Outline

- Why Assembly Language Programming
- Organization of 8086 processor
- Assembly Language Syntax
- Data Representation
- Variable Declaration
- Instruction Types
  - Data flow instructions
  - Arithmetic instructions
  - Bit manipulation instructions
  - Flow control instructions
- Memory Segmentation
Outline – Cont.

- Program Structure
- Addressing Modes
- Input and Output
- The stack
- Procedures
- Macros
- String Instructions
- BIOS and DOS Interrupts

Machine/Assembly Language

- **Machine Language:**
  - Set of fundamental instructions the machine can execute
  - Expressed as a pattern of 1’s and 0’s

- **Assembly Language:**
  - Alphanumeric equivalent of machine language
  - Mnemonics more human-oriented than 1’s and 0’s

- **Assembler:**
  - Computer program that transliterates (one-to-one mapping) assembly to machine language
  - Computer’s native language is machine/assembly language
Why Assembly Language Programming

- Faster and shorter programs.
  - Compilers do not always generate optimum code.
- Instruction set knowledge is important for machine designers.
- Compiler writers must be familiar with details of machine language.
- Small controllers embedded in many products
  - Have specialized functions,
  - Rely so heavily on input/output functionality,
  - HLLs inappropriate for product development.

Programmer’s Model: Instruction Set Architecture

- Instruction set: collection of all machine operations.
- Programmer sees set of instructions, and machine resources manipulated by them.
- ISA includes
  - Instruction set,
  - Memory, and
  - Programmer-accessible registers.
- Temporary or scratch-pad memory used to implement some functions is not part of ISA
  - Not programmer accessible.
Organization of 8086 Processor

CPU-Memory Interface

CPU Registers

- Fourteen 16-bit registers
- Data Registers
  - AX (Accumulator Register): AH and AL
  - BX (Base Register): BH and BL
  - CX (Count Register): CH and CL
  - DX (Data Register): DH and DL
- Pointer and Index Registers
  - SI (Source Index)
  - DI (Destination Index)
  - SP (Stack Pointer)
  - BP (Base Pointer)
  - IP (Instruction Pointer)
CPU Registers – Cont.

- **Segment Registers**
  - CS (Code Segment)
  - DS (Data Segment)
  - SS (Stack Segment)
  - ES (Extra Segment)

- **FLAGS Register**
  - Zero flag
  - Sign flag
  - Parity flag
  - Carry flag
  - Overflow flag

Fetch-Execute Process

- **Program Counter (PC) or Instruction Pointer (IP)**
  - Holds address of next instruction to fetch

- **Instruction Register (IR)**
  - Stores the instruction fetched from memory

- **Fetch-Execute process**
  - Read an instruction from memory addressed by PC
  - Increment program counter
  - Execute fetched instruction in IR
  - Repeat process
Assembly Language Syntax

- Program consists of statement per line.
- Each statement is an instruction or assembler directive
- Statement syntax
  - Name operation operand(s) comment
- Name field
  - Used for instruction labels, procedure names, and variable names.
  - Assembler translates names into memory addresses
  - Names are 1-31 characters including letters, numbers and special characters: `@` `_$`
  - Names may not begin with a digit
  - If a period is used, it must be first character
  - Case insensitive
Assembly Language Syntax – Cont.

- **Examples of legal names**
  - COUNTER1
  - @character
  - SUM_OF_DIGITS
  - $1000
  - Done?
  - .TEST

- **Examples of illegal names**
  - TWO WORDS
  - 2abc
  - A45.28
  - You&Me

Assembly Language Syntax – Cont.

- **Operation field**
  - instruction
    - Symbolic operation code (opcode)
    - Symbolic opcodes translated into machine language opcode
    - Describes operation’s function; e.g. MOV, ADD, SUB, INC.
  - Assembler directive
    - Contains pseudo-operation code (pseudo-op)
    - Not translated into machine code
    - Tell the assembler to do something.

- **Operand field**
  - Specifies data to be acted on
  - Zero, one, or two operands
    - NOP
    - INC AX
    - ADD AX, 2
Assembly Language Syntax – Cont.

- **Comment field**
  - A semicolon marks the beginning of a comment
  - A semicolon in beginning of a line makes it all a comment line
  - Good programming practice dictates comment on every line

- **Examples**
  - MOV CX, 0 ; move 0 to CX
    - Do not say something obvious
  - MOV CX, 0 ; CX counts terms, initially 0
    - Put instruction in context of program
  - ; initialize registers

Data Representation

- **Numbers**
  - 11011 decimal
  - 11011B binary
  - 64223 decimal
  - -21843D decimal
  - 1,234 illegal, contains a nondigit character
  - 1B4DH hexadecimal number
  - 1B4D illegal hex number, does not end with “H”
  - FFFFH illegal hex numbe, does not begin with with digit
  - 0FFFFH hexadecimal number

- **Signed numbers represented using 2’s complement.**
Data Representation - Cont.

- Characters
  - must be enclosed in single or double quotes
  - e.g. “Hello”, ‘Hello’, “A”, ‘B’
  - encoded by ASCII code
- ‘A’ has ASCII code 41H
- ‘a’ has ASCII code 61H
- ‘0’ has ASCII code 30H
- Line feed has ASCII code 0AH
- Carriage Return has ASCII code 0DH
- Back Space has ASCII code 08H
- Horizontal tab has ASCII code 09H

Data Representation - Cont.

- The value of the content of registers or memory is dependent on the programmer.
- Let AL=FFH
  - represents the unsigned number 255
  - represents the signed number -1 (in 2’s complement)
- Let AH=30H
  - represents the decimal number 48
  - represents the character ‘0’
- Let BL=80H
  - represents the unsigned number +128
  - represents the signed number -128
Variable Declaration

- Each variable has a type and assigned a memory address.
- Data-defining pseudo-ops
  - DB      define byte
  - DW      define word
  - DD      define double word (two consecutive words)
  - DQ      define quad word (four consecutive words)
  - DT      define ten bytes (five consecutive words)
- Each pseudo-op can be used to define one or more data items of given type.

Byte Variables

- Assembler directive format defining a byte variable
  - name DB    initial value
  - a question mark (“?”) place in initial value leaves variable uninitialized
- I DB 4    define variable I with initial value 4
- J DB ?    Define variable J with uninitialized value
- Name DB “Course” allocate 6 bytes for Name
- K DB 5, 3, -1 allocates 3 bytes

\[ K \begin{array}{c}
05 \\
03 \\
FF 
\end{array} \]
Word Variables

- Assembler directive format defining a word variable
  - name DW initial value

- I DW 4
  - I → 04
  - 00

- J DW -2
  - J → FE
  - FF

- K DW 1ABCH
  - K → BC
  - 1A

- L DW “01”
  - L → 31
  - 30

Double Word Variables

- Assembler directive format defining a word variable
  - name DD initial value

- I DD 1FE2AB20H
  - I → 20
  - AB
  - E2
  - 1F

- J DD -4
  - J → FC
  - FF
  - FF
  - FF
Named Constants

- EQU pseudo-op used to assign a name to constant.
- Makes assembly language easier to understand.
- No memory allocated for EQU names.
- LF EQU 0AH
  - MOV DL, 0AH
  - MOV DL, LF
- PROMPT EQU “Type your name”
  - MSG DB “Type your name”
  - MDG DB PROMPT

DUP Operator

- Used to define arrays whose elements share common initial value.
- It has the form: repeat_count DUP (value)
- Numbers DB 100 DUP(0)
  - Allocates an array of 100 bytes, each initialized to 0.
- Names DW 200 DUP(?)
  - Allocates an array of 200 uninitialized words.
- Two equivalent definitions
  - Line DB 5, 4, 3 DUP(2, 3 DUP(0), 1)
  - Line DB 5, 4, 2, 0, 0, 0, 1, 2, 0, 0, 0, 1, 2, 0, 0, 0, 1
Instruction Types

- **Data transfer instructions**
  
  - Transfer information between registers and memory locations or I/O ports.
  
  - MOV, XCHG, LEA, PUSH, POP, PUSHF, POPF, IN, OUT.

- **Arithmetic instructions**
  
  - Perform arithmetic operations on binary or binary-coded-decimal (BCD) numbers.
  
  - ADD, SUB, INC, DEC, ADC, SBB, NEG, CMP, MUL, IMUL, DIV, IDIV, CBW, CWD.

- **Bit manipulation instructions**
  
  - Perform shift, rotate, and logical operations on memory locations and registers.
  
  - SHL, SHR, SAR, ROL, ROR, RCL, RCR, NOT, AND, OR, XOR, TEST.

Instruction Types – Cont.

- **Control transfer instructions**
  
  - Control sequence of program execution; include jumps and procedure transfers.
  
  - JMP, JG, JL, JE, JNE, JGE, JLE, JNG, JC, JS, JA, JB, JAE, JBE, JNB, JNA, JO, JZ, JNZ, JP, JCXZ, LOOP, LOOPE, LOOPZ, LOOPNE, LOOPNZ, CALL, RET.

- **String instructions**
  
  - Move, compare, and scan strings of information.
  
  - MOV, MOVSB, MOVSW, CMPS, CMPSB, CMPSW, SCAS, SCASB, SCASW, LODS, LODSB, LODSW, STOS, STOSB, STOSW.
Instruction Types – Cont.

- **Interrupt instructions**
  - Interrupt processor to service specific condition.
  - INT, INTO, IRET.

- **Processor control instructions**
  - Set and clear status flags, and change the processor execution state.
  - STC, STD, STI.

- **Miscellaneous instructions**
  - NOP, WAIT.

General Rules

- **Both operands have to be of the same size.**
  - MOV AX, BL illegal
  - MOV AL, BL legal
  - MOV AH, BL legal

- **Both operands cannot be memory operands simultaneously.**
  - MOV i, j illegal
  - MOV AL, i legal

- **First operand cannot be an immediate value.**
  - ADD 2, AX illegal
  - ADD AX, 2 legal
Memory Segmentation

- A memory segment is a block of $2^{16}$ (64K) bytes.
- Each segment is identified by a segment number
  - Segment number is 16 bits (0000 - FFFF).
- A memory location is specified by an offset within a segment.
- Logical address: segment:offset
  - A4FB:4872h means offset 4872h within segment A4FBh.
- Physical address: segment * 10H + offset
  - A4FB*10h + 4872 = A4FB0 + 4872 = A9822h (20-bit address)
- Physical address maps to several logical addresses
  - physical address 1256Ah=1256:000Ah=1240:016Ah

Memory Segmentation - Cont.

- Location of segments
  - Segment 0 starts at address 0000:0000=00000h and ends at 0000:FFFF=0FFFFh.
  - Segment 1 starts at address 0001:0000=00010h and ends at 0001:FFFF= 1000Fh.
  - Segments overlap.
  - The starting physical address of any segment has the first hex digit as 0.
- Program segments
  - Program’s code, data, and stack are loaded into different memory segments, namely code segment, data segment and stack segment.
  - At any time, only four memory segments are active.
  - Program segment need not occupy entire 64K byte.
Memory Segmentation - Cont.

- **Data Segment**
  - contains variable definitions
  - declared by .DATA

- **Stack segment**
  - used to store the stack
  - declared by .STACK size
  - default stack size is 1Kbyte.

- **Code segment**
  - contains program's instructions
  - declared by .CODE

Memory Models

- **SMALL**
  - code in one segment & data in one segment

- **MEDIUM**
  - code in more than one segment & data in one segment

- **COMPACT**
  - code in one segment & data in more than one segment

- **LARGE**
  - code in more than one segment & data in more than one segment & no array larger than 64K bytes

- **HUGE**
  - code in more than one segment & data in more than one segment & arrays may be larger than 64K bytes
Program Structure: An Example

TITLE PRGM1
.MODEL SMALL
.STACK 100H
.DATA
   A   DW 2
   B   DW 5
   SUM DW ?
.CODE
MAIN PROC
 ; initialize DS
   MOV AX, @DATA
   MOV DS, AX

; add the numbers
   MOV AX, A
   ADD AX, B
   MOV SUM, AX

; exit to DOS
   MOV AX, 4C00H
   INT 21H
MAIN ENDP
END MAIN
Assembling & Running A Program

- **Assembling a program**
  - Use microsoft macro assembler (MASM)
  - MASM PRGM1.ASM
    - Translates the assembly file PROG1.ASM into machine language object file PROG1.OBJ
    - Creates a listing file PROG1.LST containing assembly language code and corresponding machine code.

- **Linking a program**
  - The .OBJ file is a machine language file but cannot be run
  - Some addresses not filled since it is not known where a program will be loaded in memory.
  - Some names may not have been defined.
  - Combines one or more object files and creates a single executable file (.EXE).
  - LINK PROG1

- **Running a program**
  - Type the name of the program to load it and run it

- **Simplified procedure**
  - MI /FI /ZI PROG1.ASM
  - Assembles and links the program

- **Debugging a program**
  - To analyze a program use CODE View debugger.
  - CV PROG1
Addressing Modes

- Addressing mode is the way an operand is specified.
- Register mode
  - operand is in a register
  - MOV AX, BX
- Immediate mode
  - operand is constant
  - MOV AX, 5
- Direct mode
  - operand is variable
  - MOV AL, i

Addressing Modes - Cont.

- Register indirect mode
  - offset address of operand is contained in a register.
  - Register acts as a pointer to memory location.
  - Only registers BX, SI, DI, or BP are allowed.
  - For BX, SI, DI, segment number is in DS.
  - For BP, segment number is in SS.
  - Operand format is [register]

- Example: suppose SI=0100h and [0100h]=1234h
  - MOV AX, SI \( AX=0100h \)
  - MOV AX, [SI] \( AX=1234h \)
Addressing Modes - Cont.

- **Based & Indexed addressing modes**
  - operand’s offset address obtained by adding a displacement to the content of a register

- **Displacement may be:**
  - offset address of a variable
  - a constant (positive or negative)
  - offset address of a variable plus or minus a constant

- **Syntax of operand**
  - [register + displacement]
  - [displacement + register]
  - [register] + displacement
  - displacement + [register]
  - displacement [register]

Addressing Modes - Cont.

- **Based addressing mode**
  - If BX or BP used

- **Indexed addressing mode**
  - If SI or DI used

- **Examples:**
  - MOV AX, W [BX]
  - MOV AX, [W+BX]
  - MOV AX, [BX+W]
  - MOV AX, W+[BX]
  - MOV AX, [BX]+W

- **Illegal examples:**
  - MOV AX, [BX]2
  - MOV BX, [AX+1]
Addressing Modes - Cont.

- **Based-Indexed mode:** offset address is the sum of
  - contents of a base register (BX or BP)
  - contents of an index register (SI or DI)
  - optionally, a variable’s offset address
  - optionally, a constant (positive or negative)

- **Operand may be written in several ways**
  - variable[base_register][index_register]
  - [base-register + index_register + variable + constant]
  - variable [base_register + index_register + constant]
  - constant [base_register + index_register + variable]

- **Useful for accessing two-dimensional arrays**

---

PTR Operator

- **Used to override declared type of an address expression.**

- **Examples:**
  - MOV [BX], 1   illegal, there is ambiguity
  - MOV Bye PTR [BX], 1   legal
  - MOV WORD PTR [BX], 1   legal

- **Let j be defined as follows**
  - j DW 10
  - MOV AL, j   illegal
  - MOV AL, Byte PTR J   legal
Input and Output

- CPU communicates with peripherals through I/O registers called I/O ports.
- Two instructions access I/O ports directly: IN and OUT.
  - Used when fast I/O is essential, e.g. games.
- Most programs do not use IN/OUT instructions
  - port addresses vary among computer models
  - much easier to program I/O with service routines provided by manufacturer
- Two categories of I/O service routines
  - Basic input/output system (BIOS) routines
  - Disk operating system (DOS) routines
- DOS and BIOS routines invoked by INT (interrupt) instruction.

System BIOS

- A set of programs always present in system
- BIOS routines most primitive in a computer
  - Talks directly to system hardware
  - Hardware specific - must know exact port address and control bit configuration for I/O devices
- BIOS supplied by computer manufacturer and resides in ROM
- Provides services to O.S. or application
- Enables O.S. to be written to a standard interface
Input/Output - Cont.

- INT 21H used to invoke a large number of DOS function.
- Type of called function specified by putting a number in AH register.
  - AH=1 single-key input with echo
  - AH=2 single-character output
  - AH=9 character string output
  - AH=8 single-key input without echo
  - AH=0Ah character string input
Single-Key Input

- Input: AH=1
- Output: AL= ASCII code if character key is pressed, otherwise 0.
- To input character with echo:
  • MOV AH, 1
  • INT 21H ; read character will be in AL register
- To input a character without echo:
  • MOV AH, 8
  • INT 21H ; read character will be in AL register

Single-Character Output

- Input: AH=2, DL= ASCII code of character to be output
- Output: AL=ASCII code of character
- To display a character
  • MOV AH, 2
  • MOV DL, ‘?’ ; displaying character ‘?’
  • INT 21H
- To read a character and display it
  • MOV AH, 1
  • INT 21H
  • MOV AH, 2
  • MOV DL, AL
  • INT 21H
Displaying a String

- Input: AH=9, DX= offset address of a string.
- String must end with a ‘$’ character.
- To display the message Hello!
  - MSG DB “Hello!$”
  - MOV AH, 9
  - MOV DX, offset MSG
  - INT 21H
- OFFSET operator returns the address of a variable
- The instruction LEA (load effective address) loads destination with address of source
  - LEA DX, MSG

Inputting a String

- Input: AH=10, DX= offset address of a buffer to store read string.
  - First byte of buffer should contain maximum string size+1
  - Second byte of buffer reserved for storing size of read string.
- To read a Name of maximum size of 20 & display it
  - Name DB 21,0,22 dup(“$”)
  - MOV AH, 10
  - LEA DX, Name
  - INT 21H
  - MOV AH, 9
  - LEA DX, Name+2
  - INT 21H
A Case Conversion Program

- Prompt the user to enter a lowercase letter, and on next line displays another message with letter in uppercase.
  - Enter a lowercase letter: a
  - In upper case it is: A

 DATA
- CR EQU 0DH
- LF EQU 0AH
- MSG1 DB 'Enter a lower case letter: $'
- MSG2 DB CR, LF, 'In upper case it is: '
- Char DB ?, '$'

A Case Conversion Program - Cont.

 CODE
- .STARTUP ; initialize data segment
- LEA DX, MSG1 ; display first message
- MOV AH, 9
- INT 21H
- MOV AH, 1 ; read character
- INT 21H
- SUB AL, 20H ; convert it to uppercase
- MOV CHAR, AL ; and store it
- LEA DX, MSG2 ; display second message and
- MOV AH, 9 ; uppercase letter
- INT 21H
- .EXIT ; return to DOS
Status & Flags Register

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<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>OF</td>
<td>DF</td>
<td>IF</td>
<td>TF</td>
<td>SF</td>
<td>ZF</td>
<td>AF</td>
<td>PF</td>
<td>CF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Carry flag (CF):** CF=1 if there is
  - a carry out from most significant bit (msb) on addition
  - a borrow into msb on subtraction
  - CF also affected differently by shift and rotate instructions

- **Parity flag (PF):** PF=1 if
  - low byte of result has an even number of one bits (even parity)

Status & Flags Register - Cont.

- **Auxiliary carry flag (AF):** AF=1 if there is
  - a carry out from bit 3 on addition
  - a borrow into bit 3 on subtraction

- **Zero flag (ZF):** ZF=1
  - if the result is zero

- **Sign flag (SF):** SF=1 if
  - msb of result is 1 indicating that the result is negative for signed number interpretation

- **Overflow flag (OF):** OF=1
  - if signed overflow occurs
How Processor Indicates Overflow

- **Unsigned overflow**
  - occurs when there is a carry out of msb

- **Signed overflow occurs**
  - on addition of numbers with same sign, when sum has a different sign.
  - on subtraction of numbers with different signs, when result has a different sign than first number.
  - If the carries into and out of msb are different.

**Example:**

```
FFFF + FFFF
--------
1 FFFFe
```

SF=1  PF=0  ZF=0  CF=1  OF=0

MOV Instruction

- **Syntax:** MOV destination, source
  - Destination ← source

- **Transfer data between**
  - Two registers
  - A register and a memory location
  - A constant to a register or memory location

<table>
<thead>
<tr>
<th>Source</th>
<th>General Register</th>
<th>Segment Register</th>
<th>Memory Location</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Register</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Segment Register</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Memory Location</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>
MOV Instruction – Cont.

- MOV instruction has no effect on flags.

- Examples:
  - MOV DS, @Data illegal
  - MOV DS, ES illegal
  - MOV [BX], -1 illegal
  - MOV [DI], [SI] illegal
  - MOV AL, offset I illegal
  - MOV [BX], offset I illegal
  - MOV [SI], I illegal
  - MOV DS, [BX] legal
  - MOV AX, [SI] legal
  - MOV [BX-1], DS legal

XCHG Instruction

- Syntax: XCHG operand1, operand2
  - Operand1 ← operand2
  - Operand2 ← operand1

- Exchanges contents of two registers, or a register and a memory location.

<table>
<thead>
<tr>
<th></th>
<th>General Register</th>
<th>Memory Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Register</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Memory Location</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

- XCHG has no effect on flags.
ADD & SUB Instructions

- Syntax:
  - ADD destination, source ; destination=destination+ source
  - SUB destination, source ; destination=destination-source

<table>
<thead>
<tr>
<th>General Register</th>
<th>Memory Location</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

- ADD and SUB instructions affect all the flags.

INC & DEC Instructions

- Syntax:
  - INC operand ; operand=operand+1
  - DEC operand ; operand=operand-1

- Operand can be a general register or memory.

- INC and DEC instructions affect all the flags.

- Examples:
  - INC AX legal
  - DEC BL legal
  - INC [BX] illegal
  - INC Byte PTR [BX] legal
  - DEC I legal
  - INC DS illegal
**NEG instruction**

- Syntax: `NEG operand`
  - `Operand ← 0 – operand`
- Finds the two’s complement of operand.
- Operand can be a general register or memory location.
- NEG instruction affects all flags.
- Examples:
  - Let `AX=FFF0h` and `I=08h`
  - `NEG AX ; AX←0010`
  - `NEG AH ; AH←01`
  - `NEG I ; I←F8`

---

**CMP instruction**

- Syntax: `CMP operand1, operand2`
  - `Operand1-operand2`
- Subtracts operand2 from operand1 and updates the flags based on the result.
- CMP instruction affects all the flags.

<table>
<thead>
<tr>
<th></th>
<th>General Register</th>
<th>Memory Location</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Register</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Memory Location</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>
ADC and SBB instruction

- Syntax:
  - ADC destination, source ; destination=destination+source+CF
  - SBB destination, source ; destination=destination-source-CF
- Achieve double-precision addition/subtraction.
- To add or subtract 32-bit numbers
  - Add or subtract lower 16 bits
  - Add or subtract higher 16 bits with carry or borrow
- Example: Add the two double words in A and B
  - MOV AX, A
  - MOV DX, A+2
  - ADD B, AX
  - ADC B+2, DX

Multiplication

- Unsigned multiplication: MUL operand
- Signed multiplication: IMUL operand
- If operand is a Byte
  - MUL operand; AX ← AL * operand
- If operand is a Word
  - MUL operand; DX:AX ← AX * operand
- Operand can be a general register or memory. Cannot be a constant.
- Flags SF, ZF, AF, and PF are undefined.
- Only CF and OF are affected.
Multiplication – Cont.

- **CF=OF=0**
  - Unsigned multiplication: if upper half of result is 0.
  - Signed multiplication: if upper half of result is a sign extension of lower half.

- **Example: Let AX=FFFFh and BX=0002h**
  - MUL BL; AX←01FEh (255 * 2 = 510) CF=OF=1
  - IMUL BL; AX←FFFEh (-1 * 2 = -2) CF=OF=0
  - MUL AL; AX←FE01 (255 * 255 = 65025) CF=OF=1
  - IMUL AL; AX←0001 (-1 * -1 = 1) CF=OF=0
  - MUL BX; DX←0001 AX←FFFE CF=OF=1
  - IMUL BX; DX←FFFF AX←FFFE CF=OF=0

---

Application: Inputting a Decimal Number

- **Inputting a 2-digit decimal number**

```
MOV AH, 1 ;read first digit
INT 21H
SUB AL, '0' ; convert digit from ASCII code to binary
MOV BL, 10
MUL BL ; multiply digit by 10
MOV CL, AL
MOV AH, 1 ; read 2nd digit
INT 21H
SUB AL, '0' ; convert digit from ASCII code to binary
ADD AL, CL ; AL contains the 2-digit number
```
Division

- Unsigned division: DIV operand
- Signed division: IDIV operand

If operand is a Byte
- DIV Operand; AX ← AX/operand
  • AH= Remainder, AL= Quotient
If operand is a Word
- DIV Operand; DX:AX ← DX:AX/operand
  • DX=Remainder, AX= Quotient

Operand can be a general register or memory. Cannot be a constant.
All flags are undefined.

Division - Cont.

- Divide Overflow
  • If quotient is too big to fit in specified destination (AL or AX)
  • Happens if divisor much smaller than dividend
  • Program terminates and displays “Divide Overflow”

Example: Let DX=0000h, AX=0005h, and BX=FFFFh
  • DIV BX; AX=0000 DX=0005
  • IDIV BX; AX=FFFF DX=0001

Example: Let DX=FFFFh, AX=FFFBh, and BX=0002h
  • IDIV BX; AX=FFFF DX=FFFF
  • DIV BX; DIVIDE Overflow

Example: Let AX=00FBh (251), and BL=FFh
  • DIV BL; AH=FB AL=00
  • IDIV BL; DIVIDE Overflow
Application: Outputting a Decimal Number

- Outputting a 2-digit decimal number in AX

```plaintext
MOV BL, 10
DIV BL          ; getting least significant digit
ADD AH, '0'     ; converting L.S. digit to ASCII
MOV DH, AH      ; storing L.S. digit temporarily
MOV AH, 0
DIV BL          ; getting most significant digit
ADD AH, '0'     ; converting M.S. digit into ASCII
MOV DL, AH      ; displaying M.S. digit
MOV AH, 2
INT 21H
MOV DL, DH      ; displaying least significant digit
INT21H
```

Logic Instructions

- The AND, OR, and XOR instructions perform named bit-wise logical operation.

- Syntax:
  - AND destination, source
  - OR destination, source
  - XOR destination, source

```
10101010  10101010  10101010
AND 11110000  OR 11110000  XOR 11110000
------------------  ------------------  ------------------
10100000  11111010  01011010
```

COE-KFUPM
Logic Instructions - Cont.

- AND instruction used to clear specific destinations bits while preserving others.
  - A 0 mask bit clears corresponding destination bit
  - A 1 mask bit preserves corresponding destination bit

- OR instruction used to set specific destinations bits while preserving others.
  - A 1 mask bit sets corresponding destination bit
  - A 0 mask bit preserves corresponding destination bit

- XOR instruction used to complement specific destinations bits while preserving others.
  - A 1 mask bit complements corresponding destination bit
  - A 0 mask bit preserves corresponding destination bit

Logic Instructions - Cont.

- Effect on flags
  - SF, ZF, PF change based on result
  - AF undefined
  - CF=OF=0

- Examples:
  - Converting ASCII digit to a number
    - SUB AL, 30h
    - AND AL, 0Fh
  - Converting a lowercase letter to uppercase
    - SUB AL, 20h
    - AND AL, 0DFh
  - Initializing register with 0
    - XOR AL, AL
Logic Instructions - Cont.

- **NOT instruction**
  - performs one’s complement operation on destination
  - Syntax: NOT destination
  - has no effect on flags.

- **TEST instruction**
  - performs an AND operation of destination with source but does not change destination
  - It affects the flags like the AND instruction
  - used to examine content of individual bits

- **Example**
  - To test for even numbers
  - TEST AL, 1; if ZF=1, number is even

Shift & Rotate Instructions

- **Shift bits in destination operand by one or more bit positions either to left or to right.**
  - For shift instructions, shifted bits are lost
  - For rotate instructions, bits shifted out from one end are put back into other end

- **Syntax:**
  - Opcode destination, 1 ; for single-bit shift or rotate
  - Opcode destination, CL ; for shift or rotate of N bits

- **Shift Instructions:**
  - SHL/SAL: shift left (shift arithmetic left)
  - SHR: shift right
  - SAR: shift arithmetic right
Shift & Rotate Instructions - Cont.

- **Rotate instructions**
  - ROL: rotate left
  - ROR: rotate right
  - RCL: rotate left with carry
  - RCR: rotate right with carry

- **Effect on flags (shift & rotate instructions):**
  - SF, PF, ZF change based on result
  - AF undefined
  - CF= last bit shifted
  - OF=1 if sign bit changes on single-bit shifts

---

Shift & Rotate Instructions - Cont.

- **Examples: Let AL=FFh**
  - SHR AL, 1 ; AL ← 7Fh
  - SAR AL, 1 ; AL ← FFh
  - SHL AL, 1 ; AL ← FEh
  - SAL AL, 1 ; AL ← FEh

- **Examples: Let AL=0Bh and CL=02h**
  - SHL AL, 1 ; AL ← 16h
  - SHL AL, CL ; AL ← 2Ch
  - SHR AL, 1 ; AL ← 05h
  - SHR AL, CL ; AL ← 02h
  - ROL AL, 1 ; AL ← 16h
  - ROR AL, 1 ; AL ← 85h
Multiplication & Division by Shift

- Multiplication by left shift
  - A left shift by 1 bit doubles the destination value, i.e. multiplies it by 2.

- Division by right shift
  - A right shift by 1 bit halves it and rounds down to the nearest integer, i.e. divides it by 2.

Example: Multiply signed content of AL by 17
- MOV AH, AL
- MOV CL, 4
- SAL AL, CL ; AL = 16*AL
- ADD AL, AH; AL = 16*AL + AL = 17 AL

Flow Control Instructions

- Unconditional jump
  - JMP label ; IP ← label

- Conditional jump
  - Signed jumps
  - Unsigned jumps
  - Common jumps

- Signed jumps
  - JG/JNLE jump if greater than, or jump if not less than or equal
  - JGE/JNL jump if greater than or equal, or jump if not less than
  - JL/JNGE jump if less than, or jump if not greater than or equal
  - JLE/JNG jump if less than or equal, or jump if not greater than
Flow Control Instructions - Cont.

- **Unsigned jumps**
  - JA/JNBE  jump if above, or jump if not below or equal
  - JAE/JNB  jump if above or equal, or jump if not below
  - JB/JNAE  jump if below, or jump if not above or equal
  - JBE/JNA  jump if below or equal, or jump if not above

- **Single-Flag jumps**
  - JE/JZ    jump if equal, or jump if equal to zero
  - JNE/JNZ  jump if not equal, or jump if not equal to zero
  - JC       jump of carry
  - JNC      jump if no carry
  - JO       jump if overflow
  - JNO      jump if no overflow

- **Jump based on CX**
  - JCXZ

- **Loop Instructions**
  - Loop
  - Loopnz/Loopne
  - Loopz/Loope

- **All jump instructions have no effect on the flags.**
Branching Structures: IF-Then

- **Example:**
  If AX < 0 Then
  Replace AX by –AX
  ENDIF

  ; if AX < 0
  CMP AX, 0
  JNL END_IF
  ; then
  NEG AX
  END_IF:

IF-Then-Else

- **Example:**
  If AL <= BL Then
  Display character in AL
  Else
  Display character in BL
  ENDIF

  MOV AH, 2
  ; if AL<=BL
  CMP AL, BL
  JNBE ELSE_
  ; then
  MOV DL, AL
  JMP DISPLAY
  ELSE_
  MOV DL, BL
  DISPLAY:
  INT 21H
  END_IF:
CASE

- Example:
  CASE AX
  <0: put –1 in BX
  =0: put 0 in BX
  >0: put 1 in BX
  END_CASE

  ; case AX
  CMP AX, 0
  JL NEGATIVE
  JE ZERO
  JG POSITIVE
  NEGATIVE: MOV BX, -1
  JMP END_CASE
  ZERO: MOV BX, 0
  JMP END_CASE
  POSITIVE: MOV BX, 1
  END_CASE:

CASE – Cont.

- Example:
  CASE AL
  1,3: display ‘o’
  2,4: display ‘e’
  END_CASE
  ; case AL
  CMP AL, 1
  JE ODD
  CMP AL, 3
  JE ODD
  CMP AL, 2
  JE EVEN
  CMP AL, 4
  JE EVEN
  JMP END_CASE
  ODD: MOV DL, ‘o’
  JMP DISPLAY
  EVEN: MOV DL, ‘e’
  DISPLAY: MOV AH, 2
  INT 21H
  END_CASE:
Branches with Compound Conditions

Example:
If ('A' <= character) and (character <= 'Z') Then
  Display character
END_IF

; read a character
MOV AH, 1
INT 21H
; If ('A' <= character) and (character <= 'Z') Then
CMP AL, 'A'
JNGE END_IF
CMP AL, 'Z'
JNLE END_IF
; display character
MOV DL, AL
MOV AH, 2
INT 21H
END_IF:

Branches with Compound Conditions

Example:
If (character='y') OR (character <= 'Y') Then
  Display character
Else terminate program
END_IF

; read a character
MOV AH, 1
INT 21H
; If (character='y') OR (character = 'Y') Then
CMP AL, 'y'
JE Then
CMP AL, 'Y'
JE Then
JMP ELSE_
Then:
MOV AH, 2
MOV DL, AL
INT 21H
JMP END_IF
ELSE:
MOV AH, 4CH
INT 21H
END_IF:
Loop Instructions

- **Loop Next**
  - Dec Cx
  - If Cx<>0   JMP Next

- **Loopz/loope Next**
  - Dec Cx
  - If (Cx<>0) AND (ZF=1)   JMP Next

- **Loopnz/loopne Next**
  - Dec Cx
  - If (Cx<>0) AND (ZF=0)   JMP Next

FOR LOOP

- **Example:**
  For 80 times DO
  Display ‘*’
  END_IF
    MOV CX, 80
    MOV AH, 2
    MOV DL, ‘*’
Next:     INT 21H
          Loop Next
While Loop

- Example:
  Initialize count to 0
  Read a character
  While character <> Carriage Return DO
    Count = Count + 1
    Read a character
  END_While

    MOV DX, 0
    MOV AH, 1
    INT 21H
    While_: CMP AL, 0DH
             JE End_While
             INC DX
             INT 21H
             JMP While_
    End_While:

Repeat Loop

- Example:
  Repeat
    Read a character
  Until character is blank

    MOV AH, 1
    Repeat:
    INT 21H
    ; until
    CMP AL, ' ' 
    JNE Repeat
Application of Loope

- Example: Search for a number in a Table

Table DB 1,2,3,4,5,6,7,8,9

XOR SI, SI
MOV CX, 9
Next: INC SI
CMP Table[SI-1], 7
Loopne Next

The Stack

- One dimensional data structure
  - Items added and removed from one end
  - Last-in first-out

- Instructions
  - PUSH
  - POP
  - PUSHF
  - POPF

- PUSH & POP have one operand
  - 16-bit register or memory word
  - Byte operands are not allowed
  - Constant operands are not allowed
Stack Instructions

- **SP points at the the top of the stack**
- **.STACK 100H**
  - SP is initialized to 100H
- **PUSH operand**
  - SP ← SP - 2
  - [SP+1:SP] ← operand
- **POP operand**
  - Operand ← [SP+1:SP]
  - SP ← SP + 2
- **PUSHF**
  - SP ← SP - 2
  - [SP+1:SP] ← flags register
- **POPF**
  - Flags register ← [SP+1:SP]
  - SP ← SP + 2

Reversing a String

- **String DB “COE-205”**

  ```assembly
  MOV CX, 7 ; CX contains length of string
  XOR BX, BX
  Next: MOV AL, String[BX]
  PUSH AX
  INC BX
  LOOP Next
  MOV CX, 7
  XOR BX, BX
  Next2: POP AX
  MOV String[BX], AL
  INC BX
  LOOP Next2
  ```


Procedures

- **Procedure Declaration**
  
  Name PROC type
  
  ;body of the procedure
  
  RET
  
  Name ENDP

- **Procedure type**
  
  - NEAR (statement that calls procedure in same segment with procedure)
  - FAR (statement that calls procedure in different segment)
  - Default type is near

- **Procedure Invocation**
  
  - CALL Name

---

Procedures – Cont.

- **Executing a CALL instruction causes**
  
  - Save return address on the stack
    
    - Near procedure: PUSH IP
    
    - Far procedure: PUSH CS; PUSH IP
  
  - IP gets the offset address of the first instruction of the procedure
  
  - CS gets new segment number if procedure is far

- **Executing a RET instruction causes**
  
  - Transfer control back to calling procedure
    
    - Near procedure: POP IP
    
    - Far procedure: POP IP; POP CS

- **RET n**
  
  - IP ← [SP+1:SP]
  
  - SP ← SP + 2 + n
Passing Parameters to Procedures

- By value using Registers
- By address using Registers
- Using the stack
  - Copy SP to BP
  - Access parameters from stack using BP register

Procedure - Example

- Read a number n from 1-9 and display an array of n x n stars
  - NL DB 10,13,"$

```assembly
MOV AH, 1 ; read a number
INT 21H
AND AX, 0FH ; convert number from ASCII
MOV CX, AX
MOV BX, AX
Next::     PUSH CX
          PUSH BX
          CALL Display
          POP CX
          MOV AH, 9
          LEA DX, NL
          INT 21H
          Loop Next

Display  Proc Near
          MOV BP, SP
          MOV CX, [BP+2]
          MOV AH, 2
          MOV DL, "$"
Next2::   INT 21H
          Loop Next2
          RET 2
          Display ENDP
```
IN/OUT Instructions

- **Direct: port number is 0-255**
  - IN AL, port ; AL ←[port]
  - IN AX, port ; AL ←[port] ; AH ←[port+1]
  - OUT port, AL ; [port] ←AL
  - OUT port, AX ; [port] ←AL; [port+1] ←AH

- **Indirect: port number is in DX**
  - IN AL, DX ; AL ←[DX]
  - IN AX, DX ; AL ←[DX] ; AH ←[DX+1]
  - OUT DX, AL ; [DX] ←AL
  - OUT DX, AX ; [DX] ←AL; [DX+1] ←AH

String Instructions

- **Five categories**
  - MOVS, MOVSB, MOVSW
  - CMPS, CMPSB, CMPSW
  - SCAS, SCASB, SCASW
  - LODS, LODSB, LODSW
  - STOS, STOSB, STOSW

- **Source is always in DS:[SI]**
- **Destination is always in ES:[DI]**
- **If DF=0, SI and DI are incremented**
- **If DF=1, SI and DI are decremented**
- **To clear direction flag: CLD**
- **To set direction flag: STD**
String Instructions – Cont.

- **MOVSB**
  - ES:[DI] ← DS:[SI]
  - DI ← DI+1; SI ← SI+1 (if DF=0)
  - DI ← DI-1; SI ← SI-1 (if DF=1)

- **MOVSW**
  - ES:[DI+1:DI] ← DS:[SI+1:SI]
  - DI ← DI+2; SI ← SI+2 (if DF=0)
  - DI ← DI-2; SI ← SI-2 (if DF=1)

- **MOVSD destination, source**
  - Replaced by either MOVSB or MOVSW depending on operands size

String Instructions – Cont.

- **CMPSB**
  - DS:[SI] - ES:[DI]
  - DI ← DI+1; SI ← SI+1 (if DF=0)
  - DI ← DI-1; SI ← SI-1 (if DF=1)

- **CMPSW**
  - DS:[SI+1:SI] - ES:[DI+1:DI]
  - DI ← DI+2; SI ← SI+2 (if DF=0)
  - DI ← DI-2; SI ← SI-2 (if DF=1)

- **CMPSD destination, source**
  - Replaced by either CMPSB or CMPSW depending on operands size
String Instructions – Cont.

- **SCASB**
  - AL - ES:[DI]
  - DI ← DI+1; (if DF=0)
  - DI ← DI-1 (if DF=1)

- **SCASW**
  - AX - ES:[DI+1:DI]
  - DI ← DI+2; (if DF=0)
  - DI ← DI-2; (if DF=1)

- **SCAS destination**
  - Replaced by either SCASB or SCASW depending on operands size

String Instructions – Cont.

- **LODSB**
  - AL ← DS:[SI]
  - SI ← SI+1; (if DF=0)
  - SI ← SI-1 (if DF=1)

- **LODSW**
  - AX ← DS:[SI+1:SI]
  - SI ← SI+2; (if DF=0)
  - SI ← SI-2; (if DF=1)

- **LODS destination**
  - Replaced by either LODSB or LODSW depending on operands size
String Instructions – Cont.

■ STOSB
  • ES:[DI] ← AL
  • DI ← DI+1; (if DF=0)
  • DI ← DI-1   (if DF=1)

■ STOSW
  • ES:[DI+1:DI] ← AX
  • DI ← DI+2; (if DF=0)
  • DI ← DI-2   (if DF=1)

■ STOS destination
  • Replaced by either STOSB or STOSW depending on operands size

Copying a String to another

.DATA
  String1 DB “Hello”
  String2 DB 5 dup(?)

.CODE
  MOV AX, @DATA
  MOV DS, AX
  MOV ES, AX
  CLD
  MOV CX, 5
  LEA SI, String1
  LEA DI, String2
  REP MOVSB
Copying a String to another in Reverse Order

.DATA
String1 DB “Hello”
String2 DB 5 dup(?)

.CODE
    MOV AX, @DATA
    MOV DS, AX
    MOV ES, AX
    STD
    MOV CX, 5
    LEA SI, String1+4
    LEA DI, String2
    Next:   MOVSB
    ADD DI, 2
    LOOP Next