Getting started with MASM

- Download Visual studio
- Setup Visual studio: https://www.youtube.com/watch?v=-fCyvipptZU
  - Start without debugging
  - C++ configuration

Assembly Language for x86 Processors

- Branches and Conditions

  To implement condition (if-else, switch-case), and loop (for, while) structures.

Outline

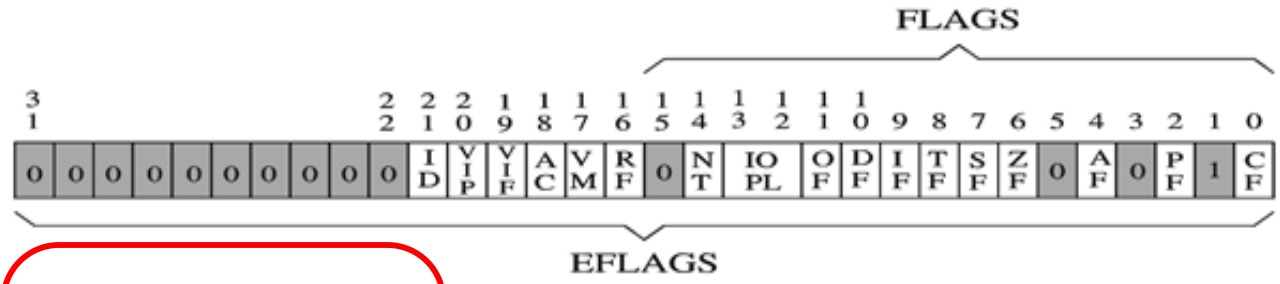
- Boolean and Comparison Instructions
- Conditional Jumps
- Conditional Structures
- The LOOP instruction

Boolean and Comparison Instructions

Operation	Description
AND	Boolean AND operation between a source operand and a destination operand.
OR	Boolean OR operation between a source operand and a destination operand.
XOR	Boolean exclusive-OR operation between a source operand and a destination operand.
NOT	Boolean NOT operation on a destination operand.
TEST	Implied boolean AND operation between a source and destination operand, setting the CPU flags appropriately.

Status Flags

- Boolean instructions affect the Zero, Carry, Sign, Overflow, and Parity flags



Status flags

CF = Carry flag
PF = Parity flag
AF = Auxiliary carry flag
ZF = Zero flag
SF = Sign flag
OF = Overflow flag

Control flags

DF = Direction flag

1 xxxx <-- xxxx

System flags

TF = Trap flag
IF = Interrupt flag
IOPL = I/O privilege level
NT = Nested task
RF = Resume flag
VM = Virtual 8086 mode
AC = Alignment check
VIF = Virtual interrupt flag
VIP = Virtual interrupt pending
ID = ID flag

Status Flags

- 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. for unsigned integer; ignore OF and SF
- The Sign flag is set if the destination operand is negative, and it is cleared if the destination operand is positive.
- The Overflow flag is set/detected when an instruction generates an invalid signed result. for signed integer; see OF and SF

msb(X)=X`

OF=(X` !^ Y`)^Z`

z = x + y all in 2c

x, y sign different ==> valid z; OF detection not needed

x, y, z sign same ==> valid z; OF detection not needed

x, y sign same but z sign different ==> invalid z; detect OF

+X +Y (CF=0, SF=1) ==> Z=result + 2^8

-X -Y (CF=1, SF=0) ==> Z=result - 2^8

Summary: CF, OF (US, Signed int)

For Z=X+Y

Analyzing the combination of MSBs of the operands and related results' combinations possible, and the analysis of the validity of the result:

```
        MSB
        0 ----------- X
        1 ----------- Y
------------
0       0 ----------- Z (e.g. P+N=P; this combo is impossible for US; for Signed, its correct results as is)
0       1 ----------- Z (e.g. P+N=N; this results is correct as is for both US and Signed)

1       0 ----------- Z      (A)
CF      SF
```

~~~~~~

```
        0
        0
------------
0       1       B1*   invalid result; correction needed
0       0       C1
```

~~~~~~

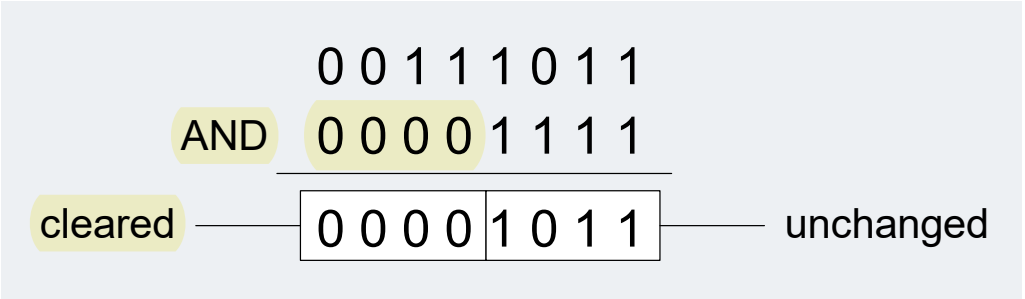
```
        1
        1
------------
1       0       B2*   invalid result; correction needed
1       1       C2
```

AND Instruction

- Performs a Boolean AND operation between each pair of matching bits in two operands --> bit-wise &
- always clears the Overflow and Carry flags
- Syntax:

```
AND destination, source
```

  - same operand types as MOV
- *Masking ==> "0 & op"*



AND

x	y	x ∧ y
0	0	0
0	1	0
1	0	0
1	1	1

OR Instruction

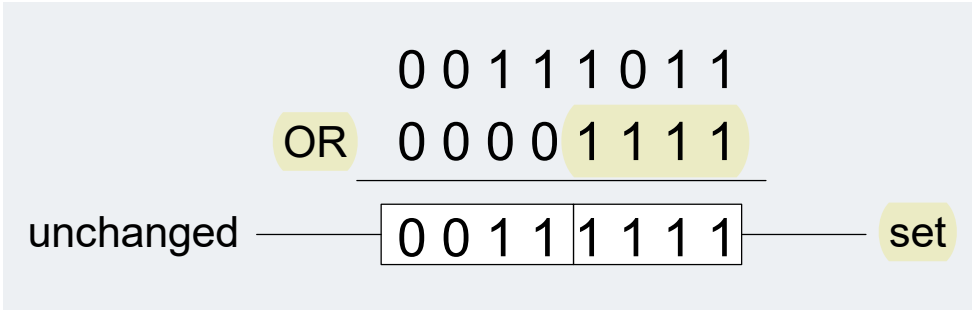
- Performs a Boolean OR operation between each pair of matching bits in two operands-->bit-wiseOR
- always clears the Carry and Overflow flags
- Syntax:

```
OR destination, source
```

- useful for setting 1 or more bits in an operand affecting any other bits

OR

x	y	x ∨ y
0	0	0
0	1	1
1	0	1
1	1	1

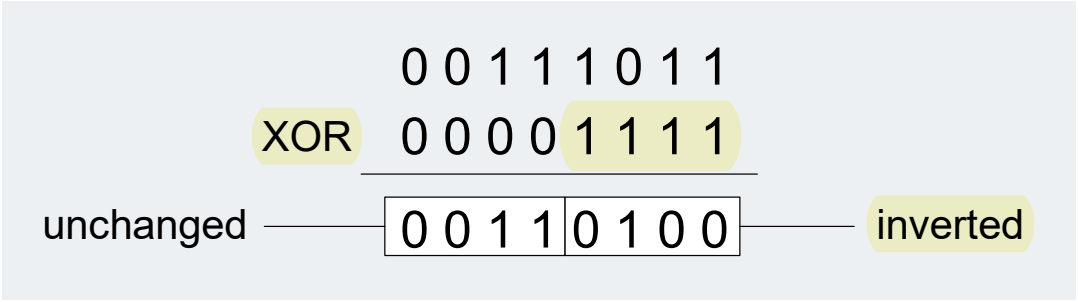


XOR Instruction

- Performs a Boolean exclusive-OR operation between each pair of matching bits in two operands
  - always clears the Overflow and Carry flags.
- Syntax:

```
XOR destination, source
```

- useful way to toggle (invert by "1 XOR op") the bits in an operand



XOR

x	y	x ⊕ y
0	0	0
0	1	1
1	0	1
1	1	0

NOT Instruction

- Performs a Boolean NOT operation on a single destination operand
- Syntax:

```
NOT destination
```

```
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

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
65	41	101	A A	97	61	141	a a
66	42	102	B B	98	62	142	b b
67	43	103	C C	99	63	143	c c

ASCII chart: case-difference=32

Applications

- 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
```

- 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

48	30	060	0	0
49	31	061	1	1
50	32	062	2	2
51	33	063	3	3
52	34	064	4	4
53	35	065	5	5
54	36	066	6	6
55	37	067	7	7
56	38	070	8	8
57	39	071	9	9

Jumps Based On . . .

CONDITIONAL JUMPS

JMP Instruction

- JMP is an unconditional jump to a label that is usually within the same procedure.
  - Syntax: JMP *target*
- Logic: EIP ← *target*
- Example:

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

Conditional jumps

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

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

Applications

- 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.

CF and OF are always cleared!
ZF, SF, PF are set accordingly.

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

ORing any number with itself does not change its value.

Applications

- 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.

CF and OF are always cleared!
ZF, SF, PF are set accordingly.

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

Your turn: Write code that jumps to a label if an integer is negative.

TEST Instruction

Z = X & Y
// stores results in Z and affects EFLAGS

test X, Y
// dont forget, one operand must be a register
// Doesn't store result, and affects EFLAGS ONLY

- Performs a nondestructive AND operation between each pair of matching bits in two operands
  - No operands are modified, but the Zero flag is affected.
- Use a *bit mask*

```
test al,00001001b
jz  ValueNotFound

    0 0 1 0 0 1 0 1  <- input value (AL)
    0 0 0 0 1 0 0 1  <- test value
    0 0 0 0 0 0 0 1  <- result: ZF = 0

    0 0 1 0 0 1 0 0  <- input value (AL)
    0 0 0 0 1 0 0 1  <- test value
    0 0 0 0 0 0 0 0  <- result: ZF = 1
```

```
test al,00001001b
jnz   ValueFound
```

Applications

- Jump to label L2 if contents of eAx is even

```
test eAx,1
jz   L2
```

- Jump to label L1 if contents of eAx equals Zero

```
Test eAx,11111111x
jz  L1
```

CMP Instruction

EFLAGS <---- D - S

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

CMP Results	ZF	CF
Destination < source	0	1
Destination > source	0	0
Destination = source	1	0

borrow!

When two unsigned operands are compared

CMP Results	Flags
Destination < source	SF ≠ OF
Destination > source	SF = OF
Destination = source	ZF = 1

When two signed operands are compared

x, y sign same but z sign different ==> invalid z; detect OF

+X +Y (CF=0, SF=1) ==> Z=result + 2^8

-X -Y (CF=1, SF=0) ==> Z=result - 2^8

Jumps Based on Equality

un-/signed

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

Jumps Based on Signed Comparisons

greater/lesser

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)

Jumps Based on Unsigned Comparisons

above/below

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)

Applications

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

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

- Task: Jump to a label if signed EAX is less than EBX
- Solution: Use CMP, followed by JL

```
cmp  eax,ebx
jl  Lesser
```

Applications

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

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

- Jump to label L1 if signed EAX is equal to Val1

```
cmp eax,Val1
je L1
```

CONDITIONAL STRUCTURES

- Block-Structured IF Statements
- Compound Expressions with AND /OR
- WHILE Loops
- LOOP instruction

Block-Structured IF Statements

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

un-/signed !

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

```
    mov eax,op1
    cmp eax,op2
    jne EndIF
    mov X,1
EndIf:
```

Applications

- 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:
```

Your turn . . .

- 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:
```

(There are multiple correct solutions to this problem.)

Block-Structured IF Statements

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

un-/singed

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

```
    mov eax,op1
    cmp eax,op2
    jne Else
    mov X,1
    jmp EndIf
Else:
    mov X,2
EndIf :
```

Your turn . . .

- 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:
```

(There are multiple correct solutions to this problem.)

Compound Expression with AND

- When implementing the logical AND operator, consider using 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;
```

Example

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

let, 8-bit registers are pre-loaded with unsigned integers

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:
```

Another !

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

```
    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:
```

seven instructions before vs. five here

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

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

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

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

```
    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 back to the top of the loop.
- Consider the following example:

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

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

Your turn . . .

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

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

```
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:
```

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.

Examples . . .

- *What will be the final value of AX?*

ax	ecx
6	4
7	3
8	2
9	1
10	0

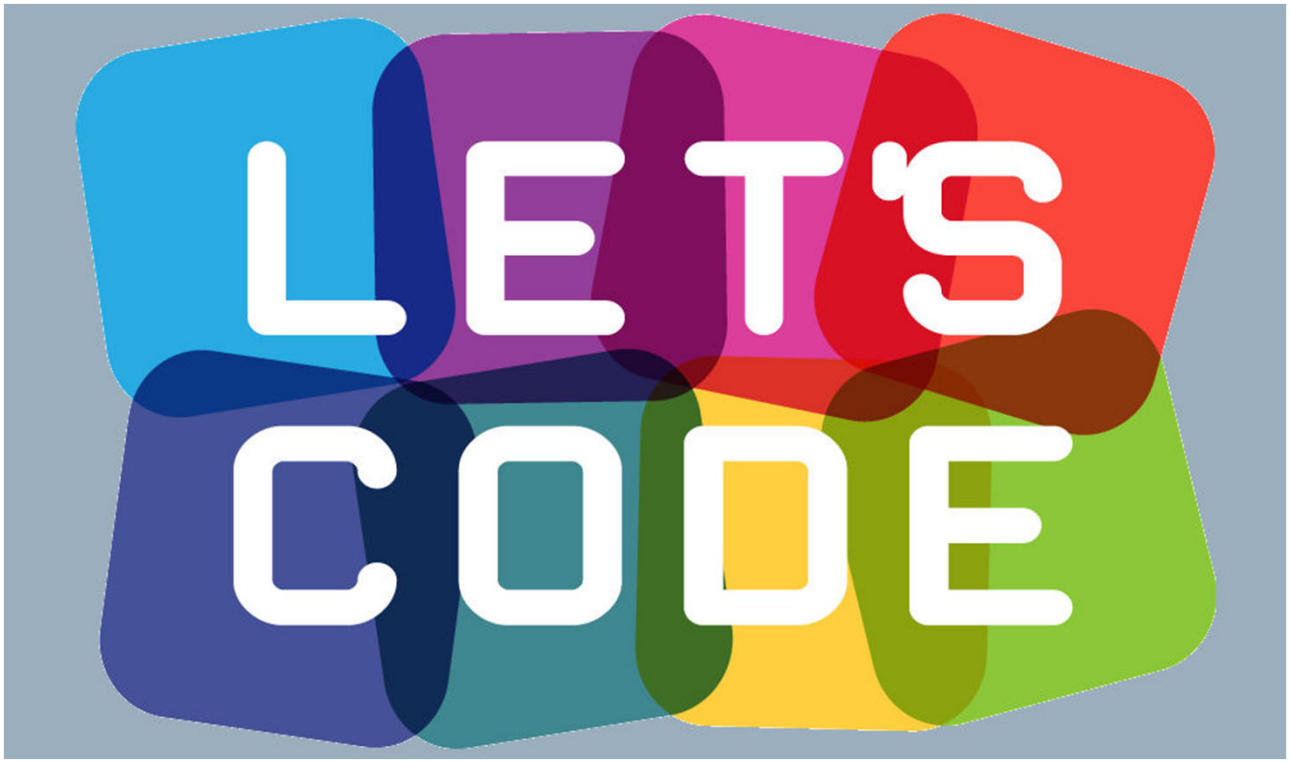
```
    mov ax,6
    mov ecx,4
L1:
    inc ax
    loop L1
```

- *How many times will the loop execute?*

2^{32} times looping

```
    mov ecx,0
X2:
    inc ax
    loop X2
```

Write a program that sums the first 10 natural numbers



Write a program that sums the first 10 natural numbers

Solution

```
.data

.code
main PROC
    mov Ax, 0          ; zero the accumulator
    mov cx,10          ; loop counter

    L1:
       add ax,cx       ; add an integer
       LOOP L1         ; repeat until ECX = 0

    INVOKE ExitProcess, 0
main ENDP
END main
```

ax	cx
0	10
10	9
19	8
27	7
.	.
.	.
.	.
54	1
55	0

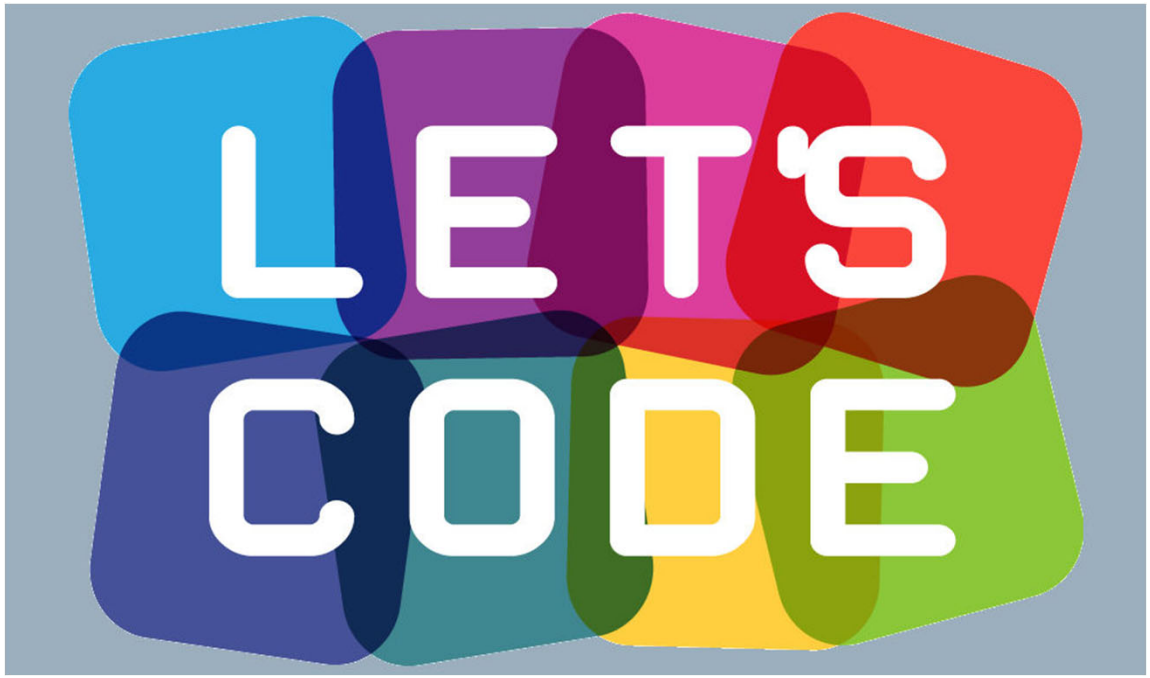
Nested Loops

- If you need to code a loop within a loop, you must save the outer loop counter's ECX value.

```
.data
count DWORD ?
.code
    mov ecx,10              ; 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
```

Write a program that sums the first 10 natural numbers 5 times.

Use a nested loop!



Write a program that sums the first 10 natural numbers 5 times.

Solution

```
.data
; (insert variables here)
count DWORD 5

.code
main PROC
     mov Ax, 0        ; zero the accumulator; holds total result
     mov cx,count     ; outer loop counter
     L1:

       mov count, ecx      ; save the current iteration
       mov cx, 10          ; inner loop counter
       mov bx, 0         ; holds partial result
     L2:
       add bx,cx           ; add an integer
       LOOP L2             ; repeat until ECX = 0
       add Ax, bx          ;update the accumulator/total result
       mov ecx, count
       LOOP L1

INVOKE ExitProcess, 0

main ENDP
```

Summary

- Bitwise instructions (AND, OR, XOR, NOT, TEST)
  - manipulate individual bits in operands; update result/EFLAGS
- CMP – compares operands using implied subtraction
  - sets condition flags only
- Conditional Jumps
  - equality: JE, JNE
  - flag values: JC, JZ, JNC, JP, ...
  - signed: JG, JL, JNG, ...
  - unsigned: JA, JB, JNA, ...
- LOOP – branching instructions