Assembly Programming Compendium

Jan G. Cornelis David Blinder

October 8, 2019

Abstract

This document serves as background for the computer systems exercise sessions. It is best consulted on a computer given that it contains many links to more detailed information on the World Wide Web. The document is in A5 format, which should make it comfortable for reading on the screen. The best way to print it on A4 paper is probably by putting two pages on one A4 page.

Although efforts have been made to minimize the number of errors and to be as clear as possible, no guarantee can be made about the contents. Therefore, if you encounter errors or some things are unclear you are welcome to send a mail to the author.

Contents

1

Preliminaries

1.1 Computers

Computers consist of a processor, memory and devices for input and output. Memory is made up of a great many hardware components that can be in one of two possible states. Such a component can store a single bit.

Most computers do not operate at the bit level: there is some minimum number of bits that are processed as a single unit. The smallest unit on which the Intel 386 processor can operate is an ordered collection of eight bits: the byte. Memory can be likened to a very large array of bytes. The index of a byte in memory is called its address or its offet. The processor can operate on bytes, but also on words (16 bits) and double words (32 bits). In general we speak of a bit string $\mathbf{b}^n$ that is composed of n bits b_i where $i \in [0, n)$. The leftmost bit b_{n-1} is the most significant bit and the rightmost bit b_0 is the least significant bit:

$$\mathbf{b}^n = b_{n-1}...b_i...b_0$$

The processor provides two types of instructions to manipulate bit strings: logic instructions and arithmetic instructions. The former interpret bits as false or true and will be discussed in section 1.2; the latter interpret bits as 0 or 1 and will be discussed in section 1.3. In the remainder of this document we will exclusively use the symbols 0 and 1. In a logic context 0 should be interpreted as false and 1 as true.

1.2 Logic Operations

Table 1.1 summarizes the truth table for the four common logic operations: NOT, OR, AND and XOR. The processor applies these operations to bit strings by applying them to the individual bits making up the bit strings. Therefore, the logic operations are sometimes called bitwise operations.

If we take $\mathbf{0}^n$ to be the bit string consisting of n zeros and $\mathbf{1}^n$ to be the bit string consisting of n ones, the following equations hold:

Table 1.1: Logic bit operations

a	b	$\neg a$	$a \vee b$	$a \wedge b$	$a \oplus b$
0	0	1	0	0	0
0	1	1	1	0	1
1	0	0	1	0	1
1	1	0	1	1	0

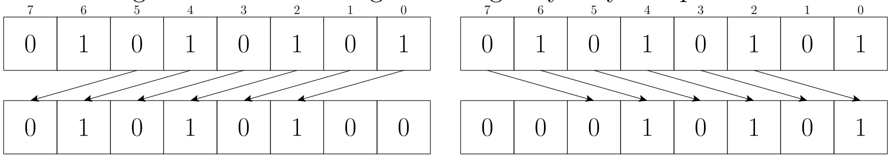
$$\begin{aligned}
\neg\mathbf{0}^n &= \mathbf{1}^n \\
\neg\mathbf{1}^n &= \mathbf{0}^n \\
\mathbf{0}^n \vee \mathbf{b}^n &= \mathbf{b}^n \\
\mathbf{1}^n \vee \mathbf{b}^n &= \mathbf{1}^n \\
\mathbf{b}^n \vee \mathbf{b}^n &= \mathbf{b}^n \\
\neg\mathbf{b}^n \vee \mathbf{b}^n &= \mathbf{1}^n \\
\mathbf{0}^n \wedge \mathbf{b}^n &= \mathbf{0}^n \\
\mathbf{1}^n \wedge \mathbf{b}^n &= \mathbf{b}^n \\
\mathbf{b}^n \wedge \mathbf{b}^n &= \mathbf{b}^n \\
\neg\mathbf{b}^n \wedge \mathbf{b}^n &= \mathbf{0}^n \\
\mathbf{0}^n \oplus \mathbf{b}^n &= \mathbf{b}^n \\
\mathbf{1}^n \oplus \mathbf{b}^n &= \neg\mathbf{b}^n \\
\mathbf{b}^n \oplus \mathbf{b}^n &= \mathbf{0}^n \\
\neg\mathbf{b}^n \oplus \mathbf{b}^n &= \mathbf{1}^n \\
\mathbf{b}^n \wedge \mathbf{b}^n = \mathbf{0}^n &\iff \mathbf{b}^n = \mathbf{0}^n
\end{aligned}$$

It is possible and useful to interpret the logic operations as set operations. A bit string may be interpreted as a set of elements from a universe of n numbered objects: bit b_i will be 1 if the object with number i is present in the set and 0 otherwise. In this case NOT corresponds to the set complement, OR to the set union, AND to the set intersection and XOR to the symmetric set difference.

1.2.1 Shifting

Two operations that are sometimes considered to be part of the logic operations are the left shift and the right shift. Left shifting a bit string $\mathbf{b}^n$ by k bit positions results in a bit string $\mathbf{c}^n$ where c_i is defined as:

Figure 1.1: Left and right shifting a byte by two positions



$$c_i = \begin{cases} 0 & if\ i < k \\ b_{i-k} & if\ i \geq k \end{cases}$$

Right shifting a bit string $\mathbf{b}^n$ by l bit positions results in a bit string $\mathbf{c}^n$ where c_i is defined as:

$$c_i = \begin{cases} b_{i+k} & if\ i < n-k \\ 0 & if\ i \geq n-k \end{cases}$$

Probably shifting is more easily understood graphically: figure 1.1 shows the effect of left shifting and right shifting the byte 01010101 by two bit positions.

1.3 Arithmetic operations

1.3.1 Number representation

Under the unsigned integer interpretation, a bit string $\mathbf{b}^n$ denotes a positive integer $x \in \mathbb{N}$ given by:

$$x = \sum_{i=0}^{n-1} b_i 2^i$$

Under the signed integer interpretation, a bit string that commences with a zero, is interpreted as a positive integer using the above formula. A bit string $\mathbf{c}^n$ that commences with a one, is interpreted as a negative integer $y \in \mathbb{Z}$ and $y < 0$:

$$y = \sum_{i=0}^{n-1} c_i 2^i - 2^n$$

A signed integer is negated by taking the complement (NOT) of its bit string and adding 1 to the result, hence the name “two’s complement notation”. $\mathbf{1}^n$ denotes the number -1. A bit string can be extended without changing its numeric value: for unsigned and positive signed integers zeros must be prepended, for negative signed integers ones must be prepended. For signed integers the more general term "sign extension" is used.

The length of a bit string determines the number of integers it can represent. A bit string of length n can represent all unsigned integers in the half-open interval $[0, 2^n)$ or all signed integers in the half-open interval $[-2^{n-1}, 2^{n-1})$.

1.3.2 Integer Operations

The Intel 386 processor provides instructions to add, subtract, multiply and divide[1]. When using them, you should be aware of the possibility that the result may not be representable. The processor provides two bits that make it possible to determine whether this was the case for some instruction.

Multiplication, division and modulo are expensive operations. They can be performed in a more efficient manner if the multiplier or divider are powers of two. Multiplication by 2^k is equivalent to left shifting the bit string by k positions. Dividing by 2^k is equivalent to right shifting the bit string by k positions. Finally, the remainder when dividing by 2^k is obtained by applying an AND to the dividend and $2^k - 1$.

Note that performing a logic right shift, which assigns zero to the most significant bits, will result in an erroneous result when it is used to divide negative numbers. Therefore, the Intel 386 provides an arithmetic right shift instruction that should be used when operating on signed integers.

1.4 Hexadecimal Notation

Often in assembly code we want to use literal values. Sometimes, these values have a numeric meaning, and it is easiest to use decimal notation. Often, however, the meaning may be closely linked to the bit pattern. In this case, using decimal notation may obfuscate that meaning and it may be better to use binary notation.

Unfortunately, binary notation quickly leads to large literals that are difficult to read and maintain. A more compact notation that is closely related is hexadecimal notation. There are 16 hexadecimal digits that may represent the numbers in the half-open interval $[0, 16)$: the well known decimal digits and the letters A, B, C, D, E and F to represent the numbers from 10 up to 15.

A hexadecimal string of length 1 can represent 16 different values. This is also the number of values that can be represented by a bit string of length 4. A bit string can be converted to a hexadecimal string by replacing every group

[1]The divide instructions will also compute the modulo.

Table 1.2: Mapping hexadecimal digits to nibbles

Hex digit	**Nibble**	**Hex digit**	**Nibble**
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

of four bits by the corresponding hexadecimal digit. Of course, the inverse operation is as easy. Table 1.2 shows the mapping between hexadecimal digits and groups of 4 bits, which are known as nibbles.

Using hexadecimal notation the byte `00000000` is more concisely denoted by `00`. Similarly, `11111111` is more concisely denoted by `FF`.

2

Turbo Assembler

2.1 Introduction

An assembler translates an assembly source file to an object file. A linker combines one or more object files into an executable program. Section 2.2 discusses the contents of a source file. Section 2.3 explains how one or more assembly source files can be converted into an executable. Finally, section 2.4 discusses some important topics related to the assembler.

2.2 Source File Contents

A minimal assembly program is shown in figure 2.1. The program begins with a preamble that contains a number of assembler directives: `IDEAL` enters ideal mode – a mode determines syntax and keywords. `P386` enables assembly of all 386 processor instructions and 387 numeric coprocessor instructions. `MODEL FLAT, C` and the following line specify the memory model as well as the language conventions the assembler should use. The flat memory model addresses all of memory using 32 bit addresses, rather than using segment registers with 16 bit offsets. We will always use the same preamble, possibly followed by macro definitions.

The code segment follows the preamble. It is indicated by the `CODESEG` directive and continues until the `DATASEG` directive, which begins the data segment. The code segment contains the instructions, which must be translated by the assembler, while the data segment is used to declare and initialize global or static variables. The instructions that belong to the code segment will be discussed in detail in the coming chapters. The data segment is discussed briefly in subsection 2.2.1. Finally, the `STACK` directive is followed by a number that states the size of the stack. The stack is typically used to preserve and restore values, to pass arguments to procedures and to store procedure local variables. The stack is discussed in chapter 4. The order of the segments in the source file does not really matter. Furthermore, it is possible to have, for example, a code segment followed by a data segment, followed by another

Figure 2.1: A basic assembly program

```
IDEAL
P386
MODEL FLAT, C
ASSUME cs:_TEXT, ds:FLAT, es:FLAT, fs:FLAT, gs:FLAT

CODESEG

PROC main:
    STI                 ; enable interrupts
    CLD                 ; clear direction flag

    MOV AH, 09H         ; interrupt 21, AH=09H
    LEA EDX, [hello]    ; prints the $ terminated string
    INT 21H             ; whose address is found in EDX

    MOV AH, 4CH         ; interrupt 21, AH=4CH
    MOV AL, 00H         ; terminates the program
    INT 21H             ; with exit code from AL
ENDP main

DATASEG
    hello   DB  "hello world", 13, 10, '$'

STACK 100 H

END main
```

code segment. Readers should decide for themselves whether this is in good taste.

The source file is ended by the line `END main`. The symbol `main` refers to the (address of the) `main` procedure in the code segment and specifies that execution should start with this procedure. Not all assembly source files must contain such an entry point: those that merely define procedures to be used elsewhere can simply end with `END`.

Note that the source code in figure 2.1 contains some literal values: for example `09H` and `21H`. These values represent numbers. Here they are specified in hexadecimal notation as indicated by the `H` suffix. In the absence of a suffix a number is interpreted as being in decimal notation. Whichever representation you chose, a literal number must start with a decimal digit, thus instead of writing `FFH` you should write `0FFH`.

It is also possible to represent character and string literals. In our environment, a character is represented by a byte whose numeric value corresponds to its ASCII code. You do not have to know the ASCII codes by heart: if you write for example `'0'`, the assembler will translate this to the appropriate ASCII code. Finally, note that it is possible to specify byte sequences as a character string. For example, `"abc"` will be translated by the compiler to a sequence of 3 bytes: `97, 98, 99`. Such strings are typically used in the data segment[1].

2.2.1 The data segment

Global or static variables are declared and initialized in the data segment. We will give a few examples. Consider:

```
DATASEG
    LUCKY   DB 13
    KILO    DW 1024
    MEGA    DD 40000000H
    STRING  DB "Hello world", 13, 10, '$'
    CIAO    DB 3 DUP('x'), 13, 10, '$'
UDATASEG
    ARRAY   DD 1000H DUP(?)
```

Each line starts with a symbol that you can use in the instructions of your program to refer to the address in memory of the first byte of the allocated data following the symbol[2]. This symbol is a label. The next symbol specifies

[1]Of course, you could load a string of up to four characters into a register.

[2]This is a simplification. It is translated to an offset from the start of the data segment. Later the linker that links all object files into a single executable determines the address.

the unit size of the data allocation: examples are DB for byte, DW for word (2 bytes) and DD for double word (4 bytes).

The first line allocates a byte of memory and initializes it with the number 13. Furthermore, you can use LUCKY to refer to its address. Similarly, a word and a double word are allocated and initialized.

The label STRING indicates the start of the character string “Hello world”, followed by the characters 13 and 10 – ASCII codes for carriage return and newline – and terminated by a dollar character. We can also allocate and initialize many bytes, words or double words using DUP preceded by the repetition count and followed by the value that should be used to initialize the data. For example, the label CIAO refers to the address of the first of 3 bytes initialized to 'x'.

Finally, we can declare uninitialized variables. This is done with ?. Although you can specify these variables in the data segment, it is better to move them to the uninitialized data segment, which is started by the symbol UDATASEG. The last line declares 4096 double words using a label ARRAY corresponding to the address of the first double word. The Wikipedia article on *.bss* explains the rationale behind an uninitialized data segment.

2.3 Building an Executable

We use *Open Watcom Make*, a proprietary version of *Make* to steer the build process. Information from a text file called MAKEFILE is used to launch *Turbo Assembler* and *Open Watcom Linker* to assemble and link the appropriate files. The following subsections show how this is done for single and multiple source file programs.

2.3.1 Single source file

Copy the MAKEFILE provided in the C:\EXERCISE directory on DosBox. Open this file in a text editor and change the name hello.obj to correspond with the name of the assembly source file. For example, if the assembly source file is called GOODBYE.ASM, the line should become objects1 = goodbye.obj. The case does not matter because MS-DOS is case insensitive. The line beginning with dest specifies the executable name. You can chose a name of your choice, as long as it adheres to the MS-DOS 8.3 file name convention as explained in appendix A.

```
# Settings
objects1 = hello.obj
dest = hello.exe
# ...
```

When typing `WMAKE` on the command line the assembler and linker will be invoked to create an executable with the name you specified. Error and warning messages will be displayed on the screen and should be carefully read.

2.3.2 Multiple source files

It is possible and even good practice to organize your code in different files. Consider a project that consists of 3 source files `A.ASM`, `B.ASM` and `C.ASM`, where the first wants to call procedures defined in the other two. To do so create text files `B.INC` and `C.INC` that declare the procedures you want to call from `A.ASM`. For example, the file `B.INC` may look as follows:

```
GLOBAL  proc1:PROC,\
        proc2:PROC,\
        proc3:PROC
```

Next, include this file in both `A.ASM`, which contains the code that calls these procedures, and in `B.ASM`, which contains the definitions of these procedures. Finally, update `MAKEFILE` accordingly and run `WMAKE` to assemble and link the code.

```
# Settings
objects1 = b.obj c.obj a.obj
dest = mygame.exe
# ...
```

2.4 Miscellaneous Topics

The above only describes the basics of Turbo Assembler. Complete books are available on the topic, for example on Canvas you can find the *Turbo Assembler Version 5 User's Guide* in the *Practica - Handleidingen* folder. For this course you can do fine without knowing all functionality. Nevertheless, we would like to mention some useful features.

Macros it is possible to assign a symbolic name to some constant value using `EQU`. For example the line `MEANING EQU 42` will make the assembler replace every occurrence of `MEANING` by 42. Thus, `EQU` is similar to `#define` in the C programming language. You can specify these macros anywhere in your source code, but good programming practice suggests you specify them in the preamble.

It is also possible to define parametrized macros that capture similar instruction sequences. Like C and C++ macros, expansion of these macros entail serious risks, making them unsuitable for beginning assembly programmers.

Expressions The assembler can compute the value of complex expressions before converting the assembly code to machine code, as long as the expression contains only constants[3]. For example, the assembler will replace `MEANING/2` by `21`.

Predefined Symbols Turbo assembler provides a number of predefined symbols. Some of them expand to a character string. For example:

- `??date`: a string representing the date of assembly.
- `??time`: a string representing the time of assembly.
- `??filename`: a string representing the source file name.

You can use them in your data segment to get some info on when and from which file your program was built:

```
date db ??date, 13, 10, '$'
time db ??time, 13, 10, '$'
file db ??filename, 13, 10, '$'
```

Debugging Type `WMAKE DEBUG` to debug your code using the Watcom debugger. You can step through your code, watch register and memory contents and do much more.

[3]This is not entirely true, register names are set apart before evaluating an expression: `5+EAX+7` will be evaluated to `EAX+12`.

3

The ISA

3.1 Introduction

Before giving an overview of the Intel 386 Instruction Set Architecture (ISA), we briefly explain how a program is executed on a CPU. Although this explanation is a simplification of reality, it should suffice to write correct assembly programs.

When a program is started, its instructions and data are loaded in memory. The first instruction is fetched from memory and executed. Then, the "next" instruction is fetched and executed and this process is repeated until the program requests the operating system to be terminated using a sequence of three instructions as in figure 2.1. Most of the time the "next" instruction is simply the instruction that follows the current instruction in memory.

There exist instructions that can change the flow of execution by changing the "next" instruction. The branch instructions allow programs to jump conditionally or unconditionally to any instruction of the program. They are used to implement `if` statements and loops. Calling and returning from procedures also causes the program to jump to another instruction. Because the "next" instruction is not always the instruction following the current one in memory, the address of the next instruction is kept and updated in a special processor register called the "program counter".

There exist instructions to move data between memory locations, instructions to apply arithmetic and logical operations on data, control flow instructions and a number of special instructions that do not belong to one of the above categories.

This chapter contains many references to the first three chapters of the "Intel 386 Reference Programmer's Manual" which can be found here.

3.2 Memory and Registers

Memory is divided in cells of one byte. Each cell has its own address. A word occupies two cells: the least significant byte is stored at the lowest address and the most significant byte at the highest address. Similarly, a double word

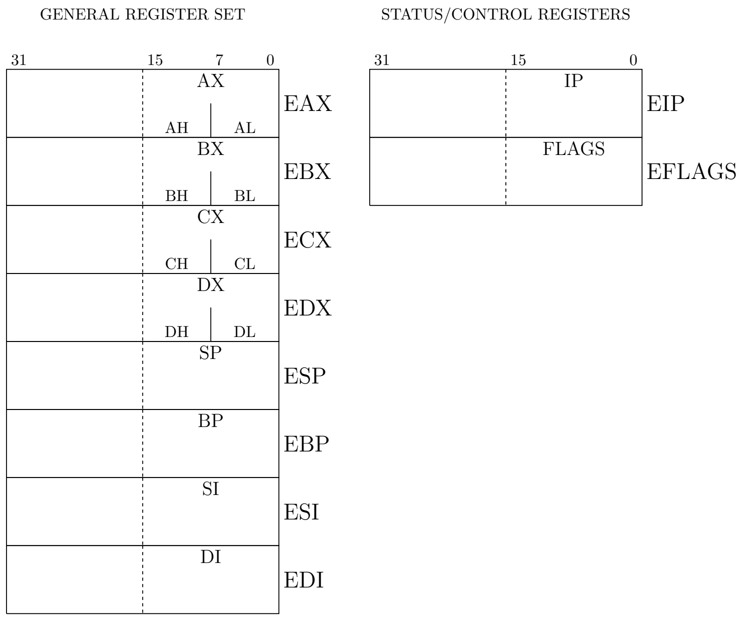


Figure 3.1: Intel 80386 register set

occupies four cells. Registers reside on the processor: they are used to reduce memory access. Figure 3.1 shows the registers of the Intel 80386. Although the general registers can be used for anything, two of them have become special purpose by convention: the stack pointer `ESP` and the base pointer `EBP`.

Each register has room for a double word (32 bits). It is, however, possible to refer to part of a register. For example, if `EAX` contains the double word $b_{31}...b_0$, `AX` should be interpreted as the word $b_{15}...b_0$, while `AH` and `AL` should be interpreted as the bytes $b_{15}...b_8$ and $b_7...b_0$ respectively. This naming convention is very useful to process words or bytes, but one should remember that these names refer to the same register: using one of them as the target of an instruction will change the values the other names refer to[1].

There are two special purpose registers: `EIP` holds the address of the next instruction. `EFLAGS` holds a number of control and condition bits. Control bits determine the behavior of the processor and can be changed with special instructions. Condition bits are used by jump instructions.

We did not mention the segment registers that allow for the use of segmented memory. Because we use a flat memory model we do not need them.

[1]Except for the byte portions that do not overlap: changing `AL` will not change `AH`.

Nevertheless, they will make a brief appearance when we discuss the string instructions.

3.3 Instructions

An ISA instruction in an assembly program looks as follows:

```
[label:] operator-mnemonic [operand1], [operand2]
```

Strictly speaking the label is not part of the instruction: it may be used as the operand of a jump instruction in which case the assembler will replace the label by the instruction's address.

Most instructions that take two operands apply an operation on those operands and store the result in the first one. Most of the time the operands must have the same size i.e. they must both be bytes, words or double words.

3.3.1 Addressing Modes

An addressing mode is a way to specify where an operand is found. Operands that are specified directly in the instruction are called immediate operands. They can only serve as input. Both operands can be registers or memory[2]. A register operand is specified by its name; a memory operand must be specified by its address.

Up to three components can be used to specify a memory address: a base register containing the base address, an index register scaled (multiplied) by 1, 2, 4 or 8 and a constant displacement. Figure 3.2 shows which registers can be used for the first two components. We will discuss some variations of the general addressing mode.

Direct Memory Operands

You can use a displacement only. In an assembly program this can be done with a label that will be translated by the assembler to an address. For the assembler we use it is necessary to enclose the label in square brackets to make the address dereferencing explicit. Although the assembler might let you use a bare label to mean the same, this use is confusing, discouraged and results in an assembler warning.

[2]But you cannot use two memory operands for the same instruction.

$$\left\{\begin{matrix}\text{none}\\ \text{EAX}\\ \text{EBX}\\ \text{ECX}\\ \text{EDX}\\ \text{ESP}\\ \text{EBP}\\ \text{ESI}\\ \text{EDI}\end{matrix}\right\} + \left\{\begin{matrix}\text{none}\\ \text{EAX}\\ \text{ECX}\\ \text{EDX}\\ \text{EBX}\\ \text{—}\\ \text{EBP}\\ \text{ESI}\\ \text{EDI}\end{matrix}\right\} * \left\{\begin{matrix}1\\ 2\\ 4\\ 8\end{matrix}\right\} + \left\{\begin{matrix}\text{none}\\ \text{number}\end{matrix}\right\}$$

Figure 3.2: Addressing data in memory

Using an Index Register

A register may be used to specify an index in an array. Furthermore, the register may be multiplied by a constant to take into account the size of the elements (the index is always interpreted as a number of bytes). This is called scaling. Consider for example: `[ARRAY + 4*ECX]`, assuming that register `ECX` contains the value 4, this refers to the fifth double word starting at the address referred to by the label `ARRAY`.

Using a Base Register

It is possible to store an address in a register and to use that register later to refer to the corresponding location and or to perform address arithmetic. There are two ways to store an address in a register: using `MOV` and `OFFSET` or using `LEA`. The following two lines have the same effect:

```
MOV EAX, OFFSET ARRAY
LEA EAX, [ARRAY]
```

Using a register to address memory is called indirect addressing. Following the above code, you can refer to the data stored at the address in `EAX` by `[EAX]`. Depending on the instruction, this will be interpreted as a pointer to a byte, word or double word. Sometimes, however, the assembler cannot deduce the type of the data. In that case, you need to help it by explicitly specifying whether the address refers to a `BYTE`, a `WORD` or a `DWORD`. For example:

```
INC [EAX]           ; points to 1, 2 or 4 bytes? Fails!
INC [DWORD PTR EAX] ; points to 4 bytes! Succeeds!
```

We could combine the base register with the index register and write more general code that can iterate an array for which the address of the first element is stored in the base register. Furthermore, if the elements of the array contained for example two separate double words the displacement could be used to specify the double word we want to access.

Figure 3.3 illustrates the above for `EDX` containing `01230h` and `ECX` containing `02h`. Each cell represents one byte in memory. The numbers on the left of each cell correspond to their absolute address, while the expressions on the right are placed next to the first byte of the byte, word or double word they may address.

3.4 Instruction Overview

A full overview of all 80386 ISA instructions can be found here. The following subsections discuss the most important instructions from a number of categories.

3.4.1 Data movement

MOV moves data to registers or memory. The source may be a register, memory or a constant (immediate) value. Both operands must have the same size. The following code contains examples of correct and erroneous use of `MOV`. In this code `BPTR`, `WPTR` and `DPTR` are labels that refer to byte data, word data and double word data[3].

```
MOV EAX, EBX        ; OK
MOV EAX, BX         ; ERROR: different size
MOV AX, BX          ; OK
MOV EAX, [BPTR]     ; ERROR BPTR points to a byte
MOV AH, [BPTR]      ; OK
MOV [WPTR], EAX     ; ERROR WPTR points to a word
MOV [WPTR], AX      ; OK
MOV EAX, [DPTR]     ; OK
```

XCHG swaps the contents of two registers or of a register and memory. Both operands must have the same size.

MOVZX like `MOV` but the source operand may be smaller than the destination operand. The missing bits are zero expanded. Please note that this will change the meaning of negative numbers.

[3]Chapter 2 shows how these labels are defined.

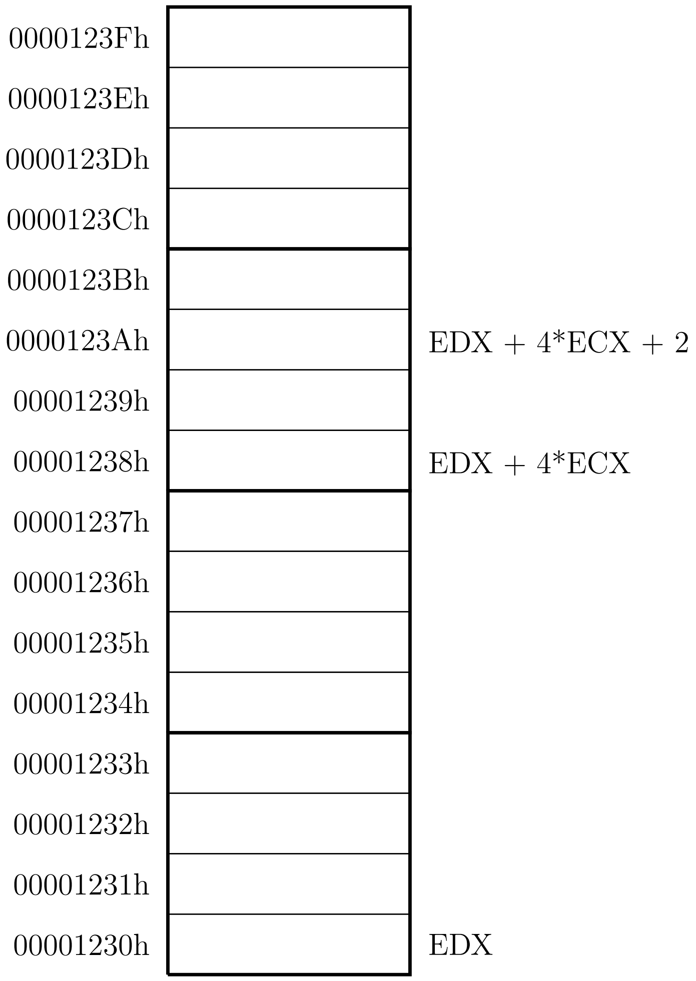


Figure 3.3: Addressing with base register `EDX` containing 01230h and index register `ECX` containing 02h. The base register can be interpreted as the start address of an array, while the latter can be interpreted as an index into this array. In this case the index register is multiplied by 4 from which we can infer the array is interpreted as an array of double words. The three-part address points to the most significant word of the third double word in the array pointed to by `EDX`.

MOVSX like `MOVZX` but the missing bits are sign expanded i.e. `0` will be used for positive numbers and `1` for negative numbers.

3.4.2 String instructions

Strictly speaking the string instructions are data movement instructions. We dedicate a special section to them because they allow to move data in a more efficient manner and share a number of quirks. The string instructions are used to move data from and to memory in an efficient manner. They take no operands: `EAX` is used as the source (destination) when copying data from the processor (memory) to memory (the processor). `ESI` is used to find the memory address when copying from memory and `EDI` when copying to memory. The address in `EDI`, however, is interpreted as an address in the `ES` segment. Because we store all data in the `DS` segment, it is necessary to make the `ES` segment register point to the `DS` segment before using these instructions. This is typically done at the start of the program as follows:

```
PUSH DS
POP ES
```

Depending on the actual string instruction, the addresses in `ESI` and `EDI` will be automatically incremented or decremented by 1, 2 or 4 after each copy. Whether they are incremented or decremented depends on the value of a control bit referred to as the "direction flag". The direction flag is cleared with the instruction `CLD` causing addresses to be incremented. It is set with the instruction `STD` causing addresses to be decremented.

Finally, a string instruction may be repeated a number of times by prepending the `REP` prefix. In this case the contents of `ECX` will be used as the number of times the instruction must be repeated.

The Store String Data Instructions `STOSB`, `STOSW` and `STOSD` transfer one, two or four bytes from `AL`, `AX` or `EAX` respectively to `ES:[EDI]`. After the transfer `EDI` is updated to point to the next datum: if the direction flag `DF` is set to zero 1, 2 or 4 will be added to `EDI`. If `DF` is set to one 1, 2 or 4 will be subtracted from `EDI`.

The following code shows how to efficiently initialize a chunk of memory starting at `0A000H` with zero bytes:

```
CLD                 ; EDI will increase
MOV ECX, 64000
MOV EDI, 0A0000h
MOV AL, 0
REP STOSB
```

This can be done four times faster using STOSD:

```
CLD
MOV ECX, 64000/4    ; the assembler computes 64000/4
MOV EDI, 0A0000H
MOV EAX, 0
REP STOSD
```

The Move Data from String to String Instructions MOVB, MOVW and MOVD transfer one, two or four bytes from [ESI] respectively to [EDI]. After the transfer ESI and EDI are updated to point to the next datum: if the direction flag DF is set to 0 1, 2 or 4 will be added to both registers, otherwise 1, 2 or 4 will be subtracted from both registers. For example, you can copy 100 bytes from some source to some destination:

```
CLD
MOV ECX, 100
MOV EDI, OFFSET DESTINATION
MOV ESI, OFFSET SOURCE
REP MOVSB
```

The Load String Data Instructions LODSB, LODSW and LODSD transfer one, two or four bytes from [ESI] to AL, AX or EAX respectively. After the transfer ESI is updated to point to the next datum in the same way as for the MOVS instructions.

The Scan String Data Instructions SCASB, SCASW and SCASD compare AL, AX or EAX to the datum pointed to by EDI. This comparison works like CMP in the sense that it performs a subtraction but only sets the flags. After the comparison EDI is updated to point to the next datum in the same way as for the MOVS instructions. Remember that EDI points to the ES segment. In combination with the REPE or REPNE prefix, these instructions can be used to skip or find a byte, word or double word in a string. For example, after executing the following code ECX will contain the index in hexits of the hexit[4] stored in AL. If AL can contain bytes not present in hexits, it is necessary to test the zero flag to differentiate between the hexit '0' and the invalid hexit. In the former case the zero flag is set, in the latter it is not.

```
    STD                     ; EDI will decrease
    LEA EDI, [hexits + 15]  ; address of the last hexit (F)
    MOV ECX, 16             ; number of hexits
```

[4] *hexit* is short for hexadecimal digit.

```
    REPNE SCASB             ; repeat until equal or ECX == 0
DATASEG
    hexits DB "0123456789ABCDEF"
```

It is interesting to note that the converse of the previous example can be achieved with the XLAT instruction. This instruction will use the contents of `AL` as an index into a bytes table starting at the address stored in `EBX` and store the byte at this index in `AL`. This can be used, for example, to convert an integer to its hexadecimal representation.

The Compare String Data Instructions Finally, we mention `CMPSB`, `CMPSW` and `CMPSD`, which compare the byte, word, or double word pointed to by `ESI` with the byte, word, or double word pointed to by `EDI`. Like the `SCAS` instructions, the `CMPS` instructions set the flags and update `ESI` and `EDI` to point to the next datum.

3.4.3 Arithmetic and logic instructions

Although most arithmetic and logic instructions are straightforward to use, there are some pitfalls to watch out for. We refer to chapter 1 for background information. In the following we will only discuss multiplication and division in detail.

INC `INC op; op = op + 1`

DEC `DEC op; op = op - 1`

NEG `NEG op; op = -op`

ADD `ADD op1, op2; op1 = op1 + op2`

SUB `SUB op1, op2; op1 = op1 - op2`

MUL takes only one operand; the destination is always implicit. The explicit operand can only be a register or memory. Depending on the size of the explicit operand `AL`, `AX` or `EAX` will be multiplied by the explicit operand and the result will be stored into `AX`, `DX:AX` or `EDX:EAX`. The extra space is needed because the result of multiplying two N bit values may need $2N$ bits. The flags `CF` and `OF` signal whether extra space was needed. In that case, they will both be 1. Note that if the extra space was not needed, the extra register will be zeroed.

IMUL like `MUL` but for signed integers. If the result is small enough to fit in the implicit operand, the extra register is sign extended. The flags `CF` and `OF` will signal whether the extra space was needed. In that case they will both be 1.

`IMUL` is slower than `MUL` but it is more flexible: it does not have to be used with an implicit operand and the second operand can be an immediate. In this case the `CF` and `OF` flags will also be set if the result does not fit in the destination, but it will not be possible to obtain the correct result.

DIV The `DIV` instruction takes only one operand; the destination is always implicit. The explicit operand can only be a register or memory. Depending on the size of the explicit operand `AX`, `DX:AX` or `EDX:EAX` will be divided by the explicit operand. The quotient is stored into `AL`, `AX` or `EAX`. The remainder is stored into `AH`, `DX` or `EDX` respectively.

Note that the above means that if you want to divide a number in `AL`, `AX` or `EAX`, you need to make sure that respectively `AH`, `DX` or `EDX` are zero.

IDIV like `DIV` but for signed integers. Note that if you want to divide a number in `AL`, `AX` or `EAX`, you need to make sure that respectively `AH`, `DX` or `EDX` are sign extended: i.e. they need to contain 0 for positive numbers and all ones for negative numbers.

AND `AND op1, op2; op1 = op1 & op2`

OR `OR op1, op2; op1 = op1 | op2`

XOR `XOR op1, op2; op1 = op1 ^ op2`

NOT `NOT op1; op1 = !op1`

The Shift Instructions The shift instructions allow you to shift the operand a number of bits to the left or right. The logical shift instructions are `SHL` and `SHR`. They will use zeros to fill in the low-order and high-order bits respectively.

A fast way to multiply a number by 2^n is to shift it n positions to the left. A fast way to divide a number by 2^n is to shift it n positions to the right. However, when using a shift to divide a number, using zeros to fill the high-order bits yields a wrong result for negative numbers. Therefore the arithmetic shift operations `SAL` and `SAR` are provided. `SAR` will sign extend the high-order bits. `SAL` is just a synonym for `SHL`.

3.4.4 Control flow instructions

By default the CPU loads and executes program instructions sequentially. But the current instruction may be conditional meaning that it may transfer control to a new location in the program based on the values of certain flags in the `EFLAGS` register. Remember that these flags are set by arithmetic instructions as explained in the "Flags Affected" sections of the 80386 programmer's manual. A (conditional) jump instruction has an operand that specifies the address of the instruction to jump to. As we have seen, the assembly programmer can use an instruction label for this purpose.

The simplest control flow instruction is the unconditional jump JMP that will always jump to the instruction at the address specified by its label.

To help the programmer achieve control flow, there exist some instructions that are only used for their effect on the flags register. These instructions perform the same operation as a particular arithmetic instruction but they do not store the result, the most used are:

- CMP performs a `SUB` but does not store the result.
- TEST performs an `AND` but does not store the result.

Beginning assembly programmers often make the mistake to use these instructions when it is not necessary. For example, they may decrement some register using the `DEC` instruction and then explicitly use `CMP` to compare the new value to 0. This is not necessary: the `DEC` instruction will set the zero flag if the new value is 0 and the sign flag if the new value is negative[5]. Thus, one can use a `JZ` or `JS` instruction immediately following the `DEC` instruction.

There are a great number of conditional jump instructions that will cause execution to jump to the target instruction if a condition is met. A list of all conditional jumps can be found here. To compare signed integers use the "greater" and "less" jumps. To compare unsigned integers use the "above" and "below" jumps. Mixing them up may have unexpected results when comparing negative and positive integers e.g. $1 < -1$. Do you understand why?

The X386 disassembly wiki-book explains how to translate control flow statements in high level languages to low level assembly. In particular, there is a page on Branches and one on Loops. Note that there exists a `LOOP` instruction that may simplify writing loops:

LOOP `LOOP` decrements `ECX` and jumps to the instruction address specified as its only operand. For example, to execute a loop 8 times, one may use the following code:

[5]If the most significant bit is one.

```
    MOV ECX, 8
MY_LOOP:
    ; some code
    LOOP MY_LOOP
```

A common mistake is to use the `LOOP` instruction when `ECX` contains 0. In this case, the first execution of `LOOP` will decrement `ECX` changing it to $\mathbf{2}^{32} - 1$, hence causing the loop to execute a great number of times. Another common mistake is to modify the contents of `ECX` in the body of the loop. The first mistake can be avoided by using the specialized `JCXZ` instruction to jump over the loop if `ECX` is 0. The second mistake can be avoided by pushing `ECX` on the stack at the beginning of the loop body and to pop it before the `LOOP` instruction. The stack and the corresponding instructions are discussed in Chapter 4. The first problem can also occur when using the string instructions with the `REP` prefixes. Of course, the same solution can be applied to prevent it from happening.

3.4.5 Special instructions

The following instructions set and clear `EFLAGS` control bits:

CLD clears the direction flag. Consequently, the string instructions will increment the registers holding the source or destination address.

STD sets the direction flag. Consequently, the string instructions will decrement the registers holding the source or destination address.

CLI `CLI` clears the interrupt flag and hence disables interrupts.

STI sets the interrupt flag and hence enables interrupts.

The INT instruction is used to call an interrupt procedure. Its use is discussed in chapter 5.

4

Procedures

4.1 Introduction

Procedures are units of reusable code that can be called from anywhere in the program. Section 4.2 explains how procedures are defined and called in Turbo assembler ideal mode. Section 4.3 discusses the stack as a preliminary to section 4.4, which explains what goes on behind the scenes. Finally, section 4.5 argues against the use of non double word parameters.

4.2 Ideal Procedures

The easiest way to define and call procedures is to use ideal mode syntax. The following defines a procedure that can be used to add two double words:

```
PROC addNumbers
    ARG @@number1:DWORD, @@number2:DWORD
    LOCAL @@local1:WORD, @@local2:BYTE
    USES EBX

    MOV EBX, [@@number2]
    MOV EAX, [@@number1]
    ADD EAX, EBX
@@return:
    RET
ENDP addNumbers
```

The procedure definition starts with the `PROC` keyword followed by the procedure name and ends with the `ENDP` keyword optionally followed by the procedure name. The `ARG` keyword is followed by names and sizes of parameters. We recommend to use double word parameters only: section 4.5 explains why. The `LOCAL` keyword is followed by names and sizes of local variables. Here, we declared a variable of word size and one of byte size. The `USES` keyword is followed by a list of registers that must be saved and restored by the procedure. In general, you want to include every register that is modified by the procedure in this list, except the one in which you want to return a value. Here, a value is returned in EAX. Every procedure should end with a RET instruction that returns control to the caller. If you forget to do so, bad things will happen.

The CALL instruction calls a procedure. The procedure addNumbers can be called with any two double word arguments: immediate, register or memory operands. For example, you can add a double word in memory whose address resides in ESI to ECX. After the call, the result is in EAX:

```
CALL addNumbers, [DWORD PTR ESI], ECX
; the result is in EAX
```

4.2.1 Parameter names are labels

Note that parameter and local variable names are labels: they refer to an address. If we want to specify their value we must dereference them, for example [@@number1].

When a label commences with @@ it is a local label: it can only be referred to from within the procedure in which it is defined. Local label names can be reused in different procedures and prevent haphazard jumping between procedures.

4.3 The Stack

A stack is a linear list for which all insertions and deletions are made at one end of the list. Typical for a stack is that elements are retrieved (popped) from the stack in the inverse order in which they were added (pushed).

Values are pushed on the stack with the PUSH instruction. Its only operand may be a register, memory or immediate value. Only words or double words can be pushed on the stack. Immediate values are always considered to be double words.

Values can be popped off the stack with the POP instruction. Its operand may be a register or memory. Depending on the operand's size a word or a double word will be popped off the stack into the register or memory. For example:

```
PUSH 1234H
PUSH 5678H
POP EAX     ; EAX contains 5678H
POP EBX     ; EBX contains 1234H
```

ESP points to the top of the stack. Because the stack grows towards decreasing addresses PUSH decreases ESP by 2 or 4, while POP increases ESP by 2 or 4.

The stack is commonly used to preserve and restore register values. Imagine you want to use EAX, but you want to preserve its initial value and restore it after you are done. This is achieved by pushing EAX before modifying it and popping the old value back into EAX once finished:

```
; save eax and ebx values
PUSH EAX
PUSH EBX
;
; instructions that modify EAX and EBX
;
POP EBX ; last in first out
POP EAX ; first in last out
```

As will become clear in the next section, the stack is also used to pass values to procedures.

4.4 Behind the Scenes

A calling convention describes the rules for passing arguments to procedures, returning values to callers and saving and restoring register values before and after execution. Programmers that write code that only they will use can chose their own calling convention. Compiler writers however must follow the calling convention of the language for which they are writing a compiler. C compilers use the CDECL convention. You are expected to understand this convention. Its rules are as follows:

- The caller pushes arguments on the stack in inverse order.
- Integer values and memory addresses are returned in EAX.
- The caller must preserve EAX, ECX, and EDX.
- The procedure must preserve other registers.

We advise you to use USES for all registers you modify in the procedure except the one in which you want to return a value. The following code shows how the procedure addNumbers from section 4.2 is defined and called in the absence of ideal mode[1]:

[1]Note that 8 bytes are foreseen to store the two local variables that in theory fit in 3 bytes. When translating ideal mode procedures Turbo assembler allocates per default four bytes for every parameter and local variable.

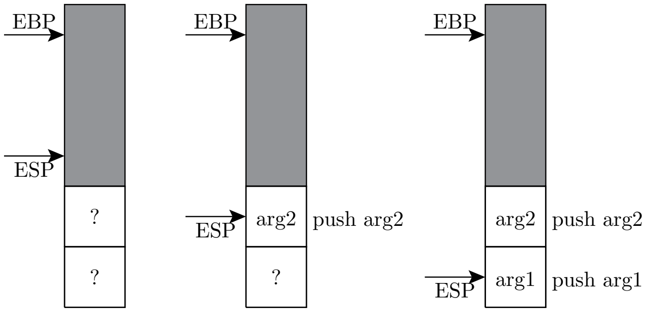


Figure 4.1: The caller pushes the arguments on the stack

```
    ; caller instructions to call addNumbers:
    PUSH 4
    PUSH 3
    CALL addNumbers
    ADD ESP, 8 ; restore esp

    ; callee's code
PROC addNumbers
    PUSH EBP
    MOV EBP, ESP
    SUB ESP, 8  ; room for @@local1 and @@local2
    PUSH EBX
    MOV EBX, [EBP + 12]
    MOV EAX, [EBP + 8]
    ADD EAX, EBX
    POP EBX
    MOV ESP, EBP
    POP EBP
    RET
ENDP addNumbers
```

Figures 4.1, 4.2 and 4.3 show the state of the stack after different portions of the code have executed, up to `PUSH EBX`. The stack is shown with higher addresses towards the top of the page and growing towards lower addresses at the bottom of the page[2]. It is interesting to note (figure 4.2) that after the `MOV EBP, ESP` instruction, `EBP` and `ESP` point to the same address on the stack where the old value of `EBP` is stored.

When the procedure has done all the preparatory work, it can use `EBP` to

[2]I prefer higher addresses to be higher up on the page.

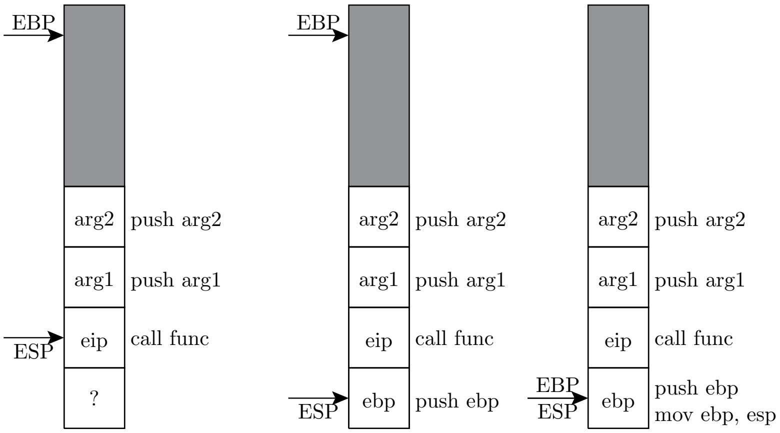


Figure 4.2: When the caller calls the callee, the callee pushes the caller's `EBP` on the stack and initializes it's own `EBP`

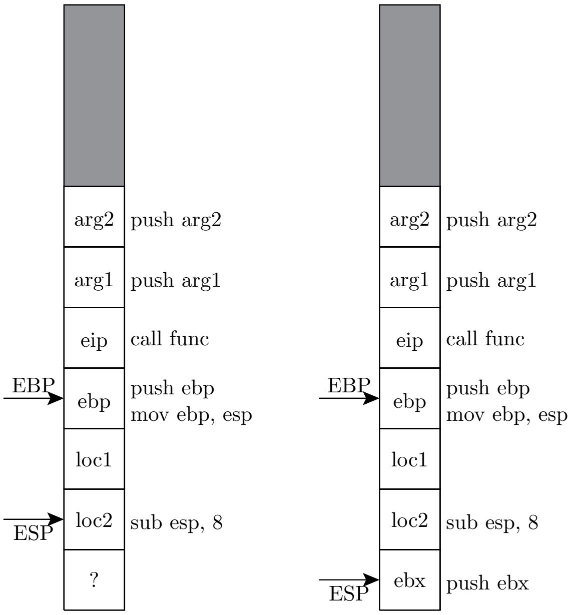


Figure 4.3: The callee makes room for two double word variables and preserves `EBX`

Table 4.1: Using EBP to refer to parameters and local variables.

Label	Indirect address
@@number2	EBP + 12
@@number1	EBP + 8
@@local1	EBP - 4
@@local2	EBP - 8

refer to its parameters and local values as shown in table 4.1.

The reader should be able to draw the state of the stack starting from POP EBX up to and including ADD ESP, 8. After the instructions MOV ESP, EBP and POP EBP, the base pointer will be the caller's base pointer and the stack pointer will point to the stack address where the CALL instruction stored the return address. This is of crucial importance because the RET instruction interprets the value at the top of the stack as the return address. Therefore, inserting a RET instruction at the wrong place will probably crash your program.

4.5 Non Double Word Parameters

We are now ready to argue against non double word parameters. As mentioned earlier Turbo assembler allocates per default 4 bytes for every parameter or local variable of a procedure. Consider the following procedure definition in ideal mode:

```
PROC troubleAhead
    ; the =argSize makes Turbo Assembler calculate the
    ; number of bytes occupied by all parameters and
    ; store it in argSize
    ARG @@word1:WORD, @@word2:WORD =argSize
    USES EAX

    MOV AX, [@@word2]
    RET
ENDP
```

The default allocation will translate @@word1 to EBP+8 and @@word2 to EBP+12. This is correct when the procedure is called with immediate operands: immediate operands are pushed as double words on the stack. When the procedure is called with 16 bit register or memory operands, however, the second operand will reside at `EBP+10`.

It is possible to make Turbo Assembler allocate only two bytes per word by adding a count of one to the parameter. Now the procedure will function correctly for 16 bit register and memory operands but not for immediate operands.

```
    ARG @@word1:WORD:1, @@word2:WORD:1 =argSize
```

Because the number of bytes allocated for a parameter only influences the location of the parameters that follow it, the final parameter may be of `WORD` or even `BYTE` size. More sophisticated workarounds to avoid the problem are possible, but best avoided because in very bad taste.

5

Input and Output

5.1 Interrupts

To perform input and output the assistance of the operating system (DOS) and the basic input output system (BIOS) is needed. A program invokes either DOS or the BIOS by issuing an interrupt using the `INT` instruction. This instruction takes a single immediate operand that determines whether DOS (21H) or the BIOS (10H or 16H) will be invoked. The desired service is communicated by storing the appropriate number in `AH`.

Additionaly, registers may be used to pass data and to store possible results of the interrupt routines. Thus, before an interrupt is raised, the appropriate values must be stored in the appropriate registers. After the interrupt has been handled its (optional) result can be found in the appropriate registers.

This chapter contains examples of most interrupts you need to implement a simple game. If this informations is not sufficient, you can resort to many excellent websites such as Ralf Brown's Interrupt List.

5.2 Output

5.2.1 Video Mode `03H`

The video mode of a program determines how it can access video memory. A program starts in text mode (video mode `03H`). In this mode the program can only display ASCII symbols on the screen through hardware accelerated circuitry. A single character can be displayed as follows:

```
MOV AH, 02H     ; write character to standard output
MOV DL, 'Z'     ; character to write in DL
INT 21H         ; raise interrupt
```

In chapter 2 we showed how to display a dollar terminated string. For convenience we show the concerned code once more:

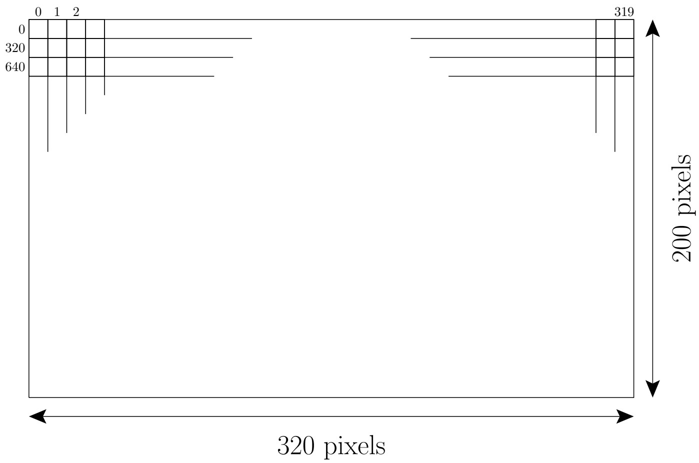


Figure 5.1: Video memory structure for mode 13H.

```
MOV AH, 09H          ; write string to standard output
LEA EDX, [hello]     ; offset of '$'-terminated string in EDX
INT 21H              ; raise interrupt
```

In order to access the individual pixels of the screen, we must change the video mode. We will use video mode 13H a.k.a. VGA mode. This mode is discussed in detail in the following subsection. You can change the video mode as follows:

```
MOV AH, 0H       ; set video mode
MOV AL, 13H      ; desired video mode in AL
INT 10H          ; raise interrupt
```

Because we will make extensive use of the string instructions in the subsequent code samples, it is necessary to make the ES segment register point to the same memory segment as DS. This can be done easily by inserting the following code at the start of your program:

```
PUSH DS
POP ES
```

5.2.2 Video Mode 13H

In VGA mode the program can access the screen as consisting of 320 × 200 256 colour pixels[1]. The screen pixels are backed by a so called frame buffer

[1]200 rows of 320 pixels

that consists of 320×200 bytes and that starts at memory address `0A0000H`. Because the screen is two dimensional and memory is one dimensional, it is necessary to map screen coordinates to memory addresses. Given that the rows of the screen are stored one after another as shown in figure 5.1 and assuming that the first pixel - i.e. the pixel in column 0 of row 0 - is at the top left of the screen, the mapping between a pixel's offset i in the frame buffer and its position $(row, column)$ on the screen is given by the following equations:

$$\begin{aligned} i &= 320row + column \\ row &= \left\lfloor \frac{i}{320} \right\rfloor \\ column &= i \bmod 320 \end{aligned}$$

In video mode `13H` a program can write immediately to the frame buffer. For example the following code stores `15` in the location that refers to the pixel at column 10 of row 2:

```
MOV EDI, 0A0000H    ; frame buffer address
ADD EDI, 320*2 + 10 ; add the appropriate offset
MOV AL, 15          ; index in the colour palette
MOV [EDI], AL       ; change pixel at column 10 of row 2
```

The Colour Palette

In the previous section we showed how values can be written to the frame buffer without specifying what these values mean. They are an index into the so-called colour palette. The colour palette is a simple RGB[2] lookup table consisting of 256 entries. Each entry specifies one colour as an RGB tuple that consists of 3 6-bit values to represent the colour's red, green and blue component respectively. It is interesting to note that the tuple $(0, 0, 0)$ corresponds to the colour black, while $(63, 63, 63)$ corresponds to the colour white.

Figure 5.2 shows the default colour palette for mode 13H. Note that in this depiction it is easy to determine a colour's index in hexadecimal notation: the first digit corresponds to the row where the colour is found, while the second digit corresponds to its column. For example, the index of the last non black colour on the last row is $F7H$ obtained by concatenating FH, the row number, and $7H$ the column number.

[2]RGB is a model to represent colours as consisting of a red, green and blue component

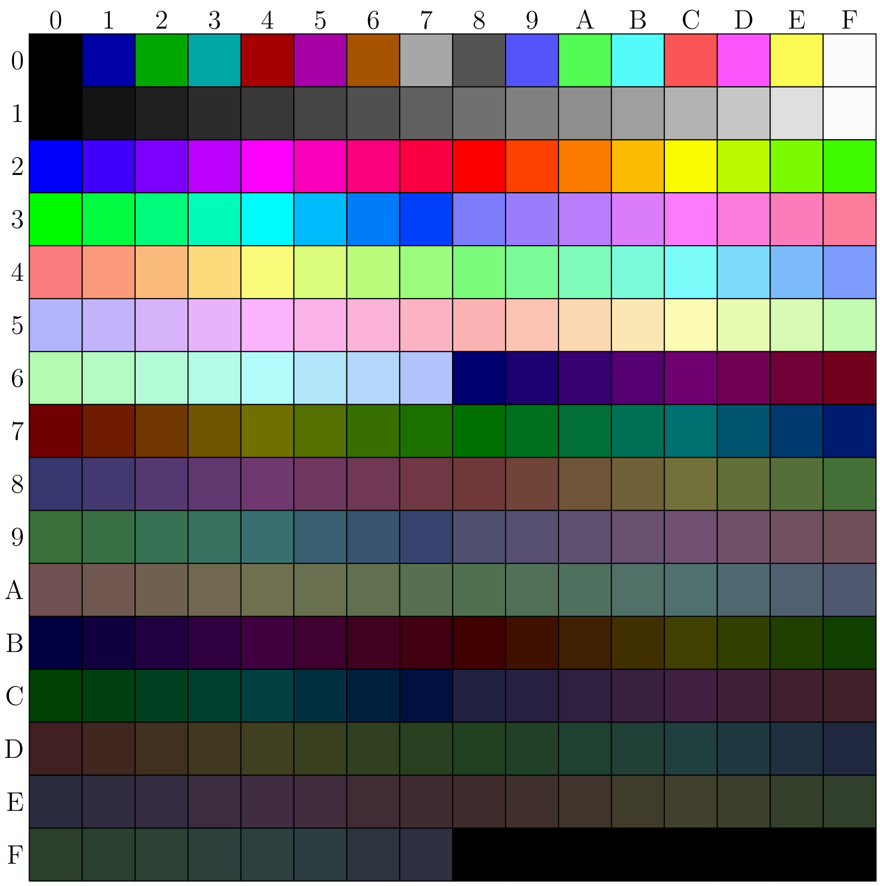


Figure 5.2: Default colour palette for mode `13H`

Changing the colour palette

It is possible to change the contents of the colour palette. This is done by communicating directly with the VGA card using the `IN` and `OUT` instructions. A single colour can be changed as follows:

```
MOV DX, 03C8H   ; port to signal index for modification
MOV AL, 0H      ; change the colour at index 0
OUT DX, AL      ; write AL to the appropriate port
MOV DX, 03C9H   ; port to communicate the new colour
MOV AL, [@@R]   ; move red value into AL
OUT DX, AL      ; write AL to the appropriate port
MOV AL, [@@G]   ; repeat for green
OUT DX, AL      ;
MOV AL, [@@B]   ; repeat for blue
OUT DX, AL      ;
```

Two things should be noted concerning this piece of code. First, only the six least significant bits of each byte are taken into account. Second, once three bytes have been written to `03C9H`, the subsequent bytes will be written automatically to the next index. Therefore, it is not necessary to specify a new index for each RGB tuple and it becomes possible to update all colours in an efficient manner:

```
CLD
MOV DX, 03C8H
MOV AL, 0H
OUT DX, AL
MOV DX, 03C9H
MOV ESI, OFFSET paletteArray
MOV ECX, 256*3
REP OUTSB
```

In the above code `paletteArray` is a label in the data segment preceding an array of 256×3 bytes specifying the 256 colours to be stored in the colour palette.

The Vertical Blank Interval

The VGA card reads the contents of the frame buffer to control the electron beam of the CRT screen. This beam traverses the screen row by row starting at the top left position and ending at the bottom right position. When the beam has reached the bottom right it moves back to the top left. This process is repeated 60 times per second. The period during which the beam moves back to its starting position is called the vertical blank interval or VBI.

To avoid annoying visual distortions it is necessary to update the frame buffer when its contents are not being used to control the electron beam i.e. during the VBI. The following code performs a busy wait for the start of this interval and can be used to update the frame buffer during this interval:

```
    MOV  DX, 03DAH      ; VGA status port
@@waitForEnd
    IN  AL, DX
    AND AL, 8           ; third bit is set during VBI
    JNZ @@waitForEnd
@@waitForBegin
    IN  AL, DX
    AND AL, 8
    JZ  @@waitForBegin
```

If your drawing procedure is very complicated you may not manage to perform all necessary updates during the VBI. In that case *double buffering* may offer a solution. Instead of writing pixels to the frame buffer directly, you write them to a temporary buffer of the same size. This temporary buffer can be copied extremely fast to the frame buffer during the VBI as follows:

```
    CLD                               ; ESI and EDI increase
    MOV ESI, OFFSET screenBuffer ; temporary buffer address
    MOV EDI, 0A0000H                  ; video memory address
    MOV ECX, 64000 / 4                ; copying double words
    REP MOVSD                         ; ecx times
```

Game Loop Timing

Timing and speed control are important factors when writing games. Without speed control, a game will run at the maximum speed that the computer offers, causing inconsistent behavior between different machines running at different clock speeds. Thus, a program needs to include a timing mechanism in order to make sure that it renders and processes the game at a constant speed, independent of the machine it runs on.

A typical way to do this is to synchronize the game loop with the VBI. The following code suggests how this could be done at a high level:

```
    ; ...
@@gameLoop:
    CALL awaitVBIStart
    CALL updateVideoBuffer
    CALL handleInput
    CALL updateGameStatus
    CMP  AL, DEAD
    JNE  @@gameLoop
@@dead:
    ; ...
```

Drawing Sprites

A sprite is a two-dimensional bitmap that is integrated into a larger scene. In our case the larger scene is contained in the frame buffer. Sprites can be of any size, but typically in a game sprites have fixed dimensions, like 16 by 16 pixels. Using sprites, a game's graphical content can be modularized into small drawing blocks. The following describes an 8 × 8 sprite. Note that the sprite begins with two words that specify the sprite's width and height respectively, making it possible to write more general code to copy the sprite to the frame buffer. Writing a procedure that copies such a sprite to a specific (x,y) position in the frame buffer is left as an exercise for the reader.

```
logo    DW  8,  8
        DB 0FH, 00H, 00H, 0EH, 0EH, 27H, 27H, 0FH
        DB 22H, 22H, 22H, 22H, 22H, 22H, 22H, 22H
        DB 22H, 22H, 0FH, 0FH, 0FH, 0FH, 22H, 22H
        DB 22H, 22H, 0FH, 0FH, 0FH, 0FH, 22H, 22H
        DB 22H, 22H, 0FH, 0FH, 0FH, 0FH, 22H, 22H
        DB 22H, 22H, 0FH, 0FH, 0FH, 0FH, 22H, 22H
        DB 0FH, 22H, 22H, 0FH, 0FH, 22H, 22H, 0FH
        DB 0FH, 0FH, 22H, 22H, 22H, 22H, 0FH, 0FH
        DB 0FH, 0FH, 0FH, 22H, 22H, 0FH, 0FH, 0FH
```

The above example sprite `logo` is hard-coded in the data segment to clarify how it is laid out in memory. In general it is a bad idea to hard-code sprites in the source code: it makes it difficult to change them and it increases the size of the executable. Therefore, it is recommended to load the sprites from binary files that contain the corresponding bytes at run time. To avoid increasing the size of the executable you should declare your sprite buffers in the undefined data segment. For the above example, this would look like:

```
logo DW ?, ?
     DB 64 DUP(?)
```

Example code of how to copy the contents of a file to memory can be found in the `C:\EXAMPLES\DANCER` folder on DosBox.

Displaying Text

To display a string in mode 13H it is necessary to first specify the desired position of the string in a grid of 25 lines of 40 characters. Here is a simple procedure that takes three arguments: the row and column where the string should be drawn and the address of the dollar terminated string.

```
PROC displayString
    ARG @@row:DWORD, @@column:DWORD, @@offset:DWORD
    USES EAX, EBX, EDX

    MOV EDX, [@@row]    ; row in EDX
    MOV EBX, [@@column] ; column in EBX
    MOV AH, 02H         ; set cursor position
    SHL EDX, 08H        ; row in DH (00H is top)
    MOV DL, BL          ; column in DL (00H is left)
    MOV BH, 0           ; page number in BH
    INT 10H             ; raise interrupt
    MOV AH, 09H         ; write string to standard output
    MOV EDX, [@@offset] ; offset of '$'-terminated string in EDX
    INT 21H             ; raise interrupt
    RET
ENDP displayString
```

5.3 Input

5.3.1 Keyboard

You can test whether some key has been pressed with `AH = 01H` for interrupt `16H`. Once you are sure a key was pressed, you can get it's corresponding ASCII value and scan code using `AH = 00H` for interrupt `16H`. Because many keys do not have an ASCII code (for example the arrow keys), it is best to use the scan code. A list of PC scan codes can be found here.

```
    MOV AH, 01H         ; check for keystroke
    INT 16H             ; raise interrupt
    JZ  @@noKeyPressed  ; ZF set if no keystroke available
@@keyPressed
    MOV AH, 00H         ; get keystroke
    INT 16H             ; raise interrupt
                        ; AH = BIOS scan code
                        ; AL = ASCII character
@@noKeyPressed
```

```
    ; ...
```

For some games the traditional way to access the keyboard may be too slow. If such is the case, you may have to install a custom keyboard handler. Example code of how this is done can be found can be found in the `C:\EXAMPLES\MYKEYB` folder on DosBox.

5.3.2 Mouse

Example code of how to use the mouse can be found can be found in the `C:\EXAMPLES\MOUSE` folder on DosBox.

5.3.3 Program Arguments

The following procedure stores the program argument string into memory starting at the address passed as an argument to the procedure. Note that it is necessary to make the segment register `DS` point to the program segment prefix. Therefore, the original value of `DS` should be saved at the beginning of the procedure and restored at the end. Furthermore, the destination offset must be written to `EDI` before `DS` is modified. Extracting the arguments from the argument string is left as an exercise for the reader.

```
PROC getArgs
    ARG @@destination:DWORD
    USES EAX, EBX, ECX, ESI

    PUSH DS                     ; save DS
    POP  ES                     ; in ES
    MOV EDI, [@@destination]    ; before changing EDI
    MOV AH, 62H                 ; get current PSP address
    INT 21H                     ; raise interrupt
    MOV DS, BX                  ; store PSP address in DS
    XOR ECX, ECX                ; ECX := 0
    MOV CL, [80H]               ; command line byte count in ECX
    MOV ESI, 81H                ; command line offset in ESI
    REP MOVSB                   ; copy command line
    PUSH ES                     ; restore DS
    POP  DS                     ; from ES
    RET
ENDP getArgs
```

5.4 Files

Example code of how to copy the contents of a file to memory can be found in the `C:\EXAMPLES\DANCER` folder on DosBox.

More information on the relevant interrupts for file manipulation can be found on Ralf Brown's Interrupt List:

Int 21/AH=3CH : Create or truncate file.

Int 21/AH=3DH : Open existing file.

Int 21/AH=3EH : Close file.

Int 21/AH=3FH : Read from file or device.

Int 21/AH=40H : Write to file or device.

Make sure you understand the file handle concept before you start working with files.

5.5 Program Termination

What better way to end this document than a section on program termination? You can request DOS to terminate your program with a specific exit code using interrupt `21H`. Before doing so, you must ensure the program is in text mode i.e. its video mode must be `03H`. If this is the case, the following code may be used to terminate the program:

```
MOV AH, 4CH ; code for program termination
MOV AL, 00H ; program's exit code
INT 21H
```

A

ASMBox

A.1 Basic Information

This appendix mentions the following files that can be found on Canvas:

- Practica - ASMBox + Voorbeelden - ASMBox.zip
- Practica - ASMBox + Voorbeelden - dosbox.pdf

DOSBox is a software that emulates an X386 computer that runs the MS-DOS operating system. A preconfigured package ASMBox.zip is provided for Windows and MacOS. This package is simply DOSBox together with the tools needed to assemble, link and debug assembly programs. To start this package on Windows you should use `Asmbox.bat` by double clicking this file in the Windows explorer. In case of problems, execute this batch file in a command prompt to see if you get additional information. To start this package on MacOs, you should use `ASMBox.app`. If you have a newer version of MacOs, you may have to carry out the work around detailed in Appendix A.1.1. If you are using Linux, or the above does not work, consult the document *dosbox.pdf*, which provides detailed information on how to install DOSBox and the assembler tools yourself.

When you start ASMBox, you will get two windows – a status window and the main DOSBox window. You can ignore the status window. The main DOSBox window emulates an MS-DOS terminal. This is a very minimal environment in which you can only achieve things by typing in DOS commands. Besides building and running programs, we will limit ourselves to a few tasks such as creating new directories and new files, navigating the file system and – possibly – renaming files. When you type `HELP` you will get a list of 11 commands that you can use to perform these tasks.

In practice, because the hard drive of DOSBox is mapped to the `c_disk` directory that can be found in the `ASMBox` directory, you can do a lot of work on your own computer using its native file manager application (e.g. Windows Explorer or Mac OSX finder).

If you use your native file manager application, you have to be careful:

- In DOS the length of a filename is limited to 8 characters and an optional extension of 3 characters. Therefore, your `ASM` files should not have names longer than 8 characters. If so, they will be truncated by DOS and you may fail to build your program.
- File managers may hide extensions. If such is the case, you cannot create a text file with extension `ASM` to hold your assembly source code and neither can you create a text file `MAKEFILE` without an extension. You should either make your file manager show file extensions or follow the procedure outlined next.
- When you create new files or directories in the `c_disk` directory on your own computer, they may not be visible immediately in DOSBox. You should use the `DIR` command to see which files are known in DOSBox. If the newly created file or directory is not listed, you should type `CTRL-F4` or whatever is appropriate on your computer, to force DOSBox to refresh its hard drive.

Let us first show how you can get the *hello example* program running in DOSBox. The file `hello_example.zip` contains two files `HELLO.ASM` and `MAKEFILE`. Using our own file manager we can simply create a directory `PRACTICA` in `c_disk` and subsequently create a subdirectory `HELLO`. Next, we copy the two files to this subdirectory. Note that if DOSBox was already running, you probably have to force a refresh of the directory cache as explained earlier. Finally, in DOSBox you should go to the appropriate directory, build and execute the program:

```
C:\>CD PRACTICA\HELLO
C:\PRACTICA\HELLO>WMAKE
C:\PRACTICA\HELLO>HELLO
```

You can use the hello example as a “blueprint” for subsequent exercises. For example, you could create a new directory in `PRACTICA` and copy the `HELLO.ASM` and `MAKEFILE` to this directory. In the following we give the copy of `HELLO.ASM` another name.

```
C:\PRACTICA\HELLO>CD ..
C:\PRACTICA>MD EX1
C:\PRACTICA>CD EX1
C:\PRACTICA\EX1>COPY ..\HELLO\MAKEFILE .
C:\PRACTICA\EX1>COPY ..\HELLO\HELLO.ASM EX1.ASM
```

You can now edit both `EX1.ASM` and `MAKEFILE` on your own computer in a text editor of your choice. In the `MAKEFILE` file you only have to adapt the values for `objects1` and `dest` to correspond with the name of the `ASM` file:

```
objects1 = ex1.obj
dest = ex1.exe
```

Of course, modifying assembly source files and building and testing the resulting programs is the main purpose of these sessions.

A.1.1 Workaround for MacOS

ASMBox may not work out of the box on the latest versions of MacOS. If this is the case you can apply the following procedure:

1. Right-click `ASMBox.app` and choose *show package contents*.
2. Navigate to the *Contents/MacOs* folder.
3. Right-click `ASMBox` and choose *open*.
4. Enter the administrator user and password if MacOs asks you to do so.

The procedure should be applied once. Thereafter ASMBox should work without further difficulties.

A.2 A Text Editor

If you have not done so already, now is a good moment to chose a text editor to edit your assembly programs. Ultimately, you get to chose yourself which text editor you want to use. Nevertheless, a few comments about what makes a good text editor are useful. Consider the following requirements:

- Be free.
- Be universally available.
- Support syntax highlighting.

The following list enumerates some text editors that I know and that fit these requirements:

Notepad++: a powerful alternative for notepad on Windows. It supports syntax highlighting and tabbing.

Vi or *Vim*: a very powerful, universally available editor with a steep learning curve. Its GUI version *gvim* is easier to use.

Emacs: another old time classic with a steep learning curve.

MacOS comes with *vim* installed. Nevertheless, if you believe it is too hard to learn you can also take a look at for example *Textmate* or *Atom*[1], both free editors that support syntax highlighting.

[1] Make sure your computer has plenty of resources