National University of Computer and Emerging Sciences

Laboratory Manual

for

Computer Organization and Assembly Language

Course Instructors

Lab Instructor(s)

Section

Semester

Department of Computer Science

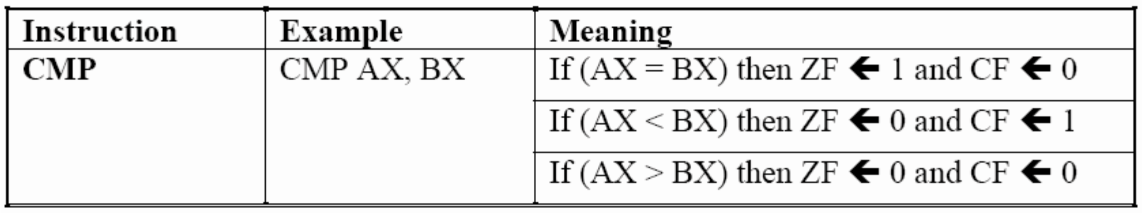
COAL Lab 7 Manual

Objectives:

- CMP Instruction
- JMP and Loop instructions
    - Unconditional transfer
        - JMP instruction
    - Conditional transfer
        - LOOP instruction
- Nested Loops
- User defined input
- Problems & Assignments

7.1 CMP instruction

The compare instruction is used to compare two numbers. At most one of these numbers may reside in memory. The compare instruction subtracts its source operand from its destination operand and sets the value of the status flags according to the subtraction result. The result of the subtraction is not stored anywhere. The flags are set as indicated in Table 7.1.



Instruction	Example	Meaning
CMP	CMP AX, BX	If (AX = BX) then ZF ← 1 and CF ← 0
		If (AX < BX) then ZF ← 0 and CF ← 1
		If (AX > BX) then ZF ← 0 and CF ← 0

Table 7.1: The Compare Instruction of the 8086 Microprocessor

Comparisons and Conditions:

The unsigned comparisons see the numbers as 0 being the smallest and 65535 being the largest with the order that 0 < 1 < 2 … < 65535. The signed comparisons see the number -32768 which has the same memory

Comparison		Smallest value					→									Largest value
Unsigned	DEC HEX	0	<	1	<	2	<	…….	<	128	<	…….	<	254	<	255
Signed	DEC	–128	<	–127	<	…….	<	–1	<	0	<	1	<	…….	<	127
	HEX	80h	<	81h	<	…….	<	FFh	<	00h	<	01h	<	…….	<	7Fh

Table 7.2: Signed and unsigned number representation using 8 bit

representation as 32768 as the smallest number and 32767 as the largest with the order -32768 < -32767 < … < -1 < 0 < 1 < 2 < … < 32767. All the negative numbers have the same representation as an unsigned number in the range 32768 … 65535 however the signed interpretation of the signed comparisons makes them be treated as negative numbers smaller than zero. Signed and unsigned number for 8 bit and 16 bit are given below.

Comparison		**Smallest value**								$\longrightarrow$						**Largest value**
Unsigned	**DEC**	0	<	1	<	2	<	…….	<	32768	<	…….	<	65534	<	65535
	HEX	0000h	<	0001h	<	0002h	<	…….	<	8000h	<	…….	<	FFFEh	<	FFFFh
Signed	**DEC**	–32768	<	–32767	<	…….	<	–1	<	0	<	1	<	…….	<	32767
	HEX	8000h	<	8001h	<	…….	<	FFFFh	<	0000h	<	0001h	<	…….	<	7FFFh

Table 7.3: Signed and unsigned number representation using 16 bit

All meaningful situations both for signed and unsigned numbers than occur after a comparison are detailed in the following table.

COMPRAISON	**FLAGS**	**EXPLAINATION**	**EXAMPLES**
		UNSIGNED COMPARISONS	

UDEST = USRC
OR
SDEST = SSRC
	ZF = 1	When the source is subtracted from the destination and both are equal the result is zero and therefore the zero flag is set. This works for both signed and unsigned numbers.	**EXAMPLE # 1**
			MOV AX, 5
			MOV BX, 5
			CMP AX, BX ; ZF = 1

UDEST < USRC	CF = 1	When an unsigned source is subtracted from an unsigned destination and the destination is smaller, borrow is needed which sets the carry flag.	**EXAMPLE # 2** **MOV AX, 2** **MOV BX, 5** **CMP AX, BX ; CF = 1**

UDEST > USRC	ZF = 0
	AND
	CF = 0
		The unsigned source and destination are not equal if the zero flag is not set and the destination is not smaller since
		no borrow was taken. Therefore the destination is greater than the source.
			EXAMPLE # 3
			MOV AX, 5
			MOV BX, 2
			CMP AX, BX ; CF = 0
			; ZF = 0

		SIGNED COMPARISONS	
SDEST < SSRC	SF ≠OF	When a signed source is subtracted from a signed destination and the answer is negative with no overflow than the destination is smaller than the source. If however there is an overflow meaning that the sign has changed unexpectedly, the meanings are reversed and a positive number signals that the destination is smaller.	**EXAMPLE # 4 (a)** **MOV AX, 2** **MOV BX, 5** **CMP AX, BX ; SF = 1** **; OF = 0** **EXAMPLE # 4 (b)** **MOV AX, 8001 H** **MOV BX, 5** **CMP AX, BX ; SF = 0** **; OF = 1**
SDEST > SSRC	SF = OF	If the zero flag is not set, it means	**EXAMPLE # 5 (a)**

		that the signed operands are not equal and if the sign and overflow match in addition to this it means that the destination is greater than the source.	MOV AX, 5 MOV BX, 2 CMP AX, BX ; SF = 0 ; OF = 0 **EXAMPLE # 5 (b)** MOV AH, 64 H MOV BH, 0CE H CMP AH, BH ; SF = 1 ; OF = 1

Table 7.4: Situations for Signed and Unsigned comparisons

7.2 JMP and LOOP Instruction

By default, the CPU loads and executes programs sequentially. But the current instruction might be conditional, meaning that it transfers control to a new location in the program based on the values of CPU status flags (Zero, Sign, Carry, etc.). Assembly language programs use conditional instructions to implement high-level statements such as IF statements and loops. Each of the conditional statements involves a possible transfer of control (jump) to a different memory address. A transfer of control, or branch, is a way of altering the order in which statements are executed. There are two basic types of transfers:

7.2.1 Unconditional Transfer

Control is transferred to a new location in all cases; a new address is loaded into the instruction pointer, causing execution to continue at the new address. The **JMP** instruction does this.

JMP Instruction

The JMP instruction causes an unconditional transfer to a destination, identified by a code label that is translated by the assembler into an offset. The syntax is

```
JMP destination
```

When the CPU executes an unconditional transfer, the offset of *destination* is moved into the instruction pointer, causing execution to continue at the new location.

Creating a Loop The JMP instruction provides an easy way to create a loop by jumping to a label at the top of the loop:

```
top:
.
.
JMP top ; repeat the endless loop
```

JMP is unconditional, so a loop like this will continue endlessly unless another way is found to exit the loop.

7.2.2 Conditional Transfer

The program branches if a certain condition is true. A wide variety of conditional transfer instructions can be combined to create conditional logic structures. The CPU interprets true/false conditions based on the contents of the ECX and Flags registers.

LOOP Instruction

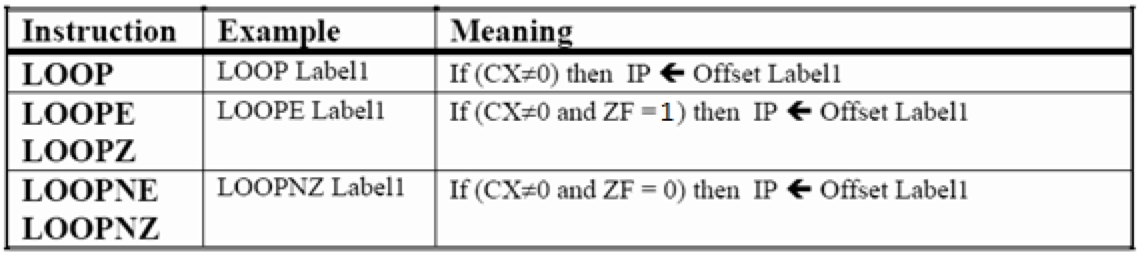
The LOOP instruction, formally known as *Loop According to ECX Counter*, repeats a block of statements a specific number of times. ECX is automatically used as a counter and is decremented each time the loop repeats. Its syntax is

```
LOOP destination
```

The loop destination must be within -128 to +127 bytes of the current location counter. The execution of the LOOP instruction involves two steps: First, it subtracts 1 from ECX. Next, it compares ECX to zero. If ECX is not equal to zero, a jump is taken to the label identified by *destination*. Otherwise, if ECX equals zero, no jump takes place, and control passes to the instruction following the loop.

In real-address mode, CX is the default loop counter for the LOOP instruction. On the other hand, the LOOPD instruction uses ECX as the loop counter, and the LOOPW instruction uses CX as the loop counter.
In the following example, we add 1 to AX each time the loop repeats. When the loop ends, AX = 5 and ECX = 0:

```
mov ax,0
mov ecx,5
L1:
inc ax
loop L1
```



Instruction	Example	Meaning
LOOP	LOOP Label1	If (CX≠0) then IP ← Offset Label1
LOOPE LOOPZ	LOOPE Label1	If (CX≠0 and ZF =1) then IP ← Offset Label1
LOOPNE LOOPNZ	LOOPNZ Label1	If (CX≠0 and ZF = 0) then IP ← Offset Label1

Table 7.5: Summary of the LOOP Instructions.

A common programming error is to inadvertently initialize ECX to zero before beginning a loop. If this happens, the LOOP instruction decrements ECX to FFFFFFFFh, and the loop repeats 4,294,967,296 times! If CX is the loop counter (in real-address mode), it repeats 65,536 times.
Occasionally, you might create a loop that is large enough to exceed the allowed relative jump range of the LOOP instruction. Following is an example of an error message generated by MASM because the target label of a LOOP instruction was too far away:

```
error A2075: jump destination too far : by 14 byte(s)
```

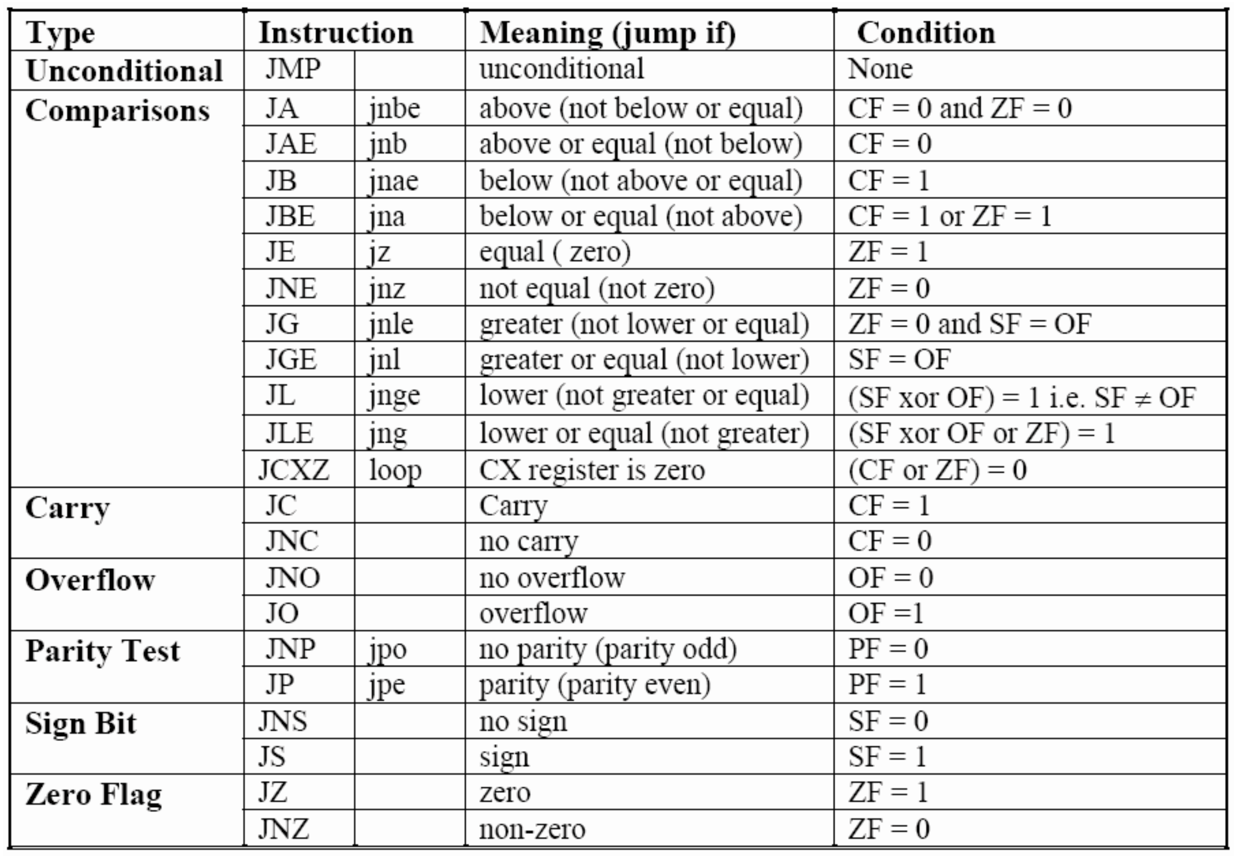
Rarely should you explicitly modify ECX inside a loop. If you do, the LOOP instruction may not work as expected. In the following example, ECX is incremented within the loop. It never reaches zero, so the loop never stops:

```
top:
.
.
inc ecx
loop top
```

If you need to modify ECX inside a loop, you can save it in a variable at the beginning of the loop and restore it just before the LOOP instruction:

```
.data
count DWORD ?
.code
mov ecx,100       ; set loop count
top:
mov count,ecx     ; save the count
.
mov ecx,20        ; modify ECX
.
mov ecx,count     ; restore loop count
loop top
```

An overview of all the jump instructions is given in Table 7.5.



Type	Instruction		Meaning (jump if)	Condition
Unconditional	JMP		unconditional	None
Comparisons	JA	jnbe	above (not below or equal)	CF = 0 and ZF = 0
	JAE	jnb	above or equal (not below)	CF = 0
	JB	jnae	below (not above or equal)	CF = 1
	JBE	jna	below or equal (not above)	CF = 1 or ZF = 1
	JE	jz	equal (zero)	ZF = 1
	JNE	jnz	not equal (not zero)	ZF = 0
	JG	jnle	greater (not lower or equal)	ZF = 0 and SF = OF
	JGE	jnl	greater or equal (not lower)	SF = OF
	JL	jnge	lower (not greater or equal)	(SF xor OF) = 1 i.e. SF ≠ OF
	JLE	jng	lower or equal (not greater)	(SF xor OF or ZF) = 1
	JCXZ	loop	CX register is zero	(CF or ZF) = 0
Carry	JC		Carry	CF = 1
	JNC		no carry	CF = 0
Overflow	JNO		no overflow	OF = 0
	JO		overflow	OF =1
Parity Test	JNP	jpo	no parity (parity odd)	PF = 0
	JP	jpe	parity (parity even)	PF = 1
Sign Bit	JNS		no sign	SF = 0
	JS		sign	SF = 1
Zero Flag	JZ		zero	ZF = 1
	JNZ		non-zero	ZF = 0

Table 7.6: Jump Instructions of the 8086 Microprocessor

7.3 Nested Loops

When creating a loop inside another loop, special consideration must be given to the outer loop counter in ECX. You can save it in a variable:

```
.data
count DWORD ?
```

```
.code
mov ecx,100      ; set outer loop count
L1:
mov count,ecx    ; save outer loop count
mov ecx,20            ; set inner loop count
L2:
.
.
loop L2          ; repeat the inner loop
mov ecx,count    ; restore outer loop count
loop L1          ; repeat the outer loop
```

As a general rule, nested loops more than two levels deep are difficult to write. If the algorithm you're using requires deep loop nesting, move some of the inner loops into subroutines.

7.4 User defined inputs

Command	Function
ReadChar	Waits for a single character to be typed at the keyboard and returns the character.
ReadDec	Reads an unsigned 32-bit decimal integer from the keyboard, terminated by the Enter key.
ReadHex	Reads a 32-bit hexadecimal integer from the keyboard, terminated by the Enter key.
ReadInt	Reads a 32-bit signed decimal integer from the keyboard, terminated by the Enter key.
ReadString	Reads a string from the keyboard, terminated by the Enter key.
WaitMsg	Displays a message and waits for a key to be pressed.
WriteBin	Writes an unsigned 32-bit integer to the console window in ASCII binary format.
WriteBinB	Writes a binary integer to the console window in byte, word, or doubleword format.
WriteChar	Writes a single character to the console window.
WriteDec	Writes an unsigned 32-bit integer to the console window in decimal format.
WriteHex	Writes a 32-bit integer to the console window in hexadecimal format.
WriteHexB	Writes a byte, word, or doubleword integer to the console window in hexadecimal format.
WriteInt	Writes a signed 32-bit integer to the console window in decimal format.

Problem(s) / Assignment(s)

Discussion & Practice **Estimated completion time: 1 hr, 30 mins**

Problem 7.1: *Separating even odd numbers* **Estimated completion time:20 mins**

Write a program that determines whether an integer is even or odd and display a message of E\O number of times of the input.

Sample:

```
Enter your value : 3
OOO
```

Problem 7.2: *Fibonacci number generation* **Estimated completion time:15 mins**

Write a program that generate the first six Fibonacci number sequence (1, 1, 2, 3, 5, 8 upto user defined integer). Use an array named Fibonacci of word type. You may initialize the first two places of array with 1's and next four places with 0. Use the following rule and save results in the same array

The rule to generate sequence is $F_n = F_{n-1} + F_{n-2}$. Call DumpMEM to display Fibonacci sequence.

Problem 7.3: *Working with different shapes* **Estimated completion time:20 mins**

Write a program that takes input from the user and displays the following pattern.
Sample:
Enter your input: 5
Output:

```
*****
 ****
  ***
   **
    *
```

You are done with your exercise(s), make your submission ☺