Assembly Arithmetic Algorithms

16-bit DOS Edition

Preface

You might be surprised to find a book in the 21st century about programming in Assembly Language on DOS. For many people, DOS seemingly disappeared and became irrelevant in the mid 1990s. Fortunately for me, this was not the case. In fact I regularly used DOS on old computers my mother's piano students gave me that they no longer wanted. I had an MS-DOS 3.3 Manual, floppy disks of both 5.5 inch and 3.5 inch sizes, software like Word Perfect and Edlin that you have probably never heard of, and I memorized how to write, copy, rename, and delete files without Microsoft Windows or Linux, both of which I did not have until the 21st century when I was 14 years old and got my first modern laptop which had Windows 98 and the ability to restart into DOS mode.

As you might expect, I spent more time in DOS than I did on Windows 98. If it had not been for the battery failure, I probably would have used it much longer than I did. A world of text terminals was my playground and I was not used to moving a mouse and clicking in a Windows graphical user interface. I used Windows 98 to download DOS games from the internet and to play "Castle of the Winds" (an obscure Norse Mythology game you probably also never heard of).

I tell you all this for context so that you understand why the Magic of older computer systems is still with me even as I write this in the year 2025. Just because MS-DOS and Windows 98 are no longer commonly installed on computers does not mean that the old games or programming styles have disappeared, at least not yet. Thanks to emulators like DOSBox and real operating systems like FreeDOS, it is possible to get programs created 40+ years ago to still run.

But I wanted to go a step further and write new programs that could run within an emulator and theoretically a computer made in the 1980s. However, the information is getting harder to find. I want to thank the authors, both dead and alive, who have worked to make sure this information was freely available on the internet. In particular, I would most like to thank Randall Hyde (author of "The Art of Assembly") and Ralf Brown (creator of "Ralf Brown's Interrupt List"). Without this information, I might have never figured out how to even write "Hello World!" in DOS 16-bit Assembly Language

Therefore, I encourage anyone brave enough to read this book to consider that I am just a nerd that feared this information would be lost forever unless I pass on the information compiled by these genius men who have worked hard to help people learn how to accomplish tasks in Assembly Language for old operating systems that are now only known by those who truly seek to understand how computers work!

Introduction

First, let me introduce this book by telling you what I will teach you. By the end of this book, you will have enough information to write any text based console program in the form of a 16-bit DOS (Disk Operating System) “.com” file.

The “.com” file was a format used by all version of MS-DOS, and even supported on Windows up to XP. It has no header information and is limited to 64 kilobytes of memory. Rather than viewing the limitation as a weakness, I view it as a strength because it forces me to be a better programmer and squeeze the most out of every byte.

Required Knowledge

To get the most out of this book, some background on the Binary and Hexadecimal numeral systems is going to be helpful, but this is not strictly required because I will be providing functions you can use in your code that will convert between decimal (base ten), binary (base two), and hexadecimal (base 16).

However, I would say that experience in at least one programming language is necessary for an understanding of terminology like “arrays”, “pointers”, “addresses”, “integers”, “floating point”, etc. I recommend the C Programming Language as a start. C++ is also a good starting language but tends to abstract details away that directly apply to Assembly Language, which is the lowest level a human can go for understanding a computer.

Low Level

Low level is a term that confuses people. People think something high level is better than low level. In simple terms, humans consider themselves superior to machines and therefore think themselves higher or more important because of their abstract though.

A computer thinks only in terms of numbers. A computer may not understand “high level” abstractions such as love, religion, philosophy, etc, but that is not its job. A computer must add, subtract, multiply, and divide. These are the four arithmetic functions which many human struggle to do.

Therefore I ask you, between a human and a computer, who is really low level or high level? In the age of Artificial Intelligence taking over human jobs and beating humans at Chess, we would all do well to take this question seriously.

I wrote this book because I think like a machine and I hope to help others think this way because it is the best way to learn programming and control your computer by writing Assembly Language programs or to go back to your favorite programming language with a greater understanding of why things work as they do.

Why DOS?

DOS is not at all like Windows or Linux, because it comes from an older time when people were expected to read books and even video games often came in the form of source code published in books. Therefore, I have decided to dedicate this book to the Disk Operating System more commonly called DOS and made famous by MS-DOS which was Microsoft’s version that people in the 80s and 90s remember. Later on, I plan to write a book on programming on Linux using similar but modern methods.

Online Example Programs

Although you can retype every example program from this book and try to run it in your DOS emulator. I also provide the examples as downloads from my Github repository for teaching Assembly.

https://github.com/chastitywhiterose/Assembly/tree/main/fasm/dos/AAA-DOS-book-examples

My samples are free and under the “GNU General Public License v3.0” because my intention is to make Assembly Language easy to learn for everyone without any restrictions. My hope is that others find the joy of programming in DOS, no matter whether they learn it from me or whether they learned it from someone else who may have picked up a few things from me.

Chapter 1: The First Program

For this chapter, I will explain give the source code of an example program that works in DOS, how to assemble it using the tools FASM or NASM, and finally, how the program works line by line.

First, here is the source code of a program that looks like nonsense but does something really cool.

```
org 100h

mov ah,2
mov dl,20h
loop_start:
int 21h
inc dl
cmp dl,7Fh
jne loop_start

mov ax,4C00h
int 21h
```

You will need an assembler. My first recommendation is FASM, the Flat Assembler.

https://flatassembler.net/

You can download FASM and install it by putting it in your path. The instructions for doing this depend on your operating system but it can be done on Windows, Linux, or even within a DOS operating system, which you will of course need to run the program.

Assemble with FASM

To assemble this program with FASM, place the source in a file named “main.asm” and run this command

```
fasm main.asm
```

FASM will automatically create a “main.com” file because it understands by the context of “org 100h” that you are intending to create a DOS executable that ends with a “.com” extension. This directive signals that the program starts at address 100 hexadecimal or 256 decimal. This kind of DOS program always starts at that address.

After you have created the “main.com” file, you will need some kind of DOS emulator to run it. I recommend DOSBox because it is easy to set up and has a lot of documentation to help you.

https://www.dosbox.com/

As an example of how to use DOSBox efficiently, I have added the path of my working directory where I test my programs directly into my DOSBox configuration file.

Instructions for doing this depend on your host operating system. Consult the DOSBox documention for the location of where it will be on your operating system.

```
[autoexec]
# Lines in this section will be run at startup.
# You can put your MOUNT lines here.
mount d ~/git/Chastity-Code-Cookbook/work/
```

This mounts a folder on my Linux system as if it was the D drive recognized by DOS. Back in the DOS and early Windows days, there were “drives” which were all a single letter. A and B were the floppy disk drives, C was the hard disk drive, and sometimes there was a D or E drive for a CDROM drive. DOSBox lets you emulate them and mount any folder on the host operating system (usually Windows or Linux) and access it as you would in DOS.

To switch to the D drive, I just enter

```
D:
```

And then I can type “dir” to see the files, and then I can type

```
main
```

and the main.com file will run. This works because “.com” and “.exe” files are seen by DOS as a program that can be executed or run.

When you run the program, it will display

```
 !"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
```

Basically the program is displaying every printable character. This is the correct behavior I expected when I wrote the program.

Assemble with NASM

You can assemble the example program with NASM instead of FASM if you wish

```
nasm main.asm -o main.com
```

Disassembling the Program

If you have a disassembler, it is possible to extract the source code from the main.com binary file! I always used “ndisasm” for this because it usually comes installed along with NASM.

```
ndisasm -o 0x100 main.com
```

If you disassemble it like this, you will get this as a result:

```
00000100  B402              mov ah,0x2
00000102  B220              mov dl,0x20
00000104  CD21              int 0x21
00000106  FEC2              inc dl
00000108  80FA7F            cmp dl,0x7f
```

```
0000010B  75F7              jnz 0x104
0000010D  B8004C            mov ax,0x4c00
00000110  CD21              int 0x21
```

You will see that it is almost identical to the source except that the loop_start label has been replaced with the number 104h. This is because at the machine code level, there are only numbers.

The first column in the ndisasm output is the address of the instruction. The second are the actual machine code bytes. The third column are the approximation of the original source code. It has small differences but it is close enough that we can figure out which program it was that was assembled!

Now let me break down why it works by repeating the source but including comments this time

```
org 100h       ;tell assembler that code begins running at this address

mov ah,2       ; move (copy) the number 2 into the ah register
mov dl,20h     ; move the number 20 hex=32 dec into the dl register
loop_start:    ; the loop starts here
int 21h        ; interrupt 21 hex=33 dec for a DOS system call
inc dl         ; add 1 to the dl register
cmp dl,7Fh     ; compare the dl register with 7F hex = 127 dec
jne loop_start ; Jump if Not Equal to loop_start

mov ax,4C00h   ; DOS exit call with ah=4C and return al=0
int 21h        ; DOS system call to complete the exit function
```

I know you are probably a little bit confused at this point and have many questions such as

- What is an interrupt?
- What is a system call?
- Why do you write your numbers in hexadecimal?
- What is a register?

I would probably give you the wrong definition if I had to explain what an interrupt is, from a hardware or software perspective. In this case, the interrupt number 21h is something put into memory by DOS which can be called as if it were a function like you would write in any language.

The reason the interrupts and other numbers are in hexadecimal is because most assembly language books and tutorials use them. Hexadecimal is objectively more convenient because it can be more easily converted to and from binary. For now just remember that interupt 21h is actually thirty-three and not twenty-one. I have chosen to stick with hexadecimal for this book because it will be relevant later on when I show you other tools which can be used if you understand hexadecimal!

A register is a special variable that exists on a specific CPU type. DOS, Windows, and most Linux operating systems run on an Intel 8086 compatible CPU. I will explain the registers and their functions.

The General Purpose Registers

There are 8 general purpose registers.

- AX: The Accumulator Register
- BX: The Base Register
- CX: The Count Register
- DX: The Data Register
- SI: The Source Index
- DI: the Destination index
- BP: The Base Pointer
- SP: The Stack Pointer

Of those 8 registers, only BX,BP,SI,DI can be used as index variables. This is only a limitation of 16 bit real mode programs like those written in this book. 32-bit and 64-bit programs do not have this limitation, but they have other limitations to worry about and will be covered in future books.

These registers are "general" in that they can do many things, but they each have more "specific" uses also.

In my source code, I use lowercase for the names of instructions and registers, but for this section, I listed them in capital letters because they are actually acronyms named for their purpose according to what Intel had in mind when making the 8086 and above Central Processing Units.

Most of the time, I stick with only AX,BC,CX,DX when writing my programs. If I need an extra registers, I will use BP,SI,DI. There are special instructions for them but these are outside the scope of what I am trying to teach with this book.

You might wonder, why isn't there EX,FX,…YX,ZX? Perhaps in a perfect world there should have been, but they probably didn't want to spend the extra money on having extra registers for every 26 letter of the alphabet.

In the next chapter I am going to introduce a program that can print any string you give to it. Basically, it will be the proper "Hello World" program that you were expecting.

Interrupt Information

All code depends on different functions of interrupt 21h in this book. I have provided the following link to where you can read about the most common functions of this interrupt which will be used in this book

https://github.com/chastitywhiterose/Assembly/blob/main/doc/Chastity-DOS-Interrupts.txt

The function chosen depends on the value of the AH register (the upper half of the AX register). Depending on which function is selected, then other registers act as options or arguments to these functions. The example I included in this chapter shows only the ah=2 call (equivalent of C putchar call) and the exit call of ah=4Ch with al being the return value.

Chapter 2: The putstring Function

For this next program, I will be introducing the “putstring” function that I wrote. This function takes the address of wherever the ax register points to, then does a routine to scan for the next zero byte. Then it subtracts the original address from the address where the zero was found. By doing this, it knows how many bytes there are to print in the string.

Then it loads the registers in the following way:

- AH = 40h (the DOS write call)
- BX = file handle
- CX = number of bytes to write
- DS:DX -> data to write

Then interrupt 21h is called and this executes the most fun system call possible. It can print any string you give it. Take the following source and assemble it with either FASM or NASM as described in chapter 1.

```
org 100h

main:

mov ax,text
call putstring

mov ax,4C00h
int 21h

text db 'Hello World!',0Dh,0Ah,0

;This section is for the putstring function I wrote.
;It will print any zero terminated string that register ax points to

stdout dw 1 ; variable for standard output so that it can theoretically
be redirected

putstring:

push ax
push bx
push cx
push dx

mov bx,ax                 ;copy ax to bx for use as index register

putstring_strlen_start:    ;this loop finds the length of the string as
part of the putstring function

cmp [bx], byte 0           ;compare this byte with 0
jz putstring_strlen_end    ;if comparison was zero, jump to loop end
because we have found the length
```

```
inc bx                     ;increment bx (add 1)
jmp putstring_strlen_start ;jump to the start of the loop and keep
trying until we find a zero

putstring_strlen_end:

sub bx,ax                  ; sub ax from bx to get the difference for
number of bytes
mov cx,bx                  ; mov bx to cx
mov dx,ax                  ; dx will have address of string to write

mov ah,40h                 ; select DOS function 40h write
mov bx,[stdout]            ; file handle 1=stdout
int 21h                    ; call the DOS kernel

pop dx
pop cx
pop bx
pop ax

ret
```

If you assembled it and ran it in DOS, you should get

```
Hello World!
```

As the result. I know this doesn't seem very impressive, but this program accomplishes a lot. You see, in Assembly, you don't have access to C's "printf" or even "puts". However, the 40h call of DOS is useful enough that during the course of this book, I will introduce how you can use my functions to replace the standard library output functions or even modify them if you don't like the way I wrote them!

If I had to compare DOS 40h to something in C, I would compare it to the "fwrite" function which writes a specified number of bytes to a specific file stream. Writing to file 1 is the same as writing to the screen.

Specifically, entry "D-2140" in "INTERRUP.F" of Ralf Brown's Interrupt list is where I got my documentation I required to write the "putstring" function.

If you look at the source, you will see I included a "main:" label. This wasn't actually necessary but I added it for clarity and to distinguish the main function from the putstring function. This is a convention I will keep for the remainder of this book.

The "Hello World!" is defined as data in the assembler like this.

```
text db 'Hello World!',0Dh,0Ah,0
```

That line is not actually assembly language but is pure data according to the way it is defined in both FASM and NASM. The "0Dh,0Ah" are bytes defining the end of a line in DOS. Finally, I ended the string with a 0 because the putstring function uses it to know when to stop printing.

Therefore, the entire main function is:

```
main:
```

```
mov ax,text
call putstring

mov ax,4C00h
int 21h
```

The “call” instruction calls a function. As far as assembly is concerned, a function is just a label the same as why you might use for a loop. The difference is that a “ret” intruction will send the program back to where it should be when the function is done. If you forget the “ret” instruction, you will cause a crash because the computer will keep trying to execute code that you did not write. Luckily, if you are running your DOS program in DOSBox, you will only crash the emulator and not your host operating system.

When I designed the putstring function, I chose ax as the register to first hold the address of the string. I did this because ‘a’ is the first letter of the alphabet and so I use it as the first argument for any of my written functions.

However, considering that the dx register is used for the data location in the DOS write call, perhaps it would have made more sense to write it that way. This is just a matter of personal taste and I mention it to show you that even assembly language allows a certain amount of personal style when writing your code.

You may have noticed the push instructions at the beginning of the putstring function and the pop instructions at the end of the function. The push and pop intructions operate the “stack”, It is a First In Last Out method of managing temporary storage.

Because we are required to use those 4 registers for the system call, we back them up and then restore them. This way, the registers retain their original value as if we had never modified them in the function. This may not seem important now, but in the following chapters, we will be printing lots of strings and numbers, so it is important that their values don’t change while we use them in integer sequence programs later on!

But all the putstring function does is print a string of text. It can’t print numbers as humans would expect to see them, at least not yet! In the next chapter, I will correct this problem by showing you a function that can print integers!

If you don’t understand the reason the programs in chapter 1 and 2 work, that’s because I am first establishing a code base which can be used to give you feedback. Without a way of printing output, we have no idea whether our code is correct!

Chapter 3: The intstr and putint functions

In this chapter, I will introduce two new functions designed to work with the putstring function from the last chapter. We can already print a string, but this doesn't work for numbers. Fortunately I have written my own functions which can convert whatever number is in the ax register into a string which can be displayed.

The source of a complete program is below. Take a good look at it even if you don't understand it at first because I will be explaining some things about it.

```
org 100h

main:

mov ax,text
call putstring

mov [radix],word 10
mov [int_width],word 1

mov ax,1

main_loop:
call putint
add ax,ax
cmp ax,0
jnz main_loop

mov ax,4C00h
int 21h

text db 'This program displays numbers!',0Dh,0Ah,0

;This section is for the putstring function I wrote.
;It will print any zero terminated string that register ax points to

stdout dw 1 ; variable for standard output so that it can theoretically
be redirected

putstring:

push ax
push bx
push cx
push dx

mov bx,ax                  ;copy ax to bx for use as index register

putstring_strlen_start:    ;this loop finds the length of the string as
part of the putstring function

cmp [bx], byte 0           ;compare this byte with 0
```

```
jz putstring_strlen_end    ;if comparison was zero, jump to loop end
because we have found the length
inc bx                     ;increment bx (add 1)
jmp putstring_strlen_start ;jump to the start of the loop and keep
trying until we find a zero

putstring_strlen_end:

sub bx,ax                  ; sub ax from bx to get the difference for
number of bytes
mov cx,bx                  ; mov bx to cx
mov dx,ax                  ; dx will have address of string to write

mov ah,40h                 ; select DOS function 40h write
mov bx,[stdout]            ; file handle 1=stdout
int 21h                    ; call the DOS kernel

pop dx
pop cx
pop bx
pop ax

ret

;this is the location in memory where digits are written to by the
intstr function
int_string db 16 dup '?' ;enough bytes to hold maximum size 16-bit
binary integer
;this is the end of the integer string optional line feed and
terminating zero
;clever use of this label can change the ending to be a different
character when needed
int_newline db 0Dh,0Ah,0 ;the proper way to end a line in DOS/Windows

radix dw 2 ;radix or base for integer output. 2=binary, 8=octal,
10=decimal, 16=hexadecimal
int_width dw 8

intstr:

mov bx,int_newline-1 ;find address of lowest digit(just before the
newline 0Ah)
mov cx,1

digits_start:

mov dx,0;
div word [radix]
cmp dx,10
jb decimal_digit
jge hexadecimal_digit

decimal_digit: ;we go here if it is only a digit 0 to 9
add dx,'0'
```

```
jmp save_digit

hexadecimal_digit:
sub dx,10
add dx,'A'

save_digit:

mov [bx],dl
cmp ax,0
jz intstr_end
dec bx
inc cx
jmp digits_start

intstr_end:

prefix_zeros:
cmp cx,[int_width]
jnb end_zeros
dec bx
mov [bx],byte '0'
inc cx
jmp prefix_zeros
end_zeros:

mov ax,bx ; store string in ax for display later

ret

;function to print string form of whatever integer is in eax
;The radix determines which number base the string form takes.
;Anything from 2 to 36 is a valid radix
;in practice though, only bases 2,8,10,and 16 will make sense to other
programmers
;this function does not process anything by itself but calls the
combination of my other
;functions in the order I intended them to be used.

putint:

push ax
push bx
push cx
push dx

call intstr
call putstring

pop dx
pop cx
pop bx
pop ax
```

```
ret
```

If you assemble and run this program, you will get the following output.

```
This program displays numbers!
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
```

The program prints ax, adds ax to itself, and then stops as soon as ax “overflows” by going higher than the 16 bits limit. When this happens, it will become zero. Our jnz means Jump if Not Zero to the main loop.

If you look at the main function you will see that I set the radix to 10 with a mov instruction, even though it defaulted to 2. This is because most humans are used to decimal, AKA base ten. The base can be changed at any time in the program however you like.

Another thing you will notices is that the “putint” function does not process anything at all. It simply backs up the registers, calls the intstr function to create a string and then calls putstring to display it. In this example, a newline is automatically added for convenience. In my own code, I usually have this done manually by another small function, but for the putposes of this book, this default behavior is fine.

The real power of this program is the intstr function and why it works as it does. I will spend the rest of this chapter explaining why it works, why I designed it this way, and why this function is essential for Assembly language programs to make sense at all.

First, before the function begins, I have defined data using db and dw directives that FASM and NASM both understand.

```
int_string db 16 dup '?'
```

This creates sixteen bytes of question marks. The actual bytes used here don’t matter but I used ‘?’ marks to signal that the data that will go here is unknown when the program starts. The actual digits of the number we convert from the ax register will replace these bytes when we call the intstr function.

```
int_newline db 0Dh,0Ah,0
```

This line takes care of two problems. First, it makes sure that there is a zero byte after the

16 bytes of data from the int_string variable. It also includes the bytes thirteen and ten in hexadecimal notation. This is how newlines are saved if you have used a text editor in DOS or Windows. When you hit the return key it generates both these bytes to define that a line of text has ended. If you were to change the 0Dh to 0, then the program would print the numbers but without separating them with lines or even spaces. Such a thing would make the numbers hard to read. That is why the default behavior is to print a number and end the line for readability. This works for most simple integer sequence programs I will include in this book.

```
radix dw 2
int_width dw 8
```

These are two variables that define the base/radix of the number string generated and also the "width" AKA how many leading zeroes are used when writing it. The width should be set to one for most programs when decimal integers are expected. However, setting the width to 8 or 16 makes sense for binary integers where seeing the exact bits in their positions lined up is essential.

The defaults I have chosen include radix 2 (the binary numeral system) and a width of 8 (for seeing 8 bits of a byte). But the defaults are irrelevant for what you need to know. See how the main function in my example program for this chapter overwrites them in the main function.

```
intstr:
```

This is the label defining the start of the intstr function. If this label were not present, then the "call putint" statement would not know what you mean. Also, keep in mind that "intstr" is just an address in memory much like "radix" and "int_width" are addresses that tell where bytes of data are. However, the convention I use is that labels ending with a colon are labels that will be called with the "call" instruction or jumped to with a jmp or j?? instruction. There will be more explanation of conditional jumps in chapter 3.

```
mov bx,int_newline-1
mov cx,1
```

Before the loop in the intstr function, we set bx equal to the address of the byte before int_newline. This will be the final '?' we defined earlier. The cx register is set to one to signal the number of bytes that will exist in the string. Every number, including 0 and 1 have at least one digit no matter which base you use. The cx register will come into use near the end of the function in its own loop.

```
digits_start:
```

This is a label defining the loop of where the digits are generated in the string.

```
mov dx,0;
div word [radix]
cmp dx,10
jb decimal_digit
jge hexadecimal_digit
```

dx is set to 0 because this has to be done before the "div" instruction. Otherwise, it will be mistaken as part of the dividend. This is a quirk of the x86 family of CPUs. The div intruction takes one argument, in this a word value from address radix and divides the ax register. If we don't zero dx, it will use the dx register as an upper 16 bits of the number we are dividing from as well as using ax as the lower 16 bits.

For a full explanation of this division behavior, see section "2.1.3 Binary arithmetic instructions" in the FASM documentation. Tomasz Grysztar explains it better than I can and his information greatly helped me when trying to figure out why my function wasn't working.

After the division, the dx register contains the remainder of the division. The ax register will be whatever it was divided by the radix. Knowing this, we "cmp dx,10" which means compare the dx register with 10. If it is less or below 10, then we know it is a decimal digit in the ranger of 0 to 9. If it is greater than or equal. Based on these conditions, we jump to one of two sections. One handles decimal digits and the other handles hexadecimal digits. Technically bases 2 to 36 are handled by my program as a consequence of the way I wrote it, but I wrote it with the idea that I would be using this function with only 3 different bases.

- base 2 or binary for my personal enjoyment
- base 16 or hexadecimal for a short form of binary
- base 10 or decimal which is what humans know how to read. It will be used mostly in this book

```
decimal_digit: ;we go here if it is only a digit 0 to 9
add dx,'0'
jmp save_digit

hexadecimal_digit:
sub dx,10
add dx,'A'
```

These two sections do the math of converting the byte digit into a character in ASCII representation that is printable. In either case, code moves on to the save_digit label after these.

```
save_digit:

mov [bx],dl
cmp ax,0
jz intstr_end
dec bx
inc cx
jmp digits_start

intstr_end:
```

This tiny section saves the digit we obtained from this pass of the loop. The dl register is the lower byte of the dx register so we store this character of the digit into the address pointed to by bx.

Keep in mind that pointers are a primary feature in Assembly Language despite being criticized in C/C++ and excluded entirely from other languages like Java.

Next, we compare ax with zero. If it is zero, there are no more digits to write and we will end this loop by jumping to “intstr_end”. Otherwise we decrement (subtract 1 from bx) so that it will point to the digit left of the one we saved this time. We also increment cx so that it knows at least one more digit is to be written because the loop will happen again. We unconditionally jump to digits_start to process digits and save them until ax equals zero.

After ax is zero, we still have one more job to do in this function. The following loop will prefix the string with extra ‘0’ characters while cx is less than the int_width variable. This will be important for those who need the digits lined up to their place values. This is much more important for binary and hexadecimal than it is decimal, but it can still be helpful in decimal as I will show in a later chapter.

```
prefix_zeros:
cmp cx,[int_width]
jnb end_zeros
dec bx
mov [bx],byte '0'
inc cx
jmp prefix_zeros
end_zeros:
```

There is only one more instruction before we return from this function. We copy the bx register to the ax register because it points to the beginning of the string. This means that my putstring function will accurately display it because it expects ax to contain the address of the string.

```
mov ax,bx
```

Last but not least, we still have to end the function by returning to the caller.

```
ret
```

Before I end this chapter, I want to explain why I chose the register ax as the foundation for the behavior of my Assembly functions. ax is a special register in the sense that multiplication and division instructions use it as the required number we are multiplying or dividing. The Intel architecture treats this register as being more important for this reason.

But it is not just that, the programmers of DOS decided that the ax register was what decided which function of interrupt 21h would be called.

Therefore, because others already treated register ax as special, and since ‘a’ is the first letter of the alphabet, I decided that it would be the foundation of all my functions in “chastelib”, my DOS Assembly Standard Library. You are not expected to take my functions as the way things must be done. Once you are done with this book, you may continue learning beyond my skills and may decide that using another register makes more sense than ax.

I wrote this book to teach Assembly Language as I understand it, not to force my coding practices on you. However, I add these extra details so that other programmers who have experience in Assembly will have answers before they start emailing me: “***Chastity, why didn’t you write the function this way! You can save a few bytes if you use***

instruction ??? instead or you could achieve faster speed if you avoided this jump here.”

I am letting you know now, I wrote the code for simplicity rather than performance. I use a very limited subset of the instructions available to the Intel 8086 family of CPUs. I firmly believe that all math for programs I want to write can be written using only **mov,push,pop,add,sub,mul,div,cmp,jmp (and conditional jumps as well).**

Chapter 4: Chastity’s Intel Assembly Reference

I use a very small subset of the Intel 8086 family instruction set. This is both because I want to limit it to my small memory (my brain memory, not computer memory) and because I only care about instructions that existed on CPUs at the time of DOS operating systems. Entire video games and operating systems were written either in Assembly or in C programs that were translated to Assembly. Newer CPUs introduced more instructions but I would argue that these were for convenience or higher speed in limited cases.

For portability, I stick with the instructions I will teach you in this chapter. By portability, I mean portable between as many CPUs of the intel family. Other processors are of course incompatible but they have their own equivalents by different names.

Important note. All program listings in this chapter assume that you also included the putstring,intstr,and putint functions as shown in chapters 2 and 3. This can be done by including external files or just copy pasting their text after the system exit call from ax=4C00 and interrupt 21h.

mov

The mov instruction copies a number from one location to another. In the FASM and NASM assemblers, the instruction always takes the form

```
mov destination,source
```

Think of it as “destination=source” as you would right in C. For example, in the following program which prints the number 8, we see that most of the required data is set up with mov instructions.

```
org 100h
main:

mov word [radix],10
mov word [int_width],1

mov ax,3
mov bx,5
add ax,bx

call putint

mov ax,4C00h
int 21h
```

That program also contains the call, int, and add instructions to make a program that does something useful. However, mov instructions take up the largest part of any program. Whether you are filling a register with a number, another register, or a memory location, the mov instruction is the way to do it.

add

Next to mov, you will see that add is going to be your friend in Assembly a lot. In the previous example, we saw that 3 and 5 were added to make 8. Just like mov, add follows the same rules.

```
add destination,source
```

- Source is left of Destination and separated by a comma
- Source and destination can be registers and memory locations
- Source and Destination cannot both be memory location

Most intructions that take two arguments follow these same rules. Once you have mastered mov and add, you can handle almost anything in a program because you know the basic rules.

There is also the “inc” instruction which takes only one item and adds 1 to it. This is just a shorter way of saying “add Destination,1”

sub

As its name implies, sub will subtract the Source from the Destination.

```
sub destination,source
```

Since it follows the same rules as mov and add (starting to see a pattern yet?), subtraction is just as easy as addition.

Just as “add” has “inc”, “sub” has the “dec” instruction which subtracts 1. Adding or subtracting 1 are probably the most common thing ever done while programming in any language.

Just as a review of the mov,add,sub instructions, here is a small program to show their effect.

```
org 100h
main:

mov word [radix],10
mov word [int_width],1

mov ax,8
call putint
add ax,ax
call putint
sub ax,4
call putint

mov ax,4C00h
int 21h
```

That program will output the following.

```
8
16
12
```

This is because we set ax to 8, then we added ax to itself, and finally we subtracted 4 from ax. Once you think about how easy this is, read on to see how multiplication and division work.

mul

The mul instruction is slightly different than The previous instructions. It takes only one operand which must be either a register or memory location. It multiplies ax by the value of this operand. If the value is too large to fit within the ax register, it puts the higher bits into dx.

div

The div instruction divides ax by the operand you give it (the divisor). However, division is a tricky operation because not every number divides evenly into another. It is also more complicated by the fact that the dx register is assumed to be the upper half of the bits in the dividend while ax is the lower bits of the dividend.

I know it sounds complicated but it is easier than I can explain. I can illustrate this with a small program that multipies and divides!

```
org 100h
main:

mov word [radix],10
mov word [int_width],1

mov ax,12
call putint
mov bx,5
mul bx
call putint
mov bx,8
mov dx,0
div bx
call putint
mov ax,dx
call putint

mov ax,4C00h
int 21h
```

The output of that program is this:

```
12
60
7
4
```

This is because 12 was multiplied by 5 to get 60. Then we attempted to divide 60 by 8. It goes in only 7 times (which equals 56). This means the remainder is 4, which is stored in the dx register after the division.

You may also notice in the source above that I set dx to zero before the div instruction. If this is not done, the dx might have mistakenly had another number and been interpreted as part of the dividend.

I also think some terminology about division is helpful here.

- Dividend: The number we are dividing from.
- Divisor: The number we are dividing the dividend by. AKA how many times can we subtract this number from the divisor?
- Quotient: The result of the division.
- Remainder: What is left over if the divisor could not divide perfectly into the dividend.

As much as I love math, I find some of these terms confusing when I try to explain it in English. Let's face it, I am better at Assembly Language and the C Programming Language than I am with English, but it looks like you're stuck with me because normal people are not autistic enough to care!

For a more in depth explanation of the mul and div instructions, I will include those written by Tomasz Grysztar (creator of the FASM assembler) in the official "flat assembler 1.73 Programmer's Manual"

mul performs an unsigned multiplication of the operand and the accumulator. If the operand is a byte, the processor multiplies it by the contents of AL and returns the 16-bit result to AH and AL. If the operand is a word, the processor multiplies it by the contents of AX and returns the 32-bit result to DX and AX.

div performs an unsigned division of the accumulator by the operand. The dividend (the accumulator) is twice the size of the divisor (the operand), the quotient and remainder have the same size as the divisor. If divisor is byte, the dividend is taken from AX register, the quotient is stored in AL and the remainder is stored in AH. If divisor is word, the upper half of dividend is taken from DX, the lower half of dividend is taken from AX, the quotient is stored in AX and the remainder is stored in DX.

Perhaps you can see that Assembly language is nothing more than a fancy calculator, except better. This is because there is no question which order the operations take place in. There is no need for mnemonics like "*Please excuse my dear Aunt Sally*" to remind us "*Parentheses, Exponents, Multiplication and Division (from left to right), and Addition and Subtraction*".

There are still two more instructions before we can construct useful programs. In fact, my previous examples have used these already, but now it is time to explain them in depth.

cmp

The cmp instruction compares two operands but does not do any math with them. They remain unchanged but modify flags in the processor that allow us to jump based on certain conditions.

jmp

The jmp instruction jumps to another location regardless of any conditions. It has a family of other jump instructions that jump only if certain conditions are true. In fact many of them have multiple names for the same operation. For example je and jz both jump if the two numbers compared would be zero if they were subtracted. This would only be true if they are the same.

Here is a small chart, but it does not cover every alias for these.

Instruction	**Meaning**
je/jz	jump if equal
ja	jump if above
jb	jump if below
jne/jnz	jump if not equal
jna	jump if not above
jnb	jump if not below

Aside from those main 6 conditional jumps that I have memorized. There also exists jl(jump if less) and jg(jump if greater). However, they are for signed/negative numbers which I have not covered. Personally I don't agree with the way negative numbers are represented in computers but I know that understanding the context of signed vs unsigned is important for more complex programs. Once again, I recommend the FASM programmers manual for details that I have excluded for the purpose of keeping this book short.

The following program can print a message telling you whether ax is less than , equal to, or more than bx. Upon this foundation all the conditional jumps in my programs and functions are based.

```
org 100h
main:

mov word [radix],10
mov word [int_width],1

mov ax,5
mov bx,8
cmp ax,bx
jb less
```

```
je same
ja more

less:
mov ax,string_less
jmp end
same:
mov ax,string_same
jmp end
more:
mov ax,string_more
jmp end

end:
call putstring

mov ax,4C00h
int 21h

string_less db 'ax is less than bx',0
string_same db 'ax is the same as bx',0
string_more db 'ax is more than bx',0
```

Personally I think that the system of conditional jumps makes a lot of sense. Other programming languages such as BASIC and C have “goto” statements that work like this. For example, `if(ax<bx){goto less;}`.

The only thing I have found difficult is remembering which acronym means which condition. However, since I created the chart in this chapter, now I can refer to it and you can too! As long as I keep these main six types of conditions in my head, and am working with unsigned numbers, I can write Almost any assembly program from scratch.

push/pop

The push and pop instructions are something you have already seen in my code. The operate on what is called the “stack”. Basically, when you push something, it is like pushing a box of cereal onto a shelf at Walmart. The last item pushed is at the front and will be the first item a customer sees. This is what is called a Last In First Out.

Not only is the stack useful for saving the value of registers temporarily as I do, but without it, it would not be possible to have callable functions. When you call a function with “call”, it is the same as a “jmp” to that location except that it pushes the address where the program was before the call. The “ret” instruction returns to the location that called the function and then proceeds to instructions after it.

The sp register, as I mentioned in chapter 1, is the stack pointer. Every time you push a value, it stores it at the address the stack pointer is pointing to and then subtracts the size of the native word size. For example, this is always 16 bits in the context of DOS programming for 16 bit .com files. This means that you can use it with the other registers to save their value for later.

In the next chapter, I will show a useful example of the push and pop instructions and explain a little bit more about this.

Take it slow

I know I hit you with a lot of information in this chapter, but trust me, I am intentionally leaving out a lot because I don’t want this book to be the size of the Intel® 64 and IA-32 Architectures Software Developer Manuals

https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

There are hundreds of instructions for Intel machines and yet if you combine the instructions I have described in this chapter with the “call”,“int”, and “ret” instructions required for calling functions for input and output, you will see that it is possible to write almost any program I want with these instructions.

I am taking what I have learned by reading the Intel Manuals and the API references available for DOS so that you don’t have to spend as much time figuring these things out as I did. What I can tell you though, is that the result was worth it because I have been able to write programs to accomplish tasks faster than my C programs could. At the same time, the Assembly versions took longer to write than the C versions did. This is the price I must pay to have high performing code.

Also, there are some bitwise instructions by the names of AND,OR,XOR,NOT,SHL,SHR that are sometimes useful for making programs faster and smaller. However, these only make sense in the context of the Binary Numeral System and I suspect that the average reader of this book does not have the 25 years of experience in Binary math that I do.

I will be explaining more about these operations in a later chapter because they help a lot when trying to optimize programs for size and speed. However they can make programming LOOK complicated and scare away potentially great new programmers who are just trying to learn to apply the 4 regular arithmetic operations of addition, subtraction, multiplication, and division which apply to all number bases.

Chapter 5: Integer Sequences and Their Application in Learning

To be a programmer in any language, a person needs more than information. There must exist the motivation of something you want to make. The challenge is that when you are a beginner, it can be easy to get discouraged because you won’t be making anything big or impressive to other people at the start. All you have to work with in most programming languages is displaying text and numbers.

Later on you can learn to use third party libraries or native APIs for your operating system. However, what I have always disliked is that the internals of how they work are hidden or obfuscated so that you don’t know how they work.

But if you love math like I do, you will never have a problem testing your ability by writing small programs to print integer sequences. In this chapter, I will be sharing 3 of my favorite sequences.

- Fibonacci numbers
- Powers of 2
- Prime Numbers

If you have the ebook edition of this book, you will be able to click the links above and learn more about these sequences. Either way, I will show you the code that makes printing these sequences easy. even in Assembly Language

Fibonacci numbers

```
org 100h
main:

mov word [radix],10
mov word [int_width],1

mov ax,0
mov bx,1

Fibonacci:

call putint
add ax,bx
push ax
mov ax,bx
pop bx
cmp ax,1000
jb Fibonacci

mov ax,4C00h
int 21h
```

```
include 'chastelib16.asm'
```

When executed, that program will output the following sequence

```
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
```

Just by looking at it, hopefully you see the pattern. Each number is the sum of the previous two numbers. The loop in this program needed to swap the numbers in ax and bx each time before the next add. Technically, there are two other ways I could have achieved it. I could have used ecx as a temporary storage. I also could have used the xchg instruction which does the same thing, but the stack provided me a convenient way of doing it. We only needed to save the ax register before it was overwritten with bx, then we pop into bx the pushed ax from earlier. None of these methods is more correct than any other, but I tend to favor simplicity and therefore am limiting the type of instructions I use. I also think that using a third register or even another memory location is acceptable for something like this, but since the stack is already used for function calls and is an expected part of Assembly, there is no reason not to use it, especially when we only need to save one register.

Powers of 2

```
org 100h
main:

mov word [radix],10
mov word [int_width],1
mov [int_newline],0

mov cx,0

mov [array],byte 1

powers_of_two:

;this section prints the digits
mov bx,[length]
array_print:
```

```
dec bx
mov ax,0
mov al,[array+bx]
call putint
cmp bx,0
jnz array_print
call putline

;this section adds the digits
mov dl,0
mov bx,0
array_add:
mov ax,0
mov al,[array+bx]
add al,al
add al,dl
mov dl,0
cmp al,10
jb less_than_ten

sub al,10
mov dl,1

less_than_ten:
mov [array+bx],al
inc bx
cmp bx,[length]
jnz array_add

cmp dl,0
jz carry_is_zero

mov [array+bx],1
inc [length]

carry_is_zero:

;keeps track of how many times the loop has run
add cx,1
cmp cx,64
jna powers_of_two

mov ax,4C00h
int 21h

length dw 1
array db 32 dup 0

include 'chastelib16.asm'
```

Prime Numbers

```
org 100h
main:
```

```
mov word [radix],10
mov word [int_width],1
mov [int_newline],0

;the only even prime is 2
mov ax,2
call putint
call putspace

;fill array with zeros up to length
mov bx,0
array_zero:
mov [array+bx],0
inc bx
cmp bx,length
jb array_zero

;start by filtering multiples of first odd prime: 3
mov ax,3

primes:

;print this number because it is prime
call putint
call putspace

mov bx,ax ;mov ax to bx as our array index variable
mov cx,ax ;mov ax to cx
add cx,cx ;add cx to itself

sieve:
mov [array+bx],1 ;mark element as multiple of prime
add bx,cx ;check only multiples of prime times 2 to exclude even numbers
cmp bx,length
jb sieve

;check odd numbers until we find unused one not marked as multiple of
prime
mov bx,ax
next_odd:
add bx,2
cmp [array+bx],0
jz prime_found
cmp bx,length
jb next_odd
prime_found:

;get next prime read to print in ax
mov ax,bx
cmp ax,length
jb primes

mov ax,4C00h
```

```
int 21h

include 'chastelib16.asm'

length=1000
array rb length
```

How to use these examples

My suggestions is that you download the examples in this chapter from my github repository rather than trying to type them by hand or copy past them. That way you can assemble them with FASM and run them in the DOSBox emulator to see how they work.

https://github.com/chastitywhiterose/Assembly/tree/main/fasm/dos/AAA-DOS-book-examples/

These programs can produce long lists of numbers and so I can't include all the output in this book. You will have to run them to get the full picture of how magnificent they are!

Chapter 6: The strint Function

In this chapter, I will only be introducing one program that uses the three previous functions I described in earlier parts of this book but also includes one more important one!

What this program does is really quite simple, it takes a string of binary integers called “test_int” and converts it into a real integer using a new function called “strint”.

Below is the source of this program. Take a minute to look it over. Afterwards, I will explain more about the “strint” function and how it interacts with the other three functions. “putstring”,“intstr”,and “putint”.

The Program

```
org 100h

main:

mov ax,main_string
call putstring

mov word [radix],2 ; choose radix for integer input/output
mov word [int_width],1

mov ax,test_int
call strint

mov bx,ax

mov ax,str_bin
call putstring
mov ax,bx
mov word [radix],2
call putint

mov ax,str_hex
call putstring
mov ax,bx
mov word [radix],16
call putint

mov ax,str_dec
call putstring
mov ax,bx
mov word [radix],10
call putint

mov ax,4C00h
int 21h
```

```
main_string db "This is the year I was born",0Dh,0Ah,0

;test string of integer for input
test_int db '11111000011',0

str_bin db 'binary: ',0
str_hex db 'hexadecimal: ',0
str_dec db 'decimal: ',0

include 'chastelib16.asm'
```

For a quick review, the 3 previous function do the following.

- putstring: prints a zero terminated string pointed to by ax register
- intstr: converts the integer in ax register into a zero terminated string and then points ax to that string for compatibility with putstring
- putint: calls intstr and then putstring to display whatever number ax equals

And now I introduce to you the final function of my 4 function library that I call “chastelib”.

This function is called “strint” and its importance cannot be overstated. It does the opposite of the “intstr” function. Instead of converting an integer to a string, it does the opposite and takes the string pointed to by ax and converts it into a number returned in the ax register. Much like “intstr”, it uses the global `[radix]` variable to know which base is being used.

But the very nature of turning a string into an integer is more complicated by necessity. Any valid 16 bit number can be turned into a string from bases 2 to 36 by the “intstr” function. However, strings can contain characters that are invalid to be converted as numbers. There is also the issue that both capital and lowercase letters might be used in radixes higher than ten. In the program above, I used a binary integer string for an example, but in real applications, such as a famous program I wrote called “chastehex”, it is necessary to write a function that can gracefully handle not only the decimal digits ‘0’ to ‘9’ but also handle letters ‘A’ to ‘Z’ or ‘a’ to ‘z’.

Below is the full source code of the strint function as written for 16 bit DOS Assembly programs. I spent more time writing this function than the three previous functions combined, but it was necessary for the programs I intended to write! Comments are included although more explanation may be required for some to understand it.

The Function

```
;this function converts a string pointed to by ax into an integer
returned in ax instead
;it is a little complicated because it has to account for whether the
character in
;a string is a decimal digit 0 to 9, or an alphabet character for bases
higher than ten
;it also checks for both uppercase and lowercase letters for bases 11 to
36
```

```
;finally, it checks if that letter makes sense for the base.
;For example, G to Z cannot be used in hexadecimal, only A to F can
;The purpose of writing this function was to be able to accept user
input as integers

strint:

mov bx,ax ;copy string address from ax to bx because ax will be replaced
soon!
mov ax,0

read_strint:
mov cx,0 ; zero cx so only lower 8 bits are used
mov cl,[bx] ;copy byte/character at address bx to cl register (lowest
part of cx)
inc bx ;increment bx to be ready for next character
cmp cl,0 ; compare this byte with 0
jz strint_end ; if comparison was zero, this is the end of string

;if char is below '0' or above '9', it is outside the range of these and
is not a digit
cmp cl,'0'
jb not_digit
cmp cl,'9'
ja not_digit

;but if it is a digit, then correct and process the character
is_digit:
sub cl,'0'
jmp process_char

not_digit:
;it isn't a decimal digit, but it could be perhaps an alphabet character
;which could be a digit in a higher base like hexadecimal
;we will check for that possibility next

;if char is below 'A' or above 'Z', it is outside the range of these and
is not capital letter
cmp cl,'A'
jb not_upper
cmp cl,'Z'
ja not_upper

is_upper:
sub cl,'A'
add cl,10
jmp process_char

not_upper:

;if char is below 'a' or above 'z', it is outside the range of these and
is not lowercase letter
cmp cl,'a'
jb not_lower
```

```
cmp cl,'z'
ja not_lower

is_lower:
sub cl,'a'
add cl,10
jmp process_char

not_lower:

;if we have reached this point, result invalid and end function
jmp strint_end

process_char:

cmp cx,[radix] ;compare char with radix
jae strint_end ;if this value is above or equal to radix, it is too high
despite being a valid digit/alpha

mov dx,0 ;zero dx because it is used in mul sometimes
mul word [radix]    ;mul ax with radix
add ax,cx

jmp read_strint ;jump back and continue the loop if nothing has exited
it

strint_end:

ret
```

If you run the program at the top of this chapter (remember the source is available on github but also can be pieced together from this book alone), it will produce this output.

The Output

```
This is the year I was born
binary: 11111000011
hexadecimal: 7C3
decimal: 1987
```

These three forms of displaying the same number are quite obviously the most useful radixes that programmers must learn.

Binary is what computers understand. Without knowing that everything is bits of only 0 or 1, very little about computers makes sense without a complete understanding of the Binary Numeral System.

Hexadecimal is the short form of Binary because every four bits equals one digit in Hexadecimal. For this reason, hex editors for editing binary files are much more common than binary editors. It just takes less typing and disk space.

Decimal actually has no value other than what humans have placed on it. Base ten is not special, and computers don’t understand it as well as Binary, but humans expect numbers to be in this form, and by extension, all the video games also display numbers in this form.

If I told people that I was born in 11111000011, they will think I am really old and should be dead by now. If I tell them I was born in 1987, they will know that I am 38 years old at the time I am writing this book. However, both numbers mean the exact same thing.

The reason I mention this is that I want you to research different bases/radixes of number systems because it will make you a better programmer. In fact there is a really good book written by another author that I will recommend.

Binary, Octal and Hexadecimal for Programming & Computer Science by Sunil Tanna
https://www.amazon.com/Binary-Hexadecimal-Programming-Computer-Science-ebook/dp/B07F6Y7JX1

To be written:

- Chapter 7: Translating Assembly to Other Programming Languages
- Chapter 8: Going from DOS to Linux or Windows
- Chapter 9: Bitwise Operations for Advanced Nerds
- Chapter 10: chastehex: Not just a program, but a philosophy.

Chapter Z: More Documentation

Below is a list of the sources I referenced the most while writing this book. I respect the work of Ralf Brown and any other people involved in keeping DOS programming information available.

https://www.cs.cmu.edu/~ralf/files.html
https://www.delorie.com/djgpp/doc/rbinter/ix/
https://stanislavs.org/helppc/int_21.html
https://www.ctyme.com/intr/int-21.htm

However, as time goes on, DOS information will become harder to find because old people die and can no longer pay to keep their websites online. This book was my attempt at keeping the information alive as long as I live. I have downloaded as much information onto my computer and have old books that are out of print. The time may come when I am the last person on earth who even knows or cares about the old way of programming in DOS.

And when I die, my only hope is that there is another young autistic programmer who will read my books about computer programming and Chess. May they be inspired to carry on the work of nerdy activities that most will never understand and criticize them for.

If at any time, something I wrote in this book is unclear to you, please email me to help me explain it better for you in future updates to this and other books.

chastitywhiterose@gmail.com