Module 2

Assembly – System Calls and Addressing Modes

Subtopic 1

System Calls and Addressing Modes

Objectives

At the end of the module the student is expected to:

- **Understand Linux system calls**
- **List the different addressing modes**
- **Apply the concepts of variables and constants in assembly programming.**

System calls are APIs for the interface between the user space and the kernel space. We have already used the system calls, sys_write and sys_exit, for writing into the screen and exiting from the program, respectively.

Linux System Calls

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.**

Linux System Calls

There are six registers that store 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.

Linux System Calls

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

```
mov   eax,1         ; system call number (sys_exit)
int   0x80          ; call kernel
```

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

```
mov   edx,4         ; message length
mov   ecx,msg       ; message to write
mov   ebx,1         ; file descriptor (stdout)
mov   eax,4         ; system call number (sys_write)
int   0x80          ; call kernel
```

Linux System Calls

All the syscalls are listed in /usr/include/asm/unistd.h, together with their numbers (the value to put in EAX before you call int 80h).

The following table shows some of the system calls used in this module:

%eax	Name	%ebx	%ecx	%edx	%esx	%edi
1	sys_exit	int	-	-	-	-
2	sys_fork	struct pt_regs	-	-	-	-
3	sys_read	unsigned int	char *	size_t	-	-
4	sys_write	unsigned int	const char *	size_t	-	-
5	sys_open	const char *	int	int	-	-
6	sys_close	unsigned int	-	-	-	-

Example

The following example reads a number from the keyboard and displays it on the screen:

```
section  .data ;Data segment
    userMsg db 'Please enter a number: ' ;Ask the user to enter a number
    lenUserMsg equ $-userMsg             ;The length of the message
    dispMsg db 'You have entered: '
    lenDispMsg equ $-dispMsg

section .bss            ;Uninitialized data
    num resb 5
section .text           ;Code Segment
       global _start
_start:
       ;User prompt
       mov eax, 4
       mov ebx, 1
       mov ecx, userMsg
       mov edx, lenUserMsg
       int 80h

       ;Read and store the user input
       mov eax, 3
       mov ebx, 2
       mov ecx, num
       mov edx, 5       ;5 bytes (numeric, 1 for sign) of that information
       int 80h
       ;Output the message 'The entered number is: '
       mov eax, 4
       mov ebx, 1
       mov ecx, dispMsg
       mov edx, lenDispMsg
       int 80h

       ;Output the number entered
       mov eax, 4
       mov ebx, 1
       mov ecx, num
       mov edx, 5
       int 80h
; Exit code
       mov eax, 1
       mov ebx, 0
       int 80h
```

When the above code is compiled and executed, it produces the following result:

```
Please enter a number:
1234
You have entered:1234
```

Addressing Modes

Most assembly language instructions require operands to be processed. An operand address provides the location, where the data to be processed is stored. Some instructions do not require an operand, whereas some other instructions may require one, two, or three operands.

Addressing Modes

When an instruction requires two operands, the first operand is generally the destination, which contains data in a register or memory location and the second operand is the source. Source contains either the data to be delivered (immediate addressing) or the address (in register or memory) of the data. Generally, the source data remains unaltered after the operation.

The three basic modes of addressing are:

- **Register addressing**
- **Immediate addressing**
- **Memory addressing**

Register Addressing

In this addressing mode, a register contains the operand. Depending upon the instruction, the register may be the first operand, the second operand or both.

For Example:

```
MOV DX, TAX_RATE    ; Register in first operand
MOV COUNT, CX        ; Register in second operand
MOV EAX, EBX         ; Both the operands are in registers
```

As processing data between registers does not involve memory, it provides fastest processing of data.

Immediate Addressing

An immediate operand has a constant value or an expression. When an instruction with two operands uses immediate addressing, the first operand may be a register or memory location, and the second operand is an immediate constant. The first operand defines the length of the data.

For Example:

```
BYTE_VALUE  DB  150     ; A byte value is defined
WORD_VALUE  DW  300     ; A word value is defined
ADD  BYTE_VALUE, 65     ; An immediate operand 65 is added
MOV  AX, 45H            ; Immediate constant 45H is transferred to AX
```

Direct Memory Addressing

When operands are specified in memory addressing mode, direct access to main memory, usually to the data segment, is required. This way of addressing results in slower processing of data. To locate the exact location of data in memory, we need the segment start address, which is typically found in the DS register and an offset value. This offset value is also called effective address.

In direct addressing mode, the offset value is specified directly as part of the instruction, usually indicated by the variable name. The assembler calculates the offset value and maintains a symbol table, which stores the offset values of all the variables used in the program.

Direct Memory Addressing

In direct memory addressing, one of the operands refers to a memory location and the other operand references a register.

For example,

```
ADD   BYTE_VALUE, DL      ; Adds the register in the memory location
MOV   BX, WORD_VALUE      ; Operand from the memory is added to register
```

Direct-Offset Addressing

This addressing mode uses the arithmetic operators to modify an address. For example, look at the following definitions that define tables of data:

```
BYTE_TABLE DB  14, 15, 22, 45      ; Tables of bytes
WORD_TABLE DW  134, 345, 564, 123  ; Tables of words
```

The following operations access data from the tables in the memory into registers:

```
MOV CL, BYTE_TABLE[2]    ; Gets the 3rd element of the BYTE_TABLE
MOV CL, BYTE_TABLE + 2   ; Gets the 3rd element of the BYTE_TABLE
MOV CX, WORD_TABLE[3]    ; Gets the 4th element of the WORD_TABLE
MOV CX, WORD_TABLE + 3   ; Gets the 4th element of the WORD_TABLE
```

Indirect Memory Addressing

This addressing mode utilizes the computer's ability of Segment:Offset addressing. Generally, the base registers EBX, EBP (or BX, BP) and the index registers (DI, SI), coded within square brackets for memory references, are used for this purpose.

Indirect addressing is generally used for variables containing several elements like, arrays. Starting address of the array is stored in, say, the EBX register.

The following code snippet shows how to access different elements of the variable.

```
MY_TABLE TIMES 10 DW 0  ; Allocates 10 words (2 bytes) each initialized to 0
MOV EBX, [MY_TABLE]     ; Effective Address of MY_TABLE in EBX
MOV [EBX], 110          ; MY_TABLE[0] = 110
ADD EBX, 2              ; EBX = EBX +2
MOV [EBX], 123          ; MY_TABLE[1] = 123
```

The MOV Instruction

We have already used the MOV instruction that is used for moving data from one storage space to another. The MOV instruction takes two operands.

The syntax of the MOV instruction is:

```
MOV  destination, source
```

The MOV instruction may have one of the following five forms:

```
MOV  register, register
MOV  register, immediate
MOV  memory, immediate
MOV  register, memory
MOV  memory, register
```

The MOV Instruction

Please note that:

- **Both the operands in MOV operation should be of same size**
- **The value of source operand remains unchanged**
- **The MOV instruction causes ambiguity at times. For example, look at the statements:**

```
MOV  EBX, [MY_TABLE]    ; Effective Address of MY_TABLE in EBX
MOV  [EBX], 110          ; MY_TABLE[0] = 110
```

It is not clear whether you want to move a byte equivalent or word equivalent of the number 110. In such cases, it is wise to use a type specifier.

The MOV Instruction

The following table shows some of the common type specifiers:

Type Specifier	Bytes addressed
BYTE	1
WORD	2
DWORD	4
QWORD	8
TBYTE	10

The following program stores a name 'Zara Ali' in the data section of the memory, then changes its value to another name 'Nuha Ali' programmatically and displays both the names.

```
section     .text
    global  _start    ;must be declared for linker (ld)
_start:    ;tell linker entry point

;writing the name 'Zara Ali'
     mov    edx,9       ;message length
     mov    ecx, name   ;message to write
     mov    ebx,1       ;file descriptor (stdout)
     mov    eax,4       ;system call number (sys_write)
     int    0x80        ;call kernel

     mov    [name],  dword 'Nuha'    ; Changed the name to Nuha Ali
;writing the name 'Nuha Ali'
     mov    edx,8       ;message length
     mov    ecx,name    ;message to write
     mov    ebx,1       ;file descriptor (stdout)
     mov    eax,4       ;system call number (sys_write)
     int    0x80        ;call kernel
     mov    eax,1       ;system call number (sys_exit)
     int    0x80        ;call kernel

section     .data
name db 'Zara Ali '
```

When the above code is compiled and executed, it produces the following result:

```
Zara Ali Nuha Ali
```

Allocating Storage Space for Initialized Data

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.

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.

Allocating Storage Space for Initialized Data

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

Allocating Storage Space for Initialized Data

Following are some examples of using define directives

```
choice          DB      'y'
number          DW      12345
neg_number      DW      -12345
big_number      DQ      123456789
real_number1    DD      1.234
real_number2    DQ      123.456
```

Please note that:

- **Each byte of character is stored as its ASCII value in hexadecimal.**
- **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 Initialized Data

The following program shows the use of define directive:

```
section .text
    global _start    ;must be declared for linker (gcc)
_start:    ;tell linker entry point

     mov    edx,1         ;message length
     mov    ecx,choice    ;message to write
     mov    ebx,1         ;file descriptor (stdout)
     mov    eax,4         ;system call number (sys_write)
     int    0x80          ;call kernel

     mov    eax,1         ;system call number (sys_exit)
     int    0x80          ;call kernel

section .data
choice DB 'y'
```

When the above code is compiled and executed, it produces the following result:

```
y
```

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
RESW	Reserve a Word
RESD	Reserve a Doubleword
RESQ	Reserve a Quadword
REST	Reserve a Ten Bytes

Multiple Definitions

You can have multiple data definition statements in a program.

For example:

```
choice      DB   'Y'          ;ASCII of y = 79H
number1     DW   12345  ;12345D = 3039H
number2   DD     12345679     ;123456789D = 75BCD15H
```

The assembler allocates contiguous memory for multiple variable definitions.

Multiple Initializations

The TIMES directive allows multiple initializations to the same value. For example, an array named marks of size 9 can be defined and initialized to zero using the following statement:

```
marks TIMES 9 DW 0
```

The TIMES directive is useful in defining arrays and tables.

Multiple Initializations

The following program displays 9 asterisks on the screen:

```
section     .text
    global _start    ;must be declared for linker (ld)
_start:    ;tell linker entry point
     mov    edx,9        ;message length
     mov    ecx, stars   ;message to write
     mov    ebx,1        ;file descriptor (stdout)
     mov    eax,4        ;system call number (sys_write)
     int    0x80         ;call kernel

     mov    eax,1        ;system call number (sys_exit)
     int    0x80         ;call kernel

section     .data
stars   times 9 db '*'
```

When the above code is compiled and executed, it produces the following result:

```
*********
```

Constants

There are several directives provided by NASM that define constants. We have already used the EQU directive in previous chapters.

We will particularly discuss three directives:

- **EQU**
- **%assign**
- **%define**

The EQU Directive

The EQU directive is used for defining constants. The syntax of the EQU directive is as follows:

```
CONSTANT_NAME EQU expression
```

For example,

```
TOTAL_STUDENTS equ 50
```

You can then use this constant value in your code, like:

```
mov  ecx,  TOTAL_STUDENTS
cmp  eax,  TOTAL_STUDENTS
```

The operand of an EQU statement can be an expression:

```
LENGTH equ 20
WIDTH  equ 10
AREA   equ length * width
```

Above code segment would define AREA as 200.

The following example illustrates the use of the EQU directive:

```
SYS_EXIT  equ 1
SYS_WRITE equ 4
STDIN     equ 0
STDOUT    equ 1
section      .text
   global _start    ;must be declared for using gcc
_start:   ;tell linker entry point
     mov eax, SYS_WRITE
     mov ebx, STDOUT
     mov ecx, msg1
     mov edx, len1
     int 0x80

     mov eax, SYS_WRITE
     mov ebx, STDOUT
     mov ecx, msg2
     mov edx, len2
     int 0x80

     mov eax, SYS_WRITE
     mov ebx, STDOUT
     mov ecx, msg3
     mov edx, len3
     int 0x80
        mov eax,SYS_EXIT    ;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 'Linux assembly programming! '
len3 equ $- msg3
```

When the above code is compiled and executed, it produces the following result:

```
Hello, programmers!
Welcome to the world of,
Linux assembly programming!
```

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.