study Assignment

Title: Introduction to Assembly Language programming

Aim: To study assembler, linker, masm, tasm and assembly language programming x86 instructions set

objective:

To be familiar with the format of assembly language program structure and instructions.

Theory:

Assembly Language:

An assembly language is a low-level programming language for computers, microprocessors, microcontrollers, and other integrated circuits. It implements a symbolic representation of the binary machine codes and other constants needed to program a given CPU architecture. This representation is usually defined by the hardware manufacturer, and is based on mnemonics that symbolize processing steps (instructions), processor registers, memory locations, and other

language features. An assembly language is thus specific to certain physical (or virtual) computer architecture.

Assembler:

It is a system program which converts the assembly language program instructions into machine executable instructions. For example: Microsoft macro Assembler (MASM), Borland Turbo Assembler (TASM), open source NetWide Assembler (NASM) etc.

MASM

The Microsoft macro Assembler (MASM) is an x86 assembler for MS-DOS and Microsoft Windows. It supports a wide variety of macro facilities and structured programming idioms, including high-level functions for looping and procedures. Later versions added the capability of producing programs for Windows.

TASM

Turbo Assembler (TASM) is an x86 assembler package developed by Borland. It is used with Borland's high-level language compilers, such as Turbo Pascal, Turbo Basic and Turbo C. The

Turbo Assembler package is bundled with the linker, Turbo Linker, and is interoperable with the Turbo Debugger.

NASM

The Netwide Assembler (NASM) is an assembler and disassembler for the Intel x86 architecture.

It can be used to write 16-bit, 32-bit (IA-32) and 64-bit (x86-64) programs. NASM is considered to be one of the most popular assemblers for Linux and is the second most popular assembler overall. behind MASM

NASM was originally written by Simon Tatham with assistance from Julian Hall, and is currently maintained by a small team led by H. Peter Anvin.

Linker

Linker or link editor is a program that takes one or more objects generated by a compiler and combines them into a single executable program. When a program comprises multiple object files, the linker combines these files into a unified executable program. Linkers can take objects

from a collection called a library. some linkers do not include the whole library in the output; they only include its symbols that are referenced from other object files or libraries.

Loader

A loader is the part of an operating system that is responsible for loading programs, one of the essential stages in the process of starting a program, it means loader is a program that places programs into memory and prepares them for execution. Loading a program involves reading the contents of executable file, the file containing the program text, into memory, and then carrying
out other required preparatory tasks to prepare the executable for running. once loading is complete, the operating system starts the program by passing control to the loaded program code.

DOS Debugger debug is a command in DOS, MS-DOS, OS/2 and Microsoft Windows (only x86 versions) which runs the program debug.exe (or DEBUG.COM in older versions of DOS).

Debug can act as an assembler,

disassembler, or hex dump program allowing users to interactively examine memory contents in assembly language, make changes, and selectively execute COM, EXE and other file types.

procedure to create and execute a simple assembly program on Ubuntu Fedora (Linux) using nasm

steps to follow –

1. Boot the machine with Ubuntu Fedora
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.
5. give the following command at the prompt to invoke the editor

gedit hello.asm

6. Type in the program in gedit window, save and exit
7. TO assemble the program write the command at the prompt as follows and press enter key

nasm -f elf32 hello.asm -o hello.o (for 32 bit)

nasm -f elf64 hello.asm -o hello.o (for 64 bit)

8. If the execution is error free, it implies hello.o Object file has been created.

9. TO link and create the executable give the command as

ld -o hello hello.o

gcc -o hell hello.o (if you are using c functions)

10. TO execute the program write at the prompt

. hello

11. hello world will be displayed at the prompt

The assembly program structure –

The assembly program can be divided into three sections:

The .data section This section is for "declaring initialized data", in other words defining "variables" that already contain stuff. However this data does not change at runtime so they're not really variables. The .data section is used for things like filenames and buffer sizes. and you can also define constants using the EQU instruction. Here you can use the DB, DW, DD, DQ and DT instructions. For example:

section .data

```
message db 'Hello World!' ; Declare message to
contain the bytes ;'Hello World!'
msglength equ $-msg ; Declare msglength to have the
buffersize dw 1024 ;Declare buffersize to be a word
with 1024 bytes
```

The bss section This section is where you declare your variables. You use the RESB, RESW, RESD, RESQ and REST instructions to reserve uninitialized space in memory for your variables, like this:

```
section .bss
filename resb 255 ; Reserve 255 bytes
number resb 1 ; Reserve 1 byte
bignum resw 1 ; Reserve 1 word (1 word == 2
bytes)
realarray resq 10 ; Reserve an array of 10 reals
```

The .text section This is where the actual assembly code is written. The .text section must begin

with the declaration global _start, which just tells the kernel where the program execution begins.

(It's like the main function in c or Java, only it's not a function, just a starting point.) Eg.:

```
section .text
```

```
global _start

_start:

; Here is the where the program actually
begins
```

Linux system calls (for 32 bit): (write system calls that you are using)

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:

1. put the system call number in the EAX register.
2. store the arguments to the system call in the registers EBX, ECX, etc.
3. call the relevant interrupt (80h)
4. The result is usually returned in the EAX register

The Linux system calls (SYSCALL for 64bit execution)

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