## High level over-view of basic:

**Pre-processing:** This is phase one, here preprocessor passes over source code, performing-

* Comment Removal
* Macro Expansion
* Include Expansion
* Conditional Compilation (IFDEF)

**Compilation:** It is the phase two of the process. this converts the output of the preprocessor into assembly instruction.

**Assemblers:** Assemblers convert assembly code into binary op-codes. Assemblers produce an object file. It contains fields that is to be filled by linker.

**Linking:** It defines entry point or starting instructions. It also defines memory regions on embedded platform.

## Output format

After linking, the output format can be in many formats. Some examples-

* PE - Portable Executable (Windows)
* ELF - Executable Linking Format (Linux)
* Mach-O (OSX)
* COFF/ECOFF etc.

## Basic Computer Architecture:

In short, when a program is running ->

1. Instruction is read into memory
2. It is processed by ALU
3. Result of the operation is stored into register or memory.

**X86-64 architecture:** This is the 64-bit version of the x86 instruction set and multiple operating mode available for backward compatibility. It has 16 (64 bit) general purpose registers.

**IP:** Instruction pointer. It points to the next instruction.

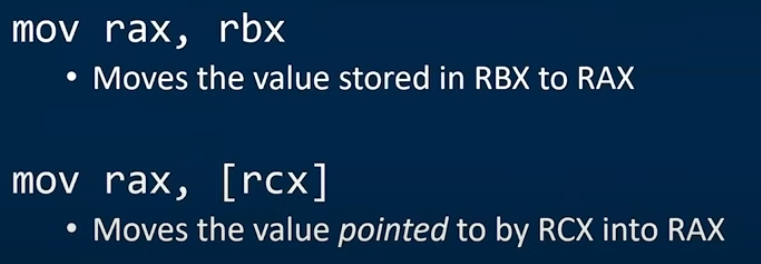
**FLAG:** Stores flags for processor flow control.

**PR0-PR7:** Floating point status and control register.

**BP/SP:** Base and stack pointer. Used for stack manipulation and uses. Base pointer is the base of the current functions stack frame.

**Common Instructions:**

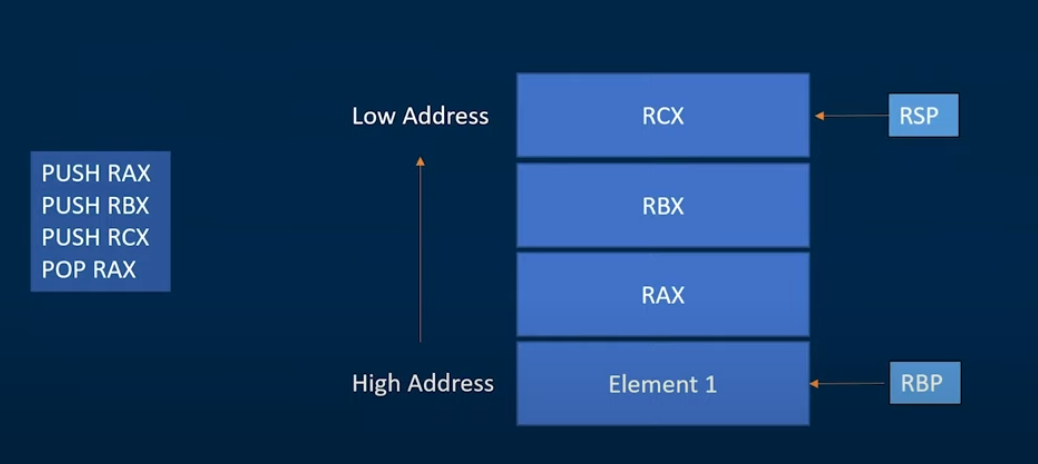
* **mov:** Moves data from one register to another.

****

* **add/sub:** In both operation result is stored in the first operand. In case of sub, second operand is subtracted from first operand.
* **and/or/xor:** These operations are similar and result is stored on the first operand.

**Stack:** It is a **LIFO** data structure. It is used to pass data to different locations, storing local variables etc. Stack address grows from **high to low**. **BP** points to the base of the stack and **SP** points to the top of the stack.

* **push:** It grows the stack by 4 in x86 architecture (8 in x64) and stores the operand content on the stack. In other words, it increases the value pointed by the SP by 4 and stores register value there.
* **pop:** loads the value pointed by SP into the operand. In this case, SP is decreased by 4.



**jmp/call:** jmp is used to changed what code will be executed. It modifies the value of the IP and executes the instruction there.call is used to implement function calls. It pushes the values of BP and IP into the stack before jumping.

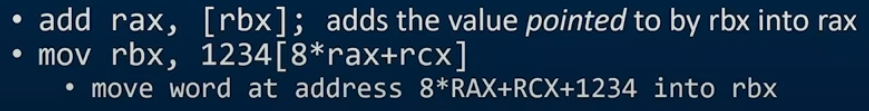
**cmp:** It perform a comparison operation by subtracting the operands. But it doesn’t store any value, only changes the flag registers depending on the result of the operation.

**Accessing modes:** Instructions can access registers and memory in various modes.

* **Immediate**: Value is stored in the instruction. Ex: add eax,14
* **Register to register**:



* **Indirect access**:



## Ghidra

During the analysis of the file, ghidra will attempt to create and label functions and identify cross reference in memory (xrefs-> these are generated when ghidra detects other locations or instruction reference this address).

* Function signature can make the output of the decompiler much better. We can alter return type, argument count and argument type. To do this right click the function name -> edit function signature.