Proceedings of the 2nd International conference on Electronics, Communication and Aerospace Technology (ICECA 2018)
IEEE Conference Record # 42487; IEEE Xplore ISBN:978-1-5386-0965-1

A Petite Guide to Programming in 64bit X86 Assembly Language for Linux

Vidya Pokale[1], Madhuri Chavan[2]
Assistant Professor, Vishwakarma Institute of Information Technology, Pune, India
[1] vidya.jadhav@viit.ac.in
[2] madhuri.chavan@viit.ac.in

***Abstract*—** since the invention of the personal computers, programmers have used assembly language to create innovative solution for a wide variety of algorithmic challenges. This paper introduces a subset of X64, including NASM (Netwide Assembler), LD-GNU linker, Linux system calls.

Keywords—Registers, NASM, Linux System call, Assembly program, Macro and Procedures.

I. INTRODUCTION

Over 3 decades the Intel processor architecture has evolved from a 16 bit processor with no memory protection, through a period with 32 bit processor with sophisticated architect sure into current generation laptop & desktop computer which support all the old modes of operation in addition to the highest degree expanded 64 bit modes of operation. In this paper we introduce 64 bit X86 assembly language programming including a small but useful subset of available instruction, registers and Linux system call.

II. REGISTERS

Registers are used for temporary storage. Computations are typically performed by the CPU using registers [1][2][11].

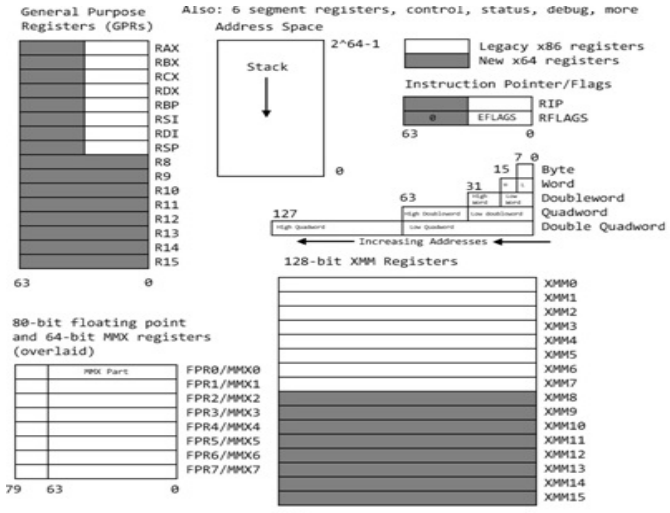

Fig. 1 Register Set

A. Basic Program Execution Registers: There are 16 general purpose register (GPRs) RAX, RBX, RCX, RDX. All these are 64 bits wide. The registers support byte, word, double word, quad word integer operation. There are six segment register of 16 bit each (CS, DS, ES, SS, GS, FS). The size of instruction pointer register is 64 bit called RIP. 64 bit RFLAGS register.

B. X87 Floating Point Registers: The eight x87 FPU data registers, the x87 FPU control registers, the status register, the x87 FPU instruction pointer register, the X87 FPU operand (data) pointer register, the x87 FPU tag register, and the x87 FPU opcode register provide an execution environment for operating on single precision, double precision and double extended-precision floating point values, word integer, double word integer and BCD values.

C. MMX registers: 8MMX registers that allows the SIMD instruction to operate on 64 bit packed bytes, word and double word integers.

D. XMM registers:16 XMM data registers of 128bit that support SIMD instruction operation on 128 bit packed single precision and double precision floating point value and 128 bit packed byte, word, double word and quad word integer.

E. Stack: basic execution environment allows the application program to support procedure calls or subroutine calls and parameter passing between procedures.

F. Control Registers: Responsible for computing the processor operating mode. It also determines the characteristics of the task that is currently being executed.

G. Memory Management registers: GDTR, LDTR, IDTR, Task register these registers are used in the protected mode for memory management and multitasking.

H. Debug Registers: DR0-DR7 can control and allow monitoring of the processor's debugging operation.

I. ***Memory Type Range Registers (MTRRs):*** These registers are used to assign memory types to register of memory.

J. Machine Specific Registers (MSRs): The processor provides a variety of machine specific registers that are used to control and report on processor performance.

III. INSTRUCTIONS

Instruction set generally divided into number of categories: 1) Data transfer 2) Arithmetic 3) Bit manipulation 4) String instruction 5) Program transfer 6) Process control [3][4][5]. SIMD (Single Instruction Multiple Data) use set of separate core register called MMX (MultiMedia eXtenstion) and XMM registers.

Overview of the SEE Instruction Set: According to Flynn's classification of parallel computers Single Instruction, Multiple Data (SIMD), is a class of parallel computers which describes computers with processing elements that perform the same operation on multiple data points on simultaneously. Thus, such machines exploit data level parallelism. There are six extension have been introduced into the Intel 64 and IA-32 architecture to perform SIMD operation [6][7]. We summarize the following main characteristics of the SEE instruction [10].

1) MX instructions perform SIMD operation on packed bytes, word or double word integers located in MMX registers. These instructions are useful in applications that operate on integer arrays and streams of integer data that lend themselves to SIMD processing. (Pentium II and Pentium with MMX technology processor families (Intel MMX technology)).

2) Streaming SIMD: SEE instructions operate on packed single precision floating point values contained in XMM registers and on packed integers contained in MMX registers. Provides state management, cache management and memory ordering operations. Some SEE instructions are targeted at applications that operate on array of single precision floating point data elements (3D geometry, 3D rendering and video encoding and decoding application Pentium III processor family).

3) Streaming SIMD Extension 2: SSE2 instructions operate on packed double-precision floating point values contained XMM registers and on packed integers contained in MMX and XMM registers. SSE2 integer instructions extend IA-32 SIMD operation by adding new 128 bit SIMD integer operation and by expanding existing 64-bit SIMD integer operation to 128-bit XMM capability.SSE2 instruction also provide new cache control and memory ordering operations. (Pentium 4 and Intel Xeon processor).

4) Streaming SIMD Extension 3: SSE3 offers 13 instruction that accelerate performance of streaming SIMD extensions technology, streaming SIMD extension 2 technology and x87 floating point math capabilities. (Intel Xeon processor 5100 series and Intel Core 2 processor families).

5) SSE4: SSE4 extensions offer 54 instructions (47 SSE4.1 instructions plus 7 SSE4.2instructions). SSE4 also improves compiler vectorization and significantly increase support for packed double word computation. (Intel Xeon processor 5400 series and Intel Core 2 extreme processor QX)

IV. WORKING ENVIRONMENT

Assembly language is dependent upon the instruction set and the architecture of processor. In this paper we focus on 64 bit processor. We need following environment,

1) Core 2duo/i3/i5/i7- 64 bit processor.
2) OS- ubuntu and fedora (Linux) 64 bit OS.
3) Assembler used- NASM (the netwide assembler NASM is portable, free and powerful).
4) Editor used- gedit a GNU editor.
5) Linker-LD a GNU linker.

NASM – The Netwide Assembler

- An 80x86 and x86-64 assembler defined for portability and modularity.
- Support a range of object file formats- a.out, ELF, COFF, OBJ, WIN32, etc.
- Default Format- binary.
- installation of NASM [8]

LD- GNU Linker

- 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 and linkable file format of object file and executable file, supported by Linux.

.

V. BASIC PROGRAM STRUCTURE

A. Assembly Language Statement

There are three types of statement in assembly language programming. Typically, one statement should appear on a line.

1) ***Executable Statement:*** generate machine code for the processor to execute at run time. These instructions tell processor what to do.

2) ***Pseudo Instruction and Macros:*** Those instructions which are translated by the assembler in to real instructions. It simplifies the programmer task.

3) ***Assembler Directives:*** Assembly directives are the statements that direct the assembler to do something. As the name says, it directs the assembler to do the task. The specialty of these statement is that they are effective only during the assembly of a program but they do not generate any code that is machine executable.

B. Assembly Language Instruction Format

The general format of an assembler instruction is as follow

Label: Mnemonics Operand Comment

- **Label (Optional)**
  - Marks the address of a memory location, must have colon.
  - Typically appear in data and text segments
- **Mnemonics**
  - Is an instruction name.
  - All the instruction should have a mnemonics.
  - Identifies the operation (e.g. add, sub, mov).
- **Operand**
  - Specify the data required by the operation.
  - Operands can be register, memory variable, or constant.
  - The type and number of operands depends entirely on the specific instruction.
  - Some instruction have no operands e.g. XLAT, AAM, DAA.
- **Comments**
  - Explain the program's purpose.
  - When it was written, revised, and by whom.
  - Explain data used in the program, input and output.
  - Allows you to annotate each line of source code in your program i.e. put a comment on each line of the program code so that it becomes simple to understand the program
  - Beginning of the notation ";".

C. Structure of assembly program

The structure of Assembly language program can be divided in to three sections: The data section, The bss section, The text section.

1) The .data section

This section is used 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
msg db 'Welcome'     ; Declare message to contain the bytes
msglen equ msg-$     ; Declare msglen  (Length of welcome)
Buffer dw 512        ; Declare buffer to be a word with 512 bytes
msglen equ msg-$ ; Declare msglen  (Length of welcome)
```

2) The .bss section (bss stands for block starting symbol)

This section is where you declare your variable. You use the RESB (Reserve Bytes), RESW, RESD, RESQ and REST instructions to reserve **uninitialized** space in memory for your variables.For Example

```
section .bss
        Number resb 3     ; Reserve 3 bytes
 Array resq 20     ; Reserve an array of 10 quad words
        Filename resb 255   ; Reserve 255 bytes
        num resw 1          ;Reserve 1 word
```

3) 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.) For Example:

```
section .text
global _start
_start:
        ; Here the program actually begins (Code)
```

Data Definition Statement

- Sets aside storage in memory for variable
- May optional assign a name to the data
- NASM present various directives for reserving storage space for variables
- The directive is used for allocation of storage space.

Storage Space: Initialized data (.data section)

Syntax

```
Variable name   directive   initializer      , [initializer]
   Var1           DB            10
```

Define Directives:

TABLE 1
DEFINE DIRECTIVES

Directives	Purpose	Storage Space	
DB	Define Byte	8 bits	1byte
DW	Define Word	16 bits	2 bytes
DD	Define Double word	32 bits	4 bytes
DQ	Define Quad Word	64 bits	8 bytes
DT	Define Ten Bytes	80 bits	10 bytes

Here,

- Every initializer is stored as its ASCII value in Hexadecimal.
- Every Decimal value converted to equivalent its 16 bit binary value and stored as a hexadecimal number.
- Negative numbers represented in 2's complement.

Storage Space: Uninitialized data (.bss section)

- Reserve directive used for allocating memory space for uninitialized data. The reserve directives take single operand that specifies the number of units of space to be reserved.

TABLE 2
RESERVE DIRECTIVES

Directives	Purpose
RESB	Reserve a Byte
RESW	Reserve a Word
RESD	Reserve a Double Word
RESQ	Reserve a Quad Word
REST	Reserve a Ten Bytes

VI LINUX SYSTEM CALL

The system calls are API for interface between an application and Linux kernel [9]. Linux system call are called in exactly the same way as DOS system call. The system call number for 64 bit defined in "user/include/asm/unistd_64.h". Write system call number in RAX and set up the parameters to the system call in RDI, RSI, RDX, etc. Instead of using software interrupt instruction, X86-64 Linux uses the "syscall" instruction to execute a system call. Return values usually are placed in RAX.

1) System Write Call (Write Character to Standard Output Device)

```
mov rax, 1          ; Function number for system write
mov rdi, 1          ; File descriptor ID for standard output
                                                    device
mov rsi, msg        ; Address of the variable to output
mov rdx, msglen     ; Count of byte to be display
syscall             ; Invoke the operating system to do the write
```

2) System Read Call (Accept Character from Standard Input Devices

```
mov rax, 0          ; Function number for system read
mov rdi,            ; File descriptor ID for standard input device
 mov rsi, arr       ; Address of the variable used to store data
mov rdx, 8          ; Maximum byte to read
 syscall            ; Invoke the operating system to do the read
```

3) System Exit call

```
 mov rax, 60            ; Function number for system exit
mov rdi, 0                      ; Return code for zero error
syscall                 ; Invoke the Operating System to exit
```

Example1: Display Hello World in Assembly!

```
section .data
        msg db 10,"Hello World!"    ;String to display
        msg_len equ $-msg           ;Length of "Hello World!"
section .bss
section .text
global _start
_start:
        mov rax,1           ; Function number for system write
        mov rdi,1           ; File descriptor ID for standard
                            output  device
         mov rsi, msg       ; Address of the variable to
                             output
        mov rdx, msg_len    ; Count of byte to be display
        syscall         ; Invoke the operating system to do write

        mov rax, 60         ; Function number for system exit
        mov rdi, 0          ; Return code for zero error
        syscall             ; Invoke operating system to exit
```

Assembly and Linking Program in NASM for Linux

1. Boot the machine with ubuntu or fedora.
2. Click on terminal icon. A terminal window will open showing command prompt.
3. Give following command at the prompt to invoke the editor.
   gedit hello.asm
4. Type the program in gedit window, save and exit
5. To assemble the program write the command at prompt as follow press enter key
   nasm –f elf64 hello.asm –o hello.o
   The –f command tells NASM which format to use for the object code file it's about to generate. In 64 bit Linux work, the format is ELF64, which can be specified on command line as simply elf.
6. If the execution is error free, it implies hello.o object file has been created.
7. To link and create the executable give the command as
   ld –o hello hello.o
8. To execute the program write at prompt
   ./hello
9. Hello World! Will be displayed on prompt.

VII MACRO AND PRODUCERS

Whenever we need to use group of instructions several times throughout a program there are two ways we can avoid having to write the group of instruction each time we want to use them. One way is to write the group of instructions as separate procedure. We can then just CALL the procedure whenever we need to execute the group of instructions. Another way is to use macro. The type of procedure depends on where the procedure is stored in the memory. If it is in the same code segment where the main program is stored then it is called near producer otherwise far procedure. When procedure is called return address is stored in stack. For near procedure CALL instruction pushes only IP on the stack, since CS register content remains unchanged for main program and procedure. But for FAR procedure CALL instruction pushes both IP and CS on the stack. Macro is group of instruction, assigned by the name and could be used in anywhere in program[9]. Macros are called by its name along with the necessary parameter. When we required to use some sequence of instructions many time in program we use macro instead of the writing the instruction all the time. Macros are defined with %macro (Start Macro) and %endmacro (End Macro) directives in NASM.

Syntax

```
%macro macro_name  no_of_parameters
        < macro body>
%endmacro
```

Where macro_ name specifies name of macro and no_of_parameters specifies number of parameters

Example

In program many times we display the messages or string characters. For displaying a string character, you need the following sequence of instruction.

```
mov rax,1
mov rdi,1
mov rsi,msg
mov rdx,msg_len
syscall
```

Following example shows defining and using macros

Example2: Accept and Display number (Using Macro)

Numerical data is generally represented in binary system. Arithmetic instructions operate on binary data. When numbers are accepted from keyboard and displayed on display screen, they are converted in to ASCII hex form. That's why we have to convert this input data to HEX for arithmetic calculation and again convert result to back to ASCII HEX for display.

Accept Number:

We can accept Hexadecimal number from user (i.e. 0 to 9 or A to F). So ASCII value for 0 to 9 is from 48 to 57 and A to F is 65 to 70. When we accept number from keyboard it is converted into ASCII and stored as a hexadecimal number.

Example:

Suppose we accept 0 from keyboard then it is converted in to equivalent 16 bit binary value (i.e. ASCII value for 0 is 48) and stored as a hexadecimal number i.e. 30(Hex number of 48). So when we accept 0 it stored as 30H in memory.

```
section .data

        msg db "Enter the Number",10
        msg_len equ $-msg
        msg1 db "Entered Number is",10
        msg1_len equ $-msg1

section .bss

        num resq 2

 %macro inout 4   ; Beginning of macro (inout name
                                        of macro)
     mov rax,%1      ;function no. for input(0) and
                      output(1)
     mov rdi,%2    ;Descriptor Id for input device(0)
                            and output Device (1)
     mov rsi,%3    ;Address of variable from   where
                          u want to display or store
     mov rdx,%4    ;Number of bytes to be read and
                                            write
     syscall
  %endmacro       ; end of the macro

section .text

global _start

_start:
     inout 1,1,msg,msg_len  ;call macro by using its
                name and pass respective parameters
     inout 0,0,num,17
     inout 1,1,msg1,msg1_len
     inout 1,1,num,17

     mov rax,60
     mov rdi,0
     syscall
```

REFERENCES

[1] x86-64 Assembly Language Programming with Ubuntu, Ed Jorgensen ,Version 1.0.79, January 2018.
[2] Modern X86 Assembly Language Programming 32-bit, 64Bit, SSE and AVX, Daniel Kusswurm.
[3] Assembly Language step by step programming with Linux, 3rd edition , Jeff Duntemann, Wiley Publication.
[4] A Tiny Guide to Programming in 32-bit x86 Assembly Language, Adam Ferrari & Mike Lack, Spring 1999.
[5] Intel® 64 and IA-32 Architectures, Software Developer's Manual Volume 3A:System Programming Guide, Part 1, May 2011, http://www.intel.com
[6] The Architecture of the Nehalem Processor and Nehalem-EP SMP Platforms, Michael E. Thomadakis, March, 17, 2011.,
[7] Assembly Language Tutorial, www.tutorialspoint.com
[8] Introduction to x64 Assembly, Martin Hirzel, November 15, 2011.
[9] Introduction to 64 Bit Intel Assembly Language Programming for Linux, Ray Seyfarth, October 27, 2011
[10] Professional Assembly Language, Richard Blum, Wiley Publishing, Inc
[11] https://software.intel.com/en-us/articles/introduction-to-x64-assembly
[12] x86-64 Machine-Level Programming Randal E. Bryant David & R. O'Hallaron, September 9, 2005