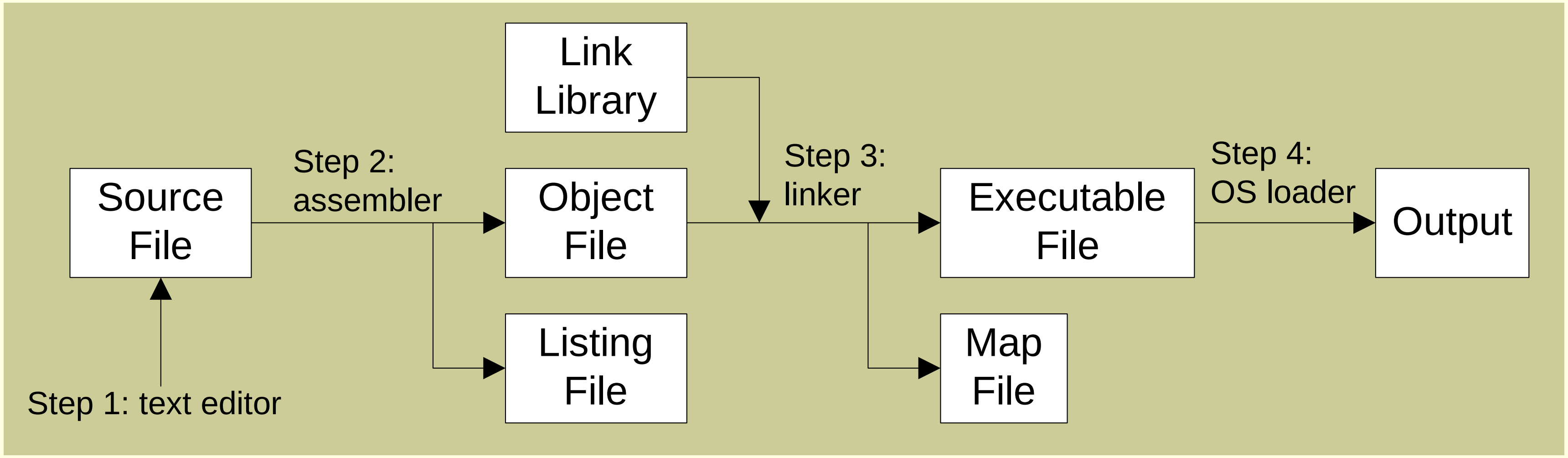
Assembly Programming

Mohsen Nickray
University of QOM

Assemble-link execute cycle

- The following diagram describes the steps from creating a source program through executing the compiled program.
- If the source code is modified, Steps 2 through 4 must be repeated.



Assembly files (.asm)

- We will be using the **nasm** assembler
- Program Components
  - Comments
  - Labels
  - Directives
  - Data
  - Main subroutine, which is a global one
  - Instructions: generally the format of an NASM instruction is as follows

Label   Instruction   Operands   ; Comment

Program Organization : Skeleton file

```
%include "io.inc"

section .data
msg db 'Hello, world!', 0

section .text
global CMAIN
        CMAIN:
                mov bp, sp
                PRINT_STRING msg
                NEWLINE
                xor ax, ax
                mov bl,4
                div bl
                PRINT_DEC 4,ax
                ret
```

Comments

- Comments are denoted by semi-colons (;).
- Everything from the semi-colon to the end of the line is ignored.

Labels

- Labels identify
  - The start of subroutines or locations to jump to in your code
  - Variables are declared as labels pointing to specific memory locations
- Labels are local to your file/module unless you direct otherwise
- The colon identifies a label (an address!)
- Example: NewLabel:
- To define a label as global we say

```
global NewLabel
```

Directives

- Direct the assembler to do something
  - Define constants
  - Define memory to store data into
  - Group memory into segments
  - Conditionally include source code
  - Include other files

Equ and % define directives

- The equ directive
  - Used to define named *constants* used in your assembly program
  - Syntax: symbol equ value
  - Similar to C's const directive :(const int symbol = value)

- The %define directive
  - Similar to C's #define directive (#define name value)
  - Most commonly used to define constant macros:

    ```
    %define SIZE 100
    mov    eax, SIZE
    ```

  - Macros can be *redefined*, and can be more complex than simple constants

Data directives

- Used in data segments to define room for memory
- There are two ways memory can be reserved
  - Defines room for data **without** initial value (segment .bss)
    - Using : **RESX** directive
  - Defines room for data **with** initial value (segment .data)
    - Using : **DX** directive
  - *Note:* X is replaced with a letter that determines the size of the object as following

Unit	Letter
byte	B
word	W
double word	D
quad word	Q
ten bytes	T

Letters for RESX and DX Directives

Example: Data Directives

```
L1 db    0        ;byte labeled L1 w/ initial value 0 decimal
L2 dw    1000     ;word labeled L2 w/ initial value 1000 decimal
L3 db    110101b  ;byte labeled L3 w/ initial value 110101 binary( 53)
L4 db    12h      ;byte labeled L4 w/ initial value 12 hex (18 decimal)
L5 db    17o      ;byte labeled L5 w/ initial value 17 octal (15 decimal)
L6 dd    1A92h    ;doubleword labeled L6 initialized to hex 1A92
L7 resb  1        ;1 uninitialized byte
L8 db    “A”      ;byte initialized to ASCII of A = 65
L9 resw  100      ; reserves room for 100 words
```

- Note: Double quotes and single quotes are treated the same

More examples

- Sequences of memory may also be defined.

```
L10   db 0, 1, 2, 3               ; defines 4 bytes
L11 db "w", "o", "r", ’d’, 0 ; defines a C string = "word"
L12 db ’word’, 0              ; same as L11
```

- For large sequences, NASM’s TIMES directive is often useful.

```
L13  times 100 db 0 ; equivalent to 100 (db 0)’s
```

Labels

- Labels point a place in memory
- In NASM assembler
  - **[L]** means contents of address L
  - **L** means address.
- Example

```
mov     al, [L1]       ; copy byte at L1 into AL
mov     eax, L1        ; EAX = address of byte at L1
mov     [L1], ah       ; copy AH into byte at L1
mov     eax, [L6]      ; copy double word at L6 into EAX
add     eax, [L6]      ; EAX = EAX + double word at L6
add     [L6], eax      ; double word at L6 += EAX
mov     al, [L6]       ; copy first byte of double word at L6 into AL
```

Labels

- Example
  - Error

```
mov     [L6], 1               ; store a 1 at L6
```

  - Correct

```
mov     dword [L6], 1         ; store a 1 at L6
```

  - BYTE, WORD, QWORD, TWORD

Input and Output (IO.inc)

Macro name	Description
PRINT_UDEC *size, data* PRINT_DEC *size, data*	Print number *data* in decimal representation. *size* – number, giving size of *data* in bytes - 1, 2, 4 or 8 (x64).*data* must be number or symbol constant, name of variable, register or address expression without size qualifier (byte[], etc.). PRINT_UDEC print number as unsigned, PRINT_DEC — as signed.
PRINT_HEX *size, data*	Similarly previous, but data is printed in hexadecimal representation.
PRINT_CHAR *ch*	Print symbol *ch*. *ch* - number or symbol constant, name of variable, register or address expression without size qualifier (byte[], etc.).
PRINT_STRING *data*	Print null-terminated text string. *data* - string constant, name of variable or address expression without size qualifier (byte[], etc.).
NEWLINE	Print newline ('\n').
GET_UDEC *size*, *data* GET_DEC *size*, *data*	Input number data in decimal representation from stdin. *size* – number, giving size of *data* in bytes - 1, 2, 4 or 8 (x64). *data* must be name of variable or register or address expression without size qualifier (byte[], etc.). GET_UDEC input number as unsigned, GET_DEC — as signed. It is not allowed to use esp register.
GET_HEX *size, data*	Similarly previous, but data is entered in hexadecimal representation with 0x prefix.
GET_CHAR *data*	Similarly previous, but macro reads one symbol only.
GET_STRING *data, maxsz*	Input string with length less than *maxsz*. Reading stop on EOF or newline and "\n" writes in buffer. In the end of string 0 character is added to the end. *data* - name of variable or address expression without size qualifier (byte[], etc.). *maxsz* - register or number constant.

Program Organization : Skeleton file

```
; file: skel.asm
; This file is a skeleton that can be used to start assembly programs.

%include   "io.inc"
segment   .data
; initialized data is put in the data segment here

segment   .bss
; uninitialized data is put in the bss segment

segment   .text
        global  asm_main
asm_main:

; code is put in the text segment. Do not modify the code before
; or after this comment.

ret
```

segment == section

Example Program **#1**

```
%include "io.inc"
section .data
      result   dw   0
      prompt  db  "Enter a number", 0
       msg1   db    "! ",0
       msg2   db    " = ",0

section    .bss
        input  resb  1

section    .text
      global   CMAIN
```

```
CMAIN:
            PRINT_STRING   prompt
            NEWLINE
            GET_DEC   1,input
            MOV     al,[input]
            MOV    cl, al
Back:       DEC cl
            MUL    cl
            MOV    bl,cl
            DEC     bl
            JNZ     Back
            MOV    [result], AX
            PRINT_STRING    msg1
            PRINT_UDEC      1,[input]
            PRINT_STRING     msg2
            PRINT_DEC       2,[result]
            NEWLINE
ret
```

Example #2

```
%include "io.inc"
section .data
   source   db   0,1,2,3
   destin   db   20,20,20,20

section .text
   global   CMAIN
   CMAIN:
            MOV CX, E000H
            MOV AX, B001H

            MOV   DL, [source]
            MOV   [destin], DL
            MOV   DL, [source+1]
            MOV   [destin+1], DL
            MOV   DL, [source+2]
            MOV   [destin+2], DL
            MOV   DL, [source+3]
            MOV   [destin+3], DL
            PRINT_UDEC   1,[destin]
            PRINT_UDEC   1,[destin+1]
            PRINT_UDEC   1,[destin+2]
            PRINT_UDEC   1,[destin+3]
ret
```

Example **#2** (cont.)

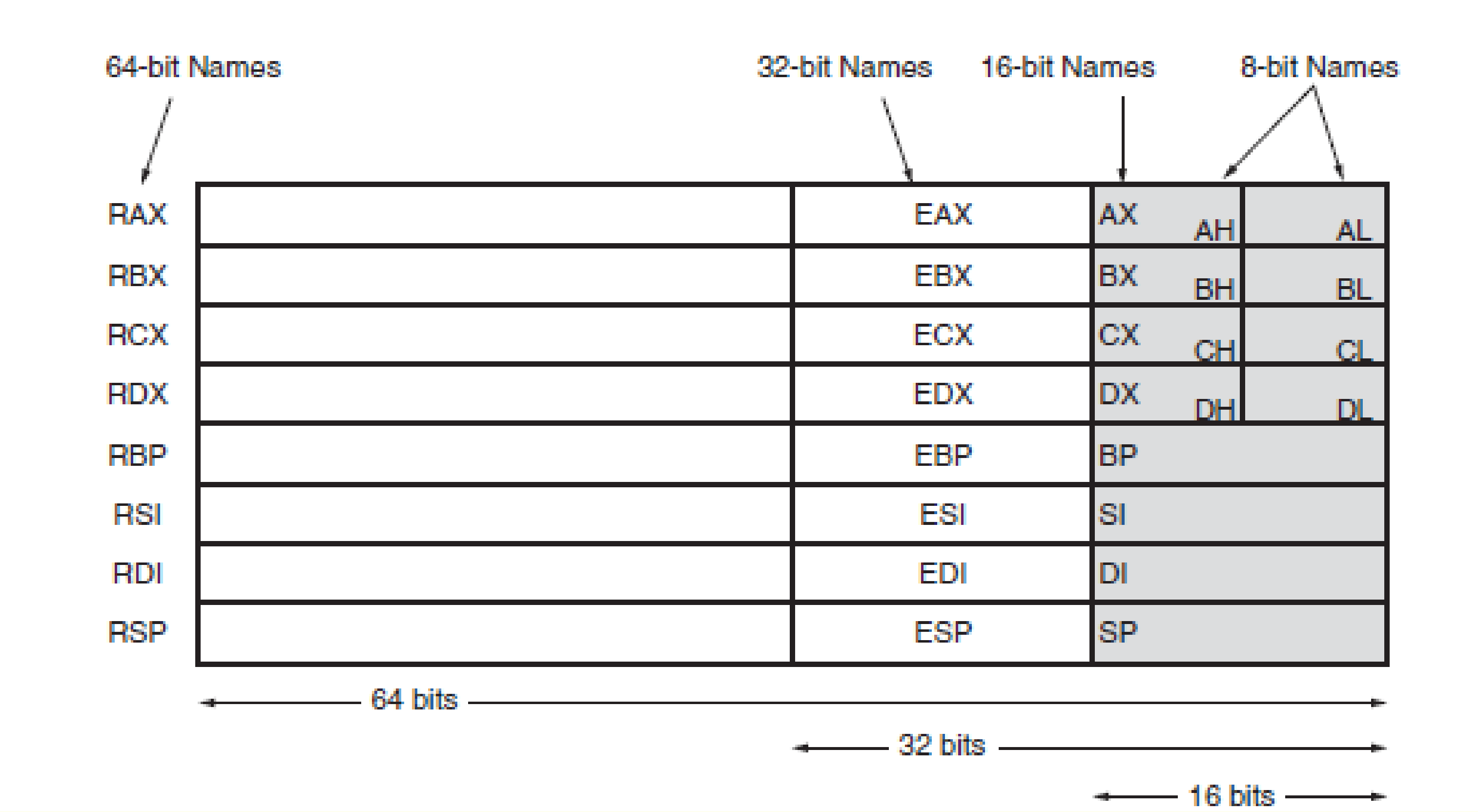
```
%include "io.inc"
section     .bss
   source    resb    4
   destin    resb    4

section    .text
    global     CMAIN
    CMAIN:

                GET_DEC    1, source
                GET_DEC    1, source+1
                GET_DEC    1, source+2
                GET_DEC    1, source+3
```

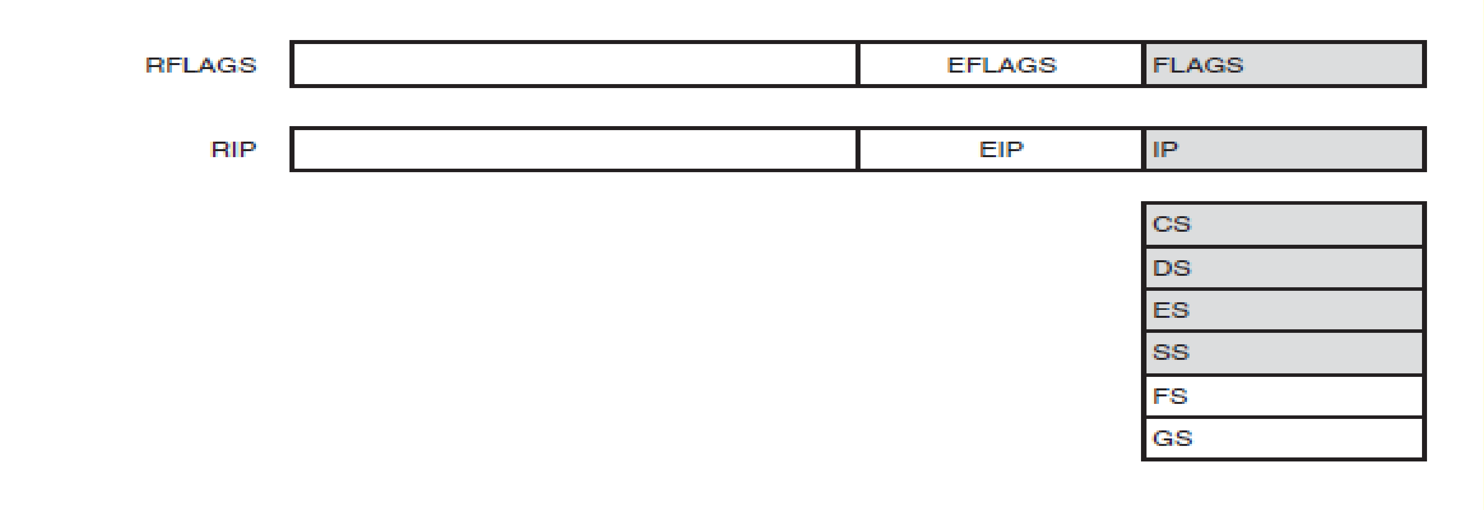
Example #3

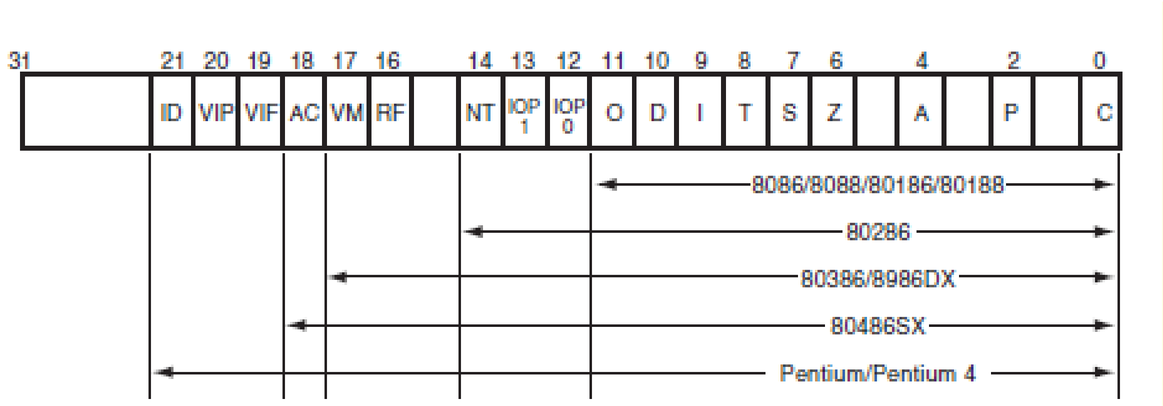
```
%include "io.inc"
section .data
    Multiplier dw 1234H
    Multiplicant dw 3456H

section .bss
    Product resw 2

section .text
    global CMAIN
    CMAIN:
```

```
MOV          AX, [Multiplicant]
MUL          word [Multiplier]
MOV          [Product], AX
MOV          [Product+2], DX
PRINT_HEX    2,[Product+2]
PRINT_HEX    2,[Product]

ret
```

General Purpose Processors From 8086 to Pentium




RFLAGS
EFLAGS
FLAGS
RIP
EIP
IP
CS
DS
ES
SS
FS
GS


31 21 20 19 18 17 16 14 13 12 11 10 9 8 7 6 4 2 0
ID VIP VIF AC VM RF NT IOP 1 IOP 0 O D I T S Z A P C
8086/8088/80186/80188
80286
80386/8986DX
80486SX
Pentium/Pentium 4

Reverse an String

```
%include "io.inc"
section .data
String1  db 'assembly language program'
Length  dw  $-String1

section    .text
global CMAIN
```

```
CMAIN:
        MOV    ESI, String1
        MOV    ECX, [Length]
        ADD     ESI, ECX
        DEC     ESI
        Back:
                MOV    DL, [ESI]
                PRINT_CHAR   DL
                DEC     ESI
        LOOP    Back
ret
```

Example (Maximum Value)

```
%include "io.inc"
section .data
List   db   0,1,2,3,98,01,13,78,18,36
Result db 0
section .text
global CMAIN
```

```
CMAIN:
                MOV   ESI, List
                MOV   AL, 00H
                MOV   ECX, 0AH
        Back:   CMP AL, [ESI]
                JNC Ahead
                MOV AL, [ESI]
        Ahead:  INC   ESI
                LOOP  Back
                MOV   [Result], AL
                PRINT_DEC   1,Result
ret
```

Example (Addition of an array elements)

```
%include "io.inc"
section   .data
List   db   10,1,2,3,98,01,13,78,18,36
Total    dw    0
CarryNum   db   0


section .text
global CMAIN
```

```
CMAIN:
          MOV    CX, 0AH ; counter
          MOV     BL, 00H ; to count carry
          MOV     ESI, List
          MOV     AL,0
Back:
          ADD    AL,[ESI]
          JC      Label
Back1:    INC    ESI
          DEC    CX
          JNZ    Back
          MOV    [Total], AX
          MOV    [CarryNum], BL
          PRINT_UDEC     2, Total
          PRINT_UDEC     1, CarryNum
          JMP     ENDT
Label:    INC     BL
          JMP    Back1
ENDT:

ret
```

imul

dest	source1	source2	Action
	reg/mem8		AX = AL*source1
	reg/mem16		DX:AX = AX*source1
	reg/mem32		EDX:EAX = EAX*source1
reg16	reg/mem16		dest *= source1
reg32	reg/mem32		dest *= source1
reg16	immed8		dest *= immed8
reg32	immed8		dest *= immed8
reg16	immed16		dest *= immed16
reg32	immed32		dest *= immed32
reg16	reg/mem16	immed8	dest = source1*source2
reg32	reg/mem32	immed8	dest = source1*source2
reg16	reg/mem16	immed16	dest = source1*source2
reg32	reg/mem32	immed32	dest = source1*source2

```
imul   dest, source1
imul   dest, source1, source2
```

Multiplication

```
mov     ebx, eax
imul    ebx, [input]          ; ebx *= [input]

imul   ecx, ebx, 25           ; ecx = ebx*25
```

64 bit adder

```
%include "io.inc"
SEGMENT .data
MyInput1 dd 1000H, 0200H
MyInput2 dd 0F000H, 0300H


SEGMENT .bss
Result resd 2


section .text
global CMAIN
```

```
CMAIN:
      MOV    EAX,[MyInput1]
      MOV    EBX,[MyInput2]
      MOV    ECX,[MyInput1+4]
      MOV    EDX,[MyInput2+4]

      ADD     EBX,EAX
      ADC     EDX,ECX
      MOV     [Result], EBX
      MOV     [Result+4], EDX
      PRINT_HEX    4,[Result]
      PRINT_HEX    4,[Result+2]
ret
```

64 bit subtractor

```
%include "io.inc"
SEGMENT .data
MyInput1  dd   0200H, 0200H
MyInput2  dd   0100H, 0100H

SEGMENT .bss
Result resd 2

section .text
global CMAIN
```

```
CMAIN:
        MOV EAX,[MyInput1]
        MOV EBX,[MyInput2]
        MOV ECX,[MyInput1+4]
        MOV EDX,[MyInput2+4]


        SUB EBX,EAX
        SBB EDX,ECX
        MOV [Result], EBX
        MOV [Result+4], EDX
        PRINT_HEX 4,[Result]
        PRINT_HEX 4,[Result+4]
ret
```

Calling procedures and using the stack

- call proc_name
  - Pushes the instruction pointer (IP/EIP)
  - Pushes CS to the stack if the call is to a procedure outside the code segment
  - Unconditional jump to the label proc_name
- ret
  - Pop saved IP/EIP and if necessary the saved CS and restores their values in the registers

Sub Routine

- CALL / RET

```
…
Call  proc_name
…

proc_name:
          …
ret
```

Sub routine (Pass parameters using Registers)

- int add(int a, int b) { return a+b; } ...

```
push eax, [b]
push ebx, [a]
call add

add:
        mov   ecx, ebx
        mov   ecx, eax
ret
```

Sub routine (Pass parameters using STACK)

- int add(int a, int b) { return a+b; } ...

```
push b
push a
call add
add esp, 8
;or pop ebx ; remove parameter
; pop ebx ; remove parameter ...

add:
        ; [esp] is the return address,
        ; [esp+4] the first parameter, etc.
        mov  eax, [esp+4]
        add   eax, [esp+8]
ret
```

Another option for updating SP

```
push b
push a
call add
…

add:
        ; [esp] is the return address,
        ; [esp+4] the first parameter, etc.
        mov  eax, [esp+4]
        add   eax, [esp+8]
ret  8
```

Update SP by Callee or caller

int add(int a, int b) { return a+b; }

By Callee

```
...
push b
push a
call add
...
add:
        ; [esp] is the return address,
        ; [esp+4] the first parameter, etc.
        mov eax, [esp+4]
        add eax, [esp+8]
        push ebx ; save ebx
        mov ebx, [esp+4] ; return address (after ebx)
        sub esp, 16 ; ebx, ret addr, 1st param, 2nd param
        push ebx ; restore return address
        mov ebx, [esp-12] ; 16-4 for return address
ret
```

By Caller

```
…
push b
push a
call add
add esp, 8
;or pop ebx ; remove parameter
; pop ebx ; remove parameter
 ...

add:
        ; [esp] is the return address,
        ; [esp+4] the first parameter, etc
        mov  eax, [esp+4]
        add   eax, [esp+8]
ret
```

Example (Initialization)

```
%include "io.inc"
section .data
        x dd 1, 5, 2, 18, 8888, 168
        n dd 3
        sum dd 0

global CMAIN
CMAIN:
        push x
        push n
        call init
        add esp, 8 ; clean up the stack
        top:
        add ebx, [ecx]
        add ecx, 4
        dec eax
        jnz top
        done: mov [sum], ebx
ret

init:
        mov ebx, 0
        mov ecx, [esp + 8]
        mov eax, [esp + 4]
ret
```

Another Example

```
%include "io.inc"
section .data
        x dd 1, 5, 2, 18, 8888, 168
        n dd 3
        sum dd 0
global CMAIN
CMAIN:
        push DWORD x
        push DWORD 168
        push DWORD 6
        call findfirst
ret

findfirst:
        mov ecx, DWORD [4+esp]
        mov ebx, DWORD [8+esp]
        mov eax, DWORD [12+esp]
        mov edx, eax
top:
        cmp    ebx, DWORD [eax]
        jz found
        dec ecx
        jz notthere
        add eax, DWORD 4
        jmp top
found:
        sub eax, edx
        add eax,4
        shr eax, 2
ret
notthere:
        mov eax, -1
ret
```

Local Variables in sub programs

- Stack is the best place for local variables
- Data not stored on the stack is using memory from the beginning of the program until the end of the program (C calls these types of variables global or static). Then ***Using the stack for variables also saves memory***.
- Data stored on the stack only use memory when the subprogram is active.
- Using the stack for variables is important if one wishes subprograms to be reentrant. A reentrant subprogram will work if it is invoked at any place, including the subprogram itself. In other words, reentrant subprograms can be invoked recursively.

Example

- Example:

```
void calc_sum( int n, int * sump )
{
  int i, sum = 0;

  for( i=1; i <= n; i++ )
    sum += i;
  *sump = sum;
}
```

- We use EBP to access local variables

```
subprogram_label:
      push   ebp                  ; save original EBP value on stack
      mov    ebp, esp             ; new EBP = ESP
      sub    esp, LOCAL_BYTES     ; = # bytes needed by locals
; subprogram code
      mov    esp, ebp             ; deallocate locals
      pop    ebp                  ; restore original EBP value
      ret
```

Example (Cont.)

```
cal_sum:
      push   ebp
      mov    ebp, esp
      sub    esp, 4                  ; make room for local sum

      mov    dword [ebp - 4], 0      ; sum = 0
      mov    ebx, 1                  ; ebx (i) = 1
for_loop:
      cmp    ebx, [ebp+8]            ; is i <= n?
      jnle   end_for

      add    [ebp-4], ebx            ; sum += i
      inc    ebx
      jmp    short for_loop

end_for:
      mov    ebx, [ebp+12]           ; ebx = sump
      mov    eax, [ebp-4]            ; eax = sum
      mov    [ebx], eax              ; *sump = sum;

      mov    esp, ebp
      pop    ebp
      ret
```

Macros

- Procedures have some extra overhead to execute (call/ret statements, push/pop IP, CS and data from the stack)
- A macro is a piece of code which is “macroexpanded” whenever the name of the macro is encountered
- Note the difference, a procedure is “called”, while a macro is just “expanded/inlined” in your program
- Macros are faster than procedures (no call instructions, stack management etc.)
- But they might
  - Significantly increase code size
  - Hard to debug

Macros

- Macros are defined using **%macro** and **%endmacro** directives
- Typical macro definition

```
%macro    macro_name    para_count
          <macro_body>
%endmacro
```

Specifies number of parameters

- Example 1: A parameterless macro

```
%macro    multEAX_by_16
          sal     EAX,4
%endmacro
```

Macros (cont'd)

- Example 2: A parameterized macro

```
%macro    mult_by_16   1
          sal      %1,4
%endmacro
```

one parameter

- Example 3: Memory-to-memory data transfer

```
%macro    mxchg   2
          xchg      EAX,%1
          xchg      EAX,%2
          xchg      EAX,%1
%endmacro
```

two parameters

Macro example

```
%macro DIV16   3          ; result=x/y
    MOV AX, %{2}          ; take the dividend
    CWD                   ; sign-extend it to DX:AX
    IDIV %{3}             ; divide
    MOV     %{1},AX       ; store quotient in result
%endmacro
```

Macro example

```
; Example: Using the macro in a program
; Variable Section
    varX1       DW      20
    varX2       DW      4
    varR        RESW
; Code Section
    DIV16 word [varR], word [varX1], word [varX2]

; Will actually generate the following code inline in your
; program for every instantiation of the DIV16 macro (You
; won't actually see this unless you debug the program).
;   MOV  AX, word [varX1]
;   CWD
;   IDIV word [varX2]
;   MOV  word [varR], AX
```

Another Example

```
%macro    write_string 2
          mov     eax, 4
          mov     ebx, 1
          mov     ecx, %1
          mov     edx, %2
          int     80h
 % endmacro
section    .text
global    _start
          write_string    msg1, len1
          write_string    msg2, len2
          write_string    msg3, len3
          mov      eax,1            ;system call number (sys_exit)
          int      0x80             ;call kernel
section .data
          msg1   db    'Hello, programmers!',0xA,0xD
          len1    equ    $ - msg1
          msg2   db    'Welcome to the world of,', 0xA,0xD
          len2     equ    $- msg2
          msg3   db     ‘Assembly programming! '
          len3     equ     $- msg3
```

Defining Constants

- NASM provides three directives:
  - **`EQU`** directive
    - No reassignment
    - Only numeric constants are allowed
  - **`%assign`** directive
    - Allows redefinition
    - Only numeric constants are allowed
  - **`%define`** directive
    - Allows redefinition
    - Can be used for both numeric and string constants

Defining Constants

- Defining constants has two main advantages
  - Improves program readability

```
NUM_OF_STUDENTS   EQU    90
         .........
mov     ECX,NUM_OF_STUDENTS
```

    is more readable than

```
mov     ECX,90
```

  - Helps in software maintenance
    - Multiple occurrences can be changed from a single place
- Convention
  - We use all upper-case letters for names of constants

Defining Constants

The EQU directive

- Syntax:

```
name    EQU   expression
```

  - Assigns the result of **`expression`** to **`name`**
  - The **`expression`** is evaluated *at assembly time*

Examples

```
NUM_OF_ROWS    EQU    50
NUM_OF_COLS    EQU    10
ARRAY_SIZE     EQU    NUM_OF_ROWS * NUM_OF_COLS
```

Defining Constants

The `%assign` directive

- Syntax:

```
%assign name expression
```

  - Similar to EQU directive
  - A key difference
    - Redefinition is allowed

```
%assign  i  j+1
. . .
%assign  i  j+2
```

      is valid
    - Case-sensitive
    - Use **`%iassign`** for case-insensitive definition

Defining Constants

The `%define` directive

- Syntax:

```
%define name constant
```

  - Both numeric and strig constants can be defined
  - Redefinition is allowed

```
%define  X1  [EBP+4]
. . .
%assign  X1  [EBP+20]
```

    is valid

    - Case-sensitive
    - Use `%idefine` for case-insensitive definition

Example

```
%define   linefeed    0xA
%define    func(a, b)     ((a) * (b) + 2)


…
func (1, 22)      ; expands to ((1) * (22) + 2)
…
```

High-Level Language Interface

- Why program in mixed-mode?
  - Focus on C and assembly
- Overview of compiling mixed-mode programs
- Calling assembly procedures from C
  - Parameter passing
  - Returning values
  - Preserving registers
  - Publics and externals
  - Examples
- Calling C functions from assembly
- Inline assembly code

Why Program in Mixed-Mode?

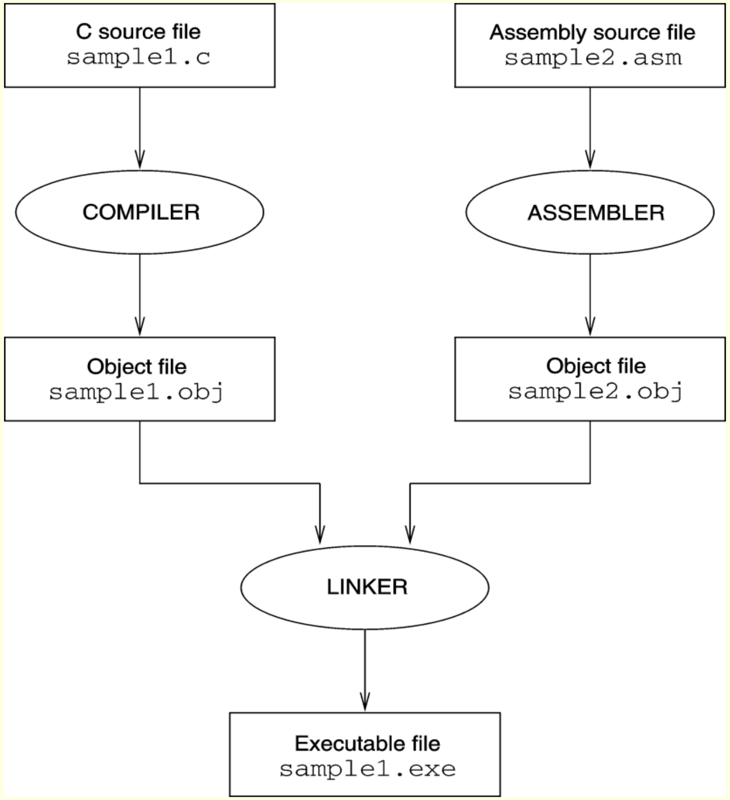
- Pros and cons of assembly language programming
  - Advantages:
    - Access to hardware
    - Time-efficiency
    - Space-efficiency
  - Problems:
    - Low productivity
    - High maintenance cost
    - Lack of portability
- As a result, some programs are written in mixed-modem (e.g., system software)

Compiling Mixed-Mode Programs

- We use C and assembly mixed-mode programming
  - Our emphasis is on the principles
- Can be generalized to any type of mixed-mode programming

- To compile

```
bcc sample1.c sample.asm
```



Example

```
#include <iostream>
using namespace std;
extern "C" int array(int a[], int length);   // external ASM procedure

int main()
{
  int a[] = {1, 3, 5, 7, 9, 2, 4, 6, 8, 0};  // array declaration
  int   array_length = 10; // length of the array

  int   sum = array(a, array_length);   // call of the ASM procedure

  cout << "sum=" << sum << endl;   // displaying the sum
cin  >> sum;
}
```

```
        global _array

        section .text
        _array:
              push ebp
              mov ebp, esp
              push ecx
              push esi

              mov ecx, [ebp+12]
              mov esi, [ebp+8]

              xor eax, eax
        lp1: add eax, [esi]
              add esi, 4
              loop lp1

              pop esi
              pop ecx
              pop ebp
              ret
```

How to run

- assemble ***array.asm***
  - `nasm -f win32 -o a.obj array.asm`
- Compile C project and link

Calling Assembly Procedures from C

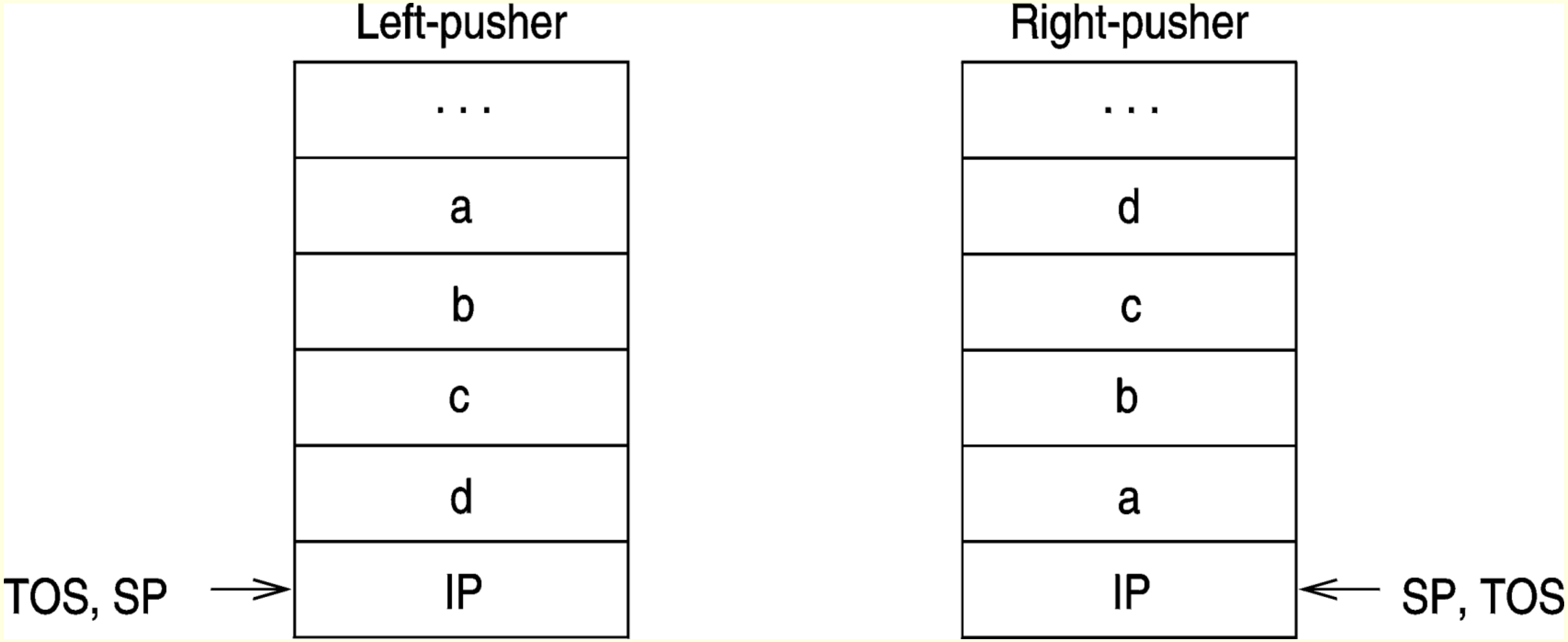
Parameter Passing

- Stack is used for parameter passing
- Two ways of pushing arguments onto the stack
  - Left-to-right
    - Most languages including Basic, Fortran, Pascal use this method
    - These languages are called *left-pusher* languages
  - Right-to-left
    - C uses this method
    - These languages are called *right-pusher* languages

Calling Assembly Procedures from C (cont'd)

Example:

```
sum(a,b,c,d)
```



Calling Assembly Procedures from C (cont'd)

Publics and External

- Mixed-mode programming involves at least two program modules
  - One C module and one assembly module
- We have to declare those functions and procedures that are not defined in the same module as external
  - **`extern`** in c
  - **`extrn`** in assembly
- Those procedures that are accessed by another modules as **`public`**

Calling Assembly Procedures from C (cont'd)

Underscores

- In C, all external labels start with an underscore
  - C and C++ compilers automatically append the required underscore on all external functions and variables
- You must make sure that all assembly references to C functions and variables begin with underscores
  - Also, you should begin all assembly functions and variables that are made public and referenced by C code with underscores

Calling C Functions from Assembly

- Stack is used to pass parameters (as in our previous discussion)
  - Similar mechanism is used to pass parameters and to return values
- C makes the calling procedure responsible for clearing the stack of the parameters
  - Make sure to clear the parameters after the **`call`** instruction as in

```
        add     SP,4
```

Example #1

```
section .data
x:
        dd 1
        dd 5
        dd 2
        dd 18
sum:
         dd 0
section   .rodata
      fmt:     db     'Sum = %d', 10, 0
section     .text
      extern      _printf
global      _main
```

```
_main:
        mov    eax, 4
        mov    ebx, 0
        mov     ecx, x
top:
        add      ebx, [ecx]
        add      ecx, 4
        dec      eax
        jnz       top
printsum:
        push    ebx
        push    DWORD fmt
        call      _printf
        add      esp, 8
ret
```

Example #2

```
section   .bss
input    resd 1
section   .rodata
          fmt:      db    'you entered %d as input', 10, 0
          fmtin    db    '%d',0
section   .text
          extern    _scanf
          extern     _printf
global _main
_main:
                    push    DWORD    input
                    push    DWORD     fmtin
                    call        _scanf
                    add        esp, 8
                    push      DWORD [input]
                    push      DWORD fmt
                    call         _printf
                    add         esp, 8
ret
```