

Chapter 4

Data Movement and Flow Control Instructions

4.1 Overview

Data movement instructions refer to the subset of instructions dedicated to the transfer of data between 1) memory buffers, 2) between registers, and 3) between registers and memory locations. This subset of instructions include those that transfer data involving the stack structure. This chapter discusses the *data definitions* provided by *nasm* and the syntax of data movement instructions with code illustrations. 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 assembly implementation of HLL structures described in the form of pseudo-code or flowcharts are discussed as immediate examples of the use of flow control instructions. A set of important instructions are the string processing instructions provided to move data between memory locations and between the processor and memory. Finally, access to video memory in real and protected mode is used to illustrate data transfer between memory buffers and internal registers using string processing instructions and flow control instructions.

4.2 Pseudo-Instructions and Operators

Data type declarations are regarded as pseudo-instructions required to define data in terms of the memory storage required and the label associated to each storage definition. The term pseudo-instruction refers to the fact that these are assembly directives used in the instruction field of the typical assembly instruction format. Likewise, operators are prefixes used to modify the data statement, the operand, or the instruction that follows them.

4.2.1 Data Definitions

A set of pseudo-instructions are used to initialize data items in the data section or segment of the program. A variable is identified by a *label*, a variable type is associated with it, and an initial value is declared. The declaration of variables has the following general format:

```
[label][data type][initial value][;comments]
```

The following data types are supported by *nasm*:

db	define byte	1 byte
dw	define word	2 bytes
dd	define double word	4 bytes
dq	define quadword	8 bytes
dt	define ten	10 bytes

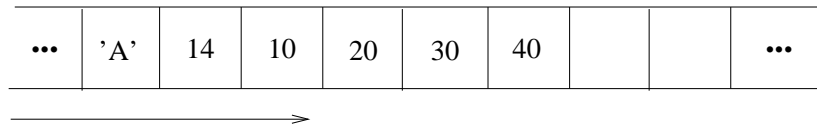
Data types *dq* and *dt* that are used to allocate 8 and 10 bytes, respectively, are not used to define numeric or string constants.

Define Byte: DB

The declaration of a DB type allocates one byte of memory. Examples:

```
a    db  'A'
x    db  14
list db  10,20,30,40
```

The above set of variables is allocated in the order in which they appear in the declaration as follows:



Note that the above memory map shows the lower addresses on the left expanded to the right by one byte at a time, and each square corresponds to one-byte location.

Different data representations are possible, for example:

```
list    db    32, 32h, 0x32, 00110010b
```

which declares an array "list" of bytes. Likewise a declaration of strings of bytes looks as follows:

```
message db    "hi there", 0    ;this is a null-terminated string
```

As the comment suggests, a 0 at the end of the string makes for a null-terminated string. Also expressions that resolve to a constant can be used to initialize data, for example:

```
rsiz    db    10*20+3
```

is a declaration that initializes *rsiz* to the value 63. The prefix operator *times* is used for large storage requirements. This is equivalent to the *DUP* operator in *tasm* and *masm*. For example the following declaration:

```
array   times 64 db 0
```

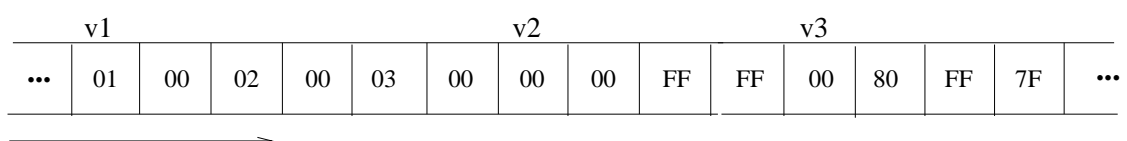
will initialize 64 bytes of "array" to a value of 0..

Define Word: DW

The declaration of a DW type allocates and initializes two-byte values. Examples:

```
v1  dw  1,2,3           ;1 = 0001h, 2 = 0002h, 3 = 0003h
v2  dw  0, 65535        ;65535 = FFFFh
v3  dw  -32768, +32767   ; -32768 = 8000h, 32767 = 7FFFh
```

The allocation of the above variables is described as follows:



As before, the memory map shows each word allocated in the order in which they appear in the declaration. Note however, that each word is allocated byte-wise following the *little endian* machine data representation observed by IA32 processors.

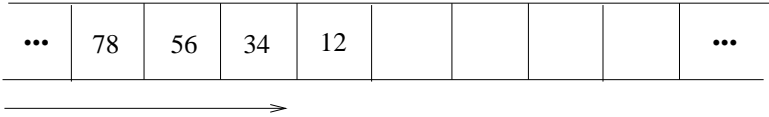
Little-endian vs. big-endian. Intel processors are *little-endian* because they write and read data keeping the lowest order data byte in correspondence with the lowest address in memory. This is illustrated in the allocation of words shown in the previous example; note that the word `7FFF` is stored in such a way that the byte `FFh` is stored in the lowest address in memory while the byte `7Fh` is placed in the highest address. If this word is read from memory and loaded in a 16-bit register such as `ax`, the `ah` register will contain the value `7Fh` and `al` will contain the value `FF`. On *big-endian* machines such as Motorola processors, the correspondence is reversed.

Define Double Word: DD

The following example illustrates the allocation of double words consistent with the little-endian data representation:

```
v    dd    12345678h
```

with an allocation illustrated as follows:



Un-initialized data

In case variables are not required to be initialized *nasm* will allocate storage using pseudo-instructions such as: `resb`, `resw`, `resd`, `resq`, and `rest` to reserve a byte, a word, a double word, a quadword and a 10-byte value, respectively. *Masm* and *tasm* use the character “?” to indicate non-initialized variables and the *dup* operator for large buffers. Examples:

```
array  resb  64  ; reserve 64 bytes labeled “array”
x      resw   10  ; reserve 10 words
y      resd    1  ; reserve 1 double word
```

Uninitialed storage space can be declared in the *bss* segment. However, these directives can also be used in the *data* segment as well.

4.2.2 More pseudo-instructions

Defining constants: *equ*

equ assigns a symbol to a string or a numeric constant. This assignment can not be changed later. Examples:

```
CR    equ    13
LF    equ    10
msg   equ    'hello world'
```

To complete the illustration the above *equ* statements can be used in the declaration of a string variable in a data segment as follows:

```
message    db    msg, CR, LF, '$'
```

Repeating data or instructions: *times*

This is a prefix followed by an integer *k* which indicates the number of times the instruction or data declaration that follows is repeated. Format:

```
times  k  instruction/data declaration
```

Examples:

```
warray:  times  100    dw 0    ; an array of 100 words is initialized to zero
```

```
times    50    movsb      ; executes movsb 50 times
```

Segment operator: *seg*

The *seg* operator returns the segment part of an address associated to a symbol. Examples:

```
mov  ax, seg label    ; extract segment part, and
mov  es, ax           ; load it into a preferred segment register
mov  di, label         ; now es : di is a valid pointer to label
```

Note that *seg* operator is useful for large 16-bit applications that extend beyond a single memory segment.

Offset operator: *wrt*

The *wrt* operator is used to define an offset with respect to a specific segment. Example:

```

mov ax, new_seg           ;new_seg is the segment base
mov es, ax
mov bx, array wrt new_seg ; es:bx point to array wrt new_seg

```

The operator *wrt* is useful for applications with several segments that may overlap, and it is not required for the offsets within the default segment.

4.3 Data Transfer Instructions

Data transfer instructions are designed to move data around from register-to-register, memory-to-register, register-to-memory, and even memory-to-memory. This set of instructions include those to manipulate the stack and those that allows transfers of arrays of data. The latter are referred to as "string processing instructions". Instructions execute operations that require operands; the way operands are accessed is specified in the instruction itself with a dedicated field referred to as the *addressing mode*. Several addressing modes supported by IA32 processors are discussed in this section.

4.3.1 The *mov* instruction

The *mov* instruction is the most common instruction designed to copy data from a source (*src*) operand to a destination (*dst*) operand. Both operands are specified in the instruction with a format as follows:

$$\text{mov } dst, src \quad ; dst \leftarrow src$$

The comment field is used to describe in standard transfer notation the internal transfer operation. This notation will be used at times to emphasize the *cpu*'s internal operation that takes place during the instruction execution. The source and destination operands can be a combination of a register (*reg*), a memory reference (*mem*), and a constant. Specifically, the following formats are common:

```

mov  reg, reg/mem
mov  reg/mem, reg/constant
mov  segreg, reg/mem
mov  reg/mem, segreg

```

Typical restrictions that apply to instructions with more than one operand include the following:

1. Destination and source operands must be of the same size. As noted before, the declaration of variable types has the purpose of allocating the right amount of memory space. However *nasm* does not keep track of the size of the variables, as intended in the code section of the program. To enforce the size declaration, the programmer must use a *size specifier* which is pre-appended to the operand used to reference memory, and ensure that the correct number of data bytes is accessed. Commonly used type specifiers are *byte*, *word*, and *dword*. Other type specifiers are *qword* and *tword*. For example the following instruction:

$$\text{mov } ax, \text{word } [x] \quad ; ax \leftarrow [x]_{16}$$

will enforce a transfer of 16 bits of data referenced by x to register ax . The following additional examples illustrate typical lines of code that enforce the number of bytes accessed by memory references:

$$\begin{array}{ll} \text{mov } al, \text{byte } [di] & ; al \leftarrow [ds : di]_8 \\ \text{neg } \text{word } [bx] & ; [ds : bx]_{16} \leftarrow \overline{[ds : bx]_{16}} + 1 \\ \text{add } eax, \text{dword } [es:esi] & ; eax \leftarrow eax + [es : esi]_{32} \end{array}$$

2. The *mov* instruction does not support transfer operations between two memory locations. If this is needed and intermediate register must be used to enforce the sequence:

$$\text{memory} \leftarrow \text{register} \leftarrow \text{memory}$$

3. A *constant* must be always used at the source operand field.
4. A transfer of a constant into a segment register (segreg) is not supported. If segment register must be pre-loaded with a constant use a general purpose register (gpreg) to enforce the sequence:

$$\text{segreg} \leftarrow \text{gpreg} \leftarrow \text{constant}$$

5. Transfers between segment registers are not supported. Instead, enforce the sequence:

$$\text{dest. segreg} \leftarrow \text{gpreg} \leftarrow \text{source segreg}$$

6. Moving data into *cs:eip* is not allowed. Since the pair *cs:eip* always points to the next instruction, an explicit over-writing of the contents of either register, will modify the program during execution with consequently unpredictable results.

4.3.2 Addressing modes

Upon fetching an instruction the control unit must also decode the way how operands are made available to the current instruction. A dedicated field in the instruction format specifies an addressing mode by which the processor will know if operands are immediately available in the instruction itself, or if they are fetched from internal registers, from the stack, or from general memory. The IA32 processors support the following addressing modes:

Register addressing. In this mode the source and destination operands specify the names of 8-bit, 16-bit, or 32-bit internal registers (except *cs* and *ip*). For example:

```
mov  ax, bx    ; ax ← bx
mov  si, ax     ; si ← ax
```

Immediate addressing. A source operand can be any 8-bit, 16-bit or 32-bit constant contained in the instruction itself and for which no additional memory reference is required. For example:

```
mov  al, 10          ; ax ← constant8
mov  ax, data         ; ax ← data
mov  ebx, 123456578h  ; ebx ← 123456578h
mov  di, msg          ; di ← msg
```

Note that *data* refers to the 16-bit value of the segment part of the base address where the data segment has been loaded in memory. In contrast *msg* is a label associated to a string allocated within the data segment; therefore, the value transferred to *di* corresponds to the offset within the data segment. Note once the program is loaded for execution both *data* and *msg* are constants.

Direct addressing. An *effective address* can be specified directly or indirectly in any of the two operand fields in the instruction format. A variable specified as a source or a destination operand is a direct reference to memory. In this case the processor uses the direct memory reference specified in the instruction to fetch the operand. *Nasm* surrounds the variable with square brackets to indicate a direct access to its contents. For example:

```
mov  eax, dword [count] ; eax ← [count]32
mov  dword [count], 10  ; [count]32 ← 10
```

Note that the subscript value emphasizes *nasm's* *dword* directive that a 32 bits are addressed by *count*.

Indirect addressing. Any register specified as a source or a destination operand contains the effective address. The processor uses the effective address to access the operand from memory. *Nasm* uses square brackets surrounding the register name to indicate a register-based indirect access. Examples:


```

mov  eax, dword [ebx]    ;  $eax \leftarrow [ebx]_{32}$ 
mov  dl, byte [bx]       ;  $bl \leftarrow [bx]_8$ 

```

The first transfer indicates that *ebx* contains an effective address that points to a 32-bit operand. The second illustrates an 8-bit transfer using a 16-bit address in *bx*. In both cases the effective address is preloaded into *ebx* and *bx*.

Index (Based) addressing. This is a form of indirect addressing in which the effective address (offset) is formed by adding the contents of an index register (index addressing) or a base register (base addressing) with a displacement. A displacement can be stated as a constant or as a *label*. This type of addressing mode can be used to access a linear array in which the *label* provides the base address of the array and an index register provides the moving offset within the array. Examples:

```

mov  al, byte [bx + 2]    ;  $al \leftarrow [bx + 2]_8$ 
mov  ebx, dword [y + si]  ;  $ebx \leftarrow [y + si]_{32}$ 
mov  word [di + z], ax    ;  $[di + z]_{16} \leftarrow ax$ 
mov  cx, word [bp + 10]   ;  $cx \leftarrow [bp + 10]_{16}$ 

```

The first transfer copies a byte at *bx*+2 into *al*. In the second move instruction, a double word at *y* + *si* is copied into *ebx* while a word is stored into *di* + *z* in the third move instruction. The last move instruction loads a word from the stack at *bp* + 10 into *cx*.

Based-indexed addressing. This is a more sophisticated indirect addressing mode designed to access multi-dimensional arrays. This mode uses both a base and an index register whose contents are added to a displacement. The displacement can be a constant or a *label*. Examples:

```

mov  byte [bx + si + 3], al    ;  $[bx + si + 3]_8 \leftarrow al$ 
mov  eax, dword [ebx*4]        ;  $eax \leftarrow [ebx * 4]_{32}$ 
mov  eax, dword [array + 8*edi] ;  $eax \leftarrow [array + 8 * edi]_{32}$ 

```

The first move instruction stores a byte at *bx* + *si* + 3. The second instruction implements a 32-bit transfer; the effective address can be calculated in several ways in terms of the address specified in *ebx* multiplied by a factor. Although *ebx* is a base register when multiplied by a factor it is treated as an index register. The last transfer loads a double word into *array* + 8 * *edi*.

A general expression to calculate the effective address for indirect access is given in terms of a base register, an index register, a factor, and a displacement as follows:

$$[base\ register + factor * index\ register + displacement]$$

where the base register can be one of *eax*, *ebx*, *ecx*, *edx*, *ebp*, *esp*, *esi* or *edi*. Likewise one of any of these registers except *esp* can be used for an index register. The factor

can be 0, 1, 2, 4 or 8 and the displacement can be a zero or a 32-bit constant or a *label*.

4.3.3 Stack operations

The stack is a *last-in-first-out* (LIFO) data structure. A data item *popped* from the stack is the last item *pushed* into the stack. IA processors control stack operations by assigning to the *esp* register the special task to always point to the *top of the stack*. To use the stack in a real-mode segmented model a stack segment must be declared in each program with the following declaration:

```
segment    name_of_stack    stack
          resb size_of_stack      ; declare size
stacktop:
```

Access to the stack segment *name_of_stack* will be possible after the initialization of the pair *ss:sp* with the following instructions at the beginning of the *code* segment:

```
mov  ax, name_of_stack
mov  ss, ax
mov  sp, stacktop
```

Note that in the flat model there is no need to initialize the stack as it is assigned the highest portion of addresses of the single flat segment.

Data transfers from and into the stack are either 16 (real-mode) or 32-bit (protected mode) items. The main instructions designed to manipulate the stack are *push* and *pop*.

Store data: **push**

The format of the *push* instruction is as follows:

```
push  reg/mem/constant    ;  $[ss : esp] \leftarrow reg/mem/constant$ 
```

The source is an explicit operand giving the name of a register, a memory reference, or a constant. The destination is an *implicit operand* given by the pointer to the top of the stack *ss:esp*. The *cpu* processes this instruction in two steps:

1. it updates the *sp* or *esp* registers by subtracting 2 or 4, (depending on the type of application) to point to the next entry in the top of the stack,

2. it transfers data to the top of the stack at $[ss:sp]$ or $[ss:esp]$.

To illustrate how the stack works during the process of a *push* operation consider the 32-bit code sequence shown in Fig. 1.1 which shows the stack before and after the execution of the sequence. The variations of the *push* instruction are given in table 1.1. Note that memory reference transfers can take place in 16 bits or 32 bits; a memory reference uses register indirect or a direct transfer with the effective address specified in the instruction itself.

Instructions that push all registers take no operands it instructs the processor to save the contents of the following sequence of registers: $(e)ax$, $(e)cx$, $(e)dx$, $(e)bx$, $(e)sp$, $(e)bp$, $(e)si$, and $(e)di$. The stack pointer is decremented by 16 or 32 bytes depending on the type of transfer. The number of bits pushed in the execution of *pusha* or *pushf* depends on the setting (16 or 32) of the directive *BITS*.

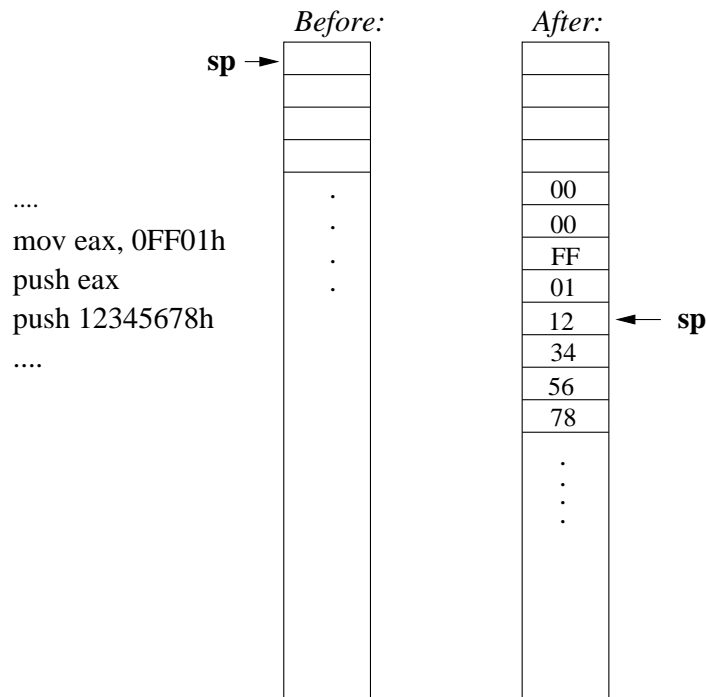


Figure 4.1: A sequence of *push* instructions

Load data: **pop**

The format of the *pop* instruction is the following:

pop *reg/mem* ; *reg/mem* $\leftarrow [ss : esp]$

Table 4.1: Variations of the *push* instructions

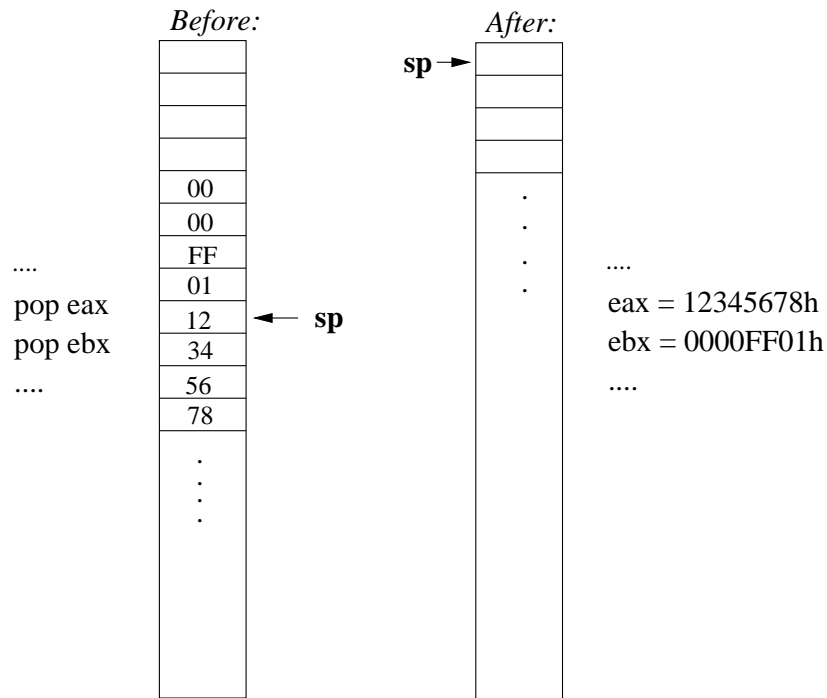
Symbolic	Operation	Note
push r	$[ss : (e)sp]_{16,32} \leftarrow r_{16,32}$	A 16,32-bit transfer
push (d)word [r/m]	$[ss : (e)sp]_{16,32} \leftarrow [r/m]_{16,32}$	A 16,32-bit mem. ref.
push imm	$[ss : (e)sp]_{8,16,32} \leftarrow imm_{8,16,32}$	An 8,16, or 32-bit transfer
pushaw	$[ss : (e)sp]_{16} \leftarrow all_regs_{16}$	16-bit transfers
pushad	$[ss : (e)sp]_{32} \leftarrow all_regs_{32}$	32-bit transfers
pusha		A 16 or 32-bit transfer
pushfw	$[ss : (e)sp]_{16} \leftarrow flags_{16}$	
pushfd	$[ss : (e)sp]_{32} \leftarrow flags_{32}$	
pushf		A 16 or 32-bit transfer

The *pop* instruction specifies a destination operand by providing the explicit name of a register or a memory reference. The source is the stack and it is an implicit operand given by the current pointer to the top of the stack *ss:esp*. The *pop* instruction is also processed in two steps but in the reverse order of the *push* instruction:

1. the *cpu* retrieves a data item from the top of the stack into the specified register or memory location,
2. the top of the stack, i.e., the pointer *ss:esp* is incremented by 2 or 4, depending of the type of application.

For illustration assume the state of the stack is as shown in Fig. 1.1, then a sequence of *pop* instructions changes the stack as shown in Fig. 1.2.

The variations of the *pop* instruction are shown in table 1.2. The programmer must be aware that for every *push* instruction a corresponding *pop* instruction must be executed. If a *pushaw* or a *pushad* are used, then the corresponding pop instructions *popaw* and *popad* must be executed to restore the contents of the registers in the opposite sequence in which they were saved: *(e)di*, *(e)si*, *(e)bp*, *(e)sp*, *(e)bx*, *(e)dx*, *(e)cx*, and *(e)ax*. Accordingly the stack pointer will be incremented two bytes for 16-bit transfers and four bytes for 32-bit transfers. Similar to the *pusha* and *pushf* instructions, the *popa* and *popf* instructions use 16 or 32 bit transfers depending on the value of the directive *BITS*.

Figure 4.2: A sequence of *pop* instructions

4.4 Flow Control Instructions

Any application development process requires a design step. Assembly applications are not the exception, particularly for a complex undertaking. The design process involves a clear understanding of the set of specifications, computational and I/O requirements. Typical design tools before writing code include flowcharts, pseudo-code, etc., where the number of modules and their interactions are defined. After the design phase implementation issues must be addressed. If assembly code is required issues that must be resolved include for example, the use of 16-bit or 32-bit binaries, coordination of several modules, and whether or not to combine assembly language modules with high level language modules, etc. Finally, the actual implementation of assembly modules requires an efficient choice of instructions that optimizes code in terms of memory space and/or execution time.

Because of the need to connect sections of code conditionally or unconditionally, *control structures* are an important element in the implementation of any application. Any language must include support mechanisms to control the flow of a program in response to the changing state of execution.

Table 4.2: Variations of the *pop* instruction

Symbolic	Operation	Note
pop r	$reg_{16,32} \leftarrow [ss : (e)sp]_{16,32}$	A 16,32-bit transfer
pop (d)word [r/m]	$[r/m]_{16,32} \leftarrow [ss : (e)sp]_{16,32}$	A 16,32-bit mem. ref. transfer
popaw	$all_regs_{16} \leftarrow [ss : (e)sp]_{16}$	16-bit transfers
popad	$all_regs_{32} \leftarrow [ss : (e)sp]_{32}$	32-bit transfers
popa		A 16 or 32-bit transfer
popfw	$flags_{16} \leftarrow [ss : (e)sp]_{16}$	
popfd	$flags_{32} \leftarrow [ss : (e)sp]_{32}$	
popf		A 16 or 32-bit transfer

4.4.1 Comparisons and Flags

The flow of a program changes according to the state of certain flags in the register flag. A program in execution must check for these flags and decide whether or not to take a jump out of the current flow. Many instructions change the set of flags during execution. One of these instructions is the comparison instruction that can be coded just 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 a subtraction operation between the source and the destination operands but leaves the *destination* operand unchanged. Using register transfer notation the operation can be described as follows:

$result \leftarrow destination - source$

The operands are located in memory or in registers 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 loaded from or stored in memory. An immediate operand (a constant) can only be specified as a source operand. The objective of the comparison is to affect flags depending on the result. The flags that may be affected by the subtraction operation are the following: the overflow flag (OF) if overflow occurs, the zero flag (ZF) is set if the result is zero, the sign flag (SF) is set if the result is negative, the carry flag (CF) is set if there is a carry (the carry flag is set if an overflow occurs after an un-signed operation).

4.4.2 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 a different instruction address. *Unconditional* branch instructions always update *cs:eip*. *Conditional* branch instructions will update *cs:eip* according to the status of some flag. Therefore, a conditional branch instruction will always follow a comparison instruction or any other instruction that may or may not change a flag. Perhaps it is fair to mention that the sequence of a comparison instruction and a conditional branch instruction implement a decision point in a flowchart, or a decision statement such as "if", "while", etc., in a pseudo-code sequence.

Unconditional jumps

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

<i>jmp</i>	<i>short</i>	<i>label</i>
<i>jmp</i>	<i>near</i>	<i>label</i>
<i>jmp</i>	<i>far</i>	<i>label</i>
<i>jmp</i>	<i>word</i>	<i>label</i>
<i>jmp</i>	<i>dword</i>	<i>label</i>
<i>jmp</i>	<i>reg/mem</i>	

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 1.3 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

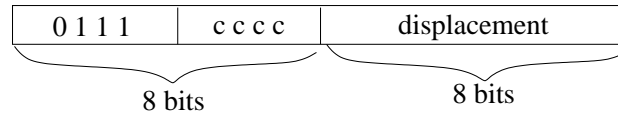
two groups is given in terms of the results of a *cmp* instruction. Group 2 deals with signed integers; the flag involved is indicated in the description column. Table 1.3 is not an exhaustive list of conditional jump instructions. For each conditional jump of the form "jcc" Intel instruction sets also includes the corresponding complemented conditional jump instruction "jnc".

Table 4.3: Classification of conditional jump instructions

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

Jumps out of range

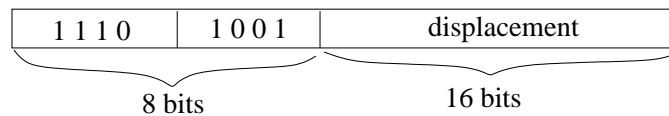
For real-mode applications conditional jumps are limited to 8-bit jumps, i.e., within a range of -128 and 127 bytes. Conditional jumps are two-byte instructions where the highest order byte contains the opcode and the lowest order byte contains a displacement. The CPU utilizes the displacement value 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 16-bit 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 the 8-bit conditional short jump with a 16-bit unconditional near jump. In real mode, near unconditional jumps use a 16-bit displacement to allow a range within $\pm 32K$ bytes. Near unconditional jump instructions have the following binary format:



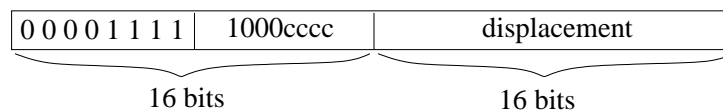
Therefore, in general terms any conditional jump of the form *jcc label* that results in a "jump-out-of range" can be replaced by the following sequence:

```

...
jncc  new_label
jmp   label
new_label:
...
```

where *jncc* 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 with a 16-bit displacement.

4.4.3 The *loop* Instruction

During the design process of an application, a decision point in a flowchart or pseudo-code is followed by a block that performs a task. The implementation of this block results in a block of code that may change the condition tested to enter the block in the first place making the testing of the condition necessary once more. The conditional execution of a block of code results in a *loop* if the condition tested at each iteration remains true. The appropriate conditional jump instruction can be used to implement a loop. Another alternative is to use a *loop* instruction that most assembly languages provide. The Intel instruction set provides a loop instruction that repeats the execution of a block of code as long as the contents of the *ecx* register are greater than zero. Therefore, if the *loop* instruction is used, it is important to always initialize *ecx* before the first iteration starts. Other variations of the *loop* instruction involve the zero flag as commented in the following available 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  $Z = 0$ 

```

Note that the term "and" is a logical connection that emphasizes that both events should be true to make the overall condition true.

4.4.4 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 with assembly instructions. In this section we illustrate only few of the structures available in high-level languages and their mapping, in general terms, into assembly blocks. Pseudo-code and flowcharts are used to emphasize the design phase that precedes the actual code implementation and because the pseudo-code specification of any program has a close resemblance to its HLL implementation.

If-then blocks.

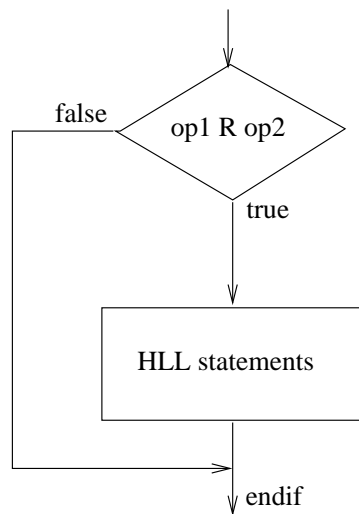
The following pseudo-code illustrates the use of *IF* statements:

```

if (op1 R op2) then
    ...
    HLL statements
    ...
endif

```

The term R is used to generalize the conditional relationship between *op1* and *op2*. The following flowchart can be used as an alternative description of *if-then* blocks:



To map HLL constructs into an assembly block the notation *jcc* and *jncc* used before, refers to the list of jump instructions in Table 1.3. The following assembly structure implements the HLL *if* statement

```

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

```

To illustrate consider the following pseudo-code:

```

if (Y<X) then
    X = Y
endif

```

The following is a possible implementation in assembly language:

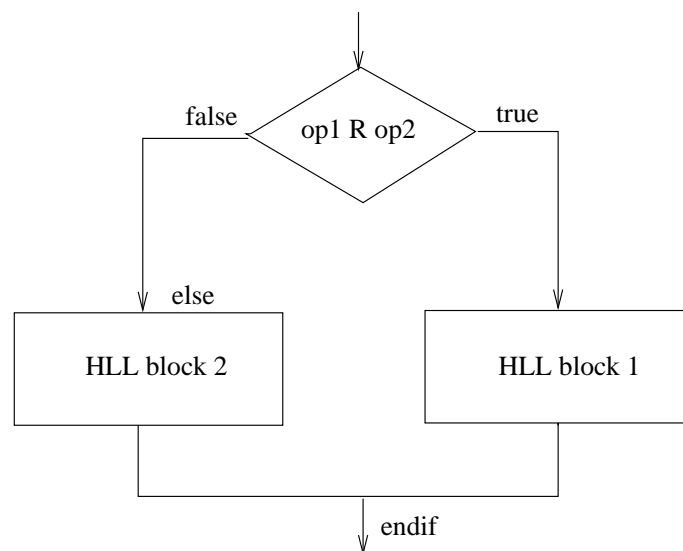
```

    mov ax, word [Y]
    cmp ax, word[X]
    jge endif          ; code is not executed if (<) is not true
    mov word [X], ax
endif:  ...

```

IF-then-else blocks

In a similar fashion *if-then-else blocks* are described by the following flowchart in which the only difference is that either one block of statements (block 1) is executed if the condition is met if the condition is not met then a second block of statements is executed (block 2).



if-then-else structures can also be described by pseudo-code as follows:

```

if (op1 R op2) then
    ...
    HLL block 1
    ...
else
    ...
    HLL block 2
    ...
endif

```

the following sequence which shows the appropriate combination of the compare instruction, conditional and unconditional jumps, 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
        ...
        jmp endif
else:
        ...
        assembly code
        for block 2
        ...
endif:

```

A simple extension of the previous example illustrates the execution of alternate blocks of code:

```

if (Y<X) then
    X = Y
else
    Y = X
endif

```

with a possible assembly code implementation as follows:

```

        mov ax, word [Y]
        cmp ax, word[X]
        jge else      ; block 1 is not executed if (<) is not true
        mov word [X], ax
        jump endif
else:   mov ax, word [x] ; execution of block 2
        mov word [y], ax
endif:  ...

```

While loops.

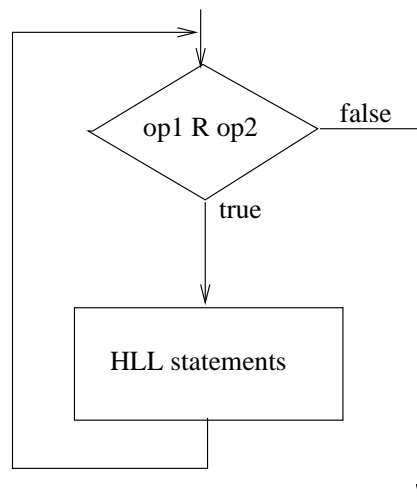
These are very common structures where the condition is first tested before another iteration is executed. The following pseudo-code illustrates a typical *while* loop structure:

```

while (op1 R op2) do
    ...
    HLL statements
    ...
endwhile

```

A flowchart can be used to describe these blocks as follows



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:           ...

```

For illustration consider the following example:

```

while (Y<X) do
    Y = Y+1
endwhile

```

This is a very simple loop that can be implemented as follows:

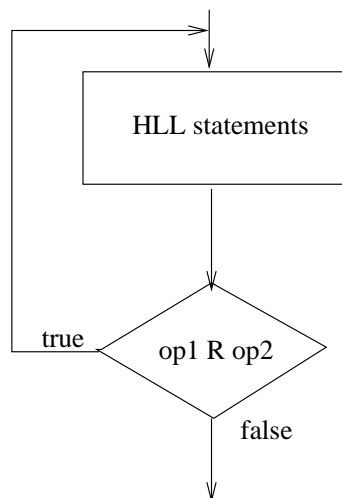
```

...
mov eax, dword [Y]
while:
    cmp eax, dword [X]
    jge endwhile      ; code is not executed if cc is not met
    inc eax
    jmp while
endwhile:
mov dword [Y], eax
...

```

Do while loops.

In these loop structures the condition is tested after each iteration as shown in the following flowchart:



The pseudo-code description looks as follows:

```

do
    ...
    HLL statements
    ...
while (op1 R op2)

```

A possible assembly implementation is given below:

```

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

```

The following example illustrates two possible assembly implementation of a very simple HLL loop described as follows:

```

do
    Y = Y+1
    X = X-1
while (X > 0)

```

A first implementation makes use of explicit conditional jumps:

```

do:
    inc dword [Y]
    dec dword [X]
    cmp dword [X], 0
    jg do           ; a new iteration is executed if X>0 is true

```

A second implementations makes use of the *ecx* register as a counter and the *loop* instruction:

```

mov ecx, dword [X]
do:
    inc dword [Y]
    loop do         ; a new iteration is executed if ecx > 0
    mov word [X], ecx ; implied result in the pseudo-code

```

4.5 Array/String Transfer Instructions

This is a set of instructions designed to transfer large amounts of data from one array to a different array or from/to the *cpu*. The execution of one of these instructions will transfer one data item (byte, word, or double word) from a source array to a destination array. The source array is located at *ds:esi* and after the execution of a single transfer, the index register *esi* is incremented or decremented automatically to

point to the next data unit. Similarly, the destination array is located at *es:edi*, and after each single transfer the index register *edi* is also incremented or decremented automatically. Source and destination pointers increment or decrement depending on the state of the direction flag *D*. A *D* = 0, which is the default state, indicates an automatic increment of the contents of the index registers, otherwise, an automatic decrement occurs. The instruction set provides two instructions to set or reset the direction flag: *cld* will establish the auto-increment mode, while *std* sets the flag *D* to one and establishes the auto-decrement mode. Automatic increment or decrement of the pointers involved makes string/array transfer instructions inserted within a loop, very useful to transfer large amounts of data between different data buffers. Array/string transfer instructions with implicit operands are classified in the following groups:

Memory-to-memory transfers. The instructions that perform these transfers are the following:

```
movsb   ; [es : edi]8 ← [ds : esi]8
movsw   ; [es : edi]16 ← [ds : esi]16
movsd   ; [es : edi]32 ← [ds : esi]32
```

Memory-to-register transfers. A data item is transferred from a source array into a register one byte, a word, or a double word at a time, depending on which of the following instructions is executed:

```
lodsb   ; al ← [ds : esi]8
lodsw   ; ax ← [ds : esi]16
lodsd   ; eax ← [ds : esi]32
```

Register-to-memory transfers. The flow of data items implemented by this instructions is from *cpu* register to a destination array one data unit at a time:

```
stosb   ; [es : edi]8 ← al
stosw   ; [es : edi]16 ← ax
stosd   ; [es : edi]32 ← eax
```

Example 1. The following code segment implements a data transfer from a source list into a destination list:

```
SEGMENT data
    list_s    dw    1000h, 2000h, 3000h, 4000h
```

```

SEGMENT bss
    list_d    resw    4

SEGMENT code
..start:
    mov ax, data
    mov ds, ax
    ...
    ...
    mov si, list_s    ; sets the pointer ds:si to list_s
    mov ax, bss
    mov es, ax
    mov di, list_d
    mov cx, 4
    rep movsw          ; ds:si --> es:di
    ...
    ...

```

The *rep* prefix causes the repetition of the next instruction a specified number of times. The *rep* prefix is not an instruction in itself but it is a byte field appended to the currently fetched instruction intended to implement a single-instruction loop with a number of iterations specified in *ecx*.

Example 2. This example shows the implementation of the transfer of the contents of video memory in page 0 to a data buffer:

```

SEGMENT bss
    dbuffer    resw    2000

SEGMENT code
..start:
    ...
    ...
    mov ax, bss
    mov es, ax
    mov di, dbuffer
    mov ax, 0B800h
    mov ds, ax
    mov cx, 2000
    mov si, 0
next:
    movsw          ; ds:si --> es:di
    loop next
    ...

```

...

As will be discussed in the next section, a page p in video memory is located at an offset of $4096 \times p$ bytes. Note that for $p = 0$ then the offset is zero as implied in the instruction *mov si, 0* in example 2.

4.6 Video Memory

Recall from the memory map discussed in Chapter 1, that the 8086/88 assigns a memory section to video display within the 1 Megabyte of its directly addressable space. The section from A0000h to AFFFFh is assigned to graphics display, from B0000h to B7FFFh to a monochrome display adapter (MDA) for text display, and from B8000h to BFFFFh is used for CGA/EGA/VGA adapter for text display. The text buffer beginning at B8000h spans a size of 32 Kbytes. This space is divided in 8 pages and the size of each page is 4 Kbytes. The programmer can select any page indexed from 0 to 7 for display. The video memory map for a color display adapter is as follows:

B800:0000 — page 0
 B800:1000 — page 1
 B800:2000 — page 2
 B800:3000 — page 3
 B800:4000 — page 4
 B800:5000 — page 5
 B800:6000 — page 6
 B800:7000 — page 7

The screen displays a maximum of $25 \times 80 = 2000$ characters in text mode. The total number of rows in this mode is 25 and the total number of columns is 80. A particular character location in the screen is referenced in terms of its row and column coordinates. A page in video memory is displayed one at a time and each character in video memory requires two bytes. The first byte is used for the character itself, and the second byte is used for the display attribute. The additional attribute byte doubles the memory used by the characters to a total of 4 Kbytes which is the size assigned to each page. The attribute format is as follows:

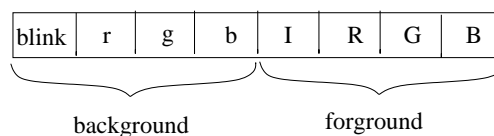
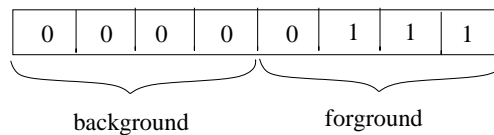


Table 1.4 shows the possible bit combinations using the three bits available for either the foreground and background colors. For example, an attribute with value $07h$ specifies a white foreground on a black foreground display.

Table 4.4: Attribute values

Foreground or Background		Foreground only	
000	black	1000	gray
001	blue	1001	light blue
010	green	1010	light green
011	cyan	1011	light cyan
100	red	1100	light red
101	magenta	1101	light magenta
110	brown	1110	yellow
111	white	1111	bright white

Consider for example a null-terminated string such as *abcdef...* To display this string with a given attribute it will have to be stored in video memory as follows:



In general to display any information, we will need to map the initial coordinates in the screen and the page number into the corresponding offset in video memory. Let O denote the offset with respect to the initial address $B8000h$, then to obtain O in bytes use the expression:

$$O = 160r + 2c + 4096p$$

where r and c are the row and column, respectively, where the initial character of a message will be displayed, and p is the page number selected. Consider again the implementation of example 2 in the previous subsection in which page 0 of video memory is copied into a data buffer. Any page, or a section of the page, can be copied into a local buffer by specifying the three main parameters (r, c, p) to find the offset of video information with respect to the base address $B8000$. Two possible implementations are discussed:

1. Replace the line with the instruction *mov si, 0* with the following:

*mov si, 160*r+2*c+4096*p*

where *r*, *c*, and *p* can be specified with *equ* statements with the desired coordinate values.

2. Use the following line in place of the instruction *mov si, 0*:

mov si, offs

and add a *%define* directive at the beginning of the program as follows:

*%define offs 160*r+2*c+4096*p*

right after the *equ* declarations for *r*, *c*, and *p*.

The code to write into or read from video memory can be generalized to be used with any set of coordinate parameters. The design and use of sub-procedures is one of the topics to be addressed in a subsequent chapter.

4.7 Exercises

1. Write the most appropriate data declaration statement for the following cases:
 - (a) Three distinct variables initialized with the following values:
 ABCD0123h , 001Fh , "ABCD"
 - (b) Draw a memory map to indicate how the values declared in a) are allocated in memory.
2. Use an appropriate data declaration to initialize the following items in memory:
 - (a) An array of 1000 bytes initialized to zero.
 - (b) A $\$$ -terminated string: "which class is this?".
 - (c) An array of 1000 double words initialized to zero.
3. Explain the difference between *big endian* and *little endian* formats for storing numerical data larger than a 8 bits in width.
4. Identify whether or not the following *mov* instructions are valid or invalid. Justify your answer.
 - (a) *mov al, 1234h*
 - (b) *mov ax, var*
 - (c) *mov bx, [var]*

- (d) `mov 10h, al`
 - (e) `mov dl, ax`
 - (f) `mov eax, 10h`
 - (g) `mov ds, data`
 - (h) `mov x, y`
 - (i) `mov eip, dword [x]`
5. Identify the source and destination addressing modes used in each of the instructions listed.
- (a) `mov di, msg`
 - (b) `xor eax, eax`
 - (c) `mov eax, dword [total]`
 - (d) `mov ax, word [di]`
 - (e) `mov ebx, dword[esi + y]`
 - (f) `mov dword [ebx + 4*edi + array], eax`
6. Explain the operation of the following instructions:
- (a) `stowb`
 - (b) `lodsw`
 - (c) `movsd`
 - (d) `stosd`
7. Consider the instruction *push reg/mem*. Explain briefly the CPU operations performed to execute this instruction.
8. Which registers move onto the stack with the execution of *pusha* ?
9. Which registers move onto the stack with the execution of *pushad*?
10. Describe the operation of each of the following instructions:
- (a) `pop eax`
 - (b) `push word [bx]`
 - (c) `pushfd`
 - (d) `pop esi`
 - (e) `pop dword [edi]`

11. Identify which conditional jump instruction follows the comparison of unsigned numbers.
12. Identify which conditional jump instruction follows the comparison of signed numbers.
13. Contrast the operation of a *jmp x* with a *jmp word [x]*.
14. Develop a sequence of instructions that searches an array of 0100h bytes, and counts all entries above or equal to 100 and place them into an array *above*; it also counts all entries below 100 and place them in an array *below*. Define all data types required for the two counts and the two arrays needed.
15. Explain the purpose of the direction flag, and indicate which instructions are used to clear it and set it.
16. Write a sequence of instructions to transfer 100 bytes of data from a memory buffer addressed by *source* into memory buffer addressed by *dest*.
17. The area of video memory for color display in text mode begins at *B8000h*. A particular page *p* is located at $4096 \times p$. Thus, to access page 0 it suffices to set the corresponding index register to zero. Write a sequence of instructions to transfer 2000 words in a data buffer to page 0 in video memory.