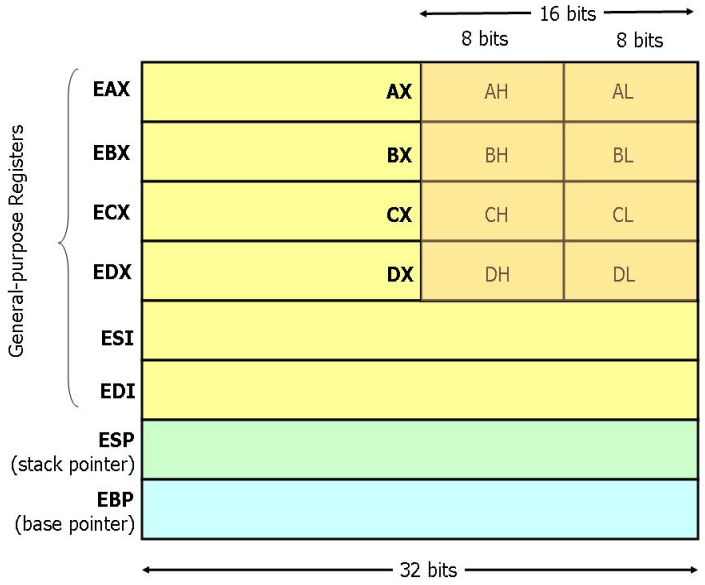
# Registers

In a computer, a register is one of a small set of data holding places that are part of a computer processor . A register may hold a computer instruction , a storage address, or any kind of data (such as a bit sequence or individual characters).

Modern (i.e 386 and beyond) x86
processors have eight 32-bit general purpose registers



# The Stack, Stack Pointer, and Base Pointer

The stack pointer is usually a register that contains the top of the stack. The stack pointer contains the smallest address **x** such that any address smaller than **x** is considered garbage.

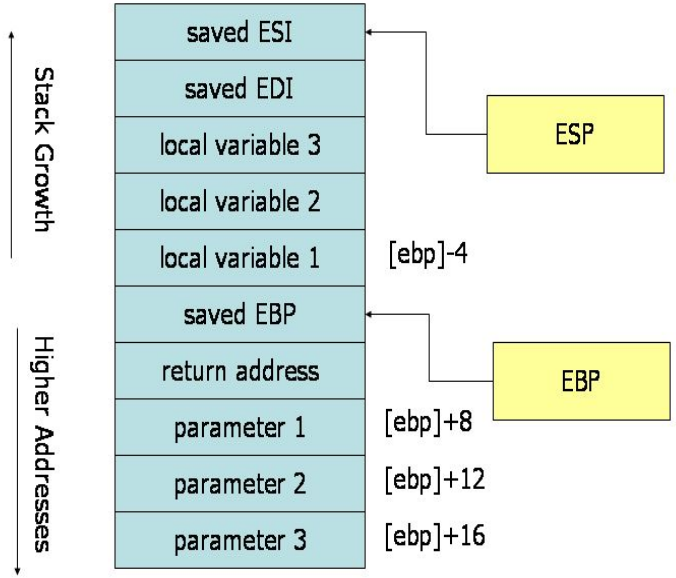
Whenever a function is called, a new stack frame is initiated.

-This is called the function prolog

```
push ebp            ; Preserve current frame pointer
mov ebp, esp        ; Create new frame pointer pointing to current stack top
sub esp, 20         ; allocate 20 bytes worth of locals on stack.
```

The stack grows towards Lower memory addresses. When a Local variable is declared, the Stack Pointer (`esp`) is decremented. The space between the Base pointer (`ebp`) and Stack Pointer is where all local variables are stored.

As shown in the picture, **local variable 1** is located at memory address `ebp-0x4`.



# 32 Bit Assembly Instructions (Intel)

```
<instruction> <destination> , <source>
     mov        eax         ,    ebx
```

The `mov` instruction copies the data item referred to by its second operand into the location referred to by its first operand . While register-to-register moves are possible, direct memory-to-memory moves are not. In cases where memory transfers are desired, the source memory contents must first be loaded into a register, then can be stored to the destination memory address.

-The [] are like a dereferencer in c (*ptr). So `esp+0x4` is the location in memory and `[esp+0x4]` is what is contained in that location

*Syntax* *Examples*

```
mov <reg>,<reg> :     mov eax, ebx          ; copy the value in ebx into eax
mov <reg>,<mem> :     mov eax, [epb-0x4]    ; copy the value of the variable stored at memory address ebp-0x4 into the eax register
mov <mem>,<reg> :     mov [ebp-0x4], ebx    ; Move the contents of ebx into the 4 bytes at memory address ebp-0x4
```

# 32 Bit Assembly Instructions (Intel)

In some cases the size of a referred-to memory region is ambiguous. Consider the instruction `mov [ebx], 2`. Should this instruction move the value 2 into the single byte at address `EBX`? Perhaps it should move the 32-bit integer representation of 2 into the 4-bytes starting at address `EBX`. Since either is a valid possible interpretation, the assembler must be explicitly directed as to which is correct. The size directives `BYTE PTR`, `WORD PTR`, and `DWORD PTR` serve this purpose, indicating sizes of 1, 2, and 4 bytes respectively.

```
mov BYTE PTR [ebx], 2       ; Move 2 into the single byte at the address stored in EBX.

mov WORD PTR [ebx], 2       ;  Move the 16-bit integer representation of 2 into the 2 bytes starting at the address in EBX

mov DWORD PTR [ebx], 2      ; Move the 32-bit integer representation of 2 into the 4 bytes starting at the address in EBX.
```

# 32 Bit Assembly Instructions (Intel)

**`push`** — Push stack (Opcodes: FF, 89, 8A, 8B, 8C, 8E, ...)

The `push` instruction places its operand onto the top of the hardware supported stack in memory. Specifically, `push` first decrements ESP by 4, then places its operand into the contents of the 32-bit location at address [ESP]. ESP (the stack pointer) is decremented by push since the x86 stack grows down - i.e. the stack grows from high addresses to lower addresses.

*Syntax*

`push <reg>`

`push <mem>`

`push <con32>`

*Examples*

`push eax` — push eax on the stack

`push [var]` — push the 4 bytes at address *var* onto the stack

# 32 Bit Assembly Instructions (Intel)

**`lea`** — Load effective address

The `lea` instruction places the *address* specified by its second operand into the register specified by its first operand. Note, the *contents* of the memory location are not loaded, only the effective address is computed and placed into the register. This is useful for obtaining a pointer into a memory region.

*Syntax*

```
lea <reg32>,<mem>
```

*Examples*

`lea edi, [ebx+4*esi]` — the quantity EBX+4*ESI is placed in EDI.
`lea eax, [esp+0x8]` — the value in *var* is placed in EAX.

# 32 Bit Assembly Instructions (Intel)

**`add/sub`** — Integer Addition and subraction

The `add` instruction adds together its two operands, storing the result in its first operand. Note, whereas both operands may be registers, at most one operand may be a memory location.

*Syntax*

```
add <destination>,<source>
```

*Examples*

```
add eax, 0x10 ;
```
EAX ← EAX + 10

```
add eax, ebx;
```
add the contents of `ebx` to the content of `eax` and store the result in `eax`

# 32 Bit Assembly Instructions (Intel)

**`cmp`** — Compare

Compare the values of the two specified operands, setting the condition codes in the machine status word appropriately.

The contents of the machine status word include information about the last arithmetic operation performed. For example, one bit of this word indicates if the last result was zero. Another indicates if the last result was negative. Based on these condition codes, a number of conditional jumps can be performed

This instruction is equivalent to the `sub` instruction, except the result of the subtraction is discarded instead of replacing the first operand.

*Syntax*

```
cmp <arg1>,<arg2>
```

# 32 Bit Assembly Instructions (Intel)

Conditional Jump

- A number of the conditional branches are given names that are intuitively based on the last compare instruction, `cmp`. For example, conditional branches such as `jle` and `jne` are based on first performing a cmp operation on the desired operands.

*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 ;  if the contents of EAX are less than or equal to the contents of EBX, jump to the label done. Otherwise, continue to the next instruction.
jle <Instruction address>
```

# 32 Bit Assembly Instructions (Intel)

**`call, ret`** — Subroutine call and return

These instructions implement a subroutine call and return. The `call` instruction first pushes the current code location onto the hardware supported stack in memory,  and then performs an unconditional jump to the code location indicated by the label operand. Unlike the simple jump instructions, the `call` instruction saves the location to return to when the subroutine completes.

Before a function is called, it's arguments are pushed onto the stack. So, If you ever want to know what is being passed to a function, just look at what is put on the stack directly proceeding it's call.

The `ret` instruction implements a subroutine return mechanism. This instruction first pops a code location off the hardware supported in-memory stack. It then performs an unconditional jump to the retrieved code location.

The return value of a function is always moved to eax. So to see what a function returned, just look at eax after the call.

*Syntax*

```
call <function>
ret
```