UNIT – 4: 8086 Programming Using Assembly Level Language

The Structure of a typical assembly program

A program has always the following general structure:

Structure of an Assembly Language Program

```
.model small          ; Select a memory model.
.stack stack_size     ; Define the stack size
.data                 ; Variable and array declarations;
                      ; Declare variables at this level
.code
main proc             ; Write the program main code at this level

main endp

; Other Procedures

                      ; Always organize your program
                      ; into procedures

end main              ;To mark the end of the source file
```

The Model directive:

The model directive specifies the total amount of memory the program would take. In other words, it gives information on how much memory the assembler would allocate for the program. This depends on the size of the data and the size of the program or code.

Segment directives:

Segments are declared using directives. The following directives are used to specify the following segments:

- stack
- data
- code

Stack Segment:

- Used to set aside storage for the stack
- Stack addresses are computed as offsets into this segment
- Use: .stack followed by a value that indicates the size of the stack

Data Segments:

- Used to set aside storage for variables.
- Constants are defined within this segment in the program source.
- Variable addresses are computed as offsets from the start of this segment
- Use: .data followed by declarations of variables or definitions of constants.

Code Segment:
The code segment contains executable instructions macros and calls to procedures.
Use: .code followed by a sequence of program statements

Implementation of control structures: IF-THEN, IF-THEN-ELSE, MULTIPLE IF-THEN-ELSE.

Assembly languages are a family of low-level languages for programming computers, microprocessors, microcontrollers, and other (usually) integrated circuits. They implement a symbolic representation of the numeric machine codes and other constants needed to program architecture particular CPU. A program written in assembly language consists of a series of instructions--mnemonics that correspond to a stream of executable instructions, when translated by an assembler that can be loaded into memory and executed. A utility program called an assembler is used to translate assembly language statements into the target computer's machine code.

There are three basic kinds of control structures:

1. Sequences
2. Branching

3. Loops

It is proved that any logic problem can be solved with only sequence, choice (for e.g., if-then-else) and repetition (do-while). This is called as Structured Theorem.

Sequential structures:
Sequential structures are structures that are stepped through sequential. These are also called sequences or iterative structures. Basic arithmetic, logical, and bit operations are in this category. Data moves and copies are sequences.

Branching structures:
Branching structures consist of direct and indirect jumps (including the infamous “GOTO”), conditional jumps (IF), nested ifs, and case (or switch) structures.

Loop structures:
The basic looping structures are DO iterative, do WHILE, and do UNTIL. An infinite loop is one that has no exit. Normally, infinite loops are programming errors, but event loops and task schedulers are examples of intentional infinite loops.

Introduction to Decisions:

UNIT – 4: 8086 Programming Using Assembly Level Language

In its most basic form, a decision is some sort of branch within the code that switches between two possible execution paths based on some condition. Normally (though not always), conditional instruction sequences are implemented with the conditional jump instructions. Conditional instructions correspond to the If…Then...else statement in Pascal:

IF (condition is true) THEN stmt1 ELSE stmt2;

Assembly language, as usual, offers much more flexibility when dealing with conditional statements. Consider the following Pascal statement:

IF ((X<Y) and (Z > T)) or (A <> B) THEN stmt1;

Approach to converting this statement into assembly language might produce:

```
            mov     cl, 1         ;Assume true
            mov     ax, X
            cmp     ax, Y
            jl      IsTrue
            mov     cl, 0         ;This one's false
IsTrue:     mov     ax, Z
            cmp     ax, T
            jg      AndTrue
            mov     cl, 0         ;It's false now
AndTrue:    mov     al, A
            cmp     al, B
            je      OrFalse
            mov     cl, 1         ;Its true if A <> B
OrFalse:    cmp     cl, 1
            jne     SkipStmt1
        <Code for stmt1 goes here>
SkipStmt1:
```

It takes a considerable number of conditional statements just to process the expression in the example above. This roughly corresponds to the (equivalent) Pascal statements:

```
cl := true;
IF (X >= Y) then cl := false;
IF (Z <= T) then cl := false;
IF (A <> B) THEN cl := true;
IF (CL = true) then stmt1;
```

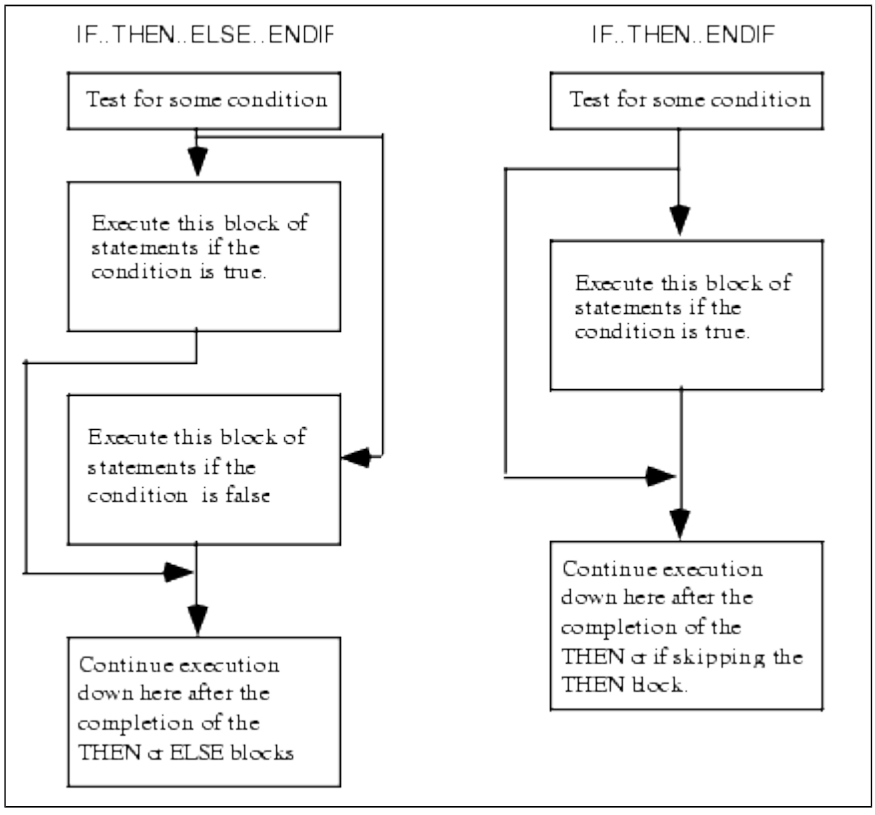
Now compare this with the following "improved" code:

```
            mov     ax, A
            cmp     ax, B
            jne     DoStmt
```

```
        mov     ax, X
        cmp     ax, Y
        jnl     SkipStmt
        mov     ax, Z
        cmp     ax, T
        jng     SkipStmt
DoStmt:
        <Place code for Stmt1 here>
SkipStmt:
```

Organization of complex expressions in a conditional sequence can affect the efficiency of the code. Therefore, care should be exercised when dealing with conditional sequences in assembly language. Conditional statements may be broken down into three basic categories: if..then..else statements, case statements, and indirect jumps.

IF-THEN-ELSE



This involves an extension to the previous IF body. The conditional false branch now jumps to an else clause, and the if body jumps unconditionally to the end of the if else statement.

```
if:                 ; comparison
                    ; branch false to else clause
                    ; if body statements
    jmp endif
else:
                    ; else statements
                    ;
endif:
```

The same principles apply to the various forms that expressions can take. eg,

```
IF X = 2 THEN Y: = Y + 4 ELSE Z: = 0;

if:     LDAA    X
        CMPA    #2
        BNE     else
        LDAA    Y
        ADDA    #4
        STAA    Y
        JMP endif
Else:   LDAA    #0
        STAA    Z
endif:
```

MULTIPLE IF-THEN-ELSE.

The Pascal case statement takes the following form :

```
CASE variable OF
    const1:stmt1;
    const2:stmt2;
     .
     .
     .
    constn:stmtn
END;
```

When this statement executes, it checks the value of variable against the constants const1 ... constn. If a match is found then the corresponding statement executes. Standard Pascal places a few restrictions on the case statement. First, if the value of variable isn't in the list of constants, the result of the case statement is undefined. Second, all the constants appearing as case, labels must be unique. The reason for these restrictions will become clear in a moment.

Most introductory programming texts introduce the case statement by explaining it as a

sequence of if..then..else statements. They might claim that the following two pieces of Pascal code are equivalent:

```
CASE I OF
        0: WriteLn ('I=0');
        1: WriteLn ('I=1');
        2: WriteLn ('I=2');
END;

IF I = 0 THEN WriteLn ('I=0')
ELSE IF I = 1 THEN WriteLn ('I=1')
ELSE IF I = 2 THEN WriteLn ('I=2');
```

While semantically these two code segments may be the same, their implementation is usually different. Whereas the if..then..else if chain does a comparison for each conditional statement in the sequence, the case statement normally uses an indirect jump to transfer control to any one of several statements with a single computation. Consider the two examples presented above, they could be written in assembly language with the following code:

```
                mov     bx, I
                shl     bx, 1           ;Multiply BX by two
                jmp     cs:JmpTbl [bx]

JmpTbl          word    stmt0, stmt1, stmt2

Stmt0:          print
                byte    "I=0",cr,lf,0
                jmp     EndCase

Stmt1:          print
                byte    "I=1",cr,lf,0
                jmp     EndCase

Stmt2:          print
                byte    "I=2",cr,lf,0

EndCase:

; IF..THEN..ELSE form:

                mov     ax, I
                cmp     ax, 0
                jne     Not 0
                print
                byte    "I=0",cr,lf,0
                jmp     End Of IF
```

```
Not 0:      cmp    ax, 1
            jne    Not 1
            print
            byte   "I=1",cr,lf,0
            jmp    End Of IF

Not 1:      cmp    ax, 2
            jne    End Of IF
            Print
            byte   "I=2",cr,lf,0
End Of IF:
```

Two things should become readily apparent: the more (consecutive) cases you have, the more efficient the jump table implementation becomes (both in terms of space and speed). Except for trivial cases, the case statement is almost always faster and usually by a large margin. As long as the case labels are consecutive values, the case statement version is usually smaller as well.

What happens if you need to include non-consecutive case labels or you cannot be sure that the case variable doesn't go out of range? Many Pascals have extended the definition of the case statement to include an otherwise clause. Such a case statement takes the following form:

```
CASE variable OF
    const:stmt;
    const:stmt;
     . .
     . .
     . .
    const:stmt;
    OTHERWISE stmt
END;
```

If the value of variable matches one of the constants making up the case labels, then the associated statement executes. If the variable's value doesn't match any of the case labels, then the statement following the otherwise clause executes. The otherwise clause is implemented in two phases. First, you must choose the minimum and maximum values that appear in a case statement. In the following case statement, the smallest case label is five, the largest is 15:

```
CASE I OF
    5:stmt1;
    8:stmt2;
    10:stmt3;
    12:stmt4;
    15:stmt5;
    OTHERWISE stmt6
END;
```

Before executing the jump through the jump table, the 8086 implementation of this case statement should check the case variable to make sure it's in the range 5..15. If not, control should be immediately transferred to stmt6:

```
        mov     bx, I
        cmp     bx, 5
        jl      Otherwise
        cmp     bx, 15
        jg      Otherwise
        shl     bx, 1
        jmp     cs:JmpTbl-10[bx]
```

The only problem with this form of the case statement as it now stands is that it doesn't properly handle the situation where I is equal to 6, 7, 9, 11, 13, or 14. Rather than sticking extra code in front of the conditional jump, you can stick extra entries in the jump table as follows:

```
        mov     bx, I
        cmp     bx, 5
        jl      Otherwise
        cmp     bx, 15
        jg      Otherwise
        shl     bx, 1
        jmp     cs:JmpTbl-10[bx]

Otherwise:      {put stmt6 here}
        jmp     CaseDone

JmpTbl  word    stmt1, Otherwise, Otherwise, stmt2, Otherwise
        word    stmt3, Otherwise, stmt4, Otherwise, Otherwise
        word    stmt5
        etc.
```

Note that the value 10 is subtracted from the address of the jump table. The first entry in the table is always at offset zero while the smallest value used to index into the table is five (which is multiplied by two to produce 10). The entries for 6, 7, 9, 11, 13, and 14 all point at the code for the Otherwise clause, so if I contains one of these values, the Otherwise clause will be executed.

Implementation of looping structures: WHILE-DO, REPEAT-UNTIL

The most general loop is the while loop. It takes the following form:

WHILE Boolean expression DO statement;

There are two important points to note about the while loop. First, the test for termination appears at the beginning of the loop. Second as a direct consequence of

the position of the termination test, the body of the loop may never execute. If the termination condition always exists, the loop body will always be skipped over.

Consider the following Pascal while loop:

```
I: = 0;
WHILE (I<100) do I: = I + 1;
```

I: = 0; is the initialization code for this loop. I is a loop control variable, because it controls the execution of the body of the loop. (I<100) is the loop termination condition. That is, the loop will not terminate as long as I is less than 100. I: =I+1; is the loop body. This is the code that executes on each pass of the loop. You can convert this to 80x86 assembly language as follows:

```
            mov     I, 0
WhileLp:    cmp     I, 100
            jge     WhileDone
            inc     I
            jmp     WhileLp

WhileDone:
```

Note that a Pascal while loop can be easily synthesized using an if and a goto statement. For example, the Pascal while loop presented above can be replaced by:

```
I: = 0;
1:      IF (I<100) THEN BEGIN
            I: = I + 1;
            GOTO 1;
END;
```

More generally, any while loop can be built up from the following:

```
    optional initialization code
1:  IF not termination condition THEN BEGIN
        loop body
        GOTO 1;
END;
```

REPEAT-UNTIL

The repeat...until (do...while) loop tests for the termination condition at the end of the loop rather than at the beginning. In Pascal, the repeat...until loop takes the following form:

```
optional initialization code
REPEAT
    loop body
UNTIL termination condition
```

This sequence executes the initialization code, the loop body, then tests some condition to see if the loop should be repeated. If the Boolean expression evaluates to false, the loop repeats; otherwise the loop terminates. The two things to note about the repeat...until loop is that the termination test appears at the end of the loop and, as a direct consequence of this, the loop body executes at least once.

Like the while loop, the repeat...until loop can be synthesized with an if statement and a goto . You would use the following:

```
        initialization code
1:          loop body
        IF NOT termination condition THEN GOTO 1
```

Based on the above example, you can easily synthesize repeat..until loop in assembly language.