

**BHARATI VIDYPEETH’S COLLEGE OF ENGINEERING, LAVALE, PUNE**



**DEPARTMENT OF COMPUTER ENGINEERING**

**Savitribai Phule Pune University (SPPU) Second Year of Computer Engineering (2019 Course)**

**Laboratory Manual For**

**A.Y. 2024-2025**

**210257: Microprocessor Laboratory Prof. Ashwini Bhapkar**

**Term Work: 25 Marks Oral: 25 Marks**

**Guidelines for Laboratory /Term Work Assessment**

Continuous assessment of laboratory work is based on overall performance and Laboratory assignments performance of student. Each Laboratory assignment assessment will assign grade/marks based on parameters with appropriate weightage. Suggested parameters for overall assessment as well as each Laboratory assignment assessment include- timely completion, performance, innovation, efficient codes, punctuality and neatness.

**Guidelines for Laboratory Conduction**

The instructor is expected to frame the assignments by understanding the prerequisites, technological aspects, utility and recent trends related to the topic. The assignment framing policy need to address the average students and inclusive of an element to attract and promote the intelligent students. The instructor may set multiple sets of assignments and distribute among batches of students. It is appreciated if the assignments are based on real world problems/applications. Use of open source software is encouraged.

In addition to these, instructor may assign one real life application in the form of a mini-project based on the concepts learned. Instructor may also set one assignment or mini-project that is suitable to respective branch beyond the scope of syllabus.

Operating System: 64-bit Open source Linux or its derivative.

Programming Tools: Preferably using Linux equivalent or MASM/TASM/NASM/FASM.

**INDEX**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Sr.No**  **.** | **Date** | **Experiment Performed** | **Page No** | **Sign** | **Remark** |
| **1** |  | Write an X86/64 ALP to accept five 64 bit Hexadecimal  numbers from the user and store them in an array and display the accepted numbers. |  |  |  |
| **2** |  | Write an X86/64 ALP to accept a string and to display its length. |  |  |  |
| **3** |  | Write an X86/64 ALP to find the largest of given Byte/Word/Dword/64-bit numbers. |  |  |  |
| **4** |  | Write an X86/64 ALP to count the number of positive and negative numbers from the array. |  |  |  |
| **5** |  | Write X86/64 ALP to convert 4-digit Hex number into its equivalent BCD number and 5-digit BCD number into its equivalent HEX number. Make your program user friendly to accept the choice from user for: (a) HEX to BCD b) BCD to HEX (c) EXIT.Display proper strings to prompt the user while accepting the input and displaying the result. (Wherever necessary, use 64-bit registers). |  |  |  |
| **6** |  | Write X86/64 ALP to detect protected mode and display the values of GDTR, LDTR, IDTR,TR and MSW Registers also identify CPU type using CPUID instruction. |  |  |  |
| **7** |  | Write X86/64 ALP to perform non-overlapped block transfer without string specific instructions. Block containing data can be defined in the data segment. |  |  |  |
| **8** |  | Write X86/64 ALP to perform multiplication of two 8-bit hexadecimal numbers. Use successive addition  and add and shift method. (use of 64-bit registers is expected). |  |  |  |
| **9** |  | Write X86 ALP to find, a) Number of Blank spaces b) Number of lines c) Occurrence of a particular character. Accept the data from the text file. The text file has to be accessed during Program\_1 execution and write FAR PROCEDURES in Program\_2 for the rest of the processing. Use of PUBLIC and EXTERN |  |  |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | directives is mandatory. |  |  |  |
| **10** |  | Write x86 ALP to find the factorial of a given integer number on a command line by using recursion. Explicit stack manipulation is expected in the code. |  |  |  |

# Experiments No:01

**Name of Student**……………………………………………….. **Roll No**.:…… **Batch**:……. **Subject:** MPL **Department :** Computer Engg. **Class:** SE COMP

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Write-up** | **Correctness Program** | **Documentation** | **Timely Submission** | **Viva** | **Total** | **Dated Sign of Subject Teacher** |
| **4** | **4** | **4** | **4** | **4** | **20** |  |
|  |  |  |  |  |  |

Date of Performance:..................................... Date of Completion:....................................

**----------------------------------------------------------------------------------------------------------------------**

**Title:** Numbers from user and store them in an array and display the accepted numbers.

**Problem Statement:** Write an X86/64 ALP to accept five 64 bit Hexadecimal numbers from user and store them in an array and display the accepted numbers.

## Objective:

* To understand assembly language programming instruction set To understand different assembler directives with example
* To apply instruction set for implementing X86/64 bit assembly language programs

**Outcomes:** On completion of this practical ,students will be able to

**C218.1:** Understand and apply various addressing modes and instruction set to implement assembly language programs

## Hardware Requirement: NA

**Software Requirement: OS:**Ubuntu Assembler: NASM version 2.10.07 Linker: ld

## Theory Contents :

**Introduction to Assembly Language Programming:**

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 like getting input from the keyboard, displaying information on screen and performing various other jobs. These set of instructions are called 'machine language instruction'. Processor understands only machine language instructions which are strings of 1s and 0s. 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. Assembly language is a low-level programming language for a computer, or other programmable device specific to particular computer architecture in contrast to most high-level programming languages, which are generally portable across multiple systems. Assembly language is converted into executable machine code by a utility program referred to as an assembler like NASM, MASM etc.

## Advantages of Assembly Language

 An understanding of assembly language provides knowledge of:

 Interface of programs with OS, processor and BIOS;

 Representation of data in memory and other external devices;

 How processor accesses and executes instruction;

 How instructions accesses and process data;

 How a program access external devices.

Other advantages of using assembly language are:

 It requires less memory and execution time;

 It allows hardware-specific complex jobs in an easier way;

 It is suitable for time-critical jobs;

## ALP Step By Step:

**Installing NASM:**

If you select "Development Tools" while installed Linux, you may 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:

Open Terminal and run below commands:

sudo apt-get update

sudo apt-get install nasm

## Assembly Basic Syntax:

An assembly program can be divided into three sections:

 The **data** section

 The **bss** section

 The **text** section

The order in which these sections fall in your program really isn’t important, but by convention the.data section comes first, followed by the .bss section, and then the .text section.

## The .data Section

The .data section contains data definitions of initialized data items. Initialized data is data that has a value before the program begins running. These values are part of the executable file. They are loaded into memory when the executable file is loaded into memory for execution. You don’t have to load them with their values, and no machine cycles are used in their creation beyond what it takes to load the program as a whole into memory. The important thing to remember about the

.data section is that the more initialized data items you define, the larger the executable file will be, and the longer it will take to load it from disk into memory when you run it.

## The .bss Section

Not all data items need to have values before the program begins running. When you’re reading data from a disk file, for example, you need to have a place for the data to go after it comes in from disk.

Data buffers like that are defined in the .bss section of your program. You set aside some number of bytes for a buffer and give the buffer a name, but you don’t say what values are to be present in the buffer. There’s a crucial difference between data items defined in the .data section and data items defined in the .bss section: data items in the .data section add to the size of your executable file. Data items in the .bss section do not.

## The .text Section

The actual machine instructions that make up your program go into the .text section. Ordinarily, no data items are defined in .text. The .text section contains symbols called *labels* that identify locations in the program code for jumps and calls, but beyond your instruction mnemonics, that’s about it.

All global labels must be declared in the .text section, or the labels cannot be ‘‘seen’’ outside your program by the Linux linker or the Linux loader. Let’s look at the labels issue a little more closely.

## Labels

A label is a sort of bookmark, describing a place in the program code and giving it a name that’s easier to remember than a naked memory address. Labels are used to indicate the places where jump instructions should jump to, and they give names to callable assembly language procedures. Here are the most important things to know about labels:

 *Labels must begin with a letter, or else with an underscore, period, or question mark.* These last

three have special meanings to the assembler, so don’t use them until you know how NASM interprets them.

 *Labels must be followed by a colon when they are defined.* This is basically what tells NASM that the identifier being defined is a label. NASM will punt if no colon is there and will not flag an error, but the colon nails it, and prevents a mistyped instruction mnemonic from being mistaken for a label. Use the colon!

 *Labels are case sensitive.* So yikes:, Yikes:, and YIKES: are three completely different labels.

## Assembly Language Statements

Assembly language programs consist of three types of statements:

 Executable instructions or instructions

 Assembler directives or pseudo-ops

 Macros

## Syntax of Assembly Language Statements

[label] mnemonic [operands] [;comment]

**LIST OF INTERRRUPTS USED:** NA

**LIST OF ASSEMBLER DIRECTIVES USED:** EQU,DB

**LIST OF MACROS USED:** NA **LIST OF PROCEDURES USED:** NA **ALGORITHM:**

INPUT: ARRAY OUTPUT: ARRAY

STEP 1: Start.

STEP 2: Initialize the data segment.

STEP 3: Display msg1 “Accept array from user. “ STEP 4: Initialize counter to 05 and rbx as 00 STEP 5: Store element in array.

STEP 6: Move rdx by 17.

STEP 7: Add 17 to rbx. STEP 8: Decrement Counter.

STEP 9: Jump to step 5 until counter value is not zero. STEP 9: Display msg2.

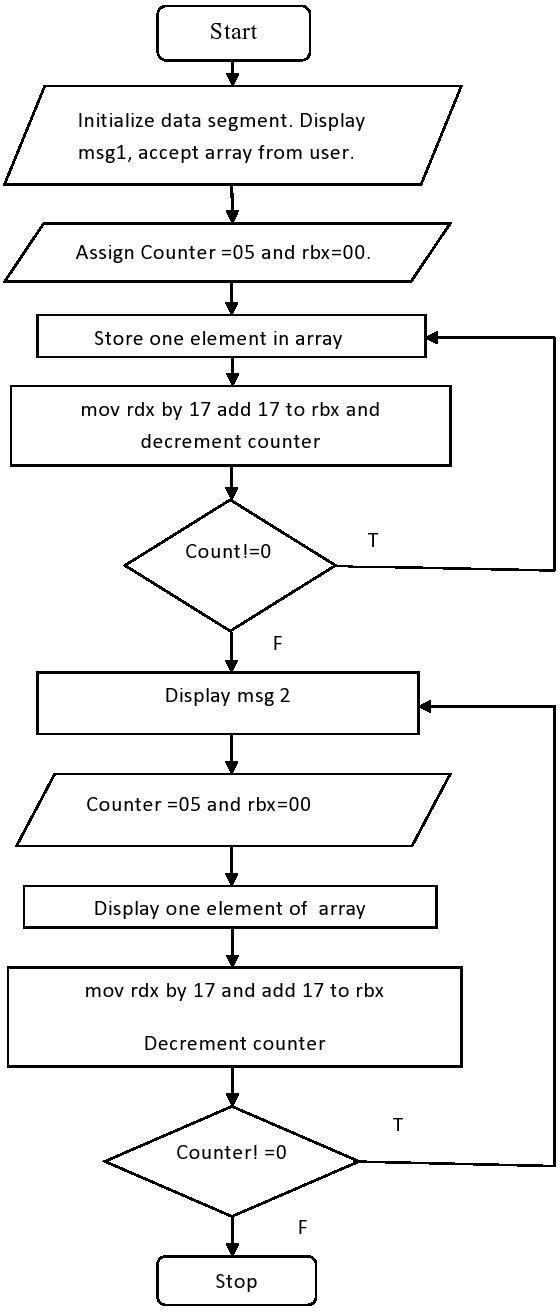
STEP 10: Initialize counter to 05 and rbx as 00 STEP 11: Display element of array.

STEP 12: Move rdx by 17.

STEP 13: Add 17 to rbx. STEP 14: Decrement Counter.

STEP 15: Jump to step 11 until counter value is not zero. STEP 16: Stop

## FLOWCHART:

****

**Conclusion:**

Hence we accept number from user and display from user.

**MPL Practical Oral Question Bank**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Sr No** | **B L** | **Questions** | **Oral 1** | **Oral 2 (improve ment)** | **Remark** |
| 1 | 1 | What is macro and its definition? |  |  |  |
| 2 | 2 | Explain the difference between 32bit and 64bit processors? |  |  |  |
| 3 | 1 | Explain steps of running an ALP program? |  |  |  |
| 4 | 2 | What is MASM ,TASM,NASM? |  |  |  |
| 5 | 1 | What is the data section,text section and bss section? |  |  |  |

**Sign of Student**

**Experiments No:2**

**Name of Student**……………………………………………….. **Roll No**.:…… **Batch**:……. **Subject:**MPL **Department :**Computer Engg. **Class:** SE COMP

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Write-up** | **Correctness Program** | **Documentation** | **Timely Submission** | **Viva** | **Total** | **Dated Sign of Subject**  **Teacher** |
| **4** | **4** | **4** | **4** | **4** | **20** |  |
|  |  |  |  |  |  |

Date of Performance:..................................... Date of Completion:....................................

**----------------------------------------------------------------------------------------------------------------------**

**Title:** Display **s**tring

**Problem Statement:** Write an X86/64 ALP to accept a string and to display its length.

**Objective:** To learn and understand the operation of accept and display string.

**Outcomes:** On completion of this practical ,students will be able to

**C218.1:** Understand and apply various addressing modes and instruction set to implement assembly language programs

**Hardware Requirement: NA**

**Software Requirement:** Ubuntu ,NASM etc.

**Theory Contents :**

**The 80x86 String Instructions:** All members of the 80 x 86 families support five different string instructions: MOVS, CMPS, SCAS, LODS and STOS. They are the string primitives since you can build most other string operations from these five instructions.

**How the String Instructions Operate:** The string instructions operate on blocks (contiguous linear arrays) of memory. For example, the MOVS instruction moves a sequence of bytes from one memory location to another. The CMPS instruction compares two blocks of memory. The

SCAS instruction scans a block of memory for a particular value. These string instructions often require three operands a destination block address a source block address and (optionally) an element count. For example, when using the MOVS instruction to copy a string you need a source address a destination address and a count (the number of string elements to move).

Unlike other instructions which operate on memory the string instructions are single-byte instructions which don't have any explicit operands. The operands for the string instructions include

The SI (source index) register

The DI (destination index) register The CX (count) register

The AX register and

The direction flag in t e FLAGS register.

For example one variant of the MOVS (move string) instruction copies a string from the source address specified by DS:SI to the destination address sspecified by ES:DI of length CX. Likewise, the CMPS instruction compares the string pointed at by DS:SI of length CX to the string pointed at by ES: DI.

Not all instructions have source and destination operands (only MOVS and CMPS support them). For example, the SCAS instruction (scan a string) compares the value in the accumulator to values in memory. Despite their differences the 80x86's string instructions all have one thing in common - using them requires that you deal with two segments the data segment and the extra segment.

**The REP/REPE/REPZ and REPNZ/REPNE Prefixes**

The string instructions by themselves do not operate on strings of data. The MOVS instruction for example will move a single byte word or double word. When executed by itself the MOVS instruction ignores the value in the CX register. The repeat prefixes tell the 80x86 to do a multi-byte string operation. The syntax for the repeat prefix is:

**Field: Label repeat mnemonic operand; comment For MOVS:**

REP MOVS

**For CMPS:**

REPE CMPS REPZ REPNE REPNZ CMPS

**For SCAS:**

REPE SCAS REPZ SCAS REPNE SCAS REPNZ SCAS **For STOS:**

REP STOS {operands}

You don't normally use the repeat prefixes with the LODS instruction.

As you can see the presence of the repeat prefixes introduces a new field in the source line

- the repeat prefix field. This field appears only on source lines containing string instructions. In your source file:

The label field should always begin in column one

The repeat field should begin at the first tab stop and

The mnemonic field should begin at the second tab stop.

When specifying the repeat prefix before a string instruction the string instruction repeats CX times. Without the repeat prefix the instruction operates only on a single byte word or double word.

You can use repeat prefixes to process entire strings with a single instruction. You can use the string instructions without the repeat prefix as string primitive operations to synthesize more powerful string operations.

The operand field is optional. If present MASM simply uses it to determine the size of the string to

operate on. If the operand field is the name of a byte variable the string instruction operates on bytes. If the operand is a word address the instruction operates on words. Likewise for double words. If the operand field is not present you must append a "B" "W" or "D" to the end of the string instruction to denote the size e.g. MOVSB MOVSW or MOVSD.

**The Direction Flag**

Besides the SI, DI and ax registers one other register controls the 80x86's string instructions - the flags register. Specifically, the direction flag in the flags register controls how the CPU processes strings.

If the direction flag is clear the CPU increments SI and DI after operating upon each string element. For example if the direction flag is clear then executing MOVS will move the byte word or double word at DS:SI to ES:DI and will increment SI and DI by one two or four. When specifying the REP prefix before this instruction the CPU increments SI and DI for each element in the string. At completion the SI and DI registers will be pointing at the first item beyond the string.

If the direction flag is set, then the 80x86 decrements si and di after processing each string element. After a repeated string operation, the si and di registers will be pointing at the first byte or word before the strings if the direction flag was set.

The direction flag may be set or cleared using the cld (clear direction flag) and std (set direction flag) instructions. When using these instructions inside a procedure keep in mind that they modify the machine state. Therefore you may need to save the direction flag during the execution of that procedure.

**Output :** The output shows us actual length of string without using inbuilt string Function.

**Conclusion:**In this way we studied about accepting and displaying strings using 80386 microprocessors.

**MPL Practical Oral Question Bank**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Sr No** | **B L** | **Questions** | **Oral 1** | **Oral 2 (improve ment)** | **Remark** |
| 1 | 1 | What is a MacroProcessor? |  |  |  |
| 2 | 1 | What are the types of registers and their use? |  |  |  |
| 3 | 2 | What is addressing mode and its type? |  |  |  |
| 4 | 1 | Which file descriptors are used in an Assembly program? |  |  |  |
| 5 | 2 | What is a System call?Give examples of the same. |  |  |  |
| 6 | 1 | What is the use of Macro? |  |  |  |
| 7 | 2 | Which Assembler directive is used for macro? |  |  |  |

**Sign of Student**

# Experiments No:03

**Name of Student**……………………………………………….. **Roll No**.:…… **Batch**:……. **Subject:** MPL **Department :**Computer Engg. **Class:** SE COMP

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Write-up** | **Correctness Program** | **Documentation** | **Timely Submission** | **Viva** | **Total** | **Dated Sign of Subject Teacher** |
| **4** | **4** | **4** | **4** | **4** | **20** |  |
|  |  |  |  |  |  |

Date of Performance:..................................... Date of Completion:....................................

**----------------------------------------------------------------------------------------------------------------------**

**Title:** To find the largest of given Byte/Word/Dword/64-bit numbers.

**Problem Statement:** Write an X86/64 ALP to find the largest of given Byte/Word/Dword/64-bit numbers.

## Objective:

* To understand assembly language programming instruction set To understand different assembler directives with example
* To apply instruction set for implementing X86/64 bit assembly language programs

**Outcomes:** On completion of this practical ,students will be able to

**C218.1:** Understand and apply various addressing modes and instruction set to implement assembly language programs

## Hardware Requirement: NA

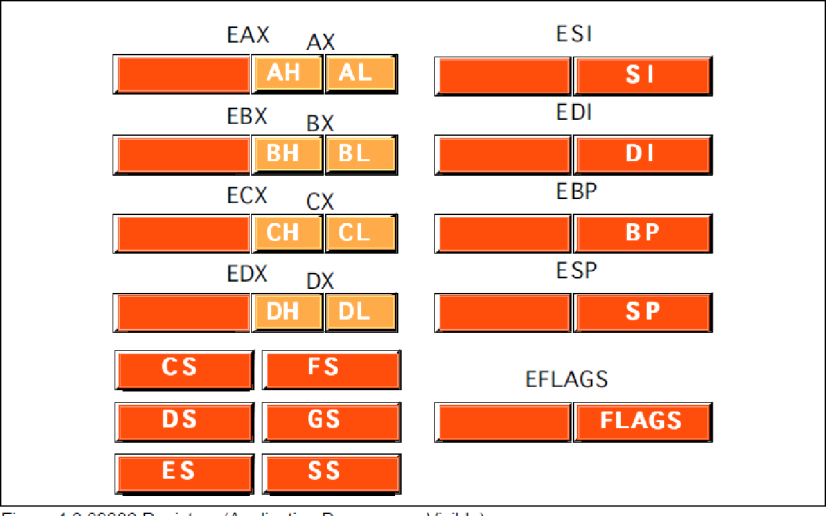
**Software Requirement: OS:**Ubuntu Assembler: NASM version 2.10.07 Linker: ld

## Theory Contents :Datatype in 80386:

The 80386 supports the following data types they are

* Bit
* Bit Field: A group of at the most 32 bits (4bytes)
* Bit String: A string of contiguous bits of maximum 4Gbytes in length.
* Signed Byte: Signed byte data
* Unsigned Byte: Unsigned byte data.
* Integer word: Signed 16-bit data.
* Long Integer: 32-bit signed data represented in 2's complement form.
* Unsigned Integer Word: Unsigned 16-bit data
* Unsigned Long Integer: Unsigned 32-bit data
* Signed Quad Word: A signed 64-bit data or four word data.
* Unsigned Quad Word: An unsigned 64-bit data.
* Offset: 16/32-bit displacement that points a memory location using any of the addressing modes.
* Pointer: This consists of a pair of 16-bit selector and 16/32-bit offset.
* Character: An ASCII equivalent to any of the alphanumeric or control characters.
* Strings: These are the sequences of bytes, words or double words. A string may contain minimum one byte and maximum 4 Gigabytes.
* BCD: Decimal digits from 0-9 represented by unpacked bytes.
* Packed BCD: This represents two packed BCD digits using a byte, i.e. from 00 to 99.

## Registers in 80386:

****

* + General Purpose Register: EAX, EBX, ECX, EDX
  + Pointer register: ESP, EBP
  + Index register: ESI, EDI
  + Segment Register: CS, FS, DS, GS, ES, SS
  + Eflags register: EFLAGS
  + System Address/Memory management Registers : GDTR, LDTR, IDTR
  + Control Register: Cr0, Cr1, Cr2, Cr3
  + Debug Register : DR0, DR,1 DR2, DR3, DR4, DR5, DR6, DR7
  + Test Register: TR0, TR,1 TR2, TR3, TR4, TR5, TR6, TR7

|  |  |  |
| --- | --- | --- |
| EAX | AX | AH,AL |
| EBX | BX | BH,BL |
| ECX | CX | CH,CL |
| EDX | DX | DH,DL |
| EBP | BP |  |
| EDI | DI |  |
| ESI | SI |  |
| ESP |  |  |

## Size of operands in an Intel assembler instruction

* + Specifying the size of an operand in Intel
  + The size of the operand (byte, word, double word) is conveyed by the operand itself
* EAX means: a 32 bit operand
* AX means: a 16 bit operand
* AL means: a 8 bit operand The size of the source operand and the destination operand must be equal

**Addressing modes in 80386:** The purpose of using addressing modes is as follows:

1. To give the programming versatility to the user.
2. To reduce the number of bits in addressing field of instruction.
3. Register addressing mode: MOV EAX, EDX
4. Immediate Addressing modes: MOV ECX, 20305060H
5. Direct Addressing mode: MOV AX, [1897 H]
6. Register Indirect Addressing mode MOV EBX, [ECX]
7. Based Mode MOV ESI, [EAX+23H]
8. Index Mode SUB COUNT [EDI], EAX
9. Scaled Index Mode MOV [ESI\*8], ECX
10. Based Indexed Mode MOV ESI, [ECX][EBX]
11. Based Index Mode with displacement EA=EBX+EBP+1245678H
12. Based Scaled Index Mode with displacement MOV [EBX\*8] [ECX+5678H], ECX
13. String Addressing modes:
14. Implied Addressing modes:

## Assignment Questions:

* 1. Explain Types of Addressing mode instruction?
  2. Write five basic instructions which are used in ALP?
  3. How to find the largest number from a given number?

## Conclusion:

Hence we find largest number from given number

**MPL Practical Oral Question Bank**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Sr No** | **B L** | **Questions** | **Oral 1** | **Oral 2 (improve ment)** | **Remark** |
| 1 | 1 | What are the type of instruction? |  |  |  |
| 2 | 2 | What is difference between byte,word and double word? |  |  |  |
| 3 | 1 | What is format of instruction? |  |  |  |
| 4 | 2 | What is use of label in instruction? |  |  |  |

**Sign of Student**

# Experiments No:04

**Name of Student**……………………………………………….. **Roll No**.:…… **Batch**:……. **Subject:**MPL **Department :**Computer Engg. **Class:** SE COMP

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Write-up** | **Correctness Program** | **Documentation** | **Timely Submission** | **Viva** | **Total** | **Dated Sign of Subject Teacher** |
| **4** | **4** | **4** | **4** | **4** | **20** |  |
|  |  |  |  |  |  |

Date of Performance:..................................... Date of Completion:....................................

**----------------------------------------------------------------------------------------------------------------------**

**Title:** Count number of positive and negative numbers from the array.

**Problem Statement:** Write an X86/64 ALP to count number of positive and negative numbers from the array.

**Objective:**To understand assembly language programming instruction set.

To understand different assembler directives with example. To apply instruction set for implementing X86/64 bit assembly language programs

**Outcomes:** On completion of this practical ,students will be able to

**C218.1:** Understand and apply various addressing modes and instruction set to implement assembly language programs

## Hardware Requirement: NA

**Software Requirement: OS:**Ubuntu Assembler: NASM version 2.10.07 Linker: ld

## Theory Contents :

Mathematical numbers are generally made up of a sign and a value (magnitude) in which the sign indicates whether the number is positive, ( + ) or negative, ( – ) with the value indicating the size of the number, for example 23, +156 or -274. Presenting numbers is this fashion is called

“sign-magnitude” representation since the left most digit can be used to indicate the sign and the remaining digits the magnitude or value of the number.

Sign-magnitude notation is the simplest and one of the most common methods of representing positive and negative numbers either side of zero, (0). Thus negative numbers are obtained simply by changing the sign of the corresponding positive number as each positive or unsigned number will have a signed opposite, for example, +2 and -2, +10 and -10, etc.

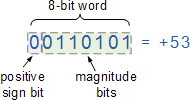
But how do we represent signed binary numbers if all we have is a bunch of one’s and zero’s. We know that binary digits, or bits only have two values, either a “1” or a “0” and conveniently for us, a sign also has only two values, being a “**+**” or a “**–**“.

Then we can use a single bit to identify the sign of a *signed binary number* as being positive or negative in value. So to represent a positive binary number (+n) and a negative (-n) binary number, we can use them with the addition of a sign.

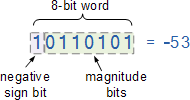
For signed binary numbers the most significant bit (MSB) is used as the sign bit. If the sign bit is “0”, this means the number is positive in value. If the sign bit is “1”, then the number is negative in value. The remaining bits in the number are used to represent the magnitude of the binary number in the usual unsigned binary number format way.

Then we can see that the Sign-and-Magnitude (SM) notation stores positive and negative values by dividing the “n” total bits into two parts: 1 bit for the sign and n–1 bits for the value which is a pure binary number. For example, the decimal number 53 can be expressed as an 8-bit signed binary number as follows.

**Positive Signed Binary Numbers**

****

**Negative Signed Binary Numbers**

****

**LIST OF INTERRRUPTS USED:** 80h

**LIST OF ASSEMBLER DIRECTIVES USED:** equ, db

**LIST OF MACROS USED:** print

**ALGORITHM:**

STEP 1: Initialize index register with the offset of array of signed numbers STEP 2: Initialize ECX with array element count

STEP 3: Initialize positive number count and negative number count to zero STEP 4: Perform MSB test of array element

STEP 5: If set jump to step 7

STEP 6: Else Increment positive number count and jump to step 8 STEP 7: Increment negative number count and continue

STEP 8: Point index register to the next element

STEP 9: Decrement the array element count from ECX, if not zero jump to step 4, else continue STEP 10: Display Positive number message and then display positive number count

STEP 11: Display Negative number message and then display negative number count STEP 12: EXIT

## Assignment Questions:

1. Write Down Write, Read, Exit System Call for 32bits Operating System?
2. What is the difference between Macros & Procedure?
3. Explain BT Instruction?
4. Explain Significance of Sign Bit?
5. Explain How Loop Instruction Works?
6. Explain resb,resw,resq, db, dw,dq?
7. Explain the role of Section .data, section .bss, section .text?

## Conclusion:

Hence we counted and displayed the number of positive and negative numbers from the array of signed numbers.

**MPL Practical Oral Question Bank**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Sr No** | **B L** | **Questions** | **Oral 1** | **Oral 2 (improve ment)** | **Remark** |
| 1 | 1 | What is use of Write, Read, Exit System Call  ? |  |  |  |
| 2 | 2 | Difference between sign number and unsigned number? |  |  |  |
| 3 | 1 | What is the use of resb,resw,resq, db,  dw,dq? |  |  |  |
| 4 | 2 | What is the role of Section .data, section  .bss, section .text? |  |  |  |

**Sign of Student**

# Experiments No:05

**Name of Student**……………………………………………….. **Roll No**.:…… **Batch**:……. **Subject:**MPL **Department :**Computer Engg. **Class:** SE COMP

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Write-up** | **Correctness Program** | **Documentation** | **Timely Submission** | **Viva** | **Total** | **Dated Sign of Subject Teacher** |
| **4** | **4** | **4** | **4** | **4** | **20** |  |
|  |  |  |  |  |  |

Date of Performance:..................................... Date of Completion:....................................

**----------------------------------------------------------------------------------------------------------------------**

**Title:** HEX to BCD and BCD to HEX

**Problem Statement:** Write X86/64 ALP to convert 4-digit Hex number into its equivalent BCD number and 5-digit BCD number into its equivalent HEX number. Make your program user friendly to accept the choice from user for: (a) HEX to BCD b) BCD to HEX (c) EXIT.Display proper strings to prompt the user while accepting the input and displaying the result. (Wherever necessary, use 64-bit registers).

**Objective:** To learn the implementation stack for its conversion of number

**Outcomes:** On completion of this practical ,students will be able to

**C218.2:** Apply logic to implement code conversion

## Hardware Requirement: NA

**Software Requirement:** Ubuntu ,NASM etc.

## Theory Contents :

1. **Memory:** model directives

These directives instruct the assembler as to how large the various segment (code, data, stack, etc) can be and what sort of segmentation register will be required.

.MODEL

<model >

Where <model>is one of the following options:

Tiny- code and data fit into single 64k and accessed via near pointers.

Small- code and segment both less than 64k and accessed via near pointers. Compact – Code segment is <64K (near ptr) and data segment is <1MB (far ptr ) Medium- Code segment is <1Mb (far ptr) and data segment is <64K (near ptr) Large- code and data segment both less than 1MB and accessed via far pointers. Huge- Like “large” model, but also permits arrays larger then 64K

## Segment directives

These directives indicate to assembler the order in which to load segment. When it encounter one of these directives, it interprets all subsequent instructor as belonging to the indicated segment (until the ne xt directives is encounter).

. data

.stack <size>; specified .code

## Data type declaration

As has been previously discu ssed, data can be of several different lengths and assembler must be able to decide what length a specific constant (or variable) is. This can be down using data type declaration in conjunction with a constant declaration or variable assignment .this is akin to strong typing of variable in high level language .the data types are:

Byte (8 bit quantity)—synonyms are byte and db Word (16) -- synonyms are word dw

Dword (32bit) -- synonyms are dword and dd Qword (64bit) -- synonyms with dq

Tword (128bit) -- synonyms with dt An example of their use is:

MOV AX, word VAR; moves a 16-bit Variable VAR into AX

## Instruction Used:

* 1. PUSH:-Push word onto stack.
  2. POP:-Pop word off stack.
  3. DIV:-Divide byte/word
  4. XOR: - Exclusive or byte/word
  5. JA/JNBE:-Jump if above/not below or equal
  6. JB/JNAE:-Jump if below /not above or equal
  7. JG/JNLE:-Jump if /not less nor equal
  8. JL/JNGE:-Jump if less /not greater nor equal
  9. INT:-It allows ISR to be activated by programmer & external H\W device

## New interrupt used:

1 INT 21h, function 0AH:- Read from keyboard and place into a memory buffer a row of character, until<CR>is pressed.

## .New directives used:

* 1. .MODEL
  2. .STACK
  3. .DATA
  4. .CODE
  5. .OFFSET: - It informs the assembler to determine the offset/displacement of a named data item.
  6. .PTR: - assign a specific type to variable/label

## Algorithm:

**HEX to BCD**

* 1. Define variable on data segment.
  2. Display message on screen ENTER 4-DIGIT HEX NO.
  3. Accept BCD NO from user.
  4. Transfer 0AH as a divisor in one of the register.
  5. Divide the no by 0AH
  6. PUSH reminder in one of the register
  7. Increment Count \_1.
  8. Repeat Till BCD NO is not zero go to step 5.
  9. Pop the content of Reminder.
  10. Display result by calling display procedure.
  11. Decrement Count \_1, till Count is not zero repeat step 9 else go to step 12.
  12. Stop

## BCD to HEX

1. Define variables in data segment
2. Display message on screen ENTER 5-DIGIT BCD NO.
3. Accept single digit from user
4. Transfer 10000 to multiplier
5. Multiply accepted BCD digit by multiplier & add it to RESULT variable.
6. Accept single digit from user
7. Transfer 1000 to multiplier
8. Multiply accepted BCD digit by Multiplier & Add it to RESULT variable.
9. Accept single digit from user
10. Transfer 100 to multiplier
11. Multiply accepted BCD digit by Multiplier & Add it to RESULT variable.
12. Accept single digit from user
13. Transfer 10 to multiplier
14. Multiply accepted BCD digit by multiplier & add it to RESULT variable.
15. Accept single digit from user
16. Transfer 1 to multiplier
17. Multiply accepted BCD digit by multiplier & add it to RESULT variable.
18. Display result by calling display procedure
19. Stop.

## Procedure for accept numbers: (ASCII to HEX)

1. Read a single character/digit from keyboard using function 0AH of INT 21H
2. Convert ASCII to HEX as per following:
   1. Compare its ASCII with 30H if No is less than 0 (i.e case of -ve no given) then go to step f else go to step c.
   2. Compare its ASCII with 39H if No is greater than 9 (i.e case of character A – F given

then go to step f else go to step c .

* 1. Store the resultant bit in NUM Variable.
  2. Check whether four digits (16-bit number) or two digits (8-bit number) are read; if yes then go to display procedure else go to step 1 for next bit
  3. Till counter is zero go to accept procedure.
  4. Display massage “I/P is invalid BCD number” & go to step 17.
  5. End of accept procedure.

## Procedure for display Result: (HEX to ASCII)

1. Compare 4 bits (one digit) of number with 9
2. If it is <= 9 then go to step 4 else go to step 3
3. Add 07 to that number
4. Add 30 to it
5. Display character on screen using function 02 of INT 21H
6. Return to main routine
7. End of display procedure.

## HEX to BCD

Divide FFFF by 10 --- note this FFFF is as decimal 65535 so Division

65535 / 10 Quotient = 6553 Reminder = 5

6553 / 10 Quotient = 655 Reminder = 3

655 / 10 Quotient = 65 Reminder = 5

65 / 10 Quotient = 6 Reminder = 5

6 / 10 Quotient = 0 Reminder = 6

and we are pushing Reminder on stack and then printing it in reverse order

## BCD to HEX

1. LOOP : DL = 06 ; RAX = RAX \* RBX = 0 ; RAX = RAX + RDX = 06
2. LOOP : DL = 05 ; 60 = 06 \* 10 ; 65 = 60 + 5
3. LOOP : DL = 05 ; 650 = 60 \* 10 ; 655 = 650 + 5
4. LOOP : DL = 03 ; 6550 = 655 \* 10 ; 6553 = 6550 + 3
5. LOOP : DL = 06 ; 65530 = 6553 \* 10 ; 65535 = 65530 + 5

Hence final result is in EAX = 65535 which is 1111 1111 1111 1111 and when we print this it is represented as FFFF

## Assignment Questions:

1. Explain various addressing modes of 8086 microprocessor used in this program.
2. Explain Different assembler directives used in this program.
3. Explain various Number Systems used in Digital Electronics.
4. What is HEX Number? Explain the steps to convert HEX No to BCD No
5. What is BCD Number? Explain the steps to convert BCD No to HEX No.

**Conclusion:** In this way we studied about hex to BCD and hex to BCD number conversion.

**MPL Practical Oral Question Bank**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Sr No** | **B L** | **Questions** | **Oral 1** | **Oral 2 (improve ment)** | **Remark** |
| 1 | 2 | Explain the use of SI register in the program. |  |  |  |
| 2 | 2 | Explain the use of DI register in the program. |  |  |  |
| 3 | 1 | Is there any difference between Rotate Bit &  Shift Bit Instruction? |  |  |  |
| 4 | 2 | Explain the difference 32 bit instruction and 64 bit instruction? |  |  |  |
| 5 | 1 | What is Push,Pop and Div instruction? |  |  |  |

**Sign of Student**

# Experiments No:06

**Name of Student**……………………………………………….. **Roll No**.:…… **Batch**:……. **Subject:**MPL **Department :**Computer Engg. **Class:** SE COMP

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Write-up** | **Correctness Program** | **Documentation** | **Timely Submission** | **Viva** | **Total** | **Dated Sign of Subject Teacher** |
| **4** | **4** | **4** | **4** | **4** | **20** |  |
|  |  |  |  |  |  |

Date of Performance:..................................... Date of Completion:....................................

**----------------------------------------------------------------------------------------------------------------------**

**Title:** display the values of GDTR, LDTR, IDTR,TR and MSW Registers

**Problem Statement:** Write X86/64 ALP to detect protected mode and display the values of GDTR, LDTR, IDTR,TR and MSW Registers also identify CPU type using CPUID instruction.

## Objective:

* To understand assembly language programming instruction set To understand different assembler directives with example
* To apply instruction set for implementing X86/64 bit assembly language programs

**Outcomes:** On completion of this practical ,students will be able to

**C218.1:** Understand and apply various addressing modes and instruction set to implement assembly language programs

## Hardware Requirement: NA

**Software Requirement: OS:**Ubuntu Assembler: NASM version 2.10.07 Linker: ld

## Theory Contents : Real Mode:

Real mode, also called real address mode, is an operating mode of all x86-compatible CPUs. Real mode is characterized by a 20-bit segmented memory address space (giving exactly 1 MiB of addressable memory) and unlimited direct software access to all addressable memory,

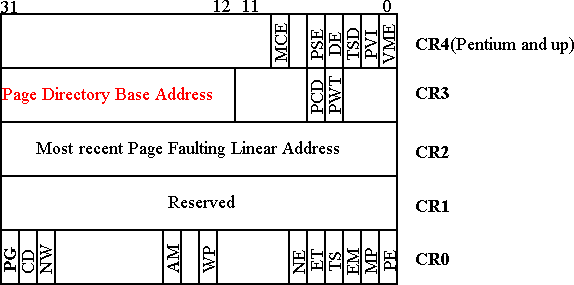
I/O addresses and peripheral hardware. Real mode provides no support for memory protection, multitasking, or code privilege levels.

## Protected Mode:

In computing, protected mode, also called protected virtual address mode is an operational mode of x86-compatible central processing units (CPUs). It allows system software to use features such as virtual memory, paging and safe multi-tasking designed to increase an operating system's control over application software.

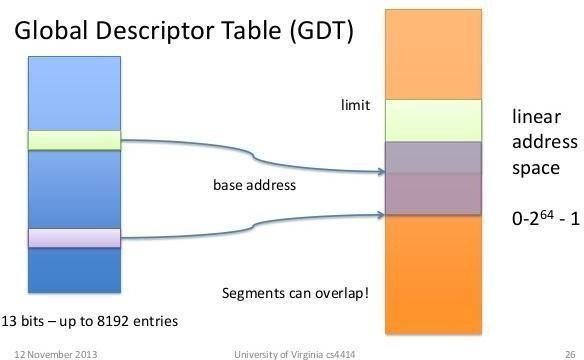
When a processor that supports x86 protected mode is powered on, it begins executing instructions in real mode, in order to maintain backward compatibility with earlier x86 processors. Protected mode may only be entered after the system software sets up several descriptor tables and enables the Protection Enable (PE) bit in the control register 0 (CR0).

## Control Register :

****

**Global Descriptor Table Register**

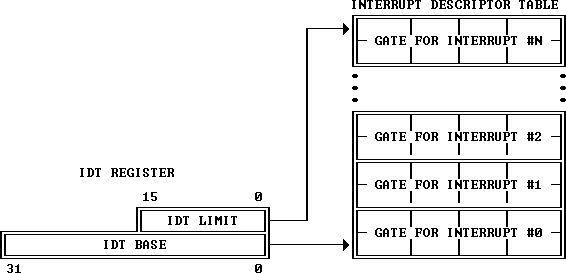
This register holds the 32-bit base address and 16-bit segment limit for the global descriptor table (GDT). When a reference is made to data in memory, a segment selector is used to find a segment descriptor in the GDT or LDT. A segment descriptor contains the base address for a segment.



## Local Descriptor Table Register

This register holds the 32-bit base address, 16-bit segment limit, and 16-bit segment selector for the local descriptor table (LDT). The segment which contains the LDT has a segment descriptor in the GDT. There is no segment descriptor for the GDT. When a reference is made to data in memory, a segment selector is used to find a segment descriptor in the GDT or LDT. A segment descriptor contains the base address for a segment

## Interrupt Descriptor Table Register

This register holds the 32-bit base address and 16-bit segment limit for the interrupt descriptor table (IDT). When an interrupt occurs, the interrupt vector is used as an index to get a gate descriptor from this table. The gate descriptor contains a far pointer used to start up the interrupt handler.

## Algorithm :

1. Start
2. Display the message using sys\_write call
3. Read CR0
4. Checking PE bit, if 1=Protected Mode
5. Load number of digits to display
6. Rotate number left by four bits
7. Convert the number in ASCII
8. Display the number from buffer
9. Exit using sys\_exit call

**Conclusion:** Hence we performed an ALP to program to use GDTR, LDTR and IDTR in Real Mode

## Assignment Question:

1. Explain instruction set of SMSW,SGDT,SLDT, STR?
2. What is GDT and GDTR?
3. What is LDT and LDTR?
4. What is IDT and IDTR?
5. What is selector?
6. Function of Descriptor (GDT ,LDT ,IDT) ?
7. What is mean by Interrupt Handler?
8. Explain Difference between Real Mode & Protected Mode?
9. Explain CR0 in Detail:
10. Explain POST Sequence?
11. Explain Flowchart of Real to Protected Mode Switch?

**MPL Practical Oral Question Bank**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Sr No** | **B L** | **Questions** | **Oral 1** | **Oral 2 (improve ment)** | **Remark** |
| 1 | 1 | What is GDT and GDTR? |  |  |  |
| 2 | 1 | What is LDT and LDTR? |  |  |  |
| 3 | 1 | What is IDT and IDTR? |  |  |  |

**Sign of Student**

# Experiments No:07

**Name of Student**……………………………………………….. **Roll No**.:…… **Batch**:……. **Subject:**MPL **Department :**Computer Engg. **Class:** SE COMP (Div-A/B)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Write-up** | **Correctness Program** | **Documentation** | **Timely Submission** | **Viva** | **Total** | **Dated Sign of Subject Teacher** |
| **4** | **4** | **4** | **4** | **4** | **20** |  |
|  |  |  |  |  |  |

Date of Performance:..................................... Date of Completion:....................................

**----------------------------------------------------------------------------------------------------------------------**

**Title:**Find factorial of a given integer number.

**Problem Statement:** Write X86/64 ALP to perform non-overlapped block transfer without string specific instructions. Block containing data can be defined in the data segment.

## Objective:

* Understand the memory Addressing
* Understand the localization of Data

**Outcomes:** On completion of this practical ,students will be able to

**C218.1:** Understand and apply various addressing modes and instruction set to implement assembly language programs

## Hardware Requirement: NA

**Software Requirement: OS:**Ubuntu Assembler: NASM version 2.10.07 Linker: ld

## Theory Contents :

* + 1. **Registers**

Registers are places in the CPU where a number can be stored and manipulated. There are three sizes of registers: 8-bit, 16-bit and on 386 and above 32-bit. There are four different types of registers:

1. **General Purpose Registers,**
2. **Segment Registers,**
3. **Index Registers,**
4. **Stack Registers.**
   * + 1. **General Registers**

Following are the registers that are used for general purposes in 8086 AX accumulator (16 bit)

AH accumulator high-order byte (8 bit) AL accumulator low-order byte (8 bit) BX accumulator (16 bit)

BH accumulator high-order byte (8 bit) BL accumulator low-order byte (8 bit) CX count and accumulator (16 bit)

CH count high order byte (8 bit) CL count low order byte (8 bit)

DX data and I/O address (16 bit) DH data high order byte (8 bit) DL data low order byte (8 bit)

## Segment Registers –

These registers are used to calculate 20 bit address from 16 bit registers. CS code segment (16 bit)

DS data segment (16 bit) SS stack segment (16 bit) ES extra segment (16 bit)

## Index Registers –

These registers are used with the string instructions. DI destination index (16 bi

SI source index (16 bit)

## Pointers –

These registers are used with the segment register to obtain 20 bit addresses SP stack pointer (16 bit) BP base pointer (16 bit)

IP instruction pointer (16 bit)

## CS, Code Segment

Used to “point” to Instructions Determines a Memory Address (along with IP) Segmented Address written as CS:IP

## DS, Data Segment

Used to “point” to Data Determines Memory Address (along with other registers) ES, Extra Segment allows 2 Data Address Registers

## SS, Stack Segment

Used to “point” to Data in Stack Structure (LIFO) Used with SP or BP

## SS: SP or SS:BP are valid Segmented Addresses IP, Instruction Pointer

Used to “point” to Instructions Determines a Memory Address (along with CS) Segmented Address written as CS:IP

## SI, Source Index; DI, Destination Index

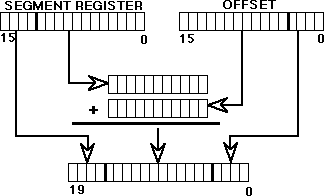
Used to “point” to Data Determines Memory Address (along with other registers) DS, ES commonly used

## SP, Stack Pointer; BP, Base Pointer

Used to “point” to Data in Stack Structure (LIFO) Used with SS SS:SP or SS:BP are valid Segmented Address

## Memory Address Calculations

The 8086 uses a 20 bit address but the registers are only sixteen bit. To derive twenty bit addresses from the registers two registers are Combined every memory reference uses one of the four segment registers plus and offset abd/or a base pointer and/or a index register. The segment register is multiplied by sixteen (shifted to the left four bits) and added to the sixteen bit result of the offset calculation.



## Figure 1Diagrammatic Representation of Address calculation

The 8086 provides four segment registers for address calculations. Each segment register is assigned a different task. The code segment register is always used with the instruction Pointer (also called the program counter) to point to the instruction that is to be executed next. The stack segment register is always used with the stack pointer to point to the last value pushed onto the stack. The extra segment is general purpose segment register. The data segment register is the default register to calculate data operations, this can be over ridden by specifying the segment register. For example *mov ax,var1* would use the offset var1 and the data segment to calculate the memory reference but *mov ax,ss:var1* would use the offset var1 and the stack segment register to calculate the memory reference.

The offset can be calculated in a number of ways. Their are three elements that can make up an offset. The first element is a base register, this can be one of the BX of BP registers (the BP register defaults to the stack segment). The second element is one of the index register, SI or DI. The third element is a displacement. A displacement can be a numerical value or an offset to a label. An offset can contain one to three of these elements, making a total of sixteen possibilities.

BX SI

or + or + Displacement BP DI

*(base) (index)*

The offset to a label in calculated using the assembler directive OFFSET. This directive makes the assembler calculattehe distant from the start of the segment that the label

resides in to the label.

## Memory Segmentation

x86 Memory Partitioned into Segments

* 8086: maximum size is 64K (16-bit index reg.)
* 8086: can have 4 active segments (**CS, SS, DS, ES**)
* 8086: 2-data; 1-code; 1-stack
* x386: maximum size is 4GB (32-bit index reg.)
* x386: can have 6 active segments (4-data; **FS, GS**)

## Why have segmented memory?

Other microprocessors could only address 64K since they only had a single 16-bit MAR (or smaller). Segments allowed computers to be built that could use more than 64K memory (but not all at the same time).

*Segment*

CS E S SS D S

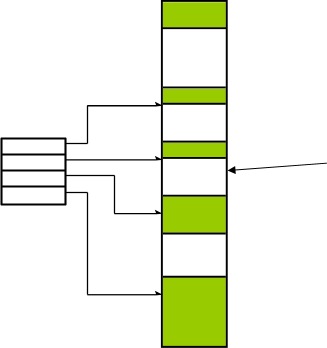
Code

Extra Stack

Data

FFFF

Fh



* + Segment Registers:

–Point to Base Address

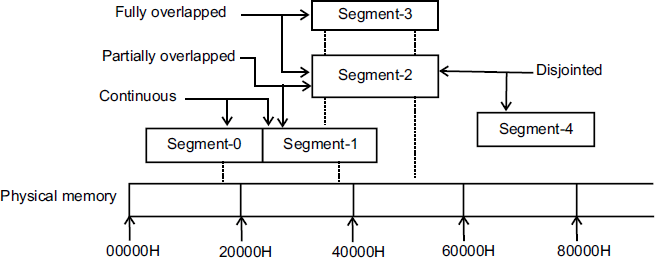
*Fragmentation*

* + Index Registers:

–Contain Offset Value

* + Notation (*Segmented Address*):





Note that segments can overlap. This means that two different logical addresses can refer to the same physical address (aliasing).

In non-overlapping method, address of source is totally different from address of destination. Therefore, we can directly transfer the data using MOVSB/MOVSW instruction for transferring the data.

The blocks are said to be overlapped if some of the memory locations are common for both the blocks.

Case I: If address in SI is greater than address in DI then start the data transfer from last memory location keeping DF=1.

Case II: If address in SI is less than address in DI, then start the data transfer from first memory location by keeping DF=0.

## Algorithm:

* + - 1. **Non-overlapped mode**

1. Initialize 2 memory blocks pointed by source and destination registers.
2. Initialize counter.
3. Move the contents pointed by source register to a register.
4. Increment address of source register.
5. Move the contents from register into location pointed by destination register.
6. Increment destination registers.
7. Decrement counter.
8. Repeat from steps 3 to step 6 until counter is 0.
9. End.

## Non-overlapped moed

1. Initialize 2 memory blockspointed by source and destination registers.
2. Initialize counter.
3. Move the contents pointedby source register [si+count] to a variable.
4. Decrement address of sourecregister.
5. Move the contents from vairable into location pointed by destination register [di+count]
6. Decrement destination registers.
7. Decrement counter.
8. Repeat from steps 3 to step 6 until counter is 0.
9. End.

## Input:

Array of number stored in location pointed by source and destination register Example: Array 1 db 10h, 20h, 30h, 40h, 33h, 0ffh, 44,55h, 23h, 45h

Array2 db 00h, 00h, 00h, 00h, 00h, 00h, 00,00h, 00h, 00h

## Output:

NON OVER LAPPED BLOCK TRANSFER

Array2 db 10h, 20h, 30h, 40h, 33h, 0ffh, 44,55h, 23h, 45h OVER LAPPED BLOCK TRANSFER

Array2 db 00h, 00h, 00h, 00h, 00h,10h, 20h, 30h, 40h, 33h, 0ffh, 44,55h, 23h, 45h

## Assignment Questions:

1. Memory: Even and odd banks
2. Address decoding techniques.
3. Comparison between memory mapped I/O and I/O mapped I/O.
4. Diagrammatic representation of the overlapped and non-overlapped block transfer
5. Comparison of overlapped and non-overlapped block transfer

## Oral Questions:

1. Specify all the memory addressing instruction
2. What is the use of the Direction flag in Block transfer?
3. What is use of Source and Destination Index in above program
4. What is the change in the contents of memory locations in overlapped and non- overlapped mode?
5. Which interrupt is used to terminate the program in 8086 kit?

**MPL Practical Oral Question Bank**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Sr No** | **B L** | **Questions** | **Oral 1** | **Oral 2 (improve ment)** | **Remark** |
| 1 | 1 | Specify all the memory addressing instruction |  |  |  |
| 2 | 1 | What is the use of the Direction flag in Block transfer? |  |  |  |
| 3 | 1 | What is use of Source and Destination Index in  above program |  |  |  |
| 4 | 1 | What is the change in the contents of memory  locations in overlapped and non- overlapped mode? |  |  |  |
| 5 | 1 | Which interrupt is used to terminate the  program in 8086 kit? |  |  |  |

**Sign of Student**

# Experiments No:08

**Name of Student**……………………………………………….. **Roll No**.:…… **Batch**:……. **Subject:**MPL **Department :**Computer Engg. **Class:** SE COMP

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Write-up** | **Correctness Program** | **Documentation** | **Timely Submission** | **Viva** | **Total** | **Dated Sign of Subject Teacher** |
| **4** | **4** | **4** | **4** | **4** | **20** |  |
|  |  |  |  |  |  |

Date of Performance:..................................... Date of Completion:....................................

**----------------------------------------------------------------------------------------------------------------------**

**Title:** HEX to BCD and BCD to HEX

**Problem Statement:** Write X86/64 ALP to perform multiplication of two 8-bit hexadecimal numbers. Use successive addition and add and shift method. (use of 64-bit registers is expected).

**Objective:** To learn and understand shift S rotate Instruction

**Outcomes:** On completion of this practical ,students will be able to

**C218.3:** Analyze and apply logic to demonstrate processor mode of operation

## Hardware Requirement: NA

**Software Requirement:** Ubuntu ,NASM etc.

## Theory Contents :

**Multiplying unsigned numbers**

Multiplying unsigned numbers in binary is quite easy. Recall that with 4 bit numbers we can represent numbers from 0 to 15. Multiplication can be performed done exactly as with decimal numbers, except that you have only two digits (0 and 1). The only number facts to remember are that 0\*1=0, and 1\*1=1 (this is the same as a logical "and").

Multiplication is different than addition in that multiplication of an n bit number by an m

bit number results in an n+m bit number. Let's take a look at an example where n=m=4 and the result is 8 bits

|  |  |
| --- | --- |
| **Decimal** | **Binary** |
|  | 1010 |
|  | x011 0 |
| 10 | 0000 |
| x6 | 1010 |
| 60 | 1010 |
|  | +0000 |

In this case the result was 7 bit, which can be extended to 8 bits by adding a 0 at the left. When multiplying larger numbers, the result will be 8 bits, with the leftmost set to

1, as shown.

|  |  |
| --- | --- |
| **Decimal** | **Binary** |
|  | 1101 |
|  | x111 |
| 13 | 0 |
| x14 | 0000 |
| 182 | 1101 |
|  | 1101 |
|  | +1101 |

As long as there are n+m bits for the result, there is no chance of overflow. For 2 four bit multiplicands, the largest possible product is 15\*15=225, which can be represented in 8 bits.

## Multiplying signed numbers

There are many methods to multiply 2's complement numbers. The easiest is to simply find the magnitude of the two multiplicands, multiply these together, and then use the original sign bits to determine the sign of the result. If the multiplicands had the same sign, the result must be positive, if they had different signs, the result is negative. Multiplication by zero is a special case (the result is always zero, with no sign bit).

Multiplication and division can be performed on signed or unsigned numbers. For unsigned numbers, MUL and DIV instructions are used, while for signed numbers IMUL and IDIV are used.

The format of the multiplication & division instruction does not specify the multiplicand as

it is implicitly specified depending on the size of Source Below Consider the example given

below 6 × 13 = 78

Sequential addition from row to row

0 1 1 0

× 1 1 0 1 Sum:

0 1 1 0 00000110

0 0 0 0 00000110

0 1 1 0 00011110

+ 0 1 1 0 01001110

0 1 0 0 1 1 1 0 Product

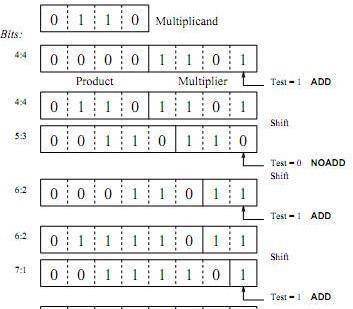
## 4.4.2 Sequential Shift/Add-Method

Method to avoid adder arrays

* shift register for partial product and multiplier with each cycle,

1. Partial product increases by one digit
2. Multiplier is reduced by one digit
   * MSBs of partial product and multiplicand are aligned in each cycle
   * not the multiplicand is shifted

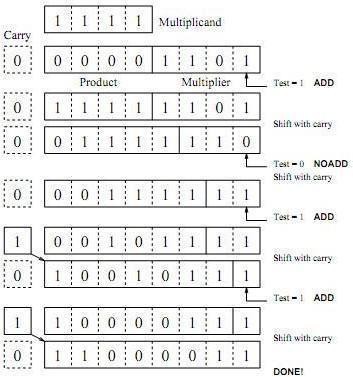
⇒ Partial product and multiplier are



## Successive Addition Method

* + - Consider that a byte is for 8 bit Numbers present on Second byte is present in BL Register.
    - We have to multiply the AL with the byte in BL
    - Multiply the number using Successive Addition Method.
    - In this method, one number is accepted and other number is taken as a counter.
    - The first number is added with itself, till the counter decrements to zero. Result is stored in DX register, Display the result, using display routine

15 x 13 =195



## Algorithm:

* + 1. Start
    2. Read multiplication& Multiplier for multiplication
    3. Display menu
       1. Successive

Addition b.Add & shift method c.Exit

* + 1. Read choice .if choice =a go to next step, if choice = b go to Sep 9, if choice =c go to step 15
    2. Assign sum =0 count= Multiplier
    3. Sum =sum Multiplicand
    4. Decrement count .if count >0 go to step 8
    5. Print sum & go to step 3
    6. Assign count =no of digit in multiplier, sum=0, shiftvar=0
    7. Shift right Multiplier by 1
    8. If carry flag set, sum =sum+ (Left shifted multiplicand by shiftvar)
    9. Shitvar =shitvar+1
    10. If count> 0 go to step 14
    11. Print sum & go to step 3
    12. Exit

## Instructions needed:

* + 1. MUL-Multiplication specified byte or word to word
    2. SHR- Shift logical right byte or word, MSB to LSB and to CF SHL-Shift logical left byte or word, LSB to MSB and to CF
    3. JMP-Unconditional jump to the specified location counter
    4. JC-Jumps if carry is generated
    5. JE/JZ-Jumps if equal or zero.

## Directive Recommended:

1. MACRO- Start of MACRO statement
2. ENDM-end

## Assignment Question:

* + 1. Explain MACRO with example. Justify where macro is suitable than procedure.
    2. Differentiate between PROC& MACRO
    3. What is the difference between a rotate & a shift instruction?

Explain with an appropriate diagram.

* + 1. Explain the difference between arithmetic shift & logical shift.
    2. Describe execution of CALL instruction.
    3. Solve 17 & 18eExample on Paper Manually for Both Methods using Program Logic ?

## Oral Question:

* + 1. Explain the Instruction used in the program
    2. With example explain Add and Shift Multiplication
    3. With example explain Successive Addition Multiplication
    4. Suggest the alternative instructions for JE/JZ (Logic of program should not change)
    5. Difference in multiplication by MUL and IMUL

**Conclusion:**In this way we studied shifting operation and multiplication using successive addition and add-shift method.

**MPL Practical Oral Question Bank**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Sr No** | **B L** | **Questions** | **Oral 1** | **Oral 2 (improve ment)** | **Remark** |
| 1 | 2 | Explain the Instruction used in the program?. |  |  |  |
| 2 | 2 | Explain the difference between arithmetic shift & logical shift.. |  |  |  |
| 3 | 1 | With example explain Successive Addition  Multiplication? |  |  |  |
| 4 | 2 | Difference in multiplication by MUL and IMUL? |  |  |  |
| 5 | 2 | Explain conversion of Ascii code to Hex code |  |  |  |
| 6 | 2 | Explain conversion of Hex code to Ascii code  ? |  |  |  |

**Sign of Student**

# Experiments No:09

**Name of Student**……………………………………………….. **Roll No**.:…… **Batch**:……. **Subject:**MPL **Department :**Computer Engg. **Class:** SE COMP

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Write-up** | **Correctness Program** | **Documentation** | **Timely Submission** | **Viva** | **Total** | **Dated Sign of Subject Teacher** |
| **4** | **4** | **4** | **4** | **4** | **20** |  |
|  |  |  |  |  |  |

Date of Performance:..................................... Date of Completion:....................................

**----------------------------------------------------------------------------------------------------------------------**

**Title:** To find the number of lines,blank spaces.

**Problem Statement:** Write X86 ALP to find, a) Number of Blank spaces b) Number of lines c) Occurrence of a particular character. Accept the data from the text file. The text file has to be accessed during Program\_1 execution and write FAR PROCEDURES in Program\_2 for the rest of the processing. Use of PUBLIC and EXTERN directives is mandatory.

## Objective:

* To understand assembly language programming instruction set To understand different assembler directives with example
* To apply instruction set for implementing X86/64 bit assembly language programs

**Outcomes:** On completion of this practical ,students will be able to

**C218.1:** Understand and apply various addressing modes and instruction set to implement assembly language programs

## Hardware Requirement: NA

**Software Requirement: OS:**Ubuntu Assembler: NASM version 2.10.07 Linker: ld

## Theory Contents :Near Procedure Far Procedure

**Explanation:**

Open given text file. Read the content of file and store it in a buffer. Call far procedure which will calculate the number of blank spaces, lines and occurrence of a particular character from the buffer.

Assembler Directives Used: (Explain it by your own)

## Extern:

 **Global:**

\_

\_

## Instructions:

**Input: Text File Output: Display of-**

1. Number of Blank spaces
2. Number of lines
3. Occurrence of a particular character.

## Main Algorithm:

A1: Algorithm for program\_1

* 1. Start
  2. Initialize all the sections needed in programming
  3. Display “Enter file name” message using Print macro expansion
  4. Accept file name using Accept macro and store in filename buffer
  5. Display “Enter character to search” message with the expansion of Print macro
  6. Read character using Accept macro expansion
  7. Open file using fopen macro
  8. Compare RAX with -1H if equal then display error message “Error in Opening File” with Print macro expansion else go to step ix
  9. Read content of opened file in buffer
  10. Store file length in abuf\_len
  11. Call far\_procedure
  12. xii. Stop

## Macros: Macro 1

1. Name : Print
2. Purpose: to display the messages by replacing the whole code by simple macro declaration
3. I/P: sys\_write call Number i.e eax= , File descriptor (for Standard output ebx=1), Buffer Address in rsi, and length of Buffer in rdx. Then Call int 80h.

## Macro 2

* 1. Name : Accept
  2. Purpose: to accept input from the user by replacing the whole code by simple macro Declaration
  3. I/P: sys\_read call Number i.e eax= , File descriptor (for Standard input rdi=0), Buffer Address in ecx, and length of Buffer in edx then call int 80h

## Macro 3

1. Name : fopen
2. Purpose: to open a file in given mode
3. I/P: sys\_write call Number i.e eax= File name in ebx, Mode of file in ecx (R=0,W=1,RW=2), and file permission in edx then call int 80h.

## Macro 4

1. Name: fread
2. Purpose: to read the content of file

4. I/P: sys\_read call Number i.e eax=3, File descriptor in ebx , Buffer Address in ecx, and Length of Buffer in edx. Then Call int 80h

## Macro 5

1. Name: fclose
2. Purpose: to close opened file
3. I/P: sys\_read call Number i.e eax=6, File handler in ebx. Then Call int 80h.

## Procedure: 1

1. Name: far\_procedure
2. Purpose: to count 1. Number of Blank spaces 2. Number of lines 3. Occurrence of a particular character.
3. I/P : Content stored in buffer
4. Algorithm for Procedures
   1. Start
   2. Load effective address of buffer in RSI
   3. Load content of abuf\_len in ECX
   4. Load content of char in BL
   5. Move value of RSI in AL
   6. Compare AL with 20H (ASCII value of space) if not equal then go to step
5. else increment content of scount vii. Compare AL with 10H (ASCII value of line) if not equal then go to step
6. else increment content of ncount viii. Compare AL with BL if not equal then go to step ix else increment content of ccount
7. Increment RSI
8. Repeat from step

vi if RCX is not equal to zero

1. Display “Number of space” message with the expansion of Print macro.
2. Move content of scount in EBX
3. Call display8num procedure
4. Display “Number of lines” message with the expansion of Print macro.
5. Move content of ncount in EBX
6. Call display8num procedure
7. Display “Number of Occurrence of Character” message with the expansion of Print macro. xviii. Move content of ccount in EBX
8. Call display8num procedure
9. Ret
10. Stop

## Procedure: 2

* 1. Name: display8num
  2. Purpose: Convert

2 digit hex number into 2 ASCII character to display Positive and Negative Number count on Standard output (stdout).

* 1. I/P : bl=pcnt/ncnt
  2. Algorithm for Procedures
     1. Move RSI with effective address of dispbuff.
     2. Initialize rcx by 2
     3. Rotate the contents of bl to the left side by 4 bits.
     4. Move the contents of bl into al
     5. And the contents of al with 0fH
     6. Compare al with 09h
        1. If al is below or equal then add 30H in al
        2. Else add 37H in al
     7. Move the content of al into memory pointed by ESI
     8. Increment ESI i. Repeat from step c to h until rcx is not equal to 0

## Conclusions:

Assembly Level Program to find,

1. Number of Blank spaces
2. Number of lines
3. Occurrence of a particular character is assembled and executed successfully.

## Assignment Questions

Q1 Explain ‘EXTERN’ and ‘EXTRN’ directive. Q2 Explain ‘GLOBAL’ and ‘PUBLIC’ directive.

Q3 How far procedure is called in masm?

Q4 How you counted the occurrences of character in the given string? Explain logic. Q5 How you assembled and linked the source files?

Q6 Explain FAR call and return ?

Q7 Explain difference between “near” and “far” procedure. Q8 Write Down the ASCII of Space, Enter ?

**MPL Practical Oral Question Bank**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Sr No** | **B L** | **Questions** | **Oral 1** | **Oral 2 (improve ment)** | **Remark** |
| 1 | 1 | What is‘GLOBAL’ and ‘PUBLIC’ directive. ? |  |  |  |
| 2 | 1 | What is difference between Far and Near Procedure? |  |  |  |
| 3 | 1 | What is use Far Procedure? |  |  |  |

**Sign of Student**

# Experiments No:10

**Name of Student**……………………………………………….. **Roll No**.:…… **Batch**:……. **Subject:**MPL **Department :**Computer Engg. **Class:** SE COMP

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Write-up** | **Correctness Program** | **Documentation** | **Timely Submission** | **Viva** | **Total** | **Dated Sign of Subject Teacher** |
| **4** | **4** | **4** | **4** | **4** | **20** |  |
|  |  |  |  |  |  |

Date of Performance:..................................... Date of Completion:....................................

**----------------------------------------------------------------------------------------------------------------------**

**Title:**Find factorial of a given integer number.

**Problem Statement:** Write x86 ALP to find the factorial of a given integer number on a command line by using recursion. Explicit stack manipulation is expected in the code.

## Objective:

* To understand assembly language programming instruction set To understand different assembler directives with example
* To apply instruction set for implementing X86/64 bit assembly language programs

**Outcomes:** On completion of this practical ,students will be able to

**C218.1:** Understand and apply various addressing modes and instruction set to implement assembly language programs

## Hardware Requirement: NA

**Software Requirement: OS:**Ubuntu Assembler: NASM version 2.10.07 Linker: ld

**Theory Contents :** A recursive procedure is one that calls itself. There are two kind of recursion: direct and indirect. In direct recursion, the procedure calls itself and in indirect recursion, the first procedure calls a second procedure, which in turn calls the first procedure.

Recursion could be observed in numerous mathematical algorithms. For example,

consider the case of calculating the factorial of a number. Factorial of a number is given by the equation −

Fact (n) = n \* fact (n-1) for n > 0

For example: factorial of 5 is 1 x 2 x 3 x 4 x 5 = 5 x factorial of 4 and this can be a good example of showing a recursive procedure. Every recursive algorithm must have an ending condition, i.e., the recursive calling of the program should be stopped when a condition is fulfilled. In the case of factorial algorithm, the end condition is reached when n is 0.

Recursion occurs when a procedure calls itself. The following for example is a recursive procedure:

Recursive proc callRecursive ret

Recursive endp

Of course the CPU will never execute the ret instruction at the end of this procedure. Upon entry into Recursive this procedure will immediately call itself again and control will never pass to the ret instruction. In this particular case run away recursion results in an infinite loop.

In many respects recursion is very similar to iteration (that is the repetitive execution of a loop). The following code also produces an infinite loop:

Recursive proc jmp Recursive ret

Recursive endp

There is however one major difference between these two implementations. The former version of Recursive pushes a return address onto the stack with each invocation of the subroutine. This does not happen in the example immediately above (since the jmp instruction does not affect the stack).

Like a looping structure recursion requires a termination condition in order to stop infinite recursion. Recursive could be rewritten with a termination condition as follows:

Recursive

proc dec ax jzQuitRecurse

call Recursive QuitRecurse:

Ret Recursiveendp

This modification to the routine causes Recursive to call itself the number of times appearing in the ax register. On each call Recursive decrements the ax register by one and calls itself again. Eventually Recursive decrements ax to zero and returns. Once this happens the CPU executes a string of ret instructions until control returns to the original call to Recursive.

So far however there hasn't been a real need for recursion. After all you could efficiently code this procedure as follows:

Recursive proc RepeatAgain:

dec ax

jnzRepeatAgain ret

Recursive endp

Both examples would repeat the body of the procedure the number of times passed in the ax register. As it turns out there are only a few recursive algorithms that you cannot implement in an iterative fashion. However many recursively implemented algorithms are more efficient than their iterative counterparts and most of the time the recursive form of the algorithm is much easier to understand.

## Assignment Questions: -

1. State the difference between Iteration and Recursion?
2. By using which instruction recursion method pushes a return address onto the stack?
3. State the Difference between CALL and JMP instruction.

**MPL Practical Oral Question Bank**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Sr No** | **B L** | **Questions** | **Oral 1** | **Oral 2 (improve ment)** | **Remark** |
| 1 | 1 | What is the difference between Iteration and  Recursion? |  |  |  |
| 2 | 1 | What is the use for stack in Recursion? |  |  |  |
| 3 | 1 | What is difference between CALL and JMP  instruction.? |  |  |  |

**Sign of Student**