Experiment 4

4 Segmentation and Addressing Modes

Introduction

Addressing modes are a means, provided by assembly to the programmer, to access memory through different ways. This helps the programmer write at his ease his programs. The different assembly addressing modes are introduced in this experiment.

Objectives:

- Segmentation
- Addressing Modes
- Indexing
- Data Manipulation and Array Manipulation
- Pentium Addressing modes

4.1 Memory Segmentation

In the Intel family of microprocessors, memory is divided into segments. Since registers are 16-bit wide, each segment has its address space limited to 64 Kbytes. The 16-bit address used by the program is actually an offset from a segment base address, saved in the segment register. Pentium processors however, can operate in one of two modes; Real Mode and Protected Mode. In the real mode, the Pentium processor behaves exactly like a fast 8086 processor. In the protected mode, the Pentium processor supports both segmentation and pagination. Pagination is useful in implementing virtual memory, concept very used in operating system. However, pagination is transparent to the programmer and therefore is not covered at this level.

Segment registers are independent and segments can be contiguous, disjoint, partially overlapped or fully overlapped.

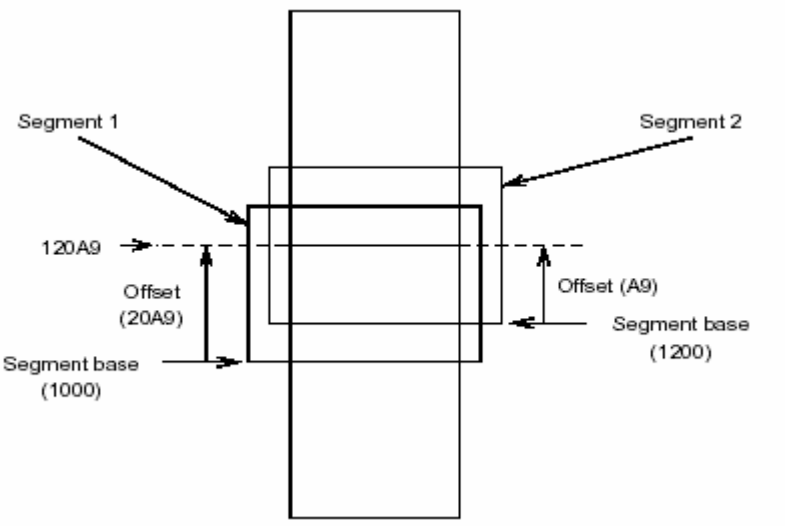

Figure 4.1: Memory Segments

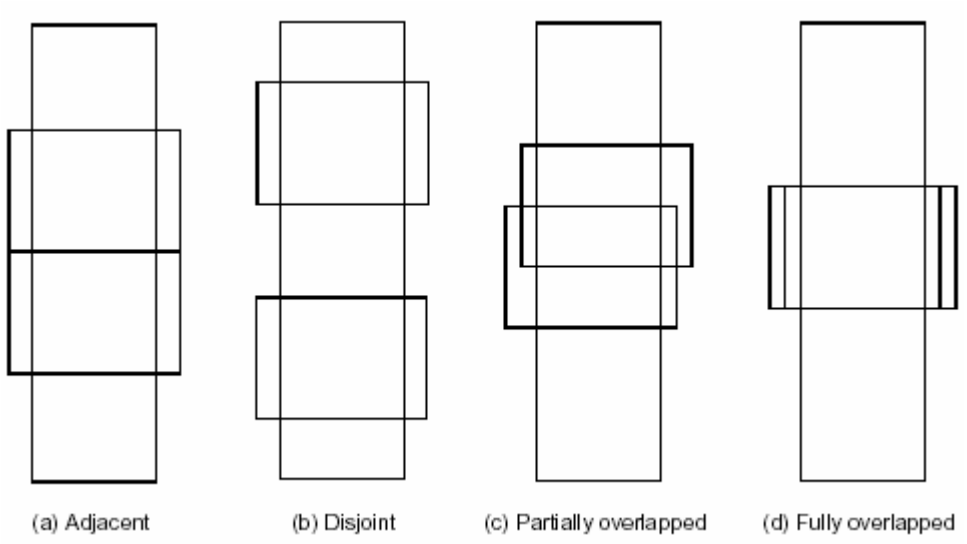

Figure 4.2: Various ways of placing segments in the memory

4.2 Real Mode Memory Architecture

The segment register is offset by four bits and added to the offset to give a 20-bit address, known as the ***physical address***.

Physical Address = Segment Register*10H + Offset

Application programs use the ***logical address*** which consists of two components: a segment base and an offset or effective address, such as **BASE ADDRESS:OFFSET**. The base address is usually in a segment register, which is a pointer to the base of the segment. The offset is also known as the ***effective address***. Code, data and stack are each put in one segment. Each segment has its own base address saved in a segment register. The segment registers are:

- CS: code segment,
- SS: stack segment,
- DS: data segment
- ES, FS, GS: extra data segments.

4.3 Addressing Modes

Table 4.1 summarizes all addressing modes used by the 8086 processor and the real mode of the Pentium processor. The address is computed only when memory is accessed, no addressing is needed when the processor accesses an immediate operand making part of the instruction, or its own internal registers. Internal registers are accessed using a different mechanism that will be covered in the organization part of the COE 205 course.

Addressing Mode	Example	Source operand	
		Assuming: DS = 1000H, SS = 2000H, BX = 0200H, SI = 0300H, BP = 0400H	
		Address Generation	Phys. Address
Immediate	MOV AX, 0F7H	-	-
Register	MOV AX, BX	-	-
Direct	MOV AX, [1234H]	DS x 10H + 1234H	11234H
Register-Indirect	MOV AX, [BX]	DS x 10H + 0200H	10200H
Register-Indirect	MOV AX, [BP]	SS x 10H + 0400H	20400H
Based	MOVAX, [BX+ 06]	DS x 10H + 0200H + 0006H	10206H
Indexed	MOVAX, [SI + 06]	DS x 10H + 0300H + 0006H	10306H
Based-Indexed	MOV AX, [BX+SI+06]	DS x 10H + 0200H + 0300H + 0006H	10506H

Table 4.1: Addressing Modes

4.3.1 Which Segment Register

Any of the registers **BX**, **SI** or **DI** can be used as an offset into the **data segment. BP** is used as an offset into the ***stack* segment**. This is the default situation unless overriding is used, as shown in the next example.

```
MOV AL, Byte Ptr [BP]        ; PA = SS*10H + BP, SS is assumed by default
MOV AL, Byte Ptr [DS:BP]     ; PA = DS*10H + BP using overriding
```

When the processor is to read a program instruction, the CS register is used to provide the segment part of the logical address of the instruction to be fetched. The offset part is supplied by IP (or EIP) register. Thus **CS:IP** points to the next instruction to be fetched from the code segment. The physical address is computed as follows:

PA = CS*10H + IP

4.3.2 Array Indexing

An array is a contiguous list of data elements in memory. All elements are of the same type and use the same number of bytes of memory. Because of these properties, arrays allow efficient access of data by its position, or index. The address of any element can be computed by knowing: (i) the address of the first element of the array, (ii) the number of bytes in each element and (iii) the index of the element. The index of the first element of the array is zero.

In 16-bit addressing mode elements of an array are usually accessed through one of the following addressing modes:

- Based addressing mode.
- Indexed addressing mode.

- Based-Indexed addressing mode.

Arrays, depending on the programmer view, may be seen as 1-dimensional or 2-diomensional structures. Based and Indeed modes are often used to access 1-dimensional arrays. The Based-Indexed addressing mode is often used to access two-dimensional arrays.

Base		**Index**		
BX or **BP**	+	**SI** or **DI**	+	*Displacement*

Table 4.2: Based Indexed Addressing modes

1-Dimensional Arrays

Example 1

```
. DATA
        NUM DB 20, 4, 32, 50, 7, 15, 80, 12, 6, 125
. CODE
        MOV BH, NUM[0]          ; BH is assigned the value 20
        MOV AL, NUM[3]          ; AL is assigned the value 50
        MOV BX, 5
        MOV AH, NUM[BX]         ; AH is assigned the value 15

        MOV SI, 7
        MOV CL, NUM[SI  + 2]    ; CL is assigned the value 125
        MOV DL, NUM[BX - 3]     ; DL is assigned the value 32
        MOV BX, 2
        MOV AL, NUM[SI + BX]    ; AL is assigned the value 125
```

Example 2

```
        Array1   DB   0, 1, 2, 3, 4, 5, 6, 7, 8, 9
        Array2   DB   10 DUP(0)
        ; Based addressing mode
        MOV BX, OFFSET Array1   ; Address Array1
        MOV AL, [BX+4]          ; Load 5th element of Array1 into AL

        ; Indexed addressing mode
        MOV DI, OFFSET Array2   ; Address Array2
        MOV [DI+6], AL          ; Save AL into 7th element of Array2
        MOV SI, 3               ;
        MOV Array2[SI], AL      ; Save AL into 4th element of Array2
```

2- Dimensional Arrays

Example 3

```
Array3   DB   11 12, 13, 21, 22, 23, 31, 32, 33      ; Array3 is a 3x3 array
RowSize EQU 3

; Based-Indexed addressing mode
MOV BX, OFFSET Array3   ; Address Array3
MOV SI, 1*RowSize       ; Beginning of 2nd row
MOV DI, 2*RowSize       ; Beginning of 3rd row
MOV AL, [BX+SI+1]       ; Load 2nd element of 2nd row into AL
MOV [BX+DI+2], AL       ; Save AL into 3rd element of 3rd row
```

Remark:

- That the R^{th} row has index (R-1)
- The n^{th} element has index (n-1)

4.4 Protected Mode Memory Architecture

The protected mode is the native mode of the Pentium processor. The processor supports a more sophisticated segmentation mechanism. Before dealing with Pentium addressing modes, the 32-bit Pentium registers are briefly introduced.

4.4.1 Pentium Registers

Each register has 32 bit, 16 bit and 8 bit names. In the real mode, the 16 bit names are used for addressing, whereas in the protected addressing mode, the 32 bit names are used.

- The primary accumulator register is called **EAX**. Secondary accumulator registers are: EBX, ECX and EDX.
- **EBX** is often used to hold the starting address of an array.
- **ECX** is often used as a counter or index for a loop.
- **EDX** is a general purpose register.
- The **EBP** register is a stack pointer. It is used to facilitate access to the stack.
- **ESI** and **EDI** are general purpose registers. If a variable is to have register storage address, it is often stored in either ESI or EDI. String handling instructions use ESI and EDI as pointers to source and destination addresses.
- The **ESP** register or stack pointer, points to the top of the stack.
- The **EFLAGS** register, or status register. Individual bits in this register are affected or checked by several instructions.
- The **EIP** register holds the instruction pointer or program counter (PC), which points to the next instruction of the currently running program.

4.4.2 32-bit Addressing Modes

In 32-bit addressing bit, a scale factor is multiplied by the index register to be added to the base register each time an instruction using a memory operand is executed. Table 4.3 shows how the elements of a memory operand appear.

To load the 9th double word-sized entry in the table *DwordArray* into EAX:

```
MOV EBX, 8
MOV EAX, DwordArray [EBX*4]
MOV EBX, 7
MOV ECX, 4
MOV EAX, DwordArray [ECX + EBX*4]
```

As seen above, index scaling can be extremely useful for accessing elements in *word*, *double-word*, and *quad-word* arrays.

By default, if the ***base register*** is either **EBP** or **ESP** then **SS** is the segment address otherwise **DS** is used.

Base		**Index**		**Scale**		
EAX		**EAX**				
EBX		**EBX**		**1**		
ECX		**ECX**		or		
EDX		**EDX**		**2**		
ESI	+	**ESI**	*	or	+	***Displacement***
EDI		**EDI**		**4**		
EBP		**EBP**		or		
ESP				**8**		

Table 4.3: 32-bit addressing mode

4.5 Some Basic Assembly Instructions

4.5.1 Procedures

A procedure is the equivalent of a method in Java. Procedures avoid rewriting pieces of code.

A procedure is declared using:

```
ProcName    PROC NEAR        ; Near is by default, therefore optional
        ….
        ….
        RET
ProcName    ENDP
```

A Procedure is invoked using the instruction CALL:

```
CALL ProcName
```

4.6 *Lab Work*

Part I

Program 1:

- Write a program that reads two 2-digit numbers, saves them in 16 bit registers then displays their sum.

- Can you write your program so that it can deal with 4-digit numbers. What do you think is the problem?

- Note that numbers are dealt with in unpacked BCD format, where each digit occupies one byte. Packed BCD numbers are dealt with in the next experiment.

Program 2:

Repeat the same program as above, but with 4-digit numbers, using array variables instead of registers. Store each number in an array. Use the following procedure to read a single digit number.

```
READC PROC
        MOV AH, 01
        INT 21H
        SUB AL, 30H
        RET
READC ENDP
```

Write a similar procedure to display a single digit number on the screen. Use the procedure in your program to display the sum of the two numbers.

Part II

Program 3:

Write a program that reads two 2x3 arrays of single digit numbers, adds the two arrays and displays the array sum in matrix format.

Program 4:

Given the 2 initial elements of a Fibonacci series, write a program that generates the first 20 elements of the series. Knowing that, each new element is equal to the sum of the two previous ones.

```
.DATA
        FIB DW 1, 1, 18 DUP(?)
```