Chapter 9
ASSEMBLY LANGUAGE FUNDAMENTALS
PC Architecture for Technicians: Level-1

Systems Manufacturing Training and Employee Development
Copyright © 1996 Intel Corp.
OBJECTIVES: At the end of this section, the student will be able to do the following:

- Define various terms associated with Assembly Language.
- Discuss Little-Endian Addressing
- Describe the 80x86 Real Mode Registers
- Discuss the use of Segments and Offsets.
- Explain how addresses are generated in Real Mode.
- Discuss the Data Transfer, Arithmetic, Logic, Shift & Rotate, Control & Transfer, Repeating Instruction set.
- Discuss the use of Procedures in Assembly Language.
- Read and understand examples of Assembly Language programs from BIOS & POST.
Assembly Language Vocabulary
Assembly Language Vocabulary

○ Assembler
  • A program which translates mnemonic code & symbolic addresses into Machine Language.

○ Assembly Language
  • A symbolic notation for writing machine instructions.

○ Byte
  • A group of 8 bits.

○ Instruction Pointer
  • A register containing the offset of the instruction currently being executed. The segment containing this instruction is always pointed to by the Code Segment (CS) Register.
Assembly Language Vocabulary

◦ Machine Language
  • An instruction for a microprocessor which is decode & interpreted by the CPU without further modification or translation by software or hardware.

◦ Masking
  • A means of examining only certain bits in a word. This is usually done by ANDing the word with a MASK containing 1’s in the desired bit positions.

◦ Mnemonics
  • Easy to remember abbreviations which characterize the machine instructions of a processor, and which are translated by an Assembler into machine instructions.
Assembly Language Vocabulary

- **Physical Address**
  - The address output onto the address bus by the processor.

- **Real Mode**
  - An 80x86 operating mode where the Segment value is multiplied by 10H and the offset is added to generate the physical memory address.

- **Register**
  - Internal memories of a CPU whose contents can be loaded or modified by instructions or the CPU itself.
ASSEMBLY LANGUAGE OVERVIEW
ASSEMBLY LANGUAGE OVERVIEW

- Assembly language is a symbolic form of a computer’s own internal language called Machine Language.
  - The CPU understands **Machine Language** (binary 1’s & 0’s), but this is very difficult to work with.
    - e.g. **B8 AA 55** (in HEX)
      » 1011 1000 1010 1010 0101 0101
  - Assembly Language uses **MNEMONICS** to represent each instruction type and makes it easier to work with.
    - e.g. **MOV AX, 55AA**

- Many board debugging tools (In Target Probes, Logic Analyzers) have the capability of displaying memory (e.g. BIOS code) in Assembly Language format.
ASSEMBLY LANGUAGE OVERVIEW

• All Intel x86 Processors behave as the original 8086/88 processor once **RESET**:
  • The Advanced features of the 286, 386, 486 & Pentium family are turned off
  • The CPU is operating in REAL MODE.
    • The CPU addresses memory using the Intel Segmented Addressing mechanism.
    • Memory Address space is limited to 1 MB.
  • Virtually all POST tests are conducted in REAL MODE.
ASSEMBLY LANGUAGE OVERVIEW

ASSEMBLY LANGUAGE - REAL MODE ONLY

It is often useful when debugging boards to be able to:

- Write small assembly language routines
- Understand BIOS, POST and test routines written by others.

EXAMPLES:

- 1. We may need a small routine to continuously read a particular memory or I/O location so we can observe bus cycles with a logic analyzer or oscilloscope.
- 3. We may need to write a routine to test a specific peripheral chip—DMA, PIC, PIT, real-time clock, keyboard controller, etc.
Little-Endian Addressing
## Data Types

Note: BT(x) instructions can manipulate **data** down to the bit level.

<table>
<thead>
<tr>
<th>Byte</th>
<th>Address N</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>High Byte</th>
<th>Low Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address N</td>
<td>Address N+1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Word</th>
<th>Address N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>High Word</th>
<th>Low Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address N</td>
<td>Address N+1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Doubleword</th>
<th>Address N</th>
</tr>
</thead>
<tbody>
<tr>
<td>High Doubleword</td>
<td>Low Doubleword</td>
</tr>
</tbody>
</table>

| Address N+7 | Address N+6 | Address N+5 | Address N+4 |
| High Doubleword |         |           |           |

| Address N+3 | Address N+2 | Address N+1 | Address N |
| Low Doubleword |         |           |           |

| Address N+3 | Address N+2 | Address N+1 | Address N |
| Low Doubleword |         |           |           |

| Address N+3 | Address N+2 | Address N+1 | Address N |
| Low Doubleword |         |           |           |

| Address N+3 | Address N+2 | Address N+1 | Address N |
| Low Doubleword |         |           |           |

| Address N+3 | Address N+2 | Address N+1 | Address N |
| Low Doubleword |         |           |           |

### Note:
BT(x) instructions can manipulate **data** down to the bit level.
Little-Endian Addressing

Intel CPUs can address memory down to the byte level (8 bits).

- By default all Intel processors use “Little-Endian” addressing format when accessing 16-bit (WORD), 32-bit (DWORD), or 64-bit (QWORD) values.
- Little-Endian addressing says that the Least Significant Byte of multiple byte quantities will be stored at the Lowest Address.
  - And the byte at the Highest Address will be the Most Significant Byte
- When writing multiple byte entries in the usual way (with highest order bit left & lowest order bit right), the arrangement stored in memory seems to be exchanged.
Little-Endian Addressing

- Suppose that a 32 bit value of 12345678H is stored at memory location 100H physical.
- So the 32 bit value of 12345678H will be stored in memory as:

<table>
<thead>
<tr>
<th>ADDR:</th>
<th>100H</th>
<th>101H</th>
<th>102H</th>
<th>103H</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA:</td>
<td>78H</td>
<td>56H</td>
<td>34H</td>
<td>12H</td>
</tr>
</tbody>
</table>

N  N+1  N+2  N+3
Little-Endian Addressing

• The first instruction executed in a PC is typically a FAR jump to the fixed entry point of the POST (POWER-ON-SELF-TEST) program.
  
• This is stored in the System BIOS from FFFF0p to FFFF4P as follows:

  - EA 5B EO 00 F0 \{FFFF0 -> FFFF4\}
    - FFFF0 = EA ;FAR JUMP - NEED CS & IP (4 more bytes)
    - FFFF1 = 5B ;Low byte of word for IP
    - FFFF2 = E0 ;High byte of word for IP
    - FFFF3 = 00 ;Low byte of word for CS
    - FFFF4 = F0 ;High byte of word for CS

• FAR JUMP TO F000:E05B (FE05Bp)
Register Set
# Application Register Set

## General Registers

<table>
<thead>
<tr>
<th>31</th>
<th>23</th>
<th>16</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>0</th>
<th>16-BIT</th>
<th>32-BIT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>AX</td>
<td>EAX</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DX</td>
<td>EDX</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CX</td>
<td>ECX</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>BX</td>
<td>EBX</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>EBP</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ESI</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>EDI</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ESP</td>
<td></td>
</tr>
</tbody>
</table>

## Segment Registers

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CS</td>
</tr>
<tr>
<td>SS</td>
</tr>
<tr>
<td>DS</td>
</tr>
<tr>
<td>ES</td>
</tr>
<tr>
<td>FS</td>
</tr>
<tr>
<td>GS</td>
</tr>
</tbody>
</table>

## Status and Control

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>EFLAGS</td>
</tr>
<tr>
<td></td>
<td>EIP</td>
</tr>
</tbody>
</table>

**NOTE:** All registers in REAL MODE DEFAULT to 16 bits wide.
GENERAL PURPOSE REGISTERS

- **32-BIT REGISTERS**
  - EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EIP

- **16-BIT REGISTERS**
  - AX, BX, CX, DX, SI, DI, BP, SP, IP

- **8-BIT REGISTERS**
  - AL, BL, CL, DL, AH, BH, CH, DH

**EXAMPLE:**
- AL = low byte of EAX
- AH = next byte of EAX
- AX = low word of EAX

The high word of EAX can't be addressed directly.
COMMON USAGE OF REGISTERS

AL / AX / EAX - INPUT/OUTPUT DATA REGISTER

BX / EBX - DATA SEGMENT BYTE POINTER

CX / ECX - IMPLIED COUNT REGISTER

DX / EDX - INPUT/OUTPUT ADDRESS REGISTER
COMMON USAGE OF REGISTERS

- (E)IP points to the next instruction within the Code Segment
  - Sometimes called the Program Counter.
- (E)SP points to the top of the Stack within the Stack Segment
  - (E)SP points to the last item pushed on the stack.
- (E)SI and (E)DI are used in string move instructions.
- (E)CX used as the "count" register in looping instructions.
- AL, AX, and (E)AX are used in I/O instructions for sending and receiving data.
- DX used to specify a port address in I/O instructions.
- Any register pointer (except EIP) can generally be used as a memory pointer (containing the address of a memory location).
  - (E)IP is reserved for the Instruction Pointer exclusively.
SEGMENT REGISTERS

- Segment registers are used to determine where segments begin.
  - All memory references are referenced to the beginning of a segment.
  - Segment Base = Address of the Beginning of a Segment
  - Segment registers are 16 bits wide.
- Most small programs use only the following:
  - CS to specify the CSBASE for the code
  - DS to specify the DSBASE for the data
  - SS to specify the SSBASE for the stack
Segmented Addressing
NOTE: We will assume, unless otherwise stated, that all examples reflect real mode.
Segmented Memory

- Code Segment
- Stack Segment
- Data Segment
- Data Segment
- Data Segment
SEGMENT REGISTERS

- In **REAL MODE**, the segment register contains the upper 16 bits of the 20-bit segment base address.
  - Simply appending an 0H to the segment register's contents gives the base address (20 bits).
  - **EXAMPLE:** If CS = F000H,
    - Then appending an 0H gives CSBASE = F0000H.
    - **NOTE:** CSBASE, DSBASE, SSBASE refer to the address of the beginning of a Segment. With an ITP debug tool you can access the CSBAS, DSBAS & SSBAS registers.

- Note: In protected mode, the segment register contains an index into a descriptor table. The base address is contained in a descriptor table.
Segmented Addressing

- Segmented addresses are written as Segment and Offset component separated by a **COLON**.
  - e.g. F000:FFFF0 or in register terms CS:IP
- In Real Mode, the Effective Address is the Physical Address put on the Address Bus.
  - The 20 address lines (A19:0) sent to memory in Real Mode form the Effective Address.
    - A31:20 are usually ZERO in Real Mode which emulates the 8086 processor which had only A19:0.
    - **NOTE**: A20 is gated by logic external to the CPU to emulate the 8088/88 processor 1 Mbyte address space.
Segmented Addressing

- The Effective Address is the Physical Address put on the Address Bus (Cont.)

  • The Effective Address is computed by:
    - Effective Address = (Segment * 10H) + Offset
    - Or, Effective Address = (Segment * 16t) + Offset
    - Or, Left Shifting the Segment by 4 bits which gives the appearance of simply appending a ZERO to the SEGMENT

  - When reset, the CS:IP values are F000:FFF0
  - The Effective Address is computed by
    - Effective Address = (F000H * 10H) + FFF0H
    - Effective Address = F0000H + FFF0H
    - Effective Address = FFFF0H (A19:0).
Segmented Addressing

Follow the first four examples, then compute the second four yourself:

1000:0, 1234:5678, F000:FFF0, 8110:091C

<table>
<thead>
<tr>
<th>Segment 1</th>
<th>Segment 2</th>
<th>Segment 3</th>
<th>Segment 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000_</td>
<td>1234_</td>
<td>F000_</td>
<td>8110_</td>
</tr>
<tr>
<td>0000</td>
<td>5678</td>
<td>FFF0</td>
<td>091C</td>
</tr>
<tr>
<td>10000</td>
<td>179B8</td>
<td>FFFF0</td>
<td>81A1C</td>
</tr>
<tr>
<td>FFFF_</td>
<td>8000_</td>
<td>F400_</td>
<td>168A_</td>
</tr>
<tr>
<td>0000</td>
<td>1A1C</td>
<td>E000</td>
<td>1118</td>
</tr>
</tbody>
</table>

Note: Answers on last slide (hidden)

NOTE: NEXT PAGE BLANK
Segmented Addressing

- **INSTRUCTIONS** -- The contents of (E)IP are added to the CSBASE to find the location of the next instruction.
  - CS = 0400H
  - CSBASE = 00004000H
  - (E)IP = 0000128H
  - INSTRUCTION ADDRESS = 00004128H

- **STACK ACCESS** -- The contents of ESP are added to the SSBASE to find the location of the stack.
  - SS = 3000H
  - SSBASE = 00030000H
  - (E)SP = 0000FFFCH
  - STACK ADDRESS = 0003FFFCH
SEGMENT REGISTERS

NOTE: The Stack expands downward.

ESP (SP) = FFFCH
SS = 3000H
EIP (IP) = 128H
CS = 400H

3000H  + FFFCH  = 3FFFCH

LOW MEMORY

HIGH MEMORY

STACK

CODE

NOTE: SETTING UP A SEGMENT REGISTER IS A 2 STEP OPERATION
MOV AX, 3000H
MOV SS, AX

4000H + 128H  = 4128H
Segmented Addressing

- **DATA** -- The offset of the data is typically added to the DS BASE. The offset of the data is determined by using a **POINTER REGISTER**
  - MOV EAX, [EBX]
  - DATA Memory references default to the DS Register.
  - We’ll see that the instruction MOV EAX, [EBX] is equivalent to MOV EAX, DS:[EBX]

- All general purpose registers can be used as memory pointers which default to the following segments:
  - EAX, EBX, ECX, EDX, ESI, EDI default to the DS BASE
  - ESP, EBP default to the SS BASE
NOTE: SETTING UP A SEGMENT REGISTER IS A 2 STEP OPERATION

MOV AX, 125H
MOV DS, AX
ADDRESSING MEMORY

<table>
<thead>
<tr>
<th>32</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS</td>
<td>CSBAS</td>
</tr>
<tr>
<td>DS</td>
<td>DSBAS</td>
</tr>
<tr>
<td>ES</td>
<td>ESBAS</td>
</tr>
<tr>
<td>FS</td>
<td>FSBAS</td>
</tr>
<tr>
<td>GS</td>
<td>GSBAS</td>
</tr>
<tr>
<td>SS</td>
<td>SSBAS</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP</td>
</tr>
<tr>
<td>BX</td>
</tr>
<tr>
<td>BP</td>
</tr>
<tr>
<td>SI</td>
</tr>
<tr>
<td>DI</td>
</tr>
<tr>
<td>SP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>32</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP</td>
<td></td>
</tr>
<tr>
<td>EIP</td>
<td></td>
</tr>
<tr>
<td>BX</td>
<td></td>
</tr>
<tr>
<td>EBX</td>
<td></td>
</tr>
<tr>
<td>BP</td>
<td></td>
</tr>
<tr>
<td>EBP</td>
<td></td>
</tr>
<tr>
<td>SI</td>
<td></td>
</tr>
<tr>
<td>ESI</td>
<td></td>
</tr>
<tr>
<td>DI</td>
<td></td>
</tr>
<tr>
<td>EDI</td>
<td></td>
</tr>
<tr>
<td>SP</td>
<td></td>
</tr>
<tr>
<td>ESP</td>
<td></td>
</tr>
</tbody>
</table>
### EFlags Register

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Bit Positions shown as “0” or “1” are Intel reserved.</td>
</tr>
<tr>
<td>30-23</td>
<td>ID X</td>
</tr>
<tr>
<td>22-15</td>
<td>VIP X</td>
</tr>
<tr>
<td>14-7</td>
<td>VIF X</td>
</tr>
<tr>
<td>6-0</td>
<td>AC X</td>
</tr>
<tr>
<td>31</td>
<td>VM X</td>
</tr>
<tr>
<td>23</td>
<td>RF X</td>
</tr>
<tr>
<td>22</td>
<td>NT X</td>
</tr>
<tr>
<td>21-16</td>
<td>IOPL X</td>
</tr>
<tr>
<td>15</td>
<td>OF S</td>
</tr>
<tr>
<td>14-7</td>
<td>DF C</td>
</tr>
<tr>
<td>6-0</td>
<td>IF X</td>
</tr>
<tr>
<td>31</td>
<td>TF X</td>
</tr>
<tr>
<td>23</td>
<td>SF S</td>
</tr>
<tr>
<td>22</td>
<td>ZF S</td>
</tr>
<tr>
<td>21</td>
<td>AF S</td>
</tr>
<tr>
<td>20</td>
<td>PF S</td>
</tr>
<tr>
<td>19</td>
<td>CF S</td>
</tr>
<tr>
<td>18</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
</tr>
<tr>
<td>14-7</td>
<td></td>
</tr>
<tr>
<td>6-0</td>
<td></td>
</tr>
</tbody>
</table>

**S** = Status Flag  
**C** = Control Flag  
**X** = System Flag
BASIC INSTRUCTION SET
BASIC INSTRUCTION SET

- The 8086 instructions fall naturally into several subsets.

- The following list includes the most commonly used instructions.

- Please refer to the Programmer's Reference Manual for complete details.
BASIC INSTRUCTION SET

- **REPEATING: LOOP, LOOPE, LOOPNE**
  - Used to make program loops to repeat sections of code.

- **ARITHMETIC: ADD, SUB, CMP, INC, DEC**
  - Used for doing simple arithmetic. Refer to documentation for MUL and DIV instructions.

- **LOGIC: AND, OR, XOR, TEST, NOT**
  - Used for bit manipulation.

- **SHIFT AND ROTATE: SAL, SHR, ROL, RCR, etc.**
  - Used to multiply and divide by powers of 2 and to move bits from one part of an operand to another.
BASIC INSTRUCTION SET

CONTROL TRANSFER

- UNCONDITIONAL: JMP, CALL, RET, IRET
  - Used to transfer control, no choice.

- CONDITIONAL: JE, JNE, JZ, JNZ, JCXZ, etc.
  - Used to make decisions by checking the flags. Typically appear after a CMP or TEST instruction.

DATA TRANSFER: MOV, XCHG, IN, OUT

- Copying bytes, words, and dwords from place to place
BASIC INSTRUCTION SET

DATA TRANSFER--I/O PORTS: IN and OUT

- Most of the peripheral devices in a PC AT are located at ports.
  - This includes DMA, PIT, PIC, keyboard controller, real-time clock, display controller, com, printer, and digital control.
- You must be familiar with IN and OUT to communicate with these devices.
- All the default ports in a PC AT are byte-wide ports, so the instructions will always use the AL register.
  - Port address maps are looked at in another section.
DATA TRANSFER INSTRUCTIONS

DATA TRANSFER--REGISTERS AND MEMORY

• SYNTAX: MOV DEST, SRC

• MOV copies the SRC to the DEST

• The SRC can be I / R / M
  • I = number (immediate)
  • R = register
  • M = memory location

• The DEST can be R/M

• The TYPE of SRC and DEST must be the same (byte, word, dword).

• CANNOT "MOV" MEMORY TO MEMORY
Data Transfer: Registers and Memory

- The SRC and DEST operand size must be the same (byte, word, dword).

Examples:

- MOV CL, AL ; R to R, byte
- MOV AX, BX ; R to R, word
- MOV DX, [9A] ; M to R, word
- MOV EAX, [SI] ; M to R, dword
- MOV [EBX], SI ; R to M, word
- MOV EDX, 5 ; l to R, dword
- MOV word ptr [EBX], 10H ; l to M, word

- NOTE: Word Ptr (pointer) is used to force a word to be moved instead of a byte: Moves 0010H instead of 10H.
Data Transfer: Registers and Memory

- The offset of the data is typically added to the DSBase.
- MOV EAX, [EBX] is equivalent to MOV EAX, DS:[EBX]
  - Using DS: in this matter is not required.
- Example: Assume DS = 10, EBX = 50 (Real Mode)
  - MOV EAX, [EBX] = MOV EAX, DS:[EBX]
  - DS:[EBX] -> 10:50 -> 100 + 50 -> 150 Physical
  - A Dword of Data is copied from memory address 150H Physical into the EAX register.
- Note: You may occasionally see MOV EAX, ES:[EBX]
  - The "ES:" is a “segment override” telling the processor to use the ESBase and not the default DSBase.
MOV BX, 50
MOV [BX], AL

MOV SI, 500
MOV [SI], AX

NOTE: SETTING UP A SEGMENT REGISTER IS A 2 STEP OPERATION
MOV AX, 125H
MOV DS, AX

LOW MEMORY

HIGH MEMORY

DATA SEGMENT

1250H (DS = 125H)

1250H + 50H = 12A0H

1250H + 500H = 1750H

OFFSET 50H

MOV SI, 500
MOV [SI], AX

MOV BX, 50
MOV [BX], AL

1250H (DS = 125H)
Data Transfer: Registers and Memory

- **XCHG DEST, SRC:**

  - XCHG exchanges the SRC and DEST.
    - The rules for MOV apply to XCHG except no immediate is allowed.

- **Examples:**
  - XCHG AX, BX
  - XCHG EAX, [ESI]
DATA TRANSFER INSTRUCTIONS

- Intel processors implement a dedicated I/O space apart from memory address space.
  - This Isolated I/O (I/O mapped I/O) clearly separates Memory and I/O devices by using the IN/OUT instructions for I/O and the MOV instructions for Memory transfers.
  - This has the advantage of being able to easily distinguish memory & I/O instructions in programs.
- Memory mapped I/O (not implemented in Intel system designs) refers to the use of memory-type instructions to access both Memory & I/O devices
  - This has the advantage of being able to use the same powerful instructions for memory and for I/O transfers.
  - This has the disadvantage of reducing the number of memory locations available and may restrict future expansion.
DATA TRANSFER INSTRUCTIONS

- **Input/Output Instructions:**
  - IN (input) and OUT (output) instructions are data movement instructions for reading data in from an input port or sending data to an output port.
  - I/O Ports are the microprocessors means of communication with hardware devices other than memory.
  - I/O Ports can be thought of as a connection through which information passes to or from an I/O device such as the Keyboard or Serial Port.
DATA TRANSFER INSTRUCTIONS

- The (accumulator) AL, AX, or EAX is ALWAYS used by the IN and OUT instructions for transferring data to/from the port.
  
  - e.g. - IN AL, 64H

- The Port Address can be specified directly, or the DX register can be used to contain the Port Address. Using the DX register is REQUIRED if the Port Address is larger than 255 (FFH)
  
  - MOV DX, 3F8H
  
  - IN AL, DX ;Cannot use IN AL, 3F8H

- Up to 65,536 (64K) (FFFFH) Ports can be accessed using the 16 bit DX register.
DATA TRANSFER INSTRUCTIONS

I/O PORTS: INPUT INSTRUCTION--IN

- **IN AL/AX/EAX, PORT#** ;PORT# <0FFH, (256t)
- **IN AL/AX/EAX, DX** ;DX CONTAINS PORT #
  - ;PORT# >0FFH (256t)

  - AL FOR BYTE PORTS
  - AX FOR WORD PORTS
  - EAX FOR DWORD PORTS

Examples:

- **IN AL, 64H** ;GET STATUS BYTE
- **MOV DX, 3F8H** ;PORT# >0FFH (256t)
- **IN AL, DX** ;GET A BYTE FROM COM1
DATA TRANSFER INSTRUCTIONS

I/O PORTS: OUTPUT INSTRUCTION--OUT

- OUT PORT#, AL/AX/EAX ; PORT#<256
- OUT DX, AL/AX/EAX ; DX CONTAINS PORT # ; PORT#>0FFH (256t)

Examples:
- MOV AL, OD1H ; SEND POST CODE TO
- OUT 80H, AL ; PORT 80
- MOV DX,0CF8H ; PORT#>0FFH (256t)
- MOV EAX, 80000060
- OUT DX,EAX ; ADDRESS PCI CFG SPACE
ARITHMETIC INSTRUCTIONS

- **ADD DEST, SRC**
  - Most two operand instructions follow the same operand rules as the MOV instruction.
  - \( \text{DEST} = \text{SRC} + \text{DEST} \)
  - **Examples:**
    - ADD AL, 5
    - ADD [EBX], AX

- **SUB DEST, SRC**
  - \( \text{DEST} = \text{DEST} - \text{SEC} \)
  - **Example:**
    - SUB EAX, EBX
ARITHMETIC INSTRUCTIONS

- INC DEST
  - ADDS ONE TO THE DEST (R/M)
  - UPDATES FLAGS
  - Example:
    - INC DX

- DEC DEST
  - SUBTRACTS ONE FROM THE DEST
  - UPDATES FLAGS
  - OFTEN USED TO MAKE LOOPS
  - Example:
    - DEC BL
ARITHMETIC INSTRUCTIONS

CMP DEST, SRC

- Non-destructive SUB
- Typically used before conditional jump instructions.
- Flags updated by DEST-SRC
  - e.g. Zero Flag Set True (1) or False (0)

Examples:

- CMP AL, 55
- JE LABEL1 ; JUMP IF AL = 55

- DEC BL
- JNZ LABEL1 ; JUMP IF BL ! = 0 (!= mean NOT EQUAL)
Logic Instructions (Bit Manipulation)

LOGIC INSTRUCTIONS ARE USED TO

- The operand rules are like those for "MOV"
- SET BITS -- OR DEST,SRC
  - DEST = DEST OR SRC  (BIT BY BIT OR)
  - Examples:
    - OR AL, 00001000B ;SET BIT3 IN AL
- CLEAR BITS -- AND DEST,SRC
  - DEST = DEST AND SRC
  - Examples:
    - AND AL, 00001111B ;CLEAR the High Nibble of AL
Logic Instructions (Bit Manipulation)

LOGIC INSTRUCTIONS ARE USED TO

- TOGGLE BITS -- XOR DEST, SRC
  - DEST = DEST XOR SRC
  - Example:
    - XOR BX, OFH ;TOGGLE the Low 4 Bits of BX

- COMPLEMENT BITS -- NOT DEST
  - Example:
    - NOT EAX ;COMPLEMENT EAX
Logic Instructions (Bit Manipulation)

LOGIC INSTRUCTIONS ARE USED TO

- TEST BITS -- TEST DEST,SRC
  - Flags updated by (DEST AND SRC).
  - DEST not changed. (non destructive AND)
  - Often used before a conditional jump.
  - Example:
    - TEST AL,10000000B
    - JNZ BIT_SET ;JUMP IF BIT 7 SET
Shift and Rotate

CL Can Be Used for Multiple Shifts.

- SHL DEST, COUNT ; SHIFTS "COUNT" BITS
- SHL DEST, CL ; SHIFT COUNT IS IN CL

Examples:

- SHL EAX, 2 ; MULTIPLY EAX BY 4
- MOV CL, 4 ; DIVIDE EAX
- SAR EAX, CL ; BY 16

Logical Shifts: SHL, SHR -- Shifts 0 into the empty bit

Arithmetic Shifts: SAL, SAR -- Makes sure the sign is correct
Shift and Rotate

CL Can Be Used for Multiple Shifts.

- ROR DEST, COUNT ;ROTATES "Count" Bits
- ROR DEST, CL ;ROTATE Count Is in CL

Examples:
- ROR EDX,10H ;EXCHANGE Low and High ; Words of EDX

![Shift and Rotate Diagram]

- ROTATES
- ROL, ROR -- SIMPLY ROTATES THE OPERAND
- ROTATE WITH CARRY RCL, ROR -- THE CARRY IS INCLUDED IN THE ROTATION
Unconditional Control Transfer

- **JMP LABEL** - Control goes to the address of label.
  - Labels are names representing addresses and are used in the Source Code which is created with an editor.
  - Disassembled code shows only the address assigned to the label.
  - Program labels are followed by colons in the source code.
    » e.g. - `DELAY1: MOV ECX, 10000H`
  - We will look at examples in the Procedure section.

- **Examples:**
  - ITP uses $ to mean the address of the current instruction
  - `JMP $-0A45` : Typical Debug Tool Display
  - `JMP DELAY1` ; Only see if have Source Code
Unconditional Control Transfer

CALL PROC_NAME

• Control goes to the procedure.
• The return address is stored on the stack.
• Control transfers can be to the same or different segment.

• Examples:
  • CALL DELAY
  • CALL $+2A78
  • ITP uses $ to mean the address of the current instruction

• We will look at an example in the Procedure section.
Unconditional Control Transfer

- **RET**
  - Return from a Procedure.
    - The Return Address must be on the Stack.
  - We will look at an example in the Procedure section.

- **IRET**
  - Used to return from an Interrupt Service Routine.
    - The Return Address must be on the Stack.
    - Also restores the Flags from the Stack.
Conditional Jump Instructions

- Conditional jump instructions make decisions based on processor flags. The important flags for conditional jumps are:

  - **ZF -- ZERO FLAG**
    - Set when result of an operation gives zero, otherwise cleared
    - ZF=1 = True (Result is ZERO)
    - ZF=0 = False (Result is NOT ZERO)

  - **SF -- SIGN FLAG**
    - SF is a copy of the Most Significant Bit (Bit 7) of "Dest"

  - **CF -- CARRY FLAG**
    - Set when Carry Out of Most Significant Bit occurs.
    - Also changed during Shifts and Rotates.
Conditional Jump Instructions

- Several conditional jump instructions check ONLY one FLAG (zero, sign, carry, etc)
  - JZ, JNZ (also written JE, JNE), JS, JNS, JC, JNC
- Examples:
  - CMP AL, 40
  - JE AL_EQUALS_40
  - DEC AX
  - JNZ KEEP_LOOPING
  - SHL AX, 1
  - JC LABEL1
Conditional Jump Instructions

- Others allow us to compare numerical values
  - Comparison of Unsigned Numbers:
    - JA -- JUMP IF ABOVE
    - JAE -- JUMP IF ABOVE OR EQUAL
    - JB -- JUMP IF BELOW
    - JBE -- JUMP IF BELOW OR EQUAL
  - Comparisons of Signed Numbers:
    - These will be less commonly used in BIOS and POST.
    - JG -- JUMP IF GREATER
    - JGE -- JUMP IF GREATER OR EQUAL
    - JL -- JUMP IF LESS
    - JLE -- JUMP IF LESS OR EQUAL
Looping Instructions

- **LOOP LABELX**
  1. FIRST DECREMENTS ECX
  2. THEN GOES TO LABELX IF ECX != 0

Example of loop used to clear 8 Bytes of Memory:

```
MOV EBX, 0 ;Will Point to Location Zero in Memory
MOV ECX, 8 ;This Is Count of # of Byte to Clear
L1: ;THIS IS A LABEL
    MOV [EBX], 0 ;Clear Memory Pointed to by [EBX]
    INC EBX ;Point to Next Byte
    LOOP L1 ;DO UNTIL ECX=0
```
Looping Instructions

- VARIATIONS OF LOOP:
  - These Instructions Check the Zero Flag As Well As ECX.

- **LOOPE, LOOPZ**
  - LOOP WHILE EQUAL OR UNTIL ECX = 0

- **LOOPNE, LOOPNZ**
  - LOOP WHILE NOT EQUAL (ZERO) OR UNTIL ECX = 0
Looping Instructions

Example of Using LOOPNE to Scan Memory for an 'A'

- NOTE: ‘A’ = 41H

MOV EBX, -1
MOV ECX, 15

L1:
INC EBX ;POINT TO FIRST BYTE
CMP [EBX], 'A' ;IS BYTE = 'A'? 
LOOPNE L1 ;KEEP LOOKING UNTIL ECX=0 or FOUND
JE A_FOUND ;IF EQUAL, 'A' WAS FOUND
Procedure
Declarations
Procedure Declarations

○ “DELAY” IS THE PROCEDURE NAME, MADE UP BY THE PROGRAMMER.
○ "PROC" AND "ENDP" ARE REQUIRED RESERVED WORDS.
○ TO CAUSE A DELAY, INVOKE THE DELAY ROUTINE.
  • MOV BX,15
  • CALL DELAY
○ NOTE: If you are using an ITP, it would look more like this (labels do not appear in the disassembly).
  • MOV BX, 15
  • CALL $+238 ; location relative to current location
Procedure Declarations

Here is an example of a routine that simply delays for awhile. Assume that BX contains the number of delay loops to execute.

DELAY PROC [NEAR/FAR]

DELAY1:

- MOV ECX, 10000H ;LOAD Internal Loop Counter

DELAY2:

- DEC ECX ;SUBTRACT One
- JNZ DELAY2 ;DO It Again Unless ECX=0
- DEC BX ;HAVE We Done It Enough?
- JNZ DELAY1 ;BX! = 0, DO IT AGAIN
- RET

DELAY ENDP
ASSEMBLY LANGUAGE EXERCISE
ASSEMBLY LANGUAGE EXERCISE

20h:3Fh  Read Cmd byte
60h:7Fh  Write Cmd byte
AAh  Self test
ABh  Test interface
ACh  NO-OP
ADh  Disable keyboard
C0h  Read input port
D0h  Read output port
D1h  Write output port
E0h  Read test input port
E1h:EFh  Reserved
F0h:FFh  Output pulse

PORT 64 Write: Command Register

8742 CONTROLLER PORTS

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

PORT 64 Read: Status Register
ASSEMBLY LANGUAGE EXERCISE

The following is an example of a small assembly program that can be used to READ the keyboard controller input port.

The READ program does the following

1. Send a command to the keyboard controller.
2. Read the keyboard controller status from port 64H.
3. Test Bit 0 to see if keyboard data can be read.
   - Output Buffer Full Flag = 1 if Data is available to be read.
4. If Bit 0 is 1
   - Read the data at port 60H (Output buffer is full)
   - Else go back to step 1 and loop until data is ready.
» Bit 0 is 0
ASSEMBLY LANGUAGE EXERCISE

To read to the digital input port on the 8742

```assembly
MOV AL, 0C0H ;CMD - READ INPUT PORT
OUT 64H, AL ;SEND IT
L1: IN AL, 64H ;GET STATUS BYTE
TEST AL, 00000001B ;CAN WE READ IT?
JZ L1 ;NO (loop until ZF= 0[FALSE])
IN AL, 60H ;AL=DESIRED INPUT
```

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Parity error</td>
</tr>
<tr>
<td>6</td>
<td>Receive time-out</td>
</tr>
<tr>
<td>5</td>
<td>Transmit time-out</td>
</tr>
<tr>
<td>4</td>
<td>Keyboard unlocked</td>
</tr>
<tr>
<td>3</td>
<td>Last input: Cmd/Data</td>
</tr>
<tr>
<td>2</td>
<td>Self test flag</td>
</tr>
<tr>
<td>1</td>
<td>Input buffer full</td>
</tr>
<tr>
<td>0</td>
<td>Output buffer full</td>
</tr>
</tbody>
</table>

PORT 64 Read: Status Register
ASSEMBLY LANGUAGE EXERCISE

The following is an another example of a small assembly program that can be used to WRITE the keyboard controller output port.

The WRITE program does the following

1. Send a command to the keyboard controller.
2. Read the keyboard controller status from port 64H.
3. Test Bit 1 to see if keyboard data can be sent.
   • Input Buffer Full Flag = 0 if Data can be sent.
4. If Bit 1 is 0
   • Write the data at port 60H (Input buffer is not full)
   • Else go back to step 1 and loop until input buffer is empty
   » Bit 1 is 1
ASSEMBLY LANGUAGE EXERCISE

To write to the digital output port on the 8742

```
MOV   AL, OD1H ;CMD - WRITE OUTPUT PORT
OUT   64H, AL  ;SEND IT
L1:   IN    AL, 64H ;GET STATUS BYTE
      TEST  AL, 00000010B ;CAN WE SEND IT?
      JNZ   L1 ;NO (loop until ZF= 1[TRUE])
      MOV   AL, MYDATA ;DATA TO SEND
      OUT   60H, AL  ;SEND THE DATA
```

PORT 64 Read: Status Register

```
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
```

Parity error  Receive time-out  Transmitter time-out  Keyboard unlocked  Last input: Cmd/Data  Self test flag  Input buffer full  Output buffer full
ASSEMBLY LANGUAGE REVIEW
ASSEMBLY LANGUAGE REVIEW

Describe some of the bus cycles generated for the previous Write Output port example

MOV AL, OD1H
OUT 64H, AL :ADDR=64H, DATA=D1H, CTRL= I/O-Data-Write
L1: IN AL, 64H :ADDR=64H, DATA=??, CTRL= I/O-Data-Read
TEST AL, 00000010B
JNZ L1
MOV AL, MYDATA :ADDR=DS:MYDATA, DATA=(*) , CTRL=Mem-Data-Read
OUT 60H, AL :ADDR=60H, DATA=(*) , CTRL= I/O-Data-Write

NOTE: (*) = Contents of byte stored as DS:MYDATA
ASSEMBLY LANGUAGE REVIEW

CPU REGISTER TEST

F000:42C8  B0 01  post_01:  mov  al,1
F000:42CA  E6 80  out  80h,al  ; Port 80h
F000:42CC  B8 AA 55  mov  ax,55AAh
F000:42CF  8E D0  reg_test:  mov  ss,ax
F000:42D1  8C D6  mov  si,ss
F000:42D3  8B DE  mov  bx,si
F000:42D5  8E DB  mov  ds,bx
F000:42D7  8C DF  mov  di,ds
F000:42D9  8B CF  mov  cx,di
F000:42DB  8E C1  mov  es,cx
F000:42DD  8C C5  mov  bp,es
F000:42DF  8B D5  mov  dx,bp
F000:42E1  8B E2  mov  sp,dx
F000:42E3  3B C4  cmp  ax,sp
F000:42E5  74 01  je  pass  ; Jump equal
F000:42E7  F4  fail:  hlt  ; Halt fail
F000:42E8  F7 D0  pass:  not  ax
F000:42EA  3D AA 55  cmp  ax,55AAh
F000:42ED  75 E0  jne  reg_test  ; Jump not equal
ASSEMBLY LANGUAGE REVIEW

- The following is an example of an Assembly Language program and the associated “Debugger Dump” of memory for this program.

- You should be able to identify the following:
  - The Code & Data Segments in memory.
  - The relationship between the CS & DS Segment Registers and the memory display of the Code & Data Segments.
**ASSEMBLY LANGUAGE REVIEW**

Assembly Language Source Code created in an Editor.

<table>
<thead>
<tr>
<th>title</th>
<th>HELLO.ASM - Typical Minimal Assembly Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>.data</td>
<td></td>
</tr>
<tr>
<td>msg</td>
<td>db &quot;Hello world!$&quot; ;Data segment</td>
</tr>
<tr>
<td>.stack</td>
<td>100h ;256-word stack</td>
</tr>
<tr>
<td>.code</td>
<td></td>
</tr>
<tr>
<td>main:</td>
<td></td>
</tr>
<tr>
<td>mov ax,@data</td>
<td>;Point ds at data segment</td>
</tr>
<tr>
<td>mov ds,ax</td>
<td>; requires 2 instructions</td>
</tr>
<tr>
<td>mov dx,offset msg</td>
<td>;Set ds:dx =addr of msg</td>
</tr>
<tr>
<td>mov ah,9</td>
<td>;Here's the main program body</td>
</tr>
<tr>
<td>int 21h</td>
<td>;It's only 3 lines long!</td>
</tr>
<tr>
<td>mov ax,4c00h</td>
<td>;Uses MS-DOS function 9</td>
</tr>
<tr>
<td>int 21h</td>
<td>;to display msg</td>
</tr>
<tr>
<td>end main</td>
<td>;ah = code to return to DOS</td>
</tr>
<tr>
<td></td>
<td>;al = 0, i.e., no error</td>
</tr>
</tbody>
</table>
ASSEMBLY LANGUAGE REVIEW

(Code Segment - Starts @ 8AA10 physical)

| cs:0000 B8A38A | main: mov ax,@data ;Point ds at data | ax 4C00 |
| cs:0003 8ED8 | mov ds,ax | bx 8AA1 |
| cs:0005 BA0000 | mov dx,offset msg ;Here's the main | cx 5CBC |
| cs:0008 B409 | mov ah,9 ;It's only 3 lines long! | dx 0000 |
| cs:000A CD21 | int 21h ;Uses MS-DOS function 9 | si 8AA1 |
| cs:000C B8004C | mov ax,4c00h;ah=code to return to DOS | di 5CBC |
| cs:000F CD21 | int 21h ;al = 0, i.e., no error | bp 0100 |
| cs:0011 0000 | add [bx+si],al | sp 0100 |

----------- Dump of DATA Segment (8AA30 physical) -----------

ds:0000 48 65 6C 6C 6F 20 77 6F Hello wo |

ds:0008 72 6C 64 21 24 00 00 00 rld!$ |

ds:0010 00 00 00 00 00 00 00 00 |

----------- Dump starting @ 8AA10 physical -----------

8AA0::0010 B8 A3 8A 8E D8 BA 00 00 +úèÄ+ |

8AA0::0018 B4 09 CD 21 B8 00 4C CD -!+ L- |

8AA0::0020 21 00 00 00 00 00 00 00 |

8AA0::0028 00 00 00 00 00 00 00 00 |

8AA0::0030 48 65 6C 6F 20 77 6F Hello wo |

+---------------------------------------------------------------- ---------------------------------------------+
SUMMARY

WE HAVE DISCUSSED THE FOLLOWING:

- Various terms associated with Assembly Language.
- Little-Endian Addressing
- 80x86 Real Mode Registers
- The use of Segments and Offsets.
- How addresses are generated in Real Mode.
- The Data Transfer, Arithmetic, Logic, Shift & Rotate, Control & Transfer, & Repeating Instruction set.
- The use of Procedures in Assembly Language.
- Examples of Assembly Language programs from BIOS & POST.
### ANSWERS TO EXERCISE

- Follow the first four examples, then compute the second four yourself:

<table>
<thead>
<tr>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000:0</td>
<td>0000</td>
</tr>
<tr>
<td>1234:5678</td>
<td>5678</td>
</tr>
<tr>
<td>F000:FFF0</td>
<td>FFF0</td>
</tr>
<tr>
<td>8110:091C</td>
<td>091C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>10000</td>
</tr>
<tr>
<td>1234</td>
<td>179B8</td>
</tr>
<tr>
<td>F000</td>
<td>FFFF0</td>
</tr>
<tr>
<td>8110</td>
<td>81A1C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFFF</td>
<td>FFFF0</td>
</tr>
<tr>
<td>8000</td>
<td>81A1C</td>
</tr>
<tr>
<td>F400</td>
<td>102000</td>
</tr>
<tr>
<td>168A</td>
<td>179B8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>1A1C</td>
<td>1A1C</td>
</tr>
<tr>
<td>E000</td>
<td>E000</td>
</tr>
<tr>
<td>1118</td>
<td>1118</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFFF0</td>
<td>FFFF0</td>
</tr>
<tr>
<td>81A1C</td>
<td>81A1C</td>
</tr>
<tr>
<td>102000</td>
<td>102000</td>
</tr>
<tr>
<td>179B8</td>
<td>179B8</td>
</tr>
</tbody>
</table>