64 Bit Programming

Make pdfs for seperate modules of these notes:

Here is a summary of the differences between the 32-bit and 64-bit versions of the AddTwoSum program:

ASPECT	32 BIT VERSION	64 BIT VERSION
Directives	.386 , .model flat, sdtcall, .stack 4096	Not used in 64 bit code
Parameter Definitions	ExitProcess PROTO, dwExitCode: DWORD	ExitProcess PROTO
Program Termination	INVOKE ExitProcess, 0	mov ecx, 0 call ExitProcess
Entry Point Specification	.start:	Not specified(main is the assumed entry point)

Example: Modifying AddTwoSum for 64-Bit:
The text mentions taking a sample program called "AddTwoSum" and modifying it for 64-bit programming. Here's a simplified version of the modifications:

• In 64-bit programming, there's no need to specify certain directives like .386, .model flat, stdcall, and .stack 4096. These were necessary in 32-bit programming but are not used in 64-bit code.

• The PROTO keyword used in 64-bit programming doesn't require parameter definitions. In contrast, 32-bit code would have specified parameters like ExitProcess PROTO, dwExitCode:DWORD. In 64-bit, you

simply use ExitProcess PROTO.

- In the 64-bit version, lines 14 and 15 are used to terminate the program. They use the mov and call instructions. In 32-bit programming, an INVOKE statement might have been used for this purpose, but 64-bit MASM doesn't support INVOKE.

- Line 17 doesn't specify a program entry point in the end directive in the 64-bit version. In 32-bit programming, an entry point might have been specified.

```
;32 bit programming
.386
.model flat, stdcall
.stack 4096

ExitProcess PROTO, dwExitCode:DWORD

.data
sum DWORD 0

.code
main PROC
    mov eax, 5
    add eax, 6
    mov sum, eax

    INVOKE ExitProcess, 0
main ENDP

END main
```

```
;64 bit programming
.data
sum DWORD 0

.code
main PROC
    mov eax, 5
    add eax, 6
    mov sum, eax

    mov ecx, 0
    call ExitProcess
main ENDP

ExitProcess PROTO

END
```

In 32-bit assembly programming, it's a common practice to declare function prototypes like ExitProcess PROTO even before they are used in the code for several reasons:

1. Linker and Library Compatibility: The PROTO declaration serves as a prototype or declaration for a function to the assembler and linker. It informs them about the function's name, parameters, and

calling convention. This information is important for ensuring that the program can be linked correctly and that library functions are used in a compatible manner.

2. Readability and Documentation: Declaring function prototypes at the beginning of your code makes the code more readable and self-documenting. It provides a clear indication of what functions will be used in the program and what parameters they expect.

3. Error Checking: It allows the assembler to perform basic error checking, such as verifying that the function names and parameters used in the code match the declared prototypes. This can help catch typos and errors early in the development process.

4. Modularity: When working with larger programs or codebases, declaring function prototypes separately allows for modularity. You can define functions in separate source files or libraries and include their prototypes at the beginning of your main program, promoting code organization and separation of concerns.

5. Compatibility Across Assemblers and Tools: Using PROTO declarations is a standard and portable way of specifying function interfaces in assembly language. This practice ensures that your code can be easily understood and maintained by other assembly programmers and tools.

While it may seem a bit redundant to declare ExitProcess PROTO at the beginning of a small program like "AddTwoSum," this practice becomes more valuable in larger and more complex assembly projects where code organization and readability are critical.

In addition to these specific differences, there are also some general differences between 32-bit and 64-bit assembly language programming:

• 64-bit processors have more registers than 32-bit processors, and the registers are 64 bits wide

instead of 32 bits wide.
• 64-bit processors can address more memory than 32-bit processors.
• 64-bit assembly language programs typically use different calling conventions than 32-bit assembly language programs.

Here are some additional things to keep in mind when programming for 64-bit processors:

• Use 64-bit data types and registers whenever possible.
• Be aware of the different calling conventions used by 64-bit processors.
• Test your code thoroughly on a 64-bit system before deploying it.

Overall, 64-bit programming is very similar to 32-bit programming. However, there are some important differences to be aware of.

```
; AddTwoSum_64b.asm

.data
sum QWORD 0

.code
main PROC

; Move the value 5 into the 64-bit register RAX.
mov rax, 5

; Add the value 6 to RAX.
add rax, 6

; Move the value of RAX to the 64-bit variable sum.
mov sum, rax

; Exit the program.
mov ecx, 0
call ExitProcess

main ENDP
END
```

This program is similar to the 32-bit version of the AddTwoSum program, but it uses 64-bit registers and variables instead. The only changes that needed to be made were:

- In the data section, the DWORD type was changed to QWORD to declare the sum variable.
- In the code section, the EAX register was changed to RAX wherever it was used.

To compile and run this program, you need to use the 64-bit version of MASM and be running the 64-bit version of Windows.
Here are some additional things to keep in mind when using 64-bit registers and variables:

- 64-bit registers and variables are twice as wide as 32-bit registers and variables, so they can store larger values.
- 64-bit registers and variables can be used to perform arithmetic on larger values, such as integers that are larger than 32 bits.
- 64-bit registers and variables can be used to access more memory, such as memory that is located above the 4 GB address limit of 32-bit processors.

Overall, using 64-bit registers and variables can make your code more efficient and capable. However, it is important to be aware of the differences between 32-bit and 64-bit programming before you start using 64-bit registers and variables.

The letter "R" in RAX, RBX, etc. stands for "register". These are special memory locations inside the CPU that can be used to store data and temporary results. 64-bit processors have 16 general-purpose registers, which are named RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP, R8, R9, R10, R11, R12, R13, R14, and R15.

Each register has a specific purpose, but they can all be used to store data and perform arithmetic

operations. For example, the RAX register is typically used to store the result of a function call. The RBX register is often used to store the base address of a data structure. And the RSP register is used to store the stack pointer.

Registers are much faster than memory, so using them can significantly improve the performance of your code. However, it is important to use them correctly, as using them incorrectly can lead to errors.

Here are some examples of how to use registers in assembly language:

```
; Move the value 5 into the RAX register.
mov rax, 5

; Add the value 6 to RAX.
add rax, 6

; Store the value of RAX in the memory location pointed to by RSI.
mov [rsi], rax

; Call the function `my_function`.
call my_function

; The result of the function call is stored in RAX.
```

When we add an "R" to the beginning of a register name, we are converting it to its 64-bit version.

For example, the 32-bit register EAX is converted to the 64-bit register RAX.

32-bit (x86)	64-bit (x64)
EAX	RAX
EBX	RBX
ECX	RCX
EDX	RDX
ESI	RSI
EDI	RDI
ESP	RSP
EBP	RBP

64-bit registers are twice as wide as 32-bit registers, so they can store larger values. This means that we can use 64-bit registers to perform arithmetic on larger values, such as integers that are larger than 32 bits.

We can also use 64-bit registers to access more memory, such as memory that is located above the 4 GB address limit of 32-bit processors.

Overall, using 64-bit registers can make our code more efficient and capable. However, it is important to be aware of the differences between 32-bit and 64-bit programming before we start using 64-bit registers.

CONCEPTS LEARNED SO FAR:

The chapter summary you provided is a good overview of the key concepts covered in Chapter 3 of an assembly language textbook. Here is a brief summary of the most important points:

- **Constants and expressions:** A constant is a fixed value, such as a number or a string. An expression is a combination of constants, operators, and other expressions that evaluates to a single value.
- **Identifiers and directives:** An identifier is a name that is used to identify a variable, symbolic constant, procedure, or code label. A directive is a command that is embedded in the source code and interpreted by the assembler.
- **Instructions and operands:** An instruction is a source code statement that is executed by the processor at runtime. An operand is a value that is passed to an instruction.
- **Program segments:** Assembly language programs contain logical segments named code, data, and stack. The code segment contains executable instructions. The data segment holds variables. The stack segment holds procedure parameters, local variables, and return addresses.
- **Assemblers and linkers:** An assembler is a program that reads the source file, producing both object and listing files. The linker is a program that reads one or more object files and produces an executable file.
- **Data types:** Assembly language recognizes intrinsic data types, such as BYTE, WORD, DWORD, QWORD, REAL4, REAL8, and REAL10.
- **Data definitions:** A data definition statement sets aside storage in memory for a variable, and may optionally assign it a name.
- **String data definitions:** To create a string data definition, enclose a sequence of characters in quotes.
- **The DUP operator:** The DUP operator generates a repeated storage allocation, using a constant

expression as a counter.

- **Little-endian order:** x86 processors store and retrieve data from memory using little-endian order: The least significant byte of a variable is stored at its starting (lowest) address value.
- **Symbolic constants:** A symbolic constant (or symbol definition) associates an identifier with an integer or text expression.

3.8.1 Terms

assembler
big endian
binary coded decimal (BCD)
calling convention
character literal
code label
code segment
compiler
constant integer expression
data definition statement
data label
data segment
decimal real
directive
encoded real
executable file
floating-point literal
identifier
initializer
instruction
instruction mnemonic
integer constant
integer literal
intrinsic data type
label
linker
link library
listing file
little-endian order
macro
memory model
memory operand
object file
operand
operator precedence
packed binary coded decimal
process return code
program entry point
real number literal
reserved word
source file
stack segment
string literal
symbolic constant
system function

3.8.2 Instructions, Operators, and Directives

+	(add, unary plus)
=	(assign, compare for equality)
/	(divide)
*	(multiply)
( )	(parentheses)
–	(subtract, unary minus)
ADD	
BYTE	
CALL	
.CODE	
COMMENT	
.DATA	
DWORD	
END	
ENDP	
DUP	
EQU	
MOD	
MOV	
NOP	
PROC	
SBYTE	
SDWORD	
.STACK	
TEXTEQU	