

Chapter 4

Data Movement 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 the data movement instructions along with code illustrations. 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.

4.2 Data Definitions

Data is initialized 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

4.2.1 Define Byte: DB

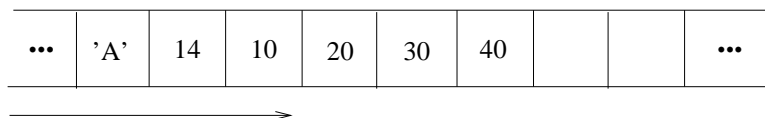
The declaration of a DB type allocates one byte of memory. Examples:

```

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

```

The above set of variables is allocated in memory as follows:



Note that *nasm* accepts different data representations, for example:

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

and declaration of strings as follows:

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

also expressions that resolve to a constant can be used to initialize data, for example:

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

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 statement:

```
array    times 64 db 0
```

will initialize 64 bytes of “array” to a value of 0..

4.2.2 Define Word: DW

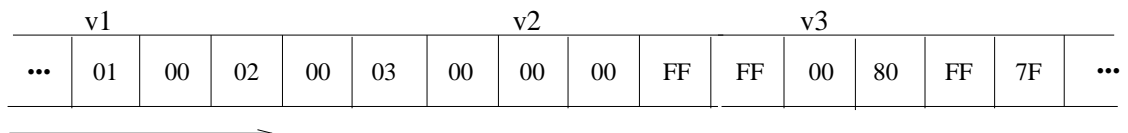
The declaration of a DW type allocates and initializes 2-byte values. Examples:

```

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:



Little-endian vs. Big-endian. Intel processors 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.

4.2.3 Define Double Word: DD

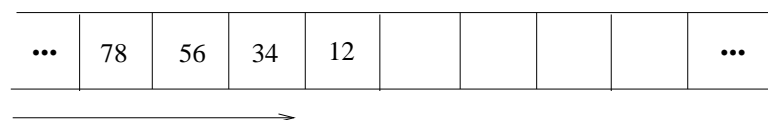
The following example illustrates the allocation of double words consistent with the little-endian data representation:

```

v  dd  12345678h

```

with an allocation illustrated as follows:



4.2.4 Un-initialized data

In case variables are not required to be initialized *nasm* will allocate storage using directives 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
```

4.3 Data Transfer Instructions

The *mov* instruction copies data from a source (*src*) operand to a destination (*dst*) operand. Both operands are specified in the instruction which has a format as follows:

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

where 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 execution. The source and destination operands can be a combination of a register (*reg*), a memory reference (*mem*), and a constant:

```
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. Declaration of variable types have the purpose of allocating the right amount of memory space. Note however that *nasm* does not keep track of the size of the variables in the code segment of the program. To enforce a uniform size, the programmer can use a *size specifier* pre-appended to the operand that is used to reference

memory to ensure that the correct number of data bytes is fetched. The most commonly used type specifiers are *byte*, *word*, and *dword*. Other type specifiers are *qword* and *tword*. For example the following instruction:

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

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 fetched from memory:

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

2. A *mov* instruction does not support transfer operations between two memory locations.
3. A *constant* must be always used at the source operand field.
4. A transfer of a constant into a segment register is not supported.
5. Transfers between segment registers are not supported.
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 unpredictable results.

4.3.1 Addressing modes

Upon fetching an instruction the control unit must also decode the way how operand are made available to the current instruction. A dedicated field in the instruction format specifies an addressing mode by which the *cpu* 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 IA processors support the following addressing modes:

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

```
mov    ax, bx    ;the contents of bx are copied to ax
mov    si, ax     ;transfer for ax to si
```

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           ;an 8-bit constant is copied into al
mov  ax, data         ;the base address of the data segment is copied
                      ;into ax
mov  ebx, 123456578h  ;a 32-bit value is copied into ebx
mov  di, msg          ;the address (offset) of a string is copied into di

```

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 *cpu* 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] ; a 32-bit value at count is copied into eax
mov  dword[count], 10   ; a double word constant is stored at count

```

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

```

mov  eax, dword [ebx]  ; ebx contains the effective address of the operand
mov  bx, msg           ; an offset of a string is placed into bx
                      ;note that this is an immediate addressing case
mov  bl, byte [bx]     ; a character from the string is indirectly
                      ;placed into bl

```

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] ; a byte at bx + 2 is copied into al
mov  ebx, dword [y + si] ; a double word at y + si is copied into ebx
mov  word [di + z], ax  ; a word is stored into di + z
mov  cx, [bp + 10]      ; load 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      ; store a byte at bx + si + 3
mov    eax, [ebp*3]                ; equivalent to moveax, [ebp * 2 + ebp]
mov    eax, dword [array + 8*edi] ; load a double at array + 8 * edi

```

Note that indirect addressing can be expressed in a general form 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 1, 2, 4 or 8 and the displacement can be up to a 32-bit constant or a *label*.

4.3.2 More Pseudo-Instructions

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, '$'

```

TIMES: This is prefix followed by an integer *k* which indicates the number of times the instruction that follows is executed. Format:

$$times\ k\ instruction$$

Examples:

```

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

times      50        movsb      ; executes movsb 50 times

```

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

4.4 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 *ss : sp* register 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 assigned.

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*.

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

```
push  reg/mem/constant    ; [ss : esp] ← 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 $[ss : esp]$.

To illustrate how the stack works during the process of a *push* operation consider the 32-bit code sequence shown in Fig. 4.1 which shows the stack before and after the execution of the sequence. The variations of the *push* instruction are given in table 4.1. 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 because the implicit operands are the registers whose contents are stored in the stack in the following sequence: $(e)ax$, $(e)cx$, $(e)dx$, $(e)bx$, $(e)sp$, $(e)bp$, $(e)si$, 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*.

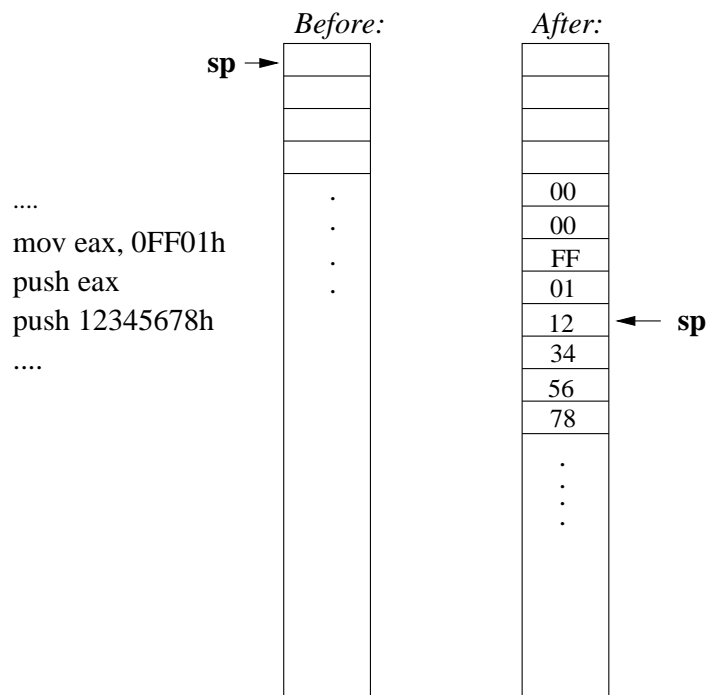


Figure 4.1: A sequence of *push* instructions

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

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

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

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

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:

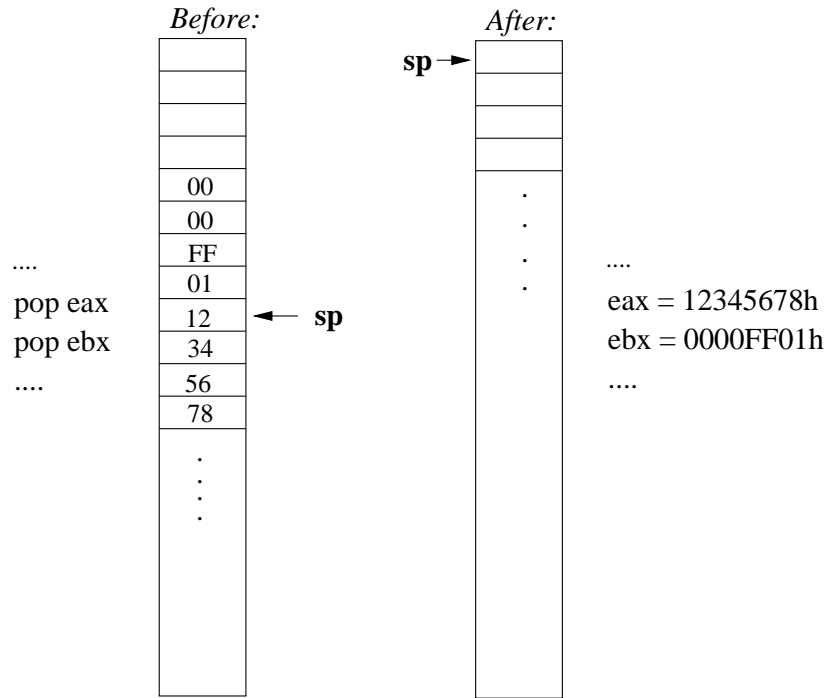
1. the *cpu* retrieves a data item from the top of the stack into the specified register or memory location,
2. the top of the stack, i.e., the content of $ss : esp$ is incremented by 2 or 4, depending of the type of application.

For illustration assume the state of the stack is as shown in Fig. 4.1, then a sequence of *pop* instructions changes the stack as shown in Fig. 4.2.

The variations of the *pop* instruction are shown in table 4.2. Note that 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 stored: $(e)di$, $(e)si$, $(e)bp$, $(e)sp$, $(e)bx$, $(e)dx$, $(e)cx$, and $(e)ax$. Accordingly the stack pointer will incremented 16 bytes for 16-bit transfers and 32 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*.

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 source array is located at $ds:esi$

Figure 4.2: A sequence of *pop* instructions

and after each 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 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. 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   ;  $[es : edi]_8 \leftarrow [ds : esi]_8$ 
movsw   ;  $[es : edi]_{16} \leftarrow [ds : esi]_{16}$ 
movsd   ;  $[es : edi]_{32} \leftarrow [ds : esi]_{32}$ 

```

Memory-to-register transfers. A data unit 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:

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

lodsb ; $al \leftarrow [ds : esi]_8$
 lodsw ; $ax \leftarrow [ds : esi]_{16}$
 lodsd ; $eax \leftarrow [ds : esi]_{32}$

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 ; $[es : edi]_8 \leftarrow al$
 stosw ; $[es : edi]_{16} \leftarrow ax$
 stosd ; $[es : edi]_{32} \leftarrow eax$

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, the 8086/88 assigns within the 1 Mbyte directly addressable space, a section to video display. The section from A0000h to AFFFFh is dedicated to graphics display, from B0000h to B7FFFh to a monochrome display adapter (MDA) text buffer, 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. 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 location in the screen is referenced in terms of its coordinates, i.e., its row and column. A page in video memory is displayed one at a time, however, two bytes are needed to display a character. The first byte is used for the character and the second byte is used for the display attribute. The additional byte required for the attribute 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:

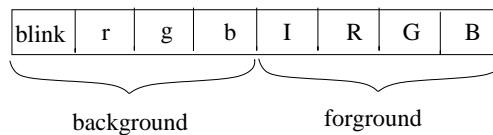
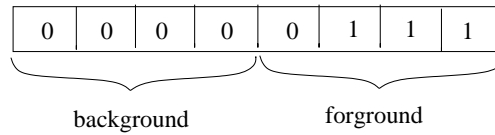


Table 4.3 shows the possible values on combinations using the three bits available for the foreground and background. For example, an attribute with value 07h specifies a white foreground display on a black foreground.

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:

Table 4.3: 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



In general to display any information, we will need to map the initial coordinates in the screen and the page number into the offset in video memory. Let O denote the offset in video memory 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 where the initial character of the 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 entire page or a section of the page can be copied into a local buffer by specifying the three main parameters to find the offset of video information with respect to the base address B8000. Two possible implementations are discussed:

- 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.

- 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.