X86 Assembly Language Programming
Part 3

Jumping, Looping, and Procedure

Charles Kim
Howard University

www.mwftr.com

x86 Instruction Set

Subject 4: x86 Assembly Programming (Link to the 80386 Instruction Set and Intel386 Programmer's Reference Manual)

Getting Started with MASM and Visual Studio:

- Visual Studio 2010: Instruction (Read this first) + Link Library (.msi file. Download and save this file in your computer)
- Visual Studio 2012: Instruction (Read this first) + Link Library (.msi file. Download and save this file in your computer)
- Visual Studio 2013: Instruction (Read this first) + Link Library (.msi file. Download and save this file in your computer)

* Note4a --- Assembly language part a

* Note4b --- Part b (Data Transfer)

X86 Assembly Language Programming Part 3

(1) Jumping and Looping

Charles Kim
Howard University

www.mwftr.com

Unconditional Jump: **JMP**

- **jmp**
  - Like a goto in a high-level language
  - Format: **jmp** StatementLabel
  - The next statement executed will be the one at StatementLabel:
    - Note that the label has a **colon (:)** at the end

Pseudo-Code for an infinite loop

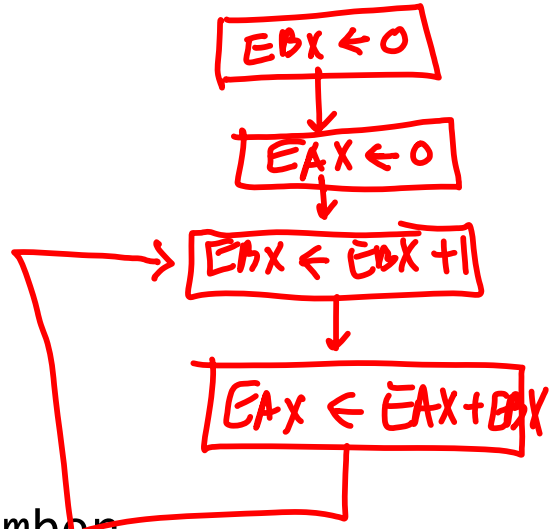
- Program Design (pseudo-code) using **jmp** for 1+2+3+… forever

```
     number := 0;
     sum := 0;
;forever loop
loop1
     add 1 to number;
     add number to sum;
    goto loop1
     end loop;
```

InfLoop.asm

```
; program to find sum 1+2+...+n for n=1, 2, ...
INCLUDE Irvine32.inc
.CODE
main      PROC
          mov     ebx,0   ; number := 0
          mov     eax,0   ; sum := 0

forever:
           call    DumpRegs
           inc     ebx          ; add 1 to number
          add     eax, ebx     ; add number to sum
          jmp     forever      ; repeat
           exit
main      ENDP
END main
```



Conditional Jumps

- Format: `j--  targetStatement`
  - The last part **(--)** of the mnemonic identifies the condition under which the jump is to be executed
  - If the condition holds, then the jump takes place and the statement executed is at `targetStatement:`
  - Otherwise, the next instruction (the one following the conditional jump) is executed
  - Most "conditions" considered by the conditional jump instructions are settings of flags in the flags register.
- Example

```
jz step2
```

  - jump to the statement with label step2 if the zero flag ZF is set to 1
  - Most common way of setting/clearing flags --- by `jmp` instruction

`cmp` Instruction

- **`cmp`**: Most common way to set flags for conditional jumps
- Format: **`cmp dst, src`**
- Flags are set the same as for the subtraction operation {**`dst – src`**}
- Operands (both dst and src) are not changed
- **`cmp`** sets or clears flags
  - CF: Carry flag (when there is borrow (‘No Carry” in subtraction)
  - OF: Overflow (Overflow)
    - OF={ **Carry out** from msb} XOR {**Carry in** to msb}
  - SF: msb (Sign bit) is 1
  - ZF: Result is zero
  - **For programmers, “no worry” on Flags; but worry on the actual numerical condition**

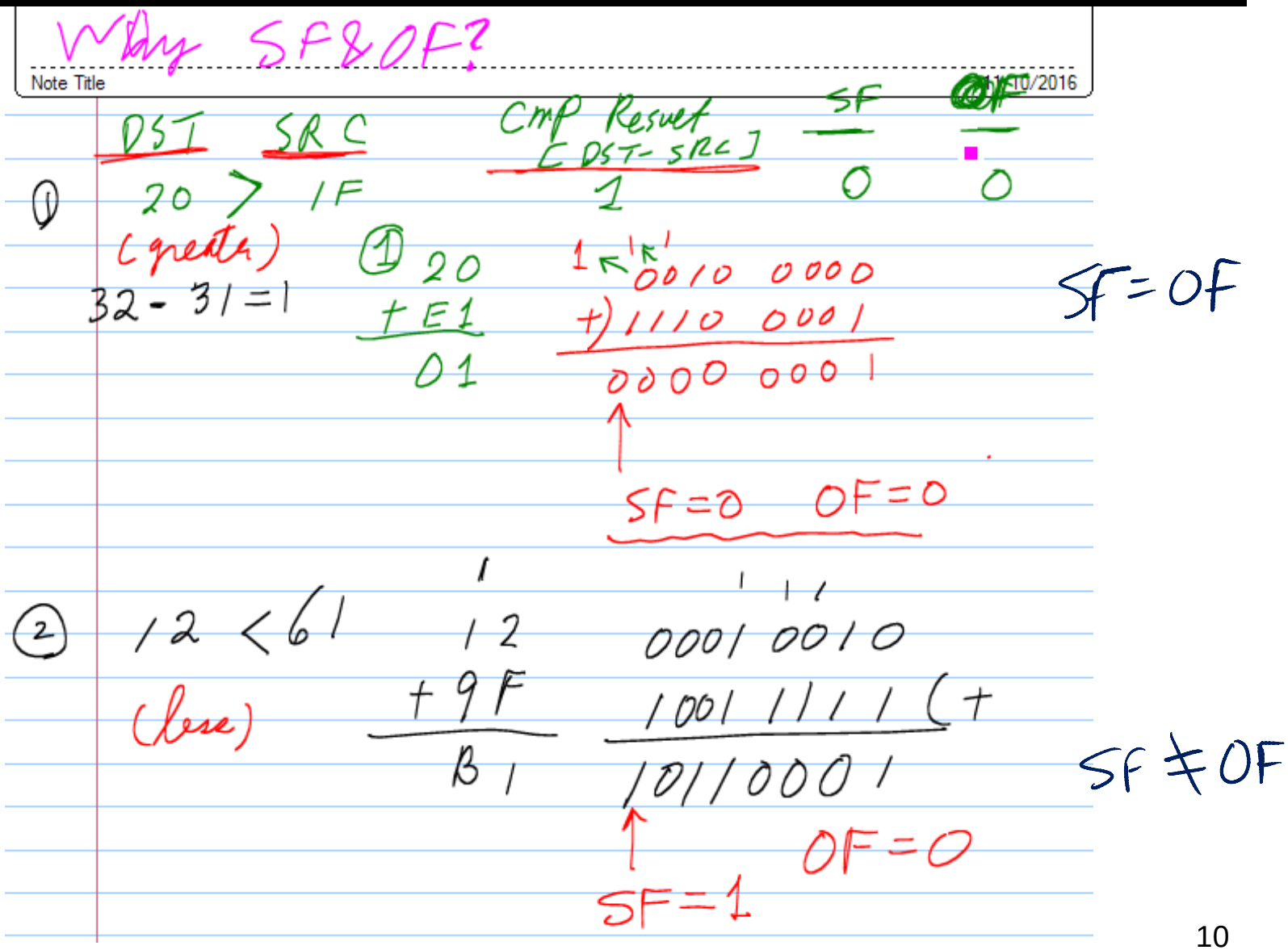
Conditional Jumps To Use After ***Signed*** Operand Comparison (`cmp`)

mnemonic		**jumps if**	
jg	jump if greater	**dst > src**	SF=OF and ZF=0
jnle	jump if not less or equal		
jge	jump if greater or equal	**dst >= src**	SF=OF
jnl	jump if not less		
jl	jump if less	**dst < src**	SF=/=OF
jnge	jump if not above or equal		
jle	jump if less or equal	**dst =< src**	SF=/=OF or ZF=1
jng	jump if not greater		

```
cmp   eax, ebx    ;[eax]-[ebx]
      jle   smaller
```

•The jump will occur if the value in **eax** is less than or equal to the value in **ebx**, where both are interpreted as **signed numbers**

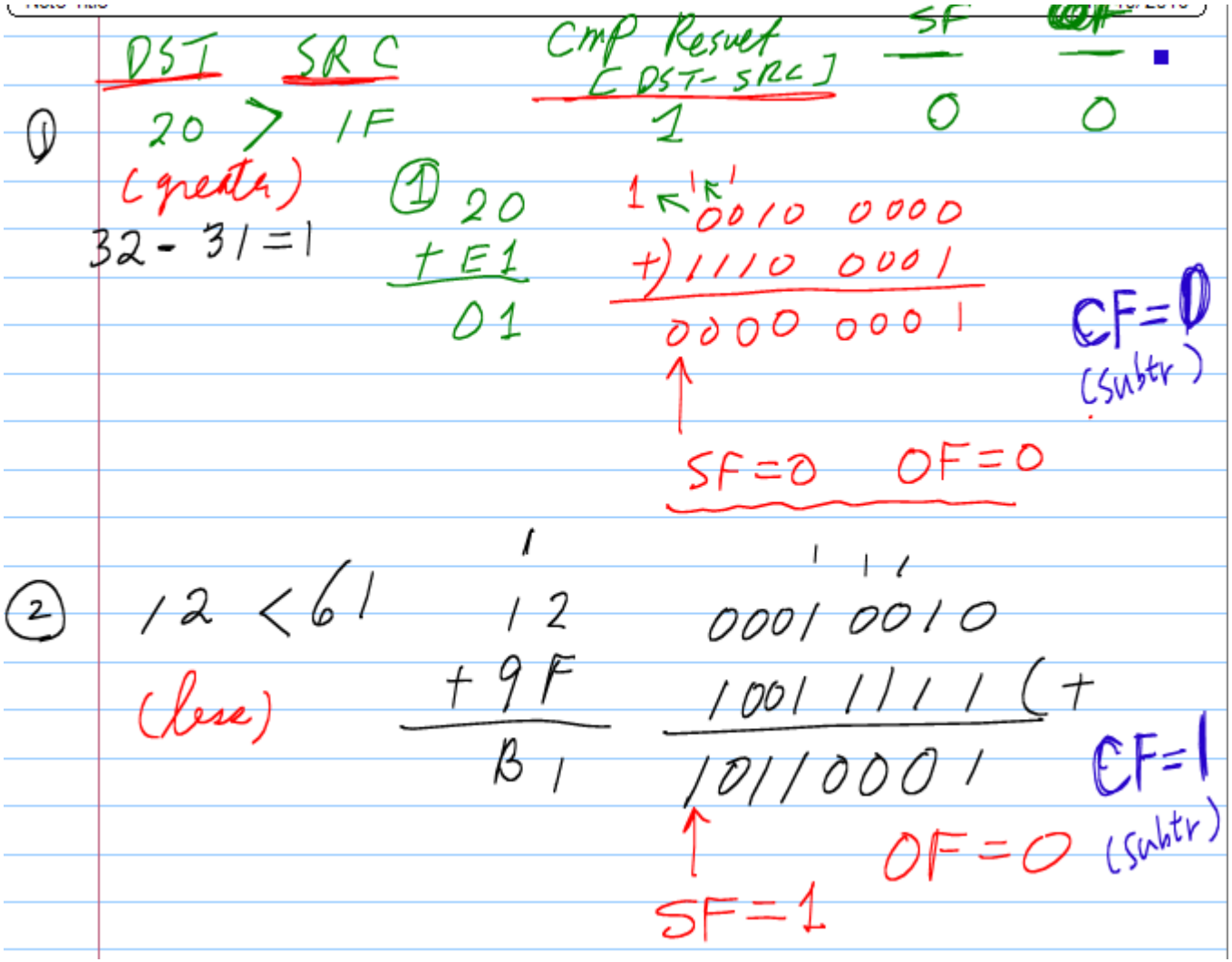
Signed Operation: SF and OF



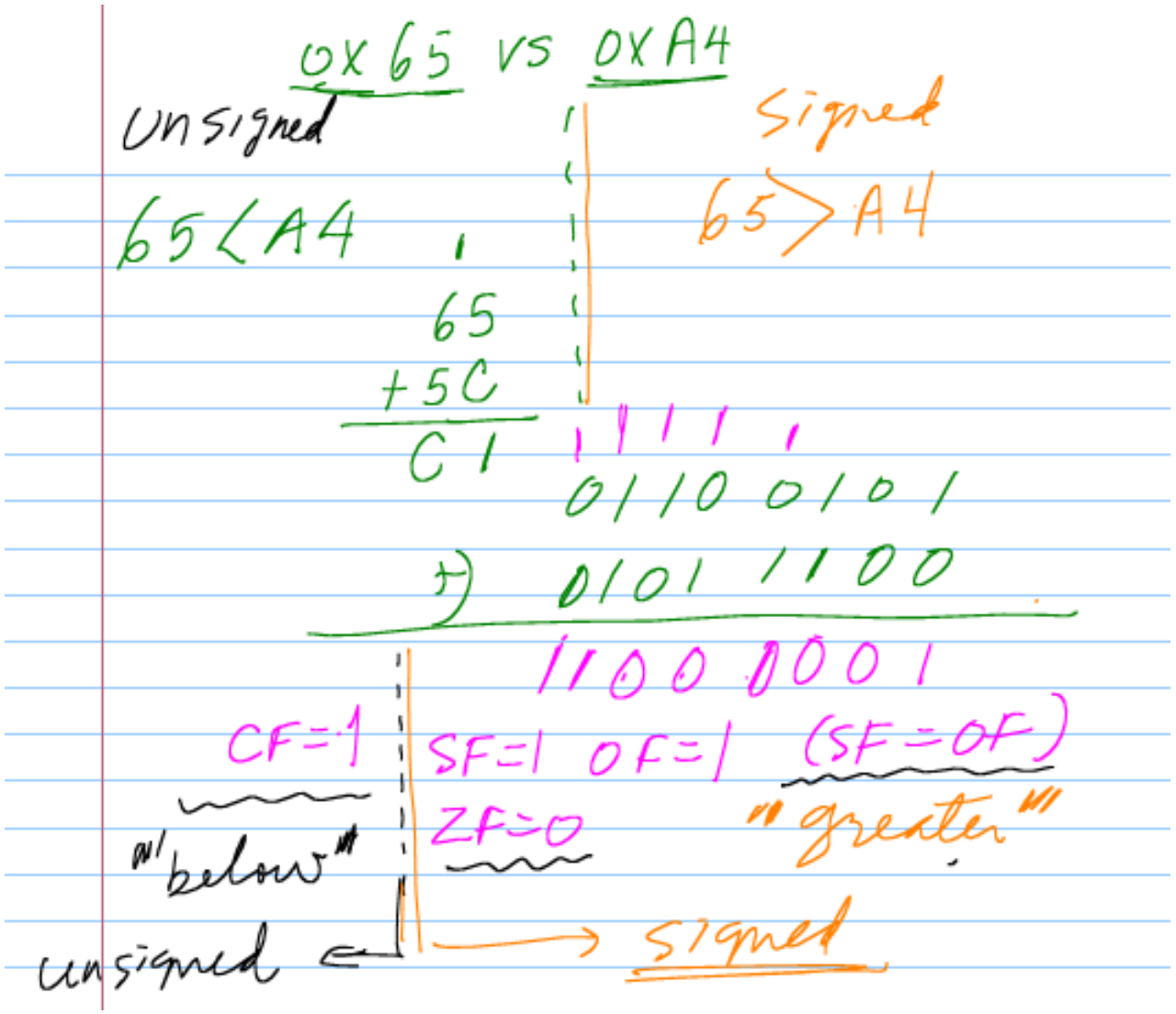
Conditional Jumps To Use After *Unsigned* Operand Comparison

mnemonic		jumps if	
ja	jump if above	**dst> src**	CF=0 and ZF=0
jnbe	jump if not below or equal		
jae	jump if above or equal	**dst >= src**	CF=0
jnb	jump if not below		
jb	jump if below	**dst<src**	CF=1
jnae	jump if not above or equal		
jbe	jump if below or equal	**dst =< src**	CF=1 or ZF=1
jna	jump if not above		

Unsigned Operand Comparison -- CF



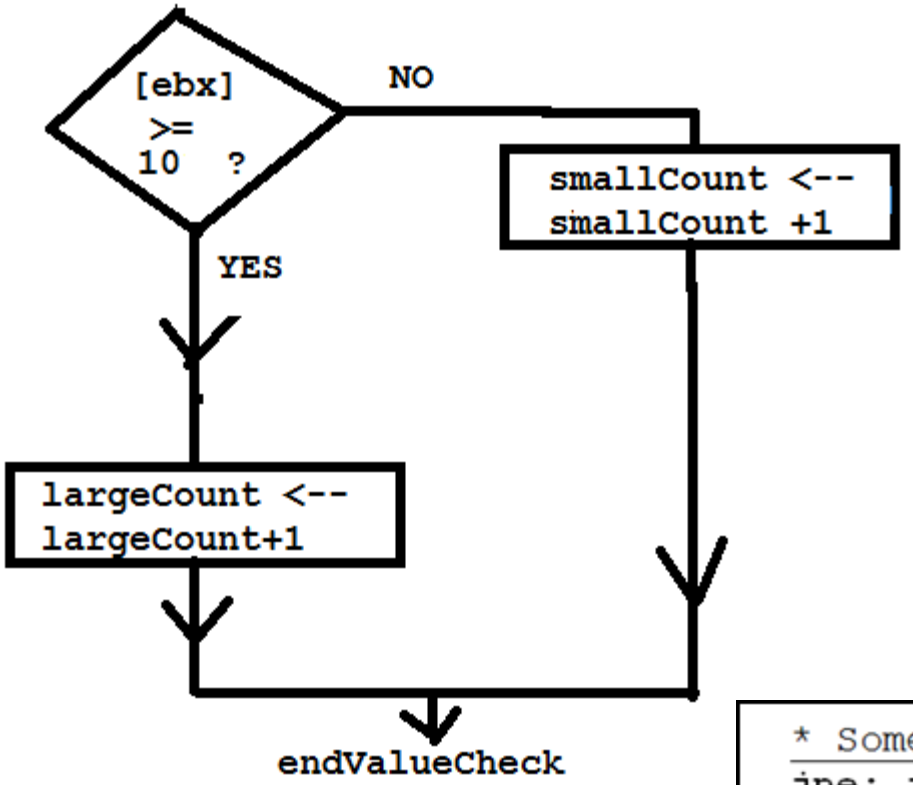
Signed (SF = OF?) **Unsigned** (CF= 1 or 0) ?



Some Other Conditional Jumps

mnemonic		jumps if	
je	jump if equal	**dst = src**	ZF=1
jz	jump if zero	**dst = 0**	
jne	jump if not equal	**dst =\= src**	ZF=0
jnz	jump if not zero	**dst =\= 0**	
js	jump if sign (negative)	SF=1	
jc	jump if carry	CF=1	
jo	jump if overflow	OF=1	

if Example 1

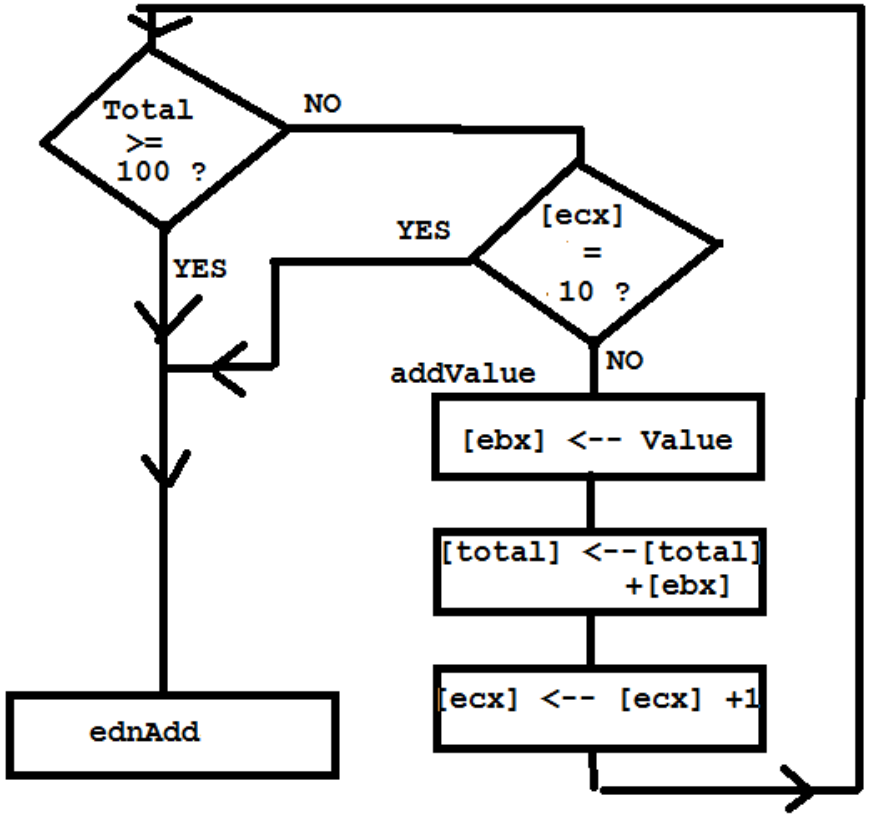


- **Assumptions**
  - **value in EBX**
  - **smallCount and largeCount in memory**

```
;Code
                cmp   ebx, 10
                jnl   elseLarge
                inc   smallCount
                jmp   endCheck
elseLarge:      inc   largeCount
endCheck:
                exit
```

```
* Some conditional jump instructions
jne: jump if not equal
jng: jump if not greater (signed)
jle: jump if less or equal (signed)
jnge: jump if not above or equal (signed)
jnae: jump if not above or equal (unsigned)
jnbe: jump if not below or equal (unsigned)
```

if Example 2

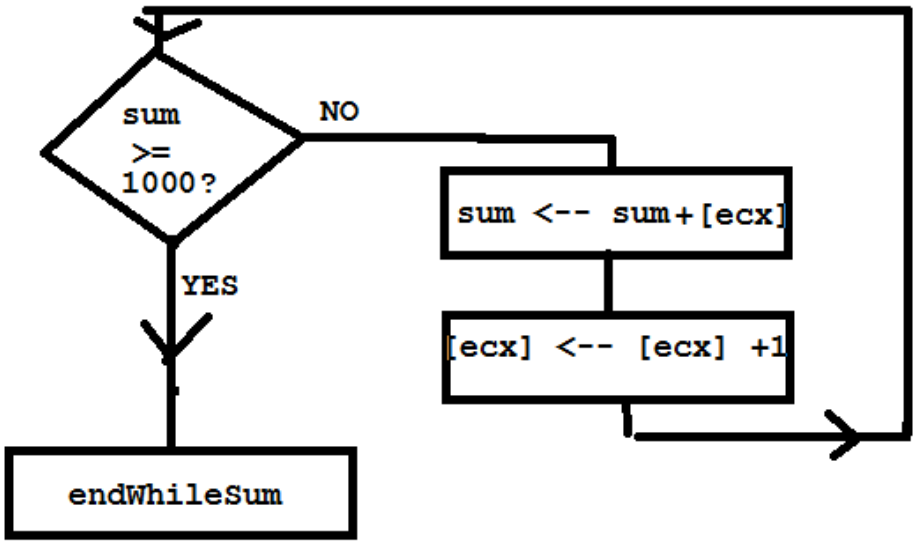



```
* Some conditional jump instructions
jne: jump if not equal
jng: jump if not greater (signed)
jle: jump if less or equal (signed)
jnge: jump if not above or equal (signed)
jnae: jump if not above or equal (unsigned)
jnbe: jump if not below or equal (unsigned)
```

- Assumptions
  - total and value in memory
  - count in ECX

```
;Code
CheckValue: cmp  total, 100
            jge  endAdd
            cmp  ecx, 10
            je   endAdd
addValue:   mov  ebx, value
            add  total, ebx
            inc  ecx
            jmp  checkValue
endAdd:     exit
```

While Example



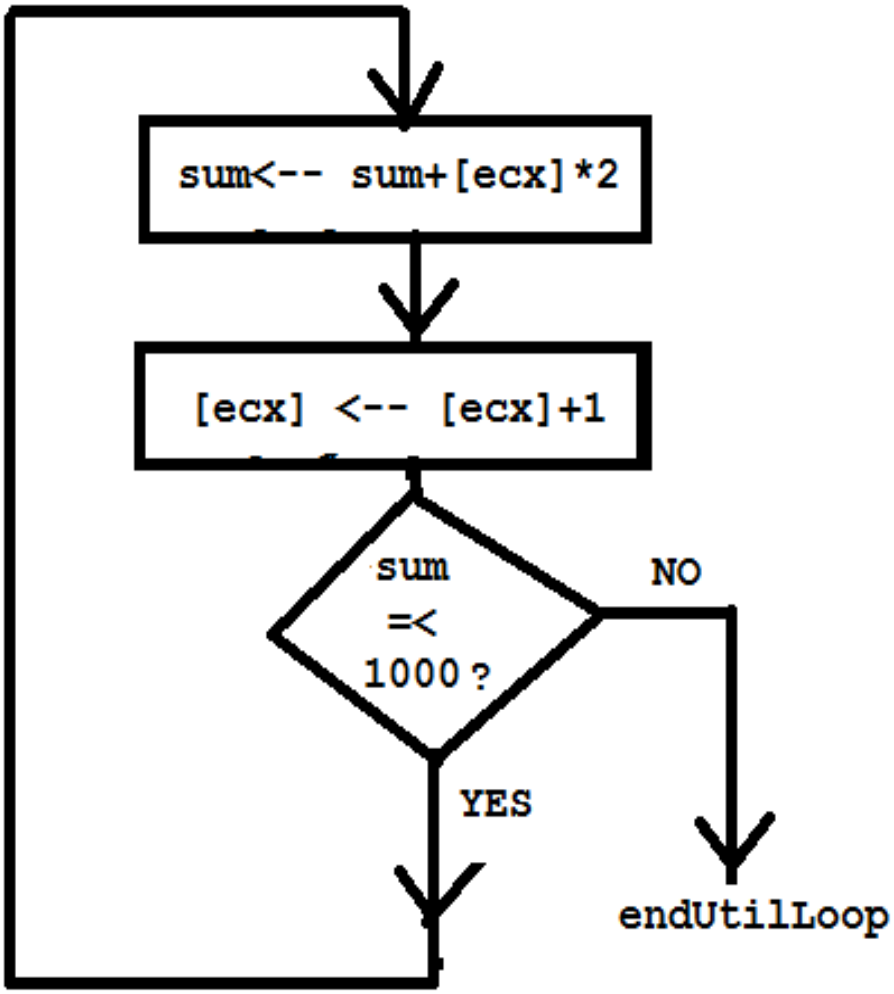
•Assumptions
- sum in memory
- count in ECX

Code

```
whileSum:     cmp   sum, 1000
              jnl   endWhileSum
              add   sum, ecx
              inc   ecx
              jmp   whileSum
endWhileSum:  exit
```

```
* Some conditional jump instructions
jne: jump if not equal
jng: jump if not greater (signed)
jle: jump if less or equal (signed)
jnge: jump if not above or equal (signed)
jnae: jump if not above or equal (unsigned)
jnbe: jump if not below or equal (unsigned)
```

Until Example




```
;Code
repeatLoop:   add    sum, ecx
              add    sum, ecx
              inc    ecx
              cmp    sum, 1000
              jng    repeatLoop
endUntilLoop:exit
```

- Assumptions
  - sum in memory
  - count in ECX

What is this code for? (ecx contains the solution)

```
INCLUDE Irvine32.inc

.DATA
number  DWORD   750

.CODE
main        PROC
            mov   ecx, 0      ; x := 0
            mov   eax, 1      ; twoToX := 1
whileLE:    cmp   eax, number ; twoToX <= number?
            jnle  endWhileLE  ; exit if not
body:       add   eax, eax    ; multiply twoToX by 2
            inc   ecx         ; add 1 to x
            jmp   whileLE     ; go check condition again
endWhileLE:
            dec   ecx         ; subtract 1 from x

            mov   eax, 0      ; exit with return code 0
            exit
main        ENDP
END main
```

Base2.asm ---- $Log_2(X)$ calc

Base2.asm

```
INCLUDE Irvine32.inc

.data
number  DWORD   750

.code
main        PROC
            mov   ECX, 0        ; x 0
            mov   EAX, 1        ; twoToX 1
whileLE:
            cmp   EAX, number ; twoToX <= number?
            jnle  endWhileLE  ; exit if not
body:
            add   EAX, EAX    ; multiply twoToX by 2
            inc   ECX         ; add 1 to x
            jmp   whileLE     ; go check condition again
endWhileLE:
            dec   ECX         ; subtract 1 from x

            exit

main        ENDP
```

```
>>> import math
>>> math.log(750,2)
9.550746785383243
>>> math.log(1024,2)
10.0
>>>
```

100 %

Memory 1

Address: 0x00404000

```
0x00404000  ee 02 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0x0040401D  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0x0040403A  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0x00404057  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0x00404074  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0x00404091  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0x004040AE  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
```

Registers

```
EAX = 00000200 EBX = 7EFDE000
ECX = 00000009 EDX = 00401005
ESI = 00000000 EDI = 00000000
EIP = 00401020 ESP = 0018FF8C
EBP = 0018FF94 EFL = 00000297
```

What is this code for?

```
INCLUDE Irvine32.inc
.DATA
BCENTS   DWORD   1000000000       ; 1 hundred billion cents = $ 1M

.CODE
main         PROC
             mov   ebx, 1       ;
             mov   eax, 0       ;
             mov   ecx, 0       ; day := 0
whilePoor:   cmp   eax, 100000000  ; total < 100,000,000?
             jnl   endLoop      ; exit if not
body:        add   eax, ebx     ; add
             add   ebx, ebx     ; multiply by 2
             inc   ecx          ; add 1 to day
             jmp   whilePoor    ; repeat
endLoop:
             mov   eax, 0       ; exit with return code 0
             exit
main         ENDP
END main
```

Million.asm

Million.asm

```
; program to determine how many days it takes to earn $1,000,000
; starting with 1 cent on day 1, 2 cents on day 2, 4 cents on day 3, etc.
; Author:  R. Detmer
; Date:    6/2008

INCLUDE Irvine32.inc
.DATA
BCENTS    DWORD     1000000000
.CODE
main          PROC
              mov   EBX, 1       ; nextDaysWage := 1
              mov   EAX, 0       ; totalEarnings := 0
              mov   ECX, 0       ; day := 0
whilePoor:    cmp   EAX, Bcents   ; totalEarnings < 100,000,000?
              jnl   endLoop      ; exit if not
body:         add   EAX, EBX     ; add nextDaysWage to totalEarnings
              add   EBX, EBX     ; multiply nextDaysWage by 2
              inc   ECX          ; add 1 to day
              jmp   whilePoor    ; repeat
endLoop:
              mov   EAX, 0       ; exit with return code 0
              exit
```

```
>>> hex(1000000000)
'0x3b9aca00'
```

100 %

Memory 1

Address: 0x00404000

```
0x00404000  00 ca 9a 3b 00 00 00 00 00 00 00 00 00 00 00 0
0x0040401D  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0x0040403A  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0x00404057  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
```

Registers

```
EAX = 00007FFF EBX = 00004000
ECX = 0000000E EDX = 00401005
ESI = 00000000 EDI = 00000000
EIP = 00401029 ESP = 0018FF8C
EBP = 0018FF94 EFL = 00000206
```

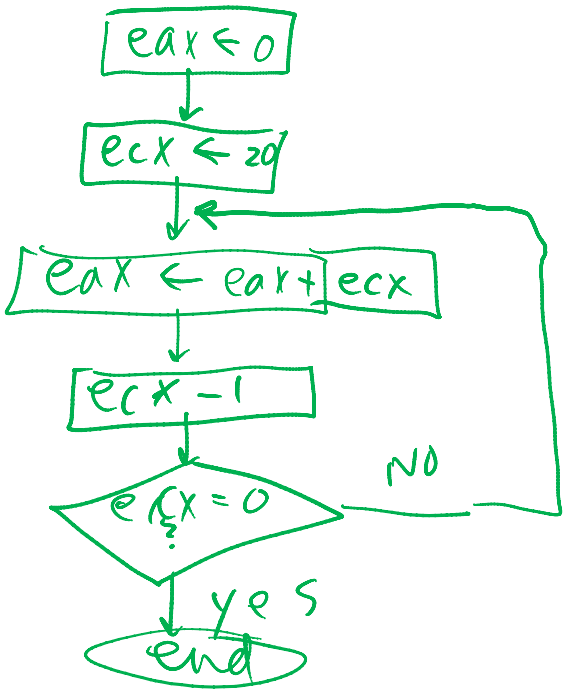
`loop` instruction

- format: **`loop`** ***`statementLabel`***
  - Execution
    - **ECX ← ECX – 1 (ECX is decremented by 1)**
      - **If [ECX] =0, Go to next line**
      - **otherwise, jump** to the *StatementLabel*

example of loop

Design

sum := 0
for *count* := 20 downto 1 loop
    add *count* to *sum*;
end for;




```
;Code
                mov     eax, 0
                mov     ecx, 20
forCount:       add     eax, ecx
                loop    forCount
```

- Assumptions
  - sum in EAX
  - count in ECX

LoopEx.asm

LoopEx.asm

```
INCLUDE Irvine32.inc

.DATA
number    DWORD     20
.CODE
main          PROC
              mov   EAX, 0
              mov   ECX, number
forCount:     add   EAX, ECX
              loop  forCount

              exit
main          ENDP
END main
```

100 %

Memory 1

Address: 0x00404000

```
0x00404000  14 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0040401D  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0040403A  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00404057  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00404074  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Registers

```
EAX = 00000084 EBX = 7EFDE000
ECX = 0000000C EDX = 00401005
ESI = 00000000 EDI = 00000000
EIP = 0040101B ESP = 0018FF8C
EBP = 0018FF94 EFL = 00000216
```

Class Activity --- Coding

- Design and implement a program that will compute the sum $1^2+2^2+3^2+ .... + n^2$.
  - The value for the number *n* (2 or 3, for example) is to be read from keyboard

```
        mov  EDX,OFFSET prompt
        call WriteString

        call ReadDec
;Others -- ReadHex (Hex number), ReadInt (signed number)
        mov  x,EAX
```

  - Short report: Code, Debugging steps, results with different values of n, with screen captures.
  - Submission (of an ASM file via email): By End of the Class or
  - File naming convention: GroupN.asm