CMSC389R

Binaries I

COMPUTER SCIENCE
UNIVERSITY OF MARYLAND

recap

HW4

--

Questions?

Itinerary

- How programs work
- Compilation process
- Instruction Set Architectures
- x86 Assembly
  - Language

Computer Programs

- Interpreted
  - Write source code (Python, Ruby, etc)
  - Run in interpreter

- Compiled
  - Write source code (Java, C, etc)
  - Compile (*javac*, *gcc*, *etc*)
  - Run it

Compilation Process

- Source Code: human written program
- Assembly: human readable mnemonics of machine language (though translation is not always one-to-one)
- Machine code: ones/zeros the CPU directly interprets

Compilation Process

- Compiler: code -> assembly
- Assembler: assembly -> machine code
- Linker: resolves external dependencies (imports, libraries)
- Output of all this?
  - Typically an ELF file (Linux) or Portable Executable / PE file (Windows)... A binary

Assembly Language

- We’ll be using x86 assembly in 32 bit mode
- Why still learn assembly?
  - Reverse Engineering (here)
  - OS development
  - Compiler writing
  - Computer architecture design

x86

- Registers
- Syntax
- Instructions
  - Arithmetic
  - Data
  - Control Flow
- Calling Conventions
- Tooling

Registers

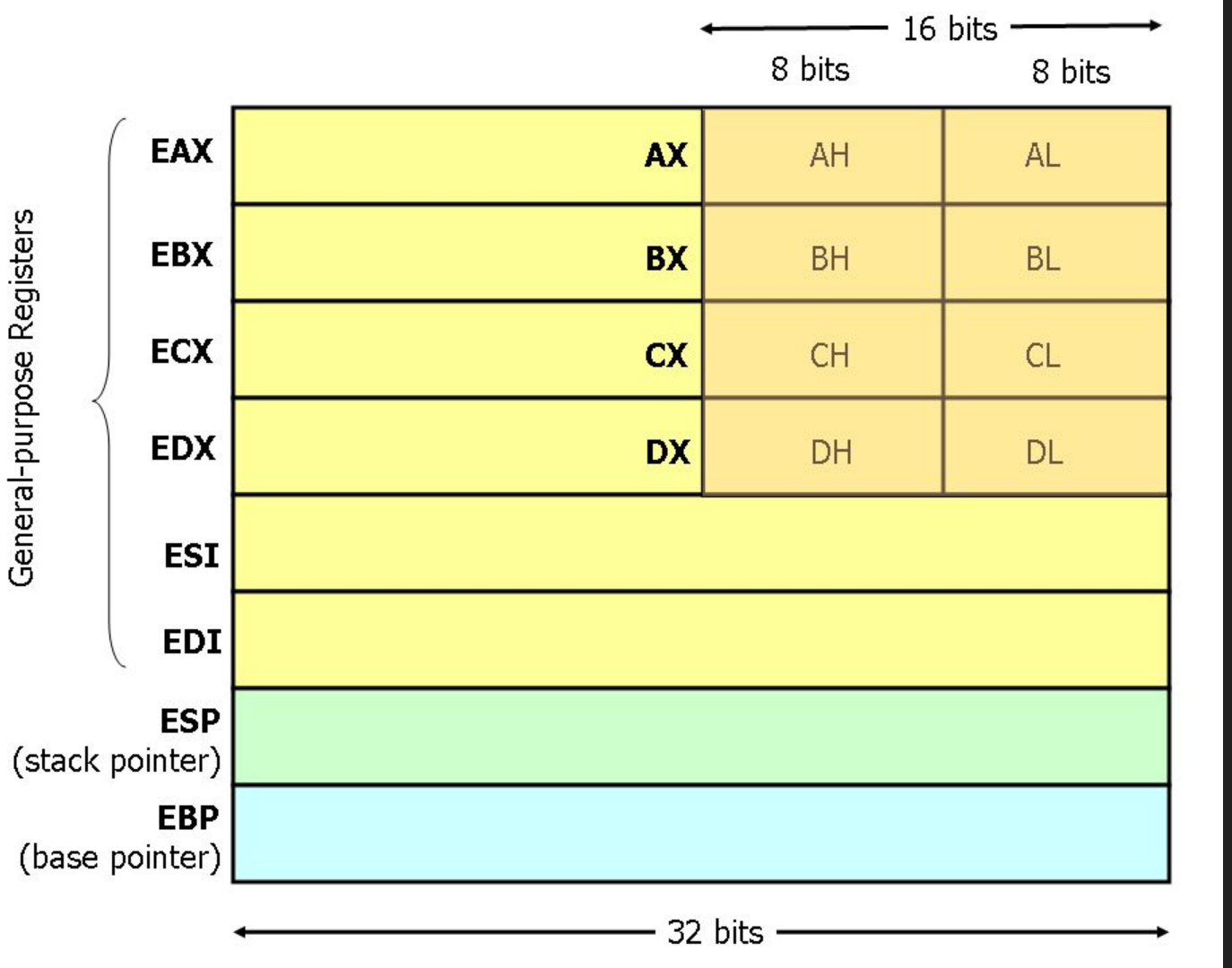
- EAX - “Accumulator” register
  - Heavy use for arithmetic, also often return value
- EDX - “Data” register
  - Closely tied with EAX operations
  - e.g. stores extra data from multiplication
- ECX - “Counter” register
  - Used as loop counter and for bit shifting
- EBX - “Base” register
  - Used to be memory base pointer in 16-bit x86, but has no special purpose now :(

Registers

- Can access lower parts of EAX/EBX/ECX/EDX with smaller registers
  - EAX - “Extended” AX
  - AX - lower 16 bits of EAX
  - AH - upper 8 bits of AX
  - AL - lower 8 bits of AX
  - Same with other letters (B, C, D)
- Can only use registers together with same size
  - Need to use expansion instructions to interface w/ bigger registers

Registers

- ESI/EDI - Source/Destination Index
  - Used as a pointer for things like string manipulation (also often params)
- EBP - Base Pointer
  - Points to the bottom of the current stack frame
  - Use to reference function parameters
- ESP - Stack Pointer
  - Points to top of stack
  - Used to grow/shrink stack for local variables/data
- More on history here https://www.swansontec.com/sregisters.html


16 bits
8 bits
8 bits
General-purpose Registers
EAX
AX
AH
AL
EBX
BX
BH
BL
ECX
CX
CH
CL
EDX
DX
DH
DL
ESI
EDI
ESP
(stack pointer)
EBP
(base pointer)
32 bits

Arithmetic Instructions

- *add* - adds two values together
- *sub* - subtracts source from destination value
- *inc* - increments by 1
- *dec* - decrements by 1
- *imul* - performs integer multiplication
- *idiv* - performs integer division
  - quotient -> EAX, remainder -> EDX
- *and/or/xor/not* - bitwise operations
- *neg* - performs two’s complement negation
- *shl/shr* - left and right shift by immediate or CL

Arithmetic Instructions

- *add* - adds two values together
- *sub* - subtracts source from destination val
- *inc* - increments by 1
- *dec* - decrements by 1
- *imul* - performs integer multiplication
- *idiv* - performs integer division
  - quotient -> EAX, remainder -> EDX
- *and/or/xor/not* - bitwise operations
- *neg* - performs two's complement negation
- *shl/shr* - left and right shift by immediate

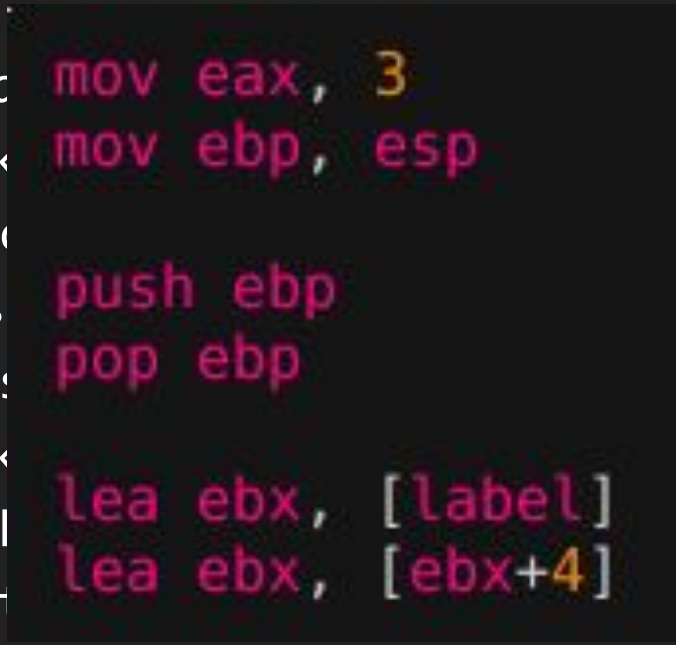
```
add eax, eax
add eax, [ebx+4]
add [ebx], 3

sub ecx, 2
dec ecx
inc [ebx+12]

imul eax, 3
idiv eax, 12

and eax, 0ffh
or eax, 2
xor eax, eax
not eax

neg edx
shl eax, 2
shr eax, 2
```

Data Manipulation Instructions

- *mov* - copies data from source operand to destination
- *push* - pushes value onto top of stack
  - Makes room on the stack by subtracting ESP by 4
    - Stack grows from higher address to lower address
  - Copies the value from operand to stack
- *pop* - removes value from top of stack
  - Copies value from top of the stack
  - Decreases stack size by adding 4 to ESP
- *lea* - “load effective address” of some value in memory
  - Use *[base + index*scale + offset]*
  - Base/index are registers, scale/offset are immediates

Data Manipulation Instructions

- *mov* - copies data from source operand
- *push* - pushes value onto top of stack
  - Makes room on the stack by subtrac
    - Stack grows from higher addres
  - Copies the value from operand to s
- *pop* - removes value from top of stack
  - Copies value from top of the stack
  - Decreases stack size by adding 4 t
- *lea* - “load effective address” of some value in memory
  - Use *[base + index*scale + offset]*
  - Base/index are registers, scale/offset are immediates



Control Flow Instructions

- Use labels to mark important sections in data
- *jmp* - unconditional jump to label (ALWAYS happens)
- *cmp* - compares two values and stores metadata in a special register called FLAGS
  - Contains status on last operation
  - *cmp* essentially does *sub* and only modifies FLAGS
- *je/jne/jz/jg/jge/jl/jle* - conditional *jmp* based on FLAGS
- *call* - jumps to label as if it were a function
- *ret* - return from a function call
- *syscall* - call OS level functions for I/O, etc

Control Flow Instructions

- Use labels to mark important sections
- *jmp* - unconditional jump to label (ALL
- *cmp* - compares two values and stores r special register called FLAGS
  - Contains status on last operation
  - *cmp* essentially does *sub* and only
- *je/jne/jz/jg/jge/jl/jle* - conditional
- *call* - jumps to label as if it were a
- *ret* - return from a function call
- *syscall* - call OS level functions for I/O, etc

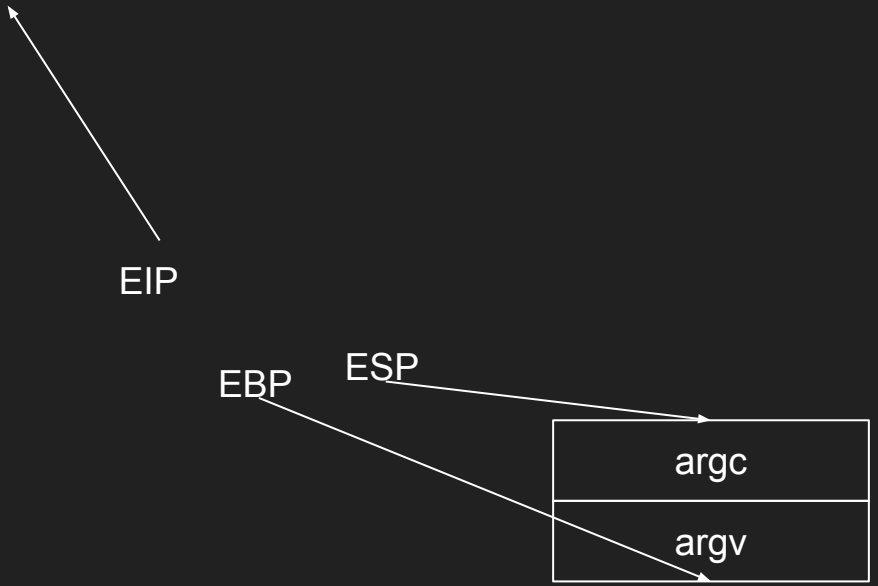
```
jmp label
cmp eax, 2
je equal_label

sub eax, 50
jz zero_label

call printf

ret

mov eax, 1
mov esi, hello_world_label
mov edx, 11
syscall
```

More Instructions

- More instructions here with explanation http://www.felixcloutier.com/x86/
- More here http://ref.x86asm.net/
- C compiler explorer https://godbolt.org/
  - Can type C code and view the disassembly

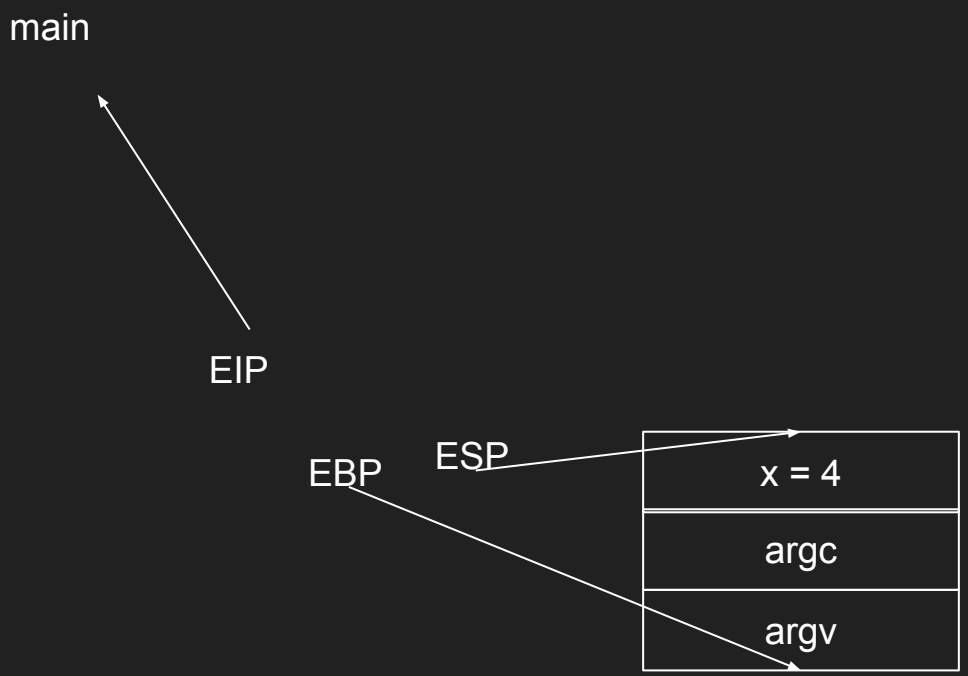
Stack Frames

EIP (instruction pointer) points to the code we are currently executing… somewhere within main.



About to call a function, “foo” with, let’s say, int x = 4, from within main.

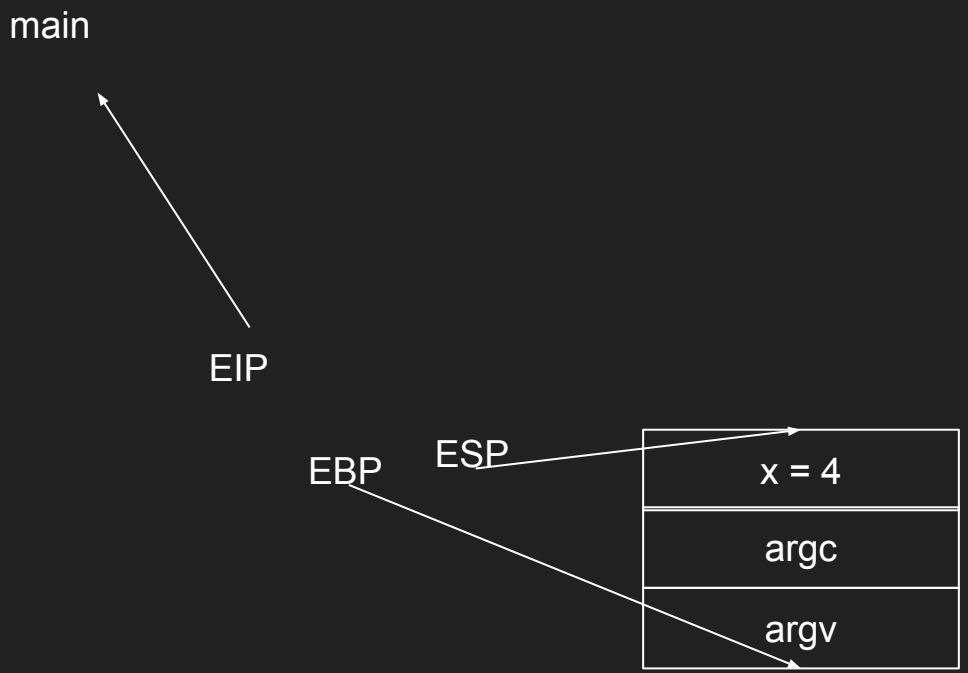
Stack Frames



We will create space on the stack for our new variable “x”. If we are not creating a variable we would pass a reference directly to a memory location in the .data section (where hardcoded values tend to exist).

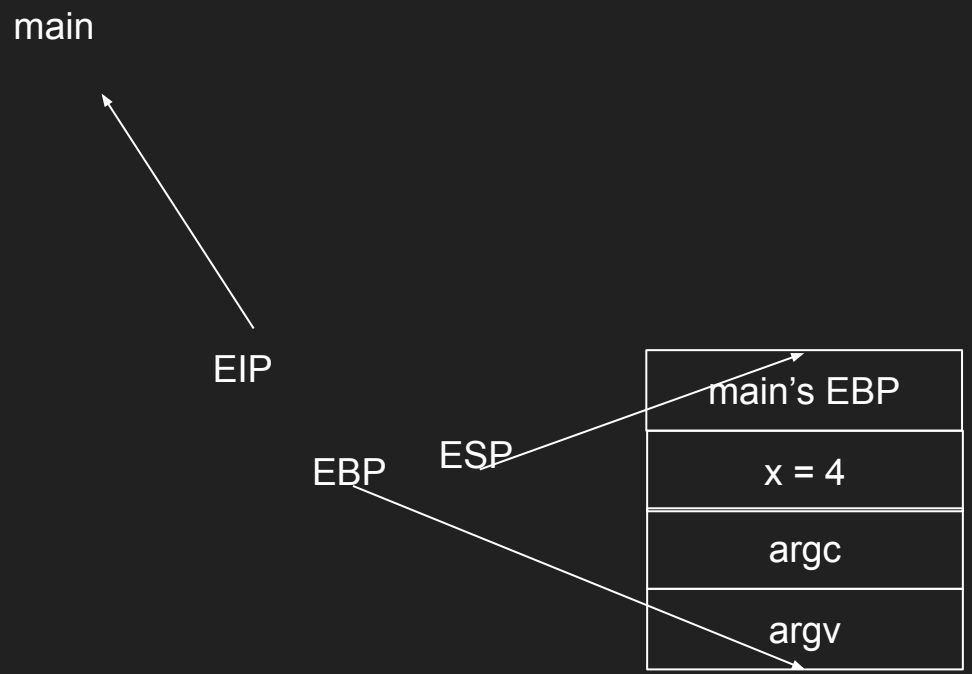
ESP (pointer to top of stack) adjusts to allow new variable in

Stack Frames



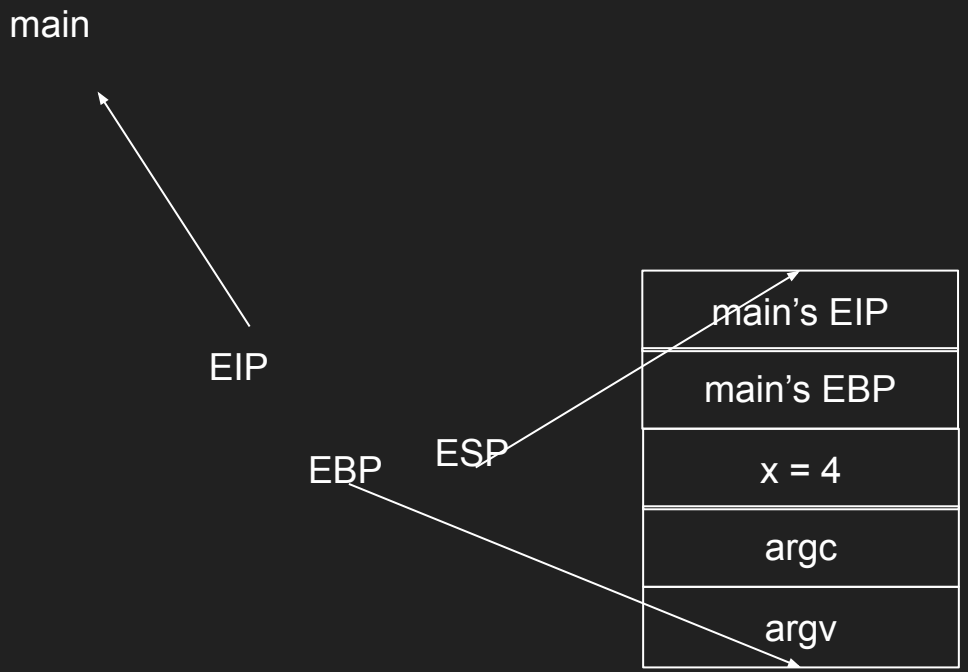
EDI=4 OR push x again… we will assume EDI=4

Stack Frames



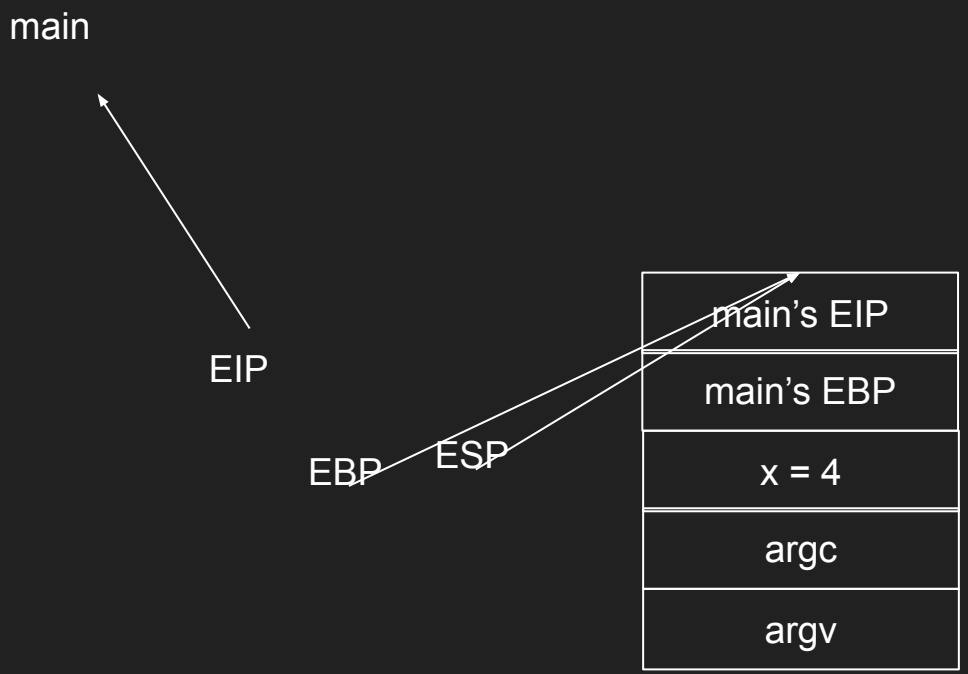
Push our EBP...

Stack Frames



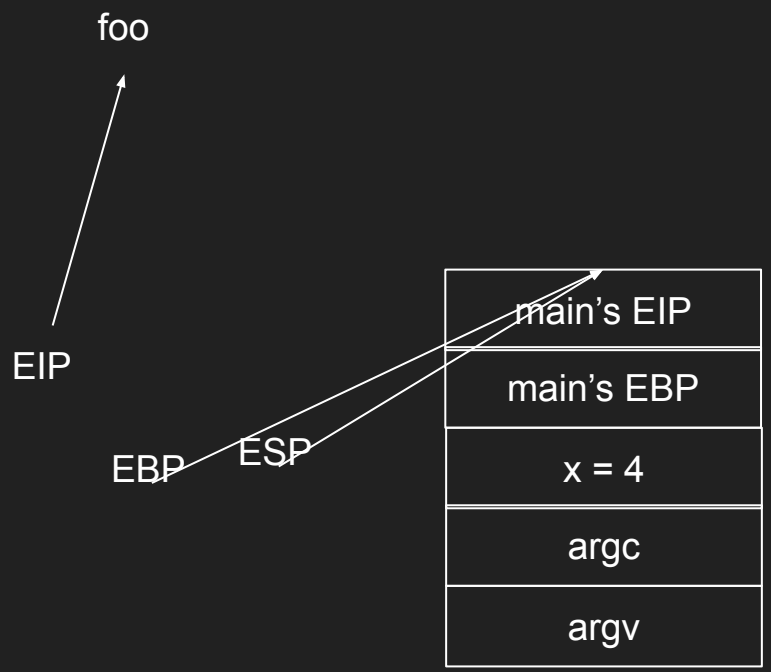
then push our EIP so we can return to main when foo is done.

Stack Frames



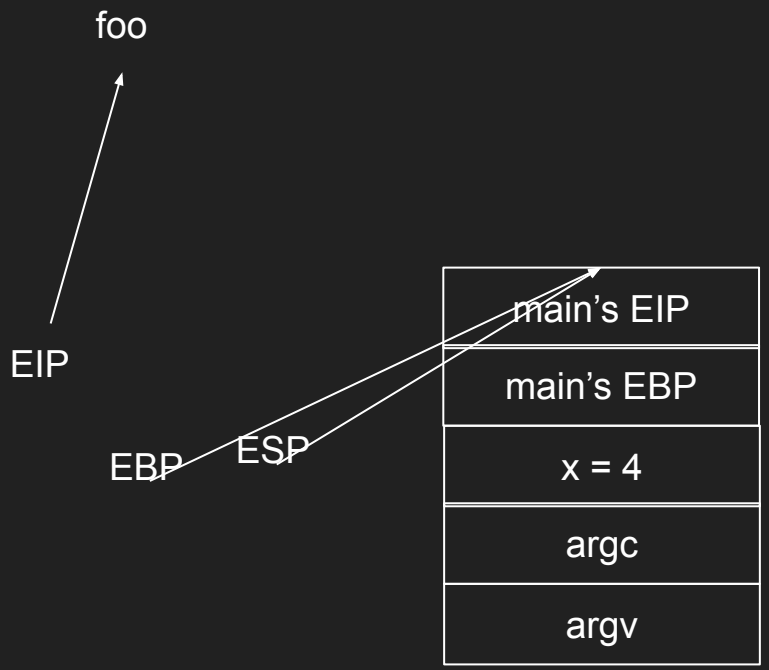
Now that we can return to main's stack from using EBP from stack, we can set EBP to ESP to reset our stack frame for a new function. Notice that EBP==ESP: our new function has no local variables, but would push them to this "new" stack if it did

Stack Frames



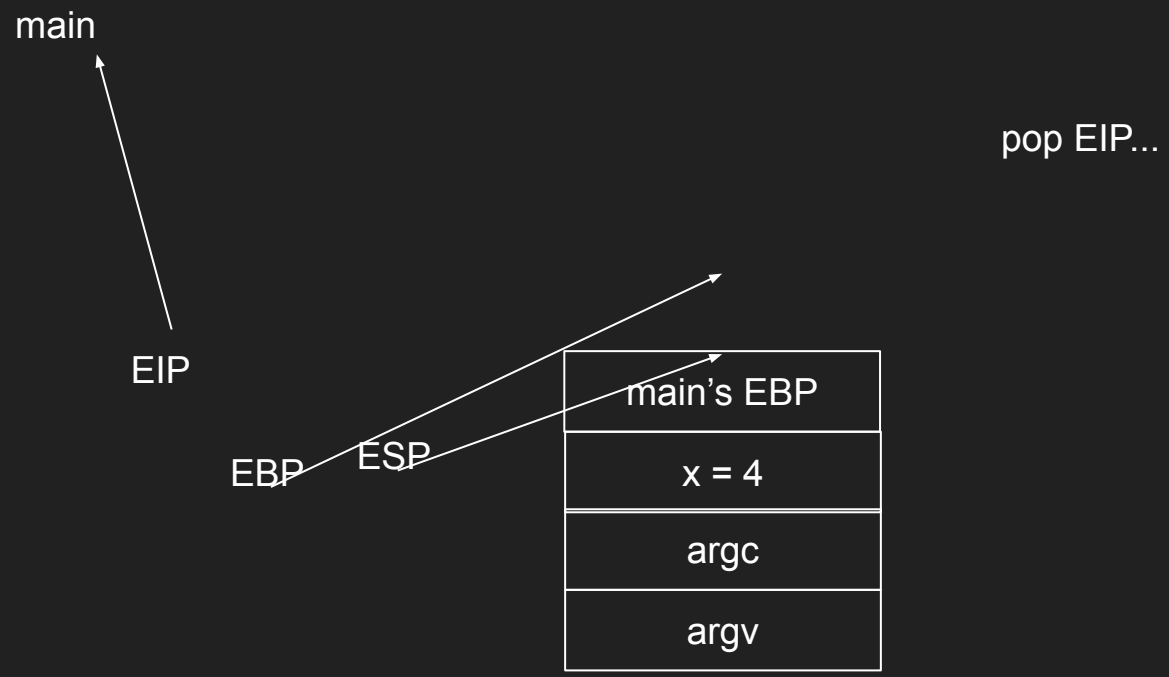
Finally, we have a new stack frame initialized, we can access our parameter (either via parameter register EDI or via the stack) and EIP (instruction pointer) is in foo -- we are executing foo's code!

Stack Frames



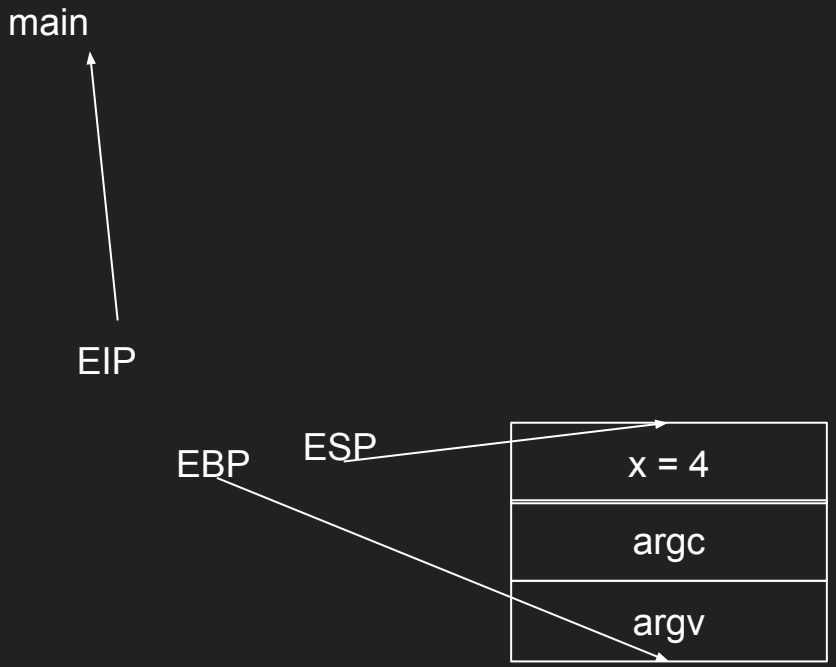
Let's imagine foo has finished executing -- how do we return?

Stack Frames



pop EIP...

Stack Frames



pop EBP!

ELF Sections

- Declare a section with *section .sect*
- *.text* - where assembly code goes
- *.data* - where hardcoded data goes
  - Strings
  - Constants
  - Formatting here https://www.tortall.net/projects/yasm/manual/html/nasm-pseudop.html
- *.comment* - comments can go here
- More http://www.tortall.net/projects/yasm/manual/html/objfmt-elf-section.html

Things to Remember

- All binaries we produce were written in C with *fairly* standard operations -- sometimes it is efficient to work backwards from fair assumptions about the program and coming back to these assumptions if you get stuck
- C creates to a layer of abstraction, especially with regards to memory
  - 4 bytes could be a pointer, or an integer, a 4 byte array, etc. And they can be treated interchangeably at times
  - Some RE tools such as Binary Ninja make it very easy to convert representation of constants between characters, decimal, hex, etc
- I skimmed over a lot of stuff -- more in-depth ELF files, fuzzing, etc. Maybe more in Binaries II

homework #5

will be posted soon.

Let us know if you have any questions!

It is due by 10/11 at 11:59PM.