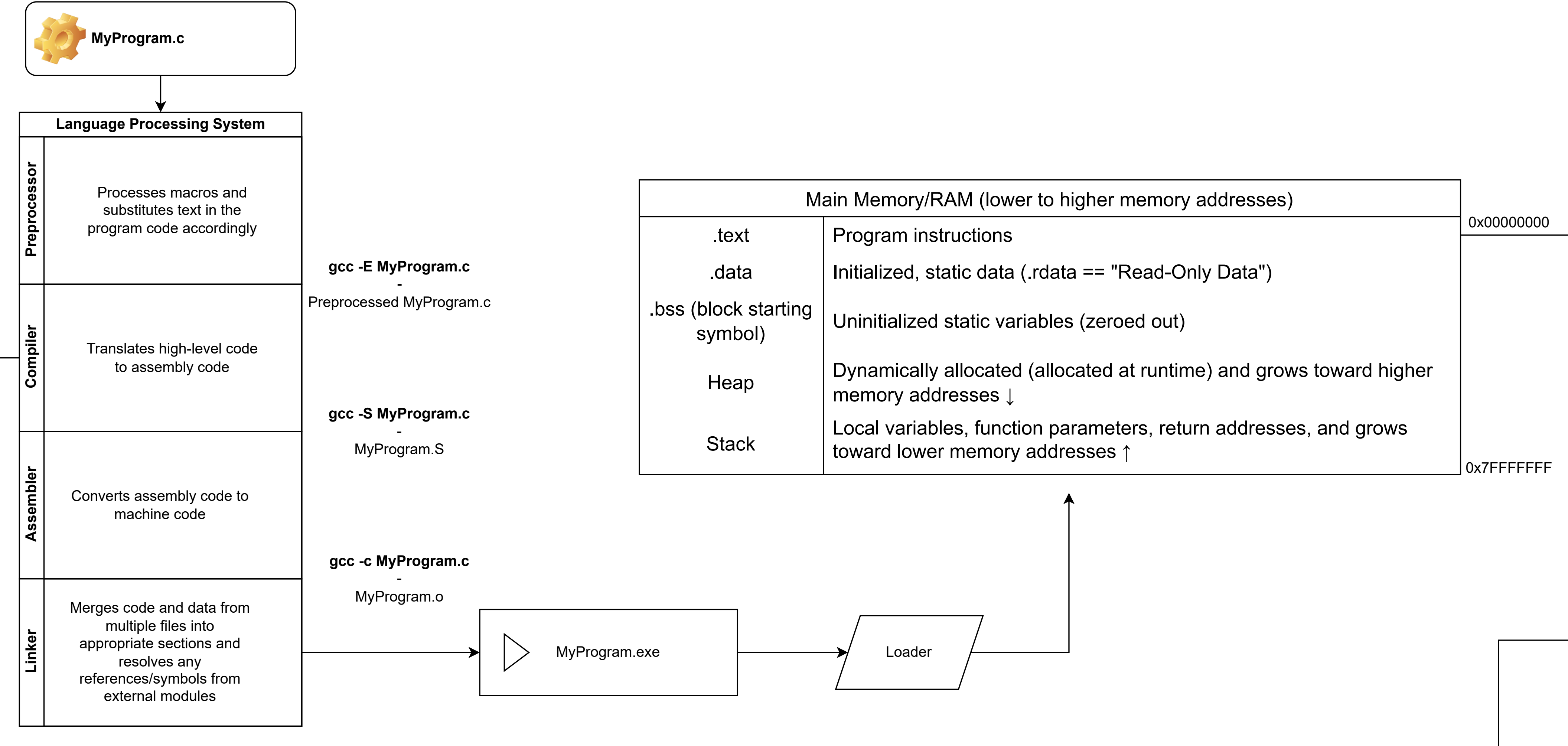
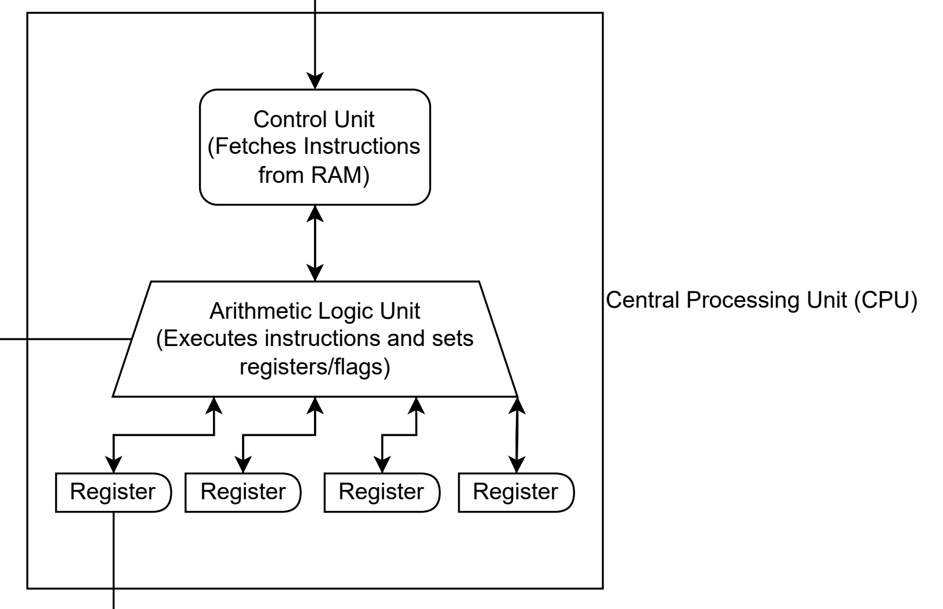

MyProgram.c
Language Processing System
Preprocessor
Processes macros and substitutes text in the program code accordingly
Compiler
Translates high-level code to assembly code
Assembler
Converts assembly code to machine code
Linker
Merges code and data from multiple files into appropriate sections and resolves any references/symbols from external modules
gcc -E MyProgram.c - Preprocessed MyProgram.c
gcc -S MyProgram.c - MyProgram.S
gcc -c MyProgram.c - MyProgram.o
MyProgram.exe
Loader

Main Memory/RAM (lower to higher memory addresses)

.text	Program instructions
.data	Initialized, static data (.rdata == "Read-Only Data")
.bss (block starting symbol)	Uninitialized static variables (zeroed out)
Heap	Dynamically allocated (allocated at runtime) and grows toward higher memory addresses ↓
Stack	Local variables, function parameters, return addresses, and grows toward lower memory addresses ↑

0x00000000

0x7FFFFFFF


Control Unit (Fetches Instructions from RAM)
Arithmetic Logic Unit (Executes instructions and sets registers/flags)
Register
Register
Register
Register
Central Processing Unit (CPU)

Common x86 Calling Conventions (decided by compiler or programmer)

cdecl	Parameters pushed onto stack from right-to-left; Caller cleans up the stack and return value stored in EAX
stdcall	Same as cdecl, except called function cleans up the stack
fastcall	First few arguments pushed to registers (commonly ECX and EDX), additional parameters are pushed right-to-left, calling function cleans up

Common x86 Instructions

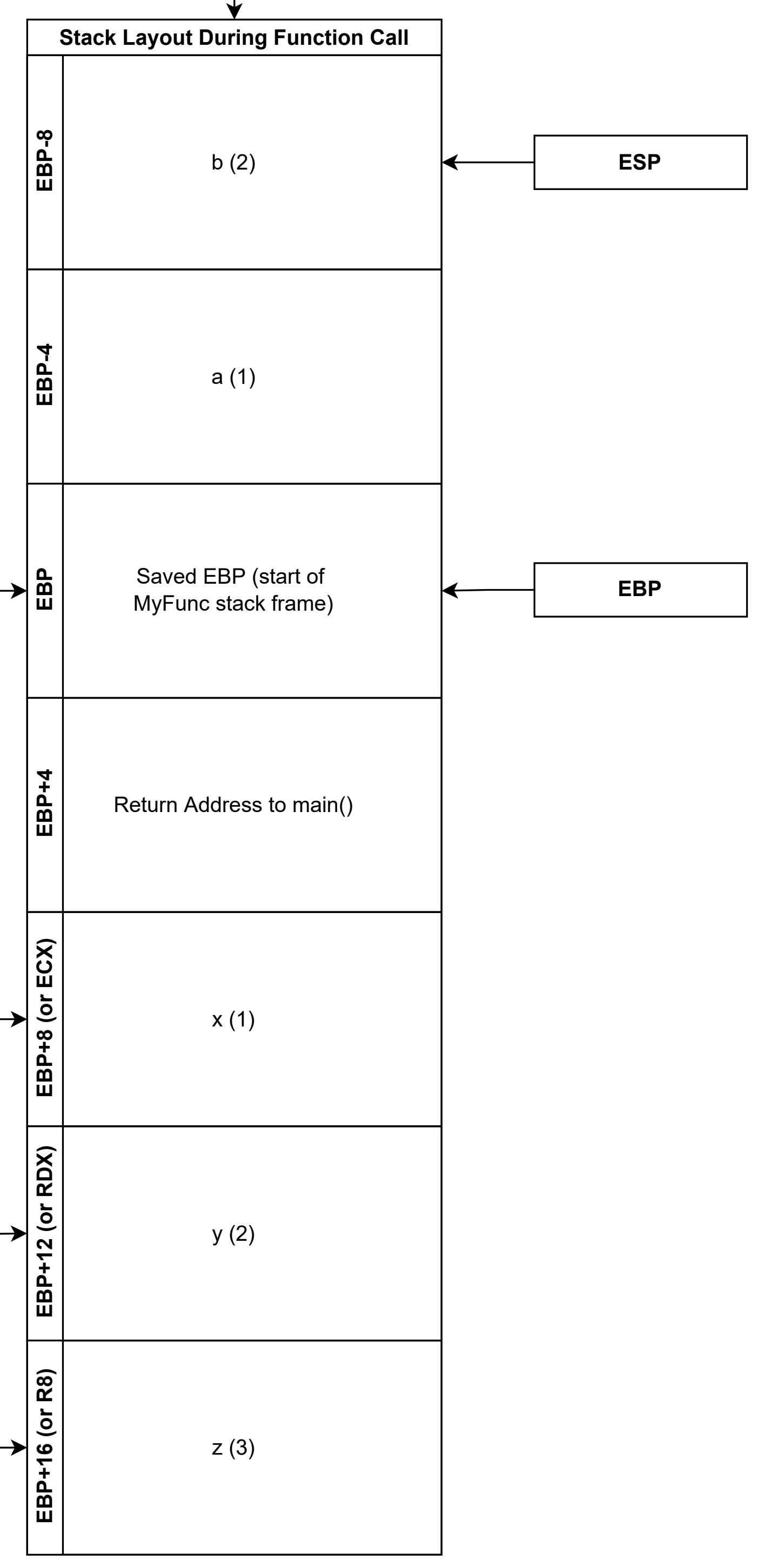
mov eax, ebx; mov eax, 0x13, mov eax, [0x4000000]	Copy a value (from register, from literal, or from address) to a register
lea eax, [ebx+esi*4]	Load effective address; Similar to move, but loads the address "ebx + esi * 4" itself into a register rather than the data at that address
add eax, 0x1; sub eax, 0x1; inc eax; dec eax	Add, subtract, increment, or decrement the value in a register
mul eax, 0x5; div eax, 0x5; imul eax, 0x5; cdq; idiv eax, 0x5;	Multiply the value in EAX or Divide the value in EDX:EAX, and store results in EDX:EAX (for division, result in EAX, remainder in EDX); imul and idiv are signed operations (cdq is used prior to idiv to sign-extend EAX to EDX)
xor eax, eax; or eax, ebx; and eax, ebx; not eax	XOR, OR, AND, and NOT bitwise operations
shl bl, 0x4; shr bl, 0x4; rol bl, 0x4; ror bl, 0x4;	Bitwise shift and and rotate operations (bits shifted "fall off" vs bits rotated are cycled back to the least significant bit) (NOTE: There are variations such as SAR/SAL which you may see used instead of SHR/SHL to preserve sign bits - also note that SAL and SHL perform exactly the same operations, whereas SHR and SAL do not).
nop	No operation; Just do absolutely nothing and wait for the next thing to happen (relatable, amirite?)
jz 0x4000000; jnz ...; je ...; jne ...; jg ...; jge ...; jl ...; jle ...; ja ...; jb ...; jae ...; jbe ...; jo ...; js ...	Conditional jumps (zero, not zero, equal, not equal, greater than, greater than or equal to, less than, less than or equal to, greater than (unsigned), greater than or equal (unsigned); less than or equal to (unsigned), overflow bit set, sign bit set
test eax, eax; cmp eax, 0x4	Test is the same as AND and sets the zero flag (test eax, eax is the same as checking if eax is 0); cmp is identical to SUB but only sets zero and carry flags
rep; repe; repz; repne; repnz;	Increments ESI and EDI offsets and decrements ECX; rep continues until ECX is 0, repe/repz/repne/repnz continue until ECX is 0 or the zero flag is set (repe/repz stop if ZG = 0; repne/repnz stop if ZF = 1)
repe cmpsb	EDI and ESI are two buffers; ECX is buffer length; Compares both buffers until ECX = 0 or a difference is found in the buffer contents
rep stosb	Initialize all values of the buffer at EDI to the value in AL
rep movsb	ESI is source buffer; EDI is destination buffer; ECX is length of bytes to copy; Copies these bytes from ESI to EDI until ECX is 0
repne scasb	EDI is the address of a buffer; AL contains a search byte; ECX is the buffer length; Searches the buffer for the search byte until it is found or ECX is 0
push eax; pop ebx; pusha; pushad; popa; popad	Pushes the value in EAX onto the stack (ESP); Pops the value at the top of the stack into EBX and adjusts ESP; Pushes 16-bit general purpose registers on the stack; Pushes the 32-bit general purpose registers on the stack; Pops 16-bit values from stack into general purpose registers; Pops 32-bit values from stack to general purpose registers
call 0x41001000	Calls a function; Moves value of EIP to stack and sets EIP to the start of the function at 0x41001000
ret	Pops the return address off of the stack into EIP
MOAR	But wait, there's more! Check the video resource links for more information on registers - there are a lot of them!

Common x86 Registers

RAX/EAX/AX/AH/AL	Accumulator; Used for input/output, arithmetic,and return values from functions
RBX/EBX/BX/BH/BL	Base; Used for indexed addressing (using one register as base and other as index)
RCX/ECX/CX/CH/CL	Count; Stores loop count variables in iterative operations
RDX/EDX/DX/DH/DL	Data; Input/output, sometimes extends RAX for multiply/divide
RSP/ESP/SP/SPL	Stack Pointer; Stores current position within the stack
RBP/EBP/BP/BPL	Base Pointer; Helps in referencing parameter and other stack variables as offsets from the "base" of the stack
RSI/ESI/SI/SIL	Used as a source index for string operations
RDI/EDI/DI/DIL	Used as a destination index for string operations
RIP/EIP/IP	Stores next instruction to be executed
R8-R15	x64 general purpose registers
CS/DS/SS/ES/FS/GS	16-bit segment registers for accessing specific areas of memory segments, including: Code (.text)/Data (.data)/Stack/Extra/General/General
RFLAGS/EFLAGS	Status register holding one-bit flags, e.g. ZF (zero-flag), CF (carry-flag), SF (sign-flag), TF (trap-flag)
MOAR	But wait, there's more! Check the video resource links for more information on registers - there are a lot of them!

```
int main() {
        ...
        int res = MyFunc(1, 2, 3);
        ...
}

int MyFunc(int x, int y, int z) {
        int a = 1;
        int b = 2;
        return ((a * x) + (b - y * z));
}
```



Parentheses show how parameters are passed in fastcall/x64.

Note that XMM0, XMM1, XMM2, etc. are used instead of RCX/RDX/R8/etc. for parameter values that are floating-point and for non-scalar return values.