

Chapter 3

Assembly Language Issues

3.1 Overview

This chapter reviews some preliminary assembly programming issues beginning with the concepts of Real and Protected mode that define the space addressability for 16-bit and 32-bit Intel architectures. Important issues from the programmer perspective such as organization of *nasm* programs, assembly, link, and compilation commands, and control structures are introduced. Program control structures are essential in any language to determine the dynamic flow of programs in execution. Therefore, the first set of *nasm* instructions covering comparison and branch instructions are discussed. The chapter ends showing the mapping of common HLL structures into possible assembly-based implementation.

3.2 Real Mode

In real mode the CPU addresses only one megabyte of memory. The organization of memory is segmented in maximum sizes of 64 Kbytes which can be directly addressable with 16 bits. The 8086/88 machines have an address bus with 20 bits and therefore capable of directly addressing $2^{20} \approx 1$ Megabyte of memory. To be able to address any location in memory in real mode the system uses two 16-bit entities referred to as segment and offset in such a way that the segment part will contain the first 16 bits of the base address with an offset of zero. To obtain the physical address both parts are added after shifting the segment part to the left four

places, that is:

$$\text{Physical address} = 2^4 \times \text{segment} + \text{offset}$$

Using hex digits, an alignment of segment and offset values is illustrated in Fig. 3.1. Assume for example that the segment value is 0928h and the offset value is 0022h then the physical address is then obtained as follows: 92800h + 0022 = 092A2h, as shown in Fig. 3.1

segment:	0	9	2	8	
offset:		0	0	2	2
Physical address:	0	9	2	A	2

Figure 3.1: Alignment of segment and offset

In a *segmented model* the segment and the offset part of a physical address are associated with segment registers and pointer/index registers, respectively. The combination of a segment register and an appropriate 16-bit register are used as pointers to the physical address. Typical *segment:offset* combinations are the following:

CS:IP
 SS:SP
 SS:BP
 DS:SI
 ES:DI

Fig. 3.2 illustrates the use of such pair registers are used as 'pointers'. It is easy to see the correspondence with the way an application is organized to run in a real mode segmented environment. The code segment CS register and the offset IP register will always be used to point to the code section in memory that corresponds to the code section in an application. Likewise the ES and DS segment registers in pair with SI, DI, registers will always point to the section in memory where data will reside and which corresponds to the data section in an application where all variables are declared. Similarly, the SS segment combined with two offset registers, the SP and BP will be used to point to the stack section in memory; an application might contain a stack section where the size of the stack is declared.

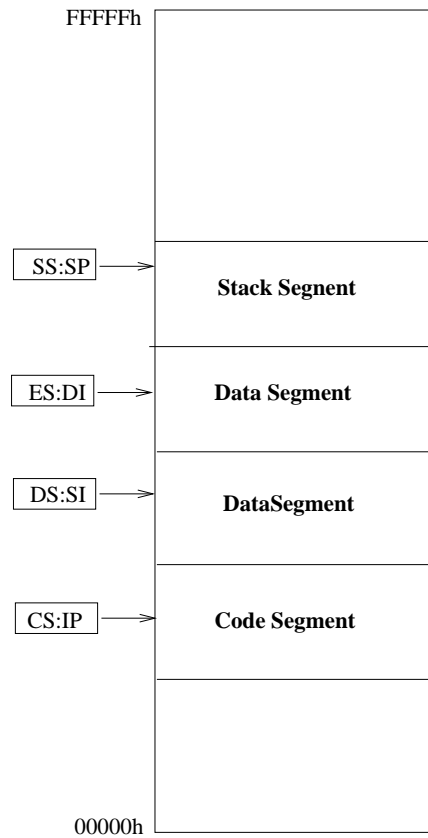


Figure 3.2: Placement of memory segments

In contrast to the segmented model, the *flat model* does not divide memory into sections. Code and data are allocated within a single 64K block of memory under a real mode environment. Segment registers are set to point to the beginning of the 64K block of memory and since offset registers can address a space of 64K, these are used to address any byte within the block simplifying the addressing scheme. In a protected flat model a program addresses memory within a single block as large as 4 Gbytes.

3.3 Protected Mode

As the complexity and size of applications increased the maximum limit of 64Kbytes in a real-mode environment was not acceptable. In real mode once a program is loaded segments remained at fixed positions in memory. The development of *virtual memory* gave each applications a "limitless" access to physical memory. In practice the system is used by several applications in a multi-tasking manner that

requires taking turns on the use of the CPU as well as main memory. Protected mode was developed to respond to additional memory requirements and to support a multitasking environment. In the 80286's 16-bit protected mode it was possible to move segments between hard disk storage and main memory as needed. However, the entire segment was transferred in and out of physical memory. The 80386 introduced the 32-bit protected mode that expanded offsets to be 32 bits. Therefore, segments can be up to 4 Gbytes in size. However, instead of segments, memory was divided into *pages* of 4K bytes each; this made possible that while a program was in execution, only sections of a segment were moved between main memory and disk.

In a protected mode each segment is assigned an entry in a *descriptor table*. This entry contains information regarding the size of the segment, its current location, and access rights. Segment registers can be used to access a descriptor table; a 13-bit *selector* field contains a pointer to the descriptor as shown in Fig. 3.3. The 2-bit field indicates the requested privilege level (RPL) and the TI field indicates if the descriptor table is global ($TI = 0$) or local ($TI = 1$). While global descriptors contain segment definitions that apply to all programs, local descriptor tables are unique to each program. Consequently each program has a total of 16,384 descriptors available at any time. Fig. 3.5 shows the descriptor formats for both the 80286 and the 80386 (and up). Recall that the 80286 can address up to 2^{24} bytes; therefore, the base address specified in the descriptor requires 24 bits with the remaining 8 bits set to 0. This indicates that a segment can be placed anywhere in memory within the range from 1 to 16M bytes. The limit of the segment is indicated with a 16-bit field denoting that segments vary from 1 to 64K bytes in size. This is not the case for the 80386 which addresses segments of size up to 4 Gbytes (2^{32}). Therefore, the base bits indicate that a segment can be placed anywhere in this range. The limit field is 20 bits in length indicating the the size of the segment varies from 1 to 1 Megabyte or from 4K bytes to 4G bytes in length depending on the value of the granularity selected.

To illustrate the use of addressing segments in protected mode assume the segment register DS contains the value 0010h which points to a descriptor as shown in Fig. 3.5.

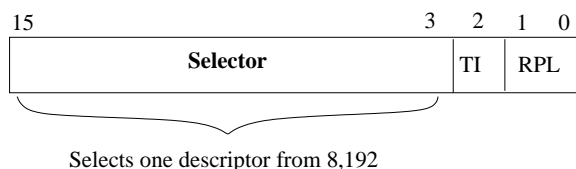


Figure 3.3: Contents of a segment register in a protected-mode operation

The formats of the descriptors used for the 80286 and 80386 — Pentium pro-

processors are shown in Fig. 3.4.

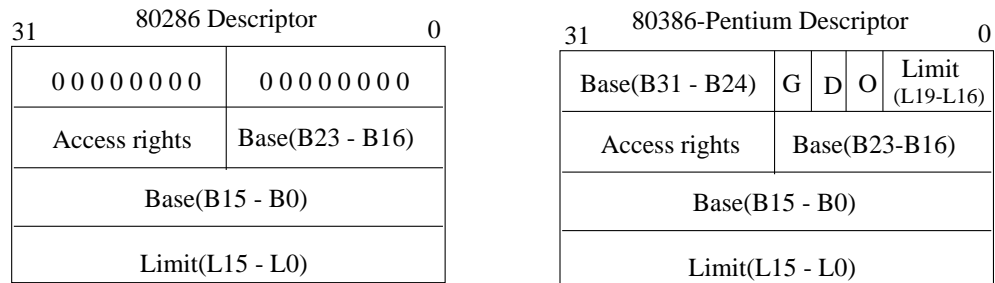


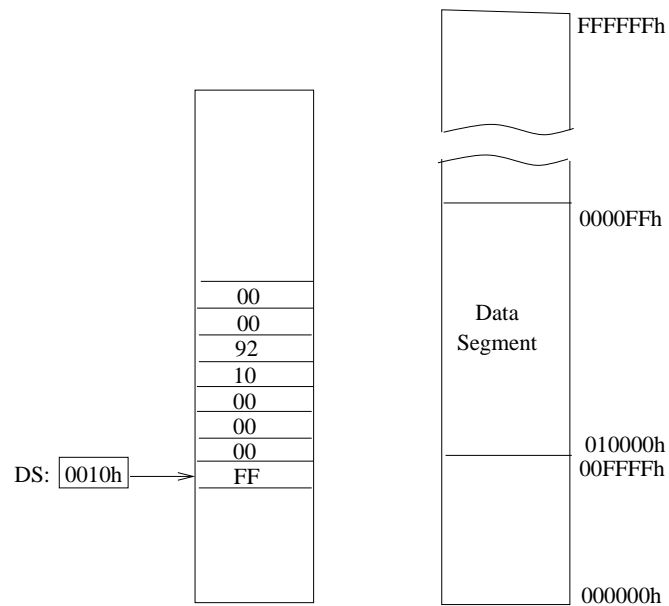
Figure 3.4: Descriptor formats for the 80286/386/486/Pentium processors.

3.4 Assembly Programs

A program running under DOS can be divided into three primary sections identified in TASM by the directives: *.stack*, *.data*, and *.code*. Each program section corresponds to the segment in memory to which it will be allocated. In *nasm* assembly programs are also divided into *sections* or *segments*. Consider for example the *nasm* code that implements a typical *hello, world* program using the segment-based model approach. Comments are preceded with a “;”. An initial block of commented lines are shown to provide information on how to produce the object code, link it and generate and executable (binary) version of the program. The assembly command shown is intended to create an intermediate file with the extension *.obj*. To produce an executable program, i.e, a program with an extension *.exe*, all object files created must be first linked. A free 16-bit linker *alink* is used in this case. The result of the linking step is an executable file. This is a 16-bit application and therefore, the *[BITS 16]* directive is used to instruct the assembler to generate a binary code for execution in a 16-bit real mode:

```
-----
; This program illustrates the segmented memory model
; assemble using the 16-bit nasm assembler:
;           nasm16 -f obj hello.asm -o hello.obj
; this will produce: hello.obj
; to link do: alink hello
; it will produce: hello.exe
; note: an entry point (...start) must be specified.
```

[BITS 16]



a) The DS register selects a segment to access locations from 010000h to 0000FFh

00	00
92	10
00	00
00	FF

b) 80286 Descriptor

00	?0
92	10
00	00
00	FF

c) Pentium Descriptor

Figure 3.5: Use of segment registers and descriptor table to access physical memory in the 80286 and Pentium processors.

```

SEGMENT mystack stack
    resb 100h
stacktop:

SEGMENT data
    msg      db      "Hello, world!", 13, 10, '$'

SEGMENT code
..start:
    mov ax, data
    mov ds, ax
    mov ax, mystack

```

```

mov ss, ax
mov sp, stacktop

mov ah, 9
mov dx, msg
int 21H

mov ax, 04C00H
int 21H

```

The directive *segment mystack stack* directs *nasm* to create a structure with a name *mystack* of type *stack*. The size of the stack is indicated with the statement *resb 100h* that allocates 256 bytes of RAM space. The declaration *segment data*, indicates the section of the program where *nasm* expects to find all the definitions of data that require initialization. An additional section referred to as the *.bss* segment can also be used to declare and allocate non-initialized data. The declaration *segment code* is the section of the program where *nasm* expects to find the actual code.

Since memory is addressed through the segment structure, the corresponding segment registers are initialized. Thus, *ds* is initialized to contain a 16-bit value used to address the data segment. Likewise, the *ss* register is initialized to *mystack*. Also the stack pointer register (*sp*) is initialized to the address (offset) value stored at *stacktop*. Recall that the pair *ss:sp* will always point to the top of the stack. Note that the program provides an entry point labeled *..start* to identify the initial module where the execution begins; the presence of this label is helpful for multi-module applications and the label *..start* will simply identify the first module.

Within the data section of the program a string variable *msg* of type *db* (define byte) is initialized. This string is *\$*-terminated to be used by a DOS interrupt call *int 21h*. This call requires a function code to be specified in the 8-bit register *ah*, and the address of the string to be placed in the 16-bit register *dx*. Note that the string contains the character codes 13 and 10 which correspond to the carriage return and new line *ascii* codes, respectively. The program terminates with the execution of another *int 21h* interrupt call; in this case, the call requires a function code 4Ch in *ah* and a value 0 in *al*. The purpose of this call is to implement a return to DOS.

The example shown in the following lines describe the implementation of the *hello, world* program using a flat model approach. Note that there is no need to initialize the data segment in order to access the string to display. The directive *ORG 0100h* sets the *origin* address where code execution begins. The implication of this directive in a flat model is that when the program is loaded for execution, the *ip* register will contain the value 0100h to point to the first instruction of the

program. A *.com* program is a binary program directly generated by *nasm* without the need of a linking step:

```

-----
; this program uses a flat memory model
; it must be assemble using the 16-bit nasm assembler
; to assemble do: nasm -f bin first.asm -o first.com
; this will produce: first.com
;

[BITS 16]          ;alternatively USE16 can be used
ORG 0100h          ;DOS will place the program at this address
                  ;for execution

SEGMENT .text

hello:
    mov  ah, 9
    mov  dx, msg
    int  21H

    mov  ax, 04C00H
    int  21H

SEGMENT .data      ;SEGMENT is equivalent to SECTION

    msg      db      "Hello, world!", 13, 10, '$'
-----

```

Instructions for Intel microprocessors have the following format:

[label:][mnemonic][operands][;comments]

The *label* is an identifier followed by a colon; it specifies an address where the main procedure begins or simply identifies the target address of a jump instruction. The *mnemonic* field specifies a reserved name for instruction opcodes used by the CPU for execution. The *operands* field specifies up to three operands required by the instruction. Most instructions operate on two operands specified as: *destination*, *source*. The results of the operation will overwrite the current contents of the destination operand. The *comments* field is preceded by a ";" and the comments are ignored by the assembler.

3.5 Assembly Commands

To assemble code using *nasm* from the command line type:

```
nasm -f object-format myprogram.asm
```

The *object-format* is one of the following: *bin*, *coff*, *elf*, *obj* or *win32*, depending of the expected output and the available compiler. Nasm can generate *.COM* files directly using the 16-bit assembler with the following command:

```
nasm16 -f bin myprogram.asm -o myprogram.com
```

Object files can be created using the command:

```
nasm16 -f obj myprogram.asm -o myprogram.obj
```

Object files for 16-bit applications, can be linked using a 16-bit linker such as *ALINK* to generate executable files with the extension *.exe*. The following command:

```
alink myprogram
```

will take the object file *myprogram.obj* and generate *myprogram.exe*. Alink was created by Anthony Williams and it is available at <http://alink.home.dhs.org>. In general to assemble *n* different object files *prog1*, *prog2*, ..., *progn*, then the linker is used as follows:

```
alink prog1 prog2 ... progn
```

this command will yield an executable file: *prog1.exe*; the module *prog1.asm* is identified as the *main* or initial module as is the only module with an entry point *..start* to indicate to the linker where code execution is to begin. The remaining modules are external and do not require additional entry points.

To compile 32-bit applications using *djgpp* use the command:

```
nasm32 -f coff myprogram.asm
```

which it will create the object file *myprogram.o*. Object files can be created from “C” source files as follows:

```
gcc -c acprogram.c
```

If both object files *acprogram.o* and *myprogram.o* need to be linked, the following command line is used to produce an executable file *myexe.exe*:

```
gcc -o myexec acprogram.o myprogram.o
```

Public and External Declarations — An *external* declaration causes the linker to look for the declared label in another module. The label corresponds to a global variable or the name of a procedure. The declaration:

```
EXTERN ClearW
```

tells the assembler that the label *ClearW* is not found in the current module and therefore it will be resolved later during the linking process. However, to make a procedure or a variable externally accessible it must first be made public. The declaration:

```
GLOBAL ClearW
```

will make the label *ClearW* public and therefore, accessible by any other external module.

3.6 Control Structures

Control structures are an important element in the code implementation of any application. Any language must include support to control the flow of a program in execution in response to the changing state of the register flag.

3.6.1 Comparisons and Flags

The flow of the program changes according to the state of certain flags in the register flag. The program must check for these flags and decide whether a jump out of the current flow should be taken. Many instructions change the status of the program during execution by changing setting or resetting the flag bits in the flag register. Sometimes the execution of a comparison instruction is necessary to set a particular flag to a state that reflects the result of the comparison. Because of this, any assembly language provides a comparison instruction in its instruction set. For the Intel processors the comparison instruction is used according to the following format:

cmp reg/mem, reg/mem/constant

The *cmp* instruction performs the subtraction operation *destination* - *source* but leaves *destination* unchanged. The operands are located in registers or in memory indicated by the notation *reg/mem* in both destination and source fields. The instruction accepts only one reference to memory, that is, only one operand can be fetched from or stored in memory. An immediate operand (a constant) can only be specified as a source operand. The flags that may be affected by the subtraction operation are the following: the overflow flag (O) if overflow occurs, the zero flag (Z) is set if the result is zero, the sign flag (S) is set if the result is negative, the carry flag (C) is set if there is a carry (the carry flag is set if an overflow occurs after an un-signed operation).

3.6.2 Branch Instructions

After fetching an instruction, the pointer *cs:eip* is updated to point to the first byte of the next instruction. However, if the current instruction is a branch instruction, then the pointer *cs:eip* could be updated once again to point to the target address. *Unconditional* branch instructions always update *cs:eip*. *Conditional* branch instructions will update *cs:eip* according to the status of some flag.

Unconditional jumps. Unconditional jumps are implemented by the *jmp* instruction which it is used with the following formats:

jmp	short	label
jmp	near	label
jmp	far	label
jmp	word	label
jmp	dword	label
jmp	reg/mem	

The *short* jump specifies a target address located anywhere from -128 to 127 bytes with respect to the current contents of *eip*. Therefore, a short jump will update the *eip* by adding or subtracting an 8-bit value to calculate the target address specified in the instruction. The *near* type is the default jump within a segment and updates only the *eip* register. The 386 and up processors support a 2-byte jump and a 4-byte jump. In protected mode the 4-byte jump is the default jump and a 2-byte jump is specified with the *word* term before the label. An explicit specification of a 4-byte jump is common with the *dword* term before the label. Another alternative specifies a 16-bit or a 32-bit register or memory location to fetch the target address directly or indirectly. With *far* jumps the flow of the program branches to a different code

segment; in this case both *cs* and *eip* are updated. In protected mode this is a rare type of jump.

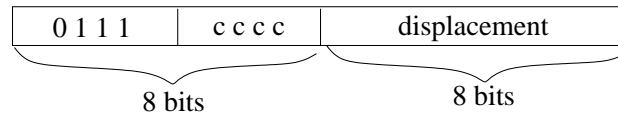
Conditional jumps. Table 3.1 shows three groups of conditional jumps. Group 1 deals with unsigned integers such as addresses and characters. In group 3 jumps are decided according to the state of an explicit flag; the description of each instruction in these two groups is given in terms of the results of a *cmp* instruction. Group 2 deals with signed integers; the flag involved is indicated in the description column. Table 3.1 is not an exhaustive list of conditional jump instructions. For each conditional jump of the form "jcc" Intel instruction sets also includes the corresponding complemented conditional jump instruction "jnc".

Table 3.1: Classification of conditional jump instructions

Mnemonic	Description	Flags
<i>group 1</i>		
ja	jump if above: dest. > source	C = 0 and Z = 0
jae	jump if above or equal: if dest. \nless source	F = 0 and Z = 0
jb	jump if below: dest. < source	C = 1
jbe	jump if below or equal: dest. \leq source	C=1 or Z=1
<i>group 2</i>		
jg	jump if greater: dest. > source	S=0 and Z=0
jge	jump if greater than or equal: dest. \leq source	S = O
jl	jump if less: dest. < source	S <> O
jle	jump if less than or equal: dest. \leq source	Z=1 or S <> O
js	jump if signed: dest. is negative	S = 1
jns	jump if not signed: dest. is positive	S = 0
jo	jump if overflow	O = 1
jno	jump if no overflow	O = 0
<i>group 3</i>		
jz	jump if zero	Z = 1
jnz	jump if not zero	Z = 0
jc	jump if carry	C = 1
jnc	jump if no carry	C = 0
jcxz(jecxz)	jump if cx (ecx) is zero	cx(ecx) = 0
jp	jump if parity even	P = 1
jnp	jump if parity odd	P = 0

Jumps out of range. For real-mode applications conditional jumps are limited to 8-bit jumps, i.e., within a range of -128 and 127 bytes. Conditional jumps are two-byte instructions where the highest order byte contains the opcode and the lowest order byte contains a displacement. The CPU utilizes the displacement value to

calculate the target address relative to the contents of the *ip* register. The following is the format of a 2-byte conditional jump instruction:

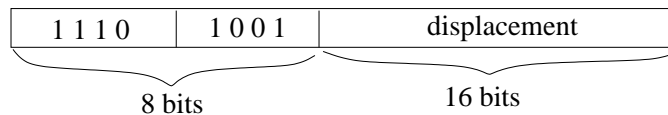


The highest order four bits give an indication to the CPU that the instruction is a conditional jump and takes the next four bits *cccc* as the code for the condition to check. For example, the entire 8-bit opcode for *jle* is 01111110. Upon execution the CPU updates the *ip* register as follows:

1. Flags are checked and if the condition holds the CPU sign extends the 8-bit displacement to a 16-bit value that is added to the current contents of the *ip* register,
2. The CPU fetches the next instruction using the updated value of the *ip* register.

If the assembler anticipates that the jump is out of range, reports an error and exits. The user can resolve this problem by combining a short jump of a conditional jump with a 16-bit unconditional near jump.

In real mode, near unconditional jumps use a 16-bit displacement to allow a range of jumps within $\pm 32K$ bytes. Near unconditional jumps have the following form:



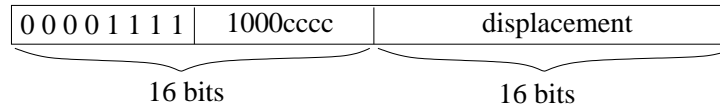
Therefore, in general terms any conditional jump *jcc label* that results in a "jump-out-of range" can be replaced by the following sequence:

```

...
jnc new_label
jmp label
new_label:
...
```

where *jnc* is a conditional jump with the original condition complemented.

For 32-bit applications, 80386 processors and up support a four-byte conditional instruction with the following format:



For example the entire 16-bit op-code for the *jle* instruction is: 00001111 10001110.

3.6.3 The *loop* Instruction

The *loop* instruction causes the repeated execution of a set of instructions within a loop. As long as the contents of the *ecx* register are greater than zero an iteration is executed. Therefore, it is important to always initialize *ecx* before the loop starts. Other variations of the *loop* instruction involve the zero flag as commented in the following possible formats:

```

loop    label    ; loop to label if  $CX > 0$ 
loopz   label    ; loop if  $CX > 0$  and  $Z = 1$ 
loope   label    ; loop if  $CX > 0$  and  $Z = 1$ 
loopne  label    ; loop if  $CX > 0$  and  $Z = 0$ 
loopnz  label    ; loop if  $CX > 0$  and  $Z = 0$ 

```

3.6.4 HLL Control Structures

The inter-relation between the state of a program (captured by the flag register) and the use of jump instructions to change the flow of execution makes high-level language control structures easy to implement. In this section we illustrate only few of the structures available in high-level languages and their mapping, in general terms, into assembly blocks.

IF statements. The following pseudo-code illustrates the use of *IF* statements:

```

if (op1  $\odot$  op2) then
    ...
    HLL statements
    ...
endif

```

To map HLL constructs into an assembly block the notation *jcc* and *jncc* used before, refers to the list of jump instructions in 3.1. The following assembly structure implements the HLL *IF* statement

```

    cmp    op1, op2
    jncc   endif          ; code is not executed if cc is not met
    ...
    assembly code
    ...
endif:   ...

```

IF-then-ELSE statements

These structures are described by the following pseudo-code:

```

if (op1  $\odot$  op2) then
    ...
    HLL block 1
    ...
else
    ...
    HLL block 2
    ...
endif

```

the following sequence illustrates a possible implementation in assembly:

```

    cmp    op1, op2
    jncc   else          ; code is not executed if cc is not met
    ...
    assembly code for block 1
    ...
else:
    ...
    assembly code for block 2
    ...
endif:

```

While loops. These are very common structures where the condition is first tested before another iteration is executed. The following pseudo-code illustrates a typical *while* loop structure:

```

while (op1  $\odot$  op2) then
    ...
    HLL statements
    ...
endwhile

```

This structure could be mapped into the following assembly sequence:

```

while:
    cmp op1, op2
    jncc endwhile    ; code is not executed if cc is not met
    ...
    assembly code
    ...
    jmp while
endwhile:            ...

```

Do while loops. In these loop structures the condition is tested after each iteration. The pseudo-code of a typical loop looks as follows:

```

do
    ...
    HLL statements
    ...
while (op1  $\odot$  op2)

```

A possible implementation is given below:

```

do:
    ...
    assembly code
    ...
    cmp op1, op2
    jcc do           ; a new iteration is executed if cc is true

```