BACS1024 INTRODUCTION TO COMPUTER SYSTEMS

Chapter 6: Assembly Language Fundamental – Part I

0. Overview

1. Fundamental of Assembly Language & Program Structure
2. Data Definition
3. Data Transfer, Arithmetic & Addressing
4. Multiple Initializer
5. Data-related Operator
6. Indirect Addressing

1. Fundamental of Assembly Language & Program Structure

1. Fundamental of Assembly Language & Program Structure

- **Basic elements of Assembly Language**
  1. Constants
  2. Comments
  3. Reserved words
  4. Identifiers
  5. Statements & directives

1. Fundamental of Assembly Language & Program Structure

- **Basic elements of Assembly Language**

  1. **Constants**

     - ❑ **Numeric constants** can be written in decimal, hexadecimal, octal or binary using radix suffix

Radix suffices	Numbering System
d	Decimal (default)
h	Hexadecimal
q or o	Octal
b	Binary

1. Fundamental of Assembly Language & Program Structure

- **Basic elements of Assembly Language**
  1. **Constants**
     - ❑ **Character constants** can be written with enclosed single / double quotation mark.
     - ❑ The assembler converts it to equivalent value in binary code ASCII.
     - ❑ E.g.: `'A' , "d"`
     - ❑ A string of characters enclosed in single / double quotation marks.
     - ❑ E.g.: `"BACS", 'BACS1024'`

1. Fundamental of Assembly Language & Program Structure

- Basic elements of Assembly Language

  1. **Constants**

     ❑ **Character constants vs Numeric constant**

     ❖ Character constant

     E.g.: `‘24’`, will generate two ASCII character, represented by **`3234H`**

     ❖ Numeric constant

     E.g.: `24`, will generate a binary number, represented as **`18H`**

1. Fundamental of Assembly Language & Program Structure

- **Basic elements of Assembly Language**

2. Comments

- ❑ Used to improve program clarity
- ❑ **Single line**
  - ❖ Begin with semicolon
- ❑ **Multi-lines**
  - ❖ Using delimiter

E.g.:(Single line comment)

```
MOV AL,12    ;Move 12H to AL register
```

E.g.:(Multi-lines comments)

```
Comment    + Move the value 12H
             into AL register
           + End of comment
```

1. Fundamental of Assembly Language & Program Structure

- **Basic elements of Assembly Language**

3. Reserved Words

- ❑ Have a special meaning & cannot be used for other than their specified purpose.

Types	Examples
Instructions	`MOV, ADD, MUL`
Directives	`END, SEGMENT`
Operators	`FAR, SIZE`
Predefined symbols	`@DATA, @MODEL`

1. Fundamental of Assembly Language & Program Structure

- **Basic elements of Assembly Language**

4. Identifier

❑ A programmer-chosen name

❑ To represent an item (variables, constants, procedures or labels)

❑ Acts as a place marker for instruction & data

Data label	Code label
Refers to the address of a data item	Refers to the address of an instruction, procedure or segment
E.g.: `PROD_ID   DB  0`	E.g.: `MAIN  PROC` `L1:   ADD  AL,2H`

1. Fundamental of Assembly Language & Program Structure

- **Basic elements of Assembly Language**

  4. Identifier

  ❑ Naming conventions:

  ❖ First character must be a letter, underscore or special character

  ❖ First character cannot be a dot '.'

  ❖ First character cannot be a digit

  ❖ Uppercase and lowercase letters are treated the same

  ❑ E.g.: `TOTAL` `var1` `@myFile`

  `QTY250` *`$50`* *`_12345`*

1. Fundamental of Assembly Language & Program Structure

- **Basic elements of Assembly Language**

  5. Statement & directive

  - ❑ An assembly language program consists of a set of statements.
  - ❑ There are 2 types of statements
    - ❖ Instructions: a statement that becomes executable when a program Assembled
    - ❖ Directives: A command embedded in the source code that will acted upon by the assembler

1. Fundamental of Assembly Language & Program Structure

- **Basic elements of Assembly Language**

  5. **Statement & Directive**

     - General format of statement

       [Identifier] Operation [Operand(s)] [;Comment]

	Identifier	Operation	Operand	Code label
Instruction	`L10:`	`MOV`	`AX,20H`	; **MOV** instruction
Directive	`COUNT`	`DB`	`1`	; **DB** directive

1. Fundamental of Assembly Language & Program Structure

- **Program Structure**

```
.MODEL SMALL
.STACK 64
.DATA
.CODE
MAIN PROC
   MOV AX,@DATA
   MOV DS,AX
       :
   MOV AX,4C00H
   INT 21H
MAIN ENDP
   END MAIN
```

1. Fundamental of Assembly Language & Program Structure

- **Program Structure**
  - ❑ **Simplified segment directive**
    - ❖ An assembly program in .EXE format consists of one or more segments
      - ✓ Stack segment : defines stack storage
      - ✓ Data segment : defines data items
      - ✓ Code segment : provides for executable codes
    - ❖ You have to initialize the memory model before defining any segment

1. Fundamental of Assembly Language & Program Structure

- **Program Structure**
  - **Memory models**
    - To tell the assembler how to:
      - Use segment
      - Provide enough space for object code
      - Ensure optimum execution speed
    - Format

      `.MODEL memory-model`
    - The *memory-model* can be **`TINY, SMALL, LARGE, HUGE`** or **`FLAT`**

Model	No. of Code Segment	No. of Data Segment
SMALL	1 <=64k	1 <=64k
MEDIUM	Any number, any size	1 <=64k
COMPACT	1 <= 64K	Any number, any size
LARGE	Any number, any size	Any number, any size
HUGE	Any number, any size	Any number, any size

2. Data Definition

2. Data Definition

- The assembler provides a set of directives that permits definitions of items by various types and lengths.
- Format: `[name] directive  initializer`
- E.g. `:  value  DB          25`

(a) **Directives**

❑ DB = Define Byte = Define 1 or more 1 byte constants

❑ DW = Define Word = Define 1 or more 2 bytes constants

❑ DD = Define Doubleword = Define 1 or more 4 bytes constants

❑ DQ = Define Quadword = Define 1 or more 8 bytes constants

2. Data Definition

(b) **Initializer**

- At least one initializer is required in a data definition, even if it is zero.
- E.g.:

```
value2 DB 0

value3 DB ?
```

- For **integer data type**, initializer is an integer constant / expression matching the size of the variable’s type
- For **string data type**, enclose a sequence of characters in quotation marks.
- E.g.:

```
msg1 DB “Hello World$”
msg2 DB “Welcome to “
        DB “Computer Systems”, “$”
```

3. Data Transfer, Arithmetic & Addressing

3. Data Transfer, Arithmetic & Addressing

- **MOV instruction**
  - Copies data from one location to another location
  - The operands must agree in size
  - Format:

    ```
    MOV (register/memory),(register/memory/immediate)
    ```

    Destination ← Source

  - E.g.:

    ```
    MOV AX, 1234
    MOV VAR1,1234
    MOV VAR1, AX
    MOV AX, VAR2
    ```

3. Data Transfer, Arithmetic & Addressing

- `MOVZX` **instruction**
  - Copies data contents of a source operand into a destination operand and zero-extends the value to 16 bit or 32 bits
  - Only used with unsigned integer
  - Format:

    ```
    MOVZX (register16),(register8/memory8)
    ```

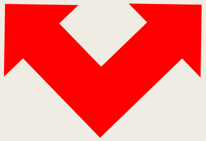
    Destination ← Source
  - E.g.:

    ```
    .DATA
          VAR1 DB 2AH
    .CODE
          MOVZX AX,VAR1    ; AX = 002AH
    ```

3. Data Transfer, Arithmetic & Addressing

- **`XCHG` instruction**
  - Swap the contents of two memory locations
  - Format:

```
XCHG (register/memory),(memory/register)
```



Swap

  - E.g.:

```
XCHG AL,BL
XCHG AL,VAR1
XCHG VAR2,BL
```

3. Data Transfer, Arithmetic & Addressing

- **ADD instruction**
  - ❑ Add the source operand to the destination operand of the same size.
  - ❑ Source and destination cannot be both memory
  - ❑ Format:

```
ADD(register/memory),(memory/register/immediate)
    Destination          Source (remain unchanged)
```

  - ❑ E.g.:

```
ADD AL,BL
ADD AL,25H
ADD VAR2,BL
```

3. Data Transfer, Arithmetic & Addressing

- `SUB` **instruction**
  - ❑ Subtracts a source operand from the destination operand of the same size
  - ❑ Source and destination cannot both be memory variables
  - ❑ The result may affect the flag register status
  - ❑ Format:

```
SUB(register/memory),(memory/register/immediate)
        Destination        Source (remain unchanged)
```

  - ❑ E.g.:

```
SUB AL,BL
SUB AL,25H
SUB VAR2,BL
```

3. Data Transfer, Arithmetic & Addressing

- **`INC / DEC` instruction**
  - ❑ Increments and decrements its operand by 1
  - ❑ All flag register status (except carry) are affected.
  - ❑ Format:

    ```
    INC memory/register
    DEC memory/register
    ```

  - ❑ E.g.:

    ```
    INC AL
    DEC BX
    ```

3. Data Transfer, Arithmetic & Addressing

- `MUL` **instruction**
  - ❑ Perform multiplication on unsigned data
  - ❑ Affect the carry and overflow flag
  - ❑ Format:

```
MUL memory/register
```

  - ❑ E.g.:

```
MOV AL,2
MUL AL
```

Instruction	Multiplier	Multiplicand	Product
MUL CL	Byte	AL	AX
MUL BX	Word	AX	DX:AX

3. Data Transfer, Arithmetic & Addressing

- `DIV` **instruction**
  - ❑ Perform division on unsigned data
  - ❑ Format:

```
DIV memory/register
```

  - ❑ E.g.:

```
MOV AX, 83
DIV BL       ; BL = 2
```

  - ❑ The size of the register determines the type of operation

Instruction	Divisor	Dividend	Quotient	remainder
DIV CL	Byte	AL	AL	AH
DIV BX	Word	DX:AX	AX	DX

3. Data Transfer, Arithmetic & Addressing

- `CBW` **instruction**
  - ❑ Stand for Convert Byte to Word
  - ❑ Convert the signed byte in **AL** to a signed word in **AX** by extending the **MSB** of **AL** into **AH**.
  - ❑ There is no effect of **CBW** on the flags
  - ❑ **CBW** has no operand
  - ❑ **CBW** is restricted to the use of **AX** register
  - ❑ Format:

```
CBW
```

  - ❑ E.g.:

```
MOV AL, 25H
CBW                    ; AX=0025H
```

3. Data Transfer, Arithmetic & Addressing

- **`SHL` / `SHR` instruction**
  - SHL (shift left): Shift the bits to the left
  - SHR (Shift right): Shift the bits to the right
  - Format:

    ```
    SHL destination, 1/CL

    SHR destination, 1/CL
    ```

  - E.g.:

    ```
    MOV CL,3          ; CL = 3
    MOV AL, 25H       ; AL = ~~0~~010 0101B (msb is shifted)
    SHL AL, 1         ; AL = 0100 101~~0~~B
    SHR AL, CL        ; AL = 0000 1001B (3 bits are added)
    ```

4. Multiple Initializer

4. Multiple Initializer

- If a definition has multiple initializer, the label is the offset for the first item
- E.g.:

```
alist           DB     10,  20,    30,   40,   50
Offset value:   0000  0001  0002   0003  0004
               alist+0      alist+2
```

- **Direct-offset operands**
  - ❑ Add a displacement to the name of a variable, creating a direct-offset operand
  - ❑ Enable access to memory locations that may not have explicit labels
  - ❑ E.g.:

```
MOV AL, alist               ; AL = 10
MOV AL, [ALIST + 1]         ; AL = 20
```

  - ❑ Different initializer can be used in a data item

4. Multiple Initializer

- **Defining 16 bit data**

`Value1 DW 65535`	`; unsigned word`
`Value2 DW -32768`	`; signed word`

`myList DW 1, 2, 3, 4, 5`

	1	2	3	4	5
offset value:	0000	0002	0004	0006	0008

- **Defining 32 bit data**

`Value3 DD 12345678H`	`; unsigned double-word`
`Value4 DD -21474836648`	`; signed double-word`

`myList DD 1, 2, 3, 4, 5`

	1	2	3	4	5
offset value:	0000	0004	0008	000C	0010

5. Data-Related Operator(s)

5. Data-related Operator

- **DUP Operator**
  - ❑ Allocates storage for multiple data items, using a constant expression as a counter
  - ❑ It is useful when allocating space for a string or array, and can be used with initialized or uninitialized data
  - ❑ E.g.:

```
Array1 DB 20 DUP(0)    ;20 bytes with zero
Array2 DB 20 DUP(?)    ;20 bytes uninitialized
Array3 DB 2 DUP(“Good”);”GoodGood”
Array4 DB 5,4,3 DUP (2, 3 DUP (0), 1)
```

5. Data-related Operator

- **OFFSET Operator**
  - ❑ The offset operator returns the number of bytes between the label and the beginning of its segment
  - ❑ It produce a 16-bit immediate value. Therefore, the destination must be 16-bit operand
  - ❑ E.g.:

```
.DATA
   blist     DB 10H, 20H, 30H, 40H
    wlist DW 1000H, 2000H, 3000H, 4000H
.CODE
   MOV DI, OFFSET blist + 1     ; DI = 0001H
   MOV BX, OFFSET wlist + 2     ; BX = 0006H
```

6. Indirect Addressing

6. Indirect Addressing

- An indirect operand is a register containing the offset for data in the memory location.
- If the register is used as an indirect operand, it may only be **SI**, **DI**, **BX** or **BP**. Avoid **BP** unless you are using it to index into the stack
- E.g.:

```
.DATA
      num         DB    10H
      myStr       DB    “ABCDE”
.CODE
      MOV SI, OFFSET num        ;SI = 0000H
      MOV AL, [SI]              ;AL = 10H
      MOV BX, OFFSET myStr      ;BX = 0001H
      ADD BX,3                  ;BX = 0004
      MOV DL, [BX]              ;DL = ‘D’
```

6. Indirect Addressing

- LEA Instruction
  - ❑ Stands for “Load Effective Address”
  - ❑ Initializes a register within offset address
  - ❑ E.g.:

```
.DATA
      aList       DB    25 DUP (0)
      value       DB    ?
.CODE
      LEA BX, aList     ; equivalent to
                        ; MOV BX, OFFSET aList
      MOV AL, [BX]
```

Chapter Review

Chapter Review

1. Fundamental of Assembly Language & Program Structure

- ❑ Constants
- ❑ Comments
- ❑ Reserved words
- ❑ Identifiers
- ❑ Statements & directives
- ❑ Program Structure

2. Data Definition

- ❑ Directive: `DB, DW, DD, DW`
- ❑ Initializer: `0, ? , others`

3. Data Transfer, Arithmetic & Addressing

- ❑ `MOV, MOVZX, XCHG`
- ❑ `ADD, SUB, MUL, DIV`
- ❑ `INC, DEC`
- ❑ `SHL, SHR`
- ❑ `CBW`

4. Multiple Initializer

5. Data-related Operator

- ❑ `DUP`
- ❑ `OFFSET`

6. Indirect Addressing

- ❑ `LEA`