Chapter Overview

• **Boolean and Comparison Instructions**
• Conditional Jumps
• Conditional Loop Instructions
• Conditional Structures
• Application: Finite-State Machines
• Decision Directives
Boolean and Comparison Instructions

- CPU Status Flags
- AND Instruction
- OR Instruction
- XOR Instruction
- NOT Instruction
- Applications
- TEST Instruction
- CMP Instruction
Status Flags - Review

- The **Zero** flag is set when the result of an operation equals zero.
- The **Carry** flag is set when an instruction generates a result that is too large (or too small) for the destination operand.
- The **Sign** flag is set if the destination operand is negative, and it is clear if the destination operand is positive.
- The **Overflow** flag is set when an instruction generates an invalid signed result (bit 7 carry is XORed with bit 6 Carry).
- The **Parity** flag is set when an instruction generates an even number of 1 bits in the low byte of the destination operand.
- The **Auxiliary Carry** flag is set when an operation produces a carry out from bit 3 to bit 4.
AND Instruction

• Performs a Boolean AND operation between each pair of matching bits in two operands

• Syntax:

  \[ \text{AND } \text{destination, source} \]

  (same operand types as MOV)

\[
\begin{array}{c c c}
0 & 0 & 1 \\
0 & 0 & 1 \\
0 & 1 & 1 \\
1 & 0 & 1 \\
1 & 1 & 1 \\
\end{array}
\]

\[
\begin{array}{c c c}
x & y & x \land y \\
0 & 0 & 0 \\
0 & 1 & 0 \\
1 & 0 & 0 \\
1 & 1 & 1 \\
\end{array}
\]

Examples
OR Instruction

• Performs a Boolean OR operation between each pair of matching bits in two operands
• Syntax:
  \[ \text{OR \hspace{5mm} destination, source} \]

```
0 0 1 1 1 0 1 1
0 0 0 0 1 1 1 1

unchanged 0 0 1 1 1 1 1 1
set
```

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x ∨ y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

XOR Instruction

• Performs a Boolean exclusive-OR operation between each pair of matching bits in two operands

• Syntax:
  
  XOR destination, source

XOR is a useful way to toggle (invert) the bits in an operand.
NOT Instruction

- Performs a Boolean NOT operation on a single destination operand
- Syntax:
  \[
  \text{NOT } \text{destination}
  \]

\[
\begin{array}{c|c|c}
0 & 0 & 1 \\
1 & 1 & 1 \\
1 & 0 & 1 \\
1 & 0 & 0 \\
0 & 1 & 0 \\
1 & 1 & 1 \\
\end{array}
\]

inverted

\[
\begin{array}{c|c}
X & \neg X \\
F & T \\
T & F \\
\end{array}
\]
Applications (1 of 4)

• Task: Convert the character in AL to upper case.
• Solution: Use the AND instruction to clear bit 5.

```
mov al, 'a' ; AL = 01100001b
and al, 11011111b ; AL = 01000001b
```
Applications (2 of 4)

• Task: Convert a binary decimal byte into its equivalent ASCII decimal digit.

• Solution: Use the OR instruction to set bits 4 and 5.

\[
\begin{align*}
\text{mov } al, 6 & \quad \text{; AL = 00000110b} \\
\text{or } al, 00110000b & \quad \text{; AL = 00110110b}
\end{align*}
\]

The ASCII digit '6' = 00110110b
Applications (3 of 4)

- Task: Jump to a label if an integer is even.
- Solution: AND the lowest bit with a 1. If the result is Zero, the number was even.

```assembly
mov ax, wordVal
and ax, 1 ; low bit set?
jz EvenValue ; jump if Zero flag set
```

JZ (jump if Zero) is covered in Section 6.3.

Your turn: Write code that jumps to a label if an integer is negative.
Applications (4 of 4)

- Task: Jump to a label if the value in AL is not zero.
- Solution: OR the byte with itself, then use the JNZ (jump if not zero) instruction.

```assembly
or   al, al
jnz  IsNotZero ; jump if not zero
```

ORing any number with itself does not change its value.
TEST Instruction

• Performs a nondestructive AND operation between each pair of matching bits in two operands
• No operands are modified, but the Zero flag is affected.
• Example: jump to a label if either bit 0 or bit 1 in AL is set.

```
test al,00000011b
jnz  ValueFound
```

• Example: jump to a label if neither bit 0 nor bit 1 in AL is set.

```
test al,00000011b
jz   ValueNotFound
```
CMP Instruction (1 of 2)

• Compares the destination operand to the source operand
  – Nondestructive subtraction of source from destination (destination operand is not changed)

• Syntax: \texttt{CMP destination, source}

• Example: destination == source

  \begin{verbatim}
  mov al,5
  cmp al,5 ; Zero flag set
  \end{verbatim}

• Example: destination < source

  \begin{verbatim}
  mov al,4
  cmp al,5 ; Carry flag set
  \end{verbatim}

• Example: destination > source

  \begin{verbatim}
  mov al,6
  cmp al,5 ; ZF = 0, CF = 0
  \end{verbatim}

(both the Zero and Carry flags are clear)
The comparisons shown here are performed with signed integers.

- Example: destination > source

```assembly
mov al, 5
cmp al, -2 ; Sign flag == Overflow flag
```

- Example: destination < source

```assembly
mov al, -1
cmp al, 5 ; Sign flag != Overflow flag
```
What's Next

• Boolean and Comparison Instructions
• **Conditional Jumps**
• Conditional Loop Instructions
• Conditional Structures
• Application: Finite-State Machines
• Decision Directives
Conditional Jumps

• Jumps Based On . . .
  – Specific flags
  – Equality
  – Unsigned comparisons
  – Signed Comparisons

• Applications
• Encrypting a String
• Bit Test (BT) Instruction
J_{cond} Instruction

• A conditional jump instruction branches to a label when specific register or flag conditions are met

• Examples:
  – JB, JC jump to a label if the Carry flag is set
  – JE, JZ jump to a label if the Zero flag is set
  – JS jumps to a label if the Sign flag is set
  – JNE, JNZ jump to a label if the Zero flag is clear
  – JECXZ jumps to a label if ECX equals 0
J_{cond} Ranges

• Prior to the 386:
  – jump must be within −128 to +127 bytes from current location counter

• IA-32 processors:
  – 32-bit offset permits jump anywhere in memory
### Jumps Based on Specific Flags

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>JZ</td>
<td>Jump if zero</td>
<td>ZF = 1</td>
</tr>
<tr>
<td>JNZ</td>
<td>Jump if not zero</td>
<td>ZF = 0</td>
</tr>
<tr>
<td>JC</td>
<td>Jump if carry</td>
<td>CF = 1</td>
</tr>
<tr>
<td>JNC</td>
<td>Jump if not carry</td>
<td>CF = 0</td>
</tr>
<tr>
<td>JO</td>
<td>Jump if overflow</td>
<td>OF = 1</td>
</tr>
<tr>
<td>JNO</td>
<td>Jump if not overflow</td>
<td>OF = 0</td>
</tr>
<tr>
<td>JS</td>
<td>Jump if signed</td>
<td>SF = 1</td>
</tr>
<tr>
<td>JNS</td>
<td>Jump if not signed</td>
<td>SF = 0</td>
</tr>
<tr>
<td>JP</td>
<td>Jump if parity (even)</td>
<td>PF = 1</td>
</tr>
<tr>
<td>JNP</td>
<td>Jump if not parity (odd)</td>
<td>PF = 0</td>
</tr>
</tbody>
</table>
### Jumps Based on Equality

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JE</td>
<td>Jump if equal ((\text{leftOp} = \text{rightOp}))</td>
</tr>
<tr>
<td>JNE</td>
<td>Jump if not equal ((\text{leftOp} \neq \text{rightOp}))</td>
</tr>
<tr>
<td>JCXZ</td>
<td>Jump if (\text{CX} = 0)</td>
</tr>
<tr>
<td>JECXZ</td>
<td>Jump if (\text{ECX} = 0)</td>
</tr>
</tbody>
</table>

### Jumps Based on Unsigned Comparisons

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JA</td>
<td>Jump if above (if $leftOp &gt; rightOp$)</td>
</tr>
<tr>
<td>JNBE</td>
<td>Jump if not below or equal (same as JA)</td>
</tr>
<tr>
<td>JAE</td>
<td>Jump if above or equal (if $leftOp \geq rightOp$)</td>
</tr>
<tr>
<td>JNB</td>
<td>Jump if not below (same as JAE)</td>
</tr>
<tr>
<td>JB</td>
<td>Jump if below (if $leftOp &lt; rightOp$)</td>
</tr>
<tr>
<td>JNAE</td>
<td>Jump if not above or equal (same as JB)</td>
</tr>
<tr>
<td>JBE</td>
<td>Jump if below or equal (if $leftOp \leq rightOp$)</td>
</tr>
<tr>
<td>JNA</td>
<td>Jump if not above (same as JBE)</td>
</tr>
</tbody>
</table>
# Jumps Based on Signed Comparisons

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JG</td>
<td>Jump if greater (if ( \text{leftOp} &gt; \text{rightOp} ))</td>
</tr>
<tr>
<td>JNLE</td>
<td>Jump if not less than or equal (same as JG)</td>
</tr>
<tr>
<td>JGE</td>
<td>Jump if greater than or equal (if ( \text{leftOp} \geq \text{rightOp} ))</td>
</tr>
<tr>
<td>JNL</td>
<td>Jump if not less (same as JGE)</td>
</tr>
<tr>
<td>JL</td>
<td>Jump if less (if ( \text{leftOp} &lt; \text{rightOp} ))</td>
</tr>
<tr>
<td>JNGE</td>
<td>Jump if not greater than or equal (same as JL)</td>
</tr>
<tr>
<td>JLE</td>
<td>Jump if less than or equal (if ( \text{leftOp} \leq \text{rightOp} ))</td>
</tr>
<tr>
<td>JNG</td>
<td>Jump if not greater (same as JLE)</td>
</tr>
</tbody>
</table>
Applications  (1 of 5)

• Task: Jump to a label if unsigned EAX is greater than EBX
  • Solution: Use CMP, followed by JA

```assembly
cmp eax,ebx 
ja Larger
```

• Task: Jump to a label if signed EAX is greater than EBX
  • Solution: Use CMP, followed by JG

```assembly
cmp eax,ebx 
jg Greater
```
Applications (2 of 5)

• Jump to label L1 if unsigned EAX is less than or equal to Val1

```assembly
cmp eax, Val1
jbe L1 ; below or equal
```

• Jump to label L1 if signed EAX is less than or equal to Val1

```assembly
cmp eax, Val1
jle L1
```
Applications (3 of 5)

• Compare unsigned AX to BX, and copy the larger of the two into a variable named **Large**

```
mov Large, bx
cmp ax, bx
jna Next
mov Large, ax
Next:
```

• Compare signed AX to BX, and copy the smaller of the two into a variable named **Small**

```
mov Small, ax
cmp bx, ax
jnl Next
mov Small, bx
Next:
```
Applications  (4 of 5)

• Jump to label L1 if the memory word pointed to by ESI equals Zero

```
cmp WORD PTR [esi], 0
je  L1
```

• Jump to label L2 if the doubleword in memory pointed to by EDI is even

```
test DWORD PTR [edi], 1
jz  L2
```
Applications  (5 of 5)

• Task: Jump to label L1 if bits 0, 1, and 3 in AL are **all set**.

• Solution: Clear all bits except bits 0, 1, and 3. Then compare the result with 00001011 binary.

```
and al,00001011b       ; clear unwanted bits
cmp al,00001011b       ; check remaining bits
je  L1                ; all set? jump to L1
```
Your turn . . .

• Write code that jumps to label L1 if either bit 4, 5, or 6 is set in the BL register.
• Write code that jumps to label L1 if bits 4, 5, and 6 are all set in the BL register.
• Write code that jumps to label L2 if AL has even parity.
• Write code that jumps to label L3 if EAX is negative.
• Write code that jumps to label L4 if the expression (EBX – ECX) is greater than zero.
Encrypting a String

The following loop uses the XOR instruction to transform every character in a string into a new value.

```assembly
KEY = 239 ; can be any byte value
BUFMAX = 128
.data
buffer BYTE BUFMAX+1 DUP(0)
bufSize DWORD BUFMAX

.code
    mov ecx,bufSize ; loop counter
    mov esi,0 ; index 0 in buffer
L1:
    xor buffer[esi],KEY ; translate a byte
    inc esi ; point to next byte
    loop L1
```
String Encryption Program

• Tasks:
  – Input a message (string) from the user
  – Encrypt the message
  – Display the encrypted message
  – Decrypt the message
  – Display the decrypted message

View the Encrypt.asm program's source code. Sample output:

Enter the plain text: Attack at dawn.
Cipher text: «¢¢Äîä–Ä¢–ïÄÿü–Gs
Decrypted: Attack at dawn.
BT (Bit Test) Instruction

• Copies bit \( n \) from an operand into the Carry flag

• Syntax: \texttt{BT bitBase, n}
  – bitBase may be \texttt{r/m16} or \texttt{r/m32}
  – \( n \) may be \texttt{r16, r32}, or \texttt{imm8}

• Example: jump to label L1 if bit 9 is set in the AX register:

```assembly
-data
Semaphore WORD 10001000b
.code
bt semaphore, 7 ; CF=1
jc L1 ; jump to L1 if CF=1
```
BTC and BTR

• **BTC bitBase n**
  - Copies bit \( n \) from an operand into the Carry flag, and complements (toggle) bit \( n \)

• **BTR bitBase n**
  - Copies bit \( n \) from an operand into the Carry flag, and reset (clear) bit \( n \)

```assembly
.data
Semaphore WORD 10001000b

.code
btc semaphore, 7 ; CF=1, Semaphore = 00001000b
btc semaphore, 6 ; CF=0, Semaphore = 01001000b
btr semaphore, 6 ; CF=1, Semaphore = 00001000b
```
What's Next

• Boolean and Comparison Instructions
• Conditional Jumps
• **Conditional Loop Instructions**
• Conditional Structures
• Application: Finite-State Machines
• Decision Directives
Conditional Loop Instructions

• LOOPZ and LOOPE
• LOOPNZ and LOOPNE
LOOPZ and LOOPE

• Syntax:
  \[
  \text{LOOPE} \text{ destination} \\
  \text{LOOPZ} \text{ destination}
  \]

• Logic:
  – \( \text{ECX} \leftarrow \text{ECX} - 1 \)
  – if \( \text{ECX} > 0 \) and \( \text{ZF}=1 \), jump to \( \text{destination} \)

• Useful when scanning an array for the first element that does not match a given value.
LOOPNZ and LOOPNE

- LOOPNZ (LOOPNE) is a conditional loop instruction
- Syntax:
  - LOOPNZ destination
  - LOOPNE destination
- Logic:
  - ECX ← ECX – 1;
  - if ECX > 0 and ZF=0, jump to destination
- Useful when scanning an array for the first element that matches a given value.
LOOPNZ Example

The following code finds the first positive value in an array:

```assembly
.data
array SWORD -3,-6,-1,-10,10,30,40,4
sentinel SWORD 0
.code
    mov esi,OFFSET array
    mov ecx,LENGTHOF array
next:
    test WORD PTR [esi],8000h ; test sign bit
    pushfd ; push flags on stack
    add esi,TYPE array
    popfd ; pop flags from stack
    loopnz next ; continue loop
    jnz quit ; none found
    sub esi,TYPE array ; ESI points to value
quit:
```
Your turn . . .

Locate the first nonzero value in the array. If none is found, let ESI point to the sentinel value:

```assembly
.data
array SWORD 50 DUP(?)
sentinel SWORD 0FFFFh
.code
    mov esi,OFFSET array
    mov ecx,LENGTHOF array
L1: cmp WORD PTR [esi],0  ; check for zero
    (fill in your code here)
quit:
```

\begin{verbatim}
.data
array SWORD 50 DUP(?)
sentinel SWORD OFFFFh
.code
    mov esi,OFFSET array
    mov ecx,LENGTHOF array
L1:  ?  WORD PTR [esi],0 ; check for zero
    pushfd ; push flags on stack
    add esi,TYPE array
    popfd ; pop flags from stack
    ? L1 ; continue loop
    ? quit
    quit:
\end{verbatim}
What's Next

• Boolean and Comparison Instructions
• Conditional Jumps
• Conditional Loop Instructions
• **Conditional Structures**
• Application: Finite-State Machines
• Decision Directives
Conditional Structures

- Block-Structured IF Statements
- Compound Expressions with AND
- Compound Expressions with OR
- WHILE Loops
- Table-Driven Selection
Block-Structured IF Statements

Assembly language programmers can easily translate logical statements written in C++/Java into assembly language. For example:

```c
if( op1 == op2 )
    X = 1;
else
    X = 2;
```

```assembly
mov eax,op1
cmp eax,op2
jne L1
mov X,1
jmp L2
L1: mov X,2
L2:
```
Your turn . . .

Implement the following pseudocode in assembly language. All values are unsigned:

```plaintext
if( ebx <= ecx )
{
    eax = 5;
    edx = 6;
}
```

(There are multiple correct solutions to this problem.)
Your turn . . .

Implement the following pseudocode in assembly language. All values are 32-bit signed integers:

```assembly
if( var1 <= var2 )
    var3 = 10;
else
    {
        var3 = 6;
        var4 = 7;
    }
```

(There are multiple correct solutions to this problem.)
Compound Expression with AND (1 of 3)

• When implementing the logical AND operator, consider that HLLs use short-circuit evaluation
• In the following example, if the first expression is false, the second expression is skipped:

```c
if (al > bl) AND (bl > cl)
    X = 1;
```
Compound Expression with AND

\[
\text{if } (al > bl) \text{ AND } (bl > cl) \\
X = 1;
\]

This is one possible implementation . . .

```
cmp al,bl ; first expression...
ja L1
jmp next

L1:  
cmp bl,cl ; second expression...
ja L2
jmp next

L2: ; both are true
mov X,1 ; set X to 1

next:
```
Compound Expression with AND (3 of 3)

```plaintext
if (al > bl) AND (bl > cl)
    X = 1;
```

But the following implementation uses 29% less code by reversing the first relational operator. We allow the program to "fall through" to the second expression:

```plaintext
cmp al, bl ; first expression...
jbe next ; quit if false
cmp bl, cl ; second expression...
jbe next ; quit if false
mov X, 1 ; both are true
next:
```

Your turn . . .

Implement the following pseudocode in assembly language. All values are unsigned:

```assembly
if( ebx <= ecx && ecx > edx ) {
    eax = 5;
    edx = 6;
}
```

(There are multiple correct solutions to this problem.)
Compound Expression with OR (1 of 2)

• When implementing the logical OR operator, consider that HLLs use short-circuit evaluation
• In the following example, if the first expression is true, the second expression is skipped:

```plaintext
if (al > bl) OR (bl > cl)  
  X = 1;
```
Compound Expression with OR  (1 of 2)

if (al > bl) OR (bl > cl)
    X = 1;

We can use "fall-through" logic to keep the code as short as possible:

    cmp al,bl      ; is AL > BL?
    ja L1         ; yes
    cmp bl,cl      ; no: is BL > CL?
    jbe next      ; no: skip next statement
    L1: mov X,1    ; set X to 1
next:
WHILE Loops

A WHILE loop is really an IF statement followed by the body of the loop, followed by an unconditional jump to the top of the loop. Consider the following example:

```assembly
while( eax < ebx)
    eax = eax + 1;
```

This is a possible implementation:

```assembly
top: cmp eax, ebx
    jae next
    inc eax
    jmp top
next:
```

; check loop condition
; false? exit loop
; body of loop
; repeat the loop
Your turn . . .

Implement the following loop, using unsigned 32-bit integers:

```assembly
while( ebx <= val1 )
{
    ebx = ebx + 5;
    val1 = val1 - 1
}
```

```assembly
top: cmp ebx, val1 ; check loop condition
    ja    next ; false? exit loop
    add ebx, 5 ; body of loop
    dec val1
    jmp top ; repeat the loop
next:
```

Table-Driven Selection (1 of 3)

- Table-driven selection uses a table lookup to replace a multiway selection structure.
- Create a table containing lookup values and the offsets of labels or procedures.
- Use a loop to search the table.
- Suited to a large number of comparisons.
Step 1: create a table containing lookup values and procedure offsets:

```assembly
.data
CaseTable BYTE 'A' ; lookup value
    DWORD Process_A ; address of procedure
EntrySize = ($ - CaseTable)
    BYTE 'B'
    DWORD Process_B
    BYTE 'C'
    DWORD Process_C
    BYTE 'D'
    DWORD Process_D

NumberOfEntries = ($ - CaseTable) / EntrySize
```
Table-Driven Selection  (3 of 3)

Step 2: Use a loop to search the table. When a match is found, we call the procedure offset stored in the current table entry:

```
mov ebx,OFFSET CaseTable ; point EBX to the table
mov ecx,NumberOfEntries ; loop counter

L1: cmp al,[ebx] ; match found?
jne L2 ; no: continue
    call NEAR PTR [ebx + 1] ; yes: call the procedure
    jmp L3 ; and exit the loop
L2: add ebx,EntrySize ; point to next entry
    loop L1 ; repeat until ECX = 0
L3: required for procedure pointers

```
What's Next

- Boolean and Comparison Instructions
- Conditional Jumps
- Conditional Loop Instructions
- Conditional Structures
- **Application: Finite-State Machines**
- Decision Directives
Application: Finite-State Machines

• A finite-state machine (FSM) is a graph structure that changes state based on some input. Also called a *state-transition diagram*.

• We use a graph to represent an FSM, with squares or circles called *nodes*, and lines with arrows between the circles called *edges* (or *arcs*).

• A FSM is a specific instance of a more general structure called a *directed graph* (or *digraph*)

• Three basic states, represented by nodes:
  – Start state
  – Terminal state(s)
  – Nonterminal state(s)
Finite-State Machine

• Accepts any sequence of symbols that puts it into an accepting (final) state
• Can be used to recognize, or validate a sequence of characters that is governed by language rules (called a regular expression)
• Advantages:
  – Provides visual tracking of program's flow of control
  – Easy to modify
  – Easily implemented in assembly language
FSM Examples

• FSM that recognizes strings beginning with 'x', followed by letters 'a'..'y', ending with 'z':

• FSM that recognizes signed integers:
Your turn . . .

• Explain why the following FSM does not work as well for signed integers as the one shown on the previous slide:
Implementing an FSM

Translate the FSM into assembly language code – each state in the diagram is represented in the program by a label.

The following is code from State A in the Integer FSM:

```
StateA:
    call Getnext ; read next char into AL
    cmp al, '+' ; leading + sign?
    je StateB   ; go to State B
    cmp al, '-' ; leading - sign?
    je StateB   ; go to State B
    call IsDigit ; ZF = 1 if AL = digit
    jz StateC   ; go to State C
    call DisplayErrorMsg ; invalid input found
    jmp Quit
```

View the Finite.asm source code.

IsDigit Procedure

Receives a character in AL. Sets the Zero flag if the character is a decimal digit.

```
IsDigit PROC
    cmp    al,'0' ; ZF = 0
    jb    ID1
    cmp    al,'9' ; ZF = 0
    ja    ID1
    test   ax,0 ; ZF = 1
ID1: ret
IsDigit ENDP
```

ax AND 0 = 0, so ZF = 1
Flowchart of State A

State A accepts a plus or minus sign, or a decimal digit.
Your turn . . .

• Draw a FSM diagram for hexadecimal integer constant that conforms to MASM syntax.
• Draw a flowchart for one of the states in your FSM.
• Implement your FSM in assembly language. Let the user input a hexadecimal constant from the keyboard.
What's Next

• Boolean and Comparison Instructions
• Conditional Jumps
• Conditional Loop Instructions
• Conditional Structures
• Application: Finite-State Machines
• Decision Directives
Using the .IF Directive

- Runtime Expressions
- Relational and Logical Operators
- MASM-Generated Code
- .REPEAT Directive
- .WHILE Directive
Runtime Expressions

• .IF, .ELSE, .ELSEIF, and .ENDIF can be used to evaluate runtime expressions and create block-structured IF statements.

• Examples:

  .IF eax > ebx
  mov edx, 1
  .ELSE
  mov edx, 2
  .ENDIF

  .IF eax > ebx && eax > ecx
  mov edx, 1
  .ELSE
  mov edx, 2
  .ENDIF

• MASM generates "hidden" code for you, consisting of code labels, CMP and conditional jump instructions.
## Relational and Logical Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$expr1 == expr2$</td>
<td>Returns true when $expression1$ is equal to $expr2$.</td>
</tr>
<tr>
<td>$expr1 != expr2$</td>
<td>Returns true when $expr1$ is not equal to $expr2$.</td>
</tr>
<tr>
<td>$expr1 &gt; expr2$</td>
<td>Returns true when $expr1$ is greater than $expr2$.</td>
</tr>
<tr>
<td>$expr1 &gt;= expr2$</td>
<td>Returns true when $expr1$ is greater than or equal to $expr2$.</td>
</tr>
<tr>
<td>$expr1 &lt; expr2$</td>
<td>Returns true when $expr1$ is less than $expr2$.</td>
</tr>
<tr>
<td>$expr1 &lt;= expr2$</td>
<td>Returns true when $expr1$ is less than or equal to $expr2$.</td>
</tr>
<tr>
<td>$! expr$</td>
<td>Returns true when $expr$ is false.</td>
</tr>
<tr>
<td>$expr1 &amp;&amp; expr2$</td>
<td>Performs logical AND between $expr1$ and $expr2$.</td>
</tr>
<tr>
<td>$expr1</td>
<td></td>
</tr>
<tr>
<td>$expr1 &amp; expr2$</td>
<td>Performs bitwise AND between $expr1$ and $expr2$.</td>
</tr>
<tr>
<td>CARRY?</td>
<td>Returns true if the Carry flag is set.</td>
</tr>
<tr>
<td>OVERFLOW?</td>
<td>Returns true if the Overflow flag is set.</td>
</tr>
<tr>
<td>PARITY?</td>
<td>Returns true if the Parity flag is set.</td>
</tr>
<tr>
<td>SIGN?</td>
<td>Returns true if the Sign flag is set.</td>
</tr>
<tr>
<td>ZERO?</td>
<td>Returns true if the Zero flag is set.</td>
</tr>
</tbody>
</table>
MASM automatically generates an unsigned jump (JBE) because `val1` is unsigned.
MASM-Generated Code

.data
val1  SDWORD  5
result  SDWORD ?
.code
mov eax,6
.cmp eax,val1
.jle  @C0001
Generated code:
 Generated code:  
mov eax,6
cmp eax,val1
  jle  @C0001
mov result,1
@C0001:
mov result,1

MASM automatically generates a signed jump (JLE) because val1 is signed.
MASM automatically generates an unsigned jump (JBE) when both operands are registers . . .
MASM-Generated Code

.data
result SDWORD ?
.code
mov ebx,5
mov eax,6
.If SDWORD PTR eax > ebx
    mov result,1
.endif

Generated code:

mov ebx,5
mov eax,6
cmp eax,ebx
jle @C0001
mov result,1
@C0001:

... unless you prefix one of the register operands with the SDWORD PTR operator. Then a signed jump is generated.
.REPEAT Directive

Executes the loop body before testing the loop condition associated with the .UNTIL directive.

Example:

; Display integers 1 – 10:

    mov eax, 0
    .REPEAT
        inc eax
        call WriteDec
        call Crlf
    .UNTIL eax == 10
.WHILE Directive

Tests the loop condition before executing the loop body.

The .ENDW directive marks the end of the loop.

Example:

```
; Display integers 1 - 10:

mov eax,0
.WHILE eax < 10
    inc eax
    call WriteDec
    call Crlf
.ENDW
```
Summary

• Bitwise instructions (AND, OR, XOR, NOT, TEST)
  – manipulate individual bits in operands
• CMP – compares operands using implied subtraction
  – sets condition flags
• Conditional Jumps & Loops
  – equality: JE, JNE
  – flag values: JC, JZ, JNC, JP, ...
  – signed: JG, JL, JNG, ...
  – unsigned: JA, JB, JNA, ...
  – LOOPZ, LOOPNZ, LOOPE, LOOPNE
• Flowcharts – logic diagramming tool
• Finite-state machine – tracks state changes at runtime