

# Chapter 3

## Assembly Programming 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 are discussed.

### 3.2 Modes of Operation

The IA32 family of processors supports three operating modes. The *real-address mode*, provides the user with the programming environment (software model) available in the Intel 8086/88 processor. The *protected mode* is the native operating mode available to users; all instructions and architectural features are available, providing the highest performance and capability. A third mode of operation available to users in protected mode is a quasi-operating mode known as *virtual-8086 mode*. This mode allows the processor execute 8086 software in a protected, multitasking environment.

### 3.2.1 Real-address Mode

In real-address mode the processor 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:

$$\text{Physical address} = 2^4 \times \text{segment} + \text{offset} \quad (3.1)$$

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 applying equation 3.1 the physical address is then obtained as:  $92800h + 0022 = 092A2h$ , 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'. Memory segmentation corresponds to the way in which 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 of an assembly program. 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 assembly program 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 programmed in assembly may contain a stack section where the size of the stack is declared.

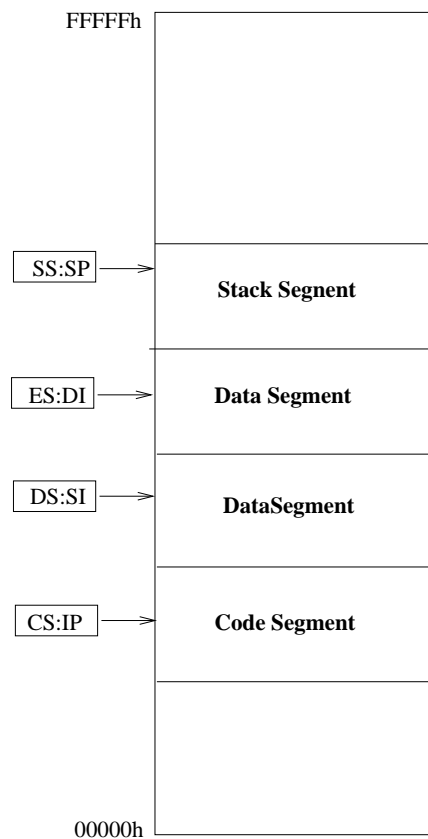


Figure 3.2: Placement of memory segments

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.

### 3.2.2 Protected Mode

As the complexity and size of applications increased, the maximum limit of 64 Kbytes 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 processor as well as main memory. Protected mode was developed to respond to additional memory requirements and to support a multitasking execution 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 an 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; therefore, 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 is a 64-bit *descriptor* that contains information regarding the size of the segment, its current location, and access rights. Segment registers are used to access a descriptor table; To access a descriptor table, the contents of a segment register are divided into three fields: a 13-bit *selector* field that points to the descriptor as shown in Fig. 3.3. The 2-bit field RPL, indicates the requested privilege level, and the one-bit *TI* field indicates whether 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. Each table type has 8192 entries; therefore, each program has a total of 16,384 descriptors available at any time. Fig. 3.6 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 16 Megabytes. 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 processor 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 (the *G* field – bit 7 in the last word in the descriptor) selected. The *D* field (bit 6) indicates the default length for effective addresses and operands referenced by the instructions in the segment. If the flag is set, 32-bit addresses and 32-bit or 8-bit operands are assumed (*32-bit instruction mode*), otherwise, 16-bit addresses 16-bit or 8=bit operands are assumed (*16-bit instruction mode*). Bit 5



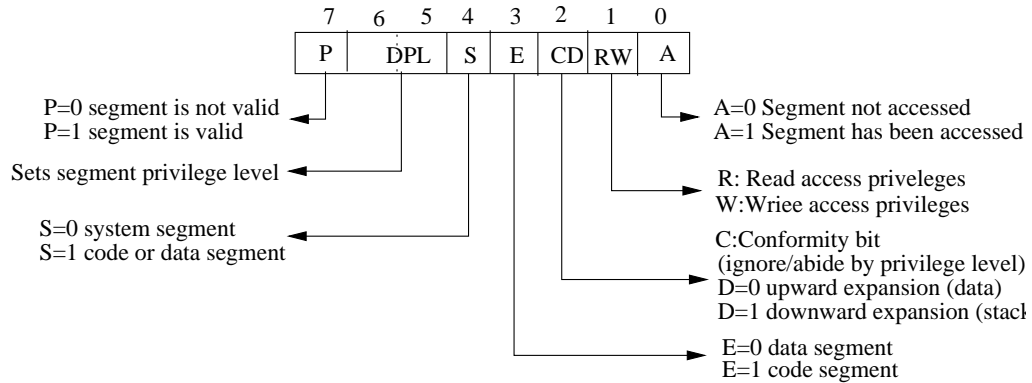


Figure 3.5: Access rights for 80286–Pentium segments.

(D=0) and read and write access (RW=1) and that it has been recently accessed (A=1).

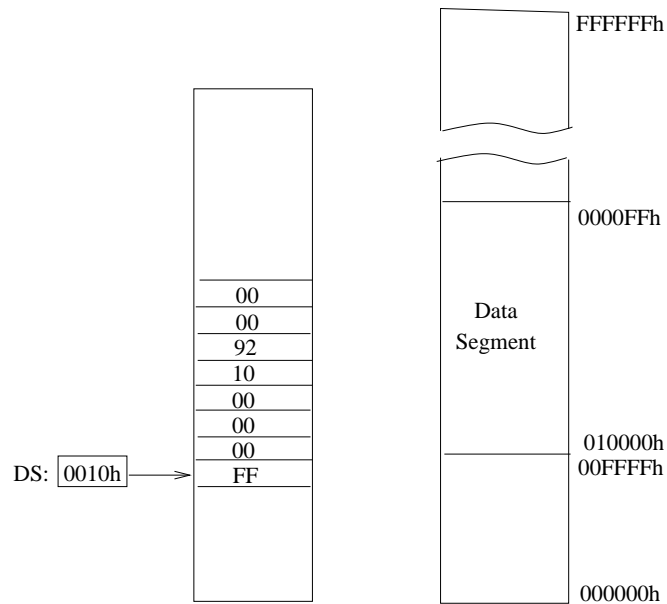
### 3.2.3 Virtual-86 Mode

This mode allows an emulation of the 8086/88 DOS-based real mode environment. The opening of any DOS window involves the launching of a virtual 8086 mode. This process is controlled by the hosting operating system supporting a multitasking switching environment. Launching a virtual-86 mode involves the activation of a new task. The operating system saves the current status of an executing task and sets the VM bit of the *eflags* register. Suspending a virtual-86 session involves suspending the controlling task by saving its status including the state of the *eflags* register. The previous state is restored and the corresponding task takes control of the processor.

## 3.3 Descriptor Tables

The data structures to manage memory in IA32-based systems include three types of descriptor tables; the Global Descriptor Table (GDT), the Local Descriptor Table (LDT) and the Interrupt Descriptor Table (IDT). The GDT is accessible across all programs and tasks and it is pointed to by the *tgdt* register. The LDT is local to each task and it is pointed to by the *ldtr* register. Each table holds up to 8192 entries. The state of the *TI* bit in the segment register (see Fig. 3.3) selects which data structure to access. The IDT contain up to 256 gates and each gate holds the destination for various interrupts subroutines.

The main purpose of the *gdtr* and the *ldtr* registers is to locate the correspond-



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

00	00
93	10
00	00
00	FF

b) 80286 Descriptor

00	50
93	10
00	00
00	FF

c) Pentium Descriptor

Figure 3.6: Use of segment registers and descriptor table to access physical memory in the 80286 and Pentium processors.

ing descriptor tables. The *gdt* is a 48-bit register with a 32-bit field for the base address and a 16-bit field to hold the table limit. The base address specifies the linear address of byte 0 in the GDT and the table limit specifies the number of bytes in the table. The instructions *ldgt* and *sgdt* are provided to load and store the *gdt* register, respectively. By default the base address is initialized to zero with a limit set to FFFFh. As part of the initialization process a new base address must be loaded for protected mode operations. Fig. 3.7 summarizes the role of segment registers, the *gdt* register, the global descriptor table, and offset registers in the address translation process. Since each descriptor contains 8 bytes the selector value is multiplied by 8 to obtain to point to the correct descriptor.

The GDT must contain a segment descriptor to locate the LDT segment. If the system supports multiple LDTs, each must have a separate segment selector and

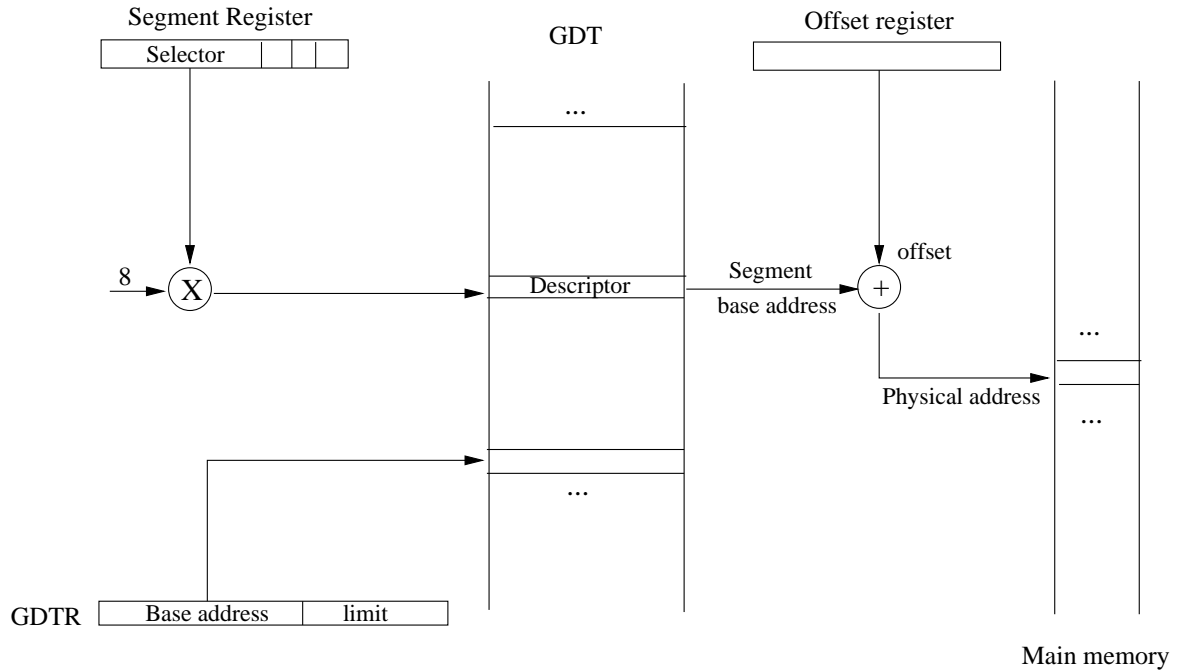


Figure 3.7: GDT-based generation of physical addresses

segment descriptor in the GDT. To select an LDT, the programmer must execute the *lldt* instruction to load the *lldt* register with a selector value just like any other segment register, in turn this selector is used to access the GDT and fetch the base address, the limit and access rights needed to access the LDT.

To eliminate address translations when accessing the LDT, the segment selector, base linear address, limit, and access rights of the LDT are stored into an extended *ldtr* register as shown in Fig. 3.8(b) which also describes the invisible part associated to segment registers Fig. 3.8a). Associated with each visible segment register is a cache structure that keeps a record of the base address, the limit and access rights of segments recently accessed. Accessing the cached information address repeated accesses to the GDT are eliminated resulting in a faster address translation process.

As part of the memory management data structures, the *tr* register holds a selector that access a descriptor stored in the GDT. The purpose of the *tr* register is to allow fast switching between tasks in a multitasking system.



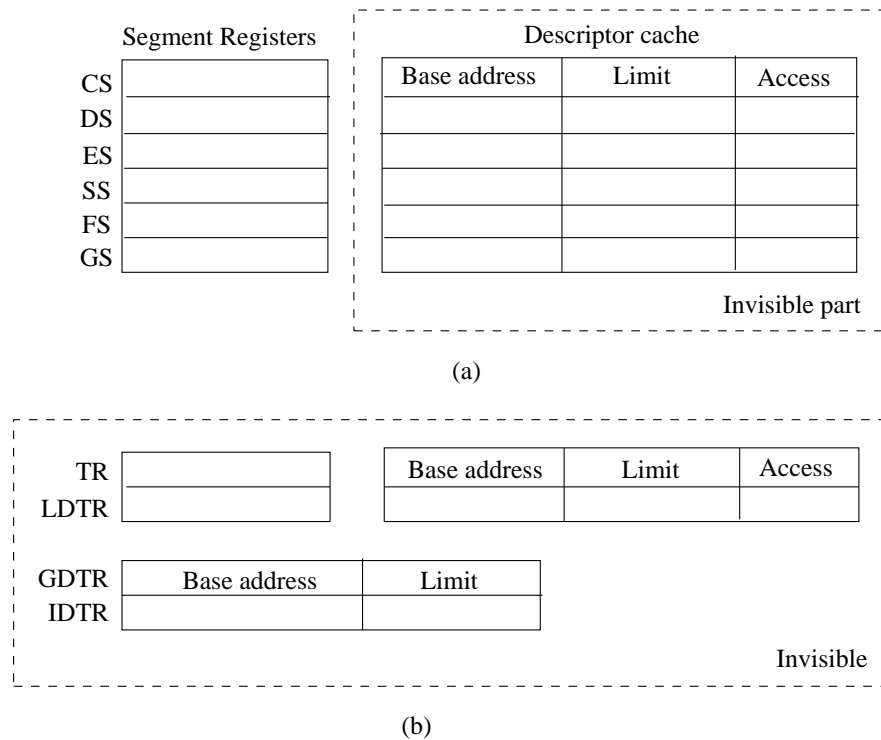


Figure 3.8: Visible and invisible registers in protected mode

## 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 an 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]
```

```
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 processor 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

The first step to develop applications in assembly language is to open a DOS window from a window-based environment, To assemble code using *nasm* type the command line:

```
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. To generate binary files without the linking step assemble source code into *.com* files. Nasm can generate *.com* files using the 16-bit assembler with the following command:

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

The assembly process can generate object files that can be linked into a single binary file. Nasm creates object files by 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*. The linker tool *ALINK* was created by Anthony Willians. The code 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 a "*..start*" entry point to indicate to the linker where code execution is to begin. The remaining modules are external and do not require additional entry points.

To assemble 32-bit applications a separate *nasm* tool is used to generate intermediate object files. The command line that calls for *nasm32* also must specify the assembly code:

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

which it will create the object file *myprogram.o*. The object file thus generated can be combined with other object code and/or "C" source code to generate an executable via a 32-bit compiler such as *linux* or *djgpp*. Using *djgpp* 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 *djgpp* command line is used to produce an executable file *myexe.exe*:

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

### 3.5.1 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 Exercises