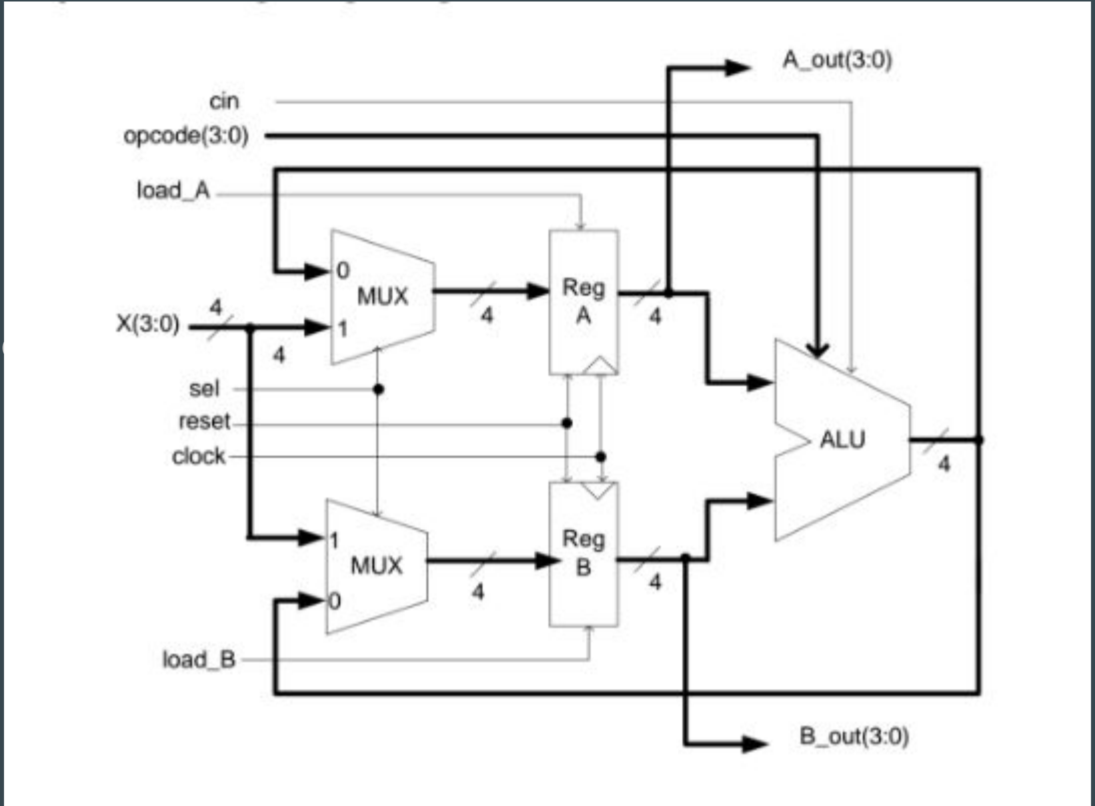
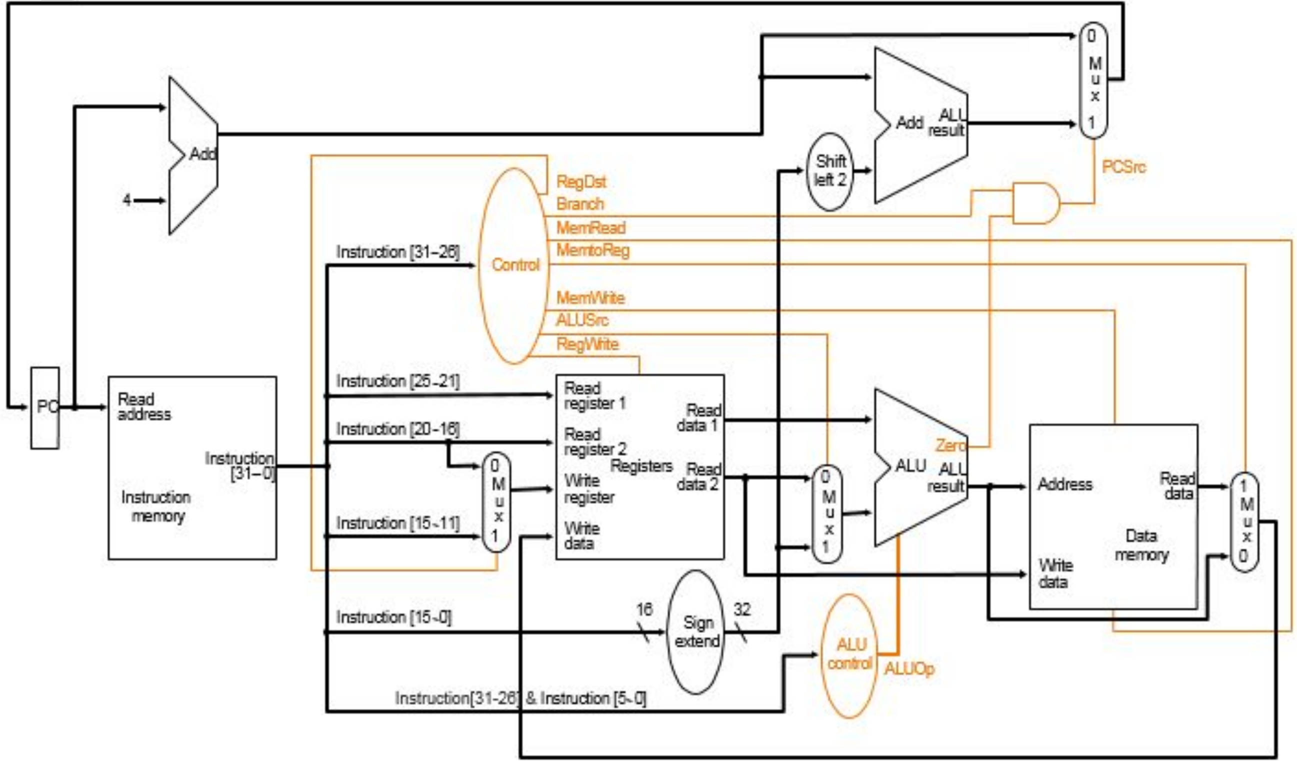
A Buffer Overflow primer

Computer Architecture 101

- Arithmetic Logic Unit(ALU):
  - The building block of a CPU
  - A calculator
- Memory: registers, cache, ram
- A bunch of control lines tell th
  processors what to do



Single Cycle CPU

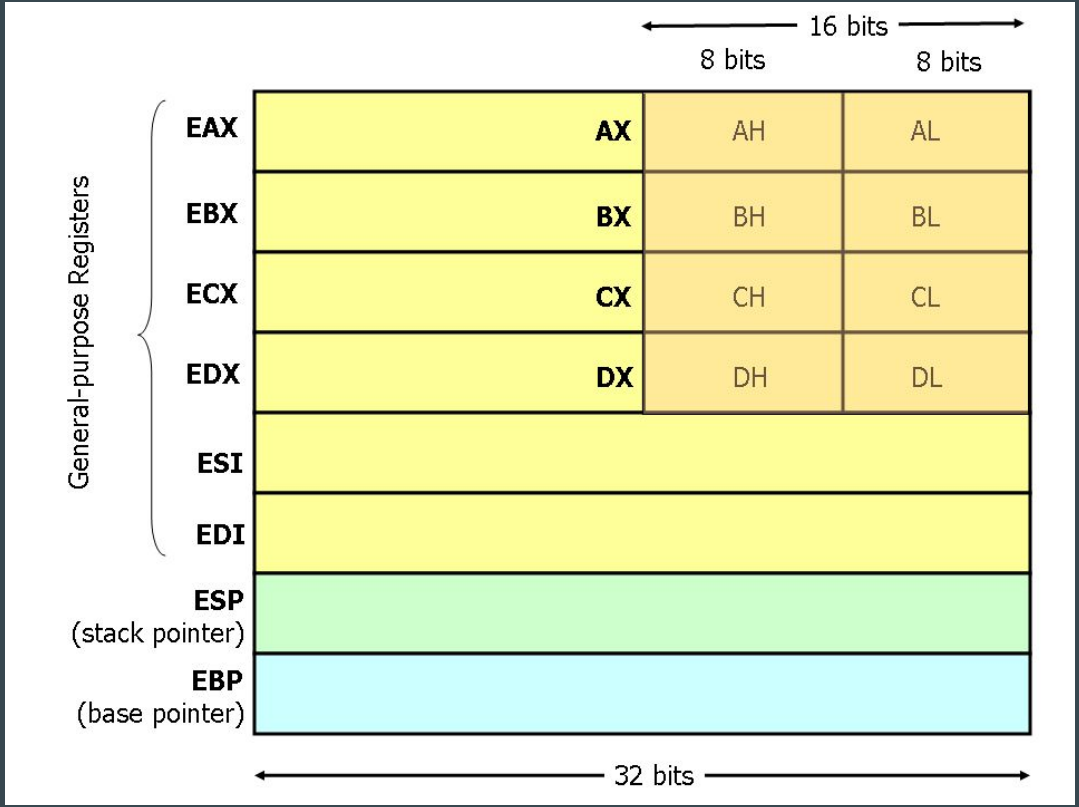


How do we get a computer to do what we want?

- By managing all of those control lines!
- That would be really tedious to program though
- So we made a language for speaking to the computers: assembly
- Assembly is human readable form of the opcodes that control the processor
- It is architecture dependent (x86, mips, x86_64, Powerpc, ARM...etc.)

A look at x86

- x86 is the common architecture for desktops and servers
- 32bit vs 64bit? What does that mean?
- Instruction Pointer
  - EIP which points to the next instruction to be executed.



A look at x86 (cont) - Some basic instructions

Assembly	Rough C equivalent
mov eax, ebx	eax = ebx
add eax, ebx	eax = eax + eax
sub eax, ebx	eax = eax-ebx
inc eax	++eax
dec eax	--eax
call foo()	foo()
ret	return eax

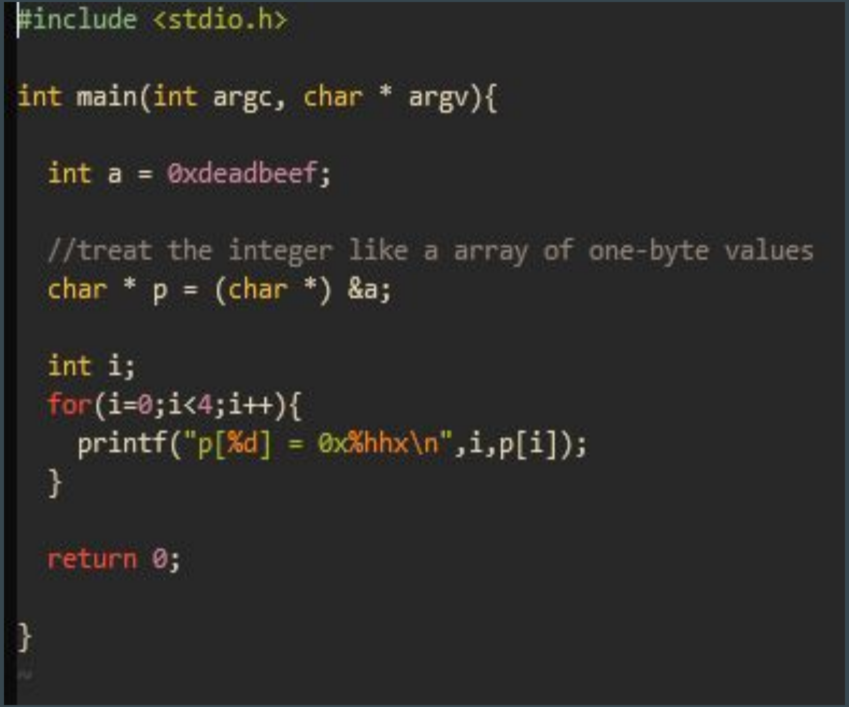
A look at x86 (cont) - Branching

- Branching: how to manipulate control flow
- jmp instruction is unconditional
- Conditional branching - make a comparison then jmp
  - Ex:
    - cmp eax, ebx
    - je
  - Will jmp is eax == ebx
- Jump can have a lot of forms: jnz, jz, je, jne, jg, etc.

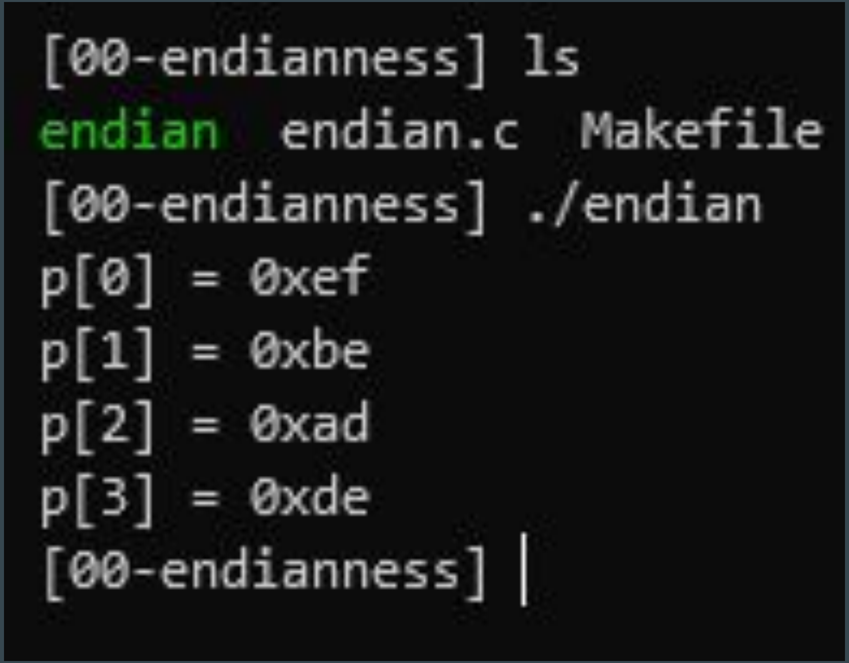
Endianness

- There are two major ways of reading data, left to right, or right to left
- Ex: 210500
  - Either 210,500 or
  - 5,012
- Where the most-significant byte is represented is referred to as endian-ness in computer science. By far the most prevalent representation is little endian, by which the least significant bytes come first.

Endianness Example



```c
#include <stdio.h>

int main(int argc, char * argv){

  int a = 0xdeadbeef;

  //treat the integer like a array of one-byte values
  char * p = (char *) &a;

  int i;
  for(i=0;i<4;i++){
    printf("p[%d] = 0x%hhx\n",i,p[i]);
  }

  return 0;

}
```



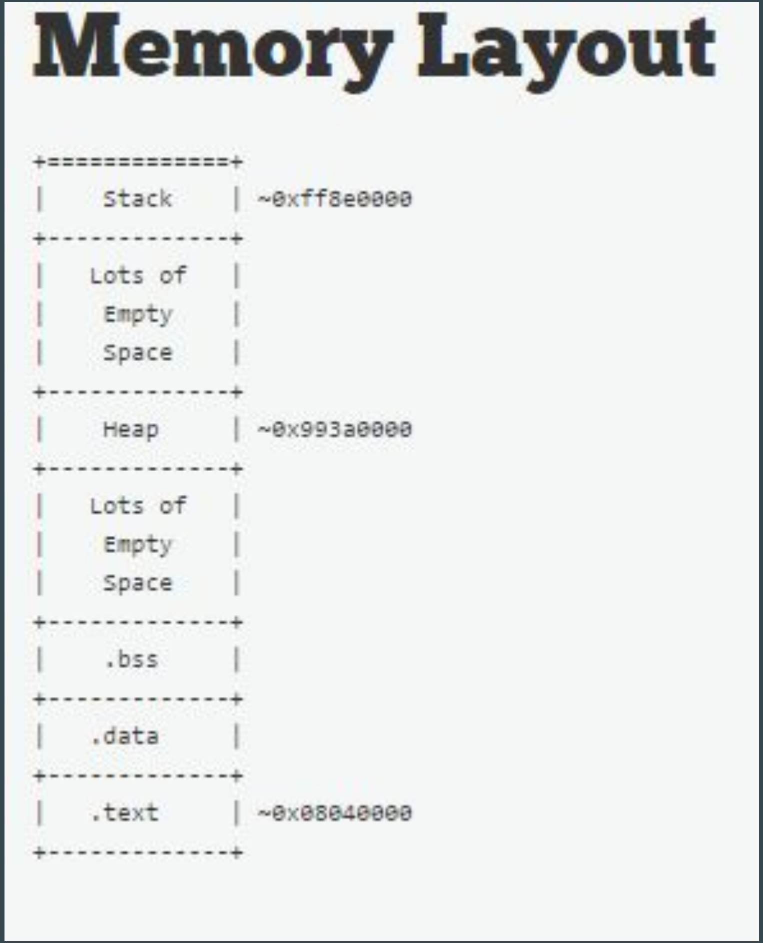
```
[00-endianness] ls
endian  endian.c  Makefile
[00-endianness] ./endian
p[0] = 0xef
p[1] = 0xbe
p[2] = 0xad
p[3] = 0xde
[00-endianness] 
```

A look at x86 (cont) - Memory addressing

- Memory references are surrounded by brackets.
  - Ex: [esp] is equivalent *esp
  - [esp] means the value at the address contained in esp.
- They can contain arithmetic
  - Ex: [ebp-0x4]
- In disassembly you will see data sizes associated with the PTRs
  - Ex: BYTE PTR [ebp] means the byte in memory at the address contained in ebp
  - DWORD PTR [ebp] means the 32bit word in memory at the address contained in ebp.

A Look at x86 (cont.)

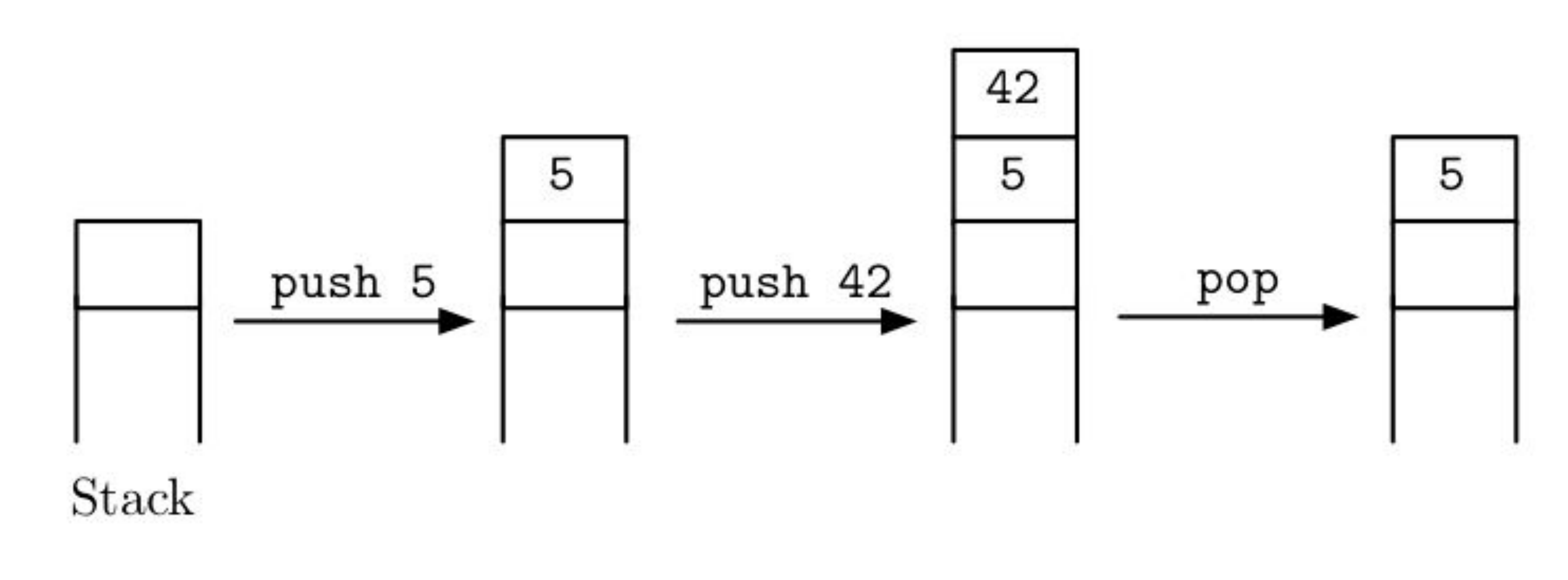
- Parts of a process:
  - .data : initialized data (int i = 4;)
  - .bss   : uninitialized data (int i;) set to zero
  - .text
    - code
    - Entry point (_start, main)
  - The stack
    - Local variables
  - The Heap
    - Dynamically allocated memory (malloc/new)
  - There are a lot more segments than this but these are the main ones



Memory Layout

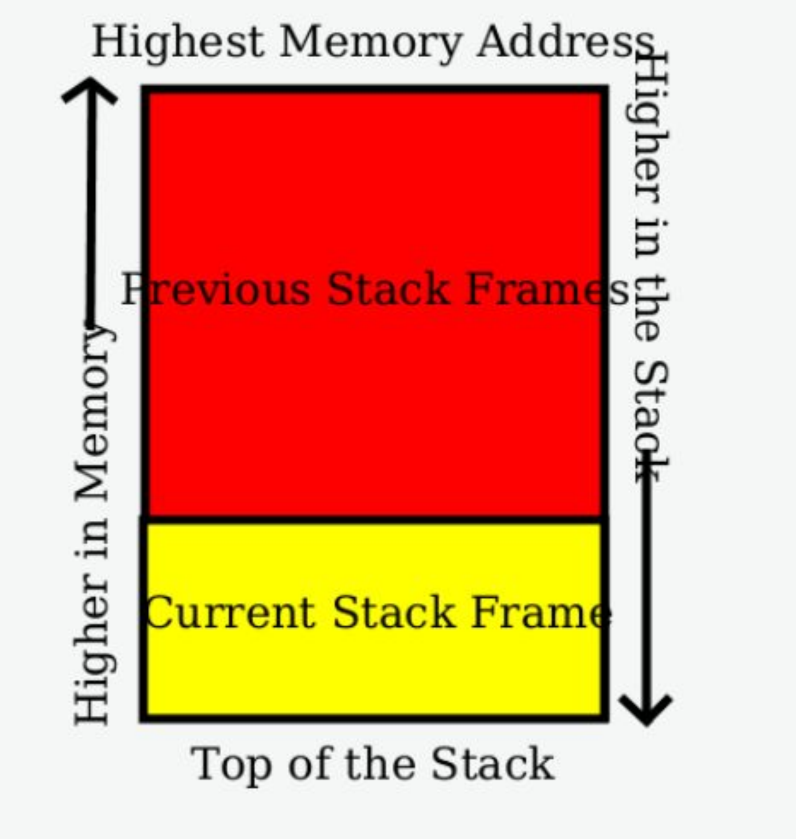
```
+=============+
|    Stack    | ~0xff8e0000
+-------------+
|   Lots of   |
|    Empty    |
|    Space    |
+-------------+
|    Heap     | ~0x993a0000
+-------------+
|   Lots of   |
|    Empty    |
|    Space    |
+-------------+
|    .bss     |
+-------------+
|   .data     |
+-------------+
|   .text     | ~0x08040000
+-------------+
```

The stack

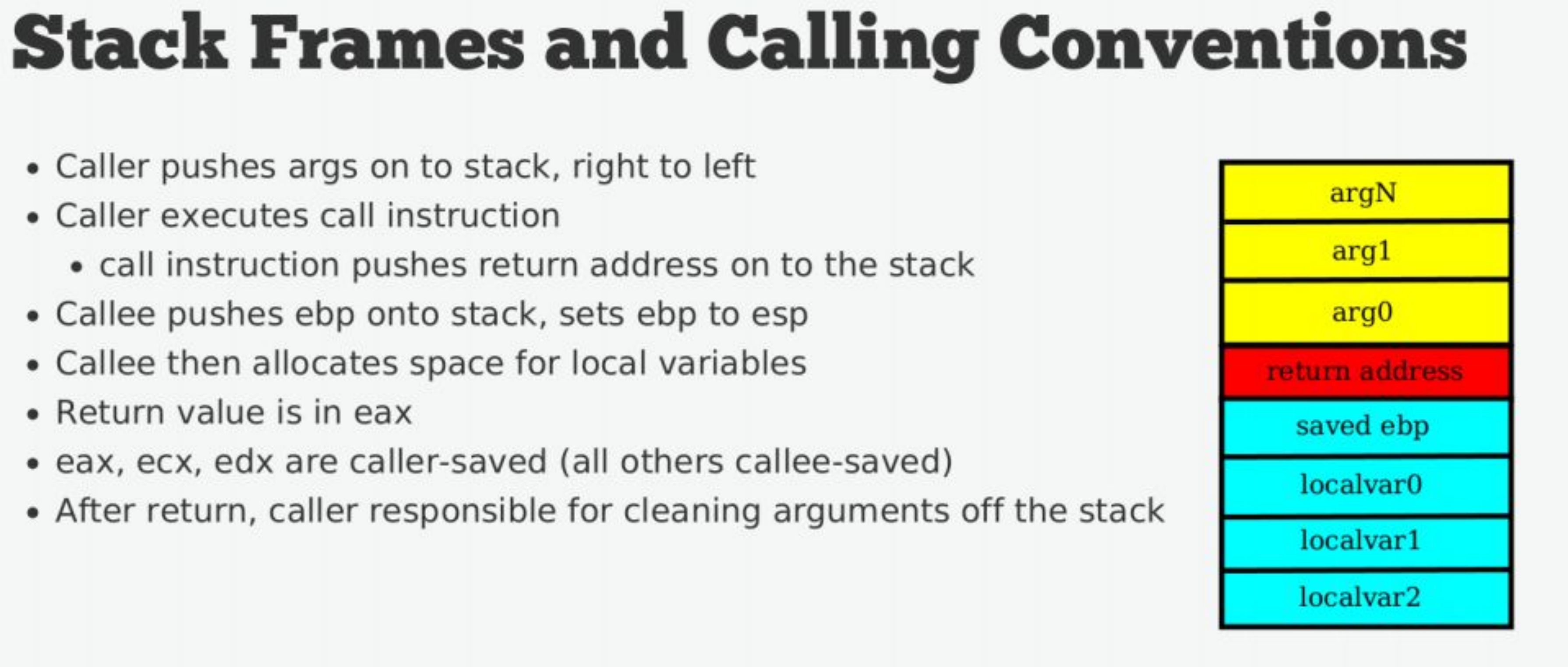


The stack (cont.)

- The stack is used to hold local variables for each function.
- Each function creates a stack frame
  - EBP (the base pointer) points to the bottom of the stack frame
  - ESP (the stack pointer) points to the top of the stack.



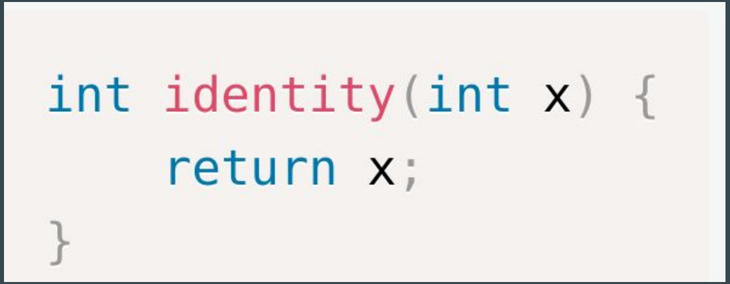
32 Bit Calling convention



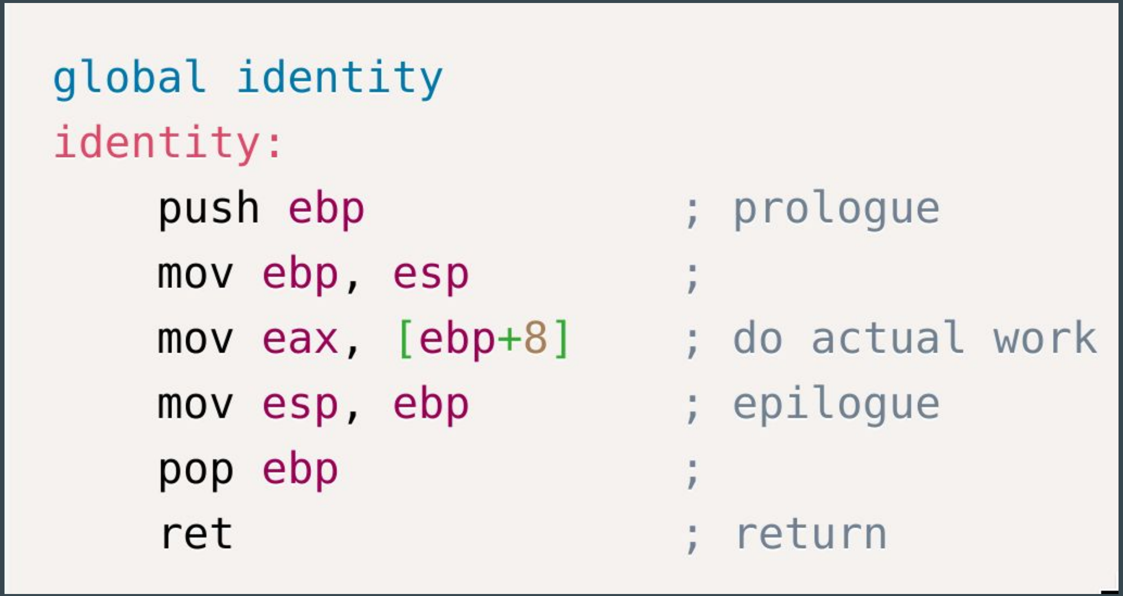
Stack Frames and Calling Conventions

- Caller pushes args on to stack, right to left
- Caller executes call instruction
  - call instruction pushes return address on to the stack
- Callee pushes ebp onto stack, sets ebp to esp
- Callee then allocates space for local variables
- Return value is in eax
- eax, ecx, edx are caller-saved (all others callee-saved)
- After return, caller responsible for cleaning arguments off the stack

Function example



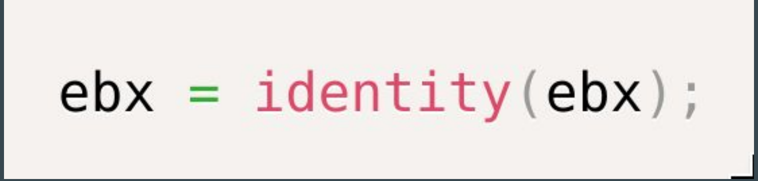
```c
int identity(int x) {
    return x;
}
```



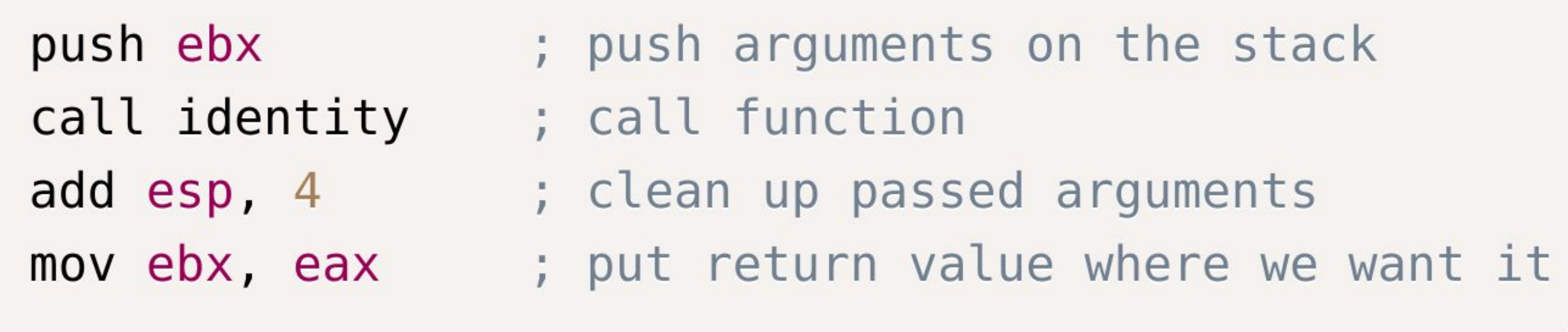
```nasm
global identity
identity:
    push ebp                ; prologue
    mov ebp, esp            ;
    mov eax, [ebp+8]        ; do actual work
    mov esp, ebp            ; epilogue
    pop ebp                 ;
    ret                     ; return
```

- [ebp+8] will grab the argument to the function
    - Return address is at ebp+4

Function call example



```
ebx = identity(ebx);
```



```
push ebx            ; push arguments on the stack
call identity       ; call function
add esp, 4          ; clean up passed arguments
mov ebx, eax        ; put return value where we want it
```