EXPERIMENT NO: 01

AIM: : 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.

OBJECTIVES:

- 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

ENVIRONMENT:

- Operating System: 64-bit Open source Linux or its derivative.
- Programming Tools: Preferably using Linux equivalent or MASM/TASM/NASM/FASM.
- Text Editor: geditor

THEORY:

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 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 isn9t 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 don9t 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 you9re 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 don9t say what values are to be present in the

buffer. There is 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, that9s about it.
All global labels must be declared in the .text section, or the labels cannot be 88seen99 outside your program by the Linux linker or the Linux loader. Let9s 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 that9s 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 don9t 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 ebx as 00

STEP 5: Store element in array.

STEP 6: Move edx by 17.

STEP 7: Add 17 to ebx.

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 ebx as

00STEP 11: Display element of array.

STEP 12: Move edx by 17.

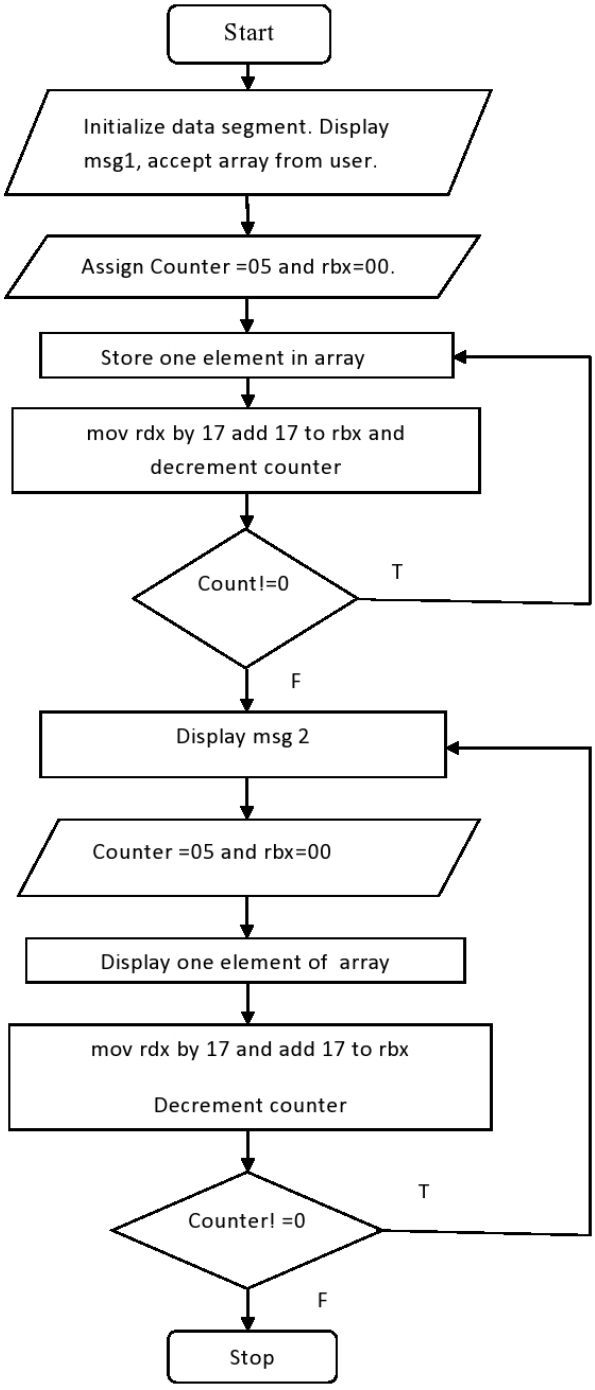
STEP 13: Add 17 to ebx.

STEP 14: Decrement Counter.

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

STEP 16: Stop

FLOWCHART:



CONCLUSION:

In this practical session we learnt how to write assembly language program and Accept and display array in assembly language.

EXPERIMENT NO: 02

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

OBJECTIVES:

- 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

ENVIRONMENT:

- Operating System: 64-bit Open source Linux or its derivative.
- Programming Tools: Preferably using Linux equivalent or MASM/TASM/NASM/FASM.
- Text Editor: geditor

THEORY:

MACRO:

Writing a macro is another way of ensuring modular programming in assembly language.

- A macro is a sequence of instructions, assigned by a name and could be used anywhere in the program.
- In NASM, macros are defined with **%macro** and **%endmacro** directives.
- The macro begins with the %macro directive and ends with the %endmacro directive.

The Syntax for macro definition −

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

Where, *number_of_params* specifies the number parameters, *macro_name* specifies the name of the macro.

The macro is invoked by using the macro name along with the necessary parameters. When you need to use some sequence of instructions many times in a program, you can put those instructions in a macro and use it instead of writing the instructions all the time.

PROCEDURE:

Procedures or subroutines are very important in assembly language, as the assembly language programs tend to be large in size. Procedures are identified by a name. Following this name, the body of the procedure is described which performs a well-defined job. End of the procedure is indicated by a return statement.

Syntax

Following is the syntax to define a procedure −

```
proc_name:
   procedure body
   ...
   ret
```

The procedure is called from another function by using the CALL instruction. The CALL instruction should have the name of the called procedure as an argument as shown below −

CALL proc_name

The called procedure returns the control to the calling procedure by using the RET instruction.

LIST OF INTERRRUPTS USED: NA

LIST OF ASSEMBLER DIRECTIVES USED: EQU, PROC, GLOBAL, DB,

LIST OF MACROS USED: DISPMSG

LIST OF PROCEDURES USED: DISPLAY

ALGORITHM:

INPUT: String

OUTPUT: Length of String in hex

STEP 1: Start.

STEP 2: Initialize data section.

STEP 3: Display msg1 on monitor

STEP 4: accept string from user and store it in esi Register (Its length gets stored in eax register by default).

STEP 5: Display the result using <display= procedure. Load length of string in data register.

STEP 6. Take counter as 16 int cnt variable

STEP 7: move address of <result= variable intoedi.

STEP 8: Rotate left ebx register by 4 bit.

STEP 9: Move bl into al.

STEP 10: And al with 0fh

STEP 11: Compare al with 09h

STEP 12: If greater add 37h into al

STEP 13: else add 30h into al

STEP 14: Move al into memory location pointed by

ediSTEP 14: Increment edi

STEP 15: Loop the statement till counter value becomes zero

STEP 16: Call macro dispmsg and pass result variable and length to it. It will print length of string.

STEP 17: Return from procedure

STEP 18: Stop

FLOWCHART: You should write flowchart by yourself by referring algorithm

CONCLUSION:

In this practical session, we learnt how to display any number on monitor.

EXPERIMENT NO: 03

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

OBJECTIVES:

- 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

ENVIRONMENT:

- Operating System: 64-bit Open source Linux or its derivative.
- Programming Tools: Preferably using Linux equivalent or MASM/TASM/NASM/FASM.
- Text Editor: geditor

THEORY:

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.

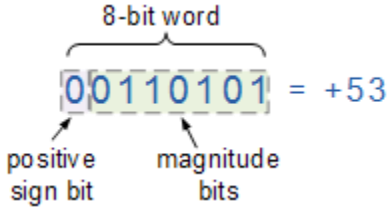
But how do we represent signed binary numbers if all we have is a bunch of one9s and zero9s. We know that binary digits, or bits only have two values, either a <1= or a <0= and conveniently for us, a sign also has only two values, being a <+= or a <–<.

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.

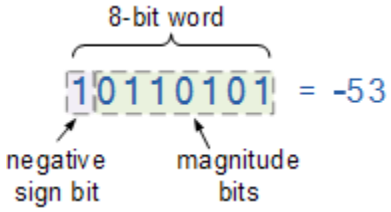
For signed binary numbers the most significant bit (MSB) is used as the sign bit. If the sign bit is <0=, this means the number is positive in value. If the sign bit is <1=, then the number is negative in value. The remaining bits in the number are used to represent the magnitude of the binary number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

LIST OF PROCEDURES USED: disp8num

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

FLOWCHART: Do by yourself

CONCLUSION: Do by yourself

EXP NO: 06

AIM: 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).

OBJECTIVES:

- 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

ENVIRONMENT:

- Operating System: 64-bit Open source Linux or its derivative.
- Programming Tools: Preferably using Linux equivalent or MASM/TASM/NASM/FASM.
- Text Editor: geditor

THEORY:

Hexadecimal Number System:

The <Hexadecimal= or simply <Hex= numbering system uses the **Base of 16** system and are a popular choice for representing long binary values because their format is quite compact and much easier to understand compared to the long binary strings of 19s and 09s.
Being a Base-16 system, the hexadecimal numbering system therefore uses 16 (sixteen) different digits with a combination of numbers from 0 to 9 and A to F.
Hexadecimal Numbers is a more complex system than using just binary or decimal and is mainly used when dealing with computers and memory address locations.

Binary Coded Decimal(BCD) Number System:

Binary coded decimal (BCD) is a system of writing numerals that assigns a four-digit binary code to each digit 0 through 9 in a decimal (base-10) numeral. The four-bit BCD code for any particular single base-10 digit is its representation in binary notation, as follows:

0 = 0000

1 = 0001

2 = 0010

3 = 0011

4 = 0100

5 = 0101

6 = 0110

7 = 0111

8 = 1000

9 = 1001

Numbers larger than 9, having two or more digits in the decimal system, are expressed digit by digit. For example, the BCD rendition of the base-10 number 1895 is

0001 1000 1001 0101

The binary equivalents of 1, 8, 9, and 5, always in a four-digit format, go from left to right.

The BCD representation of a number is not the same, in general, as its simple binary representation. In binary form, for example, the decimal quantity 1895 appears as

11101100111

Decimal Number	4-bit Binary Number	Hexadecimal Number	BCD Number
0	0000	0	0000 0000
1	0001	1	0000 0001
2	0010	2	0000 0010
3	0011	3	0000 0011
4	0100	4	0000 0100
5	0101	5	0000 0101
6	0110	6	0000 0110
7	0111	7	0000 0111
8	1000	8	0000 1000
9	1001	9	0000 1001
10	1010	A	0001 0000
11	1011	B	0001 0001
12	1100	C	0001 0010
13	1101	D	0001 0011
14	1110	E	0001 0100
15	1111	F	0001 0101

16	0001 0000	10 (1+0)	0001 0110
17	0001 0001	11 (1+1)	0001 0111

HEX to BCD
Divide FFFF by 10 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 RAX = 65535 which is 1111 1111 1111 1111 and when we print this it is represented as FFFF.

LIST OF INTERRRUPTS USED:

LIST OF ASSEMBLER DIRECTIVES USED:

LIST OF MACROS USED:

LIST OF PROCEDURES USED:

ALGORITHM:

STEP 1: Start
STEP 2: Initialize data section.
STEP 3: Using Macro display the Menu for HEX to BCD, BCD to HEX and exit. Accept the choice from user.
STEP 4: If choice = 1, call procedure for HEX to BCD conversion.
STEP 5: If choice = 2, call procedure for BCD to HEX conversion.
STEP 6: If choice = 3, terminate the program.

Algorithm for procedure for HEX to BCD conversion:

STEP 7: Accept 4-digit hex number from user.
STEP 8: Make count in RCX register 0.
STEP 9: Move accepted hex number in BX to AX.

STEP 10: Move base of Decimal number that is 10 in BX.
STEP 11: Move zero in DX.
STEP 12: Divide accepted hex number by 10. Remainder will return in DX.
STEP 13: Push remainder in DX on to stack.
STEP 14: Increment RCX counter.
STEP 15: Check whether AX contents are zero.
STEP 16: If it is not zero then go to step 5.
STEP 17: If AX contents are zero then pop remainders in stack in RDX.
STEP 18: Add 30 to get the BCD number.
STEP 19: Increment RDI for next digit and go to step 11.

Algorithm for procedure for BCD to HEX:

STEP 1: Accept 5-digit BCD number from user.
STEP 2: Take count RCX equal to 05.
STEP 3: Move 0A that is 10 in EBX.
STEP 4: Move zero in RDX register.
STEP 5: Multiply EBX with contents in EAX.
STEP 6: Move contents at RSI that is number accepted from user to DL.
STEP 7: Subtract 30 from DL.
STEP 8: Add contents of RDX to RAX and result will be in RAX.
STEP 9: Increment RSI for next digit and go to step 4 and repeat till RCX becomes zero.
STEP 10: Move result in EAX to EBX and call display procedure.

FLOWCHART:

Assignment No-3

AIM: Write an X86/64 ALP to find the largest of given Byte/Word/Dword/64-bit numbers

OBJECTIVES:

- 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

ENVIRONMENT:

- Operating System: 64-bit Open source Linux or its derivative.
- Programming Tools: Preferably using Linux equivalent or MASM/TASM/NASM/FASM.
- Text Editor: geditor

THEORY:

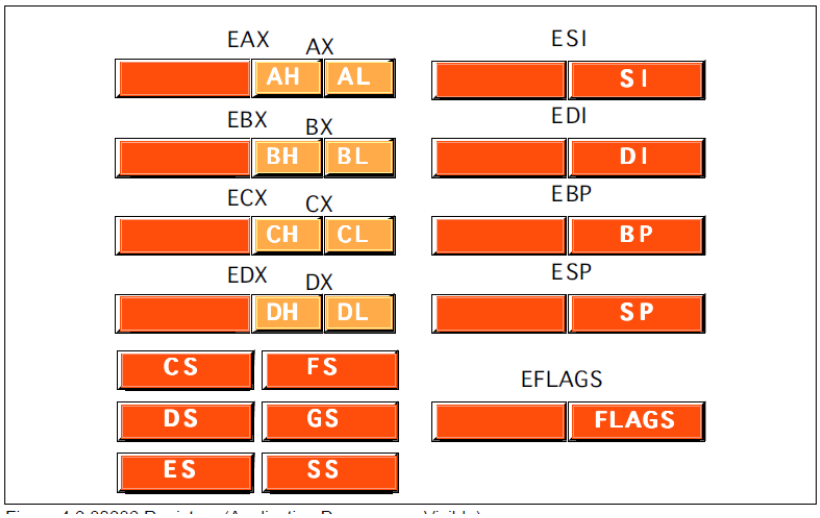
Datatype in 80386:

Datatypes of 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.

1. Register addressing mode:	MOV EAX, EDX
2. Immediate Addressing modes:	MOV ECX, 20305060H
3. Direct Addressing mode:	MOV AX, [1897 H]
4. Register Indirect Addressing mode	MOV EBX, [ECX]
5. Based Mode	MOV ESI, [EAX+23H]
6. Index Mode	SUB COUNT [EDI], EAX
7. Scaled Index Mode	MOV [ESI*8], ECX
8. Based Indexed Mode	MOV ESI, [ECX][EBX]
9. Based Index Mode with displacement	EA=EBX+EBP+1245678H
10. Based Scaled Index Mode with displacement	MOV [EBX*8] [ECX+5678H], ECX
11. String Addressing modes:	
12. Implied Addressing modes:	

ALGORITHM:

FLOWCHART:

EXP NO: 06

AIM: 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.

OBJECTIVES:

- 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

ENVIRONMENT:

- Operating System: 64-bit Open source Linux or its derivative.
- Programming Tools: Preferably using Linux equivalent or MASM/TASM/NASM/FASM.
- Text Editor: geditor

THEORY:

ALGORITHM:

FLOWCHART:

EXPERIMENT NO. 07

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

OBJECTIVES:

- 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

Explanation:

- Consider that a block of data of N bytes is present at source location. Now this block of N bytes is to be moved from source location to a destination location.
- Let the number of bytes N = 05.
- We will have to initialize this as count.
- We know that source address is in the ESI register and destination address is in the EDI register.
- For block transfer without string instruction, move contents at ESI to accumulator and from accumulator to memory location of EDI and increment ESI and EDI for next content transfer.
- For block transfer with string instruction, clear the direction flag. Move the data from source location to the destination location using string instruction.

Mathematical Model

S= {s,e,a,b, Fme, FF,MEM shared}

1. S : S is distinct start of program
   S -> Global _start
   start:
2. e: e is distinct end of program
   mov eax,1

mov ebx ,0

int 80h

a: It is input of program

a= {source block, srcblock, choice}

b: It is an output of program

b= {destination block, destblock, contents}

Fme:This is friend function used in program.

We have the following functions in friend function

III. Numascii

IV. Convert

3. FF: The system extern invalid state if

-wrong position for overlapping

-register contents which differ after calling procedure

-invalid operand specifications

-invalid addressing modes

4. MEM shared: This is memory shared in the program. We have used the shared memory in the following.

IV. Accept Procedure

V. Display Procedure

VI. Display Block Procedure

Instructions needed:

1. **MOVSB**-This is a string instruction and it moves string byte from source to destination.

2. **REP-** This is prefix that are applied to string operation. Each prefix cause the string instruction that follows to be repeated the number of times indicated in the count register.

3. **CLD-**Clear Direction flag. ESI and EDI will be incremented and DF = 0

4. **STD-** Set Direction flag. ESI and EDI will be incremented and DF = 1

5. **ROL**-Rotates bits of byte or word left.

6. **AND**-AND each bit in a byte or word with corresponding bit in another byte or word.

7. **INC**-Increments specified byte/word by1.

8. **DEC**-Decrements specified byte/word by1.

9. **JNZ**-Jumps if not equal to Zero.

10. **JNC**-Jumps if no carry is generated.

11. **CMP**-Compares to specified bytes or words.

12. **JBE**-Jumps if below or equal.

13. **ADD**-Adds specified byte to byte or word to word.

14. **CALL**-Transfers the control from calling program to procedure.

15. **RET**-Return from where call is made.

Algorithm:

1. Start
2. Initialize data section.
3. Initialize the count, source block and destination block.
4. Using Macro display the Menu for block transfer without string instruction, block transfer with string instruction and exit.
5. If choice = 1, call procedure for block transfer without string instruction.
6. If choice = 2, call procedure for block transfer with string instruction.
7. If choice = 3, terminate the program.

Algorithm for procedure for non overlapped block transfer without string instruction:

1. Initialize ESI and EDI with source and destination address.
2. Move count in ECX register.
3. Move contents at ESI to accumulator and from accumulator to memory location of EDI.
4. Increment ESI and EDI to transfer next content.
5. Repeat procedure till count becomes zero.

CONCLUSION: Write yourself

EXP NO: 08

AIM: Write X86/64 ALP to perform overlapped block transfer with string specific instructions Block containing data can be defined in the data segment.

OBJECTIVES:

- 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

Explanation:

- Consider that a block of data of N bytes is present at source location. Now this block of N bytes is to be moved from source location to a destination location.
- Let the number of bytes N = 05.
- We will have to initialize this as count.
- Overlap the source block and destination block.
- We know that source address is in the ESI register and destination address is in the EDI register.
- For block transfer without string instruction, move contents at ESI to accumulator and from accumulator to memory location of EDI and decrement ESI and EDI for next content transfer.
- For block transfer with string instruction, set the direction flag. Move the data from source location to the destination location using string instruction.

Mathematical Model

S= {s,e,a,b, Fme, FF,MEM shared}

1. S : S is distinct start of program
   S -> Global _start
   _start:
   e: e is distinct end of program

mov eax,1

movebx ,0

int 80h

a: It is input of program

a= {source block, srcblock,choice}

b: It is an output of program

b= {destination block, destblock, contents}

Fme: This is friend function used in program.

We have the following functions in friend function

V. Numascii

VI. Convert

FF: The system extern invalid state if

-wrong position for overlapping

-register contents which differ after calling procedure

-invalid operand specifications

-invalid addressing modes

MEM shared: This is memory shared in the program. We have used the shared memory in the following.

VII. Accept Procedure

VIII. Display Procedure

IX. Display Block Procedure

Instructions needed:

1. **MOVSB**-This is a string instruction and it moves string byte from source to destination.
2. **REP-** This is prefix that is applied to string operation. Each prefix cause the string instruction that follows to be repeated the number of times indicated in the count register.
3. **CLD-**Clear Direction flag. ESI and EDI will be incremented and DF = 0
4. **STD-** Set Direction flag. ESI and EDI will be decremented and DF = 1
5. **ROL**-Rotates bits of byte or word left.
6. **AND**-AND each bit in a byte or word with corresponding bit in another byte or word.

7. **INC**-Increments specified byte/word by1.
8. **DEC**-Decrements specified byte/word by1.
9. **JNZ**-Jumps if not equal to Zero.
10. **JNC**-Jumps if no carry is generated.
11. **CMP**-Compares to specified bytes or words.
12. **JBE**-Jumps if below or equal.
13. **ADD**-Adds specified byte to byte or word to word.
14. **CALL**-Transfers the control from calling program to procedure.
15. **RET**-Return from where call is made.

Algorithm:

1. Start
2. Initialize data section.
3. Initialize the count, source block and destination block.
4. Using Macro display the Menu for block transfer without string instruction, block transfer with string instruction and exit.
5. If choice = 1, call procedure for block transfer without string instruction.
6. If choice = 2, call procedure for block transfer with string instruction.
7. If choice = 3, terminate the program.

Algorithm for procedure for overlapped block transfer with string instruction:

8. Initialize ESI and EDI with source and destination address.
9. Move count in ECX register.
10. Move source block's and destination block's last content address in ESI and EDI.
11. Set the direction flag.
12. Move the data from source location to the destination location using string instruction.
13. Repeat string instruction the number of times indicated in the count register.

CONCLUSION: Write yourself

EXP NO: 9

AIM: 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).

OBJECTIVES:

- 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

Successive Addition Method:

Assuming that MUL instruction is not available in the instruction set of 80386, write a program in assembly language of 80386 to simulate the MUL instruction. Assuming that two digits are available in EAX and BL registers. Use successive addition method.

Explanation:

- Consider that a byte is present in the EAX register and second byte is present in theBL register.
- We have to multiply the byte in EAX with the byte in BL.
- We will multiply the numbers using successive addition method.
- In successive addition 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 EBX register. Display the result, using display routine.

For example : AL = 12 H, BL = 10 H

Result = 12H + 12H + 12H + 12H + 12H + 12H + 12H + 12H + 12H + 12H

Result = 0120 H

Mathematical Modelling

S= {s,e,a,b, Fme, FF,MEM shared}

1. S : S is distinct start of program

S -> Global _start

_start:

2. e: e is distinct end of program

mov eax,1

movebx ,0

int 80h

3. a: It is input of program

a-> 1) First number for multiplication

2) Second number for multiplication

4. b: It is an output of program

b-> 1) Multiplication of two numbers by successive addition method

2)Multiplication of two numbers by add and shift method

5. Fme: This is friend function used in program.

We have the following functions in friend function

I. Numascii

II. Convert

6. FF: The system enters invalid state if

-numbers are not entered properly.

-numbers entered by user are larger than buffer size.

7. MEM shared: This is memory shared in the program. We have used the shared memory in the following.

I. Accept Procedure

II. Display Procedure

Algorithm:

1. Start
2. Initialize data section.
3. Get the first number.
4. Get the second number as counter.
5. Initialize result = 0.
6. Result = Result + First number.
7. Decrement counter

8. If count is not equal to 0, go to step V.

9. Display the result.

10. Stop.

CONCLUSION: Write yourself

EXP NO: 10

AIM: Study Assignment: Motherboards are complex. Break them down, component by component, and understand how they work. Choosing a motherboard is a hugely important part of building a PC. Study- Block diagram, Processor Socket, Expansion Slots, SATA, RAM, Form Factor, BIOS, Internal Connectors, External Ports, Peripherals and Data Transfer, Display, Audio, Networking, Over clocking, and Cooling.

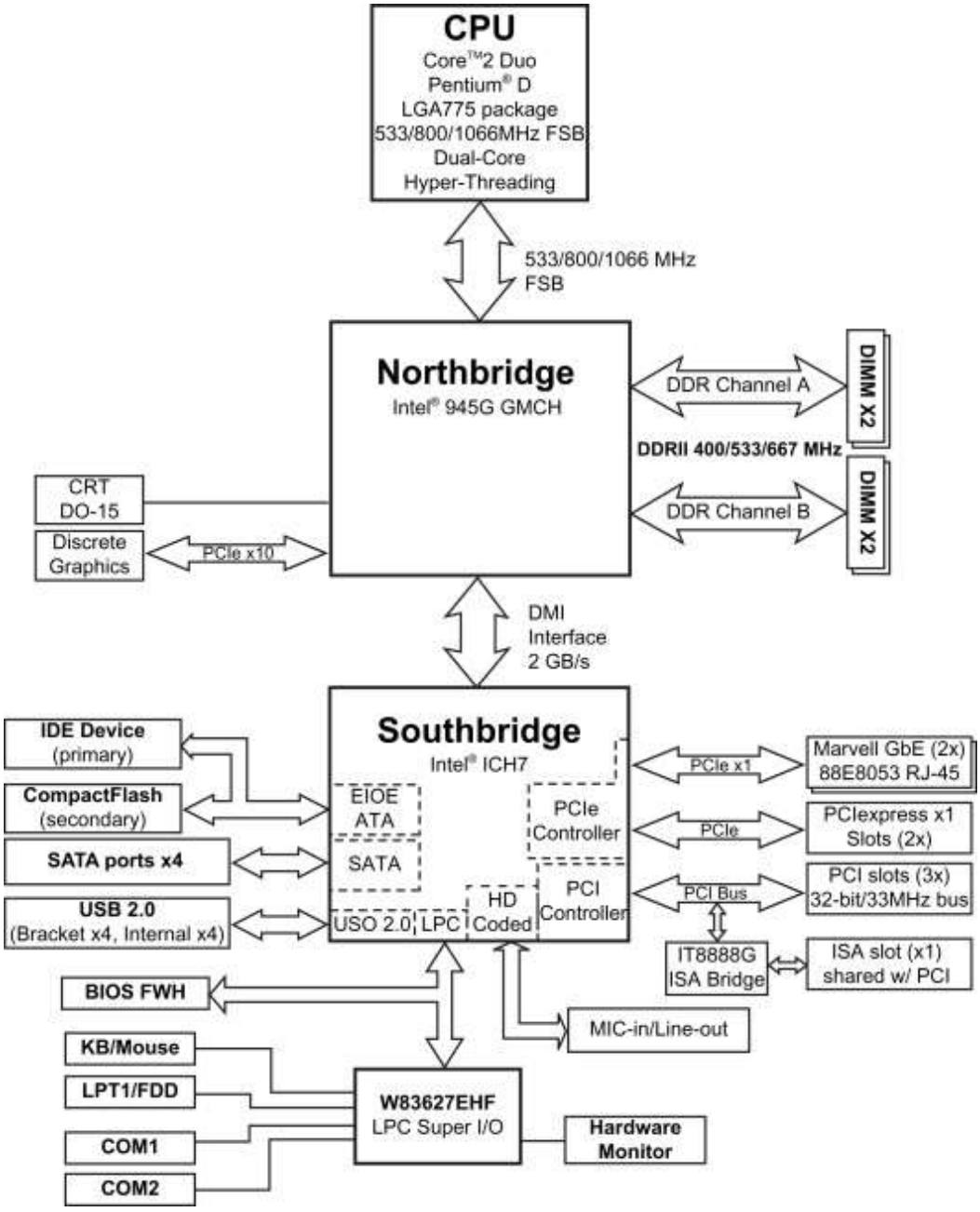
OBJECTIVES:

- To understand Motherboard of PC.

THEORY:

Write details about Processor Socket, Expansion Slots, SATA, RAM, Form Factor, BIOS, Internal Connectors, External Ports, Peripherals and Data Transfer, Display, Audio, Networking, Over clocking, and Cooling of Motherboard of PC.

CONCLUSION: Write yourself


CPU
Core™2 Duo
Pentium® D
LGA775 package
533/800/1066MHz FSB
Dual-Core
Hyper-Threading
533/800/1066 MHz
FSB
Northbridge
Intel® 945G GMCH
DDR Channel A
DIMM X2
DDRII 400/533/667 MHz
DDR Channel B
DIMM X2
CRT
DO-15
Discrete
Graphics
PCIe x10
DMI
Interface
2 GB/s
Southbridge
Intel® ICH7
IDE Device
(primary)
CompactFlash
(secondary)
SATA ports x4
USB 2.0
(Bracket x4, Internal x4)
EIOE
ATA
SATA
USO 2.0
LPC
HD
Coded
PCIe
Controller
PCI
Controller
PCIe x1
Marvell GbE (2x)
88E8053 RJ-45
PCIe
PCIexpress x1
Slots (2x)
PCI Bus
PCI slots (3x)
32-bit/33MHz bus
IT8888G
ISA Bridge
ISA slot (x1)
shared w/ PCI
BIOS FWH
MIC-in/Line-out
KB/Mouse
LPT1/FDD
COM1
COM2
W83627EHF
LPC Super I/O
Hardware
Monitor