

Assembly Language Programming Problems
Complete Guide with Line-by-Line Explanations
16-bit and 64-bit Implementation

August 20, 2025

Contents

1	Assembly Language Fundamentals for Beginners	4
1.1	Basic Assembly Instructions	4
1.1.1	Data Movement Instructions	4
1.1.2	Arithmetic Instructions	4
1.1.3	Logical Instructions	5
1.1.4	Control Flow Instructions	5
1.1.5	Function and Stack Instructions	5
1.2	Registers and Memory	6
1.2.1	16-bit Registers	6
1.2.2	64-bit Registers	6
1.3	System Calls and Interrupts	6
1.3.1	16-bit DOS Interrupts	6
1.3.2	64-bit Linux System Calls	6
1.4	Data Types and Memory Organization	7
1.4.1	Data Declaration Directives	7
1.4.2	ASCII and Number Conversion	7
2	Introduction	7
3	Problem 1: Swap Two Numbers	8
3.1	Problem Description	8
3.2	16-bit Implementation (Updated for Multi-Digit)	8
3.3	Line-by-Line Explanation (16-bit Updated)	11
3.4	64-bit Implementation	12
3.5	Line-by-Line Explanation (64-bit)	14
4	Problem 2: Find Greater Number Between Two Inputs	14
4.1	Problem Description	14
4.2	16-bit Implementation (Updated for Multi-Digit)	15
4.3	Line-by-Line Explanation (16-bit Updated)	17
4.4	64-bit Implementation	17
5	Problem 3: Prime Number Check	19
5.1	Problem Description	19
5.2	16-bit Implementation (Updated for Multi-Digit)	19
5.3	Line-by-Line Explanation (16-bit Updated)	22
6	Problem 4: Fibonacci Series Generation	22
6.1	Problem Description	22
6.2	16-bit Implementation (Updated for Multi-Digit)	22
6.3	Line-by-Line Explanation (16-bit Updated)	25
6.4	64-bit Implementation	26
6.5	Line-by-Line Explanation (64-bit)	29
7	Key Differences Between 16-bit and 64-bit Implementations	29
7.1	System Call Interface	29
7.2	Register Usage	29

7.3	Memory Management	29
7.4	Input/Output Handling	30
7.5	Multi-Digit Support Improvements	30
8	Programming Techniques and Concepts	30
8.1	Multi-Digit ASCII to Number Conversion	30
8.2	Number to ASCII Conversion	30
8.3	Conditional Branching	30
8.4	Loop Structures	31
8.5	Arithmetic Operations	31
9	Optimization Considerations	31
9.1	Register Usage Optimization	31
9.2	Algorithm Efficiency	31
9.3	Code Size Optimization	31
10	Debugging and Testing Strategies	32
10.1	Common Debugging Techniques	32
10.2	Error Prevention	32
10.3	Testing Multi-Digit Support	32
11	Conclusion	32

1 Assembly Language Fundamentals for Beginners

Before diving into the programming problems, it's essential to understand the fundamental instructions and concepts used in assembly language programming. This section provides a beginner-friendly tutorial on the most commonly used assembly instructions.

1.1 Basic Assembly Instructions

1.1.1 Data Movement Instructions

MOV - Move Data The MOV instruction copies data from source to destination.

```
1 mov ax, 5      ; Move immediate value 5 to AX register
2 mov bx, ax     ; Copy contents of AX to BX register
3 mov [num], ax  ; Move AX contents to memory location 'num'
4 mov ax, [num]  ; Load value from memory location 'num' to AX
```

LEA - Load Effective Address LEA loads the address of a memory location into a register.

```
1 lea dx, message ; Load address of 'message' into DX register
```

XCHG - Exchange XCHG swaps the contents of two operands.

```
1 xchg ax, bx    ; Exchange contents of AX and BX registers
```

1.1.2 Arithmetic Instructions

ADD - Addition ADD performs addition operation.

```
1 add ax, 5      ; Add 5 to AX register
2 add ax, bx     ; Add BX to AX, result stored in AX
```

SUB - Subtraction SUB performs subtraction operation.

```
1 sub ax, 3      ; Subtract 3 from AX
2 sub ax, bx     ; Subtract BX from AX
```

MUL - Multiplication MUL performs unsigned multiplication.

```
1 mul bx         ; Multiply AX by BX, result in DX:AX
```

IMUL - Signed Multiplication IMUL performs signed integer multiplication.

```
1 imul ax, 10    ; Multiply AX by 10
```

DIV - Division DIV performs unsigned division.

```
1 div bx         ; Divide DX:AX by BX, quotient in AX, remainder in DX
```

INC/DEC - Increment/Decrement INC adds 1 to operand, DEC subtracts 1 from operand.

```
1 inc ax          ; Increment AX by 1
2 dec bx          ; Decrement BX by 1
```

1.1.3 Logical Instructions

CMP - Compare CMP compares two operands by performing subtraction without storing result.

```
1 cmp ax, bx      ; Compare AX with BX, sets flags
2 cmp ax, 0        ; Compare AX with 0
```

XOR - Exclusive OR XOR performs bitwise exclusive OR operation.

```
1 xor ax, ax      ; Clear AX register (common idiom)
2 xor ax, bx      ; XOR AX with BX
```

1.1.4 Control Flow Instructions

JMP - Unconditional Jump JMP transfers control to specified label.

```
1 jmp label       ; Jump to 'label'
```

Conditional Jumps These instructions jump based on flag conditions set by previous instructions.

```
1 je label        ; Jump if Equal (ZF=1)
2 jne label       ; Jump if Not Equal (ZF=0)
3 jl label        ; Jump if Less ( S F OF )
4 jle label       ; Jump if Less or Equal (ZF=1 or S F OF )
5 jg label        ; Jump if Greater (ZF=0 and SF=OF)
6 jge label       ; Jump if Greater or Equal (SF=OF)
```

1.1.5 Function and Stack Instructions

CALL/RET - Function Call and Return CALL pushes return address and jumps to function. RET returns to caller.

```
1 call function   ; Call function, push return address
2 ret            ; Return to caller
```

PUSH/POP - Stack Operations PUSH stores data on stack, POP retrieves data from stack.

```
1 push ax        ; Push AX onto stack
2 pop bx         ; Pop top of stack into BX
```

1.2 Registers and Memory

1.2.1 16-bit Registers

- **AX, BX, CX, DX**: General purpose 16-bit registers
- **AL, AH, BL, BH, CL, CH, DL, DH**: 8-bit portions of general registers
- **SI, DI**: Source and Destination Index registers
- **SP, BP**: Stack Pointer and Base Pointer

1.2.2 64-bit Registers

- **RAX, RBX, RCX, RDX**: Extended 64-bit general purpose registers
- **RSI, RDI**: Extended Source and Destination Index
- **RSP, RBP**: Extended Stack and Base Pointer
- **R8-R15**: Additional 64-bit registers

1.3 System Calls and Interrupts

1.3.1 16-bit DOS Interrupts

DOS uses interrupt 21h for system services:

```
1 mov ah, 01h      ; Function: Read character
2 int 21h          ; Call DOS interrupt
3
4 mov ah, 02h      ; Function: Write character (DL contains char)
5 int 21h          ; Call DOS interrupt
6
7 mov ah, 09h      ; Function: Write string (DX points to string)
8 int 21h          ; Call DOS interrupt
9
10 mov ah, 4Ch      ; Function: Terminate program
11 int 21h          ; Call DOS interrupt
```

1.3.2 64-bit Linux System Calls

Linux uses syscall instruction with system call numbers in RAX:

```
1 mov rax, 0       ; sys_read
2 mov rdi, 0       ; stdin file descriptor
3 mov rsi, buffer  ; buffer address
4 mov rdx, size    ; buffer size
5 syscall          ; invoke system call
6
7 mov rax, 1       ; sys_write
8 mov rdi, 1       ; stdout file descriptor
9 mov rsi, message ; message address
10 mov rdx, length ; message length
11 syscall          ; invoke system call
12
```

```
13 mov rax, 60      ; sys_exit
14 xor rdi, rdi     ; exit status 0
15 syscall          ; invoke system call
```

1.4 Data Types and Memory Organization

1.4.1 Data Declaration Directives

```
1 ; 16-bit assembly
2 db ?           ; Declare byte (8-bit), uninitialized
3 db 'A'         ; Declare byte with initial value
4 dw ?           ; Declare word (16-bit), uninitialized
5 dw 1234        ; Declare word with initial value
6
7 ; 64-bit assembly
8 resb 10        ; Reserve 10 bytes
9 db "Hello", 0  ; Declare string with null terminator
```

1.4.2 ASCII and Number Conversion

Converting between ASCII characters and numeric values:

```
1 ; ASCII to number
2 sub al, '0'     ; Convert ASCII digit to numeric (subtract 48)
3
4 ; Number to ASCII
5 add al, '0'     ; Convert numeric digit to ASCII (add 48)
6
7 ; Multi-digit conversion requires multiplication by 10:
8 ; result = result * 10 + new_digit
```

2 Introduction

This document provides comprehensive assembly language solutions for four fundamental programming problems, implemented in both 16-bit (DOS) and 64-bit (Linux) environments. Each solution includes detailed line-by-line explanations to help understand the assembly programming concepts and system call interfaces.

All programs have been updated to support multi-digit input, making them more practical and useful for real-world applications.

The four problems covered are:

1. Swap Two Numbers
2. Find Greater Number Between Two Inputs
3. Prime Number Check
4. Fibonacci Series Generation

3 Problem 1: Swap Two Numbers

3.1 Problem Description

Write an assembly program to input two multi-digit numbers, display them before swapping, perform the swap operation, and display the numbers after swapping.

3.2 16-bit Implementation (Updated for Multi-Digit)

```

1  org 100h                      ; COM file format origin
2
3  ; Data section
4  .data
5  num1 dw ?                     ; First number storage (16-bit)
6  num2 dw ?                     ; Second number storage (16-bit)
7  temp dw ?                     ; Temporary variable for swapping
8  input_buffer db 10 dup(?)     ; Input buffer for multi-digit numbers
9  prompt1 db 'Enter first number: $'
10 prompt2 db 'Enter second number: $'
11 before_msg db 'Before swap: $'
12 after_msg db 'After swap: $'
13 newline db 13, 10, '$'
14
15 ; Code section
16 .code
17 start:
18     ; Display first prompt
19     mov ah, 09h
20     lea dx, prompt1
21     int 21h
22
23     ; Read first number
24     call read_number
25     mov num1, ax
26
27     ; Display second prompt
28     mov ah, 09h
29     lea dx, prompt2
30     int 21h
31
32     ; Read second number
33     call read_number
34     mov num2, ax
35
36     ; Display before swap message
37     mov ah, 09h
38     lea dx, before_msg
39     int 21h
40
41     ; Display first number
42     mov ax, num1
43     call print_number
44
45     ; Print space
46     mov dl, ' '
47     mov ah, 02h

```



```
48     int 21h
49
50     ; Display second number
51     mov ax, num2
52     call print_number
53
54     ; Print newline
55     mov ah, 09h
56     lea dx, newline
57     int 21h
58
59     ; Perform swap
60     mov ax, num1
61     mov temp, ax
62     mov ax, num2
63     mov num1, ax
64     mov ax, temp
65     mov num2, ax
66
67     ; Display after swap message
68     mov ah, 09h
69     lea dx, after_msg
70     int 21h
71
72     ; Display swapped numbers
73     mov ax, num1
74     call print_number
75
76     ; Print space
77     mov dl, ' '
78     mov ah, 02h
79     int 21h
80
81     mov ax, num2
82     call print_number
83
84     ; Print newline
85     mov ah, 09h
86     lea dx, newline
87     int 21h
88
89     ; Program termination
90     mov ah, 4Ch
91     int 21h
92
93 read_number:
94     ; Function to read multi-digit number
95     ; Output: AX = parsed number
96     push bx
97     push cx
98     push dx
99
100     mov cx, 0                ; Clear result
101
102 read_loop:
103     mov ah, 01h              ; Read character
104     int 21h
105
```

```

106     cmp al, 13                ; Check for Enter key
107     je read_done
108
109     cmp al, '0'                ; Check if digit
110     jl read_loop
111     cmp al, '9'
112     jg read_loop
113
114     sub al, '0'                ; Convert to digit
115     mov bl, al                 ; Store digit
116     mov ax, cx                 ; Get current result
117     mov dx, 10
118     mul dx                     ; Multiply by 10
119     add ax, bx                 ; Add new digit
120     mov cx, ax                 ; Store result
121     jmp read_loop
122
123 read_done:
124     mov ax, cx                 ; Return result in AX
125     pop dx
126     pop cx
127     pop bx
128     ret
129
130 print_number:
131     ; Function to print number in AX
132     push ax
133     push bx
134     push cx
135     push dx
136
137     mov cx, 0                  ; Digit counter
138     mov bx, 10                 ; Divisor
139
140     cmp ax, 0                  ; Check for zero
141     jne convert_digits
142
143     ; Handle zero case
144     mov dl, '0'
145     mov ah, 02h
146     int 21h
147     jmp print_done
148
149 convert_digits:
150     cmp ax, 0
151     je print_digits
152
153     mov dx, 0                  ; Clear remainder
154     div bx                     ; Divide by 10
155     push dx                    ; Push remainder (digit)
156     inc cx                     ; Increment digit count
157     jmp convert_digits
158
159 print_digits:
160     cmp cx, 0
161     je print_done
162
163     pop dx                     ; Get digit

```

```
164      add dl, '0'                ; Convert to ASCII
165      mov ah, 02h
166      int 21h
167      dec cx
168      jmp print_digits
169
170 print_done:
171     pop dx
172     pop cx
173     pop bx
174     pop ax
175     ret
176
177 end start
```

Listing 1: 16-bit Number Swapping Program with Multi-Digit Support

3.3 Line-by-Line Explanation (16-bit Updated)

1. **org 100h**: Sets the origin address to 100h, required for COM executable format in DOS.
2. **.data**: Begins the data segment where variables are declared.
3. **num1 dw ?**: Declares a word (16-bit) variable for the first number, supporting values up to 65,535.
4. **num2 dw ?**: Declares a word variable for the second number.
5. **temp dw ?**: Declares a temporary word variable for the swapping operation.
6. **prompt1 db 'Enter first number: \$'**: String constant for user prompt with DOS string terminator.
7. **mov ah, 09h**: Loads function code 09h into AH register (DOS string output function).
8. **lea dx, prompt1**: Loads effective address of prompt1 into DX register.
9. **int 21h**: Invokes DOS interrupt 21h to execute the function in AH.
10. **call read_number**: Calls the multi-digit input function.
11. **mov num1, ax**: Stores the parsed number from AX into num1 variable.
12. **read_number function**:
 - Initializes result accumulator (CX) to zero
 - Reads characters one by one using DOS function 01h
 - Validates each character to ensure it's a digit (0-9)
 - Converts ASCII to numeric by subtracting '0'
 - Accumulates result: $\text{result} = \text{result} \times 10 + \text{digit}$
 - Continues until Enter key (ASCII 13) is pressed

13. **print_number** function:

- Handles special case of zero
- Uses division by 10 to extract digits in reverse order
- Pushes digits onto stack to reverse their order
- Pops digits and converts to ASCII for display
- Uses DOS function 02h to display each character

14. **Swapping logic:** Uses temporary variable to exchange values between num1 and num2.

15. **mov ah, 4Ch:** Loads function code 4Ch (program termination) into AH.

16. **int 21h:** Calls DOS interrupt to terminate the program.

3.4 64-bit Implementation

```

1 section .bss
2     inbuf1 resb 64          ; Input buffer for first number
3     inbuf2 resb 64          ; Input buffer for second number
4
5 section .data
6     prompt1 db "Enter first number: ", 0
7     prompt2 db "Enter second number: ", 0
8     before_msg db "Before swap: ", 0
9     after_msg db "After swap: ", 0
10    newline db 10, 0
11
12 section .text
13     global _start
14
15 _start:
16     ; Print first prompt
17     mov rax, 1              ; sys_write system call
18     mov rdi, 1              ; stdout file descriptor
19     mov rsi, prompt1        ; message address
20     mov rdx, 20             ; message length
21     syscall                 ; invoke system call
22
23     ; Read first number
24     mov rax, 0              ; sys_read system call
25     mov rdi, 0              ; stdin file descriptor
26     mov rsi, inbuf1         ; buffer address
27     mov rdx, 64             ; buffer size
28     syscall                 ; invoke system call
29
30     ; Parse first number into r8
31     mov rsi, inbuf1         ; source buffer
32     call parse_number       ; convert ASCII to integer
33     mov r8, rax             ; store first number in r8
34
35     ; Print second prompt
36     mov rax, 1              ; sys_write system call
37     mov rdi, 1              ; stdout file descriptor

```

```

38     mov rsi, prompt2          ; message address
39     mov rdx, 21               ; message length
40     syscall                   ; invoke system call
41
42     ; Read second number
43     mov rax, 0                ; sys_read system call
44     mov rdi, 0                ; stdin file descriptor
45     mov rsi, inbuf2           ; buffer address
46     mov rdx, 64               ; buffer size
47     syscall                   ; invoke system call
48
49     ; Parse second number into r9
50     mov rsi, inbuf2           ; source buffer
51     call parse_number         ; convert ASCII to integer
52     mov r9, rax               ; store second number in r9
53
54     ; Display before swap
55     call print_before_swap
56
57     ; Perform swap using register exchange
58     xchg r8, r9               ; Exchange values in r8 and r9
59
60     ; Display after swap
61     call print_after_swap
62
63     ; Program termination
64     mov rax, 60               ; sys_exit system call
65     xor rdi, rdi              ; exit status 0
66     syscall                   ; invoke system call
67
68 parse_number:
69     ; Function to convert ASCII string to integer
70     ; Input: RSI = string address
71     ; Output: RAX = integer value
72     xor rax, rax              ; clear result
73     xor rbx, rbx              ; clear temporary register
74 parse_loop:
75     mov bl, [rsi]             ; load character
76     cmp bl, 10                ; check for newline
77     je parse_done             ; exit if newline
78     cmp bl, '0'               ; check if less than '0'
79     jl parse_done             ; exit if not digit
80     cmp bl, '9'               ; check if greater than '9'
81     jg parse_done             ; exit if not digit
82     sub bl, '0'               ; convert to digit
83     imul rax, 10              ; multiply result by 10
84     add rax, rbx               ; add current digit
85     inc rsi                   ; move to next character
86     jmp parse_loop            ; continue parsing
87 parse_done:
88     ret                       ; return with result in RAX
89
90 print_before_swap:
91     ; Implementation for printing numbers before swap
92     ret
93
94 print_after_swap:
95     ; Implementation for printing numbers after swap

```

Listing 2: 64-bit Number Swapping Program

3.5 Line-by-Line Explanation (64-bit)

1. **section .bss**: Declares uninitialized data section.
2. **inbuf1 resb 64**: Reserves 64 bytes for first number input buffer.
3. **section .data**: Declares initialized data section with strings.
4. **global _start**: Makes `_start` symbol globally visible to linker.
5. **_start::**: Program entry point label.
6. **mov rax, 1**: Loads `sys_write` system call number into RAX.
7. **mov rdi, 1**: Sets file descriptor to 1 (stdout).
8. **mov rsi, prompt1**: Points RSI to the prompt string address.
9. **mov rdx, 20**: Sets the number of bytes to write.
10. **syscall**: Invokes the system call using the Linux `syscall` interface.
11. **mov rax, 0**: Loads `sys_read` system call number.
12. **call parse_number**: Calls function to convert ASCII to integer.
13. **mov r8, rax**: Stores parsed first number in R8 register.
14. **xchg r8, r9**: Exchanges values between R8 and R9 registers (swap operation).
15. **parse_number function**: Converts ASCII string to integer using decimal accumulation.

4 Problem 2: Find Greater Number Between Two Inputs

4.1 Problem Description

Write an assembly program to input two multi-digit numbers and determine which one is greater, then display the result.

4.2 16-bit Implementation (Updated for Multi-Digit)

```

1  org 100h                ; COM file format origin
2
3  ; Data section
4  .data
5  num1 dw ?                ; Storage for first number
6  num2 dw ?                ; Storage for second number
7  prompt1 db 'Enter first number: $'
8  prompt2 db 'Enter second number: $'
9  result_msg db 'Greater number is: $'
10 newline db 13, 10, '$'
11
12 ; Code section
13 .code
14 start:
15     ; Display first prompt
16     mov ah, 09h
17     lea dx, prompt1
18     int 21h
19
20     ; Read first number
21     call read_number
22     mov num1, ax
23
24     ; Display second prompt
25     mov ah, 09h
26     lea dx, prompt2
27     int 21h
28
29     ; Read second number
30     call read_number
31     mov num2, ax
32
33     ; Compare numbers
34     mov ax, num1
35     mov bx, num2
36     cmp ax, bx
37     jge first_greater
38
39     ; Second number is greater
40     mov ax, bx
41
42 first_greater:
43     ; Display result message
44     mov ah, 09h
45     lea dx, result_msg
46     int 21h
47
48     ; Display the greater number
49     call print_number
50
51     ; Print newline
52     mov ah, 09h
53     lea dx, newline
54     int 21h
55
56     ; Program termination

```

```

57     mov ah, 4Ch
58     int 21h
59
60 read_number:
61     ; Function to read multi-digit number
62     ; Output: AX = parsed number
63     push bx
64     push cx
65     push dx
66
67     mov cx, 0                ; Clear result
68
69 read_loop:
70     mov ah, 01h              ; Read character
71     int 21h
72
73     cmp al, 13                ; Check for Enter key
74     je read_done
75
76     cmp al, '0'               ; Check if digit
77     jl read_loop
78     cmp al, '9'
79     jg read_loop
80
81     sub al, '0'               ; Convert to digit
82     mov bl, al                ; Store digit
83     mov ax, cx                ; Get current result
84     mov dx, 10
85     mul dx                    ; Multiply by 10
86     add ax, bx                ; Add new digit
87     mov cx, ax                ; Store result
88     jmp read_loop
89
90 read_done:
91     mov ax, cx                ; Return result in AX
92     pop dx
93     pop cx
94     pop bx
95     ret
96
97 print_number:
98     ; Function to print number in AX
99     push ax
100    push bx
101    push cx
102    push dx
103
104    mov cx, 0                  ; Digit counter
105    mov bx, 10                 ; Divisor
106
107    cmp ax, 0                  ; Check for zero
108    jne convert_digits
109
110    ; Handle zero case
111    mov dl, '0'
112    mov ah, 02h
113    int 21h
114    jmp print_done

```



```

115
116 convert_digits:
117     cmp ax, 0
118     je print_digits
119
120     mov dx, 0                ; Clear remainder
121     div bx                  ; Divide by 10
122     push dx                 ; Push remainder (digit)
123     inc cx                  ; Increment digit count
124     jmp convert_digits
125
126 print_digits:
127     cmp cx, 0
128     je print_done
129
130     pop dx                  ; Get digit
131     add dl, '0'             ; Convert to ASCII
132     mov ah, 02h
133     int 21h
134     dec cx
135     jmp print_digits
136
137 print_done:
138     pop dx
139     pop cx
140     pop bx
141     pop ax
142     ret
143
144 end start

```

Listing 3: 16-bit Greater Number Program with Multi-Digit Support

4.3 Line-by-Line Explanation (16-bit Updated)

1. **org 100h**: Sets origin for COM executable format.
2. **num1 dw ?, num2 dw ?**: Declares word variables for two numbers.
3. **call read_number**: Calls multi-digit input function.
4. **mov num1, ax**: Stores first parsed number.
5. **cmp ax, bx**: Performs comparison between the two numbers.
6. **jge first_greater**: Conditional jump if first number \geq second number.
7. **call print_number**: Displays the greater number using the print function.

4.4 64-bit Implementation

```

1 section .bss
2     inbuf1 resb 64          ; Input buffer for first number
3     inbuf2 resb 64          ; Input buffer for second number
4
5 section .data

```

```
6      prompt1 db "Enter first number: ", 0
7      prompt2 db "Enter second number: ", 0
8      result_msg db "Greater number is: ", 0
9
10     section .text
11         global _start
12
13     _start:
14         ; Display first prompt and read number
15         mov rax, 1
16         mov rdi, 1
17         mov rsi, prompt1
18         mov rdx, 20
19         syscall
20
21         mov rax, 0
22         mov rdi, 0
23         mov rsi, inbuf1
24         mov rdx, 64
25         syscall
26
27         ; Parse first number
28         mov rsi, inbuf1
29         call parse_number
30         mov r8, rax
31
32         ; Display second prompt and read number
33         mov rax, 1
34         mov rdi, 1
35         mov rsi, prompt2
36         mov rdx, 21
37         syscall
38
39         mov rax, 0
40         mov rdi, 0
41         mov rsi, inbuf2
42         mov rdx, 64
43         syscall
44
45         ; Parse second number
46         mov rsi, inbuf2
47         call parse_number
48         mov r9, rax
49
50         ; Compare and select maximum
51         cmp r8, r9
52         jge first_is_greater
53         mov rbx, r9
54         jmp display_result
55
56     first_is_greater:
57         mov rbx, r8
58
59     display_result:
60         ; Display result message
61         mov rax, 1
62         mov rdi, 1
63         mov rsi, result_msg
```

```

64     mov rdx, 19
65     syscall
66
67     ; Convert and display the greater number
68     mov rax, rbx
69     call print_number
70
71     ; Program termination
72     mov rax, 60
73     xor rdi, rdi
74     syscall
75
76 parse_number:
77     ; Convert ASCII string to integer
78     xor rax, rax
79     xor rbx, rbx
80 parse_loop:
81     mov bl, [rsi]
82     cmp bl, 10
83     je parse_done
84     cmp bl, '0'
85     jl parse_done
86     cmp bl, '9'
87     jg parse_done
88     sub bl, '0'
89     imul rax, 10
90     add rax, rbx
91     inc rsi
92     jmp parse_loop
93 parse_done:
94     ret
95
96 print_number:
97     ; Convert integer to ASCII and display
98     ; Implementation would convert RAX to ASCII string and print
99     ret

```

Listing 4: 64-bit Greater Number Program

5 Problem 3: Prime Number Check

5.1 Problem Description

Write an assembly program to check whether a given multi-digit number is prime or not. A prime number is divisible only by 1 and itself.

5.2 16-bit Implementation (Updated for Multi-Digit)

```

1  org 100h                ; COM file format origin
2
3  ; Data section
4  .data
5  num dw ?                ; Storage for input number
6  prompt db 'Enter a number:

```

```

7 prime_msg db 'Prime
8 not_prime_msg db 'Not Prime
9 newline db 13, 10, '
10
11 ; Code section
12 .code
13 start:
14     ; Display prompt
15     mov ah, 09h
16     lea dx, prompt
17     int 21h
18
19     ; Read number
20     call read_number
21     mov num, ax
22
23     ; Print newline
24     mov ah, 09h
25     lea dx, newline
26     int 21h
27
28     ; Check if number is less than 2
29     mov ax, num
30     cmp ax, 2
31     jl not_prime
32
33     ; Check if number equals 2
34     cmp ax, 2
35     je is_prime
36
37     ; Check if number is even (except 2)
38     mov dx, 0
39     mov bx, 2
40     div bx
41     cmp dx, 0
42     je not_prime
43
44     ; Initialize divisor for trial division
45     mov bx, 3                ; Start with divisor 3
46     mov ax, num              ; Reload number
47
48 loop_check:
49     ; Check if divisor * divisor > number
50     mov cx, bx
51     mov ax, bx
52     mul bx                    ; BX * BX
53     mov dx, ax
54     mov ax, num                ; Reload number
55     cmp dx, ax
56     jg is_prime                ; If divisor^2 > number, it's prime
57
58     ; Check if number is divisible by current divisor
59     mov dx, 0
60     div bx
61     cmp dx, 0
62     je not_prime                ; If remainder is 0, not prime
63
64     ; Increment divisor by 2 (check only odd numbers)

```

```

65     add bx, 2
66     mov ax, num                ; Reload number for next iteration
67     jmp loop_check
68
69 is_prime:
70     ; Display prime message
71     mov ah, 09h
72     lea dx, prime_msg
73     int 21h
74     jmp exit
75
76 not_prime:
77     ; Display not prime message
78     mov ah, 09h
79     lea dx, not_prime_msg
80     int 21h
81
82 exit:
83     ; Print newline
84     mov ah, 09h
85     lea dx, newline
86     int 21h
87
88     ; Program termination
89     mov ah, 4Ch
90     int 21h
91
92 read_number:
93     ; Function to read multi-digit number
94     ; Output: AX = parsed number
95     push bx
96     push cx
97     push dx
98
99     mov cx, 0                ; Clear result
100
101 read_loop:
102     mov ah, 01h              ; Read character
103     int 21h
104
105     cmp al, 13                ; Check for Enter key
106     je read_done
107
108     cmp al, '0'                ; Check if digit
109     jl read_loop
110     cmp al, '9'
111     jg read_loop
112
113     sub al, '0'                ; Convert to digit
114     mov bl, al                ; Store digit
115     mov ax, cx                ; Get current result
116     mov dx, 10
117     mul dx                    ; Multiply by 10
118     add ax, bx                ; Add new digit
119     mov cx, ax                ; Store result
120     jmp read_loop
121
122 read_done:

```

```

123     mov ax, cx                ; Return result in AX
124     pop dx
125     pop cx
126     pop bx
127     ret
128
129 end start

```

Listing 5: 16-bit Prime Check Program with Multi-Digit Support

5.3 Line-by-Line Explanation (16-bit Updated)

1. **org 100h**: Sets origin address for COM executable.
2. **num dw ?**: Declares word variable for input number (supports up to 65,535).
3. **call read_number**: Calls multi-digit input function.
4. **cmp ax, 2**: Compares input with 2 (smallest prime).
5. **jl not_prime**: Jumps to not_prime if input < 2.
6. **je is_prime**: Jumps to is_prime if input equals 2.
7. **Even number check**: Divides by 2 to check if even (composite if < 2).
8. **mov bx, 3**: Initializes divisor to 3 for odd number trial division.
9. **Square root optimization**: Checks if divisor² > number to limit search.
10. **add bx, 2**: Increments divisor by 2 (checks only odd divisors).
11. **Prime determination**: Uses trial division algorithm optimized for efficiency.

6 Problem 4: Fibonacci Series Generation

6.1 Problem Description

Write an assembly program to generate and display the first N numbers of the Fibonacci series, where each number is the sum of the two preceding ones.

6.2 16-bit Implementation (Updated for Multi-Digit)

```

1  org 100h                ; COM file format origin
2
3  ; Data section
4  .data
5  count dw ?              ; Storage for count of numbers
6  fib1 dw 0               ; First Fibonacci number
7  fib2 dw 1               ; Second Fibonacci number
8  prompt db 'Enter count:
9  space db '
10 newline db 13, 10, '

```

```
11
12 ; Code section
13 .code
14 start:
15     ; Display prompt
16     mov ah, 09h
17     lea dx, prompt
18     int 21h
19
20     ; Read count
21     call read_number
22     mov count, ax
23
24     ; Print newline
25     mov ah, 09h
26     lea dx, newline
27     int 21h
28
29     ; Initialize Fibonacci sequence
30     mov cx, count                ; Load count into CX
31     mov ax, 0                    ; F(0) = 0
32     mov bx, 1                    ; F(1) = 1
33
34     ; Check if count is 0
35     cmp cx, 0
36     je done
37
38     ; Print first number (0)
39     call print_number
40     mov ah, 09h
41     lea dx, space
42     int 21h
43
44     dec cx
45     cmp cx, 0
46     je done
47
48     ; Print second number (1)
49     mov ax, bx
50     call print_number
51     mov ah, 09h
52     lea dx, space
53     int 21h
54
55     dec cx
56
57 fibonacci_loop:
58     cmp cx, 0
59     je done
60
61     ; Calculate next Fibonacci number
62     mov dx, ax                    ; Store F(n-2)
63     add ax, bx                    ; F(n) = F(n-1) + F(n-2)
64     mov bx, dx                    ; Update F(n-1) = old F(n-2)
65     xchg ax, bx                    ; Swap for next iteration
66
67     ; Print the number
68     mov ax, bx
```

```
69      call print_number
70      mov ah, 09h
71      lea dx, space
72      int 21h
73
74      dec cx
75      jmp fibonacci_loop
76
77 done:
78     ; Print newline
79     mov ah, 09h
80     lea dx, newline
81     int 21h
82
83     ; Program termination
84     mov ah, 4Ch
85     int 21h
86
87 read_number:
88     ; Function to read multi-digit number
89     ; Output: AX = parsed number
90     push bx
91     push cx
92     push dx
93
94     mov cx, 0                ; Clear result
95
96 read_loop:
97     mov ah, 01h              ; Read character
98     int 21h
99
100    cmp al, 13                ; Check for Enter key
101    je read_done
102
103    cmp al, '0'               ; Check if digit
104    jl read_loop
105    cmp al, '9'
106    jg read_loop
107
108    sub al, '0'               ; Convert to digit
109    mov bl, al                ; Store digit
110    mov ax, cx                ; Get current result
111    mov dx, 10
112    mul dx                    ; Multiply by 10
113    add ax, bx                ; Add new digit
114    mov cx, ax                ; Store result
115    jmp read_loop
116
117 read_done:
118     mov ax, cx                ; Return result in AX
119     pop dx
120     pop cx
121     pop bx
122     ret
123
124 print_number:
125     ; Function to print number in AX
126     push ax
```



```

127     push bx
128     push cx
129     push dx
130
131     mov cx, 0                ; Digit counter
132     mov bx, 10              ; Divisor
133
134     cmp ax, 0                ; Check for zero
135     jne convert_digits
136
137     ; Handle zero case
138     mov dl, '0'
139     mov ah, 02h
140     int 21h
141     jmp print_done
142
143 convert_digits:
144     cmp ax, 0
145     je print_digits
146
147     mov dx, 0                ; Clear remainder
148     div bx                   ; Divide by 10
149     push dx                  ; Push remainder (digit)
150     inc cx                   ; Increment digit count
151     jmp convert_digits
152
153 print_digits:
154     cmp cx, 0
155     je print_done
156
157     pop dx                   ; Get digit
158     add dl, '0'              ; Convert to ASCII
159     mov ah, 02h
160     int 21h
161     dec cx
162     jmp print_digits
163
164 print_done:
165     pop dx
166     pop cx
167     pop bx
168     pop ax
169     ret
170
171 end start

```

Listing 6: 16-bit Fibonacci Series Program with Multi-Digit Support

6.3 Line-by-Line Explanation (16-bit Updated)

1. **count dw ?**: Declares word variable to store the count of Fibonacci numbers.
2. **call read_number**: Reads multi-digit count from user.
3. **mov cx, count**: Loads count into CX register for loop control.
4. **mov ax, 0; mov bx, 1**: Initializes first two Fibonacci numbers.

5. **Fibonacci calculation:** $F(n) = F(n-1) + F(n-2)$ using register arithmetic.
6. **xchg ax, bx:** Efficiently swaps values for next iteration.
7. **call print_number:** Displays each Fibonacci number using the print function.
8. **Loop control:** Uses CX as counter, decrements after each iteration.

6.4 64-bit Implementation

```

1 section .bss
2     inbuf resb 64           ; Input buffer for count
3     outbuf resb 20          ; Output buffer for numbers
4
5 section .data
6     prompt db "Enter N: ", 0
7     space db " ", 0
8     newline db 10, 0
9
10 section .text
11     global _start
12
13 _start:
14     ; Display prompt
15     mov rax, 1               ; sys_write system call
16     mov rdi, 1               ; stdout file descriptor
17     mov rsi, prompt          ; prompt message address
18     mov rdx, 9               ; message length
19     syscall                  ; invoke system call
20
21     ; Read count
22     mov rax, 0               ; sys_read system call
23     mov rdi, 0               ; stdin file descriptor
24     mov rsi, inbuf           ; input buffer address
25     mov rdx, 64              ; buffer size
26     syscall                  ; invoke system call
27
28     ; Parse count into RCX
29     mov rsi, inbuf           ; source buffer
30     call parse_number        ; convert to integer
31     mov rcx, rax             ; store count in rcx
32
33     ; Initialize Fibonacci sequence
34     mov r8, 0                ; F(0) = 0
35     mov r9, 1                ; F(1) = 1
36
37     ; Check if count is 0
38     cmp rcx, 0               ; compare count with 0
39     je exit_program          ; exit if no numbers to print
40
41     ; Print first number (0)
42     mov rax, r8              ; move first number to rax
43     call print_number         ; display the number
44     call print_space          ; print space
45
46     ; Check if count is 1
47     dec rcx                  ; decrement count

```

```

48     cmp rcx, 0                ; compare with 0
49     je exit_program          ; exit if only one number
50
51     ; Print second number (1)
52     mov rax, r9               ; move second number to rax
53     call print_number         ; display the number
54     call print_space          ; print space
55
56     ; Decrement count for remaining numbers
57     dec rcx                   ; decrement count
58
59 fibonacci_loop:
60     ; Check if more numbers needed
61     cmp rcx, 0                ; compare count with 0
62     je print_newline          ; exit loop if done
63
64     ; Calculate next Fibonacci number
65     mov rax, r8               ; load F(n-2)
66     add rax, r9               ; add F(n-1) to get F(n)
67     mov r8, r9                ; F(n-2) = old F(n-1)
68     mov r9, rax               ; F(n-1) = new F(n)
69
70     ; Display the new number
71     call print_number         ; display F(n)
72     call print_space          ; print space
73
74     ; Continue loop
75     dec rcx                   ; decrement counter
76     jmp fibonacci_loop        ; continue loop
77
78 print_newline:
79     ; Print newline character
80     mov rax, 1                ; sys_write system call
81     mov rdi, 1                ; stdout file descriptor
82     mov rsi, newline          ; newline character
83     mov rdx, 1                ; character length
84     syscall                   ; invoke system call
85
86 exit_program:
87     ; Program termination
88     mov rax, 60               ; sys_exit system call
89     xor rdi, rdi              ; exit status 0
90     syscall                   ; invoke system call
91
92 parse_number:
93     ; Convert ASCII string to integer
94     xor rax, rax              ; clear result accumulator
95     xor rbx, rbx              ; clear temporary register
96 parse_digit_loop:
97     mov bl, [rsi]             ; load current character
98     cmp bl, 10                ; check for newline
99     je parse_complete         ; exit if newline found
100    cmp bl, '0'                ; validate lower bound
101    jl parse_complete          ; exit if not a digit
102    cmp bl, '9'                ; validate upper bound
103    jg parse_complete          ; exit if not a digit
104    sub bl, '0'                ; convert ASCII to digit
105    imul rax, 10               ; multiply result by 10

```

```

106     add rax, rbx                ; add current digit to result
107     inc rsi                    ; move to next character
108     jmp parse_digit_loop       ; continue parsing
109 parse_complete:
110     ret                        ; return with result in RAX
111
112 print_number:
113     ; Convert integer to ASCII and display
114     push rax
115     push rbx
116     push rcx
117     push rdx
118
119     ; Handle special case of 0
120     cmp rax, 0
121     je print_zero
122
123     ; Convert number to string (reverse order)
124     mov rbx, 10                ; divisor for base 10
125     mov rcx, 0                ; digit counter
126     mov rsi, outbuf           ; output buffer
127     add rsi, 19               ; point to end of buffer
128     mov byte [rsi], 0         ; null terminator
129
130 convert_loop:
131     dec rsi                    ; move backward in buffer
132     xor rdx, rdx               ; clear remainder
133     div rbx                    ; divide by 10
134     add dl, '0'                ; convert remainder to ASCII
135     mov [rsi], dl              ; store digit
136     inc rcx                    ; increment digit count
137     cmp rax, 0                ; check if more digits
138     jne convert_loop          ; continue if more digits
139
140     ; Print the converted string
141     mov rax, 1                 ; sys_write system call
142     mov rdi, 1                 ; stdout file descriptor
143     mov rdx, rcx               ; number of digits
144     syscall                    ; invoke system call
145     jmp print_number_done      ; skip zero handling
146
147 print_zero:
148     ; Print single zero character
149     mov rax, 1                 ; sys_write system call
150     mov rdi, 1                 ; stdout file descriptor
151     mov rsi, zero_char         ; zero character
152     mov rdx, 1                 ; single character
153     syscall                    ; invoke system call
154
155 print_number_done:
156     pop rdx
157     pop rcx
158     pop rbx
159     pop rax
160     ret
161
162 print_space:
163     ; Print space character for formatting

```

```
164     mov rax, 1           ; sys_write system call
165     mov rdi, 1           ; stdout file descriptor
166     mov rsi, space       ; space character
167     mov rdx, 1           ; single character
168     syscall              ; invoke system call
169     ret
170
171 section .data
172 zero_char db '0'        ; character for printing zero
```

Listing 7: 64-bit Fibonacci Series Program

6.5 Line-by-Line Explanation (64-bit)

1. **section .bss**: Declares uninitialized data section.
2. **inbuf resb 64**: Reserves 64 bytes for input buffer.
3. **outbuf resb 20**: Reserves 20 bytes for number-to-string conversion.
4. **global _start**: Makes entry point visible to linker.
5. **syscall**: Invokes Linux system call interface.
6. **parse_number**: Converts ASCII string to integer using decimal accumulation.
7. **mov r8, 0; mov r9, 1**: Initializes Fibonacci sequence in 64-bit registers.
8. **fibonacci_loop**: Main loop for generating series using 64-bit arithmetic.
9. **print_number**: Converts integer to ASCII and displays using Linux system calls.

7 Key Differences Between 16-bit and 64-bit Implementations

7.1 System Call Interface

- **16-bit**: Uses DOS interrupts (INT 21h) with function codes in AH register
- **64-bit**: Uses Linux syscall interface with system call numbers in RAX

7.2 Register Usage

- **16-bit**: Limited to 8-bit (AL, BL, CL, DL) and 16-bit (AX, BX, CX, DX) registers
- **64-bit**: Extended 64-bit registers (RAX, RBX, RCX, RDX, R8-R15) available

7.3 Memory Management

- **16-bit**: Segmented memory model with .data and .code sections
- **64-bit**: Flat memory model with .bss, .data, and .text sections

7.4 Input/Output Handling

- **16-bit:** Character-by-character I/O with manual multi-digit parsing
- **64-bit:** Buffer-based I/O requiring string parsing and conversion

7.5 Multi-Digit Support Improvements

- **Enhanced Input Functions:** Both implementations now include robust multi-digit input parsing
- **Number Display Functions:** Complete integer-to-ASCII conversion and display
- **Larger Data Types:** Use of word (16-bit) and quadword (64-bit) for storing larger numbers
- **Validation:** Input validation ensures only valid numeric input is accepted

8 Programming Techniques and Concepts

8.1 Multi-Digit ASCII to Number Conversion

The updated implementations demonstrate advanced conversion techniques:

- **Accumulation Method:** $\text{result} = \text{result} \times 10 + \text{digit}$
- **Input Validation:** Check each character is within '0' to '9' range
- **Loop Termination:** Use Enter key (ASCII 13) or newline (ASCII 10) as delimiter

8.2 Number to ASCII Conversion

- **Division Method:** Repeatedly divide by 10 to extract digits
- **Stack Usage:** Use stack to reverse digit order for correct display
- **Zero Handling:** Special case handling for zero value

8.3 Conditional Branching

Enhanced conditional logic:

- **JE/JZ:** Jump if equal/zero
- **JGE:** Jump if greater than or equal
- **JL:** Jump if less than
- **JLE:** Jump if less than or equal

8.4 Loop Structures

Advanced looping mechanisms:

- **Counter-controlled loops:** Using CX/RCX for iteration count
- **Conditional loops:** Testing conditions for loop termination
- **Input validation loops:** Continue reading until valid input

8.5 Arithmetic Operations

Comprehensive arithmetic support:

- **Addition:** ADD instruction for sum calculations
- **Multiplication:** MUL/IMUL for multi-digit number parsing
- **Division:** DIV instruction for digit extraction and modular arithmetic
- **Increment/Decrement:** INC/DEC for counter manipulation

9 Optimization Considerations

9.1 Register Usage Optimization

- Minimize memory access by keeping frequently used values in registers
- Use appropriate register sizes (8-bit, 16-bit, 32-bit, 64-bit) based on data range
- Leverage register exchange (XCHG) for efficient swapping operations
- Preserve registers using PUSH/POP in functions

9.2 Algorithm Efficiency

- **Prime Check:** Optimized by checking divisors only up to \sqrt{n}
- **Fibonacci:** Iterative approach is more memory-efficient than recursive
- **String Operations:** Buffer-based I/O reduces system call overhead
- **Input Parsing:** Single-pass parsing with validation

9.3 Code Size Optimization

- Use short jumps when possible to reduce instruction size
- Combine operations where feasible (e.g., XOR for clearing registers)
- Reuse code segments through function calls and labels
- Efficient use of stack for temporary storage

10 Debugging and Testing Strategies

10.1 Common Debugging Techniques

- **Step-by-step execution:** Use debugger to trace instruction execution
- **Register monitoring:** Watch register values during program execution
- **Memory inspection:** Verify data storage and retrieval operations
- **Boundary testing:** Test with edge cases (0, 1, maximum values)

10.2 Error Prevention

- **Input validation:** Check for valid numeric input ranges and reject invalid characters
- **Overflow handling:** Consider arithmetic overflow in calculations, especially with large numbers
- **Division by zero:** Ensure divisors are non-zero before division operations
- **Buffer bounds:** Prevent buffer overruns in string operations

10.3 Testing Multi-Digit Support

- **Single digit:** Verify backward compatibility with original functionality
- **Multi-digit:** Test with various number lengths (2-digit, 3-digit, etc.)
- **Maximum values:** Test with largest supported values (65535 for 16-bit)
- **Edge cases:** Test with 0, 1, and boundary conditions
- **Invalid input:** Verify rejection of non-numeric characters

11 Conclusion

This comprehensive guide demonstrates fundamental assembly language programming concepts through practical implementations with enhanced multi-digit support. The updated line-by-line explanations provide insights into:

- **Enhanced I/O Operations:** Robust multi-digit input and output functions
- **System call interfaces:** For different architectures (16-bit DOS, 64-bit Linux)
- **Register management:** Efficient use of available registers and memory organization
- **Control flow:** Advanced conditional execution and loop structures
- **Arithmetic operations:** Complex data manipulation and number processing

- **String processing:** ASCII-to-number and number-to-ASCII conversion techniques
- **Function design:** Modular programming with reusable functions

The comparison between 16-bit and 64-bit implementations highlights the evolution of computer architectures and programming paradigms. The multi-digit support makes these programs practical for real-world use, handling numbers from 0 to 65535 in 16-bit implementations and much larger ranges in 64-bit versions.

These enhanced examples serve as building blocks for more complex assembly language programming projects and provide a solid foundation for understanding low-level system programming concepts. The multi-digit functionality demonstrates advanced programming techniques while maintaining clarity and educational value.

Understanding assembly language programming with multi-digit support enhances appreciation for:

- **Computer architecture:** And instruction set design principles
- **Compiler optimization:** Techniques and low-level code generation
- **Operating system interfaces:** And system programming methodologies
- **Performance-critical applications:** Development and optimization strategies
- **Embedded systems:** And microcontroller programming approaches
- **Algorithm implementation:** At the lowest level with maximum efficiency

The skills developed through these exercises form the foundation for advanced topics in systems programming, compiler design, operating system development, and performance optimization.