MIT ADT University
School of Computing
Department of Computer Science Engineering

“Introduction to Processor & Chips” UNIT I Introduction to Assembly Language Programming

Class : S.Y. [CSE]

Prof. Bhushan Bhokse
Department of Computer Science Engineering
School of Computing

Introduction to Processors & Chips

Course Code: 19BTCS303
L: 03Hrs/Week
CA:40 Marks
FE:60 Marks
Credits: 03

Course Objectives

- Understand & learn assembly language programming.
- Demonstrate programming proficiency using 8086,80386 microprocessors & AVR microcontrollers to design practically motivated systems.
- Apply knowledge of the microprocessor, microcontroller & assembly language programming theory course to solve problems. This course will provide a bridge to future courses where alternative embedded system designs and their performance tradeoffs will be studied in depth.
- Relate and communicate effectively about the microprocessors, peripherals & microcontrollers used in the present gadgets.
- Develop an ability to use the techniques, skills, and modern engineering tools to accommodate different practically-motivated circuits.

Course Outcomes

- Develop program using assembly language over hardware and software.
- Apply the knowledge of memory organization.
- Develop solutions using embedded programming kit.

Unit-I: Assembly Language Programming

- Introduction to assembly language programming
- ALP tools- Assembler, linker, loader, debugger, emulator concepts
- Assembler directives
- far and near procedures, macros.
- LINUX installation, LINUX basics,
- Introduction to NASM,
- LINUX internals & system functions.
- Debugging with GNU debugger.

What is Assembly Language?

- Each personal computer has a microprocessor that manages the computer's arithmetical, logical and control activities.
- Each family of processors has its own set of instructions for handling various operations such as getting input from keyboard, displaying information on screen and performing various other jobs.
- These set of instructions are called 'machine language instructions'.
- A processor understands only machine language instructions, which are strings of 1's and 0’s.
- However, machine language is too obscure and complex for using in software development.
- So, the low-level assembly language is designed for a specific family of processors that represents various instructions in symbolic code and a more understandable form.

Language used for programming

Machine Language	Assembly Language	High level Language
Binary codes	Mnemonics	English like statements
Processor dependent	Processor dependent	Processor independent
Require less memory	Require less memory	Require more memory
Less execution time	Less execution time	More execution time
Program development is difficult	Program development is simpler than m/c language	Program development is easy
Not user friendly	Less user friendly	User friendly
E.G:Machine Language	E.G: Assembly language	E.G:BASIC,PASCAL,C

Advantages of Assembly Language

Having an understanding of assembly language makes one aware of –

- How programs interface with OS, processor, and BIOS.
- How data is represented in memory and other external devices.
- How the processor accesses and executes instruction.
- How instructions access and process data.
- How a program accesses external devices.
- It requires less memory and execution time.
- It allows hardware-specific complex jobs in an easier way.
- It is suitable for time-critical jobs.
- It is most suitable for writing interrupt service routines and other memory resident programs.

Environment Setup

- ❖Assembly language is dependent upon the instruction set and the architecture of the processor. For the practical, you will need
  - ➢An IBM PC or any equivalent compatible computer.
  - ➢A copy of Linux operating system.
  - ➢A copy of NASM assembler program.
- ❖There are many good assembler programs, such as –
  - ➢Microsoft Assembler (MASM)
  - ➢Borland Turbo Assembler (TASM)
  - ➢The GNU assembler (GAS)
- ❖We will use the NASM assembler, as it is –
  - ➢Free and can download it from various web sources.
  - ➢Well documented and will get lots of information on net.
- ❖Could be used on both Linux and Windows

Assembly - Basic Syntax

- An assembly program can be divided into three sections –
- The data section,
- The bss section, and
- The text section.

- The data Section:

- The data section is used for declaring initialized data or constants. This data does not change at runtime.
- You can declare various constant values, file names, or buffer size, etc., in this section.
- The syntax for declaring data section is –

```
section .data
```

- The bss Section
  - The bss section is used for declaring variables.
  - The syntax for declaring bss section is –

```
section .bss
```

- The text section
  - The text section is used for keeping the actual code.
  - This section must begin with the declaration `global _start` , which tells the kernel where the program execution begins.
  - The syntax for declaring text section is –

```
section .text
global _start
_start:
```

Comments

- Assembly language comment begins with a semicolon (;).
- It may contain any printable character including blank. It can appear on a line by itself, like −

```
 ; This program displays a message on screen
```

or, on the same line along with an instruction, like −

```
        add eax , ebx             ; adds ebx to eax
```

Assembly Language Programming Statements

Parameter	Instructions	Assembler Directives	Data Declaration
Basic Difference	These are commands to processor	These are commands to Assembler	These are used for declaring memory variables
Memory Usage	Yes	No	Yes
Executable / Non Executable	Executable	Non Executable	Non Executable
Examples	MOV AX,BX INC	Section .data Section .bss	Num1 DB 10h

Assembly Language Statements

- Assembly language programs consist of three types of statements –
  - Executable instructions or instructions,
  - Assembler directives or pseudo-ops, and
  - Macros.
- The Executable instructions or simply instructions tell the processor what to do. Each instruction consists of an operation code (opcode). Each executable instruction generates one machine language instruction.
- The assembler directives or pseudo-ops tell the assembler about the various aspects of the assembly process. These are non-executable and do not generate machine language instructions.
- Macros are basically a text substitution mechanism.

Syntax of Assembly Language Statements

- Assembly language statements are entered one statement per line. Each statement follows the following format −

```
[label]          mnemonic    [operands]    [;comment]
```

- The fields in the square brackets are optional.
- A basic instruction has two parts, the first one is the name of the instruction (or the mnemonic), which is to be executed, and the second are the operands or the parameters of the command.
- Following are some examples of typical assembly language statements −

```
INC COUNT          ; Increment the memory variable COUNT
MOV TOTAL, 48      ; Transfer the value 48 in the memory variable TOTAL
ADD AH, BH         ; Add the content of the BH register into the AH register
AND MASK1, 128     ; Perform AND operation on the  variable MASK1 and 128
ADD MARKS, 10      ; Add 10 to the variable MARKS
MOV AL, 10         ; Transfer the value 10 to the AL register
```

Memory Segments

- A segmented memory model divides the system memory into groups of independent segments referenced by pointers located in the segment registers.
- Each segment is used to contain a specific type of data.
- One segment is used to contain instruction codes, another segment stores the data elements, and a third segment keeps the program stack.

- **Data segment**

- It is represented by .data section and the .bss .
- The .data section is used to declare the memory region, where data elements are stored for the program.
- This section cannot be expanded after the data elements are declared, and it remains static throughout the program.
- The .bss section is also a static memory section that contains buffers for data to be declared later in the program.
- This buffer memory is zero-filled.

Memory Segments Contd…

Code segment

- It is represented by .text section.
- This defines an area in memory that stores the instruction codes.
- This is also a fixed area.

Stack Segment

- This segment contains data values passed to functions and procedures within the program.

Assembly - Variables

- NASM provides various define directives for reserving storage space for variables.
- The define assembler directive is used for allocation of storage space.
- It can be used to reserve as well as initialize one or more bytes.

Allocating Storage Space for Initialized Data

- The syntax for storage allocation statement for initialized data is −
  [variable-name] define-directive initial-value [,initial-value]...
- Where, variable-name is the identifier for each storage space. The assembler associates an offset value for each variable name defined in the data segment.
- There are five basic forms of the define directive −

Directive	**Purpose Storage**	**Space**
DB	Define Byte allocates	1 byte
DW	Define Word allocates	2 bytes
DD	Define Doubleword allocates	4 bytes
DQ	Define Quadword allocates	8 bytes
DT	Define Ten Bytes allocates	10 bytes

- Each decimal value is automatically converted to its 16-bit binary equivalent and stored as a hexadecimal number.
- Processor uses the little-endian byte ordering. Negative numbers are converted to its 2's complement representation.
- Short and long floating-point numbers are represented using 32 or 64 bits, respectively.

Allocating Storage Space for Uninitialized Data

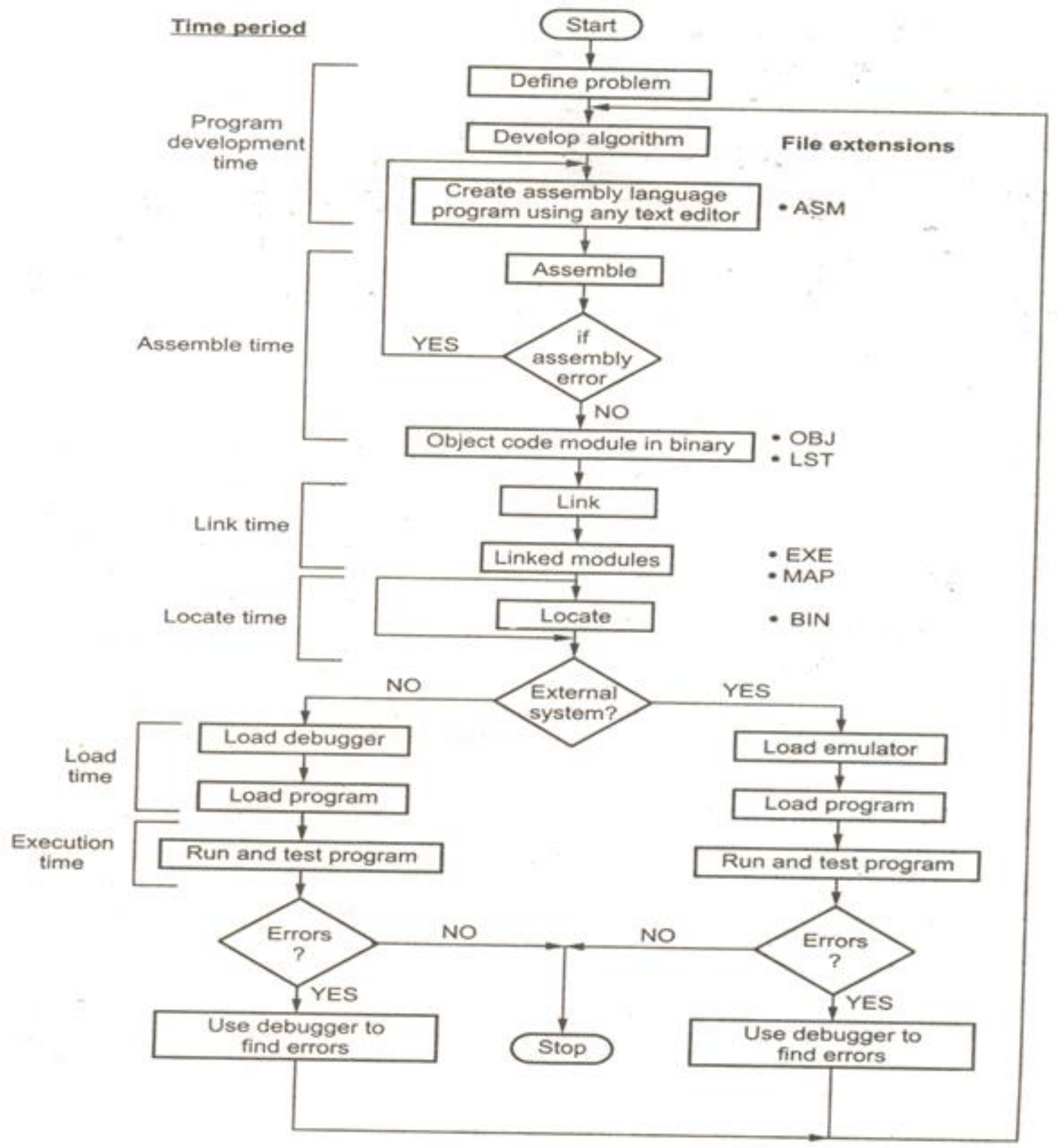
- The reserve directives are used for reserving space for uninitialized data.
- The reserve directives take a single operand that specifies the number of units of space to be reserved.
- Each define directive has a related reserve directive.
- There are five basic forms of the reserve directive −

Directive	Purpose
RESB	Reserve a Byte
RESD	Reserve a Doubleword
RESQ	Reserve a Quadword
REST	Reserve a Ten Bytes

Steps involved in programming

- Specify the problem
- Designing the problem solution
- Coding
- Debugging

Program Development Algorithm


Time period
Start
Define problem
Program development time
Develop algorithm
File extensions
Create assembly language program using any text editor
• ASM
Assemble
Assemble time
YES
if assembly error
NO
Object code module in binary
• OBJ
• LST
Link
Link time
Linked modules
• EXE
• MAP
Locate time
Locate
• BIN
NO
External system?
YES
Load debugger
Load emulator
Load time
Load program
Load program
Execution time
Run and test program
Run and test program
Errors ?
NO
NO
Errors ?
YES
YES
Use debugger to find errors
Stop
Use debugger to find errors

Assembly Language Programming Tools

- Assembler
- Linker
- Loader
- Locator
- Debugger
- Emulator

Assembler

- Assembler translates source file into binary or object code.
- It reads the source file from the disk.
- Assembler generates 2 files on disk during two passes.
- Object file:Contains binary codes of the instructions & information about the addresses of the instructions.
- List file: Contains assembly language statements, binary codes for each instructions & the offset for each instruction.
- In the first pass assembler performs following operations:
  - Reading source program
  - Creating symbol table
  - Replacing all mnemonic codes by binary codes
  - Detecting any syntax errors
  - Assigning relative addresses to instructions and data.
- In the second pass it replaces the symbols in the operand field with the addresses from symbol table.

Linker

- Linker is a program used to join together several object files into one large object file.
- Linker produces a link file which contains the binary codes for all the combined modules.
- The linker also produces the link map which contains the address information about the link files.
- Linker generates re-locatable program which can be placed anywhere in memory.
- Linker generates .exe, .map files.

Loader

- The loader is a part of the operating system that brings an executable file residing on disk into memory.
- Loader is responsible for loading and relocation.
- Loaders are of 2 types
- Absolute loader:
  - This loader loads the file into memory at the location specified by the beginning portion of the file called header. Then the control is passed to the program.
  - It does not perform relocation and linking.
- Relocating loader:
  - It relocates the program in memory.
  - For calculating all the relative offsets, a delay is produced.

Locator

- A LOCATOR is a program used to assign the specific addresses of where the segments of object code are to be loaded into memory.
- E.g: A Locator program called EXE2BIN provided by IBM PC Disk operating system converts a .EXE file to a .BIN file which has physical addresses.

Debugger

- A debugger is a program which allows us to load our object code program in system memory, execute it and debug it.
- Debugger also allows us to stop execution after each instruction so we can check and alter memory and register contents.
- A debugger allows us to set a breakpoint at any point in our program.

Emulator

- An emulator is a combination of hardware and software.
- It is usually used to test and debug the hardware and software of an external system.
- The hardware part of the emulator consists of multi-wire cable which connects the host system to the system being developed.
- This connection is used to download object code program into RAM in the system being tested and run it with the help of software of emulator.
- Emulator also allows us to debug the program.
- It stores the trace data of the execution for future reference.

ASSEMBLER DIRECTIVES

ASSEMBLER DIRECTIVES

- Definition:-Assembler Directives are the statements which help us to control the manner in which a source program assembles and lists.
- Specialty:-This statements are effective only during the assembly of a program but they do not generate any code that is machine executable.

.CODE

- Format: .CODE [name]
- This assembler directive indicates the beginning of the code segment.
- The name in this format is optional.
- In NASM→

$$\left.\begin{array}{l}\textbf{Section .text}\\ \textbf{global \_start}\\ \textbf{\_start:}\end{array}\right\}\ \textbf{Declaration of text section}$$

.STACK

- Format: .STACK[size]
- This directive is used for defining the stack.
- By default the size is 1024 bytes.
- E.g. .STACK[98];reserves 98 bytes for the stack

Initializing the STACK

Method 1:

```
ASSUME CS:CODE,DS:DATA,SS:STACK
………
……
STACK SEGMENT
S-DATA DB 100 DUP(?)
STACK ENDS
```

Method 2:

Syntax: .STACK [SIZE]

Example: .STACK 200

.DATA

- Format: .DATA
- This directive indicates the beginning of the data segment.
- In NASM→

  The syntax for declaring data section is –

  section .data

Define Byte [DB]

- Format: [name] DB initial value

  $\hookrightarrow$ Numeric value or (?)

- This directive defines the byte type variable.
- The initial value can be a signed or unsigned number.
- Range:

  i)-128 to +127(for signed)

  ii)0 to 255(for unsigned)

- E.g. List DB 08H

Define Word or Word [DW]

- Format: [name] DW initial value
- This directive defines items that are one word(2 bytes) in length.
- Range:

  i)0 to 65,535;unsigned numbers

  ii)-32,768 to +32767;signed numbers
- E.g. List DW 2534H

Define Double Word [DD]

- Format: [name] DD initial value
- It defines data items that are a double word(4 bytes) in length.
- It creates storage for 32-bit data.
- E.g. List DD ?

Define Quad Word [DQ]

- Format: [name] DQ initial value, [initial value]
- This directive is used to tell the assembler to declare the variable of 4 words in length or to reserve 4 words of storage in memory.
- It may define one or more constants, each with a maximum of 8 bytes or 16 Hex digits.
- E.g. List DQ 1234567898765432H

Define Ten Bytes [DT]

- Format: [name] DT initial value ,[initial value]
- It is used to define the data items that are 10 bytes long.
- Unlike the other data directives which stores the hexadecimal numbers, DT will directly store the data in decimal form.
- E.g. List DT 1234567890

DUP Operator

- Format: [name] Data-Type Number DUP (value)
- Whenever we want to allocate space for a table or an array the DUP directive can be used.
- DUP operator will be used after a storage allocation directive like (DB,DW,DQ,DT,DD).
- With DUP, we can repeat one or more values while assigning the storage values.
- E.g. List DB 20 DUP(0)

  A list of 20 bytes, where each byte is 0.
- A DUP operator can be nested.

EQU-Equate

- Format: [name] EQU initial value
- It is used to give a name to some value or symbol in the program.
- Each time when the assembler finds that name in the program, it replaces that name with value assigned to that ‘r’ variable.
- E.g. FACTORIAL EQU 05H
- This statement is written during the beginning of the program and whenever now FACTORIAL appears in any instruction or another directive, the assembler substitute the value for it.

The %assign Directive

- The %assign directive can be used to define numeric constants like the EQU directive.
- This directive allows redefinition. For example, you may define the constant TOTAL as:

```
%assign TOTAL 10
```

- Later in the code you can redefine it as:

```
%assign TOTAL 20
```

- This directive is case-sensitive.

The %define Directive

- The %define directive allows defining both numeric and string constants.
- This directive is similar to the #define in C.
- For example, you may define the constant PTR as:

```
%define PTR [EBP+4]
```

- The above code replaces PTR by [EBP+4].
- This directive also allows redefinition and it is case sensitive.

EXTRN

- It indicates that the names or labels that follow the EXTRN directive are in some other assembly module.

  e.g. : EXTRN DISP: FAR

- To call a procedure that is in a program module assembled at a different time from that which contains the CALL instruction. The assembler has to be told the procedure is external.
- The assembler will then put information in object code file so that linker can connect the two modules.

PUBLIC

- It informs the assembler & linker that the identified variables in a program are to be referenced by another modules linked with the current one.
- Format: PUBLIC variable,[variable]
- The variable can be a no.(up to 2 bytes) or a lable or a symbol.
- E.g:

PUBLIC MULTIPLIER,DIVISOR.

; This makes the 2 variables Multiplier & Divisor available to other assembly modules.

GLOBAL-Declare symbols as PUBLIC or EXTERN

- This directive can be used instead of PUBLIC directive or instead EXTERN directive.
- The global directive is used to make the variable available to all other modules.
- e.g.

```
GLOBAL FACTOR  ;It makes  the variable factor public so that it can be
                accessed from all other  modules that are linked.
```

Length

L

- It is an operator.
- It informs assembler to find the number of elements in a named data item like a string or an array.
- The length of string is always stored in Hex by the 8086.
- Its format is :

e.g. MOV CX,LENGTH STRING ; Loads the Length of string in CX.

OFFSET

- It is an operator.
- It informs the assembler to determine the offset or displacement of a named data item.
- It may also determined the offset of a procedure from the start of the segment which contains it.
- Its format is :

  e.g. MOV AX,OFFSET NUM ; It will load the offset of num in the AX register.

ORG-Originate

- The ORG directive allows us to set the location counter to any desired value at any point in the program.
- The location counter is automatically set to 0000H when the assembler reads a segment.
- Its format is. ORG expression

  e.g. ORG 500 H ; Set the location counter to 500H
- A $ is used to represent the current value of LC
- The $ represent the next available byte location where assembler can put a data or code byte.
- The $ often used in ORG statements to inform the assembler to make change in location counter relative to its current value.

  e.g. ORG $+50 ; Increments the location counter by 50 from its current.

PROC-Procedure

- Used to indicate the start of the procedure.
- The procedure is called from another function by using the CALL instruction. The CALL instruction should have the name of the called procedure as argument as shown below:
- CALL proc_name
- 2 types: FAR & NEAR
- Its format is:

[Procedure-name]

PROC NEAR

ENDP-End Procedure.

- It is used to indicate the end of the Procedure.
- E.g:

```
FACT PROC NEAR
;Statements inside the procedure.
FACT ENDP
```

Accessing a Procedure and Data from another Assembly module

File1.asm

```
EXTRN
   ROUTINE:FAR
```

File2.asm

```
PUBLIC ROUTINE PROC
   FAR
  |
  |
ROUTINE ENDP
```

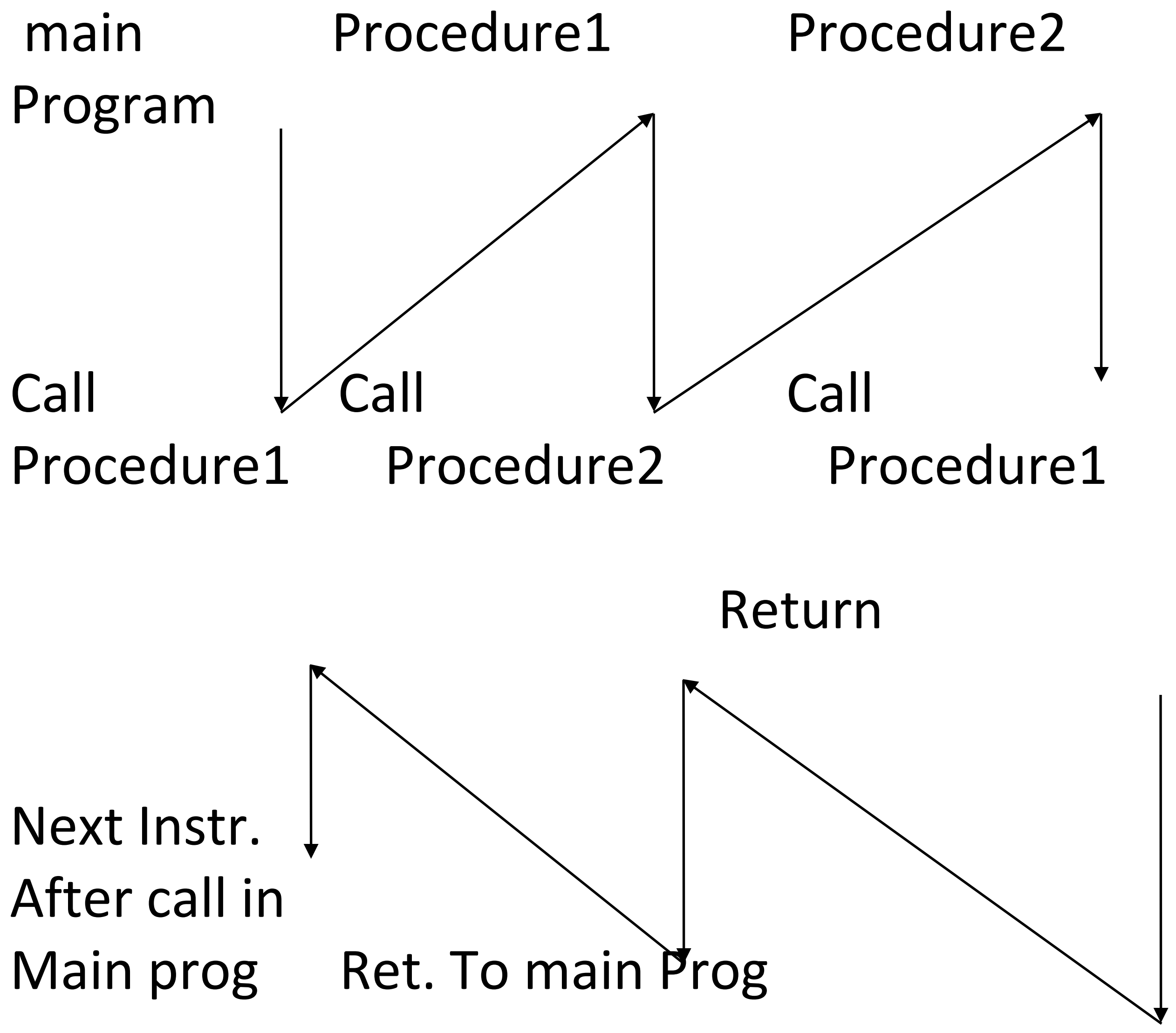
```
.WHILE CL!=3
 MOV AH,01H
 INT 21H
LEA BX,PASS
MOV AH,[BX+DI]
.IF AL==AH
   ADD DL,01
.ELSE
.BREAK
.END IF
INC DL
INC CL
.ENDW
```

Procedures & Macro

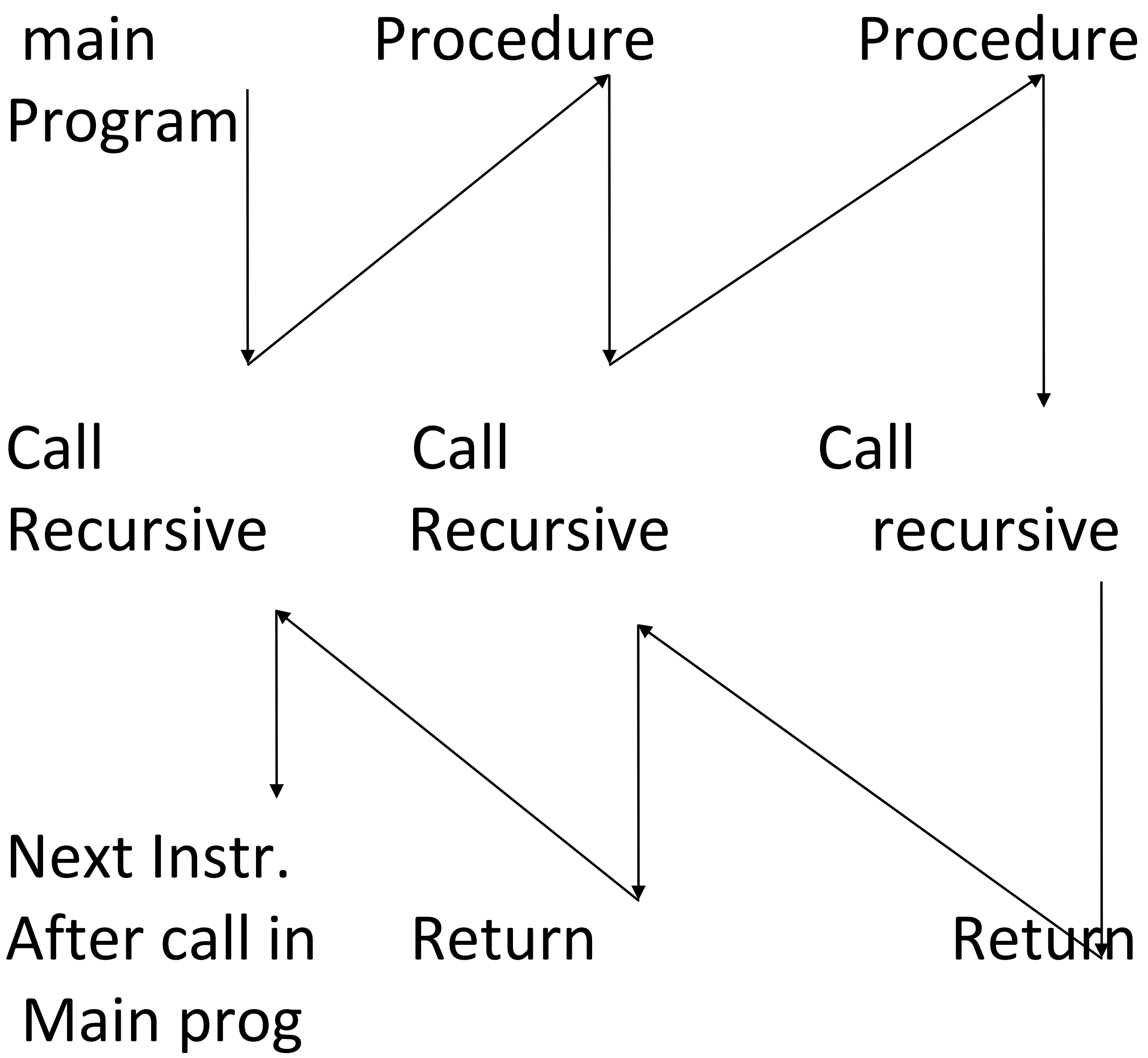
- Whenever we need to use a group of instructions several times throughout a program, there are 2 ways to avoid writing the set of instructions again & again –

  Procedure & Macro.
- We use **CALL** instruction to execute the procedure which directs microprocessor to the start of the procedure.
- A **RET** instruction at the end of the procedure returns execution to the next instruction in the main program.
- Besides transferring the execution control to the procedure, the **CALL** instruction also saves the return address on the **stack**.
- The procedures are of 2 types – **Near** & **Far**

Reentrant Procedure



Recursive Procedure



Difference between Near and Far Procedure

Sr. No.	Near Procedure	Far Procedure
1	It is declared in the same code segment .	It is defined in other code segment .
2	In this procedure contents of IP get changed.	While calling far procedure both CS and IP get changed.
3	In near procedure call less stack memory is required.	In far procedure call more stack memory is required.
4	It is called intra segment call.	It is called inter segment call.
5	Less time is required.	More time is required
6	Near directive is used.	Far ,Extrn ,Public directives are used.

MACRO - Introduction

- Macros are predefined functions which involve a group of instructions to perform a special task which can be used repeatedly.

For example:

- In order to print a string to the screen INT 80H together with 4 more instructions can be used (5 lines of code).
- It doesn’t make sense to rewrite them every time they are needed.
- In order to reduce the time to write the code and reduce the length of the code macros can be used.
- Macros allow programmer to define the task (set of codes to perform a specific job) once only and invoke it whenever/wherever it is needed.

MACROS

- To simplify and reduce the amount of repetitive coding
- To reduce the errors caused by repetitive coding
- To make the ALP more readable.
- Macro executes faster because there is no need of CALL and Return.
- In NASM, macros are defined with %macro and %endmacro directives.
- The basic format is:

```
    %Macro Name Macro     ;define macro
                         ; body of macro
     %Endmacro               ;End macro
```

MACRO Definition

- **%MACRO** name **Total no. of dummy arguments**

  …

  %**ENDMACRO**

- Ex: Write a macro called STRING to display a string of text to the monitor.

```
%MACRO STRING 2
MOV EAX,4
MOV EBX,1
MOV ECX, %1
MOV EDX,%2
INT 80H
%ENDMACRO
```

Invoking Macro

- You can invoke the above macro as follows:

; from the data segment

- MESSAGE1 DB ‘What is your name?’
- MESSAGE1LEN equ $-MESSAGE1

  :

  :

- ;from the code segment

  :

- STRING MESSAGE1, MESSAGE1LEN ; Assembler will invoke the macro to perform the defined function

Comparison between MACRO and PROCEDURE

Procedure	Macro
Procedure accessed by call and ret instructions	Macro called by macro name.
Memory required does not increases with more calls to procedure.	Memory required increases with more call to macro due to macro expansion.
Procedure requires latency period as it transfers control to a another location.	Macro does not require any latency period.
Procedure used for large amount of repetitive task.	Macro used for small amount of repetitive task
Time taken to execute program with procedure is more.	Time taken to execute program with macro is less.

Working with ALP in NASM

NASM - The Netwide Assembler

- An 80x86 and x86-64 assembler designed for portability and modularity
- Supports a range of object file formats - a.out, ELF, COFF, OBJ, WIN32, WIN64, etc.
- Default format - binary can read, combine and write object files in many different formats such as COFF, ELF, etc
- Different formats may be linked together to produce any available kind of object file.
- Elf – executable & linkable file format of object file & executable file – supported by Linux

NASM Commands

Steps to follow -

1. Boot the machine with ubuntu
2. Select and click on **<dash home>** icon from the toolbar.
3. Start typing “terminal” . Different terminal windows available will be displayed.
4. Click on “terminal” icon. A terminal window will open showing command prompt.

- To Type/Edit the source program

***gedit* hello.asm**

Type in the program in gedit window, save & exit

- To assemble

***nasm* –f elf64 hello.asm -l hello.lst**

This command creates an object file hello.o with elf64 file format & list file hello.lst

- To link

***ld* hello.o -o hello**

This command creates an executable file hello from hello.o

- To execute -

./hello

This command executes hello program

- To Debug using GNU debugger

***gdb* hello**

This command enters into GNU debugger window & programmer gets command prompt to enter required subcommand

GNU Debugger Command Summary

- **run** start program
- **quit** quit out of gdb
- **cont** continue execution after a break
- **break [addr]** ***break *_start+5*** sets a breakpoint
- **delete [n]** ***delete 4*** removes nth breakpoint
- **delete** removes all breakpoints
- **info break** lists all breakpoints
- **stepi** execute next instruction
- **stepi [n]** ***stepi 4*** execute next n instructions
- **nexti** execute next instruction, stepping over function calls

GNU Debugger Command Summary

- **nexti [n]** ***nexti 4*** execute next n instructions, stepping over function calls
- **where** show where execution halted
- **disas [addr]** ***disas _start*** disassemble instructions at given address
- **info registers** dump contents of all registers
- **print/d [expr]** ***print/d $ecx*** print expression in decimal
- **print/x [expr]** ***print/x $ecx*** print expression in hex
- **print/t [expr]** ***print/t $ecx*** print expression in binary
- **x/NFU [addr] x/12xw &msg** Examine contents of memory in given format
- **display [expr]** ***display $eax*** automatically print the expression each time the program is halted
- **info display** show list of automatically displays
- **undisplay [n]** ***undisplay 1*** remove an automatic display
- **set disassembly-flavor intel** You can configure gdb to always use Intel-style syntax

Assembly System Calls

System calls are APIs for the interface between user space and kernel space. You can make use of Linux system calls in your assembly programs. You need to take the following steps for using Linux system calls in your program −

Linux System Calls (32 bit)

- You can make use of Linux system calls in your assembly programs. You need to take the following steps for using Linux system calls in your program:
- Put the system call number in the EAX register.
- Store the arguments to the system call in the registers EBX, ECX, etc.
- Call the relevant interrupt (80h)
- The result is usually returned in the EAX register

There are six registers that stores the arguments of the system call used. These are the EBX, ECX, EDX, ESI, EDI, and EBP. These registers take the consecutive arguments, starting with the EBX register. If there are more than six arguments then the memory location of the first argument is stored in the EBX register.

Use of Linux System Functions

- **The following code snippet shows the use of the system call sys_exit:**

```
MOV EAX, 1          ; system call number (sys_exit)
MOV EBX,0              ; Return code
INT 80h             ; call kernel
```

- **The following code snippet shows the use of the system call sys_write:**

```
MOV EAX,4       ; system call number (sys_write)
MOV EBX,1         ; file descriptor (stdout)
MOV ECX, MSG   ; message to write
MOV EDX, 4        ; message length
INT 80h         ; call kernel
```

- **The following code snippet shows the use of the system call sys_Read:**

```
MOV EAX,3       ; system call number (sys_read)
MOV EBX,0         ; file descriptor (stdinput)
MOV ECX, buffer   ; buffer
MOV EDX, 4        ; message length including enter
INT 80h          ; call kernel
```

Linux System Calls (64 bit)

Linux system calls are called in exactly the same way as DOS system calls:
1. write the system call number in RAX
2. set up the arguments to the system call in RDI, RSI, RDX, etc.
3. make a system call with "SYSCALL" instruction
4. The result is usually returned in RAX

Sys_write:

- MOV RAX,1
- MOV RDI,1
- MOV RSI,message
- MOV RDX,msg_length
- SYSCALL

Linux System Calls (64 bit)

Sys_read:

- MOV RAX,0
- MOV RDI,0
- MOV RSI,array_name
- MOV RDX,array_size
- SYSCALL

Sys_exit:

- MOV RAX,60
- MOV RDI,0
- SYSCALL

Installing NASM

If you select "Development Tools" while installing Linux, you may get NASM installed along with the Linux operating system and you do not need to download and install it separately. For checking whether you already have NASM installed, take the following steps –

- Open a Linux terminal.
- Type **whereis nasm** and press ENTER.
- If it is already installed, then a line like, *nasm: /usr/bin/nasm* appears. Otherwise, you will see just *nasm:*, then you need to install NASM.

To install NASM, take the following steps –

- Check The netwide assembler (NASM) website for the latest version.
- Download the Linux source archive nasm-X.XX.ta.gz, where X.XX is the NASM version number in the archive.
- Unpack the archive into a directory which creates a subdirectory nasm-X. XX.
- cd to nasm-X.XX and type **./configure**. This shell script will find the best C compiler to use and set up Makefiles accordingly.
- Type **make** to build the nasm and ndisasm binaries.
- Type **make install** to install nasm and ndisasm in /usr/local/bin and to install the man pages.
- This should install NASM on your system. Alternatively, you can use an RPM distribution for the Fedora Linux. This version is simpler to install, just double-click the RPM file.