special type of label called a **global label** (see Section 5.5.2.3 for details).
LOOP Instruction

• The LOOP instruction creates a counting loop
• Syntax: LOOP target
• Logic:
  • ECX ← ECX – 1
  • if ECX != 0, jump to target
• Implementation:
  • The assembler calculates the distance, in bytes, between the offset of the following instruction and the offset of the target label. It is called the relative offset.
  • The relative offset is added to EIP.
LOOP Example

The following loop calculates the sum of the integers $5 + 4 + 3 + 2 + 1$:

<table>
<thead>
<tr>
<th>offset</th>
<th>machine code</th>
<th>source code</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>66 B8 0000</td>
<td>mov ax, 0</td>
</tr>
<tr>
<td>00000004</td>
<td>B9 00000005</td>
<td>mov ecx, 5</td>
</tr>
<tr>
<td>00000009</td>
<td>66 03 C1</td>
<td>L1: add ax, cx</td>
</tr>
<tr>
<td>0000000C</td>
<td>E2 FB</td>
<td>loop L1</td>
</tr>
<tr>
<td>0000000E</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

When LOOP is assembled, the current location = 0000000E (offset of the next instruction). –5 (FBh) is added to the current location, causing a jump to location 00000009:

$$00000009 \leftarrow 0000000E + FB$$
Your turn . . .

If the relative offset is encoded in a single signed byte,
(a) what is the largest possible backward jump?
(b) what is the largest possible forward jump?

(a) ?
(b) ?
Your turn . . .

What will be the final value of AX?

mov ax, 6
mov ecx, 4
L1:
  inc ax
loop L1

How many times will the loop execute?

mov ecx, 0
X2:
  inc ax
loop X2
Nested Loop

If you need to code a loop within a loop, you must save the outer loop counter's ECX value. In the following example, the outer loop executes 100 times, and the inner loop 20 times.

```assembly
.data
count DWORD ?
.code
    mov ecx,100 ; set outer loop count
L1:
    mov count,ecx ; save outer loop count
    mov ecx,20 ; set inner loop count
L2:       .
    loop L2 ; repeat the inner loop
    mov ecx,count ; restore outer loop count
    loop L1 ; repeat the outer loop
```
Summing an Integer Array

The following code calculates the sum of an array of 16-bit integers.

```assembly
.data
intarray  WORD 100h, 200h, 300h, 400h
.code
    mov edi,OFFSET intarray ; address of intarray
    mov ecx, LENGTHOF intarray ; loop counter
    mov ax, 0 ; zero the accumulator
L1:
    add ax, [edi] ; add an integer
    add edi, TYPE intarray ; point to next integer
    loop L1 ; repeat until ECX = 0
```
Your turn . . .

What changes would you make to the program on the previous slide if you were summing a doubleword array?
Copying a String

The following code copies a string from source to target:

```assembly
.data
source  BYTE   "This is the source string",0
target BYTE SIZEOF source DUP(0)

.code
    mov esi,0           ; index register
    mov ecx,SIZEOF source ; loop counter
L1:
    mov al,source[esi]   ; get char from source
    mov target[esi],al    ; store it in the target
    inc esi              ; move to next character
    loop L1              ; repeat for entire string
```
Your turn . . .

Rewrite the program shown in the previous slide, using indirect addressing rather than indexed addressing.
Summary

• Data Transfer
  – MOV – data transfer from source to destination
  – MOVSX, MOVZX, XCHG

• Operand types
  – direct, direct-offset, indirect, indexed

• Arithmetic
  – INC, DEC, ADD, SUB, NEG
  – Sign, Carry, Zero, Overflow flags

• Operators
  – OFFSET, PTR, TYPE, LENGTHOF, SIZEOF, TYPEDEF

• JMP and LOOP – branching instructions