X86 Assembly Language Programming Part 4

Jumping, Looping, and Array

EECE416 uC

Charles Kim
Howard University

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

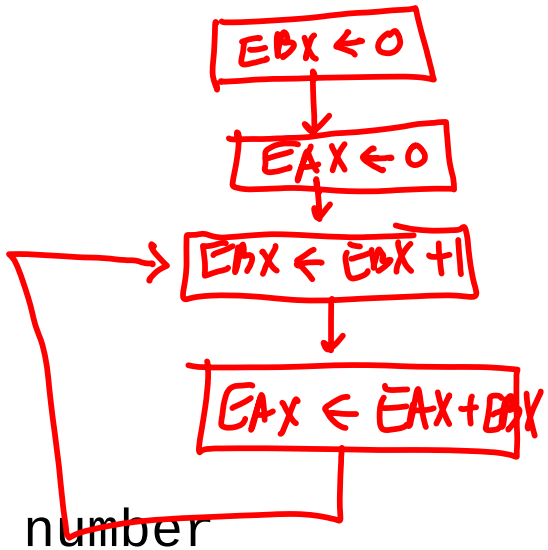
Pseudo-Code for an infinite loop

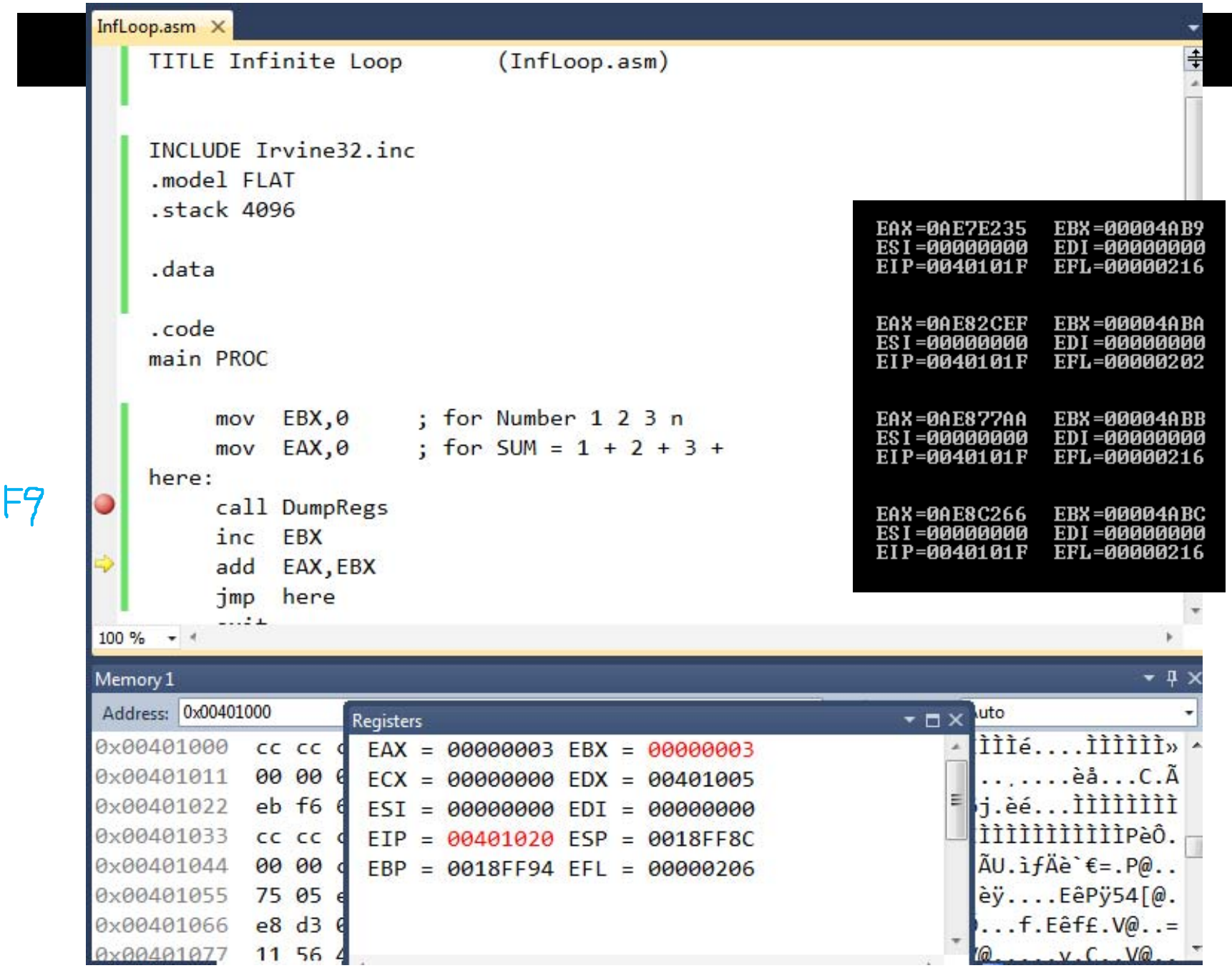
- Program Design (pseudo-code) using **`jmp`** for 1+2+3+… forever

```
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
```



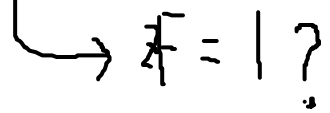

InfLoop.asm
TITLE Infinite Loop (InfLoop.asm)
INCLUDE Irvine32.inc
.model FLAT
.stack 4096
.data
.code
main PROC
mov EBX,0 ; for Number 1 2 3 n
mov EAX,0 ; for SUM = 1 + 2 + 3 +
here:
call DumpRegs
inc EBX
add EAX,EBX
jmp here
EAX=0AE7E235 EBX=00004AB9
ESI=00000000 EDI=00000000
EIP=0040101F EFL=00000216
EAX=0AE82CEF EBX=00004ABA
ESI=00000000 EDI=00000000
EIP=0040101F EFL=00000202
EAX=0AE877AA EBX=00004ABB
ESI=00000000 EDI=00000000
EIP=0040101F EFL=00000216
EAX=0AE8C266 EBX=00004ABC
ESI=00000000 EDI=00000000
EIP=0040101F EFL=00000216
100 %
Memory 1
Address: 0x00401000
Registers
EAX = 00000003 EBX = 00000003
ECX = 00000000 EDX = 00401005
ESI = 00000000 EDI = 00000000
EIP = 00401020 ESP = 0018FF8C
EBP = 0018FF94 EFL = 00000206

F9

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



`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

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, nbr    ;[eax]-nbr
      jle  smaller
```

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

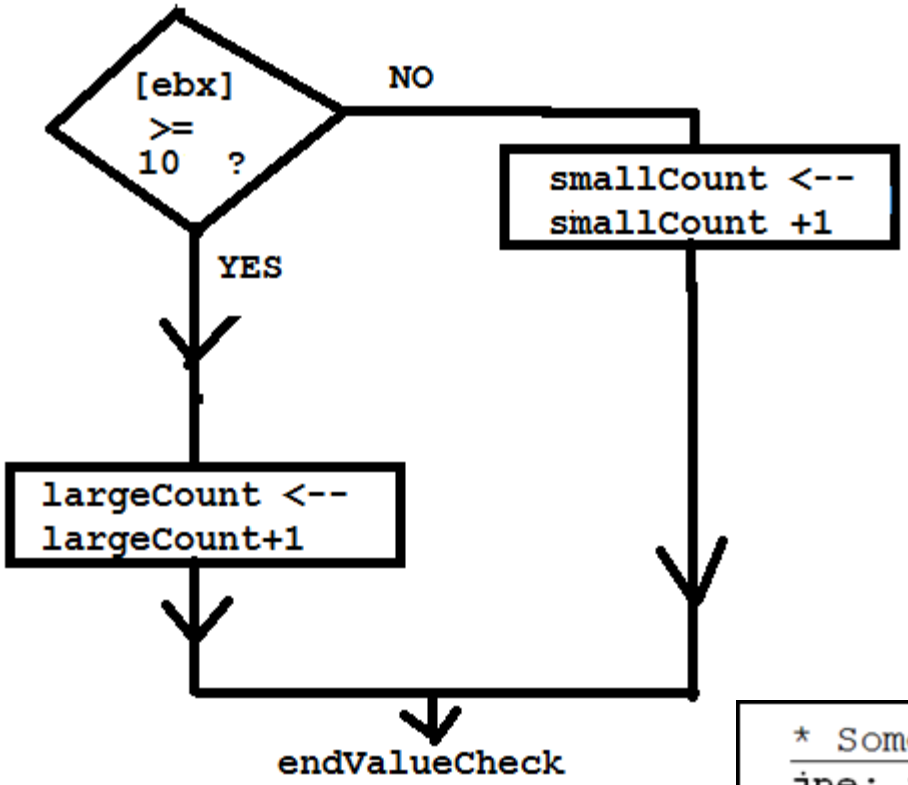
Conditional Jumps To Use After *Unsigned* Operand Comparison

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		

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

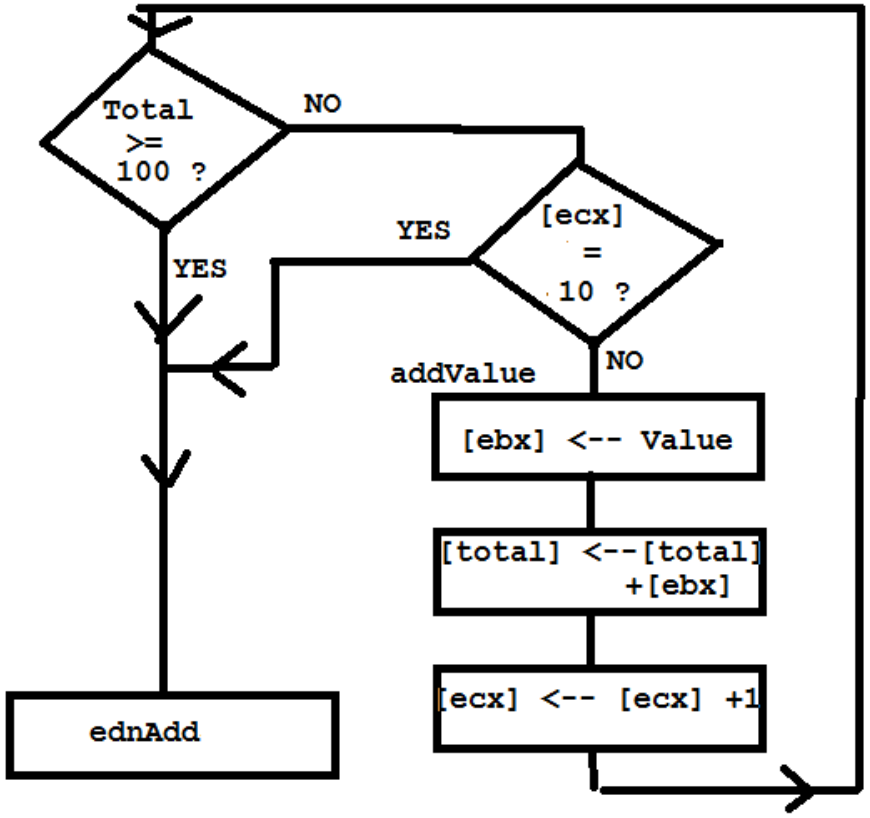


- **Assumptions**
  - **value in EBX**
  - **smallCount and largeCount in memory**

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

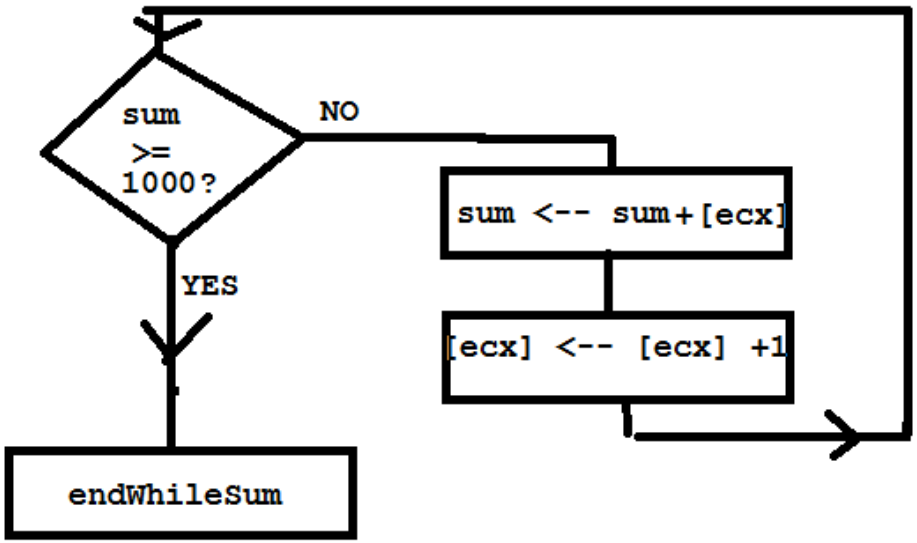



```
* Some conditional jump instructions
jne: jump if not equal
jng: jump if not greater (signed)
jle: jump if less or equal (signed)
jnge: jump if not above or equal (signed)
jnae: jump if not above or equal (unsigned)
jnbe: jump if not below or equal (unsigned)
```

- 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



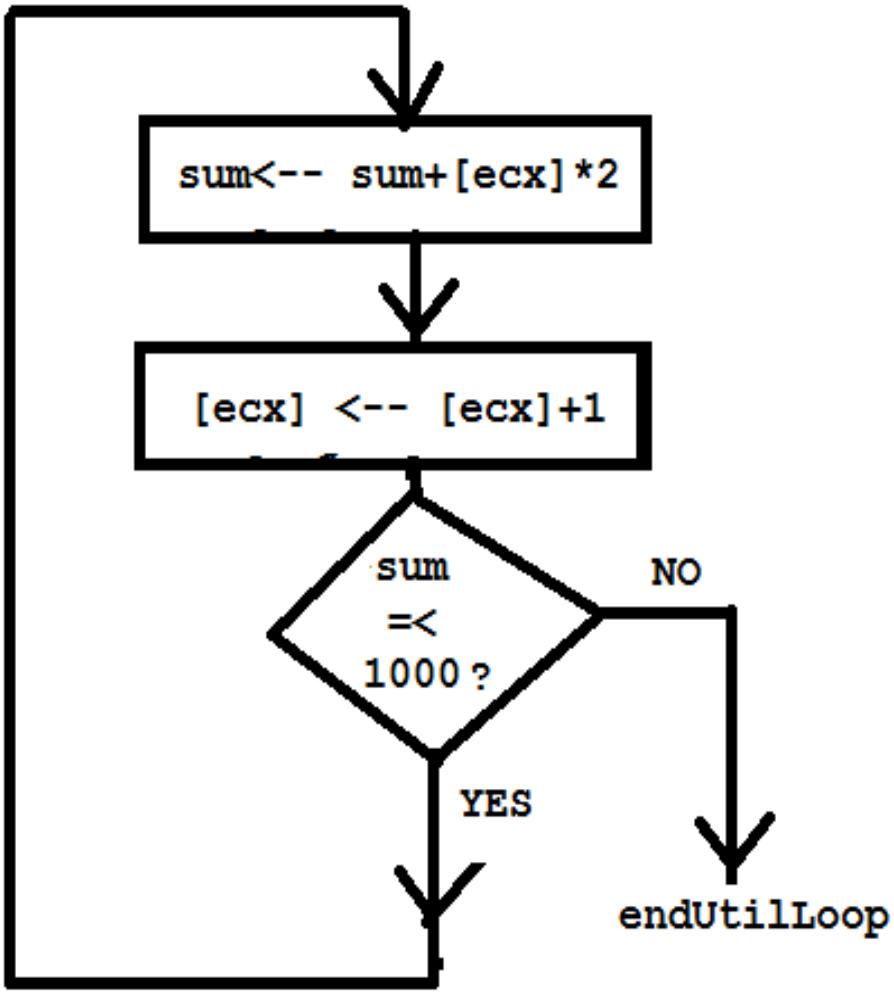
•Assumptions
- sum in memory
- count in ECX

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

.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
```

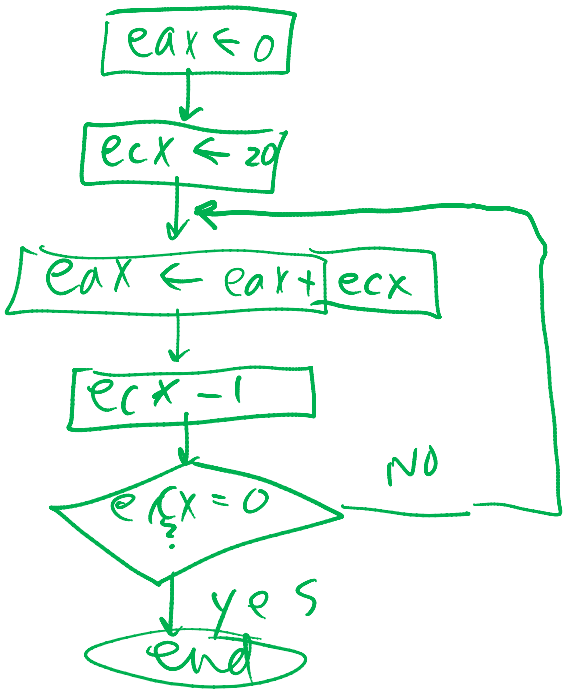
`loop` instruction

- 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
```

LoopEx.asm --- Breakpoint (F9)

Solution Explorer | LoopEx.asm

```
INCLUDE Irvine32.inc

.DATA
number    DWORD     20
.CODE
main          PROC
              mov   EAX, 0
              mov   ECX, number
forCount:     add   EAX, ECX
              loop  forCount
              call  DumpRegs
              exit
main          ENDP
END main
```

C:\Windows\system32\cmd.exe

```
  EAX=000000D2  EBX=7EFDE000  ECX=00000000  EDX=00401005
  ESI=00000000  EDI=00000000  EBP=0018FF94  ESP=0018FF8C
  EIP=00401024  EFL=00000206  CF=0  SF=0  ZF=0  OF=0  AF=0  PF=1

Press any key to continue . . .
```

Coding Assignment 2

- 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: Flowchart, Code, Debugging steps, results with different values of n, with screen captures.
  - Submission: TUESDAY Nov 24: **5:10pm (softcopy)**

Schedule of Final Weeks

- T 11/24/2015:
  - Regular Class on “Arrays” [Lecture4 Part 4 continued--]
  - Explanation of Extra Credit Work {Making up the score in Exam1 - up to 85}
  - Explanation of ASM Coding Project
  - ASM assignment 2 submission: **5:10pm (softcopy)**
- R 11/26/2015: No Class – Thanksgiving Holiday
- T 12/1/2015:
  - Regular Class on “Procedures”[Lecture4 Part 5]
  - Submission of ASM Extra Credit Work (optional) for your not so good score in Exam #1: **5:10pm (softcopy Report + ASM code) 2 separate files**
- R 12/3/2015:
  - **Final Exam** (covers all the subjects since Exam #1) **5:10 – 6:30pm**
  - Lecture 4 Parts 1, 2, 3, 4, and 5
- T 12/8/2015:
  - Submission of ASM Coding Project: **11:59pm (softcopy Report + ASM code) 2 separate files**

Array

- We use Arrays to store collection of data values
- We use LOOPs to manipulate the data in arrays
- Storage for an array can be reserved using the DUP ("duplicate") directive in the DATA segment of a program
- **Approach 1 – Indirect Address**
  - **`lea`** (Load Effective Address) to get the address of first array element (EX) **`lea EBX, nArray`**
  - [EBX] format, for example, to get the content of the address stored in the EBX --- EBX has the memory location of the content.
  - ECX is reserved as a Loop counter
- Example Code: ArrayAvg.asm
  - For an array of DWORD integers, (1) find their average and (2) add 10 to each number smaller than average.

Array

- We use Arrays to store collection of data values
- We use LOOPs to manipulate the data in arrays
- Storage for an array can be reserved using the DUP ("duplicate") directive in the DATA segment of a program
- **Approach 2 – Index Addressing**
  - Index Addressing Mode for Effective Address (EA)
  - EA = Displacement + Size*Content_of_Index_Register

    Format: `D[Size*Index]`

    Example: `nArray[4*ECX];` contents of the address determined by nArray+ 4*[ECX]
- Example Code: ArrayAvg2.asm
  - For an array of DWORD integers, (1) find their average and (2) add 10 to each number smaller than average.

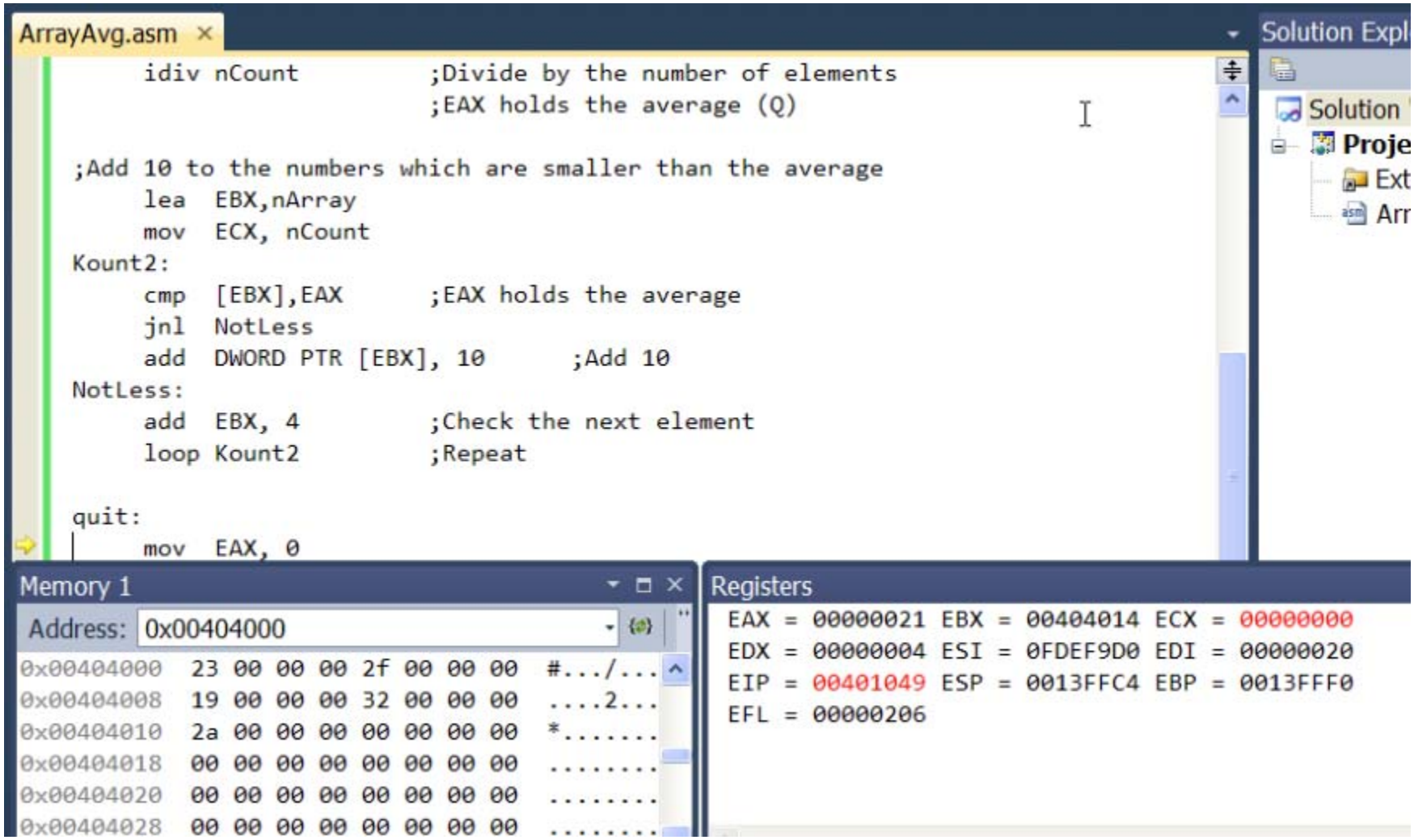
ArrayAvg1.asm

LEA --- Load effective address

PTR --- override/declare the size of an operand

```
TITLE ARRAY Example      (ArrayAvg.asm)


INCLUDE Irvine32.inc
.data
;prompt BYTE    "Enter your number:  ",0
nArray  DWORD   25, 47, 15, 50, 32, 95 DUP (?)
nCount  DWORD   5

.code
main PROC
        mov     EAX,0           ;EAX is for sum
        lea     EBX,nArray      ;Get the address of nArray to   EBX
        mov     ECX, nCount     ;Number of element in ECX for counting
        jecxz   quit            ; jecxz (Jump if (E)CX = 0 )

;Count and get Average
Kount1:
        add     EAX, [EBX]              ; the element
        add     EBX, 4          ; the next element
        loop    Kount1          ; Repeat nCount times

        cdq                             ;Convert to Quad Word
        idiv    nCount          ;Divide by the number of elements
                                ;EAX holds the average (Q)

;Add 10 to the numbers which are smaller than the average
        lea     EBX,nArray
        mov     ECX, nCount
Kount2:
        cmp     [EBX],EAX               ;EAX holds the average
        jnl     NotLess
        add     DWORD PTR [EBX], 10             ;Add 10
NotLess:
        add     EBX, 4          ;Check the next element
        loop    Kount2          ;Repeat

quit:
        mov     EAX, 0


        exit
main ENDP
END main
```

ArrayAvg1.asm DEBUGGING



ArrayAvg2.asm

Data/Content in an array

=[d + size*index]

Format

d[size*index]

nArray[4*ECX]

displacement: d address of the first data in the array

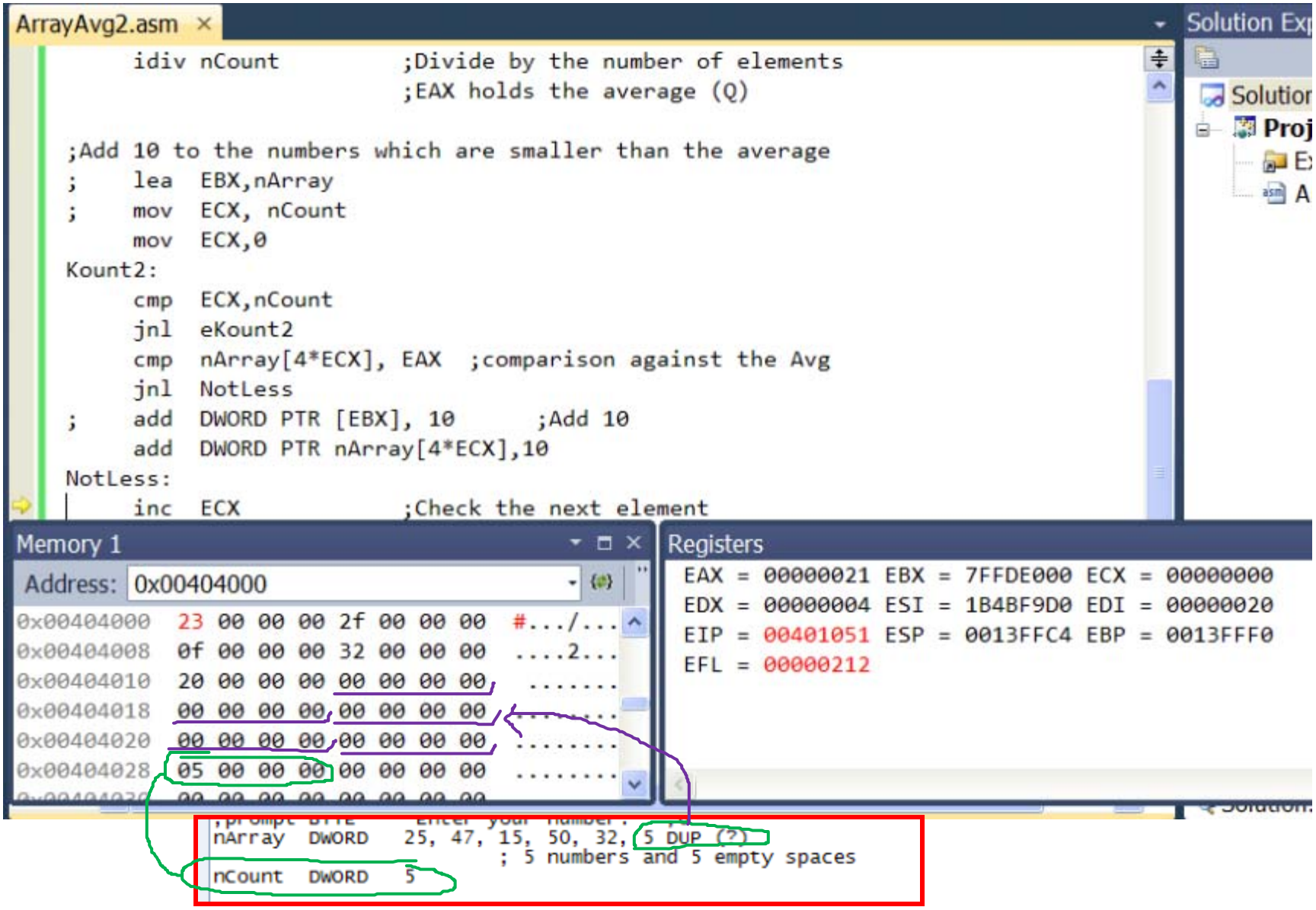
index: number of data in the array

size: Size of the data

PTR --- override/declare the size of an operand

```
; Effective address = Address_of_nArry + 4*[ECX]
;
INCLUDE Irvine32.inc
.data
;prompt BYTE    "Enter your number:  ",0
nArray  DWORD   25, 47, 15, 50, 32, 5 DUP (?)
                        ; 5 numbers and 5 empty spaces
nCount  DWORD   5

.code
main PROC
        mov     EAX,0           ;EAX is for sum
        mov     ECX,0           ;Index for EA

;       lea     EBX,nArray      ;Get the address of nArray to   EBX
;       mov     ECX, nCount     ;Number of element in ECX for counting
;       jecxz   quit            ; jecxz (Jump if (E)CX = 0 )

;Count and get Average
Kount1:
        cmp     ECX, nCount             ;
        jnl     eKount1                 ;Index<= 5 ?
        add     EAX, nArray[4*ECX]
        inc     ECX                                     ;the next element
        jmp     Kount1          ; Repeat nCount times
eKount1:
        cdq                             ;Convert to Quad Word
        idiv    nCount          ;Divide by the number of elements
                                ;EAX holds the average (Q)

;Add 10 to the numbers which are smaller than the average
;       lea     EBX,nArray
;       mov     ECX, nCount
        mov     ECX,0
Kount2:
        cmp     ECX,nCount
        jnl     eKount2
        cmp     nArray[4*ECX], EAX      ;comparison against the Avg
        jnl     NotLess
;       add     DWORD PTR [EBX], 10             ;Add 10
        add     DWORD PTR nArray[4*ECX],10
NotLess:
        inc     ECX                     ;Check the next element
        jmp     Kount2          ;Repeat

eKount2:
        mov     EAX, 0
```

ArrayAvg2.asm DEBUGGING



Extra Credit Coding Work [Optional]

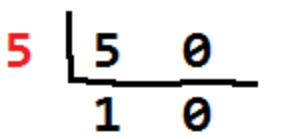
- This optional work may improve Exam #1 score up to 85 points
- Design and write an ASM code which
  - Determines a number read from Keyboard is a prime number of not
    - Prime Number: A Natural number (N) that is divisible only by 1 and itself
  - Divide N by every single number between 2 and N
    - Pseudo-Code
      - If N = 1 or 2, it is a prime. END
      - Ans = Prime
      - Loop [k = 2; N] : Is the remainder of N/k zero?
        - » If yes, Ans=Non-Prime
      - END

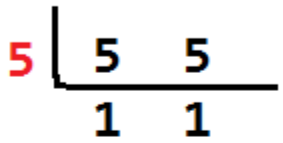
Extra Credit Coding Work [Optional]

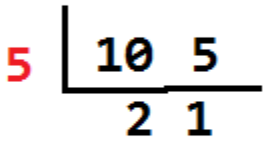
- Build and Debug
- Report [YourName.docx]
  - Code
  - Debug results
  - Debug Screen captures
  - Other relevant information
- ASM Code [YourName.asm]
- **Submit 2 separate files in 1 email**
- Submission Due: Tuesday Dec 1, 2015
  5:10 pm

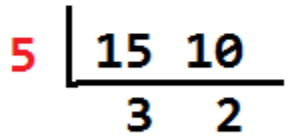
ASM Coding Project

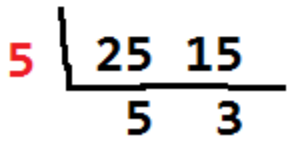
- GCD (Greatest common divisor)

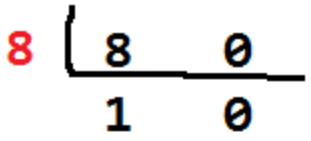


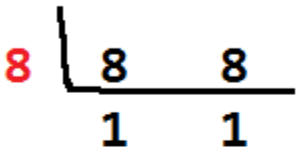


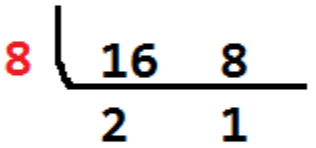


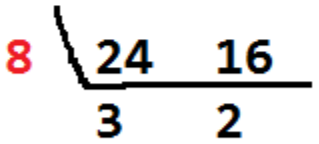


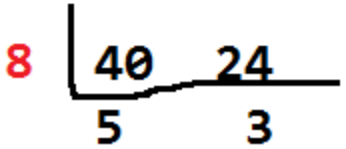












What pattern do we see here in GCD calculation ?

ASM Coding Project

- GCD Algorithm?
  - Euclidean Algorithm
  - Bezout's Identity (for $n_1 > n_2$)
    - **GCD(n_1, n_2) = GCD (n_1-n_2, n_2)**
    - Rename or arrange so that: **n_1** >= **n_2**
    - Repeat the process
    - Until one of the numbers is 0
    - Then, GCD is the non-zero number

GCD – Euclidean Algorithm

```
 GCD(2,3)
=GCD(3-2,2)
=GCD(1,2)
=GCD(2-1,1)
=GCD(1,1)
=GCD(1-1,1)
=GCD(0,1)
--> GCD(2,3)=1
```

```
 GCD(9,15)
=GCD(15-9,9)
=GCD(6,9)
=GCD(9-6,6)
=GCD(3,6)
=GCD(6-3,3)
=GCD(3,3)
=GCD(3-3,3)
=GCD(0,3)
--> GCD(9,15)=3
```

```
 GCD(105,252)
=GCD(252-105,105)
=GCD(147,105)
=GCD(147-105,105)
=GCD(42,105)
=GCD(105-42,42)
=GCD(63,42)
=GCD(63-42,42)
=GCD(21,42)
=GCD(42-21,21)
=GCD(21,21)
=GCD(21-21,21)
=GCD(0,21)
--> GCD(105,252)=21
```

Coding Project --- Instruction

- 1. Write a Code which
  - reads 2 numbers from keyboard (the first number is not always bigger than the second number)
  - calculates GCD of the 2 numbers,
  - Stores the GCD result to a memory location
- 2. **Submission Report (DOC(X) file) which includes**
  - Flow Chart of your code
  - Your code
  - Sample execution [Debug] of the code with result (screen captures & explanations of numbers)
  - Report (DOCX file): **yourname_project.docx**
- **4. Submission of asm code itself.**
  - : **yourname_project.asm**
- **5. Submission (2 separate files in 1 email) Due: Tuesday, Dec 8, 2015 11:59pm**