Study Assignment

Title: Introduction to Assembly language programming

Aim: To study Assembler, linker, masm, tasm, & assembly language programming x86 instruction set.

Objective: To be familiar with the format of assembly language programming structured instruction.

Theory:

Assembly language: An assembly language programing is a low level programming language for computer, microprocessor, microcontroller & other integrated circuit. It implement a symbolic representation of the binary machine and other constants needed to program a CPU architecture. This representation is usually defined by the hardware manufacturer.

Assembler: It is a system program which converts the assembly language program instruction into machine executable instruction. For example: Microsoft Macro Assembler (MASM), Open source Netwide Assembler (NASM)

MASM: The Microsoft macro assembler is an x86 assembler for MS-DOS & Microsoft Windows. It supports a wide variety of macro facilities & structured programming idioms including high level programme/function for looping & procedures.

TASM. Turbo Assembler (TASM) is a x86 assembler package developed by Borland. It is used with Borland's high level language compiler, such as Turbo Pascal & Turbo C

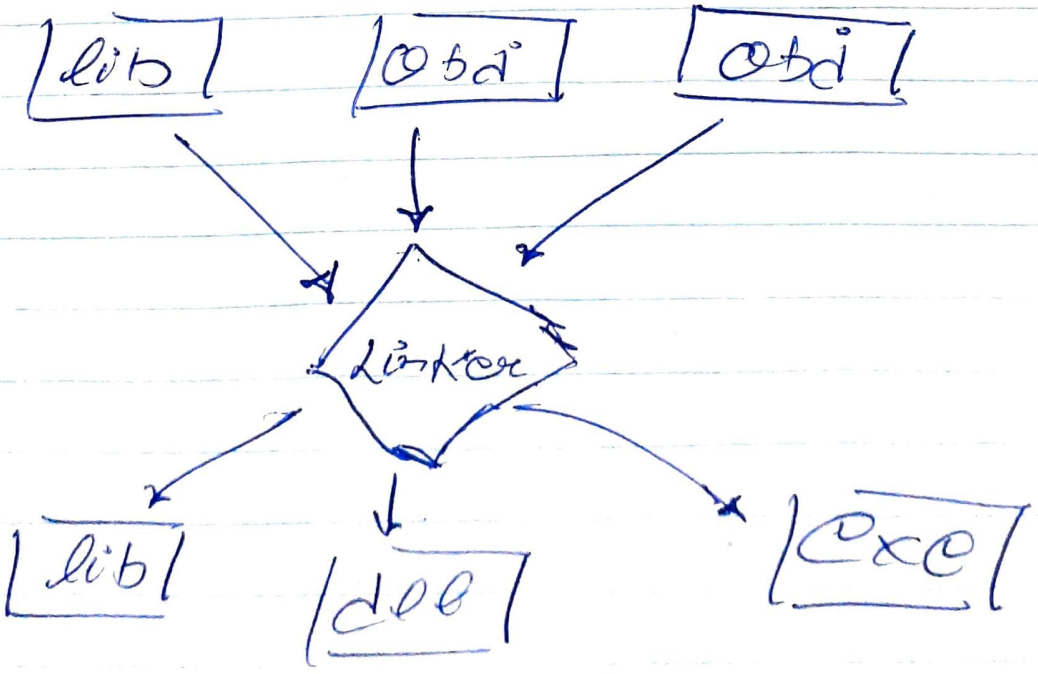
NASM: The Netwide Assembly (NASM) is an assembler & disassembler for intel x86 architectures. It can be used to write 16 bit 32 bit (IA-32) & 64-bit (x86-64) programs. NASM is considered to be one of the most popular assembler overall.

Linker:

Linker or link edition is a program that takes one or more object generated by a compiler & combine them into a single executable program. When a program composes multiple object files, the linker combines these files into unified form a collection called library in the output.

Loader: A loader is the part of the operating system that is responsible for loading programs, one of the essential stages in starting a program it means that loader is a program that loads the program into the memory and prepares them for execution. loading the program includes reading the content of executable file. the file containing the programme text, into memory and then carrying out other acquired preparatory tasks. Once loading is completed the operating system starts the program by passing control to the load program code. Dos debugger. "debug" is a command in Dos, MS-Dos & Microsoft windows only (x86 versions) which runs the program debug.exe.

Debug can act as an assembler, disassembler, or examine memory contents in assembly language



⇒ Procedure to create and execute a simple assembly program on Ubuntu using nasm.

Steps to follow:

1. Boot the machine with ubuntu / Fedora.
2. Select & click on "desktop home" icon from the toolbar.
3. Start typing "terminal". Different terminal windows available will be displayed.
4. Click on "terminal" icon. A terminal window will open showing command prompt.
5. Give the following command to invoke the editor gedit hello.asm
6. Type in the program in gedit window, save & exit
7. To assemble the program write the cmd at the prompt as follows & press enter key.
   nasm -f elf32 hello.asm ↵ hello.o (for 32bit)
   nasm -f elf64 hello.asm ↵ hello.o (for 64-bit)
8. If the execution is error free, that means hello.o object file has been created.
9. To link & create the executable given the cmd as
   ld -o hello hello.o
   gcc -o hello hello.o (if you are using function)
10. To execute the program, write at prompt ./hello.
11. "Hello world" will be displayed.

The assembly program structure

The assembly program can be devided into three section.

The data section :- This section is for "declaring initialized data". However this data does not change at runtime so they are not really variable. The data section is used for thing like file names & buffer size where you can use the DB, DW, DD, DQ etc

Ex:-

```
section.data
message db "Hello world!" ; Declare message
mylength equ $-message ; Declaring mylength
buffersize dw 1024 ; Declare buffersize
```

The .bss section :- This section is where you declare your variables. You use the Resb RESW, RESD instructions.

```
section. .bss
filename resb 255 ; Reserve 255 bytes
number resb 1 ; Reserve 1 byte
bignum resw 1 ; Reserve 1 word (1W=2B)
```

The text.section: This is where the actual assembly code is written. The .text section must begin with the declaration global_start. which just tells the kernel where the program execution begins.

section .text
global _start
_start:
; Here is the where the program actually begins

Linux system calls (for 32 bit):
Write system calls that you are using.
You can make use of linux system calls in your assembly program. You need to take the following steps for using linux system calls in your program.

1. Put the system call number in the EAX register
2. Store the arguments to the system call in the register EBX, ECX, etc.
3. Call the relevant interrupt
4. The result is usually returned in the EAX register.

; following table shows some of the system calls frequently used.

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

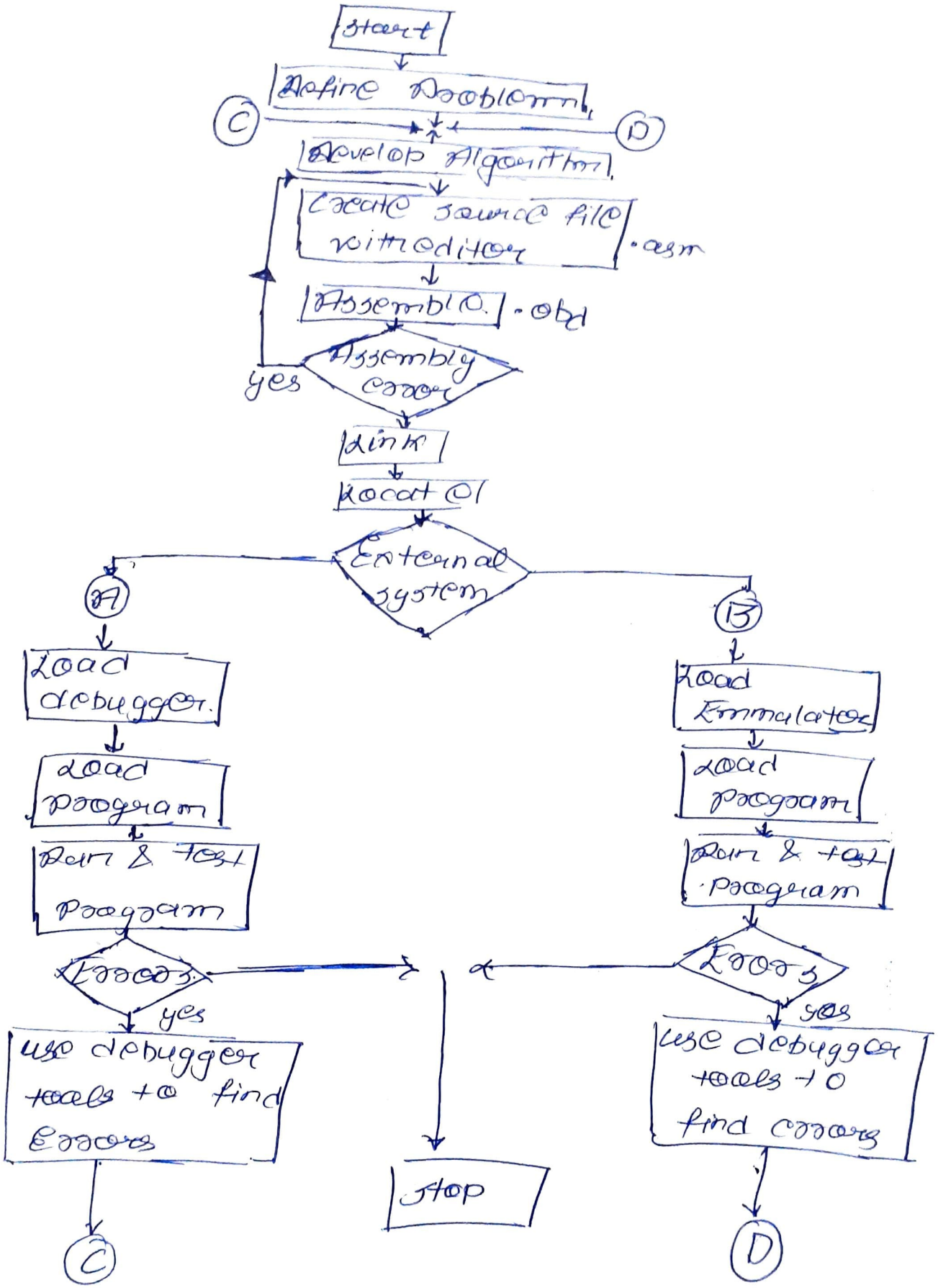
=> The Linux system calls - (syscall for 64 bit execution)

Linux system calls are called in the same ways as DOS system calls:

1. write the system call number in rax.
2. set up the argument in system call in RDI, RSI, RDX etc.
3. make a syscall call "syscall" instruction
4. The result is usually returned in RAX

%0 rax	System call	%0 rdi	%0 rsi	%0 rdx	%0 r10	%0 r8
0	sys_read	unsigned int fd	char * buf	size_t count.		
1.	sys_write	unsigned int fd	const char * buf	size_t count.		
2.	sys_open	const char * filename	int flag	int mode		
3.	sys_close	unsigned int fd.				
60	sys_exit	int error_code.				

Program Development Process



X86 Instruction sets

Write all the Instruction set of 80x86 architecture

Eg: CMP → Compare Two operands.

Operation / Algorithms

Left SRC - sign Extend (Right SRC);

(* CMP does not store a result; Its purpose is to set the flags *)

Description :- CMP substracts the second operand from the first but unlike the SUB instruction does not store to results only the flags are changed.

Flags Affected

OF, SF, ZF, AF, PF and CF

Instruction Set

→ Data Transfer Group

* General purpose

1) MOV Assign operand 2 to operand 1

Algorithm :

operand 1 = operand 2

Description :

MOV instruction is used to copy a word or a byte or double word or Quad-word to/from Register / memory

Flags affected - No flags are affected

2) MOVS - Move string data.

Algorithm

ES : [DI] = DS : [SI]

if Df = 0 then

SI = SI + 1

DI = DI + 1

else

SI = SI - 1

DI = DI - 1

Description

Copies data from addressed by DS:SI to the location ES:DI destination and updates SI & DI Based on the size of operand or instruction. Use with REP prefixes.

Flags affected :- None

MOVSX - Move with Sign-Extend

Algorithm

Operand 1 = operand 2 (with sign)

Description

Copies a byte or a word from a source operand to a source destination register & sign extend into the upper bits of destination

Flags Affected - None

MOVZX - Move with Zero-extend

Algorithm - Operand 1 = operand 2 (with zero in upper Bits)

Description Copy a byte or a word from a source operand to destination register and zero extends into the upper bits of destination. Used to copy an 8-bit or 16 bit operand into a bigger destination

5) PUSH - Push value / register to stack

Algorithm

SP = SP-2 (for 16 bit value)

SS : [SP] (top of the stack) = Operand

Description

PUSH instruction decrement SP by size of operand and transfer one word from source to the stack top (SS : SP)

Flags Affected - None

6) PUSHA / PUSHAD - Push all general purpose Register onto stack

Algorithm

```
PUSH AX
PUSH CX
PUSH DX
PUSH BX
PUSH SP
PUSH BP
PUSH SI
PUSH DI
```

Description: Pushes all general purpose Register onto stack in following order :- (E) AX, (E) CX, (E) DX, (E) BX, (E) SP, (E) BP, (E) SI, (E) DI. The value of SP is the value before the arterial push of SP

Flag Affected : None.

(7) POP : Pop value / Register from stack

Algorithm - Operand = SS:(SP) (top of the stack)
SP = SP + 2 (for 16 bit operand)

Description

Transfer operand at the current stack top (SS:SP) to the destination the increment SP by the size of operand to point to the new stack top CS is not a valid destination.

Flags affected - None.

(8) POPA / POPAD - Pop top 8 value of stack.

Algorithm

```
POP DI
POP SI
POP BP
POP XX   (SP value ignored)
POP BX
POP DX
POP CX
POP AX
```

Description

It Pops, the top 8 value of f the stack into the 8 general purpose register. Register are popped in order (E)DI, (E)SI, (E)BP, (E)SP, (E)DX, (E)CX & (E)AX. The (E)SP value popped from the stack is actually discarded.

Flags Affected - None.

(9) XCHG - Exchange content of source & destination

Algorithm - temp = Operand 2
Operand 2 = operand 1
Operand 1 = temp
} for swapping two value

Description
This instruction is used to swap the content of source & destination operands.

Usage
XCHG dest, SRC
flags affected : None

(10) XLAT / XLATB : Translate byte from table

Algorithm
AL = DS : [BX + unsigned AL]

Description
Replaces the byte in AL with byte from user table addressed by BX. The original value is the index into the translation table i.e copies value of memory byte at DS [BX + unsigned AL] to AL register
Flags affected - None

(11) LAR - Load access Rights

Description: The higher byte of the destination register is overwritten by the value of the access right byte and the low ordered byte is zeroed depending on the selector in source operand. The zero flag is set if the operation is successful.

Flag Affected - ZF

(12) BSWAP - Byte Swap

Usage - BSWAP reg32

Description : Changes the byte order of a 32 Bit Register from Big Endian to Little endian or vice-versa. Result left in destination Register is undefined if the Operand is a 16-Bit Register

Flag affected - None.

INPUT | OUTPUT Instructions.

(13) IN : Input from port into Accumulator

Usage : IN accum, port

Description : A Byte, word & word or Guard is used from "port" & placed in AL, AX, EAX & RAX resp. If the port No. is in the range of 0-255 it can be specified as an immediate otherwise it must be specified in DX.

Flags Affected - None

(14) INS - Input String from port

Description

It Loads data from port to destination ES:(E)DI (E)DI is adjusted by the size of the operand & incr if the direct flag is cleared & decr if set for INSB, INSW, INSD, no operands allowed & size determined

by the mnemonic

Flag affected : None

(15) OUT : output from Accumulator to Port

Description

Transfers byte in AL, word in AX, dword in EAX to be specific hardware port address. first operand is a port no. If required to access port no. over 255 - DX register should be used otherwise it can be specified as immediate.

flag affected - None.

(16) OUTS - output string from accumulator to port

Description - Transfer a byte or word or double word from source accumulator register to the port specified as immediate / in DX register. for instruction with no operands the "src" is located at DS : SI & SI is incremented or decrement by the size of the operand or size dictated by the instruction format

flags affected - None

(17) LEA : Load effective address

Usage : LEA dest, Src EX - LEA reg, memory

Algorithm :

REG = address of memory (offset)

Description Transfers offset address of "Src" to the destination register

flag affected - None

(18) LDS - Load Pointer with DS

Algorithm -
REG = first word
DS = second word

Description
The instruction load memory double word into word register & DT.

flag affected - None

(19) LES : Load Pointer using ES

Algorithm
REG = first word
ES = second word.

Description - This instruction load memory double word into word register & ES

flag affected - None

20) LFS : Load pointer using fS

Algorithm - REG = first word
PS = second word

Description - This instruction loads memory double word into word register & fS.

flags affected - None

(21) LGS - Load Pointer using GS

Algorithm - REG - 1st word
GS - 2nd word

Description : This instruction load memory double word into word registers & GS

flag affected : None

(22) LSS - Load Pointer using SS

Algorithm - REG = first word
SS = 2nd word.

Description : This instruction load memory double word into word register & SS

flag affected - None

FLAG - TRANSFER INSTRUCTIONS

(23) LAHF : Load AH register from flag

Algorithm : AH = flag register

Description : Load AH from low 8 bit of flag register

flag registers : None

(24) SAHF : Store AH register into flags

Algorithm - flag register = AH

Description - Store AH register into low 8 bit of flags register

flag Affected : CF, ZF, SF, OF, PF, AF (all flags)

(25) PUSHF | PUSHFD : push flag | eflag onto stack
Algorithm - SP = SP-2 (for PUSHF)
SS : [SP] (top of flag) = flags

Description : Transfer the flag register onto the stack. PUSHF save a 16 bit value while PUSHFD a 32 bit value.

(26) POPF | POPFD : Pop flags off stack
Algorithm - flags = SS:[SP] (top of the stack)
SP = SP + 2 (for POPF)

Description : Pops word | dword from stack into the flags Register and increment SP by 2(POPF) or 4 (POPFD)

flag affected : All flags

(27) CMPX CHG : compare & Exchange

Algorithm : for & CMPXCHG dest, src

```
    CMP dest EAX
JE = Swap dest src
MOV  EAX dest
 JMP skip

swap_dest_src :
     MOV dest, src
skip:
     ; other instruction
```

```
if ( dest = EAX )
   Set ZF
   dest = src

else
     clear ZF
   EAX = dest
```

Description : compare destination to accumulator
If they are equal source is copied to destination. Else destination is copied to the accumulator.

Flags affected - OF, SF, ZF, AF, PF, CF

ARITHMETIC INSTRUCTIONS

* Addition

(28) ADD - Add byte or word or double word or Quadword

Algorithm:
Operand 1 = Operand 1 + Operand 2

Description
Perform addition of both operand & store the result in 1st operand

Flags affected - All flags

(29) ADC - Add byte or word or double word or Quad word with carry

Algorithm:
Operand 1 = Operand 1 + operand 2 + CF

Description
Perform addition of operand 1 & operand 2 along with carry & store result in operand 1

Flag Affected - All flags

30. INC : Increment Byte / word / double word / Quad word by 1.

Algorithm - Operand = Operand + 1

Description - Increment the operand by 1 byte

Flag affected - All flags except CF

31. AAA - ASCII adjust for addition

Algorithm

if low nibble of AL > 9 or AF = 1 then

AL = AL + 6

AH = AH + 1

AF = 1

CF = 1

else

AF = 0

CF = 0

in both cases, it clears higher nibble of AL.

Description - corrects result in AH & AL after addition when working with BCD values

Flags affected → AF, CF

(32) DAA : Decimal adjust for addition

Algorithm : If low nibble of AL > 9 or AF = 1 then

AL = AL + 6

AF = 1

if AL > 9FH or CF = 1 then

AL = AL + 60H

CF = 1

Description : correct the result of addition of two packed BCD

flag affected : All value

SUBTRACTION

(33) SUB : Subtracted Byte / word / dword / qword

Algorithm - Operand 1 = operand 1 - operand 2

Description : Perform subtraction & store result in Operand 1

Flag Affected : All flags

(34) SBB : Subtracted byte / word / double word / q word with borrow

Algorithm - Operand 1 = Operand 1 - Operand 2 - CF

Description - Perform subtraction & store result in Operand 1

flag affected - All flags

(35) DEC - Decrement instruction
Algorithm - operand = operand - 1
Description - Decrement byte / word / d word / q word by 1
Flag affected - All flags

(36) NEG - Negate byte / word / d word / q word
Algorithm - Inverts all bit of operand. Add 1 to inverted operand
Description - It makes operand -ve (two's compliment)
Flags affected - All flags

(37) AAS - ASCII adjust after subtraction
Algorithm - If low nibble of AL > 9 or AF = 1 then
AL = AL - 6
AH = AH - 1
AF = 1
CF = 1
else :
AF = 0
CF = 0
In both case : clear high nibble of AL
Description : Corrects result in AH & AL after subtraction when working with BCD values
Affected flags : AF, CF

(38) DAS : Decimal adjust after subtraction
Algorithm : If low nibble of AL > 9 or AF = 1 then
AL = AL - 6 and AF = 1
If AF > 9FH or CF = 1 then
AL = AL - 60H & CF = 1

Description: Corrects result of subtraction of two packed BCD values
Flags affected: All flags.

(39) CMP - Compare two operands
Algorithm - Left SRC - Sign Extend (Right SRC)
Description: CMP subtracts the second operand from the first but unlike SUB instruction, does not store the result: only the flags are changed
Flags affected: OF, SF, ZF, AF and CF.

MULTIPLICATION

(40) MUL - multiplies byte | word | dword | qword (unsigned)
Algorithm: when operand is a byte
AX = AL * operand
when operand is a word
DX . AX = AX * operand.
Description: Multiplies AL, AX, EAX or RAX by source operand
If source is 8 bits, it is multiplied by AL and product is stored in AX. If the source is 16 but it is multiplied by AX and product is stored in DX:AX. If source is 32 bit, it is multiplied by EAX & Product is stored in EDX:EAX If the source is 64 bit, it is multiplied by RAX and product is stored in RDX:RAX
flag Affected: CF, OF

(41) IMUL - Multiply byte | word | dword | qword (signed)
Algorithm: when operand is a quad word
(RDX, RAX) = RAX * operand
Description: Perform a signed integer multiplication on AL, AX, EAX.
Flag affected = CF, OF

(42) AAM = ASCII adjust after multiplication
Algorithm = AH = AL | 10
AL = remainder
Description: correct the result of multiplication of two BCD values
Flags affected: ZF, SF, PF

DIVISION

(43) DIV: Unsigned Divide
Algorithm: when operand is a word
AX = (DX AX) | operand
DX = remainder (modulus)
Description - Perform 8-, 16-, 32- bit or 64- bit unsigned division. If the divisor is 8 bit the dividend is AX, the quotient is AL and remainder is AH. If divisor is 16 bit the dividend is DX:AX quotient is AX & remainder is DX. If the divisor is 32 bit, dividend is EDX:EAX, quotient is EAX & remainder is EDX. If division is 64 bit, the dividend is RDX:RAX, quotient is RAX & remainder RDX
Flags affected - All flags modified after this instruction

(44) IDIV - Signed Divide

Algorithm - When operand is a byte

AL = AX / operand and AH = remainder (modulus)

Description: Perform a signed integer division operation

Usually the IDIV operation is preffed by CBW to sign-extend dividend

Flag affected - All flags undefined

(45) AAD, ASCII adjust before division

Algorithm: AL = (AH * 10) + AL

AH = 0

Description: Prepare two BCD value for division

Flags affected = ZF, SF, PF

(46) CBW convert byte into word

Algorithm: if high bit of AL = 1 then:

AH = 255 (0ffH)

else AH = 0

Description: It converts a byte into word

Flags affected: None

(47) CWD - convert word to double word

Algorithm: If high byte of AX is 1 then

DX = 65535 (0FFFFH)

else DX = 0

Description: It converts a word to double word

flag affected: None

Q8. CDQ: Convert dword to qword.

Algorithm: If right bit of EAX=1 then:

EDX = 0FFFFFFFFH

else: EDX = 0

Description: It extends the sign bit in EAX throughout the EDX register.

Flag Affected: None.

Bit Manipulation Instruction

Logical.

49. Not : Not byte / word / dword / qword.
Algorithm: if bit is 1, turn it to 0
if bit is 0, turn it to 1.
Description: invert each bit of the operand
Flag Affected: None

50. AND And byte / word / dword / qword.
Rules 1 And 1 = 1
1 And 0 = 0
0 And 1 = 0
0 And 0 = 0
Description: Applies logical And b/w all bit of two Operands
Flags Affected: ZF, SF, PF

51. OR - OR byte / word / dword / qword
Rules 1 OR 1 = 1
1 OR 0 = 1
0 OR 1 = 1
0 OR 0 = 0
Description: Apply logical OR b/w 2 operands
Flag affected: ZF, SF, PF.

52. XOR :- Exclusive OR byte / word / dword / qword.
Rules: 1 XOR 1 = 0
1 XOR 0 = 1
0 XOR 1 = 1
0 XOR 0 = 0

Description: Applies logical XOR b/w all bits of 2 operands.

Flag affected: ZF, SF, PF

53) TEST:- Test byte/dword/word/qword.

Rules: same as logical AND

Description: Logical AND b/w all bits of 2 operand for flags only.

Flags affected: ZF, SF, PF

54) BT: Bit Test.

Description: the destination bit indexed by the source value.

usage: BT dest, src

Flag affected: carry flag (CF)

55) BTS: Bit Test & Set.

usage: BTS dest, src.

Description: the destination bit indexed by src value, is copied to carry flag and then set in the destination

Flags affected: Carry flag (CF)

56 BTR: Bit Test & Reset.

usage: BTR dest, src.

Description: the distination bit is in indexed by src value and then cleared in destination

Flags affected: carry flag (CF)

57 BTC: Bit test with compliment
Usage: BTC dest, src.
Description: The destination bit index by source value is copied into the carry flag after being complimented
Flag Affected: carry flag (CF)

58 BSF: Bit scan forward.
Usage: BSF dest, src.
Description: scans source operand from first bit set sets ZF if a bit is found set bit: clears ZF if no bits are found set. BSF scans forward across bit pattern (0-n)
Flag affected: zero flag (ZF)

Shift Instruction

59 BSR: Bit scan Reverse.
Description: Scans source operand for first set bits sets ZF if a set bit is found and loads destination with an index to first set bit. BSR scans in reverse (n-0)
flags affected: zero flag (ZF)

60 SHL: Shift ~~logical~~ Logical left
Algorithm: Shift all bits left
• zero bit is inserted to right
Description: Shift operand left
Flags Affected: CF, AF

61 SAL: Shift Arithmetic Left

Algorithm: • Shift all bits left

• zero bit is inserted to right

Description: Shift arithmetic operand 1 left.

Flag affected: CF, AF

62) SHR - Shift logical Right.

Algo: shift all bits right, the bit that goes off is set to CF

• zero bits is inserted to left

Description: shift operand 1 right & no of shift is set by operand 2.

Flags affected: CF, AF

63: SAR: Shift Arithmetic Right

Algorithm: shift all bits right, the bits that goes off is set to CF

• the sign bit that is inserted to left most position has the same value as before shift.

Description: shift Arithmetic operand 1 Right the number of shift is set by operand 2

Flag affected: CF, AF

64 SHLD: Double Precision Shift left
Description: Shift the bits of the second operand into the first operand. The third operand indicate the number of bit to be shifted the position opened by the the shift are filled by the most significant bit of second operand.
Flags Affected: AF, SF, ZF PF, CF

65 SHRD: Double Precision shift Right
Description: Shift the bits of second operand into first operand. the third operand indicates the number of operand. the third operand indicates the number of bits is shifted
Flags affected: AF, SF, ZF, PF, CF

Rotate Instructions

66 ROL: Rotate Left.
Algorithm: Shift all bits left, the bit that goes off to CF and same bit is inserted to the right.
Description: Rotate operand 1 left, the no of position is set by OP2.
Flags affected: CF, AF

67 ROR: Rotate Right.
Algorithm: Shift all bits right. the bit that goes off is set to CF and the same bit is inserted to the ~~right~~ left

Description: Rotate operand right. the number of of rotation is set by operand.

Flag affected: CF, OF

68. RCL: Rotate through carry flag.

Algorithm: Shift all bits left. the bit that goes off is set to CF & previous value of CF is inserted.

Description: Rotate operand left through carry flag. the number of rotation is set by operand.

Flag affected: CF, OF

69. RCR: Rotate through carry Right.

Algorithm: Shift a/c bits right, the bit that goes off is set to CF & previous value of CF is inserted.

Description: Rotate operand 1 right through carry flag number of rotation.

Flag affected: CF, OF.

String Instruction Group.

70) CMPS, CMPSB, CMPSW, CMPSD: compare strings
Algorithm: compares string in memory addressed by DS:(E)SI & ES
Description: (E)SI and (E)DI are increased or decreased according to the operand size and status of direction flag.
Flags affected: AF, SF, ZF, PF, CF

71) SCAS, SCASB, SCASW, SCASD: scan string
Description: scans a string in memory pointed ES:(E)DI for a value that matches the accumulator. SCAS requires the operands to be specified. (E)DI is increased or decreased acc. to operand.
Flag affected: AF, SF, ZF, PF, CF

72) LODS, LODSB, LODSW, LODSD: load accumulator into string.
Description: Loads a memory byte/word addressed by DS:(E)SI into the accumulator. If LODS is used operand must be specified. (E)SI is increased or decreased accordingly to the operand size.

Flag affected: None.

23 STAB, STOSB, STOSW, STOSD: Store Data

Description: Stores the accumulator in the memory location addressed by ES:(ED) DI. If STOS is used, a destination operand must be specified.

Flag affected: None.

24) REP: Repeat string

Description: Repeat execution of string instruction while CX != 0 & zero flag is reset. CX is decremented each time the instruction is repeated.

Flag affected: None

25) REPE/REPZ: Repeat equal/repeat zero.

Description: Repeat execution of string instruction while CX != 0 & zero flag is set.

Flag Affected: none.

26) REPNE/REPNZ: Repeat Not equal/Repeat Not zero.

Description: Repeats execution of string instruction while CX != 0 and zero flag is clear. CX is decremented and zero flag tested after each string operation.

Flags Affected: None.

Process control instruction

Flag operations.

77 CLC : clear carry flag.
Algorithm: CF = 0
Description: This instruction clear the carry flag
Flag affected: CF

78 STC : set the carry
Algorithm: CF = 1
Description: This instruction set the carry flag
flag affected: CF

79 CMC : complementary carry flag.
Algorithm: AF = 0.
Description: It clear direction flag. SI & DI will be incremental.
Flag affected: DF

80 CLD : clear direction flag.
Algorithm: DF = 0.
Description: It clears direction flag. SI & DI will be incremental by chain instructions like CMPS, LODS, STOS.
Flag affected: DF

81 STD - set direction flag.
Algorithm: DF = 1
Description: It sets direction flag. SI & DI will be decremental by chain instructions
Flag affected: DF

82) CLI: clear Interrupt flag
Algorithm: IF = 0
Description: It clears interrupt flag which disable hardware interrupt.
Flag affected: IF

83) STI: set Interrupt flag.
Algorithm: It = 1
Description: It sets Interrupt flag which enables hardware Interrupt.
Flag affected: IF

84) NOP: no operation.
Algorithm: Do nothing.
Description: This instruction is used to do nothing.
Flag affected: none

External synchronization:

85) HLT: Halt the system
Description: This instruction is used to halt the system until interrupt or reset.
Flag affected: No flag are affected.

86) wait: wait till interupt.
Algorithm: None
Description: CPU enters wait state until the coprocessor signals have finished operations
Flag affected: None.

87) ESC : Escape
Description: Provides access to the data bus for other resident process. The cpu treats it as a NOA but places memory operand on bus
Flag Affected: None.

88) LOCK:- Lock Bus during next instruction.
Description:- Provides access to the data bus for other resident processor. The CPU treats it as a NOA but places memory operands on bus. Used to avoid 2 processor.
Flags affected: None.

89) BOUND : Check against Array Bound.
Description:- Array index in source register is checked against upper and lower bounds in Memory source. The first word located at "limit" is lower boundary and word at "limit+2" is upper array bound.
Flag affected: None.
Usage: BOUND SRC, Limit.

90) ENTER: Make stack frame for Procedure entry.
Description: Modifies stack for entry to procedure for high level language operand "locals", specifies amount of storage to be allocated.
Flags Affected: None

91. LEAVE: Restore stack for Procedure Exit.
Description: Releases the local variables created by the previous Enter instruction. by restoring SP and BA to their condition before the Procedure. stack frame initialised.
Flag Affected: None.

Unconditional Transfers.

92. CALL: Call is Procedure
Description: Pushes the location of the next instructions onto the stack and transfer to the destination location. If the process is near / in the same segment.
Flag Affected: None.

93. RET: Return from procedure
Algorithm: POP from stack
IA
if immediate operand is present. SP=
SP+operand.
Description: It is used ~~RET~~ to return from near procedure.
Flags Affected: None.

94. RETF: Return from far procedure.
Algorithm: POP from stack
IP
CS
Description: If immediate operand is present
SP = SP + operand. Return from far procedure by popping IP.
Flag Affected: None

95) JMP- Jump.

Algorithm: Jump to label.

Description: It is used to jump the program control to the label. It can be made conditional like JA, JE, JNC etc

Flag Affected: AF, OF, ZF, PF, AF, CF

Iteration control Instruction.

96) LOOP:

Algorithm: Cx = Cx-1

if Cx != 0 then

Jump

else

no jump, continue

Description: It decreases cx and jump to label if cx is not zero

Flags Affected: None

97) LOOPE

Algorithm: CX = CX+1

if (CX != 0) and (ZF = 1) then

jump

else:

no jump continue.

Description: Decreases Cx, jump to label if CX != 0 and Equal (ZF = 1)

Flag affected: None.

98:- LOOPZ:

Algorithm: CX = CX - 1
if (CX != 0) & (ZF = 1) then,
jump.
else
no dump, continue

Description: It decreases CX and jump to label if CX != 0 & ZF = 1

Flag Affected: None

99:- LOOPNE

Algorithm: CX = CX - 1
if (CX != 0) & (ZF = 0) then jump
else no jump, continue

Description: It decreases CX & jumps to label if CX != 0 & Not equal (ZF = 0)

Flag Affected: None.

100:- LOOPNZ

Algorithm: CX = CX - 1
if (CX != 0) & (ZF = 0) then jump.
else: NO dump, continue.

Description: It decreases CX & jump to label if CX != 0 & ZF = 0

Flag Affected = None.

Interrupts

101 INT: Interrupts
Algorithm: Push to stack
Flags register.
CS & IP
AF = 0
Transfer control to interrupt procedure
Description: It is used to all the kernel for processing interrupt and is numbered by an immediate byte.
Flag Affected: IF, other flags are pushed in stack

102 INTO: Interrupt overflow
Algorithm: if OF = 1 the INT 4
Description: It calls Interrupt 4 if overflow flag is 1.
Flags Affected: IF

103 IRET: Interrupt Return
Algorithm: POP from stack:
IP
CS
Flags register.
Description: It is used to return from Interrupt
Flag Affected: All flags are popped from the stack.

Protected Mode Specific Instruction.

104.> ARPL: Adjust Request Privilege level of selector
Description: compare the RPL bits of ~~dest~~
against "src" if the RPL bits are ~~[illegible]~~ set
equal to the source RPL bits and zero
flag is set. otherwise the zero bit flag
is cleared.
Flag Affected: ZF.

105> CLTS: Clear Task switched flag
Description: Clears the task switched flag
in the Machine Status Register. This is a
Privileged operation and is generally used
only by operating system code.
Flags Affected: ZF

106> LAR: Load Access Right.
Description: The High byte of the destination
register is overwritten by the value of
the access right byte and the low order
byte is zeroed depending on the selection
in the source operand.
Flag Affected: ZF

107> LDT: Load Local Description Table.
Description: Load a value from an operand
into the Local Description Table Register (LDTR)

Flags Affected: None

108 LIDT: Load Interrupt Description Table.
Description: Load a value from an operand into the Interrupt Descriptor Table.
Flag Affected: None.

109 LGDT: Load Global descriptor Table.
Description: Loads a value from an operand into the Global Descriptor Table(GDT) register.
Flag Affected: None.

110 LMSW: Load Machine status word.
Description: Load the machine status word (MSW) from data found at "SRC"
flag affected: None:

111 SGDT: Store Global Descriptor Table.
Description: Store the Global Description table (GDT) Register into the specified.
flags affected: None.

112 SIDT: Store Interrupt Description Table.
Description: Stores the Interrupt Description Table(IDT) register into the specified Operand ("test")
flags affected: None.

113 SLDT: Store Local Description Table.
Description: Stores the Local Descriptor Table(LDT) register into the specified operand.
flag affected: None.

114.) SMSW: Store Machine Status word.

Description: Stores Machine Status word (MSW) into "test"

Flag affected: None.

115.) STR: Store task Register.

Description: Store the current Task Register to the specified operand.

Flags affected: None.

116.) VERR: Verify Read.

Description verifies the read of the specified segment selector is valued and is readable at the current privilages level. If the segment is readable the zero flag is set, otherwise it is cleared.

117.) VERW: verify write

Description: Verify the specified segment selector is valid and is writable at current privilage level. If segment is writable, the zero flag is set, otherwise it is cleared.

Flag affected: ZF (zero flag)

118.) LTR: Load Task Register.

Description: Loads the current task register with the value specified in "r"

Flags affected: None

119. LSL: Load segment limit.
Description: Load the segment limit of a selector is valid and visible at the current privilage level of loading is successful the zero flag is set otherwise it is cleared.

Flag Affected: ZF (zero flag)

* Conditional Jump statement.

	Mnemonics	Meaning	Jump condition
1.	JA	Jump if Above	CF=0 & ZF=0
2.	JAE	Jump if Above or equal	CF=0
3.	JB	Jump below	CF=1
4.	JBE	Jump below or equal	CF=1 or ZF=1
5.	JC	Jump if carry.	CF=1
6.	JCXZ	Jump if CX zero	Cx=0
7.	JE	Jump if Equal	ZF=1
8.	JG	Jump if greater (signed)	ZF=0 & SF=OF
9.	JGE	Jump if greater or equal (signed)	SF=OF
10.	JL	Jump if less (signed)	SF!=OF
11.	JLE	Jump if less or equal (signed)	ZF=1 OR SF!=O
12.	JMP	Un conditional Jump	Unconditional.
13.	JNA	Jump not above	CF=1 or ZF=1
14.	JNAE	Jump if not above or equal	CF=1
15.	JNB	Jump if not below	CF=0
16.	JNBE	Jump if not below or equal	CF=0 =& ZF=0
17.	JNC	Jump if no carry.	CF=0
18.	JNE	Jump if not equal.	ZF=0
19.	JNG	Jump if not greater (signed)	ZF=1 or SF!=O

	Mnemonics	meaning	Jump condition
20	JNGE	Jump if not greater or equal (signed)	SF != OF
21	JNL	Jump if not less (signed	SF = OF
22	JNLE	Jump if not less or equal(signed)	ZF = 0 & SF = OF
23	JNO	Jump if not overflow (signed)	OF = 0
24	JNP	Jump if no parity	PF = 0
25	JNS	Jump if not signed. (signed)	SF = 0
26	JNZ	Jump if not zero,	ZF = 0
27	JO	Jump if overflow (signed)	OF = 1
28	JP	Jump if parity.	PF = 1
29	JPE	Jump if Parity Even.	PF = 1
30	JPO	Jump if Parity Odd	PF = 0
31	JS	Jump if signed (signed)	SF = 1
32	JZ	Jump if zero.	ZF = 1