Experiment 5

5 Arithmetic Instructions

Introduction

In this experiment, the arithmetic instructions of the 8086 family of processors are introduced. Conversion of numbers from one radix to another is also discussed.

Objectives

- Addition ADD ADC INC
- Subtraction: SUB SBB DEC
- Two's and One's Complement: NOT, NEG
- BCD numbers manipulation: DAA DAS
- Multiplication: MUL, IMUL
- Division: DIV, IDIV
- Sign Extension: CBW, CWD, XCHG, MOVZX, MOVSX
- Number reading/displaying, Base conversion
- More on array manipulation XLAT, XLATB, XADD

5.1 Arithmetic Instructions

Arithmetic instructions include the four basic operations: addition, subtraction, multiplication, division, then increment and decrement, comparison and negation.

Table 5. 1 to 5.4 summarize most of the arithmetic instructions used with the x86 microprocessor family. The effect of each instruction is briefly shown through an example and the affected flags. The “M” means that the corresponding flag may change as a result of executing the instruction. The “-“ means that the corresponding flag is not affected by the instruction, whereas the “*” means that the flag is undefined after executing the instruction.

5.1.1 Addition and Subtractions

The ADD, INC, SUB and DEC instructions operate on 8 and 16-bit values on the 8086 processor, and on 8-, 16-, and 32-bit values on the Pentium processor. Combined with the ADC and SBB instructions operations can be performed on 32-bit values for the 8086 and 64-bit values for the Pentium processor. These instructions require that, both operands be the same size and only one operand can be a memory operand.

Type	Instruction	Example	Effect	Flags Affected					
				OF	SF	ZF	AF	PF	CF
Addition	ADD	ADD AX, 7BH	AX ← AX + 7B	M	M	M	M	M	M
	ADC	ADC AX, 7BH	AX ← AX + 7B +CF	M	M	M	M	M	M
	INC	INC [BX]	[BX]←[BX]+1	M	M	M	M	M	-
	DAA	DAA		*	M	M	M	M	M
Subtraction	SUB	SUB CL, AH	CL ← CL – AH	M	M	M	M	M	M
	SBB	SBB CL, AH	CL ← CL – AH – CF	M	M	M	M	M	M
	DEC	DEC DAT	[DAT] ← [DAT] – 1	M	M	M	M	M	-
	DAS	DAS		*	M	M	M	M	M
	NEG	NEG CX	CX ← 0 – CX	M	M	M	M	M	M

Table 5. 1: Addition and Subtraction Instructions

Adding Integers

The next example shows how to use the ADD and ADC to perform 32-bit addition using the 16-bit 8086 processor registers.

Suppose you have to add two 64-bit numbers on a Pentium processor. Numbers are stored as follows:

```
; Number1:          EAX          EBX
; Number2:          ECX          EDX
; Let's perform: Number1 = Number1 + Number2
ADD EBX, EDX
ADC EAX, ECX
```

The second instruction takes care of the carry if any from adding the two low words of the two 64-bit numbers.

If numbers are given in decimal format, the BCD system should be used. DAA (Decimal Adjust after Addition) and DAS (Decimal Adjust after Subtraction) instructions are used to deal with BCD numbers. The DAA instruction, used immediately after normal addition instruction, allows addition of numbers represented in 8-bit packed BCD code. The sum in AL is adjusted to packed BCD format. The AL register is the source and the destination and hence no operand is required. The DAS instruction used after a subtraction operation has an effect similar to DAA.

Example:

```
 MOV AL, 23H        ; Number1: 23H
 MOV CL, 59H        ; Number2: 59H
; Let's perform: Number1 = Number1 + Number2
ADD AL, CL          ; AL = 7CH, result in Hex
DAA                 ; AL = 82H, result in BCD
```

5.1.2 Multiplication and Division

Type	Instruction	Example	Effect	Flags Affected					
				OF	SF	ZF	AF	PF	CF
Multiplication	MUL	MUL CL	AX ← AL * CL	M	*	*	*	*	M
		MUL CX	(DX,AX) ← AX * CX						
	IMUL	IMUL BYTE PTR X	AX ← AL * [X]	M	*	*	*	*	M
		IMUL WORD PTR X	(DX,AX) ← AX * [X]						
Division	DIV	DIV WORD PTR X	AX ← Q([DX,AX]/[X]) DX ←R([DX,AX]/[X])	*	*	*	*	*	*
	IDIV	IDIV BH	AL ← Q(AX/BH) AH ←R(AX/BH)	*	*	*	*	*	*

Table 5. 2: Multilply and Divide Instructions

Example

The following code multiplies 32 bit-numbers using Pentium registers, to get a 64 bit number value.

```
MOV EAX, NUMBER1
MOV EBX, NUMBER2
MUL EBX
; (EDX,EAX) =  NUMBER2 X NUMBER2
```

5.1.3 Divide Overflow and Sign Extension

However, overflow may occur if the result does not fit in the destination: such as dividing a 16 bit number by a small number so that the result is greater than an 8 bit register. Overflow occurs in one of two cases:

```
Byte form:  AH = divisor
Word form: DX = divisor
```

To avoid overflow, the dividend has to be prepared. CBW and CWD instructions are used to facilitate division of 8 and 16 bit signed numbers. Since division requires a double-width dividend, CBW converts an 8-bit signed number (in AL), to a word, where the most significant bit of AL register is duplicated into AH register. Similarly, CWD converts a 16-bit signed number to a 32-bit signed number (DX, AX); the most significant bit of AX register is duplicated into DX register.

Instruction	Effect	Flags Affected					
		OF	SF	ZF	AF	PF	CF
CBW	AH ← MSB(AL)	-	-	-	-	-	-
CWD	DX ← MSB(AX)	-	-	-	-	-	-
CWDE	High Word of EAX ← MSB(AX)	-	-	-	-	-	-

Table 5. 3: Sign Extension Instructions

To divide a word in AX by a word, AX must be converted to a double-word in DX:AX.

- If signed: CWD (called sign-extension)
- If unsigned: MOV DX, 0
- To divide a word in AL by a byte, AL must be converted to a word in AX.
- If signed: CBW
- If unsigned: MOV AH, 0

The following table shows the use of MOVSX and MOVZX to prepare the operand for extended multiplication or division to avoid overflow.

Instruction	Example	Effect	Flags Affected					
			OF	SF	ZF	AF	PF	CF
MOVSX	MOVSX AX, Byte Ptr X	AL ← X, AH ← MSB(X)	-	-	-	-	-	-
	MOVSX EAX, Byte Ptr X	AL ← X, EAX_{8-31} ← MSB(X)						
	MOVSX EAX, Word Ptr X	AL ← X, EAX_{16-31} ← MSB(X)						
MOVZX	MOVZX AX, Byte Ptr X	AL ← X, AH ← 0	-	-	-	-	-	-
	MOVZX EAX, Byte Ptr X	AL ← X, EAX_{8-31} ← 0						
	MOVZX EAX, Word Ptr X	AL ← X, EAX_{16-31} ← 0						

Table 5.4: Copy with Sign/Zero Extension Instructions

5.2 Byte Manipulations for Reading and Displaying Purposes

Assembly does not provide instructions to read decimal numbers. One character is read at a time. Decimal input and output is explained in the following.

5.3 Reading a Decimal Number

1 / To read a two digit decimal number and save it in one byte:

```
MOV AH, 01H
INT 21H
SUB AL, 30H
MOV CH, AL          ; Read high digit        e.g. 8
```

```
MOV AH, 01H
INT 21H
SUB AL, 30H
MOV CL, AL          ; Read low digit        e.g. 3

MOV AL, 10000B      ; Use MUL by 10000B to shift left by 4 bits
MUL CH              ; Shift AL 4 bits to the left
XOR AH, AH          ; Clear AH
OR AL, CL           ; Result in AL ← 83
```

To perform addition:

```
ADD AL, CL          ; assuming:  AL  contains the first BCD number
                    ; and CL  contains the second BCD number
DAA                 ; Decimal adjust
                    ; New result in AL in BCD format
MOV CL, AL          ; Number in CL register.
```

5.4 Displaying a Decimal Number

2 / To display a number in BCD format use the following:

```
; The number is in the CL register:
MOV AL, CL          ; Move CL to AL
MOV AH, 000         ; Clear AH
MOV BL, 10000B
DIV BL              ; Shift  AX 4 bits to the
ADD AL, 30H         ; Convert to character
; Now Display AL as the most significant digit first
MOV AL, CL          ; Read number again
AND AL, 0FH         ; Clear 4 high nibbles of AL
ADD AL, 30H         ; Convert to character
; Now Display AL as the least significant digit
```

5.4.1 Displaying a Number in Base r:

The basic idea behind displaying data in any number base is **division**. The number is divided by r, and the remainder of the division is saved as a significant digit of the result. The remainder is a number between zero and (r-1). Because of this, the resultant remainder will be a different number base than the input which is base 2. To convert from binary to any other base, use the following algorithm.

5.4.2 Quick Conversion between Bases

From base b to decimal

To convert from any base to decimal, Horner's algorithm is used. A number in base b can be considered as a polynomial p in the variable b. Let:

$$\mathbf{P(b) = a_n b^n + a_{n-1} b^{n-1} + \ldots + a_1 b^1 + a_0 b^0}$$

This would be written as a series of coefficient or digits. For example, the base 6 value 32501_6 represents $3*(6^4)+2*(6^3)+5*(6^2)+0*(6^1)+1*(6^0)$. Using nested multiplications this can be rewriten without needing to compute powers of the base ($\mathbf{b^i}$): ((((3*6) + 2)*6 + 5)*6 + 0)*6 + 1

$$\mathbf{P(b) = ((\ldots(((a_n b + a_{n-1}) b + \ldots + a_1) b + a_0}$$

Converting from decimal to other bases

In determining what a decimal value would look like when represented in another base (b), we need to figure out what digit in base b goes in each position of the representation. To do this we use the following algorithm.

Algorithm:

1. Divide the number to be converted by the desired radix (**r**).
2. Save the remainder as the most significant digit of the result.
3. Repeat steps 1 and 2 until the resulting quotient is zero.
4. Display the remainders as digits of the result.

Note that the first remainder is the least significant digit, while the last remainder is the most significant one.

Two's and One's Complement

NEG replaces the contents of its operand by the two's complement negation of the original value. This is accomplished by inverting all the bits and then adding one. NOT, similarly, performs one's complement (inverts all the bits).

Instruction	Example	Effect	Flags Affected					
			OF	SF	ZF	AF	PF	CF
NOT	NOT AX	AX ← 1's complement of (AX)	M	M	M	M	M	M
NEG	NEG BX	BX ← 2's complement of (BX)						

Table 5.5: One's and Two's Complement Instructions

Look-Up table Indexing

XLAT adds the value in AL, treated as an unsigned byte, to BX or EBX, and loads the byte from the resulting address (in the segment specified by DS) back into AL. The base register used is BX if the address size is 16 bits and EBX if it is 32 bits. If you need to use an address size not equal to the current BITS setting, you can use an explicit a16 or a32 prefix. The segment register used to load from [BX+AL] or [EBX+AL] can be overridden by using a segment register name as a prefix (for example, ES XLATB).

The effect of this instruction is shown in the following example:

```
MOV BX, OFFSET TABLE      ; BX ← Offset TABLE
MOV AL, INDEX             ; AL ← Index of the element in table
XLAT                      ; AL ← [DS:BX+AL]
; AL will contain the data in TABLE at index INDEX.

MOV EBX, OFFSET TABLE     ; BX ← Offset TABLE
MOV AL, INDEX             ; AL ← Index of the element in table
ES XLAT                   ; AL ← [ES:BX+AL]
; AL will contain the data in TABLE at index INDEX.
```

Exchange Instruction

XCHG exchanges the values in its two operands. It can be used with a LOCK prefix for purposes of multi-processor synchronization. XCHG AX,AX or XCHG EAX, EAX (depending on the BITS setting) generates the opcode 90H, and so is a synonym for NOP.

Instruction	Example	Effect	Flags Affected					
			OF	SF	ZF	AF	PF	CF
XLAT	XLAT	AL ← [DS:BX+AL]	-	-	-	-	-	-
	ES XLAT	AL ← [ES:BX+AL]	-	-	-	-	-	-
XCHG	XCHG AX,BX	SWAP(AX, BX)	-	-	-	-	-	-
XADD	XADD AX,BX	SWAP(AX, BX); AX← AX + BX	-	-	-	-	-	-

Table 5.6: Exchange and XLAT Instructions

Exchange and Add Instruction

XADD exchanges the values in its two operands, and then adds them together and writes the result into the destination (first) operand. This instruction can be used with a LOCK prefix for multi-processor synchronization purposes.

5.5 Lab Work

Pre Lab Work:

1. Study program 5.1, and explain how base conversion is performed?
2. Write, assemble and link program 5.1. You will run it in the lab using CodeView.
3. Write, assemble, link and run program 5.1.
4. Modify the program so that it prompts the user for the RADIX and the number NUM to be converted. Call the new program prog-5.3.
5. Write a program that converts from decimal to hexadecimal. Name it Prog 5.4.
6. Bring your work to the lab.

Lab Work:

1- Run program 5.2, Use CodeView to trace program and see what value is displayed.
2- Change the value of the variable NUM and see the output value.
3- Now change the value of RADIX and see the value displayed.

4- Write a program that prompts the user to enter two numbers of 4 digits each. Converts these each number to hexadecimal. Then calculates the sum and the difference of the two numbers, and finally displays the result in decimal format. Name it program 5.3.
5- Show all your work to the instructor.
6- Submit all your work at the end of the lab session.

TITLE "Lab Exp. # 5 Program # 5.1"

```
; This program converts a number NUM from Hexadecimal,
; to a new numbering base (RADIX).

.MODEL SMALL
.586
.STACK 200
.DATA
            RADIX DB 10             ; radix: 10 for decimal
            NUM   DW 0EFE4H         ; the number to be converted
                                    ; put here any other number.
            ; Note that: 0EFE4H = 61412(10)
            TEMP  DB    10 DUP(?)   ; Used to simulate a stack

.CODE
.STARTUP
            MOV   AX, NUM           ; load AX with number NUM
                                    ; display AX in decimal
            MOV   CX, 0000          ; clear digit counter
            MOVZX BX, RADIX         ; for decimal and clear BH
            MOV   SI, 0000          ; Clear register SI

DISPX1:     MOV   DX, 0000          ; clear DX
            DIV   BX                ; divide DX:AX by 10
            MOV   TEMP[SI], DL      ; save remainder
            INC   SI
            INC   CX                ; count remainder
            ADD   AX, 00            ; test for quotient of zero
            JNZ   DISPX1            ; Jump back if quotient is not zero
            DEC   SI

DISPX2:     MOV   DL, TEMP[SI]      ; get remainder
            MOV   AH, 06H           ; select function 06H
            ADD   DL, 30H           ; convert to ASCII
            INT   21H               ; display digit
            DEC   SI
            LOOP  DISPX2            ; repeat for all digits
.EXIT                               ; exit to dos
END
```

Appendix –A-

The Binary Coded Decimal system

The Binary Coded Decimal system (BCD) is used to represent the binary equivalent of the decimal system. In BCD, the binary patterns 1010 through 1111 and are not used. BCD numbers may be represented in unpacked or packed BCD number representations.

Packed versus Unpacked BCD

In unpacked format, the upper nibble of each byte is wasted. Arithmetic operations may be performed, but extra operations must be used to obtain a correct answer.

Thousands	Hundreds	Tens	Units
5	3	1	9
00000101	00000011	00000001	00001001

Table A5. 1: Unpacked BCD Representation

To eliminate wasted storage, BCD numbers are represented in packed format, where each nibble is assigned 4 bits only. Therefore, instead of requiring 4 bytes to store the BCD number 5319, we would require only 2 bytes.

Thousands-Hundreds	Tens-Units
53	19
0101 0011	0001 1001

Table A5.2: Packed BCD

BCD Arithmetic Rules

If the addition of any two digits results in a binary number between 1010 and 1111, or there is a carry into the next digit, then 6 (0110) is to be added to the current digit. Essentially, the rule is needed to "skip over" the six bit combinations that are unused by the BCD format whenever such a skip is warranted.

- Case 1

Carry from previous digit ➔	0	0	
	7	0111	
	+ 6	+ 0110	
	13	1101	
		110	← Add 6 because 1101 is not a valid
Carry to next digit ➔	1	**0011**	BCD digit

- Case 2

Carry from previous digit ➔	**1**	**1**	
	9	1001	
	+ 9	+ 1001	
	19	1 0011	
		110	← Add 6 because of carry to next digit
Carry to next digit ➔	1	**1001**	