CHAPTER 3

Assembly and Syntax Fundamentals

Objectives

- Distinguish differences in Assembly syntaxes
- Identify sections of Assembly code and explain the use of each
- Construct semantically correct data definitions
- Create working Assembly programs

Outline

1. Web Resources
2. Introduction
3. Basic Elements
   a. Pillars of Assembly Code
   b. Literals
   c. Labels and Comments
4. Data Definition
5. Working Examples
6. Summary
7. Key Terms
8. Code Review
9. Questions
   a. Short Answer
   b. True/False
10. Assignments

Web Resources

- https://sourceware.org/binutils/docs/as/ (GAS Reference)
- https://docs.microsoft.com/en-us/cpp/assembler/masm/microsoft-macro-assembler-reference (MASM Reference)
- https://www.nasm.us/doc/nasmdoc0.html (NASM Reference)

Introduction

This chapter introduces x86_64 Assembly language. Because software engineers often work in various operating systems depending on the type of problem needing solved, this chapter and subsequent chapters examine Assembly language using three prominent assemblers—GAS, MASM, and NASM. The approach offers examples and preparedness for common operating systems (Mac, Windows, and Linux) and development environments. Development environments you might use to write Assembly include Apple Xcode, Microsoft Visual Studio, and command-line assembling within Linux.

We take great care to show the differences in assembler syntaxes, when such differences exist, by denoting code snippets with a border and a title of the assembler for which the code is written. In some instances, the purpose of a code snippet is not to teach syntax specifics but rather to explain a concept. Such code snippets do not have a border or a title. Because different assemblers can use different Assembly syntaxes, we use the convention detailed in Table 3.1.

Table 3.1 Syntax conventions

Assembler	Syntax	Development environment
GAS	AT&T	Apple Xcode
MASM	Intel	Microsoft Visual Studio
NASM	Intel	Command-line in Linux

ATTENTION: Xcode does not use the standard GNU Assembler (GAS). Instead, Xcode uses an integrated assembler with Clang as the front end and LLVM as the back end. While minor differences exist between GAS and the Clang/LLVM integrated assembler, Clang was designed to be a replacement for the GNU toolchain. Thus, Assembly code written in Xcode will also assemble with current versions of GAS All GAS code in this book has been tested in Xcode on macOS and GAS 2.x on Linux to ensure compatibility. For consistency, we refer to the assembler as GAS throughout the text.

By the end of this chapter, you will be familiar with the basic structure and syntax of x86_64 Assembly programs. A solid understanding of Assembly syntax fundamentals is necessary for the remainder of the book.

Basic Elements

Unlike higher-level programming languages, Assembly language operates at a level where each line of code performs a single operation. Assembly cannot be written without knowledge of the computer's architecture, such as CPU registers, flags, floating-point capabilities, etc. Low-level details and coding can be daunting for the novice programmer and the experienced programmer alike, but knowledge of low-level details allows a programmer to write code with control and efficiency unmatched by high-level languages. As you master Assembly language, you will likely find that your programs written in higher-level languages benefit in design.

Pillars of Assembly Code

In order to begin the journey of Assembly programming, five pillars found in Assembly programs are important to discuss: reserved words, identifiers, directives, sections (segments), and instructions. A clear understanding of the five pillars is paramount for your ability to follow future sections of this chapter and book. Let us examine a simple Assembly program in which we will identify the five pillars. Program 3.1 defines a 64-bit variable, adds two numbers, and saves the result in the aforementioned variable.

Reserved words, as in any programming language, are words that can only be used for their defined purpose. For example, MOV is a reserved word because it is an instruction. MOV cannot be used as a variable name or in any other way except to execute the MOV instruction. In Assembly language, reserved words are case-insensitive. Examples of reserved words include instructions (e.g., MOV), directives (e.g., PROC), registers (e.g., *rax*), and attributes (e.g., _main for the global directive).

Table 3.2 shows the sample program with the reserved words appearing in black. Notice that most of the program consists of reserved words. Some of the reserved words are also other pillars, such as directives and code sections.

PROGRAMMING: The GitHub repository contains programs for each of the assemblers, but also three variants per assembler: x86, x86/64, and x86_64. The x86 versions are the 32-bit programs from Edition 1. The x86/64 versions mix 32- and 64-bit code and register usage and are direct translations of the x86 versions to be x86_64 compatible. The x86_64 versions mainly use x86_64 code and registers. For example, we start with Program 3.1 to introduce the basics using only x86_64 code and registers. The data type used is a 64-bit .quad/QUAD/DQ (`long int`), but if we were to use a 32-bit .long/DWORD/DD (`int`), then the compiler would likely mix 32- and 64-bit instructions and register usage, similar to our x86/64 example of Program 3.1 on GitHub.

Program 3.1 *Sample Assembly program*

GAS

```
.data
sum: .quad 0

.text
.global _main
_main:
mov $25, %rax
mov $50, %rbx
add %rbx, %rax
mov %rax, sum(%rip)

mov $0x2000001, %rax
xor %rdi, %rdi
syscall
.end
```

MASM

```
extrn ExitProcess : proc

.data
sum QWORD 0

.code
_main PROC
mov rax, 25
mov rbx, 50
add rax, rbx
mov sum, rax

xor rcx, rcx
call ExitProcess
_main ENDP
END
```

NASM

```
SECTION .data
sum: DQ 0

SECTION .text
global _main
_main:
mov rax, 25
mov rbx, 50
add rax, rbx
mov QWORD [sum], rax

mov rax, 60
xor rdi, rdi
syscall
```

Table 3.2 Sample reserved words

GAS

```
.data
sum: .quad 0

.text
.global _main
_main:
mov $25, %rax
mov $50, %rbx
add %rbx, %rax
mov %rax, sum(%rip)

mov $0x2000001, %rax
xor %rdi, %rdi
syscall
.end
```

MASM

```
extrn ExitProcess : proc

.data
sum QWORD 0

.code
_main PROC
mov rax, 25
mov rbx, 50
add rax, rbx
mov sum, rax

xor rcx, rcx
call ExitProcess
_main ENDP
END
```

NASM

```
SECTION .data
sum: DQ 0

SECTION .text
global _main
_main:
mov rax, 25
mov rbx, 50
add rax, ebx
mov QWORD [sum], rax

mov rax, 60
xor rdi, rdi
syscall
```

Identifiers are programmer-defined names given to items such as variables, constants, and procedures. The length of identifiers is limited to 247 characters and cannot begin with a number. Also, in order to be a valid identifier the name must begin with a letter (`A-Z, a-z`), underscore (`_`), question mark (`?`), at-symbol (`@`), or dollar sign (`$`), though we do not recommend using `?`, `@`, or `$` as part of standard identifiers. Numbers may be used in identifiers as any character after the first character. The following list shows valid identifiers using characters we recommend.

- `userInputValue`
- `sum_of_values`
- `weight1`

Table 3.3 highlights the identifiers in the sample program.

Table 3.3 Sample identifiers

GAS	MASM	NASM

`.data`
`sum: .quad 0`
 	
`.text`
`.globl _main`
`_main:`
`mov $25, %rax`
`mov $50, %rbx`
`add %rbx, %rax`
`mov %rax, sum(%rip)`
 	
`mov $0x2000001, %rax`
`xor %rdi, %rdi`
`syscall`
`.end`
	`extrn ExitProcess : proc`
	
	`.data`
	`sum QWORD 0`
	
	`.code`
	`_main PROC`
	`mov rax, 25`
	`mov rbx, 50`
	`add rax, rbx`
	`mov sum, rax`
	
	`xor rcx, rcx`
	`call ExitProcess`
	`_main ENDP`
	`END`
		`SECTION .data`
		`sum: DQ 0`
		
		`SECTION .text`
		`global _main`
		`_main:`
		`mov rax, 25`
		`mov rbx, 50`
		`add rax, ebx`
		`mov QWORD [sum], rax`
		
		`mov rax, 60`
		`xor rdi, rdi`
		`syscall`

Directives are assembler-specific commands not related to the instruction set that allow you to define variables, indicate memory segments, and many other things. Directives direct the assembler to do something. For example, in the following code snippet the directive QWORD tells the assembler to reserve 64 bits of memory (a quadword), which we initialize with a value of 42, and assign the name "answer" to that specific memory block for future programmer use.

```
answer QWORD 42          ; QWORD is the directive
```

Table 3.4 shows the sample program with the directives highlighted.

Table 3.4 Sample directives

GAS	MASM	NASM

`.data`
`sum: .quad 0`
 	
`.text`
`.global _main`
`_main:`
`mov $25, %rax`
`mov $50, %rbx`
`add %rbx, %rax`
`mov %rax, sum(%rip)`
 	
`mov $0x2000001, %rax`
`xor %rdi, %rdi`
`syscall`
`.end`
	`extrn ExitProcess : proc`
	
	`.data`
	`sum QWORD 0`
	
	`.code`
	`_main PROC`
	`mov rax, 25`
	`mov rbx, 50`
	`add rax, rbx`
	`mov sum, rax`
	
	`xor rcx, rcx`
	`call ExitProcess`
	`_main ENDP`
	`END`
		`SECTION .data`
		`sum: DQ 0`
		
		`SECTION .text`
		`global _main`
		`_main:`
		`mov rax, 25`
		`mov rbx, 50`
		`add rax, ebx`
		`mov QWORD [sum], rax`
		
		`mov rax, 60`
		`xor rdi, rdi`
		`syscall`

At this point we need to discuss some major differences with MASM directives. You may have noticed that, before the data section, the beginning of the MASM sample program has code that GAS and NASM do not have. The MASM directive used is necessary for 64-bit programs; it defines the prototype for the ExitProcess function, which is necessary for properly exiting your program. If, however, you are writing 32-bit MASM code, Table 3.5 lists and describes the MASM-specific directives used at the beginning of those programs

Table 3.5 32-bit MASM-specific directives

Directive	Description
.386	Enables the 80386 processor instructions and disables newer instructions. Other valid settings to enable additional instructions are .486, .586, .686, .MMX, and .XMM, among others.
.MODEL	Sets the memory model. The only valid parameter for 32-bit programs is FLAT (protected mode). The .MODEL directive also takes a second parameter to set the function-calling convention, which is discussed in CHAPTER 6.
.STACK	Sets the size of the stack memory segment for the program. The directive cannot be used without the .MODEL directive. While the default value is 1024, we recommend using 4096 to make stack the same size as a memory page in 32-bit Windows.

Program sections, also called segments, are denoted with directives that specify the segment for the assembler. Segments are special sections predefined by the assembler. Table 3.6 lists the common Assembly sections for use in programs.

Table 3.6 Assembler-specific program sections

Directive			Description
GAS	MASM	NASM	
.bss	.data	SECTION .bss	Uninitialized variables
.data		SECTION .data	Initialized variables
.text	.code	SECTION .text	Executable code/instructions

Table 3.7 shows the sample program with the section directives highlighted.

Table 3.7 Sample program sections

GAS	MASM	NASM

.data
sum: .quad 0
 	
.text
.global _main
_main:
mov $25, %rax
mov $50, %rbx
add %rbx, %rax
mov %rax, sum(%rip)
 	
mov $0x2000001, %rax
xor %rdi, %rdi
syscall
.end
	extrn ExitProcess : proc
	
	.data
	sum QWORD 0
	
	.code
	_main PROC
	mov rax, 25
	mov rbx, 50
	add rax, rbx
	mov sum, rax
	
	xor rcx, rcx
	call ExitProcess
	_main ENDP
	END
		SECTION .data
		sum: DQ 0
		
		SECTION .text
		global _main
		_main:
		mov rax, 25
		mov rbx, 50
		add rax, ebx
		mov QWORD [sum], rax
		
		mov rax, 60
		xor rdi, rdi
		syscall

The section on Data Definition later in this chapter further explains how program segments are used and what information they contain.

Instructions are the executable statements in a program that we begin covering in detail in **CHAPTER 4**; however, a preliminary explanation is warranted here. Instructions are comprised of two basic parts defined by the following syntax.

```
mnemonic [operands]
```

The **mnemonic** is the instruction name a programmer uses to refer to a particular instruction in the architecture's instruction set. A mnemonic is an abbreviation or acronym that identifies the action of the instruction and in reality is an English-like representation of a numeric opcode (as shown in **CHAPTER 1**). Table 3.8 highlights the instructions in the sample program.

Table 3.8 Sample instructions

GAS	MASM	NASM

`.data`
`sum: .quad 0`
 	
`.text`
`.globl _main`
`_main:`
`mov $25, %rax`
`mov $50, %rbx`
`add %ebx, %rax`
`mov %rax, sum(%rip)`
 	
`mov $0x2000001, %rax`
`xor %rdi, %rdi`
`syscall`
`.end`
	`extrn ExitProcess : proc`
	
	`.data`
	`sum QWORD 0`
	
	`.code`
	`_main PROC`
	`mov rax, 25`
	`mov rbx, 50`
	`add rax, rbx`
	`mov sum, rax`
	
	`xor rcx, rcx`
	`call ExitProcess`
	`_main ENDP`
	`END`
		`SECTION .data`
		`sum: DQ 0`
		
		`SECTION .text`
		`global _main`
		`_main:`
		`mov rax, 25`
		`mov rbx, 50`
		`add rax, ebx`
		`mov QWORD [sum], rax`
		
		`mov rax, 60`
		`xor rdi, rdi`
		`syscall`

Some instructions do not require any operands, some require one, some two, and some three. The following lines of code are examples of instructions that require a different number of operands.

```
stc                 ; no operands, sets the carry flag
inc eax             ; 1 operand, increments eax by 1
mov eax, 5          ; 2 operands, moves literal value 5 into eax
imul eax, ebx, 5    ; 3 operands, multiplies literal value 5 and the
                    ; value in ebx and stores the result in eax
```

LEARNING: At this point it will be helpful to use Appendices B and D to write, execute, and debug Program 3.1 to ensure that you have a working programming environment and get a feel for the basics. Before continuing through the chapter, make sure the sample program assembles, and use a breakpoint to watch register values change as you step through the program line by line. Then, you can modify the program as you learn new concepts throughout the chapter and reinforce your understanding of each concept.

Literals

Literal values, often called **immediates**, are explicit values specified by the programmer, such as integers, real numbers, characters, and strings. For example, the following line of code would add the literal value 5 to the current contents of the 64-bit register *rax*.

```
add rax, 5
```

PROGRAMMING: Immediates are used in cases where the programmer knows the exact value to be used in an instruction. Many programs rely on user input for data, but in other cases constant values are useful. For example, consider a simple program that converts a person's weight from pounds to kilograms. The program would expect one piece of data from the user (weight in pounds) and would multiply pounds by a conversion factor (2.2) in order to convert the weight to kilograms. The conversion factor is a constant value that can be specified as a literal value.

By default, **integer literals** are whole-number decimal (base-10) values. In some situations, using a different base for literal values is useful or even necessary. In MASM and NASM, writing numbers in other bases can be done by appending a radix (numeric base) character to the literal value, as shown in Table 3.9.

Table 3.9 MASM/NASM integer radix characters

Radix	Base
`b`	Binary (base-2)
`d`	Decimal (base-10)
`h`	Hexadecimal (base-16)
`q, o`	Octal (base-8)

The following examples show how to encode the base-10 value 31 in each of the supported bases with MASM and NASM.

MASM/NASM

```
00011111b   ; b is the radix character for binary
31          ; decimal values do not need radix characters
31d         ; but you can specify d for decimal
1Fh         ; h is the radix character for hexadecimal
37o         ; o is the radix character for octal
```

A case worth noting is when a hexadecimal value begins with a character, such as memory addresses. When a literal value begins with a character, the assembler interprets it as an identifier instead of a value. The conflict can be overcome by prefixing the hexadecimal value with a zero.

MASM/NASM

```
0FFFF0342h   ; the actual value is FFFF0342 in hexadecimal
```

GAS takes a different approach to integer literals depending on the program section (sections were discussed in Pillars of Assembly Code). As shown in Table 3.10, literal values in the *.data* section must be prefixed with radix characters to delineate the base, while literal values in the *.text* section must be prefixed with the dollar sign (`$`) in addition to the radix characters. Base-10 numbers are the exception, which are written without radix characters.

Table 3.10 GAS prefix and radix characters

.data Prefix	*.text* Prefix	Base
`0b`	`$0b`	Binary (base-2)
n/a	`$`	Decimal (base-10)
`0x`	`$0x`	Hexadecimal (base-16)
`0`	`$0`	Octal (base-8)

The following examples show how to encode the base-10 value 31 in both the *.data* and *.text* sections using GAS syntax.

GAS

```
.data                    .text
0b00011111               $0b00011111
31                       $31
0x1F                     $0x1F
037                      $037
```

Real numbers (floating-point values) are much more complex than integers and so we cover real numbers in **CHAPTER 8**.

Character literals are single character values that, like integer literals, are explicit values specified by the programmer. Single quotes or double quotes can be used to enclose a character literal in MASM and NASM, thus making the following two values equivalent.

MASM/NASM

```
"A"
'A'
```

In GAS, character literals are specified differently depending on the section. In the *.data* section, characters are surrounded by single quotes while in the *.text* section characters are also prefixed with the dollar sign (`$`).

GAS

```
.data                    .text
'A'                      $'A'
```

ATTENTION: As described in **CHAPTER 1**, characters are stored in memory as ASCII-encoded values (integers). For example, the letter `'A'` is stored in memory as 65 decimal (41 hexadecimal).

String literals are multiple character literals grouped together, typically forming a word or phrase. Like characters, strings can be enclosed in either single or double quotes in MASM and NASM, but GAS requires double quotes. Sometimes you may want quotes to be part of a string. In MASM and NASM, you must use the opposite type of quotes to enclose the string. With GAS, you must escape the inner quote(s) with a backslash (`\`) so the string is not prematurely ended.

GAS

```
"This string \"contains\" double quotes!"
```

MASM/NASM

```
"I don't understand contractions."              ; strings that have one
'"Good job," said the father to his son.'       ; type of quotes on the
                                                ; outside and a different
                                                ; type on the inside
```

Strings are usually stored as a byte array, with each byte containing an ASCII-encoded value of an individual character in the string. The following example shows a string literal and the individual array elements stored in memory as ASCII decimal values.

String Characters	D	a	i	s	y	,		d	a	i	s	y
ASCII Decimal Values	68	97	105	115	121	44	32	100	97	105	115	121

Labels and Comments

Labels give you the ability to partition code for programmatic or design purposes. Labels not only allow for greater clarity when reading code, but more importantly they facilitate jumping or looping to different parts of a program when necessary. Labels are created by using an identifier followed by a colon.

```
identifier:
```

Labels can be created on their own line or on the same line as an instruction. Depending on the assembler, labels are either case-sensitive and used in the *.text* section (GAS and NASM), or case-insensitive and used in the equivalent *.code* section (MASM).

```
userLoop:
            inc counter

otherLoop: inc counter2
```

Chapter 5 discusses using labels to implement loops in Assembly language.

Comments are an integral part of every program. Comments allow you to explain the *why* and *how* of the code (as opposed to the *what*, which is usually more obvious). Comments are very important with Assembly code since more abstract objectives are not always easily comprehendible when reading a sequence of primitive instructions.

Two types of comments exist in Assembly: single-line comments and multi-line comments. **Single-line comments** begin with a hash (`#`) in GAS or a semi-colon (`;`) in MASM and NASM. A comment can be on its own line or be appended to any existing line of code.

GAS

```
# Moves the counter value into rax
mov counter(%rip), %rax
```

MASM/NASM

```
mov rax, counter ; Moves the counter value into rax
```

Multi-line comments can only be used in GAS and MASM. In GAS, multi-line comments are identical to C-style multi-line comments, beginning with `/*` and ending with `*/` and they can appear anywhere in the code. In MASM, multi-line comments must be separate from other lines of code and they consist of four parts: the word `COMMENT`, a beginning character, the comment text, and an ending character, which must be the same as the beginning character. One caveat to MASM multi-line comments is that the character used to begin and end the comment cannot be found within the comment text. Traditionally, the exclamation point (`!`) is used for multi-line comments in MASM.

GAS

```
mov $10, %rax        /* This is a multi-line comment that is
                     partially on the same line as
                     an instruction and partially on
                     separate lines */
```

MASM

```
COMMENT !
     This is the section of the code where
     employee salaries are calculated. Note
     how the exclamation point is not in the
     text of the comment.
!
```

Data Definition

Data types in Assembly language are indicative of their size (8 bits, 16 bits, 32 bits, etc.) rather than their contents (integer, double, string, etc.) as in high-level programming languages. Data, no matter the contents, is defined using a default set of data types. Table 3.11 lists the data type naming conventions by assembler.

Table 3.11 Default data type directives

Directive			Description
GAS	MASM	NASM	
.byte, .ascii	DB, BYTE	DB	1 byte (8-bit) integer
	SBYTE		1 byte (8-bit) signed integer
.word	DW, WORD	DW	2 byte (16-bit) integer
	SWORD		2 byte (16-bit) signed integer
.long	DD, DWORD	DD	4 byte (32-bit) integer
	SDWORD		4 byte (32-bit) signed integer
.quad	DQ, QWORD	DQ	8 byte (64-bit) integer
	DT, TBYTE	DT	10 byte (80-bit) integer
.octa			16 byte (128-bit) integer

Table 3.11 appears to suggest that MASM can work with more data types than GAS or NASM, but that is not the case. MASM performs many of its operations by assuming the data type from context, whereas GAS and NASM do not make assumptions. You can work with all of the same data in all three assemblers. However, as you will learn in future chapters, in GAS and NASM you must explicitly tell the assembler what data types (size) are being used when executing an instruction.

PROGRAMMING: Choosing the correct size for variables is important. A programmer should anticipate the possible values that will be stored in a variable to avoid semantic errors. Semantic (logic) errors will not halt the assembling or execution of a program, but they can produce incorrect results when the program runs. For example, if you create two 8-bit variables and ask the user to enter two values to be multiplied together, the result of the multiplication could be larger than 8 bits. So, you would need at least a 16-bit variable for the result. Using variables that are too small for their intended use could result in the loss of data, and thus an incorrect result.

A **variable** definition has the following syntax.

GAS/NASM

```
[identifier:] directive initializer [,initializer]...
```

MASM

```
[identifier]  directive initializer [,initializer]...
```

All variables defined in the *.data* section (for all assemblers) must be initialized (given an initial value). The following examples show valid variable definitions.

GAS

```
val1:       .byte 17
valArray:   .quad 0xFFFFFF, 0xFFFFFE, 0xFFFFFD
```

MASM

```
charInput   BYTE 'A'
myArray     QWORD 41h, 75, 0C4h, 01010101b
```

NASM

```
counter:    DB 0
wageArray:  DQ 75, 100, 125
```

Notice the examples that use multiple initializers separated by commas. Using multiple initializers for an identifier is the method for creating an **array**. An array is a sequence of same-size values referenced by a single name that identifies the first location/value in the sequence. Instead of creating four separate identifiers, we created one identifier that represents the start of a four-value sequence, each one quadword (64 bits) in size. Each array identifier is actually the memory reference of the first value. We address methods of accessing values in an array in **Chapter 4**.

Sometimes **uninitialized variables** are necessary in a program. Uninitialized variables are variables that are not given an initial value; they simply reserve a certain amount of memory for programmer use. Different assemblers accomplish uninitialized memory allocation in very different ways. MASM offers the question mark (`?`) as an initializer so that uninitialized variables can be created in the *.data* section, which is only meant for initialized variables in GAS and NASM.

MASM

```
.data
num DWORD  6                ; defines an initialized identifier
sum SDWORD ?                ; defines an uninitialized identifier
myArray   BYTE 10 DUP (1)   ; defines an array of initialized bytes
myUArray  BYTE 10 DUP (?)   ; defines an array of uninitialized bytes
```

The **DUP** directive can be used in MASM to create duplicate values of a specified size and initializer in sequence. Any valid initializer can be used, including the question mark. In the MASM example, `myArray` is an array of 10 byte values each initialized to `1`, and in `myUArray` each is initialized to `?` (uninitialized).

GAS and NASM require that all uninitialized variables be created in the *.bss* (Block Started by Symbol) program segment. Both GAS and NASM require you to use data type directives specifically for uninitialized data, which are listed in Table 3.12.

Table 3.12 Uninitialized data type directives

Directive		Description
GAS	NASM	
`.lcomm`	`RESB`	Reserve a byte (8 bits)
	`RESW`	Reserve a word (16 bits)
	`RESD`	Reserve a double word (32 bits)
	`RESQ`	Reserve a quad word (64 bits)
	`REST`	Reserve a ten-byte (80 bits)

The following examples show valid uninitialized variable definitions in GAS and NASM. Note how defining uninitialized variables in GAS differs from the syntax rule of `[identifier: directive initializer]`. Instead, the format is `[directive identifier, reserved_bytes]`. You can use the uninitialized data type directives in GAS and NASM to reserve memory for a single variable or an array. For example, in the NASM code we create a 64-byte `buffer`, which can be used as an array containing 64 values.

GAS

```
.bss
.lcomm sum, 2          # Reserve 2 bytes
.lcomm answer, 4       # Reserves 4 bytes
```

NASM

```
SECTION .bss
memAddr: RESD 1        ; Reserves 1 DWORD (4 bytes)
buffer:  RESB 64       ; Reserves 64 bytes
```

As previously explained, **strings** are stored as BYTE arrays. Strings need to be null-terminated, which means that the last byte must be ASCII-zero. Null-terminating a string is achieved differently depending on the assembler. GAS uses the `\0` sequence to add the null terminator to the end of a string, while MASM and NASM just use the literal value `0` as the last byte. GAS also has an .ASCIZ directive that automatically adds a zero byte to the end of a string.

Line breaks are also an important aspect of strings. Inserting line breaks is different depending on the assembler. In GAS, the escape sequence `\n` will insert a newline (line-feed) character. In MASM, the CR/LF (carriage-return / line-feed) hexadecimal codes `0Dh` and `0Ah` need to be inserted where a line break is desired. In NASM, only the LF (line-feed) hexadecimal code `0Ah` needs to be inserted.

GAS

```
# The \n in the middle of the string adds a newline
motd: .ascii "The only way to win\nis not to play\0"
motd: .asciz "The only way to win\nis not to play"
```

MASM

```
; motd contains a single-line string
motd  BYTE "Welcome to Earth...",0

; motd2 contains a multi-line string with a newline at the end
motd2 BYTE "Thank you for using our system.",0Dh,0Ah
      BYTE "All of your activity will be monitored"
      BYTE "by our system administrators",0Dh,0Ah,0
```

NASM

```
; The 0Ah in the middle of the string adds a newline
motd: DB "How about a nice",0Ah,"game of chess",0
```

Symbolic constants can be used in MASM in lieu of variables when you have certain values in a program that never change as a result of the program's execution. Symbolic constants hold 32-bit integers in x86 and 64-bit integers in x86_64 and are defined with the equal sign (=). Symbolic constants are for integer-based data.

MASM

```
identifier = value
```

Symbolic constants have some advantages. If you are going to use a value multiple times throughout a program, such as an employee's hourly wage, assigning the value to a regular identifier has two distinct advantages: (1) use of a name, such as `HOURLY_WAGE`, throughout a program makes the code more readable, and (2) having the numeric value in a single place makes changes easier should the hourly wage go up or down. Symbolic constants in MASM provide the same two advantages plus one more: symbolic constants do not use any memory. At assembly time, MASM replaces all instances of a symbolic constant with the programmer-defined value.

MASM

```
.data
HOURLY_WAGE = 75
.code
mov eax, HOURLY_WAGE
```

When assembled, the above code will be transformed automatically and instances of the symbolic constant will be replaced with the literal value.

```
mov eax, 75    ; MASM automatically replaces the symbol with the value
```

The symbolic constant example using the equal sign only works with the MASM assembler. Symbolic constants can also hold string values in MASM.

All assemblers can make use of symbolic constants when they hold expressions. In order to define a symbol that holds an expression, the **EQU** directive must be used. Unlike the previously explained methods of defining variables, symbols can be created with EQU in both the data and code segments (i.e., *.data* and *.text* for GAS, *.data* and *.code* for MASM, and *SECTION .data* and *SECTION .text* for NASM). The assemblers will replace every occurrence of a symbol with the expression at assembly time.

GAS

```
.equ identifier, expression    # numeric operations
.equ identifier, value         # a single value (constant)
.equ identifier, "symbol"      # another existing symbol
```

MASM

```
identifier EQU expression      ; numeric operations
identifier EQU symbol          ; another existing symbol
identifier EQU <text>          ; useful for non-integer data (float, string)
```

NASM

```
identifier: EQU expression     ; numeric operations
identifier: EQU value          ; a single value (constant)
identifier: EQU "text"         ; text can be four characters max (32-bit)
```

As demonstrated in the examples, symbolic constant syntax and behavior is very different in the assemblers. We recommend limiting the use of EQU to numeric expressions.

```
.equ test, (2 * 6 / 3)    # GAS
test EQU (2 * 6 / 3)      ; MASM
test: EQU (2 * 6 / 3)     ; NASM
```

EQU can also be useful for creating a constant symbol for the length of an identifier's memory space. The GAS and NASM examples below show how to use an expression to get the size of a string in bytes. The **current location counter**, the period (`.`) in GAS and the dollar sign (`$`) in MASM and NASM, represents the memory address of where the counter symbol exists in code. The current location counter can be used to subtract the starting memory address of the previous string from the current location, resulting in the size of the string in bytes. In MASM, the equal sign (=) can be used with the current location counter.

GAS

```
motd: .byte "Are you in the matrix?\0"
.equ len, (. - motd)
```

MASM

```
motd  BYTE "Are you in the matrix?",0
len = ($ - motd)
```

NASM

```
motd: DB "I'm in the matrix?",0
len:  EQU ($ - motd)
```

Another feature MASM offers is a way to create a symbol that is dynamic, called a text macro. The **TEXTEQU** directive is used to create such symbols, which can be expressions using other symbols and even instructions.

MASM

```
identifier TEXTEQU %(expression)
identifier TEXTEQU textmacro
identifier TEXTEQU <text>
```

Note how the instructions in the following example are not enclosed in quotes but are enclosed in angled brackets, since instructions are non-integer expressions.

MASM

```
MULTIPLIER = 2
freq    TEXTEQU %(400 * MULTIPLIER)
movFreq TEXTEQU <mov eax, freq>
```

With the `movFreq` text macro created, you could use it at any point in your code to execute the instruction (`mov eax, freq`).

MASM

```
.code
movFreq
```

Working Examples

The basic information explained in this chapter is crucial to comprehending subsequent chapters. Although **CHAPTERS 4 AND 5** cover instructions in detail, it is important to see this chapter's content used in the context of a working Assembly program. The Program 3.2 examples are the same Assembly program written for the three assemblers. For brevity, the GAS version is fully commented, while the MASM and NASM programs are shown with just the code. Fully commented versions for all three assemblers are available for download (https://github.com/brianrhall/Assembly).

LEARNING: Using your preferred development environment, write and execute one of the following code examples and ensure you have a working program. Then, purposefully create syntax errors in the program to see how those errors are described by the system. Errors such as mistyping variable names, using mismatched sizes with variables and registers, and forgetting a colon with program sections are typical errors that a programmer might make. Knowing what you did to cause an error and seeing the error message can help you decipher error messages you encounter in the future.

Program 3.2 *Working example*

GAS

```
.data                              # Section for variable definitions

decimalLiteral:    .byte 31        # Variable storing 31
hexLiteral:        .quad 0xF       # Variable storing F (15 in decimal)
charLiteral:       .byte 'A'       # Variable storing 65 in decimal

# Variable containing a string that has a line break and is null-terminated
stringLiteral:     .ascii "This string has\na line break in it.\0"

# Variable that calculates the value of an expression to determine the
# length, in bytes, of the variable "stringLiteral" by subtracting the
# starting memory address of the variable from the current memory address
.equ lenString, (. - stringLiteral)

.bss                               # Section for uninitialized variables
.lcomm unInitVariable, 8           # Uninitialized, 8-byte variable

.text                              # Section for instructions
.global _main                      # Make the label "_main"
                                   # available to the linker as an
                                   # entry point for the program
_main:                             # Label for program entry

# Label and instruction on
# the same line below
partOne: movl $10, %rax            # Assign 10 to the rax register
addl hexLiteral(%rip), %rax        # Add the value in hexLiteral to
                                   # the contents of the rax register
                                   # and store the result in rax

partTwo:                           # Label on its own line
inc %rax                           # Increment the value in rax

mov $0x2000001, %rax               # Set the system call for exit
xor %rdi, %rdi                     # Set the return value to 0
movl $1, %eax                      # Set the system call to 1 for exit
syscall                            # Issue the kernel interrupt
.end                               # End assembling
```

MASM

```
extrn ExitProcess : proc

.data
decimalLiteral  BYTE 31
hexLiteral      QWORD 0Fh
charLiteral     BYTE 'A'
unInitVariable  QWORD ?

stringLiteral
  BYTE "This is a stringthat",0dh,0ah
  BYTE "has a line break in it.",0
lenString       EQU ($ - stringLiteral)

.code
_main PROC

partOne: mov rax, 10
add rax, hexLiteral

partTwo:
inc rax

xor rcx, rcx
call ExitProcess
_main ENDP
END
```

NASM

```
SECTION .data

decimalLiteral:   DB 31
hexLiteral:       DQ 0Fh
charLiteral:      DB 'A'

stringLiteral:
  DB "This is a string that",0ah
  DB "has a line break in it.",0
lenString: EQU ($ - stringLiteral)

SECTION .bss
unInitVariable:   RESQ 1

SECTION .text
global _main

_main:

partOne: mov rax, 10
add rax, [hexLiteral]

partTwo:
inc rax

mov rax, 60
xor rdi, rdi
syscall
```

PROGRAMMING: Program 3.2 demonstrates how to properly terminate (exit) an Assembly program for the given environments (i.e., 64-bit Mac, Windows, and Linux) via system calls. Other exit routines are necessary in different environments (e.g., GAS on Linux, 32-bit, etc.). **Appendix A** includes 32-bit and 64-bit exit routines for GAS, NASM, and MASM, while system calls are covered in **Chapter 10**.

Summary

In this chapter, we covered the basic elements of Assembly programs. You should now understand Assembly structure and program sections. You should also be able to define and initialize data in an Assembly program. We discussed important differences between the assembler syntaxes, from creating variables to making comments. The two working programs for each assembler in this chapter helped you get started with writing and assembling code. The programs can serve as templates for future programs.

Key Terms

array
character literals
comments
current location counter
directives
DUP
EQU
identifiers
immediate
instructions
integer literals
labels
mnemonic
multi-line comments
program sections
real numbers
reserved words
single-line comments
string literals
strings
symbolic constants
TEXTEQU
uninitialized variables
variable

Code Review

.ASCIZ	GAS data directive that automatically adds a zero byte to the end of a string
ADD	Instruction that adds the value in one register/variable to another register/variable
ADDL	GAS instruction that adds the value in one register/variable to another register/variable (long integer)
DUP	MASM directive to create duplicate values of a specified size and initializer in sequence
EQU, .equ	Directive to create a new symbol containing the result of an expression
INC	Instruction that increments the value in a register/variable
MOV	Instruction that copies the value in one register/variable to another register/variable
MOVL	GAS instruction that copies the value in one register/variable to another register/variable (long integer)
STC	Instruction that sets the processor's carry flag
TEXTEQU	MASM directive that creates a new symbol containing the resulting value of a text substitution

Questions

Short Answer

1. A _____ value is a value directly specified by the programmer rather than the result of an expression.
2. By default, integer literals are in base _____.
3. In order to use the base-10 value `50` as a hexadecimal value in NASM, you would specify it as _____.
4. In order to use the hexadecimal value `0x34` as a binary value in GAS, you would specify it as _____.
5. Character literals are stored as _____ in memory.
6. This book recommends only using the following characters in identifier names: _____, _____, and _____.
7. _____ are assembler-specific commands that allow you to do many things, such as define variables, indicate memory segments, and so on.
8. Labels must be followed by a _____.
9. _____ and _____ are the only assemblers that make use of multi-line comments.
10. The _____ character signifies a single-line comment in MASM.

11. The _____ directive is used to declare a 32-bit signed integer variable in MASM.
12. The _____ directive is used to reserve 64-bits of uninitialized memory in NASM.
13. In MASM, a newline in a string is represented by the _____ hexadecimal value(s). In NASM, a newline in a string is represented by the _____ hexadecimal value(s).
14. The EQU directive can be used with the _____ to determine the length of a string.
15. An abbreviated version of a longer word or words that explains the action of an instruction is a(n) _____.

True/False

1. The semicolon represents a single-line comment in GAS.
2. Instructions may not use any operands.
3. The current location symbol in NASM is the dollar sign.
4. The *.octa* directive is used to declare an 80-bit variable in GAS.
5. Uninitialized variables are declared in the *.bss* section of a program in MASM.

Assignments

Assignments 1 and 2 require you to use a specific assembler. Since one of the purposes of this chapter is to introduce you to differences in assemblers, we ask you to demonstrate what you have learned. Assignment 3 does not have any specific assembler requirement; solve the problem using the assembler with which you are most comfortable.

3.1 Defining Variables

Create a program for GAS that contains the following items:

- A `sum` variable of the appropriate size to hold the initializer `0x10000`
- A `message` variable that holds the text (preserving line breaks):

```
Welcome to Assembly programming.
Your grade will be randomly assigned
by the Intel 8086 processor!
```

- An `input` variable that does not contain an initial value and is able to hold the values `0-255`
- An instruction to assign the base-10 value 200 to the `input` variable as a binary literal

3.2 Using Expressions with Symbols

Create a program for NASM that contains the following items:

- A `message` variable that holds the text (preserving line breaks):

```
You already know what the next
variable will be, don't you?
```

- A `length` variable that holds the size of the `message` variable
- A `length5` variable (also in the data section) that holds the result of adding 5 to `length` (use EQU)
- A one-operand instruction that adds `1` to the value in the `length5` variable (think outside the box for this requirement)

3.3 Syntax Translation

Translate the following GAS program into either MASM or NASM. Refer to **Appendix A** and the first Programming Note in Chapter 4 (p. 60).

```
# Syntax translation - GAS
.data

.bss
.lcomm letter, 1
.lcomm r, 4
.lcomm s, 4
.lcomm t, 4
.lcomm x, 2
.lcomm y, 2
.lcomm z, 2

.text
.globl _main
_main:

movb $0x77, letter
movl $0x5, r
movl $0x2, s
movw $0xa, x
movw $0x4, y

movw x, %ax
addw y, %ax
movw %ax, z

movw x, %ax
subw y, %ax
movw %ax, z

movl $0x0, %edx
movl r, %eax
movl s, %ecx
divl %ecx
movl %eax, t

movl $0x0, %edx
movl r, %eax
movl s, %ecx
divl %ecx
movl %edx, t

pushl $0
subl $4, %esp
movl $1, %eax
int $0x80
.end
```

3.4 Order Of Operations (Challenge Assignment)

Create a program that calculates the following expression: `answer = (A + B) - (C + D)`

- The answer must be stored in a variable of the correct data type given your data (`A, B, C, D`).
- The values for your data (`A, B, C, D`) must be stored in registers (e.g., *eax*, *ebx*), not variables.
- You must supply the initial values for the data (`A, B, C, D`).
- Create a string that *could* accompany the answer in an output statement (e.g., `"The answer is:"`). You do not have to output the string.
- Comment each line of code, as demonstrated in the Working Examples Section Program 3.2 for GAS, to briefly describe each line's meaning.