Chapter 3

Assembly Programming Issues

3.1 Introduction

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, respectively. 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 mode*, in reference to "real" direct memory addressing, provides the user with the programming environment (software model) native to the Intel 8086/88 processor. All Intel processors boot in real mode before they are switched into a different working mode. The *protected mode* is the native operating mode available for users of IA32 machines. In this mode, all instructions and state of the art architectural features are available, providing the highest performance and capability. A third mode of operation available for users is a quasi-operating mode known as *virtual-8086 mode*. This mode allows the processor the execution of 8086 software in a protected, multitasking environment.

3.2.1 Real Mode

In real 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 the segment part is shifted four places to the left and added to the offset part:

$$Physical\ address = 2^4 \times segment + offset \tag{3.1}$$

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: 09280h + 0022 = 092A2h, as shown in Fig. 3.1. Note that right shifting the segment value four positions to the left is equivalent to adding four zeroes on the right creating the physical address of the very first location (offset zero) within the segment.

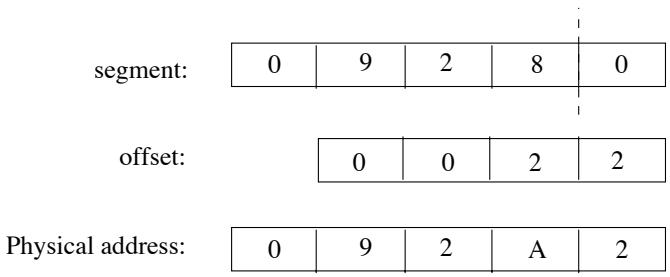


Figure 3.1: Alignment of segment and offset

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 of registers as 'pointers' to memory segments. *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 paired with SI or DI registers will always point to the section in memory where data resides; the data segment in memory 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, are 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.

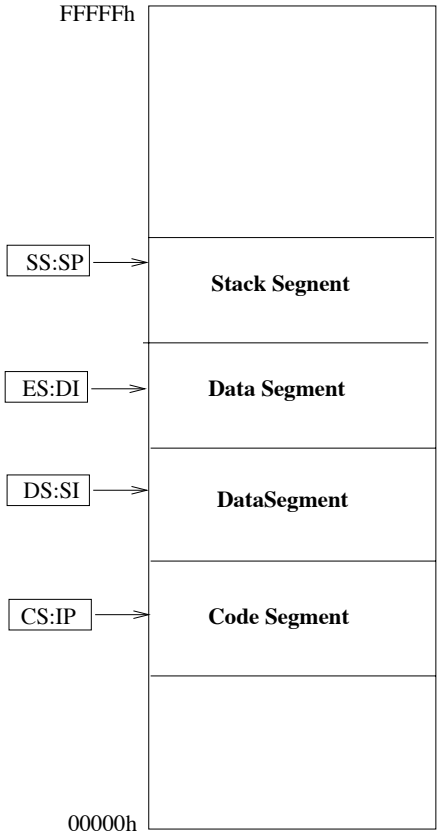


Figure 3.2: Placement of memory segments

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 in response 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. The contents of a segment register are divided into three fields as shown in Fig. 3.3: a 13-bit *selector* field points to the descriptor in table, a 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. A base address field of 24 bits specifies the starting location of a segment anywhere in memory within a range of 16 Megabytes. A 16-bit field specifies the limit of the segment and corresponds to the last location in the segment whose maximum size is 64 Kbytes. This is not the case for the 80386 processor which addresses segments of size up to 4 Gbytes (2^{32}). Therefore, the 32-bit base field in the descriptor indi-

cates that a segment can be placed anywhere in this range. The limit field for the 80386 descriptor is 20 bits in length indicating the maximum size of a segment is 1 Megabyte if the granularity bit G = 0 (the one-bit *G* field in the descriptor). If G =1, the limit field specifies a multiple of 4 Kbytes resulting in a segment size from 4 Kbytes to 4 Gbytes. The *D* field (bit 6) indicates the default length for effective addresses and operands referenced by the instructions in the segment. If this bit 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 is reserved and should be set to 0. Bit 4, $A = 0$ will indicate that the segment is available. The formats of the descriptors used for the 80286 and 80386–Pentium processors are shown in Fig. 3.4.

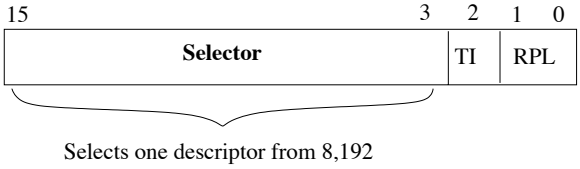


Figure 3.3: Contents of a segment register in a protected-mode operation

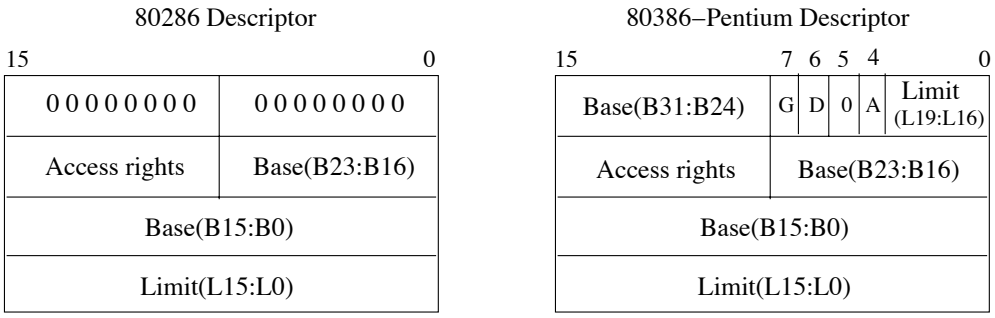


Figure 3.4: Descriptor formats for the 80286/386/486/Pentium processors.

Both descriptors in Fig. 3.4 show an 8-bit *access rights* field which is described in Fig. 3.5. The two-bit privilege level field is used to activate protection mechanisms for segments containing critical software. The highest priority level (00) of protection is assigned to critical software. Lower priority protection levels are assigned to service programs and applications.

To illustrate the use of addressing segments in protected mode assume the segment register *ds* contains the value 0010h which points to a descriptor as shown in Fig. 3.6. The data shown (50h) for the Pentium descriptor indicates that with G=0, the maximum size of the segment is one Megabyte; a bit D=1 indicates that

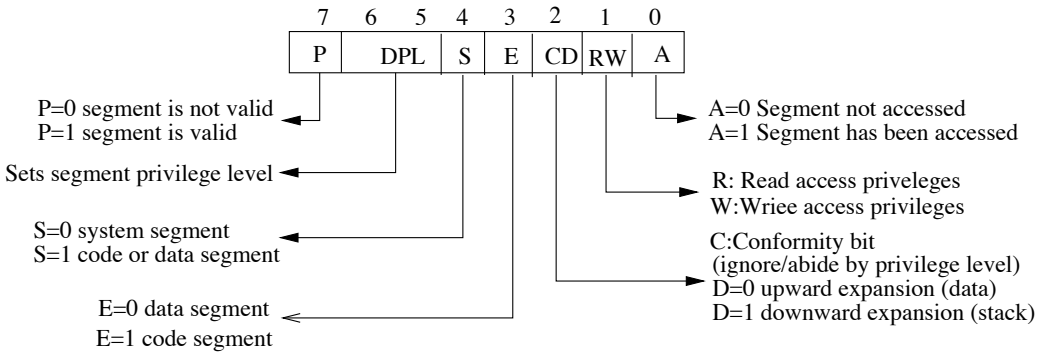


Figure 3.5: Access rights for 80286–Pentium segments.

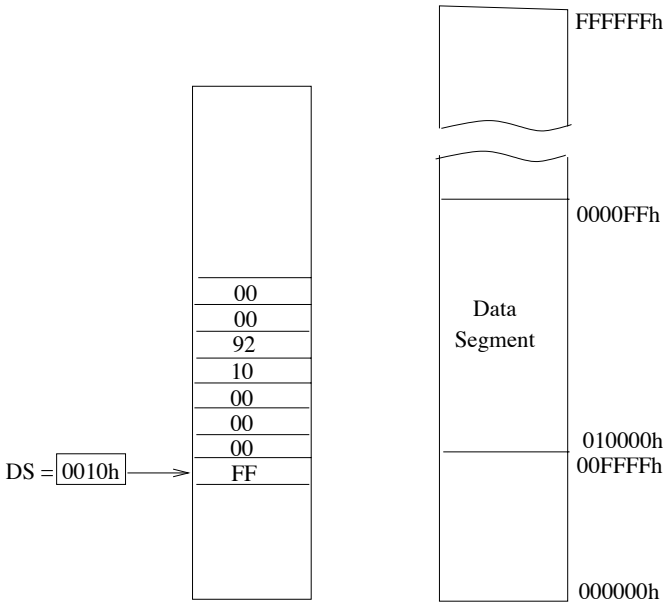
32-bit addresses are used (protected mode), and the availability bit A=1 shows that the segment is not available.

The access rights field has an entry value of 93h in both descriptors; the bit P=1 indicates the segment is valid; DPL = 00 shows the lowest level of protection; with S=1 the segment is a data or a code segment; E= 0 shows that a data segment is addressed with upward expansion (D=0) with read and write access (RW=1) and that it has been recently accessed (A=1).

3.2.3 Virtual 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 creating a virtual machine with all the 8086 real-mode functionality. Multiple 8086 virtual machines can be created and hence, multiple DOS-based applications can be executed concurrently. Each virtual machine has its own one Megabyte addressing space that can be placed anywhere in physical memory.

A virtual-8086 mode is entered from a protected mode and controlled by the hosting operating system supporting a multitasking switching environment. Launching a virtual 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 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.



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) Peintium Descriptor

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

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 *tgdtr* 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 corresponding descriptor tables. The *gdtr* 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 zero 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 *gdtr* 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 *gdtr* 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 point to the correct descriptor. For illustration consider the following examples.

Example: Assume that the contents of the *gdtr* are 003A00000FFFh, the contents of the CS register are 2128h, and an offset value of 0000A123h is stored in *eip* . What is RPL, TI? Determine the address of the descriptor in the GDT, and the size of the GDT.

Solution: Note that CS = 0010 0001 0010 1000, indicates that RPL = 00 and since TI = 0 then the descriptor is in the GDT. The address d of the descriptor is obtained as follows:

$$\begin{aligned} d &= 0000010000100101 \times 8 + 003A0000h \\ &= 0010000100101000 + 003A0000h \\ &= 2128h + 003A0000h \\ &= 003A2128h \end{aligned}$$

The size of the GDT is given in the *gdtr* as OFFFh bytes.

Example: Assume that the descriptor in the GDT accessed in the previous example contains a segment base address of 00123456h, calculate the final physical address referenced by the contents of *cs:eip*.

Solution:

$$\begin{aligned} physical\ address &= 00123456h + 0000A123h \\ &= 0012D579h \end{aligned}$$

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 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

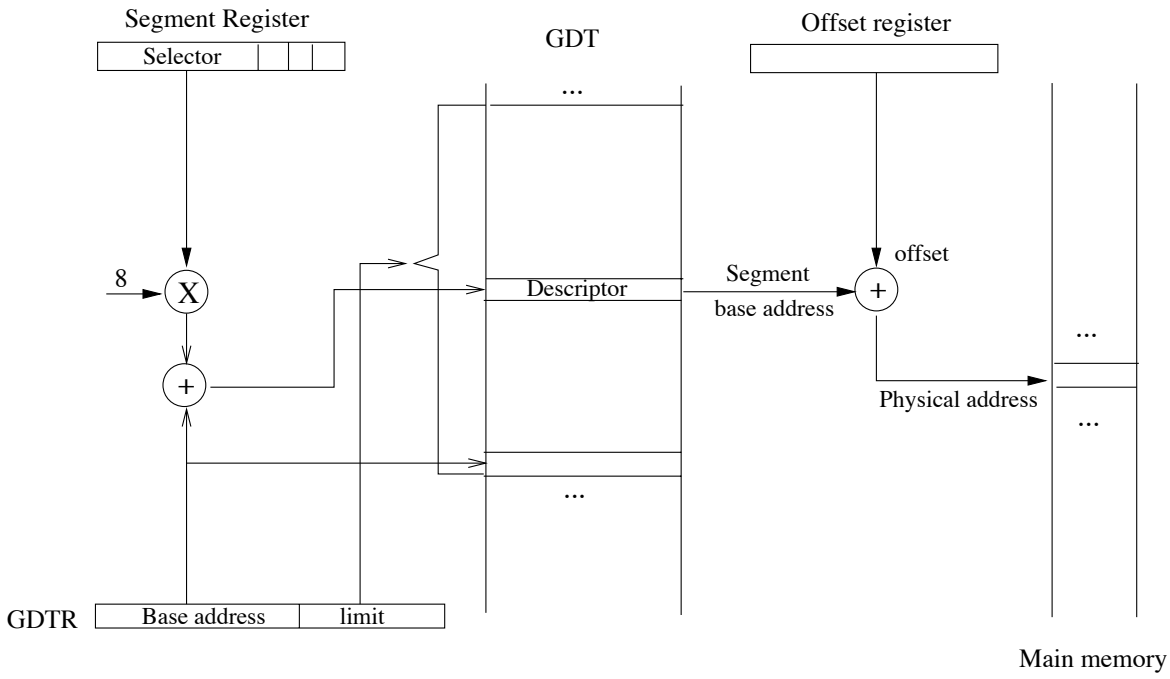


Figure 3.7: GDT-based generation of physical addresses

extended *ldtr* register as shown in Fig. 3.8(b) which also describes the invisible part associated to segment registers Fig. 3.8(a). 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, 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.

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*. To illustrate consider the code shown in example 1 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

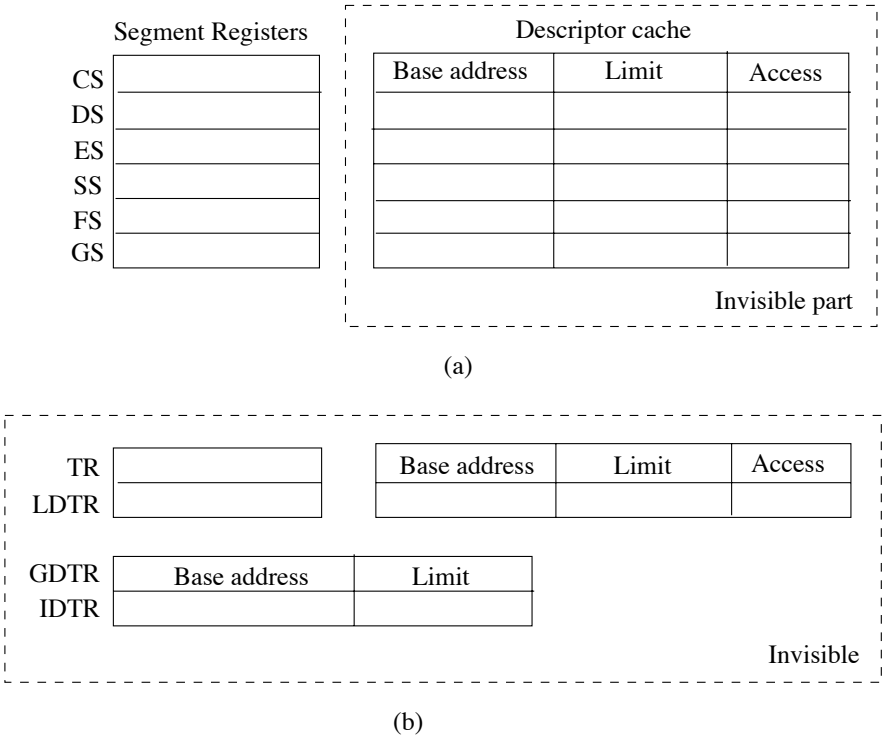


Figure 3.8: Visible and invisible registers in protected mode

generate an executable (binary) version of the program. The assembly command shown is intended to create an intermediate file with *.obj* extension. 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:

Example 1:

```
; 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 of the stack segment is not necessary for the correct execution of the program, however, *alink* will issue a warning and/or generate an erroneous output if the segment of type *stack* is not found.

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 100h which is the offset 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 implementation of the *hello, world* program using a flat model approach is shown in Example 2. 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 0100*h* 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.

Example 2:

```
; 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, '$'
```

3.4.1 Instruction Formats

The general instruction format for Intel processor is as follows:

[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 generates 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 and is now available at *http://alink.sourceforge.net*. In general to assemble n different object files *prog1, prog2, . . . ,progn*, 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 and must be 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* myprogam.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

The code shown in Example 3 implements again the *hello world!* program in 32 bits. This small program is assembled using the command *nasm32 -f coff first.asm*. The *coff* is the compiling option that the *djgpp* linker reads. The object file *first.o* is generated and can be used in the linking step. The *djgpp* provides the linking step and generates an executable program. The command line *gcc -o first first.o* links the single object file provided and generates an executable named *first*.

Example 3:

```
; To assemble use the 32-bit nasm assembler: nasm32 -f coff first.asm
; this will produce: first.o
```

```
; To link use djgpp: gcc -o first first.o

BITS 32
global _main

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

section .text
_main:
              mov  ah, 9
              mov  edx, msg
              int  21h
              ret
```

The first line *BITS 32* directs *nasm* to use 32-bit instructions. Now the addressable space is a huge flat 32-bit space. The second line *global _main* is actually a directive to the linker to locate a global label *_main*.

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. For any global declaration, *nasm* will record its name and address into the object file it creates. Object files with a *external* declarations are interpreted as references, or unresolved links, to other object files, variables, etc., The linker takes all the object files named in the command line, matches up the names, and resolves the addresses of the global procedures or variables into the code.

3.6 Exercises

1. A typical stack segment declaration is as follows:

```
segment stackno1 stack
        resb 100h
stacktop:
```

   (a) What are the hexadecimal contents of SP when the program begins?

   (b) What is the maximum number of words that the stack may contain?

2. Determine the memory locations addressed by the following combinations of segment:offset registers:

   (a) DS:DI = 1000:2000h

   (b) DS:SI = 2000:1002h

   (c) SS:BP = 2300:3200h

   (d) SS:SP = 2900:3A00h

3. The following hexadecimal values represent a physical 20-bit address. Provide a possible representation using 2 16-bit values in the segmented model, i.e., *segment:offset* format:

   (a) 02008h

   (b) BF00h

   (c) 4200h

   (d) BFA2h

4. What is a selector, where is it found, and what is its purpose?

5. Explain the purpose of a segment register in protected mode memory addressing.

6. What information is contained in a segment descriptor?

7. Explain how a linear address is generated using segment selectors and descriptors.

8. If the limit and base in the global descriptor table are 0FFFh and 00210000h, respectively.

(a) What is the starting and ending address of the descriptor table?

(b) What is the size of the table in bytes?

(c) How many descriptors can be stored in the table?

9. If DS = 0305h in a protected mode environment derive which entry, what table, and requested privileges are selected?

10. How is the local descriptor table addressed in the memory system?

11. Three sections can be identified in a real-mode assembly program: data, stack, and code sections. For each section indicate at least one possible pair registers that can be used to addressed the corresponding section in memory when the program is loaded.

12. A virtual 8086-mode corresponds to a task controlled by the operating system. Explain how the operating system establishes a 8086 virtual machine.