Module 4

Assembly Conditions

Subtopic 1

Assembly Conditions

Objectives

- Understand conditions and loops in assembly programming
- Understand how to use array
- Apply the concept of numbers and strings in assembly programming

Conditional execution in assembly language is accomplished by several looping and branching instructions. These instructions can change the flow of control in a program. Conditional execution is observed in two scenarios:

SN	Conditional Instructions
1	**Unconditional jump** This is performed by the JMP instruction. Conditional execution often involves a transfer of control to the address of an instruction that does not follow the currently executing instruction. Transfer of control may be forward, to execute a new set of instructions or backward, to re-execute the same steps.
2	**Conditional jump** This is performed by a set of jump instructions j<condition> depending upon the condition. The conditional instructions transfer the control by breaking the sequential flow and they do it by changing the offset value in IP.

CMP Instruction

The CMP instruction compares two operands. It is generally used in conditional execution. This instruction basically subtracts one operand from the other for comparing whether the operands are equal or not. It does not disturb the destination or source operands. It is used along with the conditional jump instruction for decision making.

Syntax

```
CMP destination, source
```

CMP Instruction

CMP compares two numeric data fields. The destination operand could be either in register or in memory. The source operand could be a constant (immediate) data, register or memory.

Example

```
CMP DX,     00  ; Compare the DX value with zero
JE  L7      ; If yes, then jump to label L7
.
.
L7: ...
```

CMP is often used for comparing whether a counter value has reached the number of times a loop needs to be run. Consider the following typical condition:

```
INC   EDX
CMP   EDX, 10       ; Compares whether the counter has reached 10
JLE   LP1           ; If it is less than or equal to 10, then jump
                    ; to LP1 Unconditional Jump
```

CMP Instruction

As mentioned earlier, this is performed by the JMP instruction. Conditional execution often involves a transfer of control to the address of an instruction that does not follow the currently executing instruction. Transfer of control may be forward, to execute a new set of instructions or backward, to re-execute the same steps.

Syntax

The JMP instruction provides a label name where the flow of control is transferred immediately. The syntax of the JMP instruction is:

```
JMP label
```

CMP Instruction

The following code snippet illustrates the JMP instruction:

```
MOV  AX, 00    ; Initializing AX to 0
MOV  BX, 00    ; Initializing BX to 0
MOV  CX, 01    ; Initializing CX to 1
L20:
ADD  AX, 01    ; Increment AX
ADD  BX, AX    ; Add AX to BX
SHL  CX, 1     ; shift left CX, this in turn doubles the CX value
JMP  L20       ; repeats the statements
```

Conditional Jump

If some specified condition is satisfied in conditional jump, the control flow is transferred to a target instruction. There are numerous conditional jump instructions depending upon the condition and data.

Following are the conditional jump instructions used on signed data used for arithmetic operations:

Instruction	Description	Flags tested
JE/JZ	Jump Equal or Jump Zero	ZF
JNE/JNZ	Jump not Equal or Jump Not Zero	ZF
JG/JNLE	Jump Greater or Jump Not Less/Equal	OF, SF, ZF
JGE/JNL	Jump Greater or Jump Not Less	OF, SF
JL/JNGE	Jump Less or Jump Not Greater/Equal	OF, SF
JLE/JNG	Jump Less/Equal or Jump Not Greater	OF, SF, ZF

Conditional Jump

Following are the conditional jump instructions used on unsigned data used for logical operations:

Instruction	Description	Flags tested
JE/JZ	Jump Equal or Jump Zero	ZF
JNE/JNZ	Jump not Equal or Jump Not Zero	ZF
JA/JNBE	Jump Above or Jump Not Below/Equal	CF, ZF
JAE/JNB	Jump Above/Equal or Jump Not Below	CF
JB/JNAE	Jump Below or Jump Not Above/Equal	CF
JBE/JNA	Jump Below/Equal or Jump Not Above	AF, CF

Conditional Jump

The following conditional jump instructions have special uses and check the value of flags:

Instruction	Description	Flags tested
JXCZ	Jump if CX is Zero	none
JC	Jump If Carry	CF
JNC	Jump If No Carry	CF
JO	Jump If Overflow	OF
JNO	Jump If No Overflow	OF
JP/JPE	Jump Parity or Jump Parity Even	PF
JNP/JPO	Jump No Parity or Jump Parity Odd	PF
JS	Jump Sign (negative value)	SF
JNS	Jump No Sign (positive value)	SF

Conditional Jump

The syntax for the J<condition> set of instructions:

```
CMP   AL, BL
JE    EQUAL
CMP   AL, BH
JE    EQUAL
CMP   AL, CL
JE    EQUAL
NON_EQUAL: ...
EQUAL: ...
```

Conditional Jump

The following program displays the largest of three variables. The variables are double- digit variables. The three variables num1, num2 and num3 have values 47, 72 and 31, respectively:

```
section     .text
    global _start         ;must be declared for using gcc

_start:     ;tell linker entry point
     mov    ecx, [num1]
            cmp    ecx, [num2]
            jg     check_third_num
            mov    ecx, [num3]
   check_third_num:
            cmp    ecx, [num3]
            jg     _exit
            mov    ecx, [num3]
   _exit:
        mov    [largest], ecx
        mov    ecx,msg
        mov    edx, len
        mov    ebx,1      ;file descriptor (stdout)
        mov    eax,4      ;system call number (sys_write)
        int    0x80;call kernel
        mov    ecx,largest
        mov    edx, 2
        mov    ebx,1      ;file descriptor (stdout)
        mov    eax,4      ;system call number (sys_write)
        int    0x80;call kernel

        mov    eax, 1
        int    80h

section     .data
    msg db "The largest digit is: ", 0xA,0xD
    len equ $- msg
    num1 dd '47'
    num2 dd '22'
    num3 dd '31'

segment .bss
   largest resb 2
```

When the above code is compiled and executed, it produces the following result:

```
The largest digit is:
47
```

Loops

The JMP instruction can be used for implementing loops. For example, the following code snippet can be used for executing the loop-body 10 times.

```
MOV  CL, 10
L1:
<LOOP-BODY>
DEC  CL
JNZ  L1
```

The processor instruction set, however, includes a group of loop instructions for implementing iteration. The basic LOOP instruction has the following syntax:

```
LOOP label
```

Loops

Where, label is the target label that identifies the target instruction as in the jump instructions. The LOOP instruction assumes that the ECX register contains the loop count. When the loop instruction is executed, the ECX register is decremented and the control jumps to the target label, until the ECX register value, i.e., the counter reaches the value zero.

The above code snippet could be written as:

```
mov ECX,10
l1:
<loop body>
loop l1
```

Loops

The following program prints the number 1 to 9 on the screen:

```
section     .text
    global _start         ;must be declared for using gcc
_start:                     ;tell linker entry point
     mov ecx,10
     mov eax, '1'

l1:
     mov [num], eax
     mov eax, 4
     mov ebx, 1
     push ecx
     mov ecx, num
     mov edx, 1
     int 0x80
     mov eax, [num]
     sub eax, '0'
     inc eax
     add eax, '0'
     pop ecx
     loop l1
     mov eax,1       ;system call number (sys_exit)
     int 0x80        ;call kernel
section     .bss
num resb 1
```

When the above code is compiled and executed, it produces the following result:

```
123456789:
```

Numbers

Numerical data is generally represented in binary system. Arithmetic instructions operate on binary data. When numbers are displayed on screen or entered from keyboard, they are in ASCII form.

Numbers

So far, we have converted this input data in ASCII form to binary for arithmetic calculations and converted the result back to binary. The following code shows this:

```
section     .text
    global _start         ;must be declared for using gcc
_start:     ;tell linker entry point
     mov    eax,'3'
     sub    eax, '0'
     mov    ebx, '4'
     sub    ebx, '0'
     add    eax, ebx
     add    eax, '0'
     mov    [sum], eax
     mov    ecx,msg
     mov    edx, len
     mov    ebx,1  ;file descriptor (stdout)
     mov    eax,4  ;system call number (sys_write)
     int    0x80   ;call kernel
     mov    ecx,sum
     mov    edx, 1
     mov    ebx,1  ;file descriptor (stdout)
     mov    eax,4  ;system call number (sys_write)
     int    0x80   ;call kernel
     mov    eax,1  ;system call number (sys_exit)
     int    0x80   ;call kernel
section .data
msg db "The sum is:", 0xA,0xD
len equ $ - msg
segment .bss
sum resb 1
```

When the above code is compiled and executed, it produces the following result:

```
The sum is:
7
```

Such conversions, however, have an overhead, and assembly language programming allows processing numbers in a more efficient way, in the binary form. Decimal numbers can be represented in two forms:

- ASCII form
- BCD or Binary Coded Decimal form

ASCII Representation

The following example will ask two digits from the user, store the digits in the EAX and EBX register, respectively, add the values, store the result in a memory location 'res' and finally display the result.

In ASCII representation, decimal numbers are stored as string of ASCII characters. For example, the decimal value 1234 is stored as:

```
31 32 33 34H
```

Where, 31H is ASCII value for 1, 32H is ASCII value for 2, and so on. There are four instructions for processing numbers in ASCII representation:

- AAA - ASCII Adjust After Addition
- AAS - ASCII Adjust After Subtraction
- AAM - ASCII Adjust After Multiplication
- AAD - ASCII Adjust Before Division

These instructions do not take any operands and assume the required operand to be in the AL register.

ASCII Representation

The following example uses the AAS instruction to demonstrate the concept:

```
section     .text
    global _start         ;must be declared for using gcc
_start:     ;tell linker entry point
     sub     ah, ah
     mov     al, '9'
     sub     al, '3'
     aas
     or      al, 30h
     mov     [res], ax

     mov    edx,len      ;message length
     mov    ecx,msg      ;message to write
     mov    ebx,1  ;file descriptor (stdout)
     mov    eax,4  ;system call number (sys_write)
     int    0x80   ;call kernel

     mov    edx,1  ;message length
     mov    ecx,res      ;message to write
     mov    ebx,1  ;file descriptor (stdout)
     mov    eax,4  ;system call number (sys_write)
     int    0x80   ;call kernel
     mov    eax,1  ;system call number (sys_exit)
     int    0x80   ;call kernel

section     .data
msg db 'The Result is:',0xa
len equ $ - msg
section .bss
res resb 1
```

When the above code is compiled and executed, it produces the following result:

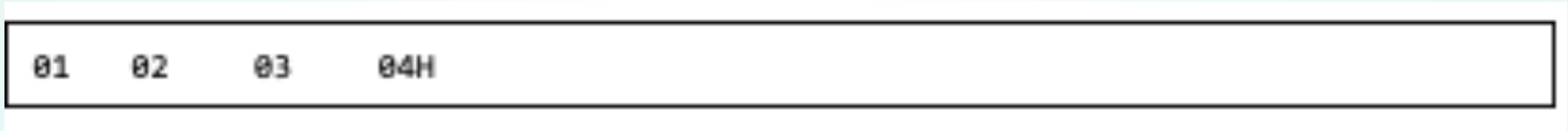
```
The Result is:
6
```

BCD Representation

There are two types of BCD representation:

- **Unpacked BCD representation**
- **Packed BCD representation**
- **In unpacked BCD representation, each byte stores the binary equivalent of a decimal digit.**

For example, the number 1234 is stored as:



BCD Representation

There are two instructions for processing these numbers:

- **AAM - ASCII Adjust After Multiplication**
- **AAD - ASCII Adjust Before Division**

The four ASCII adjust instructions, AAA, AAS, AAM, and AAD, can also be used with unpacked BCD representation. In packed BCD representation, each digit is stored using four bits. Two decimal digits are packed into a byte.

For example, the number 1234 is stored as:

```
12    34H
```

BCD Representation

There are two instructions for processing these numbers:

- **AAM - ASCII Adjust After Multiplication**
- **AAD - ASCII Adjust Before Division**

The four ASCII adjust instructions, AAA, AAS, AAM, and AAD, can also be used with unpacked BCD representation. In packed BCD representation, each digit is stored using four bits. Two decimal digits are packed into a byte.

For example, the number 1234 is stored as:

12 34H

There are two instructions for processing these numbers:

- **DAA - Decimal Adjust After Addition**
- **DAS - decimal Adjust After Subtraction**

There is no support for multiplication and division in packed BCD representation.

BCD Representation

The following program adds up two 5-digit decimal numbers and displays the sum. It uses the above concepts:

```
section     .text
    global _start         ;must be declared for using gcc

_start:     ;tell linker entry point

     mov     esi, 4  ;pointing to the rightmost digit
     mov     ecx, 5  ;num of digits
     clc
add_loop:
     mov    al, [num1 + esi]
     adc    al, [num2 + esi]
     aaa
     pushf
     or     al, 30h
     popf
     mov    [sum + esi], al
     dec    esi
     loop   add_loop
     mov    edx,len      ;message length
     mov    ecx,msg      ;message to write
     mov    ebx,1  ;file descriptor (stdout)
     mov    eax,4  ;system call number (sys_write)
     int    0x80   ;call kernel

     mov    edx,5  ;message length
     mov    ecx,sum      ;message to write
     mov    ebx,1  ;file descriptor (stdout)
     mov    eax,4  ;system call number (sys_write)
     int    0x80   ;call kernel
```

```
     mov    eax,1  ;system call number (sys_exit)
     int    0x80   ;call kernel

section     .data
msg db 'The Sum is:',0xa
len equ $ - msg
num1 db '12345'
num2 db '23456'
sum db '     '
```

When the above code is compiled and executed, it produces the following result:

```
The Sum is:
35801
```

Strings

We have already used variable length strings in our previous examples. The variable length strings can have as many characters as required.

Generally, we specify the length of the string by either of the two ways:

- **Explicitly storing string length**
- **Using a sentinel character**

We can store the string length explicitly by using the $ location counter symbol that represents the current value of the location counter. In the following example:

```
msg  db  'Hello, world!',0xa ;our dear string
len  equ  $ - msg            ;length of our dear string
```

Strings

$ points to the byte after the last character of the string variable msg. Therefore, $-msg gives the length of the string. We can also write:

```
msg db 'Hello, world!',0xa ;our dear string
len equ 13                 ;length of our dear string
```

Alternatively, you can store strings with a trailing sentinel character to delimit a string instead of storing the string length explicitly. The sentinel character should be a special character that does not appear within a string.

For example:

```
message DB 'I am loving it!', 0
```

String Instructions

Each string instruction may require a source operand, a destination operand or both. For 32-bit segments, string instructions use ESI and EDI registers to point to the source and destination operands, respectively.

For 16-bit segments, however, the SI and the DI registers are used to point to the source and destination, respectively.

There are five basic instructions for processing strings.

- MOVS- This instruction moves 1 Byte, Word or Doubleword of data from memory location to another.
- LODS - This instruction loads from memory. If the operand is of one byte, it is loaded into the AL register, if the operand is one word, it is loaded into the AX register and a doubleword is loaded into the EAX register.
- STOS - This instruction stores data from register (AL, AX, or EAX) to memory.
- CMPS - This instruction compares two data items in memory. Data could be of a byte size, word or doubleword.
- SCAS - This instruction compares the contents of a register (AL, AX or EAX) with the contents of an item in memory.

String Instructions

Each of the above instruction has a byte, word, and doubleword version; and string instructions can be repeated by using a repetition prefix.

These instructions use the ES:DI and DS:SI pair of registers, where DI and SI registers contain valid offset addresses that refers to bytes stored in memory. SI is normally associated with DS (data segment) and DI is always associated with ES (extra segment).

The DS:SI (or ESI) and ES:DI (or EDI) registers point to the source and destination operands, respectively. The source operand is assumed to be at DS:SI (or ESI) and the destination operand at ES:DI (or EDI) in memory.

For 16-bit addresses, the SI and DI registers are used, and for 32-bit addresses, the ESI and EDI registers are used.

String Instructions

The following table provides various versions of string instructions and the assumed space of the operands.

Basic Instruction	Operands at	Byte Operation	Word Operation	Double word Operation
MOVS	ES:DI, DS:EI	MOVSB	MOVSW	MOVSD
LODS	AX, DS:SI	LODSB	LODSW	LODSD
STOS	ES:DI, AX	STOSB	STOSW	STOSD
CMPS	DS:SI, ES: DI	CMPSB	CMPSW	CMPSD
SCAS	ES:DI, AX	SCASB	SCASW	SCASD

Repetition Prefixes

The REP prefix, when set before a string repetition of the instruction based on a counter placed at the CX register. REP executes the instruction, decreases CX by 1, and checks whether CX is zero. It repeats the instruction processing until CX is zero.

The Direction Flag (DF) determines the direction of the operation.

- **Use CLD (Clear Direction Flag, DF = 0) to make the operation left to right.**
- **Use STD (Set Direction Flag, DF = 1) to make the operation right to left.**

Repetition Prefixes

The REP prefix also has the following variations:

- **REP: it is the unconditional repeat. It repeats the operation until CX is zero.**
- **REPE or REPZ: It is conditional repeat. It repeats the operation while the zero flag indicates equal/zero. It stops when the ZF indicates not equal/zero or when CX is zero.**
- **REPNE or REPNZ: It is also conditional repeat. It repeats the operation while the zero flag indicates not equal/zero. It stops when the ZF indicates equal/zero or when CX is decremented to zero.**

Arrays

We have already discussed that the data definition directives to the assembler are used for allocating storage for variables. The variable could also be initialized with some specific value. The initialized value could be specified in hexadecimal, decimal or binary form.

For example, we can define a word variable 'months' in either of the following way:

```
MONTHS      DW      12
MONTHS      DW      0CH
MONTHS      DW      0110B
```

The data definition directives can also be used for defining a one-dimensional array. Let us define a one-dimensional array of numbers.

```
NUMBERS     DW  34,  45,  56,  67,  75, 89
```

Arrays

The above definition declares an array of six words each initialized with the numbers 34, 45, 56, 67, 75, 89. This allocates 2x6 = 12 bytes of consecutive memory space. The symbolic address of the first number will be NUMBERS and that of the second number will be NUMBERS + 2 and so on.

Let us take up another example. You can define an array named inventory of size 8, and initialize all the values with zero, as:

```
INVENTORY   DW  0
            DW  0
            DW  0
            DW  0
            DW  0
            DW  0
            DW  0
            DW  0
```

Which can be abbreviated as:

```
INVENTORY   DW  0, 0 , 0 , 0 , 0 , 0 , 0 , 0
```

The TIMES directive can also be used for multiple initializations to the same value. Using TIMES, the INVENTORY array can be defined as:

```
INVENTORY TIMES 8 DW 0
```

Arrays

The following example demonstrates the above concepts by defining a 3-element array x, which stores three values: 2, 3 and 4. It adds the values in the array and displays the sum 9:

```
section     .text
    global _start ;must be declared for linker (ld)
_start:

      mov  eax,3      ;number bytes to be summed
      mov  ebx,0      ;EBX will store the sum
      mov  ecx, x     ;ECX will point to the current element to be summed
top:  add  ebx, [ecx]
      add  ecx,1      ;move pointer to next element
      dec  eax        ;decrement counter
      jnz  top        ;if counter not 0, then loop again
done:
      add   ebx, '0'
      mov  [sum], ebx ;done, store result in "sum"
display:
      mov  edx,1      ;message length
      mov  ecx, sum   ;message to write
      mov  ebx, 1     ;file descriptor (stdout)
      mov  eax, 4     ;system call number (sys_write)
      int  0x80       ;call kernel
      mov  eax, 1     ;system call number (sys_exit)
      int  0x80       ;call kernel

section     .data
global x
x:
      db  2
      db  4
      db  3
sum:
      db  0
```

When the above code is compiled and executed, it produces the following result:

```
9
```