🔧 Complete x86 Assembly Programming Guide

Table of Contents

1. Introduction to x86 Assembly
2. General Purpose Registers
3. Special Registers
4. Memory Addressing
5. Data Types and Sizes
6. Basic Instructions
7. Arithmetic Instructions
8. Logical Instructions
9. Control Flow Instructions
10. Stack Operations
11. String Instructions
12. System Calls
13. Advanced Topics
14. Programming Examples
15. Tips and Tricks
16. Common Mistakes
17. Reference Tables

🚀 Introduction to x86 Assembly

x86 Assembly is a low-level programming language that provides direct control over the processor. It's the human-readable form of machine code for Intel x86 processors.

Key Concepts:

- **Instructions**: Commands that tell the CPU what to do
- **Registers**: High-speed storage locations in the CPU
- **Memory**: RAM storage accessed through addresses
- **Flags**: Status indicators that reflect the result of operations

Assembly Process:

```
# Assemble source code to object file
nasm -f elf32 program.asm -o program.o

# Link object file to create executable
ld -m elf_i386 program.o -o program

# Run the program
./program
```

📊 General Purpose Registers

x86 has **8 general-purpose registers**, each 32-bit wide in 32-bit mode:

Primary Registers:

Register	32-bit	16-bit	8-bit High	8-bit Low	Primary Use
EAX	EAX	AX	AH	AL	Accumulator (arithmetic)
EBX	EBX	BX	BH	BL	Base (memory addressing)
ECX	ECX	CX	CH	CL	Counter (loops, shifts)
EDX	EDX	DX	DH	DL	Data (I/O, multiplication)

Index and Pointer Registers:

Register	32-bit	16-bit	Description
ESI	ESI	SI	Source Index (string operations)
EDI	EDI	DI	Destination Index (string operations)
ESP	ESP	SP	Stack Pointer
EBP	EBP	BP	Base Pointer (stack frame)

Register Usage Examples:

```
mov eax, 42         ; Load 42 into EAX
mov ax, 1000        ; Load 1000 into AX (lower 16 bits of EAX)
mov al, 255         ; Load 255 into AL (lower 8 bits of EAX)
mov ah, 128         ; Load 128 into AH (upper 8 bits of AX)
```

⚡ Special Registers

Segment Registers (16-bit):

- **CS**: Code Segment - Points to code segment
- **DS**: Data Segment - Points to data segment
- **ES**: Extra Segment - Additional data segment

- **FS**: Additional segment register
- **GS**: Additional segment register
- **SS**: Stack Segment - Points to stack segment

Control Registers:

- **EIP**: Instruction Pointer (Program Counter)
- **EFLAGS**: Status and control flags register

EFLAGS Register Bits:

Bit	Flag	Name	Description
0	CF	Carry Flag	Set if arithmetic carry/borrow
2	PF	Parity Flag	Set if even number of 1 bits
4	AF	Auxiliary Flag	BCD arithmetic carry
6	ZF	Zero Flag	Set if result is zero
7	SF	Sign Flag	Set if result is negative
8	TF	Trap Flag	Single-step debugging
9	IF	Interrupt Flag	Enable/disable interrupts
10	DF	Direction Flag	String operation direction
11	OF	Overflow Flag	Signed arithmetic overflow

🎯 Memory Addressing

Addressing Modes:

1. Immediate Addressing:

```
mov eax, 42        ; Load immediate value 42
add ebx, 10        ; Add immediate value 10
```

2. Register Addressing:

```
mov eax, ebx       ; Copy EBX to EAX
add ecx, edx       ; Add EDX to ECX
```

3. Direct Memory Addressing:

```
mov eax, [1000]    ; Load from memory address 1000
mov [2000], ebx    ; Store EBX to memory address 2000
```

4. Indirect Addressing:

```
mov eax, [ebx]     ; Load from address in EBX
mov [ecx], eax     ; Store EAX to address in ECX
```

5. Indexed Addressing:

```
mov eax, [ebx + 4]     ; Load from EBX + 4
mov [esi + 8], edx     ; Store EDX to ESI + 8
```

6. Scaled Index Addressing:

```
mov eax, [ebx + esi*2]     ; Load from EBX + (ESI * 2)
mov [ebp + ecx*4 + 8], eax ; Store to EBP + (ECX * 4) + 8
```

📏 Data Types and Sizes

Data Declaration Directives:

Directive	Size	Description	Example
DB	1 byte	Define Byte	`db 42, 'A', 0`
DW	2 bytes	Define Word	`dw 1000, 0x1234`
DD	4 bytes	Define Double Word	`dd 100000, 0x12345678`
DQ	8 bytes	Define Quad Word	`dq 0x123456789ABCDEF0`
DT	10 bytes	Define Ten Bytes	`dt 3.14159`

String Declarations:

```
section .data
    msg db 'Hello, World!', 0      ; Null-terminated string
    nums db 1, 2, 3, 4, 5          ; Array of bytes
    words dw 100, 200, 300         ; Array of words
```

Uninitialized Data:

```
section .bss
    buffer resb 64         ; Reserve 64 bytes
    number resd 1          ; Reserve 1 double word
    array resw 10          ; Reserve 10 words
```

🔧 Basic Instructions

Data Movement:

```
mov dest, src       ; Move data from source to destination
xchg op1, op2       ; Exchange two operands
lea dest, src       ; Load effective address
```

Examples:

```
mov eax, 42         ; Load immediate value
mov ebx, eax        ; Register to register
mov [1000], eax     ; Register to memory
mov eax, [ebx]      ; Memory to register
lea eax, [ebx + 4] ; Load address of EBX + 4
```

Data Conversion:

```
cbw                ; Convert byte (AL) to word (AX)
cwde               ; Convert word (AX) to double word (EAX)
cdq                ; Convert double word (EAX) to quad word (EDX:EAX)
movsx dest, src    ; Move with sign extension
movzx dest, src    ; Move with zero extension
```

➕ Arithmetic Instructions

Basic Arithmetic:

```
add dest, src      ; Addition: dest = dest + src
sub dest, src      ; Subtraction: dest = dest - src
mul src            ; Unsigned multiply: EDX:EAX = EAX * src
imul src           ; Signed multiply
div src            ; Unsigned divide: EAX = EDX:EAX / src, EDX = remainder
idiv src           ; Signed divide
```

Increment/Decrement:

```
inc dest           ; Increment by 1
dec dest           ; Decrement by 1
neg dest           ; Two's complement negation
```

Examples:

```
mov eax, 10
add eax, 5         ; EAX = 15
sub eax, 3         ; EAX = 12
inc eax            ; EAX = 13

mov eax, 6
mov ebx, 4
mul ebx            ; EAX = 24, EDX = 0

mov eax, 20
```

```
mov ebx, 3
div ebx              ; EAX = 6 (quotient), EDX = 2 (remainder)
```

Advanced Arithmetic:

```
adc dest, src        ; Add with carry
sbb dest, src        ; Subtract with borrow
imul dest, src, imm ; Signed multiply with immediate
```

🔍 Logical Instructions

Bitwise Operations:

```
and dest, src        ; Bitwise AND
or dest, src         ; Bitwise OR
xor dest, src        ; Bitwise XOR
not dest             ; Bitwise NOT (one's complement)
```

Bit Shifts:

```
shl dest, count      ; Shift left (logical)
shr dest, count      ; Shift right (logical)
sal dest, count      ; Shift arithmetic left
sar dest, count      ; Shift arithmetic right
```

Rotations:

```
rol dest, count      ; Rotate left
ror dest, count      ; Rotate right
rcl dest, count      ; Rotate left through carry
rcr dest, count      ; Rotate right through carry
```

Examples:

```
mov eax, 0b11110000
and eax, 0b00001111    ; EAX = 0b00000000
or eax, 0b10101010     ; EAX = 0b10101010
xor eax, 0b11111111    ; EAX = 0b01010101

mov eax, 8
shl eax, 2             ; EAX = 32 (multiply by 4)
shr eax, 1             ; EAX = 16 (divide by 2)
```

Bit Testing:

```
test op1, op2        ; Bitwise AND without storing result
bt dest, bit         ; Bit test
bts dest, bit        ; Bit test and set
btr dest, bit        ; Bit test and reset
btc dest, bit        ; Bit test and complement
```

🔄 Control Flow Instructions

Unconditional Jumps:

```
jmp label           ; Jump to label
jmp eax             ; Jump to address in EAX
call label          ; Call procedure
ret                 ; Return from procedure
```

Conditional Jumps:

```
cmp op1, op2        ; Compare operands (sets flags)
je label            ; Jump if equal (ZF = 1)
jne label           ; Jump if not equal (ZF = 0)
jl label            ; Jump if less (SF ≠ OF)
jle label           ; Jump if less or equal
jg label            ; Jump if greater
jge label           ; Jump if greater or equal
ja label            ; Jump if above (unsigned >)
jb label            ; Jump if below (unsigned <)
jc label            ; Jump if carry (CF = 1)
jnc label           ; Jump if no carry (CF = 0)
jz label            ; Jump if zero (ZF = 1)
jnz label           ; Jump if not zero (ZF = 0)
```

Loop Instructions:

```
loop label          ; Decrement ECX and jump if ECX ≠ 0
loope label         ; Loop while equal
loopne label        ; Loop while not equal
```

Example - Simple Loop:

```
mov ecx, 10         ; Loop counter
loop_start:
    ; Loop body here
    dec ecx
    jnz loop_start ; Continue if ECX ≠ 0

; Or using loop instruction:
mov ecx, 10
loop_start:
    ; Loop body here
    loop loop_start
```

📚 Stack Operations

Basic Stack Operations:

```
push src            ; Push source onto stack
pop dest            ; Pop from stack to destination
pushf               ; Push flags register
popf                ; Pop flags register
```

```
pusha               ; Push all general-purpose registers
popa                ; Pop all general-purpose registers
```

Stack Frame Operations:

```
enter size, level ; Create stack frame
leave             ; Destroy stack frame (mov esp, ebp; pop ebp)
```

Examples:

```
; Save registers
push eax
push ebx
push ecx

; ... do some work ...

; Restore registers (in reverse order)
pop ecx
pop ebx
pop eax

; Function prologue/epilogue
function_start:
    push ebp          ; Save old base pointer
    mov ebp, esp      ; Set up new base pointer

    ; Function body

    mov esp, ebp      ; Restore stack pointer
    pop ebp           ; Restore base pointer
    ret               ; Return
```

🔤 String Instructions

String Move Instructions:

```
movsb               ; Move string byte
movsw               ; Move string word
movsd               ; Move string double word
```

String Compare Instructions:

```
cmpsb               ; Compare string bytes
cmpsw               ; Compare string words
cmpsd               ; Compare string double words
```

String Scan Instructions:

```
scasb               ; Scan string byte
scasw               ; Scan string word
scasd               ; Scan string double word
```

String Store Instructions:

```
stosb               ; Store string byte
stosw               ; Store string word
stosd               ; Store string double word
```

String Load Instructions:

```
lodsb               ; Load string byte
lodsw               ; Load string word
lodsd               ; Load string double word
```

Direction Control:

```
cld                 ; Clear direction flag (forward)
std                 ; Set direction flag (backward)
```

Repeat Prefixes:

```
rep                 ; Repeat while ECX ≠ 0
repe/repz           ; Repeat while equal/zero
repne/repnz         ; Repeat while not equal/not zero
```

Example - String Copy:

```
section .data
    source db 'Hello, World!', 0

section .bss
    dest resb 20

section .text
    mov esi, source    ; Source pointer
    mov edi, dest      ; Destination pointer
    mov ecx, 13        ; Number of bytes
    cld                ; Forward direction
    rep movsb          ; Copy string
```

🖥️ System Calls

Linux System Call Interface:

- **EAX**: System call number
- **EBX, ECX, EDX, ESI, EDI, EBP**: Arguments
- **INT 0x80**: Invoke system call

Common Linux System Calls:

Number	Name	EBX	ECX	EDX	Description
1	sys_exit	exit_code	-	-	Exit program

Number	Name	EBX	ECX	EDX	Description
3	sys_read	fd	buffer	count	Read from file
4	sys_write	fd	buffer	count	Write to file
5	sys_open	filename	flags	mode	Open file
6	sys_close	fd	-	-	Close file

Examples:

Hello World Program:

```
section .data
    msg db 'Hello, World!', 10, 0  ; Message with newline
    msg_len equ $ - msg - 1         ; Length of message

section .text
    global _start

_start:
    ; Write system call
    mov eax, 4          ; sys_write
    mov ebx, 1          ; stdout
    mov ecx, msg        ; message
    mov edx, msg_len    ; message length
    int 0x80            ; call kernel

    ; Exit system call
    mov eax, 1          ; sys_exit
    mov ebx, 0          ; exit status
    int 0x80            ; call kernel
```

Read User Input:

```
section .bss
    buffer resb 64      ; Buffer for input

section .text
    ; Read from stdin
    mov eax, 3          ; sys_read
    mov ebx, 0          ; stdin
    mov ecx, buffer     ; buffer
    mov edx, 64         ; max bytes
    int 0x80            ; call kernel
```

🎓 Advanced Topics

Floating Point Instructions (x87 FPU):

```
fld src             ; Load floating point value
fst dest            ; Store floating point value
fstp dest           ; Store and pop
fadd                ; Add top two stack values
fsub                ; Subtract
fmul                ; Multiply
fdiv                ; Divide
fcos                ; Cosine
fsin                ; Sine
fsqrt               ; Square root
```

MMX Instructions:

```
movq mm0, [mem]     ; Move quad word to MMX register
paddb mm0, mm1      ; Packed add bytes
psubw mm0, mm1      ; Packed subtract words
pmullw mm0, mm1     ; Packed multiply low words
emms                ; Empty MMX state
```

SSE Instructions:

```
movss xmm0, [mem] ; Move scalar single precision
addss xmm0, xmm1  ; Add scalar single precision
mulss xmm0, xmm1  ; Multiply scalar single precision
```

Interrupt Handling:

```
cli                 ; Clear interrupt flag
sti                 ; Set interrupt flag
int num             ; Software interrupt
iret                ; Interrupt return
```

Processor Control:

```
nop                 ; No operation
hlt                 ; Halt processor
cpuid               ; CPU identification
rdtsc               ; Read time stamp counter
```

💡 Programming Examples

Example 1: Calculate Factorial

```
section .data
    number dd 5
    result dd 1

section .text
    global _start
```

```
_start:
    mov eax, [number]     ; Load number
    mov ebx, 1            ; Initialize result

factorial_loop:
    cmp eax, 0            ; Check if done
    je done               ; Jump if zero

    mul ebx               ; Multiply result by current number
    mov ebx, eax          ; Store result
    mov eax, [number]     ; Reload number
    dec eax               ; Decrement
    mov [number], eax     ; Store back
    jmp factorial_loop    ; Continue loop

done:
    mov [result], ebx     ; Store final result

    ; Exit
    mov eax, 1
    mov ebx, 0
    int 0x80
```

Example 2: String Length Function

```
string_length:
    push ebp
    mov ebp, esp
    push edi

    mov edi, [ebp + 8]    ; Get string address
    mov eax, 0            ; Initialize counter

count_loop:
    cmp byte [edi], 0     ; Check for null terminator
    je count_done         ; Jump if found
    inc edi               ; Next character
    inc eax               ; Increment counter
    jmp count_loop

count_done:
    pop edi
    mov esp, ebp
    pop ebp
    ret
```

Example 3: Array Sum

```
section .data
    array dd 1, 2, 3, 4, 5
    array_size equ 5
    sum dd 0
```

```
section .text
    global _start

_start:
    mov esi, array          ; Point to array
    mov ecx, array_size     ; Loop counter
    mov eax, 0              ; Initialize sum

sum_loop:
    add eax, [esi]          ; Add current element
    add esi, 4              ; Move to next element
    loop sum_loop           ; Continue until ECX = 0

    mov [sum], eax          ; Store result

    ; Exit
    mov eax, 1
    mov ebx, 0
    int 0x80
```

Example 4: Bubble Sort

```
section .data
    array dd 64, 34, 25, 12, 22, 11, 90
    array_size equ 7

section .text
    global _start

_start:
    mov ecx, array_size
    dec ecx                 ; Outer loop counter

outer_loop:
    push ecx                ; Save outer counter
    mov esi, array          ; Reset array pointer
    mov ecx, array_size
    dec ecx                 ; Inner loop counter

inner_loop:
    mov eax, [esi]          ; Load current element
    mov ebx, [esi + 4]      ; Load next element
    cmp eax, ebx            ; Compare
    jle no_swap             ; Jump if in order

    ; Swap elements
    mov [esi], ebx
    mov [esi + 4], eax

no_swap:
```

```
    add esi, 4              ; Move to next element
    loop inner_loop

    pop ecx                 ; Restore outer counter
    loop outer_loop

    ; Exit
    mov eax, 1
    mov ebx, 0
    int 0x80
```

🎯 Tips and Tricks

Performance Optimization:

1. **Use LEA for arithmetic**:

   ```
   ; Instead of:
   mov eax, ebx
   add eax, ecx
   add eax, 8

   ; Use:
   lea eax, [ebx + ecx + 8]
   ```

2. **Clear registers efficiently**:

   ```
   ; Instead of:
   mov eax, 0

   ; Use (faster and smaller):
   xor eax, eax
   ```

3. **Multiply/divide by powers of 2**:

   ```
   ; Instead of:
   mov eax, 16
   mul ebx

   ; Use:
   shl ebx, 4     ; Multiply by 16 (2^4)
   ```

4. **Test for zero**:

   ```
   ; Instead of:
   cmp eax, 0
   je zero_label

   ; Use:
   ```

```
test eax, eax
jz zero_label
```

Memory Management:

1. **Align data for better performance**:

```
section .data
    align 4
    my_data dd 12345
```

2. **Use appropriate data sizes**:

```
; Use smallest appropriate size
mov al, 1     ; For values 0-255
mov ax, 1000  ; For values 0-65535
```

Debugging Tips:

1. **Use meaningful labels**:

```
main_loop:
error_handler:
cleanup_and_exit:
```

2. **Add comments for complex operations**:

```
; Calculate array offset: base + (index * element_size)
lea eax, [ebx + ecx*4]
```

3. **Preserve registers in functions**:

```
my_function:
    push eax      ; Save registers
    push ebx

    ; Function code

    pop ebx       ; Restore in reverse order
    pop eax
    ret
```

Code Organization:

1. **Use procedures for repeated code**:

```
print_string:
    ; String printing code
    ret

main:
    call print_string
```

2. **Separate data and code sections**:

```
section .data
    ; Initialized data

section .bss
    ; Uninitialized data

section .text
    ; Code
```

⚠️ Common Mistakes

1. Register Size Mismatches:

```
; WRONG:
mov al, 256      ; AL can only hold 0-255

; CORRECT:
mov ax, 256      ; Use appropriate register size
```

2. Incorrect Memory Addressing:

```
; WRONG:
mov eax, 1000    ; Loads immediate value 1000

; CORRECT:
mov eax, [1000]  ; Loads from memory address 1000
```

3. Stack Imbalance:

```
; WRONG:
push eax
push ebx
pop eax          ; Stack becomes unbalanced!

; CORRECT:
push eax
push ebx
pop ebx
pop eax
```

4. Forgetting to Set Direction Flag:

```
; WRONG:
rep movsb        ; Direction undefined

; CORRECT:
cld              ; Set forward direction
rep movsb
```

5. Not Preserving Registers:

```
; WRONG function:
my_func:
```

```
    mov eax, 42
    ret          ; EAX changed!

; CORRECT function:
my_func:
    push eax     ; Save register
    mov eax, 42
    ; Do work with EAX
    pop eax      ; Restore register
    ret
```

6. Infinite Loops:

```
; WRONG:
loop_start:
    ; Code that doesn't change loop condition
    jmp loop_start   ; Infinite loop!

; CORRECT:
mov ecx, 10
loop_start:
    ; Loop body
    loop loop_start  ; ECX automatically decremented
```

📋 Reference Tables

ASCII Character Codes:

Char	Dec	Hex	Char	Dec	Hex
' '	32	20h	'0'	48	30h
'!'	33	21h	'1'	49	31h
'"'	34	22h	'9'	57	39h
'A'	65	41h	'a'	97	61h
'B'	66	42h	'b'	98	62h
'Z'	90	5Ah	'z'	122	7Ah

Number System Conversions:

Decimal	Binary	Hexadecimal	Octal
0	0000	0	0
1	0001	1	1
8	1000	8	10
15	1111	F	17
16	10000	10	20
255	11111111	FF	377

Powers of 2:

Power	Value	Hex	Use
2^8	256	100h	Byte overflow
2^16	65536	10000h	Word overflow
2^32	4294967296	100000000h	Dword overflow

Instruction Timing (Approximate):

Instruction	Cycles	Notes
mov reg, reg	1	Fastest
add reg, reg	1	Simple arithmetic
mul reg	10-20	Expensive
div reg	20-40	Most expensive
int 0x80	100+	System call overhead

Assembler Directives:

Directive	Purpose	Example
global	Export symbol	`global _start`
extern	Import symbol	`extern printf`
section	Define section	`section .text`
equ	Define constant	`BUFFER_SIZE equ 64`
align	Align data	`align 4`
times	Repeat data	`times 10 db 0`

🔗 Additional Resources

Useful NASM Options:

```
# Generate listing file
nasm -f elf32 -l program.lst program.asm

# Define symbols
nasm -f elf32 -DDEBUG=1 program.asm

# Include directories
nasm -f elf32 -I./includes/ program.asm
```

Debugging with GDB:

```
# Compile with debug info
nasm -f elf32 -g -F dwarf program.asm
ld -m elf_i386 program.o -o program

# Debug with GDB
gdb ./program
```

```
(gdb) break _start
(gdb) run
(gdb) stepi           # Step one instruction
(gdb) info registers # Show register values
```

Memory Layout (Linux):

```
High Memory
+-------------------+
|       Stack       | <- ESP
+-------------------+
|         ↓         |
|                   |
|         ↑         |
+-------------------+
|        Heap       |
+-------------------+
|       .bss        | (Uninitialized data)
+-------------------+
|       .data       | (Initialized data)
+-------------------+
|       .text       | (Code)
+-------------------+
Low Memory
```

🧮 Mathematical Operations & Algorithms

Extended Arithmetic Operations:

64-bit Addition:

```
; Add two 64-bit numbers (stored as two 32-bit parts)
add_64bit:
    ; Input: EDX:EAX = first number, ECX:EBX = second number
    ; Output: EDX:EAX = sum
    add eax, ebx      ; Add lower parts
    adc edx, ecx      ; Add upper parts with carry
    ret
```

Integer Square Root:

```
; Calculate integer square root using binary search
isqrt:
    push ebp
    mov ebp, esp
    push ebx
    push ecx
    push edx

    mov eax, [ebp + 8]    ; Get input number
    cmp eax, 0
```

```
    je sqrt_zero

    mov ebx, 1            ; Low bound
    mov ecx, eax          ; High bound

sqrt_loop:
    cmp ebx, ecx
    jg sqrt_done

    mov edx, ebx
    add edx, ecx
    shr edx, 1            ; Mid = (low + high) / 2

    push eax              ; Save original number
    mov eax, edx
    mul edx               ; EDX = mid * mid
    pop eax               ; Restore original

    cmp edx, eax
    je sqrt_found
    jl sqrt_too_small

    ; Mid^2 > target, search lower half
    mov ecx, edx
    dec ecx
    jmp sqrt_loop

sqrt_too_small:
    ; Mid^2 < target, search upper half
    mov ebx, edx
    inc ebx
    jmp sqrt_loop

sqrt_found:
    mov eax, edx
    jmp sqrt_exit

sqrt_done:
    mov eax, ebx
    dec eax
    jmp sqrt_exit

sqrt_zero:
    mov eax, 0

sqrt_exit:
    pop edx
    pop ecx
    pop ebx
    mov esp, ebp
```

```
    pop ebp
    ret
```

Binary Search Implementation:

```
; Binary search in sorted array
; Returns index or -1 if not found
binary_search:
    push ebp
    mov ebp, esp
    push ebx
    push ecx
    push edx
    push esi

    mov esi, [ebp + 8]    ; Array pointer
    mov eax, [ebp + 12]   ; Array size
    mov ebx, [ebp + 16]   ; Target value

    mov ecx, 0            ; Left index
    dec eax               ; Right index (size - 1)

search_loop:
    cmp ecx, eax
    jg not_found

    mov edx, ecx
    add edx, eax
    shr edx, 1            ; Mid = (left + right) / 2

    push eax              ; Save right index
    mov eax, [esi + edx*4] ; Get array[mid]
    cmp eax, ebx
    pop eax               ; Restore right index

    je found
    jl search_right

    ; Target < array[mid], search left half
    mov eax, edx
    dec eax
    jmp search_loop

search_right:
    ; Target > array[mid], search right half
    mov ecx, edx
    inc ecx
    jmp search_loop

found:
    mov eax, edx          ; Return index
```

```
    jmp search_exit

not_found:
    mov eax, -1           ; Return -1

search_exit:
    pop esi
    pop edx
    pop ecx
    pop ebx
    mov esp, ebp
    pop ebp
    ret
```

🔧 Advanced Programming Techniques

Function Calling Conventions:

CDECL Convention:

```
; Caller pushes arguments right to left
; Caller cleans up stack
; Return value in EAX

; Calling a function
push arg3
push arg2
push arg1
call my_function
add esp, 12       ; Clean up 3 arguments (3 * 4 bytes)

; Function implementation
my_function:
    push ebp
    mov ebp, esp

    ; Access arguments
    mov eax, [ebp + 8]    ; First argument
    mov ebx, [ebp + 12]   ; Second argument
    mov ecx, [ebp + 16]   ; Third argument

    ; Function body

    mov esp, ebp
    pop ebp
    ret                 ; Return value in EAX
```

STDCALL Convention:

```
; Function cleans up its own stack
my_function_stdcall:
    push ebp
    mov ebp, esp

    ; Function body

    mov esp, ebp
    pop ebp
    ret 12              ; Clean up 3 arguments (3 * 4 bytes)
```

Recursive Functions:

Fibonacci Sequence:

```
fibonacci:
    push ebp
    mov ebp, esp

    mov eax, [ebp + 8]    ; Get n
    cmp eax, 1
    jle fib_base_case

    ; Recursive case: fib(n-1) + fib(n-2)
    dec eax
    push eax              ; Push n-1
    call fibonacci
    add esp, 4            ; Clean up
    push eax              ; Save fib(n-1)

    mov eax, [ebp + 8]    ; Get n again
    sub eax, 2
    push eax              ; Push n-2
    call fibonacci
    add esp, 4            ; Clean up

    pop ebx               ; Get fib(n-1)
    add eax, ebx          ; fib(n-1) + fib(n-2)
    jmp fib_exit

fib_base_case:
    ; Return n for n <= 1
    mov eax, [ebp + 8]

fib_exit:
    mov esp, ebp
    pop ebp
    ret
```

Macro Definitions:

```
; Define macros for common operations
%macro PRINT_CHAR 1
    push eax
    push ebx
    push ecx
    push edx

    mov eax, 4        ; sys_write
    mov ebx, 1        ; stdout
    mov ecx, %1       ; character address
    mov edx, 1        ; length
    int 0x80

    pop edx
    pop ecx
    pop ebx
    pop eax
%endmacro

%macro EXIT 1
    mov eax, 1        ; sys_exit
    mov ebx, %1       ; exit code
    int 0x80
%endmacro

; Usage:
section .data
    newline db 10

section .text
    PRINT_CHAR newline
    EXIT 0
```

🎮 Input/Output Operations

Advanced Console I/O:

Number to String Conversion:

```
; Convert 32-bit integer to ASCII string
int_to_string:
    push ebp
    mov ebp, esp
    push ebx
    push ecx
    push edx
    push edi
```

```
    mov eax, [ebp + 8]     ; Number to convert
    mov edi, [ebp + 12]    ; Buffer pointer
    mov ecx, 0             ; Digit counter

    ; Handle zero case
    cmp eax, 0
    jne convert_loop
    mov byte [edi], '0'
    mov byte [edi + 1], 0
    jmp convert_done

convert_loop:
    cmp eax, 0
    je reverse_digits

    mov edx, 0
    mov ebx, 10
    div ebx                ; EAX = quotient, EDX = remainder

    add dl, '0'            ; Convert to ASCII
    push edx               ; Save digit
    inc ecx                ; Count digits
    jmp convert_loop

reverse_digits:
    mov ebx, 0             ; Buffer index

reverse_loop:
    cmp ecx, 0
    je add_null_term

    pop edx
    mov [edi + ebx], dl
    inc ebx
    dec ecx
    jmp reverse_loop

add_null_term:
    mov byte [edi + ebx], 0

convert_done:
    pop edi
    pop edx
    pop ecx
    pop ebx
    mov esp, ebp
    pop ebp
    ret
```

String to Number Conversion:

```
; Convert ASCII string to 32-bit integer
string_to_int:
    push ebp
    mov ebp, esp
    push ebx
    push ecx
    push edx
    push esi

    mov esi, [ebp + 8]    ; String pointer
    mov eax, 0            ; Result
    mov ebx, 0            ; Character
    mov ecx, 10           ; Base

convert_char_loop:
    mov bl, [esi]         ; Get character
    cmp bl, 0             ; Check for null terminator
    je string_convert_done

    cmp bl, '0'           ; Check if valid digit
    jb string_convert_done
    cmp bl, '9'
    ja string_convert_done

    sub bl, '0'           ; Convert to number
    mul ecx               ; Multiply result by 10
    add eax, ebx          ; Add new digit

    inc esi               ; Next character
    jmp convert_char_loop

string_convert_done:
    pop esi
    pop edx
    pop ecx
    pop ebx
    mov esp, ebp
    pop ebp
    ret
```

File Operations:

Read File into Buffer:

```
read_file:
    push ebp
    mov ebp, esp
    push ebx
    push ecx
    push edx
```

```
    ; Open file
    mov eax, 5              ; sys_open
    mov ebx, [ebp + 8]      ; filename
    mov ecx, 0              ; O_RDONLY
    mov edx, 0644           ; permissions
    int 0x80

    cmp eax, 0
    jl file_error

    mov ebx, eax            ; Save file descriptor

    ; Read file
    mov eax, 3              ; sys_read
    mov ecx, [ebp + 12]     ; buffer
    mov edx, [ebp + 16]     ; buffer size
    int 0x80

    push eax                ; Save bytes read

    ; Close file
    mov eax, 6              ; sys_close
    int 0x80

    pop eax                 ; Restore bytes read
    jmp file_done

file_error:
    mov eax, -1

file_done:
    pop edx
    pop ecx
    pop ebx
    mov esp, ebp
    pop ebp
    ret
```

🔍 Debugging and Profiling

Assembly Debugging Techniques:

Debug Print Macro:

```
%macro DEBUG_PRINT 2    ; Message and register
    pusha

    ; Print debug message
```

```
    mov eax, 4
    mov ebx, 1
    mov ecx, %1
    mov edx, %2
    int 0x80

    popa
%endmacro

section .data
    debug_msg db 'Debug: EAX = ', 0
    debug_msg_len equ $ - debug_msg

section .text
    ; Usage:
    mov eax, 12345
    DEBUG_PRINT debug_msg, debug_msg_len
```

Performance Timing:

```
; Measure execution time using RDTSC
measure_performance:
    push ebp
    mov ebp, esp
    push ebx
    push ecx
    push edx

    ; Get start time
    rdtsc                   ; Read time stamp counter
    mov ebx, eax            ; Save low 32 bits
    mov ecx, edx            ; Save high 32 bits

    ; Execute code to measure
    call [ebp + 8]          ; Function pointer

    ; Get end time
    rdtsc
    sub eax, ebx            ; Calculate difference (low)
    sbb edx, ecx            ; Calculate difference (high)

    ; Result in EDX:EAX
    pop edx
    pop ecx
    pop ebx
    mov esp, ebp
    pop ebp
    ret
```

🌐 Interfacing with C

Calling C Functions from Assembly:

```
section .data
    format db 'Number: %d', 10, 0
    number dd 42

section .text
    extern printf
    global main

main:
    push ebp
    mov ebp, esp

    ; Call printf
    push dword [number]    ; Second argument
    push format            ; First argument
    call printf
    add esp, 8             ; Clean up stack

    mov eax, 0             ; Return 0
    mov esp, ebp
    pop ebp
    ret
```

Assembly Function Called from C:

```
; assembly_func.asm
section .text
    global add_numbers

add_numbers:
    push ebp
    mov ebp, esp

    mov eax, [ebp + 8]     ; First parameter
    add eax, [ebp + 12]    ; Second parameter

    mov esp, ebp
    pop ebp
    ret
```

```c
// main.c
extern int add_numbers(int a, int b);

int main() {
    int result = add_numbers(5, 3);
    printf("Result: %d\n", result);
    return 0;
}
```

Compile with:

```
nasm -f elf32 assembly_func.asm
gcc -m32 main.c assembly_func.o -o program
```

🔐 Security Considerations

Buffer Overflow Prevention:

```
; Safe string copy with bounds checking
safe_strcpy:
    push ebp
    mov ebp, esp
    push esi
    push edi
    push ecx

    mov edi, [ebp + 8]     ; Destination
    mov esi, [ebp + 12]    ; Source
    mov ecx, [ebp + 16]    ; Max length

    dec ecx                ; Reserve space for null terminator

copy_loop:
    cmp ecx, 0
    je copy_done

    mov al, [esi]
    cmp al, 0
    je copy_done

    mov [edi], al
    inc esi
    inc edi
    dec ecx
    jmp copy_loop

copy_done:
    mov byte [edi], 0      ; Null terminate

    pop ecx
    pop edi
    pop esi
    mov esp, ebp
    pop ebp
    ret
```

Stack Canary Implementation:

```
section .data
    canary dd 0xDEADBEEF

section .text
secure_function:
    push ebp
    mov ebp, esp

    ; Place canary on stack
    push dword [canary]
    sub esp, 100            ; Local buffer

    ; Function body here

    ; Check canary before return
    add esp, 100
    pop eax
    cmp eax, [canary]
    jne stack_overflow_detected

    mov esp, ebp
    pop ebp
    ret

stack_overflow_detected:
    ; Handle stack overflow
    mov eax, 1
    mov ebx, 1
    int 0x80                ; Exit with error
```

🧪 Optimization Techniques

Loop Unrolling:

```
; Original loop
mov ecx, 1000
loop_original:
    add eax, [esi]
    add esi, 4
    loop loop_original

; Unrolled loop (4x)
mov ecx, 250                ; 1000 / 4
loop_unrolled:
    add eax, [esi]
    add eax, [esi + 4]
    add eax, [esi + 8]
    add eax, [esi + 12]
```

```
    add esi, 16
    loop loop_unrolled
```

Branch Prediction Optimization:

```
; Arrange code so common case falls through
check_condition:
    test eax, eax
    jnz rare_case          ; Rare case jumps

    ; Common case code here (no jump needed)
    jmp done

rare_case:
    ; Rare case code here

done:
    ret
```

Cache-Friendly Memory Access:

```
; Access memory sequentially when possible
process_array:
    mov esi, array_ptr
    mov ecx, array_size

sequential_loop:
    mov eax, [esi]         ; Sequential access
    ; Process element
    add esi, 4             ; Next element
    loop sequential_loop
```

📊 Performance Benchmarks

Instruction Performance Comparison:

```
section .data
    iterations dd 1000000

section .text
; Benchmark different multiplication methods
benchmark_multiply:
    ; Method 1: MUL instruction
    mov ecx, [iterations]
    rdtsc
    mov ebx, eax

mul_loop:
    mov eax, 123
    mov edx, 456
    mul edx
```

```
    loop mul_loop

    rdtsc
    sub eax, ebx
    ; Store result for MUL method

    ; Method 2: Shift and add
    mov ecx, [iterations]
    rdtsc
    mov ebx, eax

shift_loop:
    mov eax, 123
    shl eax, 3            ; Multiply by 8
    shl eax, 1            ; Multiply by 2 more (total 16)
    ; Adjust for actual multiplier if needed
    loop shift_loop

    rdtsc
    sub eax, ebx
    ; Store result for shift method

    ret
```

🔬 Real-World Applications

Simple Bootloader:

```
; Simple bootloader (boot.asm)
BITS 16
ORG 0x7C00

start:
    mov ax, 0x07C0
    mov ds, ax

    mov si, msg
    call print_string

    cli
    hlt

print_string:
    lodsb
    cmp al, 0
    je done
    mov ah, 0x0E
    int 0x10
    jmp print_string
```

```
done:
    ret

msg db 'Hello from bootloader!', 13, 10, 0

times 510-($-$) db 0
db 0x55, 0xAA           ; Boot signature
```

Embedded System Timer:

```
; Timer interrupt handler for embedded systems
timer_interrupt:
    pusha

    inc dword [tick_count]

    ; Check if 1 second has passed (assuming 1000 Hz timer)
    cmp dword [tick_count], 1000
    jl timer_done

    mov dword [tick_count], 0
    inc dword [seconds]

    ; Update display or perform periodic tasks
    call update_display

timer_done:
    ; Send EOI to interrupt controller
    mov al, 0x20
    out 0x20, al

    popa
    iret

section .data
    tick_count dd 0
    seconds dd 0
```

📚 Extended Instruction Set

Bit Manipulation Instructions:

```
; Bit scan forward/reverse
bsf eax, ebx          ; Find first set bit
bsr eax, ebx          ; Find last set bit

; Bit test and modify
bt [mem], 5           ; Test bit 5
bts [mem], 5          ; Test and set bit 5
```

```
btr [mem], 5           ; Test and reset bit 5
btc [mem], 5           ; Test and complement bit 5

; Population count (newer processors)
popcnt eax, ebx        ; Count set bits in EBX
```

String Processing Instructions:

```
; String compare with length
repe cmpsb             ; Compare while equal
repne scasb            ; Scan while not equal

; Block memory operations
rep movsd              ; Copy blocks of double words
rep stosd              ; Fill blocks with double words
```

Advanced Arithmetic:

```
; Multiply with immediate
imul eax, ebx, 10      ; EAX = EBX * 10
imul eax, [mem], 5     ; EAX = [mem] * 5

; Conditional moves (Pentium Pro+)
cmove eax, ebx         ; Move if equal
cmovg eax, ebx         ; Move if greater
cmovl eax, ebx         ; Move if less
```

🎯 Assembly Programming Exercises

Exercise 1: Prime Number Checker

```
; Check if a number is prime
is_prime:
    push ebp
    mov ebp, esp
    push ebx
    push ecx

    mov eax, [ebp + 8]     ; Get number
    cmp eax, 2
    jl not_prime
    je prime_found

    ; Check if even
    test eax, 1
    jz not_prime

    ; Check odd divisors up to sqrt(n)
    mov ebx, 3

check_divisor:
```

```
    mov ecx, ebx
    mul ecx
    cmp eax, [ebp + 8]
    jg prime_found

    mov eax, [ebp + 8]
    xor edx, edx
    div ebx
    cmp edx, 0
    je not_prime

    add ebx, 2
    jmp check_divisor

prime_found:
    mov eax, 1
    jmp prime_done

not_prime:
    mov eax, 0

prime_done:
    pop ecx
    pop ebx
    mov esp, ebp
    pop ebp
    ret
```

Exercise 2: Matrix Multiplication

```
; Multiply two 3x3 matrices
matrix_multiply:
    push ebp
    mov ebp, esp
    push esi
    push edi
    push ebx
    push ecx
    push edx

    mov esi, [ebp + 8]     ; Matrix A
    mov edi, [ebp + 12]    ; Matrix B
    mov ebx, [ebp + 16]    ; Result matrix

    mov ecx, 0             ; Row counter

row_loop:
    cmp ecx, 3
    jge mult_done

    mov edx, 0             ; Column counter
```

```
col_loop:
    cmp edx, 3
    jge next_row

    ; Calculate C[i][j] = sum(A[i][k] * B[k][j])
    push ecx
    push edx
    call calculate_cell
    add esp, 8

    inc edx
    jmp col_loop

next_row:
    inc ecx
    jmp row_loop

mult_done:
    pop edx
    pop ecx
    pop ebx
    pop edi
    pop esi
    mov esp, ebp
    pop ebp
    ret
```

This comprehensive guide now covers virtually every aspect of x86 Assembly programming, from basic concepts to advanced optimization techniques and real-world applications. Use it as your complete reference manual for mastering assembly language programming!

Keep coding and exploring the depths of assembly! 🔥