x86 Assembly Guide

Assembly and its concepts


WHEN SOMEONE TELLS ME NOW
ASSEMBLY PROGRAMMING IS EASY
memegenerator.net

What you're going to learn

```
#include <stdio.h>
int main(){
    printf(“Hello World!\n”);
    return 0x1234;
}
```

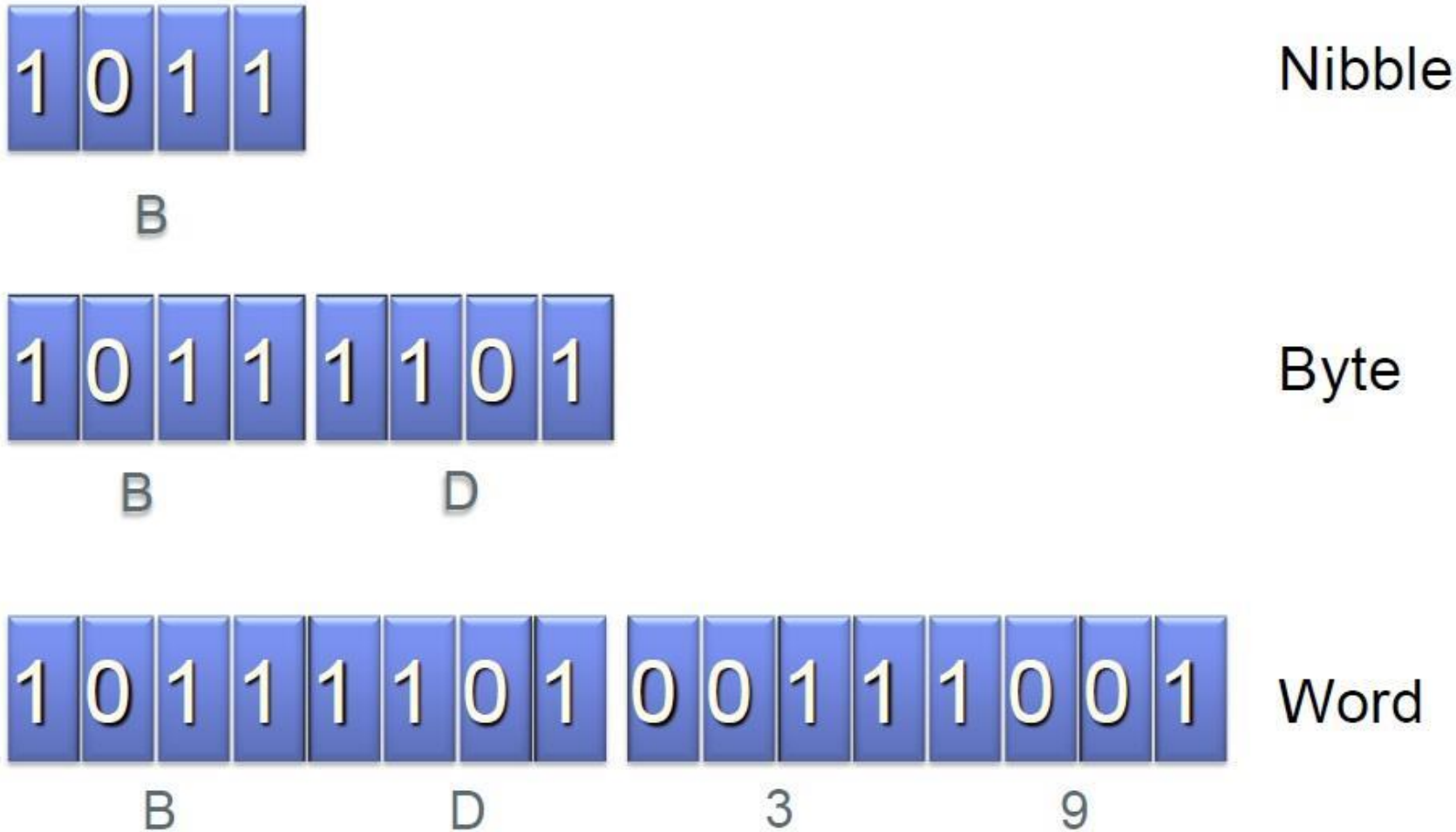
Is the same as…

```
.text:00401730 main
.text:00401730                 push    ebp
.text:00401731                 mov     ebp, esp
.text:00401733                 push    offset aHelloWorld ; "Hello world\n"
.text:00401738                 call    ds:__imp__printf
.text:0040173E                 add     esp, 4
.text:00401741                 mov     eax, 1234h
.text:00401746                 pop     ebp
.text:00401747                 retn
```

Back to the basic

Decimal, Binary, Hexidecimal

Decimal (base 10)	Binary (base 2)	Hex (base 16)
00	0000b	0x00
01	0001b	0x01
02	0010b	0x02
03	0011b	0x03
04	0100b	0x04
05	0101b	0x05
06	0110b	0x06
07	0111b	0x07
08	1000b	0x08
09	1001b	0x09
10	1010b	0x0A
11	1011b	0x0B
12	1100b	0x0C
13	1101b	0x0D
14	1110b	0x0E
15	1111b	0x0F


1 0 1 1
B
Nibble
1 0 1 1 1 1 0 1
B D
Byte
1 0 1 1 1 1 0 1 0 0 1 1 1 0 0 1
B D 3 9
Word

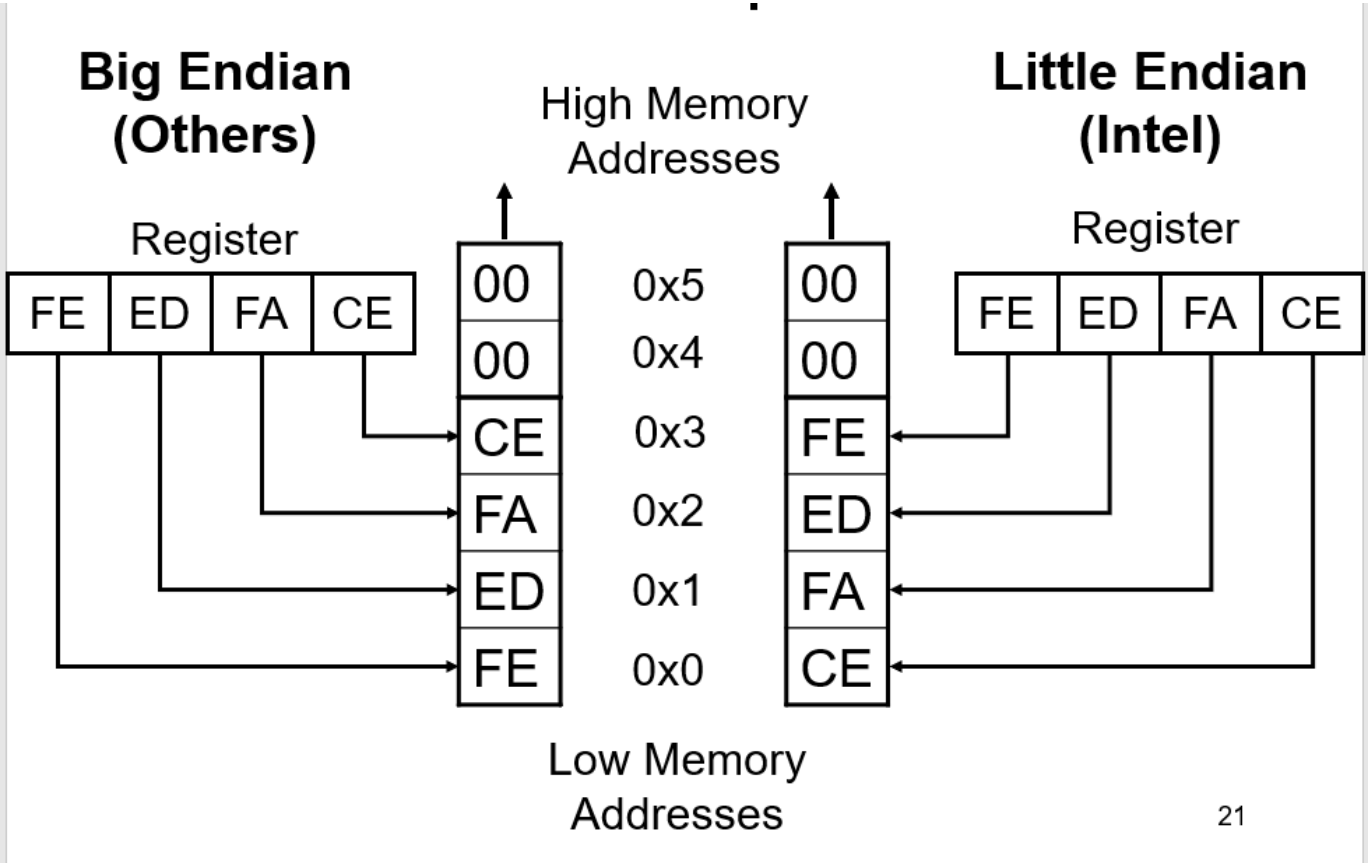
Byte Order

Endianness (Little and Big Endian)

Endianness

- Little endian
  - Offset value: 78 56 34 12
  - Actual value: 0x12345678
- Big endian
  - Offset value: 12 34 56 78
  - Actual value: 0x12345678

Endianess pictures



String Styles

- ASCIIZ – C style

H	e	l	l	o	
48	65	6C	6C	6F	00

- Null-terminated Unicode

H		e		l		o		
48	00	65	00	6C	00	6F	00	00

Registers

Registers

- Registers are small memory storage areas built into the processor
- On x86-32, registers are 32 bits long
- On x86-64, they're 64 bits

8/16/32bit general purpose registers

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EAX																															
reserved																AX															
																AH								AL							
ECX																															
reserved																CX															
																CH								CL							
EDX																															
reserved																DX															
																DH								DL							
EBX																															
reserved																BX															
																BH								BL							

Given, 12345678h

EAX = 12345678h

AX = 5678h

AH = 56h

AL = 78h

Data Registers

Register	Description	Usage
AL / AH / AX/ EAX	Accumulator Register	Arithmetic operations
BL / BH / BX / EBX	Base register	General data storage, index
CL / CH / CX / ECX	Counter register	Loop constructs
DL / DH / DX / EDX	Data register	Arithmetic

Register

- These are Intel's suggestions to compiler developers (and assembly handcoders).
- Registers don't have to be used these ways, but if you see them being used like this, you'll know why.

Example

- ADD EAX, EBX
  - add the value in EBX and EAX, and store the result into EAX
- SUB EAX, EBX
  - substitute the value in EBX from EAX, and store the result into EAX
- IMUL EAX, EBX
  - multiply EAX and EBX, and store the result into EAX

Address Registers

- IP / EIP
  - Instruction Pointer – Program execution counter
- SP / ESP
  - Stack Pointer - ESP will hold an offset to top of stacks memory location
- BP / EBP
  - Base Pointer - Stack frame
- SI / ESI
  - Source Index – String operation
- DI / EDI
  - Destination Index - String operation

X86 Instructions

NOP

- NOP - No Operation! No registers, no values, no nothin'!
- Bad guys use it to make simple exploits more reliable. But that's another class ;)
- The one-byte NOP instruction is an alias mnemonic for the XCHG (E)AX, (E)AX instruction.
  - NOP = XCHG EAX, EAX

The Stack

- The stack is a conceptual area of main memory (RAM) which is designated by the OS when a program is started.
- A stack is a Last-In-First-Out (LIFO/FILO) data structure where data is "pushed" on to the top of the stack and "popped" off the top.
- By convention the stack grows toward lower memory addresses. Adding something to the stack means the top of the stack is now at a lower memory address.

The Stack (cont.)

- As already mentioned, esp points to the top of the stack, the lowest address which is being used
- While data will exist at addresses beyond the top of the stack, it is considered undefined
- The stack keeps track of which functions were called before the current one, it holds local variables and is frequently used to pass arguments to the next function to be called.
- The push instruction automatically decrements the stack pointer, esp, by 4.

push

- Push stack
- Push the value into the stack
- Syntax
  - push <reg32>
  - push <mem>
- Examples
  - push eax

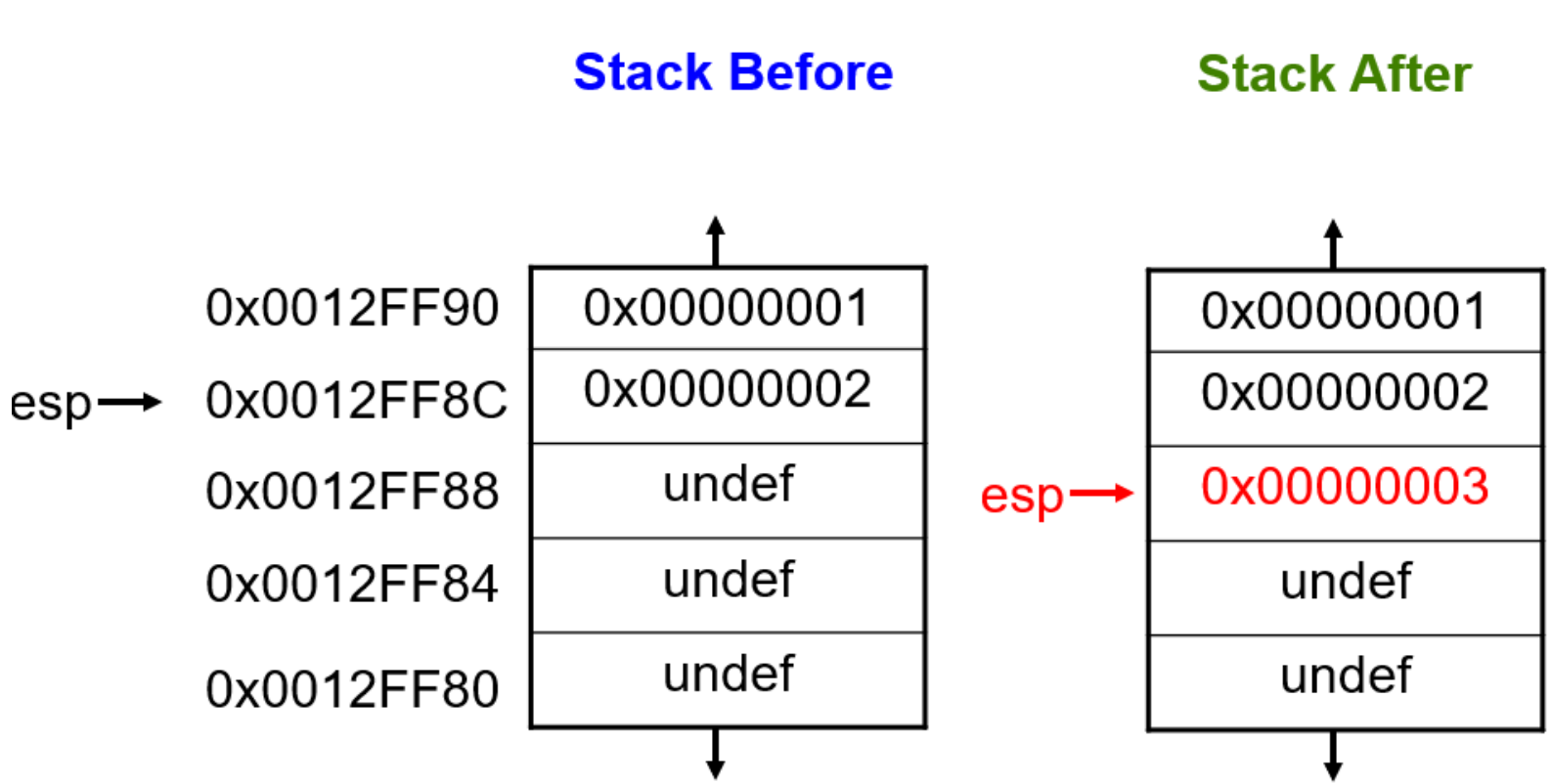
Registers Before

eax	0x00000003
esp	0x0012FF8C

push eax

Registers After

eax	0x00000003
esp	0x0012FF88



POP

- Pop a Value from the Stack
- Take a DWORD off the stack, put it in a register, and increment esp by 4

Registers Before

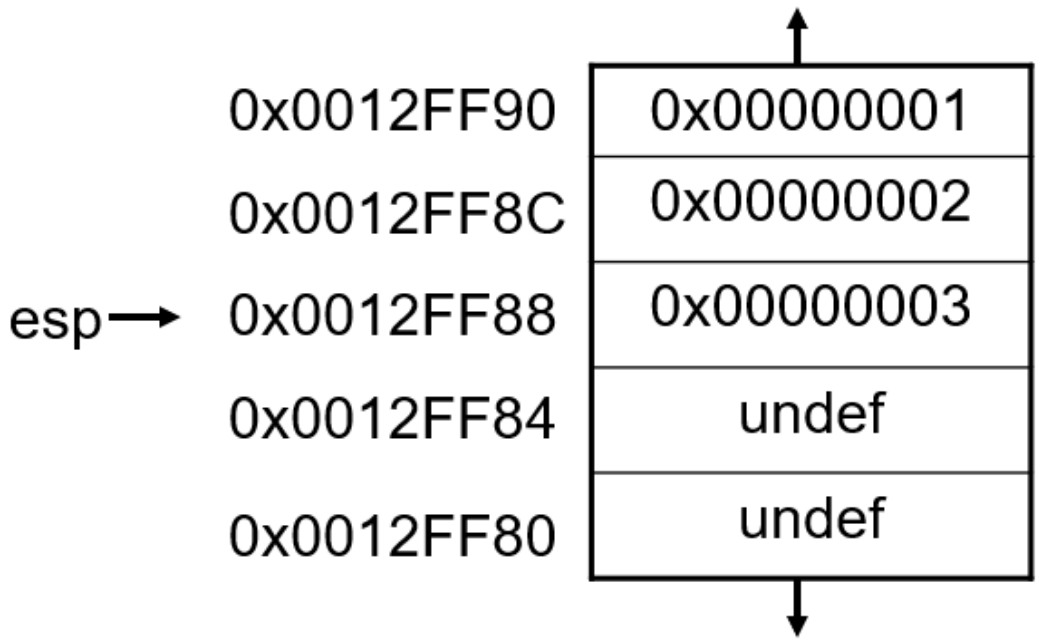
eax	0xFFFFFFFF
esp	0x0012FF88

pop eax

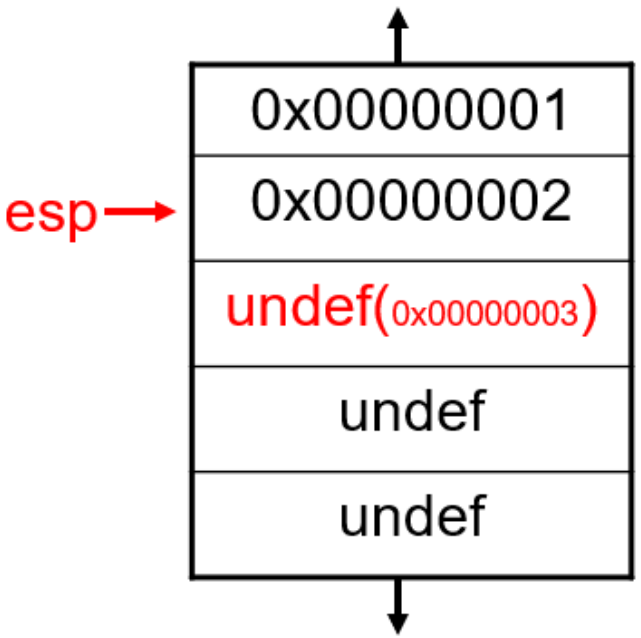
Registers After

eax	0x00000003
esp	0x0012FF8C

Stack Before



Stack After



CALL - Call Procedure

- CALL's job is to transfer control to a different function, in a way that control can later be resumed where it left off
- First it pushes the address of the next instruction onto the stack
  - For use by RET for when the procedure is done
- Then it changes eip to the address given in the instruction

RET - Return from Procedure

- Pop the top of the stack into eip

MOV

- Basically move data item from the second operand into first operand
- Never memory to memory!
- Syntax
  - mov <reg>,<reg>
  - mov <reg>,<mem>
  - mov <mem>,<reg>
- Examples
  - mov eax, ebx — copy the value in ebx into eax
  - mov byte ptr [var], 5 — store the value 5 into the byte at location var

Let’s look on example 1

LEA - Load Effective Address

- Frequently used with pointer arithmetic, sometimes for just arithmetic in general
- The contents of the memory location are not loaded, only the effective address is computed and placed into the register.
- Syntax
  - lea <reg32>,<mem>
- Examples
  - lea eax, [var] — the address of var is placed in EAX.

Arithmetic and Logic Instructions

Add - Integer Addition

- The add instruction adds together its two operands, storing the result in its first operand.
- Syntax
  - add <reg>,<reg>
  - add <reg>,<mem>
  - add <mem>,<reg>
- Examples
  - add eax, 10 — EAX ← EAX + 10
  - add BYTE PTR [var], 10 — add 10 to the single byte stored at memory address var

sub — Integer Subtraction

- The sub instruction stores in the value of its first operand the result of subtracting the value of its second operand from the value of its first operand.
- Syntax
  - sub <reg>,<reg>
  - sub <reg>,<mem>
  - sub <mem>,<reg>
- Examples
  - sub al, ah — AL ← AL - AH
  - sub eax, 216 — subtract 216 from the value stored in EAX

inc, dec — Increment, Decrement

- The inc instruction increments the contents of its operand by one.
- The dec instruction decrements the contents of its operand by one.
- Syntax
  - inc <reg>
  - inc <mem>
  - dec <reg>
  - dec <mem>
- Examples
  - dec eax — subtract one from the contents of EAX.
  - inc DWORD PTR [var] — add one to the 32-bit integer stored at location var

imul — Integer Multiplication

- The two-operand form multiplies its two operands together and stores the result in the first operand. The result (i.e. first) operand must be a register.
- Syntax
  - imul <reg32>,<reg32>
  - imul <reg32>,<mem>
  - imul <reg32>,<reg32>,<con>
  - imul <reg32>,<mem>,<con>
- Examples
  - imul eax, [var] — multiply the contents of EAX by the 32-bit contents of the memory
  - location var. Store the result in EAX.

idiv — Integer Division

- The idiv instruction divides the contents of the 64 bit integer EDX:EAX
- The quotient result of the division is stored into EAX, while the remainder is placed in EDX.
- Syntax
  - idiv <reg32>
  - idiv <mem>
- Examples
  - idiv ebx — divide the contents of EDX:EAX by the contents of EBX. Place the quotient in
  - EAX and the remainder in EDX.

and, or, xor

- Bitwise logical and, or and exclusive or
- These instructions perform the specified logical operation (logical bitwise and, or, and exclusive or, respectively) on their operands, placing the result in the first operand location.
- Syntax
  - and <reg>,<reg>
  - and <reg>,<mem>
  - and <mem>,<reg>
  - or <reg>,<reg>
  - or <reg>,<mem>
  - or <mem>,<reg>
  - xor <reg>,<reg>
  - xor <reg>,<mem>
  - xor <mem>,<reg>

and, or, xor

- Examples
  - and eax, 0fH — clear all but the last 4 bits of EAX.
  - xor edx, edx — set the contents of EDX to zero.

Control Flow Instructions

JMP - Jump

- Transfers program control flow to the instruction at the memory location indicated by the operand.
- Syntax
  - jmp <label>
- Example
  - jmp begin — Jump to the instruction labeled begin.

jcondition — Conditional Jump

- Jump if
- If else in C programming
- Syntax
  - je <label> (jump when equal)
  - jne <label> (jump when not equal)
  - jz <label> (jump when last result was zero)
  - jg <label> (jump when greater than)
  - jge <label> (jump when greater than or equal to)
  - jl <label> (jump when less than)
  - jle <label> (jump when less than or equal to)
- Example
  - cmp eax, ebx
  - jle done

cmp — Compare

- Compare the values of the two specified operands, setting the condition codes in the machine status word appropriately.
- Syntax
  - Syntax
  - cmp <reg>,<reg>
  - cmp <reg>,<mem>
  - cmp <mem>,<reg>
  - cmp <reg>,<con>
- Examples
  - cmp DWORD PTR [var], 10
  - je loop