Conditional Processing

Computer Organization and Assembly Languages

Yung-Yu Chuang

2006/11/13

with slides by Kip Irvine

Assignment #2 CRC32 checksum

```
unsigned int crc32(const char* data,
                   size_t length)
{
  // standard polynomial in CRC32
  const unsigned int POLY = 0xEDB88320;
  // standard initial value in CRC32
  unsigned int reminder = 0xFFFFFFFF;
  for(size_t i = 0; i < length; i++){
    // must be zero extended
    reminder ^= (unsigned char)data[i];
    for(size_t bit = 0; bit < 8; bit++)
      if(reminder & 0x01)
        reminder = (reminder >> 1) ^ POLY;
      else
        reminder >>= 1;
  }
  return reminder ^ 0xFFFFFFFF;
}
```

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.
- Less important:
  - 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

NOT instruction

- Performs a bitwise Boolean NOT operation on a single destination operand
- Syntax: (no flag affected)

```
NOT destination
```

- Example:

```
mov al, 11110000b
not al
```

NOT 0 0 1 1 1 0 1 1

1 1 0 0 0 1 0 0 — inverted

NOT

X	¬X
F	T
T	F

AND instruction

- Performs a bitwise Boolean AND operation between each pair of matching bits in two operands
- Syntax: (O=0,C=0,SZP)

```
AND destination, source
```

- Example:

```
mov al, 00111011b
and al, 00001111b
```

0 0 1 1 1 0 1 1

AND 0 0 0 0 1 1 1 1

cleared — 0 0 0 0 1 0 1 1 — unchanged

bit extraction

AND

x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

OR instruction

- Performs a bitwise Boolean OR operation between each pair of matching bits in two operands
- Syntax: (O=0,C=0,SZP)

```
OR destination, source
```

- Example:

```
mov dl, 00111011b
or  dl, 00001111b
```

0 0 1 1 1 0 1 1

OR 0 0 0 0 1 1 1 1

unchanged — 0 0 1 1 1 1 1 1 — set

OR

x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

XOR instruction

- Performs a bitwise Boolean exclusive-OR operation between each pair of matching bits in two operands
- Syntax: (O=0,C=0,SZP)

```
XOR destination, source
```

- Example:

```
mov dl, 00111011b
xor dl, 00001111b
```

0 0 1 1 1 0 1 1

XOR 0 0 0 0 1 1 1 1

unchanged — 0 0 1 1 0 1 0 0 — inverted

XOR is a useful way to invert the bits in an operand and data encryption

XOR

x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

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.

```
mov al,6            ; AL = 00000110b
or  al,00110000b    ; AL = 00110110b
```

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.

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

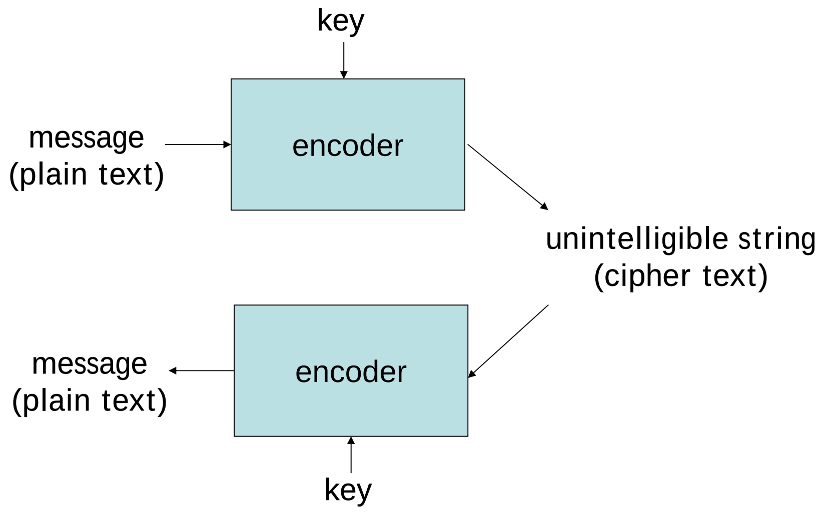
Applications (4 of 4)

- 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.

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

ORing any number with itself does not change its value.

String encryption



Encrypting a string

```
KEY = 239
.data
buffer BYTE BUFMAX DUP(0)
bufSize DWORD $-buffer
.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
```

```
Message: Attack at dawn.
Cipher text: «¢¢ÄÎä-Ä¢-ïÄÿü-Gs
Decrypted: Attack at dawn.
```

TEST instruction

- Performs a nondestructive **AND** operation between each pair of matching bits in two operands
- No operands are modified, but the flags are 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 3)

- Compares the destination operand to the source operand
  - Nondestructive subtraction of source from destination (destination operand is not changed)
- Syntax: (OSZCAP)

  CMP *destination, source*
- Example: destination == source

```
mov al,5
cmp al,5          ; Zero flag set
```

- Example: destination < source

```
mov al,4
cmp al,5          ; Carry flag set
```

CMP instruction (2 of 3)

- Example: destination > source

```
mov al,6
cmp al,5          ; ZF = 0, CF = 0
```

(both the Zero and Carry flags are clear)

The comparisons shown so far were unsigned.

CMP instruction (3 of 3)

The comparisons shown here are performed with signed integers.

- Example: destination > source

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

- Example: destination < source

```
mov al,-1
cmp al,5       ; Sign flag != Overflow flag
```

Conditions

unsigned	ZF	CF
destination<source	0	1
destination>source	0	0
destination=source	1	0

signed	flags
destination<source	SF != OF
destination>source	SF == OF
destination=source	ZF=1

Setting and clearing individual flags

```
and al, 0        ; set Zero
or  al, 1        ; clear Zero
or  al, 80h      ; set Sign
and al, 7Fh      ; clear Sign
stc              ; set Carry
clc              ; clear Carry

mov al, 7Fh
inc al           ; set Overflow

or eax, 0        ; clear Overflow
```

Conditional jumps

Conditional structures

- There are no high-level logic structures such as if-then-else, in the IA-32 instruction set. But, you can use combinations of comparisons and jumps to implement any logic structure.
- First, an operation such as **CMP**, **AND** or **SUB** is executed to modified the CPU flags. Second, a conditional jump instruction tests the flags and changes the execution flow accordingly.

```
      CMP AL, 0
      JZ  L1
      :
L1:
```

J*cond* instruction

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

  Jcond *destination*

- Four groups: (some are the same)

1. based on specific flag values
2. based on equality between operands
3. based on comparisons of unsigned operands
4. based on comparisons of signed operands

Jumps based on specific flags

Mnemonic	Description	Flags
JZ	Jump if zero	ZF = 1
JNZ	Jump if not zero	ZF = 0
JC	Jump if carry	CF = 1
JNC	Jump if not carry	CF = 0
JO	Jump if overflow	OF = 1
JNO	Jump if not overflow	OF = 0
JS	Jump if signed	SF = 1
JNS	Jump if not signed	SF = 0
JP	Jump if parity (even)	PF = 1
JNP	Jump if not parity (odd)	PF = 0

Jumps based on equality

Mnemonic	Description
JE	Jump if equal (*leftOp* = *rightOp*)
JNE	Jump if not equal (*leftOp* ≠ *rightOp*)
JCXZ	Jump if CX = 0
JECXZ	Jump if ECX = 0

Jumps based on unsigned comparisons

Mnemonic	Description
JA	Jump if above (if *leftOp* > *rightOp*)
JNBE	Jump if not below or equal (same as JA)
JAE	Jump if above or equal (if *leftOp* >= *rightOp*)
JNB	Jump if not below (same as JAE)
JB	Jump if below (if *leftOp* < *rightOp*)
JNAE	Jump if not above or equal (same as JB)
JBE	Jump if below or equal (if *leftOp* <= *rightOp*)
JNA	Jump if not above (same as JBE)

＞≧＜≦

Jumps based on signed comparisons

Mnemonic	Description
JG	Jump if greater (if *leftOp* > *rightOp*)
JNLE	Jump if not less than or equal (same as JG)
JGE	Jump if greater than or equal (if *leftOp* >= *rightOp*)
JNL	Jump if not less (same as JGE)
JL	Jump if less (if *leftOp* < *rightOp*)
JNGE	Jump if not greater than or equal (same as JL)
JLE	Jump if less than or equal (if *leftOp* <= *rightOp*)
JNG	Jump if not greater (same as JLE)

Examples

- 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:
```

Examples

- Find the first even number in an array of unsigned integers

```
.date
intArray DWORD 7,9,3,4,6,1
.code
...
        mov  ebx, OFFSET intArray
        mov  ecx, LENGTHOF intArray
L1:     test DWORD PTR [ebx], 1
        jz   found
        add  ebx, 4
        loop L1
...
```

BT (Bit Test) instruction

- Copies bit *n* from an operand into the Carry flag
- Syntax: **BT *bitBase, n***
  - bitBase may be *r/m16* or *r/m32*
  - n may be *r16, r32*, or *imm8*
- Example: jump to label L1 if bit 9 is set in the AX register:

```
bt AX,9          ; CF = bit 9
jc L1            ; jump if Carry
```

- **BTC *bitBase, n:*** bit test and complement
- **BTR *bitBase, n:*** bit test and reset (clear)
- **BTS *bitBase, n:*** bit test and set

Conditional loops

LOOPZ and LOOPE

- Syntax:

  LOOPE *destination*

  LOOPZ *destination*
- Logic:
  - ECX ← ECX – 1
  - if ECX != 0 and ZF=1, jump to *destination*
- The destination label must be between -128 and +127 bytes from the location of the following instruction
- Useful when scanning an array for the first element that meets some condition.

LOOPNZ and LOOPNE

- Syntax:

  LOOPNZ *destination*

  LOOPNE *destination*
- Logic:
  - ECX ← ECX – 1;
  - if ECX != 0 and ZF=0, jump to *destination*

LOOPNZ example

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

```
.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:

```
.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




quit:
```

Solution

```
.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
   pushfd                ; push flags on stack
   add esi,TYPE array
   Popfd                 ; pop flags from stack
   loope next            ; continue loop
   jz quit               ; none found
   sub esi,TYPE array    ; ESI points to value
quit:
```

Conditional structures

Block-structured IF statements

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

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

```
    mov eax,op1
    cmp eax,op2
    jne L1
    mov X,1
    jmp L2
L1: mov X,2
L2:
```

Example

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

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

```
      cmp ebx,ecx
      ja  next
      mov eax,5
      mov edx,6
next:
```

Example

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

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

```
    mov eax,var1
    cmp eax,var2
    jle L1
    mov var3,6
    mov var4,7
    jmp L2
L1: mov var3,10
L2:
```

Compound expression with AND

- 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:

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

Compound expression with AND

```
if (al > bl) 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

```
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:

```
   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:

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

```
        cmp ebx,ecx
        ja  next
        cmp ecx,edx
        jbe next
        mov eax,5
        mov edx,6
next:
```

(There are multiple correct solutions to this problem.)

Compound Expression with OR

- In the following example, if the first expression is true, the second expression is skipped:

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

Compound Expression with OR

```
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:

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

```
_while:
    cmp eax,ebx    ; check loop condition
    jae _endwhile  ; false? exit loop
    inc eax        ; body of loop
    jmp _while     ; repeat the loop
_endwhile:
```

Your turn . . .

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

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

```
_while:
    cmp ebx,val1   ; check loop condition
    ja  _endwhile  ; false? exit loop
    add ebx,5      ; body of loop
    dec val1
    jmp while      ; repeat the loop
_endwhile:
```

Example: IF statement nested in a loop

```
while(eax < ebx)
{
  eax++;
  if (ebx==ecx)
    X=2;
  else
    X=3;
}
```

```
_while:   cmp   eax, ebx
          jae   _endwhile
          inc   eax
          cmp   ebx, ecx
          jne   _else
          mov   X, 2
          jmp   _while
_else:    mov   X, 3
          jmp   _while
_endwhile:
```

Table-driven selection

- Table-driven selection uses a table lookup to replace a multiway selection structure (switch-case statements in C)
- 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

Table-driven selection

Step 1: create a table containing lookup values and procedure offsets:

```
.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

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

Application: finite-state machines

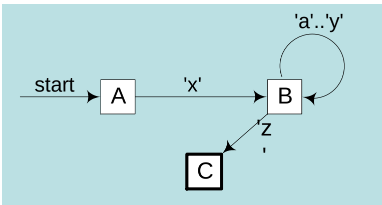
- A finite-state machine (FSM) is a graph structure that changes state based on some input. Also called a state-transition diagram.
- We use a graph to represent an FSM, with squares or circles called nodes, and lines with arrows between the circles called edges (or arcs).
- A FSM is a specific instance 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 machines

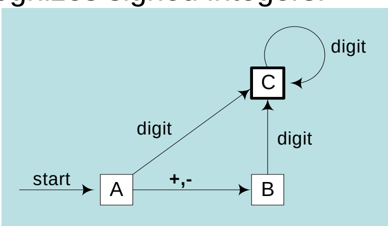
- Accepts any sequence of symbols that puts it into an accepting (final) state
- Can be used to recognize, or validate a sequence of characters that is governed by language rules (called a regular expression)

FSM Examples

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

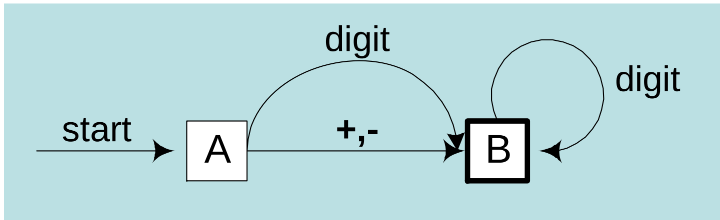


- FSM that recognizes signed integers:

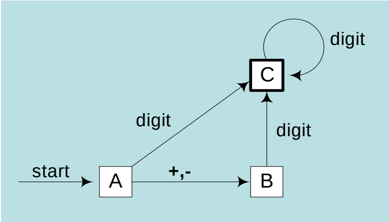


Your turn . . .

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



Implementing an FSM



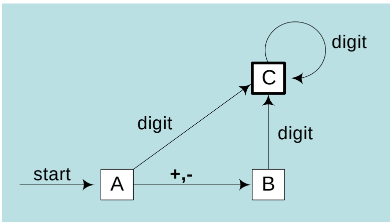
The following is code from State A in the Integer FSM:

```
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
```

Isdigit

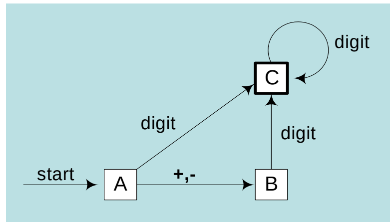
```
Isdigit PROC
     cmp al,'0'
     jb    L1
     cmp al,'9'
     ja L1
     test ax,0
L1: ret
Isdigit ENDP
```

Your turn




```
StateB:
  call  Getnext     ; read next char into AL
  call  Isdigit     ; ZF = 1 if AL is a digit
  jz    StateC
  call  DisplayErrorMsg  ; invalid input found
  jmp   Quit
```

Implementing an FSM




```
StateC:
  call  Getnext    ; read next char into AL
  jz    Quit       ; quit if Enter pressed
  call  Isdigit    ; ZF = 1 if AL is digit
  jz    StateC
  cmp   AL,ENTER_KEY ; Enter key pressed?
  je    Quit             ; yes: quit
  call  DisplayErrorMsg ; no: invalid input
  jmp   Quit
```

Finite-state machine example

- [*sign*]*integer*.[*integer*][*exponent*]

  sign → {+|-}

  exponent → E[{+|-}]integer

High-level directives

- .IF, .ELSE, .ELSEIF, and .ENDIF can be used to 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

Operator	Description
expr1 == *expr2*	Returns true when *expression1* is equal to *expr2*.
expr1 != *expr2*	Returns true when *expr1* is not equal to *expr2*.
expr1 > *expr2*	Returns true when *expr1* is greater than *expr2*.
expr1 >= *expr2*	Returns true when *expr1* is greater than or equal to *expr2*.
expr1 < *expr2*	Returns true when *expr1* is less than *expr2*.
expr1 <= *expr2*	Returns true when *expr1* is less than or equal to *expr2*.
! *expr*	Returns true when *expr* is false.
expr1 && *expr2*	Performs logical AND between *expr1* and *expr2*.
expr1 || *expr2*	Performs logical OR between *expr1* and *expr2*.
expr1 & *expr2*	Performs bitwise AND between *expr1* and *expr2*.
CARRY?	Returns true if the Carry flag is set.
OVERFLOW?	Returns true if the Overflow flag is set.
PARITY?	Returns true if the Parity flag is set.
SIGN?	Returns true if the Sign flag is set.
ZERO?	Returns true if the Zero flag is set.

MASM-generated Code

```
.data
val1   DWORD 5
result DWORD ?
.code
mov eax,6
.IF eax > val1
mov result,1
.ENDIF
```

Generated code:

```
   mov eax,6
   cmp eax,val1
   jbe @C0001
   mov result,1
@C0001:
```

MASM automatically generates an unsigned jump (JBE).

The use of signed or unsigned comparison depends on data type. If not defined (such as **.IF eax>ebx**), MASM sue unsigned comparisons.

.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
```