Chapter 5 – Branching and Looping

Computers derive much of their power from their ability to execute code selectively and from the speed at which they execute repetitive algorithms.

The 80×86 microprocessor can execute some instructions that are roughly comparable to **for** statements, but most branching and looping is done with 80×86 statements that are similar to, but even more primitive than, simple **if** and **goto** statements.

The objective of this chapter is to describe the machine implementation of language structures such as if-then, if-then-else, while, until, and for.

Chapter 5 – Branching and Looping

5.1 Unconditional Jumps

The 80×86 jmp (jump) instruction corresponds to **goto** in a high-level language. As coded in assembly language, jmp usually has the form

```
        jmp     StatementLabel
```

where *StatementLabel* corresponds to the name field of some other assembly language statement.

```
        jmp     quit                ;exit from program
                .
                .
quit:   INVOKE ExitProcess,0        ;exit with return code 0
                .
                .
```

5.1 Unconditional Jumps

Figure 5.1 shows a complete example:

A program that will input numbers repeatedly and, after each number is entered, display the count of the numbers so far, the cumulative sum, and the average. The program implements the following pseudocode design.

```
display instructions;
 sum := 0;
count := 0;
forever loop
        prompt for number;
        input ASCII characters for number;
        convert number to 2's complement form;
        add number to sum; add 1 to count;
        convert count to ASCII;
        display label and count;
        convert sum to ASCII;
        display label and sum;
        average := sum / count;
        display label and average;
end loop;
```

Chapter 5 – Branching and Looping

5.1 Unconditional Jumps

There are several 80×86 jmp instructions, in two major groups. All work by changing the value in the instruction pointer register EIP, so that the next instruction to be executed comes from a new address rather than from the address immediately following the current instruction. Jumps can be **intersegment**, changing the code segment register CS as well as EIP.

However, this does not happen with flat memory model programming, so these instructions will not be covered. The **intrasegment** jumps are summarized in Fig. 5.3; the first two are the most commonly used.

Type	Clock Cycles 386	486	Pentium	Number of Bytes	opcode
relative near	7+	3	1	5	E9
relative short	7+	3	1	2	EB
register indirect	10+	5	2	2	FF
memory indirect	10+	5	2	2+	FF

5.1 Unconditional Jumps

Each relative jump instruction contains the displacement of the target from the jmp statement itself. This displacement is added to the address of the next instruction to find the address of the target.

The displacement is a signed number, positive for a forward reference and negative for a backward reference.

For the relative short version of the instruction, only a single byte of displacement is stored; this is changed to a sign extended to a doubleword before the addition. The 8-bit displacement in an relative short jump can serve for a target statement up to 128 bytes before or 127 bytes after the jmp instruction.

The relative near format includes a 32-bit displacement. The 32-bit displacement in a relative near jump instruction can serve for a target statement up to 2,147,483,648 bytes before or 2,147,483,647 bytes after the jmp instruction.

The assembler uses a short jump if the target is within the small range in order to generate more compact code.

Chapter 5 – Branching and Looping

5.1 Unconditional Jumps

The indirect jump instructions use a 32-bit address for the target rather than a displacement. However, this address is not encoded in the instruction itself. Instead, it is either in a register or in a memory doubleword. Thus the format

```
jmp    edx
```

means to jump to the address stored in EDX. The memory indirect format can use any valid reference to a doubleword of memory. If Target is declared as a DWORD in the data section, then

```
jmp    target
```

jumps to the address stored in that doubleword, not to that point in the data section. Using register indirect addressing, you could have

```
jmp    DWORD PTR [ebx]
```

that causes a jump to the address stored at the doubleword whose address is in EBX! Fortunately, these indirect forms are rarely needed.

5.2 Conditional Jumps, Compare Instructions, and …

Conditional jump instructions make it possible to implement if structures, other selection structures, and loop structures in 80×86 machine language. There are many of these instructions. Each has the format

```
j-  targetStatement
```

where the last part of the mnemonic identifies the condition under which the jump is to be executed. If the condition holds, then the jump takes place; otherwise, the next instruction (the one following the conditional jump) is executed.

For example, the instruction

```
jz endWhile
```

means to jump to the statement with label endWhile if the zero flag ZF is set to 1; otherwise fall through to the next statement.

5.2 Conditional Jumps, Compare Instructions, and ...

Conditional jump instructions do not modify the flags; they only react to previously set flag values.

Suppose, for example, that the value in the EAX register is added to a sum representing an account balance, and three distinct treatments are needed, depending on whether the new balance is negative, zero, or positive. A pseudo code design for this could be

```
add value to balance;
if balance < 0 then
... { design for negative balance }
elseif balance = 0
then
... { design for zero balance }
else
... { design for positive balance }
end if;
```

Chapter 5 – Branching and Looping
5.2 Conditional Jumps, Compare Instructions, and …

The following 80×86 code fragment implements this design.

```
        add balance,eax        ;add value to balance
        jns elseIfZero         ;jump if balance not negative
        ...                    ;code for negative balance
        jmp endBalanceCheck

elseIfZero: jnz elsePos        ;jump if balance not zero
        ...                    ;code for zero balance
        jmp endBalanceCheck

elsePos: ...                   ;code for positive balance

endBalanceCheck:
```

5.2 Conditional Jumps, Compare Instructions, and ...

In the previous code, the label endBalanceCheck is on a line by itself. Technically this label will reference the address of whatever statement follows it, but it is far simpler to treat it as the part of the current design structure without worrying about what comes next.

If what comes after this structure is changed, the code for this structure can remain the same.

If the next statement requires another label, that is perfectly okay- multiple labels can reference the same spot in memory. Labels are not part of object code,

so extra labels do not add to the length of object code or to execution time.

When writing code to mirror a design, one often wants to use labels like if, then, else, and endif. Unfortunately, IF, ELSE, and ENDIF are MASM directives, so they cannot be used as labels.

In addition, IF1, IF2, and several other desirable labels are also reserved for use as directives.

One solution is to use long descriptive labels like elseIfZero in the above example.

Since no reserved word contains an underscore, another solution is to use labels like if_1 and endif_2 that parallel keywords in the original design.

The terms set a flag and reset a flag are often used to mean "give the value 1" to a flag and "give the value 0" to a flag, respectively. (Sometimes the word clear is used instead of reset.) As you have seen, many instructions set or reset flags. However, the cmp (compare) instructions are probably the most common way to establish flag values.

5.2 Conditional Jumps, Compare Instructions, and ...

The cmp (compare) instructions are probably the most common way to establish flag values.

Each cmp instruction compares two operands and sets or resets AF, CF, OF, PF, SF, and ZF. The *only* job of a cmp instruction is to fix flag values; this is not just a side effect of some other function. Each has the form

```
cmp       operand1,operand2
```

A cmp executes by calculating *operand1* minus *operand2*, exactly like a sub instruction; the value of the difference and what happens in doing the subtraction determines the flag settings. The flags that are of most interest in this book are CF, OF, SF, and ZF.

Chapter 5 – Branching and Looping

5.2 Conditional Jumps, Compare Instructions, and …

Here are a few examples showing how the flags are set or reset when some representative byte length numbers are compared.

	operand1	Operand2	Difference	flags				interpretation	
				CF	OF	SF	ZF	signed	Unsigned
1	3B	3B	00	0	0	0	1	op1=op2	op1=op2
2	3B	15	26	0	0	0	0	op1>op2	op1>op2
3	15	3B	DA	1	0	1	0	op1<op2	op1<op2
4	F9	F6	03	0	0	0	0	op1>op2	op1>op2
5	F6	F9	FD	1	0	1	0	op1<op2	op1<op2
6	15	F6	1F	1	0	0	0	op1>op2	op1<op2
7	F6	15	E1	0	0	1	0	op1<op2	op1>op2
8	68	A5	C3	1	1	1	0	op1>op2	op1<op2
9	A5	68	3D	0	1	0	0	op1<op2	op1>op2

5.2 Conditional Jumps, Compare Instructions, and ...

The cmp instructions are listed in Fig. 5.4. Looking back at Fig. 4.5, one sees that the entries in the various columns are almost all the same as for sub instructions.

Destination Operand	Source Operand	Clock Cycles 386	Clock Cycles 486	Clock Cycles Pentium	Number of Bytes	opcode
register 8	immediate 8	2	1	1	3	80
register 16	immediate 8	2	1	1	3	83
register 32	immediate 8	2	1	1	3	83
register 16	immediate 16	2	1	1	4	81
register 32	immediate 32	2	1	1	6	81
AL	immediate 8	2	1	1	2	3C
AX	immediate 16	2	1	1	3	3D
EAX	immediate 32	2	1	1	5	3D

5.2 Conditional Jumps, Compare Instructions, and …

Destination Operand	Source Operand	Clock Cycles			Number of Bytes	opcode
		386	486	Pentium		
memory byte	immediate 8	5	2	2	3+	80
Memory word	immediate 8	5	2	2	3+	83
Memory doubleword	immediate 8	5	2	2	3+	83
Memory word	immediate 16	5	2	2	4+	81
Memory doubleword	immediate 32	5	2	2	6+	81
register 8	register 8	2	1	1	2	38
register 16	register 16	2	1	1	2	3B
register 32	register 32	2	1	1	2	3B

5.2 Conditional Jumps, Compare Instructions, and ...

Destination Operand	Source Operand	Clock Cycles 386	486	Pentium	Number of Bytes	opcode
register 8	memory byte	6	2	2	2+	3A
register 16	Memory word	6	2	2	2+	3B
register 32	Memory doubleword	6	2	2	2+	3B
Memory word	immediate 16	5	2	2	2+	38
Memory doubleword	immediate 32	5	2	2	2+	39
register 8	register 8	5	2	2	2+	39

5.2 Conditional Jumps, Compare Instructions, and …

A few reminders are in order about immediate operands. These can be coded in your choice of bases or as characters. Assuming that *pattern* references a word in the data segment, each of the following is allowable.

```
cmp         eax,356
cmp         pattern,0d3a6h
cmp         bh,'$'
```

Note that an immediate operand must be the second operand. The instruction

```
cmp 100,total                 ;illegal
```

is not acceptable since the first operand is immediate.

Chapter 5 – Branching and Looping

5.2 Conditional Jumps, Compare Instructions, and …

Finally it is time to list the conditional jump instructions. Many of these have alternative mnemonics that generate exactly the same machine code.

Approptiate for use after comparison of unsigned operands

			opcode	
mnemonic	**description**	**Flags to jump**	**short**	**near**
Ja jnbe	Jump if above Jump if not below or equal	CF=0 and ZF=0	77	0F 87
Jae jnb	Jump if above or equal Jump if not below	CF=0	73	0F 83
Jb jnae	Jump if below Jump if not above or equal	CF=1	72	0F 82
Jbe jna	Jump if below or equal Jump if not below	CF=1 or ZF=1	76	0F 86

Chapter 5 – Branching and Looping

5.2 Conditional Jumps, Compare Instructions, and …

Appropriate for use after comparison of signed operands

mnemonic	description	Flags to jump	opcode short	opcode near
Jg jnle	Jump if greater Jump if not less or equal	SF=0F and ZF=0	7F	0F 8F
Jge jnl	Jump if greater or equal Jump if not less	SF=0F	7D	0F 8D
Jl jnge	Jump if less Jump if not grater or equal	SF≠0F	7C	0F 8C
Jle jng	Jump if less or equal Jump if not greater	SF≠0F or ZF=1	7E	0F 8E

Chapter 5 – Branching and Looping

5.2 Conditional Jumps, Compare Instructions, and …

mnemonic	description	Flags to jump	opcode short	opcode near
Je	Jump if equal	ZF=1	74	0F 84
Jz	Jump if zero			
Jne	Jump if not equal	ZF=0	75	0F 85
Jnz	Jump if not zero			
Js	Jump if sign	SF=1	78	0F 88
Jns	Jump if not sign	SF=0	79	0F 89
Jc	Jump if carry	CF=1	72	0F 82
Jnc	Jump if not carry	CF=0	73	0F 83
Jp	Jump if parity	PF=1	7A	0F 8A
Jpe	Jump if parity even			
Jnp	Jump if not parity	PF=0	78	0F 8B
Jpo	Jump if parity odd			
Jo	Jump if overflow	OF=1	70	OF 880
Jno	Jump if not overflow	OF=0	71	OF 81

Chapter 5 – Branching and Looping

5.2 Conditional Jumps, Compare Instructions, and …

Each conditional jump instruction takes a single clock cycle for execution.

No conditional jump instruction changes any flag value.

Each instruction has a short version and a near version.

The number of bytes and number of clock cycles for conditional jump instructions is summarized in Fig. 5.6.

	Clock Cycles			Number of bytes
	386	486	pentium	
Short conditional jump	7+,3	3,1	1	2
Near conditional jump	7+,3	3,1	1	6

One more pair of examples will illustrate the difference between the conditional jumps appropriate after comparison of signed and unsigned numbers.

Suppose a value is stored in EAX and some action needs to be taken when that value is larger than 100. If the value is unsigned, one might code

```
cmp     eax,100
ja      bigger
```

The jump would be chosen for any value bigger than 00000064_{16}, including values between 80000000_{16} and $FFFFFFFF_{16}$, which represent both large numbers and negative 2's complement numbers. If the value in EAX is interpreted as signed, then the instructions

```
cmp     eax,100
jg      bigger
```

are appropriate. The jump will only be taken for values between 00000064_{16} and $7FFFFFFF_{16}$, not for those bit patterns that represent negative 2's complement numbers.

Chapter 5 – Branching and Looping

5.2 Conditional Jumps, Compare Instructions, and ...

We now look at three examples showing implementation of if structures. The implementations are consistent with what a high-level language compiler would use. First consider the design

Suppose that *value* is stored in the EBX register and that *smallCount* and *largeCount* reference words in memory. The following 80×86 code implements this design.

```
if value < 10
then
    add 1 to smallCount;
else
    add 1 to largeCount;
end if;
```

```
             cmp   ebx, 10
             jnl   elseLarge
             inc   smallCount
             jmp   endValueCheck
elseLarge:   inc   largeCount
endValueCheck:
```

Chapter 5 – Branching and Looping

5.2 Conditional Jumps, Compare Instructions, and …

Now consider the design

```
if (total ≥ 100) or (count = 10)
then
    add value to total;
end if;
```

Assume that *total* and *value* reference doublewords in memory and that *count* is stored in the CX register. Here is assembly language code to implement this design.

```
                cmp total, 100
                jge addValue
                cmp cx, 10
                jne endAddCheck
addValue:       mov ebx, value
                add total, ebx
endAddCheck:
```

Finally consider the design

```
if (count > 0) and (ch = backspace)
then
    subtract 1 from count;
end if;
```

For this third example, assume that *count* is in the CX register, *ch* is in the AL register and that *backspace* has been equated to 08_{16}, the ASCII backspace character. This design can be implemented as follows.

```
        cmp count,0              ;count > 0 ?
        jng endCheckCh
        cmp al,backspace         ;ch a backspace?
        jne endCheckCh
        dec count                ;subtract 1 from count
endCheckCh:
```

Chapter 5 – Branching and Looping

5.3 Implementing Loops and Structures

Most programs contain loops. Commonly used loop structures include **while**, **until**, and **for** loops. This section describes how to implement all three of these structures in 80×86 assembly language.

A **while** loop can be indicated by the following pseudo code design.

```
while continuation condition loop
 ... { body of loop }
end while;
```

The ***continuation condition***, a Boolean expression, is checked first. If it is true, then the body of the loop is executed. The continuation condition is then checked again. Whenever the value of the Boolean expression is false, execution continues with the statement following end while.

Chapter 5 – Branching and Looping

5.3 Implementing Loops and Structures

An 80×86 implementation of a while loop follows a pattern much like this one.

```
while:          .               ;code to check Boolean expression
                .
                .
body:           .               ;loop body
                .
                .
        jmp     while           ;go check condition again
endWhile:
```

It often takes several statements to check the value of the Boolean expression. If it is determined that the value is false, then there will be a jump to *endWhile*. If it is determined that the continuation condition is true, then the code will either fall through to body or there will be a jump to its label.

Chapter 5 – Branching and Looping

5.3 Implementing Loops and Structures

For an example, suppose that the design

```
while (sum < 1000) loop
     ... {body of loop}
end while;
```

is to be coded in 80×86 assembly language. Assuming that *sum* references a doubleword in memory, one possible implementation is

```
whileSum:      cmp sum, 1000          ;sum < 1000?
               jnl endWhileSum        ;exit loop if not
                .                     ;body of loop
                .
                .
               jmp whileSum           ;go check condition again
endWhileSum:
```

Chapter 5 – Branching and Looping

5.3 Implementing Loops and Structures

For a short example showing a complete loop body, suppose that the integer base 2 logarithm of a positive number needs to be determined. The integer base 2 logarithm of a number is the largest integer *x* such that

$$2^x \leq \text{number}$$

The following design does the job.

```
x := 0;
twoToX := 1;
while twoToX ≤ number
        multiply  twoToX  by
    2;
        add 1 to x;
end while;
subtract 1 from x;
```

Chapter 5 – Branching and Looping

5.3 Implementing Loops and Structures

Assuming that *number* references a doubleword in memory, the following 80×86 code implements the design, using the EAX register for *twoToX* and the CX register for *x*.

```
            mov     cx, 0           ;x := 0
            mov     eax, 1          ;twoToX := 1
whileLE:    cmp     eax, number     ;twoToX <= number?
            jnle    endWhileLE      ;exit if not
body:       add     eax,eax         ;multiply twoToX by 2
            inc     cx              ;add 1 to x
            jmp     whileLE         ;go check condition again
 endWhileLE:
            dec     cx              ;subtract 1 from x
```

Often the continuation condition in a while is compound, having two parts connected by Boolean operators **and** or **or**. Both operands of an **and** must be true for a true conjunction. With an **or**, the only way the disjunction can be false is if both operands are false.

Chapter 5 – Branching and Looping
5.3 Implementing Loops and Structures

Changing a previous example to include a compound condition, suppose that the following design is to be coded.

```
while (sum < 1000) and (count ≤ 24) loop
    ... { body of loop }
end while;
```

Assuming that ***sum*** references a doubleword in memory and the value of ***count*** is in CX, an implementation is

```
whileSum:  cmp    sum,1000        ;sum < 1000?
           jnl    endWhileSum     ;exit if not
           cmp    cx,24           ;count <= 24
           jnle   endWhileSum     ;exit if not
            .                     ;body of loop
            .
            .
           jmp whileSum           ;go check condition again
endWhileSum:
```

Chapter 5 – Branching and Looping

5.3 Implementing Loops and Structures

Modifying the example another time, here is a design with an **or** instead of an **and**.

```
while (sum < 1000) or (flag = 1)
loop
    ... { body of loop }
end while;
```

This time, assume that *sum* is in the EAX register and that *flag* is a single byte in the DH register. Here is 80×86 code that implements the design.

```
whileSum:   cmp     eax,1000        ;sum < 1000?
            jl      body            ;execute body if so
            cmp     dh,1            ;flag = 1?
            jne     endWhileSum     ;exit if not
body:        .                      ;body of loop
             .
             .
            jmp whileSum            ;go check condition again
endWhileSum:
```

Chapter 5 – Branching and Looping

5.3 Implementing Loops and Structures

Sometimes processing in a loop is to continue while normal values are encountered and to terminate when some sentinel value is encountered. If data are being entered from the keyboard, this design can be written

```
get value from keyboard;
while (value is not sentinel) loop
    . . . {body of loop}
    get value from keyboard;
endwhile;
```

For a concrete example illustrating implementation of such a design, suppose that non-negative numbers entered at the keyboard are to be added, with any negative entry serving as a sentinel value. A design looks like

```
    sum := 0;
while  (number  keyed  is  not  sentinel)
loop
    add number to sum;
endwhile;
```

Chapter 5 – Branching and Looping
5.3 Implementing Loops and Structures

Assuming appropriate definitions in the data segment, the 80×86 code could be

```
            mov     ebx,0           ;sum := 0
whileNotNeg: output prompt          ;prompt for input
            input   number,10       ;get number from keyboard
            atod    number          ;convert to 2's complement
            js      endWhile        ;exit if negative
            add     ebx,eax         ;add number to sum
            jmp     whileNotNeg     ;go get next number
endWhile:
```

Recall that the **atod** macro affects the sign flag SF, setting it if the ASCII characters are converted to a negative number in the EAX register and clearing it otherwise.

Chapter 5 – Branching and Looping

5.3 Implementing Loops and Structures

The body of a **for** loop, a counter-controlled loop, is executed once for each value of a loop index (or counter) in a given range. A **for** loop can be described by the following pseudocode.

```
for index := initialValue to finalValue loop
    . . .  {loop of loop}
end for;
```

A for loop can easily be translated into a while structure.

```
index := initialValue;
while index ≤ finalValue loop
    ... { body of loop }
    add 1 to index;
end while;
```

Such a **while** is readily coded in 80×86 assembly language.

Chapter 5 – Branching and Looping

5.3 Implementing Loops and Structures

As an example, suppose that a collection of numbers needs to be added and no value is convenient as a sentinel. Then one might want to ask a user how many numbers are to be entered and loop for that many entries. The design looks like

```
prompt for tally of numbers;
input tally;
sum := 0
for count := 1 to tally loop
    prompt for number;
    input number;
    add number to sum;
end for;
```

Chapter 5 – Branching and Looping

5.3 Implementing Loops and Structures

Making straightforward assumptions about definitions in the data segment, here is an 80×86 implementation of the design.

```
            output   prompt1        ;prompt for tally
            input    value,20       ;get tally (ASCII)
            atoi     value          ;convert to 2's complement
            mov      tally,ax       ;store tally
            mov      edx,0          ;sum := 0
            mov      bx,1           ;count := 1

forCount:   cmp      bx,tally       ;count <= tally?
            jnle     endFor         ;exit if not
            output   prompt2        ;prompt for number
            input    value,20       ;get number (ASCII)
            atod     value          ;convert to 2's complement
            add      edx,eax        ;add number to sum
            inc      bx             ;add 1 to count
            jmp forCount            ;repeat
endFor:
```

Chapter 5 – Branching and Looping

5.3 Implementing Loops and Structures

You have already seen examples of **until** loops. In general, an **until** loop can be expressed as follows in pseudocode.

```
until termination condition loop
    . . . {body of loop}
end until;
```

The body of the loop is executed at least once; then the termination condition is checked. If it is false, then the body of the loop is executed again; if true, execution continues with the statement following **end until**.

An 80x86 implementation of an **until** loop usually looks like the following code fragment.

```
until:      .               ;start of loop body
            .               .
            .               .
endUntil:   .               ;code to check termination condition
```

Chapter 5 – Branching and Looping

5.3 Implementing Loops and Structures

If the code to check the termination condition determines that the value is *false*, then there will be a jump to *until*. If it is determined that the value is *true*, then the code will either fall through to *endUntil* or there will be a jump to that label.

The game program implemented in Fig. 5.8 contained two simple until loops. Here is an example with a compound terminating condition. Given the design

```
count := 0;
until (sum > 1000) or (count = 100) loop
    ... { body of loop }
    add 1 to count;
end until;
```

Chapter 5 – Branching and Looping

5.3 Implementing Loops and Structures

the following 80×86 code provides an implementation. Assume that *sum* references a word in the data segment and that *count* is stored in CX.

```
        mov     cx,0            ;count := 0
until:   .                      ;body of loop
         .
         .
        inc     cx              ;add 1 to count
        cmp     sum,1000        ;sum > 1000 ?
        jg      endUntil        ;exit if sum > 1000
        cmp     cx,100          ;count = 100 ?
        jne     until           ;continue if count not = 100
endUntil:
```

Chapter 5 – Branching and Looping
5.3 Implementing Loops and Structures

Other loop structures can also be coded in assembly language. The forever loop is frequently useful. As it appears in pseudocode, it almost always has an exit loop statement to transfer control to the end of the loop; this is often conditional—that is, in an if statement. Here is a fragment of a typical design.

```
     forever loop
          .
          .
          .
          if (response='s') or (response='S')
          then
                   exit loop;
          end if;
          .
          .
          .
End loop;
```

Chapter 5 – Branching and Looping

5.3 Implementing Loops and Structures

Assuming that the value of *response* is in the AL register, this can be implemented as follows in 80×86 assembly language.

```
forever:        .
                .
                .
                cmp     al,'s'          ;response = 's'?
                je      endLoop         ;exit loop if so
                cmp     al,'S'          ;response = 'S'?
                je      endLoop         ;exit loop if so
                .
                .
                .
                jmp     forever         ;repeat loop body
endLoop:
```

Chapter 5 – Branching and Looping

5.4 for Loops in Assembly Language

The 80×86 microprocessor has instructions that make coding certain **for** loops very easy. Consider the following two for loops, the first of which counts forward and the second of which counts backward.

```
for index := 1 to count loop
    . . . {body of loop}
end for;
```

and

```
for index := count downto 1 loop
    . . . {body of loop}
end for;
```

The body of each loop executes *count* times. Backward for loops are very easy to implement in 80×86 assembly language with the loop instruction.

Chapter 5 – Branching and Looping

5.4 for Loops in Assembly Language

The loop instruction has the format

```
loop        statementLabel
```

where *statementLabel* is the label of a statement that is a short displacement (128 bytes backward or 127 bytes forward) from the loop instruction.

The loop instruction causes the following actions to take place:

- ✓ **the value in ECX is decremented**
- ✓ **if the new value in ECX is zero, then execution continues with the statement following the loop instruction**
- ✓ **if the new value in ECX is nonzero, then a jump to the instruction at *statementLabel* takes place**

Chapter 5 – Branching and Looping

5.4 for Loops in Assembly Language

In addition to the loop instruction, there are two conditional loop instructions that are less frequently used. Features of all three instructions are summarized in Fig. 5.9. None of these instructions changes any flag.

Mnemonic	Clock Cycles			Number of bytes	Opcode
	386	486	pentium		
loop	11+	6/7	5/6	2	E2
Loope/loopz	11+	6/9	5/6	2	E1
Loopne/loopnz	11+	6/9	7/8	2	E0

Although the ECX register is a general register, it has a special place as a counter in the loop instruction and in several other instructions. No other register can be substituted for ECX in these instructions.

Chapter 5 – Branching and Looping
5.4 for Loops in Assembly Language

The backward for loop structure

```
for count := 20 downto 1 loop
    . . . {body of loop}
end for;
```

can be coded as follows in 80×86 assembly language.

```
          mov     ecx,20        ;number of iterations
forCount:  .                    ;body of loop
           .
           .
          loop    forCount      ;repeat body 20 times
```

Chapter 5 – Branching and Looping

5.4 for Loops in Assembly Language

This is safe code only if the value stored at *number* is not zero. If it is zero, then the loop body is executed, the zero value is decremented to FFFFFFFF, the loop body is executed again, the value FFFFFFFF is decremented to FFFFFFFE, and so forth. The body of the loop is executed 4,294,967,296 times before the value in ECX gets back down to zero! To avoid this problem, one could code

```
            mov     ecx,number        ;number of iterations
            cmp     ecx,0             ;number = 0 ?
            je      endFor            ;skip loop if number = 0
forIndex:    .                        ;body of loop
             .
             .
            loop forIndex             ;repeat body number times
endFor:
```

If *number* is a signed value and might be negative, then

```
            jle     endFor            ;skip loop if number <= 0
```

Chapter 5 – Branching and Looping

5.4 for Loops in Assembly Language

There is another way to guard a for loop so that it is not executed when the value in ECX is zero. The 80×86 instruction set has a jecxz conditional jump instruction that jumps to its destination if the value in the ECX register is zero. Using the jecxz instruction, the example above can be coded as

```
          mov     ecx,number      ;number of iterations
          jecxz   endFor          ;skip loop if number = 0
forIndex:  .                      ;body of loop
            .
            .
          loop    forIndex        ;repeat body number times
endFor:
```

There is also a jcxz instruction that checks the CX register rather than the ECX register. Like the other conditional jump instructions, jcxz/jecxz affects no flag value.

Chapter 5 – Branching and Looping

5.4 for Loops in Assembly Language

The jecxz instruction can be used to code a backward for loop when the loop body is longer than 127 bytes, too large for the loop instruction's single-byte displacement. For example, the structure

```
for  counter := 50 downto 1 loop
    . . . {body of loop}
end for;
```

could be coded as

```
            mov     ecx,50          ;number of iterations
forCounter: .                       ;body of loop
            .
            .
            dec     ecx             ;decrement loop counter
            jecxz   endFor          ;exit if counter = 0
            jmp     forCounter      ;otherwise repeat body
endFor:
```

Chapter 5 – Branching and Looping

5.4 for Loops in Assembly Language

It is often convenient to use a loop statement to implement a **for** loop, even when the loop index increases and must be used within the body of the loop. The loop statement uses ECX to control the number of iterations, while a separate counter serves as the loop index.For example, to implement the **for** loop

```
for index := 1 to 50 loop
    ...{loop body using index}
end for;
```

the EBX register might be used to store *index* counting from 1 to 50 while the ECX register counts down from 50 to 1.

```
          mov     ebx,1        ;index := 1
          mov     ecx,50       ;number of iterations for loop
forNbr:    .
           .                   ;use value in EBX for index
           .
          inc     ebx          ;add 1 to index
          loop    forNbr       ;repeat
```

Chapter 5 – Branching and Looping
5.4 for Loops in Assembly Language

Figure 5.9 listed two variants of the loop instruction, loopz/loope and loopnz/loopne. Each of these work like loop, decrementing the counter in ECX. However, each examines the value of the zero flag ZF as well as the new value in the ECX register to decide whether or not to jump to the destination location. The loopz/loope instruction jumps if the new value in ECX is nonzero and the zero flag is set (ZF=1). The loopnz/loopne instruction jumps if the new value in ECX is nonzero and the zero flag is clear (ZF=0).

The loopz and loopnz instructions are useful in special circumstances. Some programming languages allow loop structures such as

```
for year := 10 downto 1 until balance=0 loop
    . . . {body of loop}
end for;
```

Chapter 5 – Branching and Looping

5.4 for Loops in Assembly Language

This confusing structure means to terminate loop execution using whichever loop control is satisfied first. That is, the body of the loop is executed 10 times (for year = 10, 9,…,1) unless the condition *balance = 0* is true at the bottom of some execution of the loop body, in which case the loop terminates with fewer than 10 iterations. If the value of balance is in the EBX register, the following 80×86 code could be used.

```
         mov     ecx,10   ;maximum number of iterations
forYear:  .               ;body of loop
          .
          .
         cmp     ebx,0    ;balance = 0 ?
         loopne  forYear  ;repeat 10 times if balance not 0
```

Chapter 5 – Branching and Looping

5.5 Arrays

Programs frequently use arrays to store collections of data values. Loops are commonly used to manipulate the data in arrays. This section shows one way to access 1-dimensional arrays in 80×86 assembly language.

This section contains a complete program to implement the design below. The program

- ✓ **first accepts a collection of positive numbers from the keyboard,**
- ✓ **counting them and**
- ✓ **storing them in an array.**
- ✓ **It then calculates the average of the numbers by going back through the numbers stored in the array, accumulating the total in *sum*.**
- ✓ **Finally the numbers in the array are scanned again, and this time the numbers larger than the average are displayed.**

The first two loops could be combined, of course, with the sum being accumulated as the numbers are keyed in.

Chapter 5 – Branching and Looping
5.5 Arrays

```
nbrElts := 0;                    {input numbers into array}
get address of first item of array;

while (number from keyboard > 0) loop
    convert number to 2's complement;
    store number at address in array;
    add 1 to nbrElts;
    get address of next item of array;
end while;

sum := 0;                        {find sum and average}
get address of first item of array;

for count := nbrElts downto 1 loop
    add doubleword at address in array to sum;
    get address of next item of array;
end for;
average := sum/nbrElts;
display average;
```

Chapter 5 – Branching and Looping

5.5 Arrays

```
average := sum/nbrElts;
display average;
get address of first item of array;    {list big numbers}
for count := nbrElts downto 1 loop
    if doubleword of array > average
    then
        convert doubleword to ASCII;
        display value;
    end if;
    get address of next item of array;
end for;
```

Chapter 5 – Branching and Looping

5.5 Arrays

This design contains the curious instructions "get address of first item of array" and "get address of next item of array." These reflect the particular assembly language implementation, one which works well if the task at hand involves moving sequentially through an array. The 80×86 feature which makes this possible is register indirect addressing, first discussed in Chapter 3. The example will use the EBX register to contain the address of the word currently being accessed; then [ebx] references the doubleword at the address in the EBX register rather than the doubleword in the register itself. In the 80×86 architecture any of the general registers EAX, EBX, ECX, and EDX or the index registers EDI and ESI are appropriate for use as a "pointer." The ESI and EDI registers are often reserved for use with strings, which are usually arrays of characters. String operations are covered in Chapter 7. The program listing appears in Fig. 5.10.

Chapter 5 – Branching and Looping

5.6 Pipelining

Chapter 2 discussed the central processing unit's basic operation cycle:

- ✓ **fetch an instruction from memory**
- ✓ **decode the instruction**
- ✓ **execute the instruction**

A CPU must have circuitry to perform each of these functions. One of the things that computer designers have done to speed up CPU operation is to design CPUs with stages that can carry out these (and other) operations almost independently.

This sort of design is called a pipeline.

5.6 Pipelining

If the pipeline is kept full, the resulting throughput of the CPU is three times faster than if it had to finish the complete fetch-decode-execute process for each instruction before proceeding to the next one.

Figure 5.11 illustrates the operation of a pipeline. The instructions being processed are shown as horizontal strips of three boxes labeled with 1, 2, and 3 to indicate stages. The horizontal axis shows time. You can see that at any given time parts of three instructions are being executed.

CPU Stage	Instruction being processed										
1	1	2	3	4	5	6	7	8	9	10	11
2		1	2	3	4	5	6	7	8	9	10
3			1	2	3	4	5	6	7	8	9
Time Interval	1	2	3	4	5	6	7	8	9	10	11

Chapter 5 – Branching and Looping
5.6 Pipelining

A pipelined CPU is not as simple as illustrated above. One problem may occur if, say, stage 2 needs to compute an address based on the contents of a register modified by stage 3 of the previous instruction; the register might not yet contain the correct address. A CPU can be designed to avoid such problems, usually at the cost of a "hole" in the pipeline.

A more serious problem occurs when the CPU executes a conditional jump instruction. With a conditional jump the CPU cannot tell which of two possible sequences of instructions will be executed next until the condition itself is evaluated by the last stage. Earlier stages may be working on one instruction stream, only to be forced to discard all this work and refill the pipeline from the beginning with instructions from the alternative stream.

5.6 Chapter Summary

This chapter introduced 80×86 instructions that can be used to implement many high-level design or language features including **if** statements, various loops structures, and arrays.

The jmp instruction unconditionally transfers control to a destination statement. It has several versions, including one that jumps to a short destination 128 bytes before or 127 bytes after the jmp and one that jumps to a near destination a 32-bit displacement away. The jmp instruction is used in implementing various loop structures, typically transferring control back to the beginning of the loop, and in the **if-then-else** structure at the end of the "then code" to transfer control to **endif** so that the **else** code is not also executed. A jmp statement corresponds directly to the **goto** statement that is available in most high-level languages.

Conditional jump instructions examine the settings of one or more flags in the flag register and jump to a destination statement or fall through to the next instruction depending on the flag values. Conditional jump instructions have short and near displacement versions. There is a large collection of conditional jump instructions. They are used in **if** statements and loops, often in combination with compare instructions, to check Boolean conditions.

5.6 Chapter Summary

The cmp (compare) instructions have the sole purpose of setting or resetting flags in the EFLAGS register. Each compares two operands and assigns flag values. The comparison is done by subtracting the second operand from the first. The difference is not retained as it is with a sub instruction. Compare instructions often precede conditional jump instructions.

Loop structures like **while**, **until**, and **for** loops can be implemented using compare, jump, and conditional jump instructions. The loop instruction provides another way to implement many **for** loops. To use the **loop** instruction, a counter is placed in the ECX register prior to the start of the loop. The loop instruction itself is at the bottom of the loop body; it decrements the value in ECX and transfers control to a destination (normally the first statement of the body) if the new value in ECX is not zero. This results in the body of the loop being executed the number of times originally placed in the ECX register. The conditional jump jecxz instruction can be used to guard against executing such a loop when the initial counter value is zero.

5.6 Chapter Summary

Storage for an array can be reserved using the DUP directive in the data segment of a program. The elements of an array can be sequentially accessed by putting the address of the first element of the array in a register and adding the size of an array element repeatedly to get to the next element. The current element is referenced using register indirect addressing. The lea (load effective address) instruction is commonly used to load the initial address of the array.

Pipelining is done by a CPU with multiple stages that work on more than one instruction at a time, doing such tasks as fetching one, while decoding another, while executing a third. This can greatly speed up CPU operation.