Registers – working memory for CPU, they hold values during the computation, but data isn’t meant to be stored in them for long. Some registers are general purpose while others are special purpose.

Registers are fixed size. The size of registers depend on your CPU architecture, 32-bit machines have 32 bit registers, and 64 bit CPU’s have 64 bit registers.

Backwards compatibility is maintained by allowing access to subsets of registers, 32 bit code can still run on 64 bit machines because the processor can treat half of the 64 bit register as though it was a 32 bit register.

The stack, this is a region of memory that behaves as a last in first out LIFO data structure, this means that we can push values onto the stack to store them and then pop values off of the stack to read them. Like the name suggest it behaves similar to a stack of books we’re pushing is like putting a book on top of the stack, and popping is like taking a book from the top of the stack.

The stack is a contiguous array, pushing and popping are accomplished by moving an index or pointer, this pointer holds the location of the top of the stack, so when you push a value the pointer is moved, and the value is written to the new pointer location. Push another value and the pointer is moved again, and the new value is written into this new pointer location. Then when you pop a value it reads from the current pointer location, and then moves the pointer location backwards. It’s important to know this because we actually do have random access to all of this memory and we aren’t limited to just pushing and popping the stack the stack pointer that I mentioned is also just a register that we can change and alter, we can change whenever we want to alter the top of the stack.

Assembler converts programs into assembly code into machine code

;entry point into the program

; this is where the processor will start executing instructions from

; global keywords is used to make an identifier accessible to the linker

global \_start

; the identifier is start, start followed by a colon will create a label

; labels are used to name locations in code

\_start:

; mov is used to move data into a register

; here the move instruction moves the integer 1 into the register eax

mov eax, 1

; moves integer into ebx register

mov ebx, 42

; preform interrupt, processor transfers control to interrupt handler

; that we have specified by the value. in this case 0x80 is the interrupt

; handler for system calls. the system call that it makes is determined by the eax

; register. the value 1 means that we're making an exit call, system exit call

; this will signal the end of out program

int 0x80

; the value stored in ebx is the exit status for our program

; used 42 for example but this could be any integer

; to compile on Linux:

; nasm -f elf32 filename.asm -o objectname.o

; -f elf32 means build 32 bit elf object file

; elf stands for executable and linking format

; executable format used by linux

; to compile on windows

; nasm -f win32 filename.asm -o objectname.o

; win32 in executable format for windows

Unlike most languages assembly lacks the usual constructs like iteration, conditions and function calls. Instead the programmer has to build this kind of behavior using jump instructions.

Instruction Pointer

EIP internal pointer inside of the processor, holds the location for your machine code that the processor is executing. This means the processor can jump around to different locations in the code by simply altering this pointer. Unlike a register you can’t change the instruction pointer by using the normal move, add and subtract operations. Instead the instruction pointer is changed by using jump operations.