Chapter 5

High-Level Language Interface

5.1 Introduction

Main programs, subroutines, and sub-procedures interact by exchanging parameter values and results. Parameter values can be passed using registers, shared memory, or the stack structure. A large part of this chapter deals with the standard C calling conventions that can be used to interface assembly subprograms with C programs. Calling programs not only pass values but often pass the addresses of data (i.e. pointers) to allow subprograms to access data in memory. Applications involving C programs are compiled with *djgpp* which is a 32-bit compiler designed for virtual DOS environments. The use of Dos Protected Mode Interface (DPMI) libraries is introduced to provide a protected mode environment for applications developed using real-mode software.

5.2 Procedure calls and returns

The IA32 family provides two instructions that rely on the stack to make calls to sub-procedures quick and easy. The *call* instruction *pushes* the address of the next instruction onto the stack and makes an unconditional jump to a sub-procedure address. The *ret* instruction *pops off* an address into the *eip* register and the *cpu* continues the execution of the calling program. When using these instructions, it is very important to manage the stack correctly so that the right address is popped off by the *ret* instruction.

5.2.1 The *call* instruction

Prior to branching to the first instruction of the called procedure, the *call* pushes the *eip* register onto the stack. The address just pushed points to the instruction the calling procedure should resume execution following a return from the called procedure. The last instruction in the called procedure is a *ret* instruction that pops the the return address from the stack back into the *eip* register. Execution of the calling procedure then resumes. The processor does not keep track of the location of the return instruction. It is up to the programmer to insure that the stack pointer is pointing to the return address on the stack, prior to issuing a *ret* instruction.

When the *call* instruction transfers control to a procedure within the current code segment, it is referred to as a *near* call. The *call* instruction can also transfer control to procedures in a different code segment; these calls are referred to as *far* calls. The following formats are supported by *nasm*:

call reg/mem	$;[ss:(e)sp] \leftarrow (e)ip;$
	$;(e)ip \leftarrow reg/mem_{16,32}$
call imm	$;[ss:(e)sp] \leftarrow (e)ip;$
	$;(e)ip \leftarrow (e)ip + imm_{16,32}$
call far mem	$;[ss:(e)sp] \leftarrow cs:(e)ip;$
	$;cs:(e)ip \leftarrow seg:offset_{16,32}$
call imm:imm	$;[ss:(e)sp] \leftarrow (e)ip;$
	$;cs:(e)ip \leftarrow imm_{16}:imm_{16,32}$

Near calls: The first form *call reg/mem* is a *near* call instruction. The target address is loaded into *(e)ip* out of memory or a register encoded in the instruction. The form *call imm* is also a near call where an immediate offset is loaded into *(e)ip* as shown. The *imm* value is a relative offset specified in the instruction and as such is added to the current contents of *(e)ip*. This is a 3-byte long call for 16-bit applications and a 5-long byte for 32-bit applications; one byte contains the *opcode* and the *displacement* occupies the remaining bytes. Recall that this is also the format for near unconditional jumps. The displacement is a relative offset specified in the fetched instruction such that the target address is calculated with respect to the current contents of *(e)ip*:

$$target\ address = (e)ip + displacement$$

A 16-bit two's complement displacement allows a jump in the range of ±32 Kbytes. Displacements with 32 bits have a jump range of ±2 Gbytes; in this case after saving *eip* into the stack, control to the procedure is transferred by adding to the current contents of *eip* the 32-bit displacement specified in the instruction. From the programmer perspective, the processor does the following: (see Fig. 5.1):

1. Pushes the current value of the *eip* register onto the stack.
2. Loads the offset of the called procedure into the *(e)ip* register.
3. Begins execution of the called procedure.

Far calls: The *call far mem* form executes a far call by loading the destination address from a memory location. This format specifies a far indirect call with a pointer determined based on the 3-byte information provided in the instruction. In real-mode the effective address consists of four bytes that are loaded into *cs:ip*; in protected mode six bytes are loaded in to *cs:eip*.

The format *call imm:imm* is a direct far call as it loads the effective address from the instruction itself as noted in the comment line. This call instruction requires 5 bytes with the first byte containing the opcode and the remaining four bytes containing a far pointer to the procedure. For 16-bit applications bytes 2 and 3 are loaded into *ip* and bytes 4 and 5 are loaded into *cs*. For 32-bit applications a 32-bit offset is loaded into *eip* and two additional bytes are needed to load the segment selector into *cs*. As shown in Fig. 5.1, from the programmer perspective, the processor performs the following actions:

1. Pushes the current value in *cs* and *(e)ip* registers onto the stack.
2. Loads the segment value (or selector in protected mode) needed to find the address of the called procedure into the *cs* register.
3. Loads the offset of the called procedure into the *eip* register.
4. Begins execution of the called procedure.

5.2.2 The *ret* instruction

The *ret* instruction also allows near and far returns to match the near and far *call* instructions. In addition, the *ret* instruction allows a program to increment the stack pointer on a return to release parameters from the stack. The number of bytes released from the stack is determined by an optional argument n to the *ret* instruction. The following formats are supported by *nasm*:

ret	$;(e)ip \leftarrow [ss:(e)sp]_{16,32}$
ret n	$;(e)ip \leftarrow [ss:(e)sp]_{16,32}; (e)sp + n$
retf	$;cs:(e)ip \leftarrow [ss:(e)sp]_{16,32}$
retf n	$;cs:(e)ip \leftarrow [ss:(e)sp]_{16,32}; (e)sp + n$

Near returns: When *ret* executes a *near* return, it pops only *ip* or *eip* from the stack and transfers control to the new address. The form *ret n* increments the stack pointer by an *n* number of bytes after popping the return address. The *cpu* actions in the execution of a *ret* instruction are summarized as follows:

1. Pops the top-of-stack value (the return instruction pointer) into the *(e)ip* register.
2. If the *ret* instruction has an optional *n* operand then increments the stack pointer by the number of bytes specified by *n* to release parameters from the stack.
3. Resumes execution of the calling procedure.

Far returns: When *retf* executes a far return, it pops *ip/eip* followed by *cs*. The second form *retf n* will increment the stack pointer a number of bytes given by the argument *n*. When executing a far return, the processor does the following:

1. It pops the top-of-stack value (the return instruction pointer) into the *(e)ip* register, and pops the segment selector for the code segment being returned to into the *cs* register.
2. If the *ret* instruction has an optional *n* argument then increments the stack pointer by the number of bytes specified with the *n* operand to release the stack space taken by the call parameters.
3. The *cpu* resumes execution of the calling procedure.

5.3 Passing Parameters

Unlike HLL call statements, call instructions in assembly do not provide a list of arguments in the instruction itself to pass along to the sub-procedure being called. There are mainly three ways by which a caller procedure can pass parameters to a callee procedure:

1. **Use of registers**. Parameter values or addresses are stored into registers before the call is made.
2. **Use of global variables**. The caller must declare as *global* all variables used to pass values or addresses.

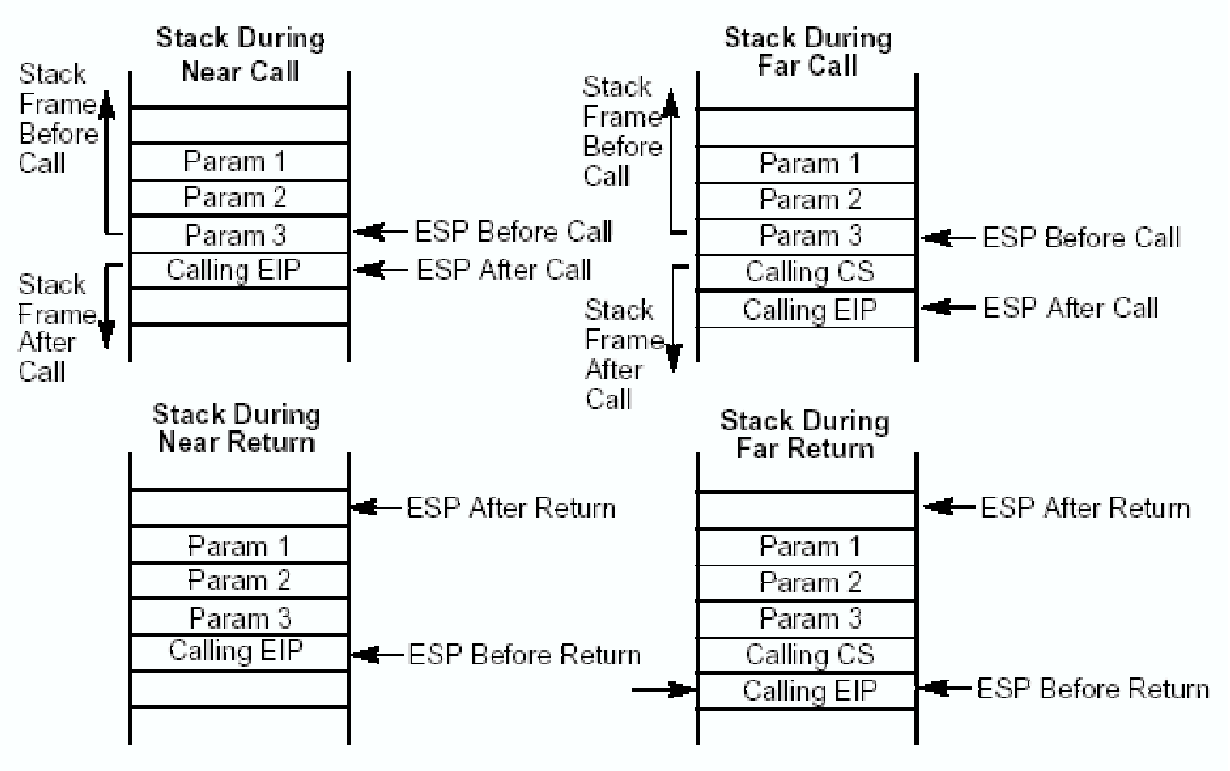

Figure 5.1: Stack operations on Near and Far Calls.

3. **Use of the stack**. The caller procedure executes one push instruction for each parameter passed as illustrated in Fig. 5.1.

A common strategy to access parameters in the stack is to capture the contents of the *esp* register into *ebp* register and then access any location in the stack relative to the contents of *ebp*. However, since the caller program is also using the *ebp* register, its contents must also be saved into the stack before overwriting it. Therefore, the first set of instructions to be performed by the callee program are the following:

```
subprocedure:
    push ebp
    mov ebp, esp
    ...
```

5.3.1 Examples

In this section four examples are discussed to illustrate the use of *call* and *ret* instructions. The first example illustrates the use of registers to pass a pointer to a simple DOS-based display function. The second example calls a sub-procedure to access video memory to display a null-terminated string; the pointers to the string are passed via registers. The third example calls the string display sub-procedure but the expected pointers are retrieved from the stack. The fourth example uses the console to display strings but relies on C-calls.

Example 1: Display $-terminated strings using a sub-procedure. The only parameter needed is the offset of the message to display; since the offset is needed in register *dx* it is stored there before the call is made:

```
; Assemble using the 16-bit nasm assembler:
;                        nasm16 -f obj test_calls.asm -o test_calls.obj
; this will produce: test_calls.obj
; to link do: alink test_calls
; it will produce: test_calls.exe

USE16

SEGMENT mystack stack
        resb 100h
stacktop:

SEGMENT data
        msg1 db "hello world", 13, 10, '$'
        msg2 db " ", 13,10, '$'
        msg3 db "this is the next line", 13, 10, '$'

SEGMENT code

..start:
        mov ax, data            ;initialize ds to point to
        mov ds, ax              ;the data segment
        mov ax, mystack
        mov ss, ax              ;initialize ss:sp to access the
        mov sp, stacktop        ;stack segment

        mov dx, msg1
        call display
        mov dx, msg2
        call display
```

```
        mov dx, msg3
        call display

        mov  ax, 04C00H         ; select a DOS function code
        int  21H                ; to return to DOS

display:
        mov ah, 09h             ; select a DOS function code
        int 21h                 ; call DOS service
        ret                     ; to display the string
```

Example 2: Display null-terminated strings in page 0 in video memory. The parameters required are a pointer to the message (*ds:si*) and a pointer to video memory (*es:di*). The corresponding registers are initialized and used to pass these pointers.

```
; this code uses the segmented memory mode
; to assemble do: nasm16 -f obj vmcall.asm -o vmcall.obj
; this will produce: vmcall.obj
; to link do: alink vmcall
; It will produce: vmcall.exe

USE16                           ;equivalent to [BITS 16]

video_seg       equ 0B800h      ;video memory
attribute       equ 47h         ;color attribute

SEGMENT mystack stack
        resb 100h
stacktop:

SEGMENT data
        msg1    DB      'Hello, World !             ',00h
        msg2    DB      '                           ',00h
        msg3    DB      'This is the next line      ',00h

SEGMENT code PUBLIC

..start:
        mov ax, data
        mov ds, ax
        mov ax, mystack
        mov ss, ax
        mov sp, stacktop
```

```
;display messages
        mov si, msg1
        mov di, 10*160 + 2*25           ;offset of first message
        call vmshow
        mov si, msg2
        mov di, 11*160 + 2*25           ;offset of 2nd message
        call vmshow
        mov si, msg3
        mov di, 12*160 + 2*25           ;offset of 3rd. message
        call vmshow
;wait for a character
        mov ah, 8
        int 21h
;return to dos
        mov ax, 4c00h
        int 21h

vmshow:
        mov ax, video_seg
        mov es, ax
        mov ah, attribute
        mov cx, 80
next_char:
        lodsb               ;get the input string byte
        cmp al,00h          ;check for a null character
        je null_ch          ;if null, then quit
        stosw               ;store ax to video memory
        loop next_char      ;loop back to do it again

null_ch:
        ret
```

Example 3: This example is the same as example 2 but illustrates parameter passing using the stack:

```
USE16                           ;equivalent to [BITS 16]

video_seg       equ 0B800h      ;video memory
attribute       equ 47h         ;color attribute

SEGMENT mystack stack
        resb 100h
```

```
stacktop:

SEGMENT data
        msg1    DB      'Hello, World !             ',00h
        msg2    DB      '                           ',00h
        msg3    DB      'This is the next line      ',00h

SEGMENT code PUBLIC

..start:
        mov ax, data
        mov ds, ax
        mov ax, mystack
        mov ss, ax
        mov sp, stacktop

;display messages
        push word msg1          ;store offset of string
        push word 10*160+2*25   ;store offset of video memory
        call vmshow             ;display
        push word msg2
        push word 11*160+2*25
        call vmshow
        push word msg3
        push word 12*160+2*25
        call vmshow
;wait for a character
        mov ah, 8
        int 21h
;return to dos
        mov ax, 4c00h
        int 21h

vmshow:
        push bp
        mov bp, sp
        mov di, [bp+4]          ;load offset of video memory          ;
        mov si, [bp+6]          ;load offset of string
        mov ax, video_seg
        mov es, ax              ;initialize video segment
        mov ah, attribute
        mov cx, 80
next_char:
        lodsb                   ;get the input string byte
```

```
        cmp al,00h              ;check for a null character
        je null_ch              ;if null, then quit
        stosw                   ;store ax to video memory
        loop next_char          ;loop back to do it again

null_ch:
        pop bp
        ret
```

Example 4: This example illustrates the use of C-functions in a 32-bit protected mode environment to display messages (null-terminated strings) using the system call *printf*. The only parameter passed using the stack is the offset of the message to display. The mechanism to interface with C programs is covered in the next section.

```
; This program illustrates the use of system calls.
; Assemble using the 32-bit nasm assembler
;                   nasm32 -f coff wprintf.asm
; this will produce: wprintf.o
; to link under djgpp do: gcc -o wprintf wprintf.o

SEGMENT .text
        global _main
        extern _printf

_main:
        push dword msg1         ;store pointer to message
        call _printf            ;display it
        add esp,4               ;restore stack pointer

        push dword msg2
        call _printf
        add esp,4

        push dword msg3
        call _printf
        add esp,4
        ret

SEGMENT .data
        msg1:   db      "Hello, World !             ", 10, 0
        msg2    db      "                           ", 10, 0
        msg3    db      "This is the next line      ", 10, 0
```

As shown in example 4 some system calls such as *printf* and *scanf* allow a varying number of arguments. As the function is called from assembly parameter values are pushed into the stack in the inverse order in which they would be listed. After the parameters are used by the function, the top of the stack is restored by adding to the *esp* register the space in bytes required by the arguments passed.

5.4 Interfacing C and Assembly Language

As shown in the previous section an alternative to passing parameter values is the use of the stack. This is what C programs do. The use of global variables to pass parameter values involve the allocation of memory to be used during execution. In contrast, the use of the stack allows the use of memory only while the procedure called is active. Another advantage of using the stack is the flexibility of storing the status of the calling program. As a result of such flexibility procedures are made *reentrant*, i.e., they can be called any time at any place even *recursively*.

The section of the stack assigned to each call is referred to as the *stack frame*. It is a fixed block of memory within the stack used for parameters, return address, local variables and register storage. Any time a procedure is called, its stack frame is pushed into the stack; when finished, the stack frame is popped off the stack. Since a stack frame is assigned to each call and not to the procedure itself, *recursive (nested)* calls are possible.

5.4.1 Interfacing with 16-bit programs

In the following description, the words *caller* and *callee* are used to denote the programs doing the calling (a C-program) and the program which gets called (an assembly program), respectively. Also, C compilers require that all global variables and procedures have a leading underscore appended to the global symbol. The *elf* compiling specification does not require this *renaming* of global symbols. The caller pushes the function parameters onto the stack, one after another, in reverse order (right to left, so that the first argument specified to the function is pushed last). Since this mechanism is used by C calls, it is referred to as the *C-convention* to parameter passing and it is illustrated in Fig. 5.2*a*.

After placing parameters into the stack, the caller program executes a *call* instruction to pass control to the callee program. The callee receives control, and saves the contents of *(e)sp* into *(e)bp* to locate parameters in the stack frame relative to the contents of *(e)bp*. Hence, the callee, must push the previous value of *(e)bp* as shown in Fig. 5.2*b*. The following template illustrates the entry code and the exit

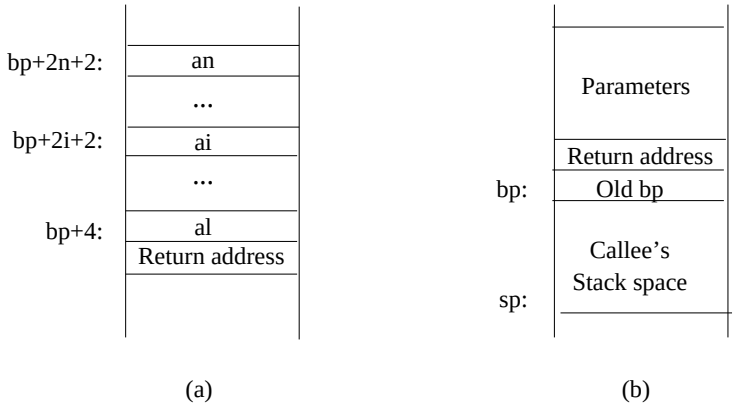

Figure 5.2: (a) C Parameter convention, (b) Callee's stack frame

code of a 16-bit sub-procedure *myfunc*:

```
global myfunc

myfunc:
        push bp
        mov bp, sp
        ...
        code for myfunc
        ...
        mov sp, bp           ;restore the value of sp
        pop bp
        ret
```

After pushing *bp* into the stack and copying the value of *sp* into *bp*, *myfunc* can then access parameters relative to *bp*. The word at $bp + 0$ holds the previous value of *bp* as it was pushed last; the next word, at $bp + 2$, holds the offset part of the return address, pushed implicitly by the *call* instruction. For a near *call* the parameters start at $[bp+4]$ as in Fig. 5.2*a* where the value of the ith parameter is at $bp+2i+2$. For a far *call* the return address requires the segment part stored at $bp + 4$, and parameters begin at $bp + 6$; therefore, in general the ith parameter is located at $bp + 2i + 4$.

A callee program can use the stack by pushing local variables and decrease *sp* further; thus, saved local variables and registers are accessible at negative offsets from *bp*. To return a value to the caller, a common practice is to place the value in *al, ax* or *dx:ax* depending on the size of the value.

Once the callee program has finished processing, it restores *sp* from *bp*; this step is necessary if local variables have been pushed into the stack. The callee then pops into *bp* its previous value, and returns via *ret* or *retf* depending on the memory model used. When the caller regains control, the function parameters are still on the stack, so it typically adds an immediate constant to *sp* to remove them (instead of executing a number of slow *pop* instructions).

Example 5: To illustrate the C-passing passing parameter convention, consider the stack frame of a 16-bit video display function callable from a C program.

video(string ptr, row, column, attribute, length, page)

The function *video* transfers a null-terminated string into video memory for display at the coordinates given by the *row* and *column* parameters. The string will be displayed with the selected color *attribute*. Furthermore, the *row, column* and *page* will be used to calculate the corresponding offset in the video buffer in text mode. The set of parameters must be available on the stack in the order shown in Fig. 5.3.

Note that a far call is assumed as both the *segment* and the *offset* part of a physical address where the string buffer is located are passed. Likewise the reference to the next instruction is passed using both the segment and offset part.

5.4.2 Interfacing with 32-bit programs

Saving and loading parameters from the stack for 32-bit applications, is basically the same mechanism described for 16-bit applications. The ith entry within the parameter stack space changes to $ebp + 4i + 4$ to take into account that values now take 4 bytes. A callee procedure must preserve the values of *ebx, esp, ebp, esi* and *edi* as well as the contents of the segment registers *cs, ds, es* and *ss*; the values in these registers should be the same before and after the *call* is executed because under protected sub-procedures should not alter the contents of segment registers. Values are returned in *eax* if they are 32-bit or smaller in size. Values are returned in *edx:eax* if they require a 64-bit representation. Strings, structures, and other items above 32 bits in size are returned by reference, i.e, a pointer is returned in *eax*. As shown in example 4, C-library functions can be called directly from assembly procedures. C-libraries, however, may alter the state of the caller assembly program; therefore, to preserve the values in registers in use they must be saved into the stack before the library function is called. Fig. 5.4 shows the stack frame associated to a call under a 32-bit flat model programming environment. Notice that there is no need to pass the segment component of pointers.

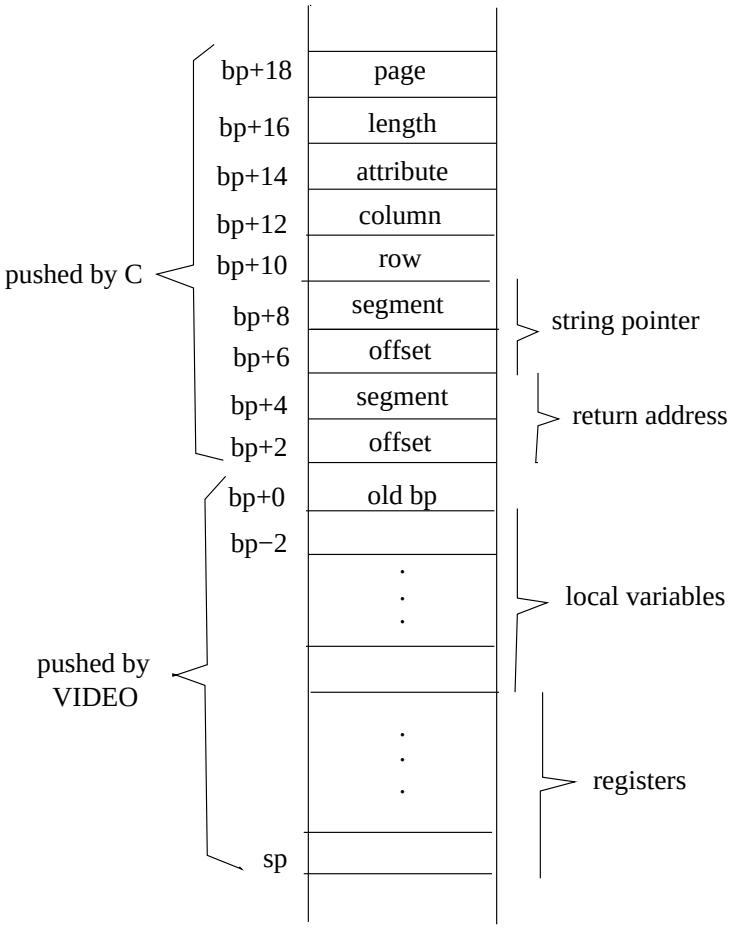

Figure 5.3: Stack frame for the call to VIDEO in real mode

Example 6: The sub-procedure *vmshow* previously coded, is re-designed to be callable from a C-program in a protected mode environment. Both the C-code and the assembly code are listed:

```
/* to compile and link: gcc -o dvtest dvtest.c vmshow.o */

#include <stdio.h>
#include <conio.h>
#include <dpmi.h>                /* we need this for the DPMI wrapper functions */
#include <string.h>

extern void vmshow(char *, int);

/* This will hold the selector we need to access the text mode
   video memory buffer. */
```

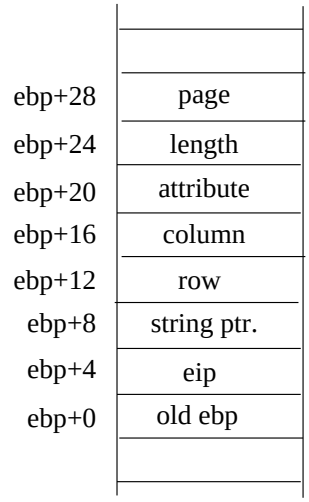

Figure 5.4: Stack frame for the call to VIDEO in protected mode

```
int vm_buffer = -1;

int main(void){
int row;
int col;

/* Set up our frame buffer descriptor, and store the selector to use */

   vm_buffer = __dpmi_segment_to_descriptor(0xb800);
   if(vm_buffer < 0) return 1;

   clrscr();
   row = 10; col = 25;
   vmshow("Hello world", 160*row+2*col);
   row = 11;
   vmshow(" ", 160*row+2*col);
   row = 12;
   vmshow("This a third line", 160*row+2*col);

   getchar();           /*type any character to terminate the program*/
   clrscr();

/* The descriptor must be released because it takes up resources */

   __dpmi_free_ldt_descriptor(vm_buffer);
```

```
   return 0;
}

void clrscr(void){
int row;
int col;
        for(row=0; row<25; row++)
                for(col=0; col<80; col++) vmshow(" ", 160*row+2*col);
}

; vmshow.asm
; to assemble do: nasm32 -f coff vmshow.asm -o vmshow.o

attribute       equ 47h         ;color attribute

global _vmshow
extern _vm_buffer

SEGMENT .text
_vmshow:
        push ebp
        mov ebp, esp
        push edi
        push esi
        push eax
        push es
        push ecx
        mov esi, [ebp+8]        ;offset message
        mov es, [_vm_buffer]
        mov edi, [ebp+12]
        mov ah, attribute
        mov ecx, 80
next_char:
        lodsb                           ;get the input string byte
        cmp al,00h                   ;check for a null character
        je null_ch                     ;if null, then quit
        stosw                            ;store ax to video memory
        loop next_char         ;loop back to do it again
null_ch:
        pop ecx
        pop es
        pop eax
```

```
pop esi
pop edi
pop ebp
ret
```

As in example 3 *vmshow* will display a message at the coordinates *row, col* at page 0. A stack frame is setup to access at $[ebp + 8]$ the offset where the message is stored, and at $[ebp + 12]$ a constant for the offset within page 0. Note that in protected mode there is no need to initialize the *ds* register needed to access the message in real mode. However, this is not the case for the video text buffer which is located at $B800h$ in real-mode memory space and can not be accessed directly in protected mode. To ensure access to video memory, *dpmi* services are invoked.

The Dos Protected Mode Interface (DPMI) is a set of library calls that allow *dos* programs to access the extended memory of IA machines while maintaining system protection. Any operating system that currently supports virtual *dos* sessions should be capable of supporting DPMI without affecting system security. Note that DPMI services are only available to protected mode programs.

Usually memory-mapped devices or absolute addresses below the 1Megabyte mark can not be accessed directly. This is because the combination *segment:offset* does not apply in protected mode. A special selector is required to gain access to a device or an absolute address. To create such a selector in example 5, the *djgpp* library function *__dpmi_segment_to_descriptor* with the $0xB800$ argument is used. An *int* type variable *vm_buffer* is declared and used to store the selector needed by *vmshow* to initialize *es* with an offset already popped from the stack frame at $[ebp + 12]$. The remaining code is similar to the sub-procedure shown in example 3. An additional change is that the stack frame is used to store and restore the registers used by the sub-procedure and thus maintain the integrity of the calling program.

5.5 Exercises

1. With a one-line statement explain the following terms:
   (a) stack frame
   (b) call imm:imm
   (c) retf n
   (d) dpmi
2. Define the term "stack frame" and briefly explain when and how is it formed.

3. Combine words and register-level transfer notation to explain how the *cpu* processes the instructions:

   (a) A near *call*.

   (b) A far *call*, and

   (c) *ret*.

4. Suppose a program contains the following code lines:

   ```
   call subproc1
   mov ax, cx
   ```

   If the instruction *mov ax, cx* is stored at 0A23:103A, the current contents of SP = 0200h, and *subproc1* is a *near* procedure that begins at 0A23:10C0h, then:

   (a) Determine the contents of CS:IP, and SP just after *subproc1* is executed

   (b) Determine the word stored into the top of the stack.

5. Consider the following prototype of a function to be called from a *C* program and a 32-bit flat model environment:

   void check(string1 ptr, int num, string2 ptr)

   Show in a memory diagram of the *stack frame* how each parameter is allocated when the *check* function gets control. Indicate where the "old ebp" is placed as well as all the displacements needed to locate parameters relative to the contents of "ebp".

6. Repeat the previous exercise assuming a ıfar call to the *check* procedure in a real-mode environment.

7. A particular application requires to display some text in video memory. The display coordinates selected are row = 10, and column = 20. Using page seven in video memory the program must calculate the exact initial physical location in video memory to begin display.

   (a) Provide the segment part and the offset part of the desired address in video memory.

   (b) Calculate the actual physical address,