x86 Assembly

Adapted from CS 161 Spring 2022 - Lecture 3

Next: x86 Assembly and Call Stack

- Part CS 61C review
  - How do computers represent numbers as bits and bytes?
  - How do computers interpret and run the programs we write?
  - How do computers organize segments of memory?
- Part new content
  - How does x86 assembly work?
  - How do you call a function in x86?

Number Representation

Textbook Chapter 2.1

Units of Measurement

- In computers, all data is represented as bits
  - **Bit**: a binary digit, 0 or 1
- Names for groups of bits
  - 4 bits = 1 **nibble**
  - 8 bits = 1 **byte**
- How many bits/nibbles/bytes in `0b 1000 1000 1000 1000`?
  - 16 bits, or 4 nibbles, or 2 bytes

Hexadecimal

- 4 bits can be represented as 1 hexadecimal digit (base 16)
- How would you write **0b**11000110 in hex?
  - **0x**C6
  - Note: For clarity, we add **0b** in front of bits and **0x** in front of hex

Binary	Hexadecimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7

Binary	Hexadecimal
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Running C Programs

Textbook Chapter 2.2

CALL (Compiler, Assembler, Linker, Loader)


```
int add_one(int a) {
    int added = a + 1;
    return added;
}
```

C code

Compiler →

```
add_one:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp
    mov 8(%ebp), %eax
    mov %eax, -4(%ebp)
    inc -4(%ebp)
    mov -4(%ebp), %eax
    leave
    ret
```

Assembly code
(RISC-V, x86)

Assembler →

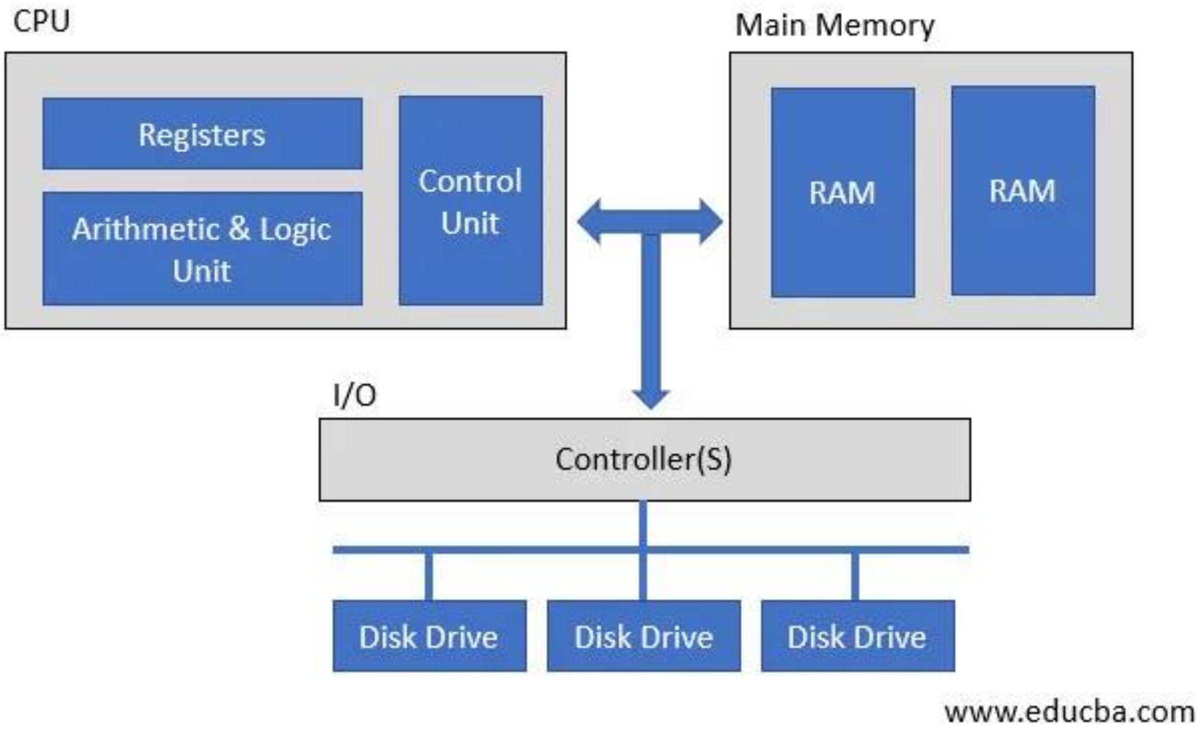
```
0x55 0x89 0xe5 0x83
0xec 0x04 0x8b 0x45
0x08 0x89 0x45 0xfc
0x45 0x89 0xe8 0xc9
0xc3
```

Machine code
(raw bits)

CALL (Compiler, Assembler, Linker, Loader)

- Compiler: Converts C code into assembly code (RISC-V, x86)
- Assembler: Converts assembly code into machine code (raw bits)
  - Think 61C's RISC-V "green sheet"
- Linker: Deals with dependencies and libraries
  - You can ignore this part for 161
- Loader: Sets up memory space and jumps into the machine code
  - Sometimes, there is an additional linking step right before loading, called dynamic loading
    - Result: The executable doesn't need to contain all the dependencies
    - This also provides additional mitigations, as we'll see later
- After these steps, execution begins, and C runtime library calls **`main`**!

CPU, Memory, and Registers

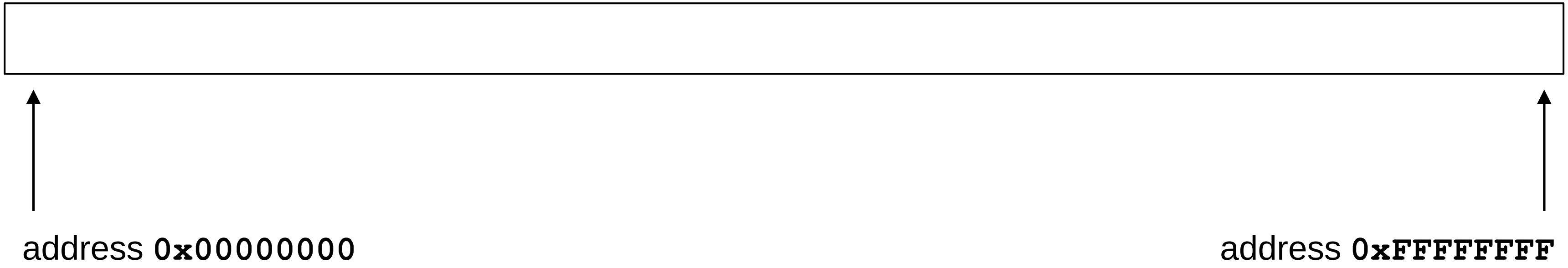


Memory Layout

Textbook Chapter 2.3 & 2.5

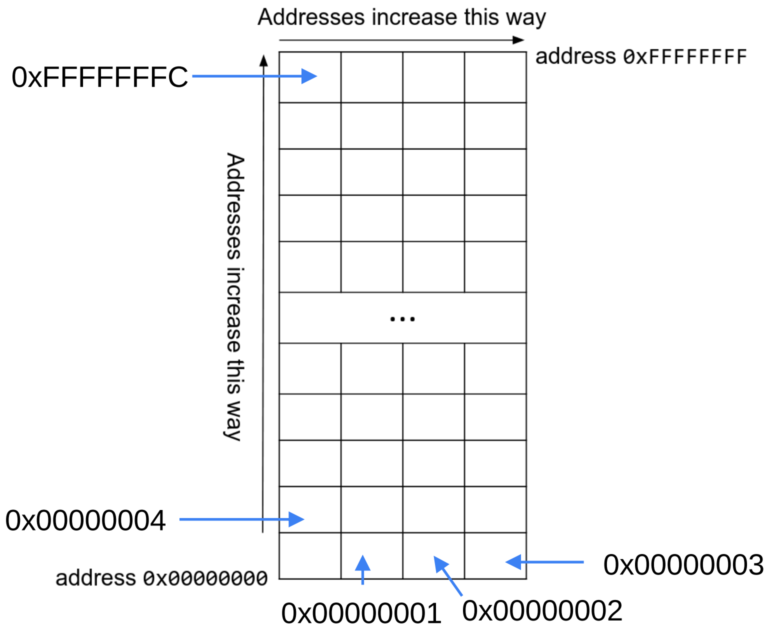
C Memory Layout

- At runtime, the loader tells your OS to give your program a big blob of memory
- On a 32-bit system, the memory has 32-bit addresses
  - On a 64-bit system, memory has 64-bit addresses
  - We use 32-bit systems in this class
- Each address refers to one byte, which means you have 2^{32} bytes of memory



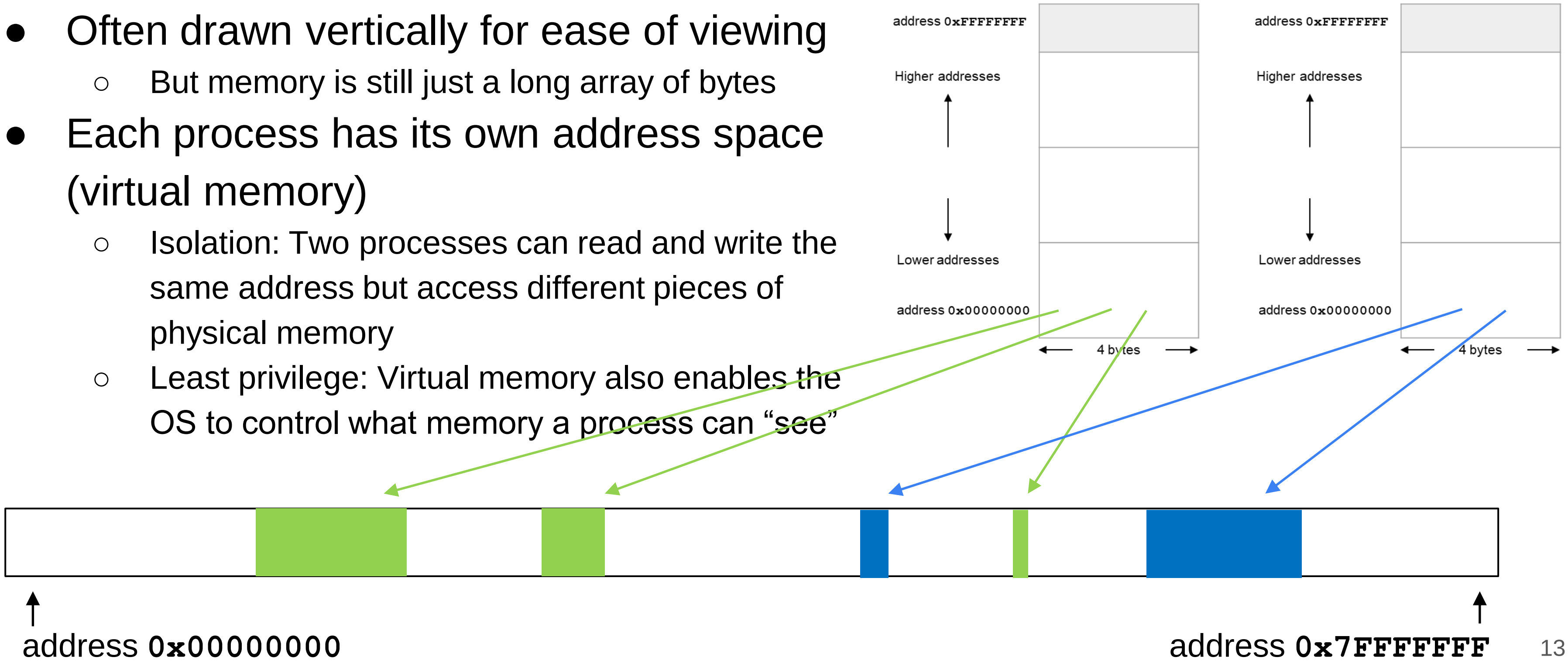
C Memory Layout

- Often drawn vertically for ease of viewing
    - But memory is still just a long array of bytes



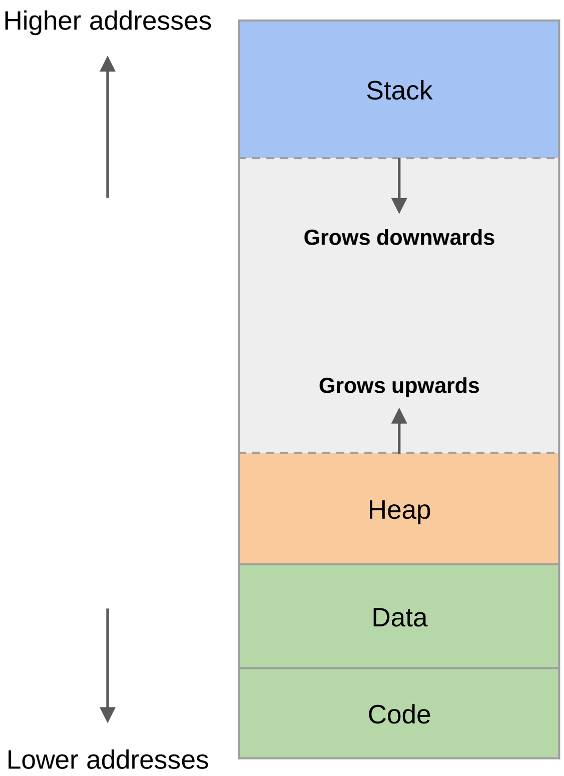
C Memory Layout

- Often drawn vertically for ease of viewing
  - But memory is still just a long array of bytes
- Each process has its own address space (virtual memory)
  - Isolation: Two processes can read and write the same address but access different pieces of physical memory
  - Least privilege: Virtual memory also enables the OS to control what memory a process can “see”



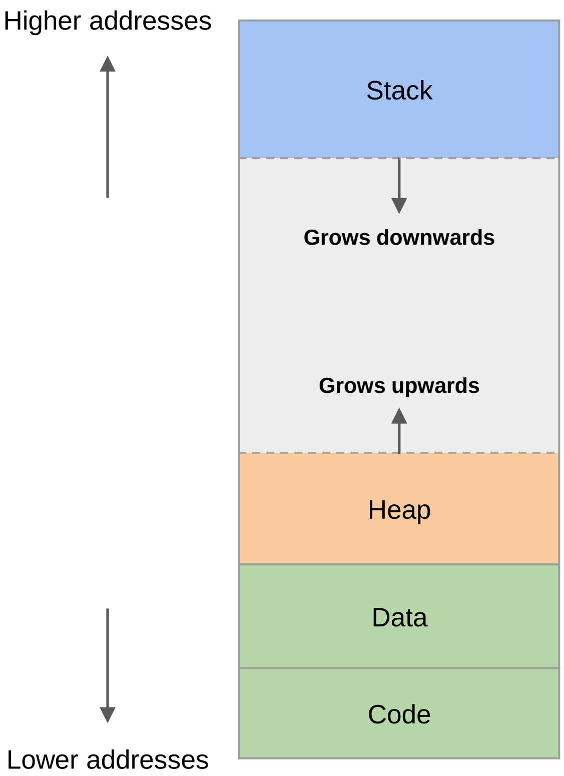
x86 Memory Layout

- Code
  - The program code itself (also called "text")
- Data
  - Static variables, allocated when the program is started
- Heap
  - Dynamically allocated memory using **`malloc`** and **`free`**
  - As more and more memory is allocated, it grows *upwards*
- Stack:
  - Local variables and stack frames
  - As you make deeper and deeper function calls, it grows *downwards*



Registers

- Recall registers from CS 61C
  - Examples of RISC-V registers: **`a0`**, **`t0`**, **`ra`**, **`sp`**
- Registers are located on the CPU
  - This is different from the memory layout
  - Memory: addresses are 32-bit numbers
  - Registers: addresses are names (**`ebp`**, **`esp`**, **`eip`**)



x86 Architecture

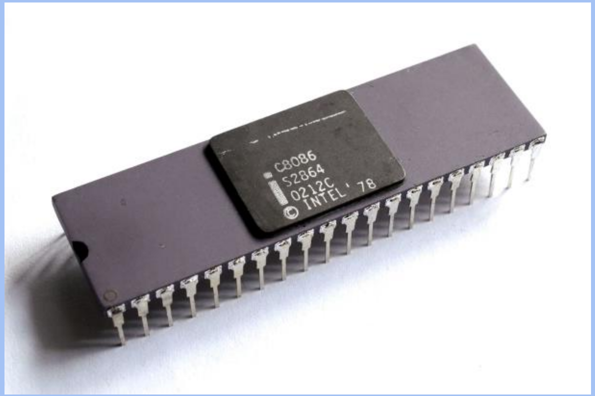
Textbook Chapter 2.4 & 2.7

Why x86?

- It’s the most commonly used instruction set architecture in consumer computers!
  - You are probably using an x86 computer right now… unless you’re on a phone, tablet, or M1 Mac
- You only need enough to be able to read it and know what is going on
  - We will make comparisons to RISC-V, but it’s okay if you haven’t taken 61C and don’t know RISC-V; you don’t need to understand the comparisons to understand x86
- However, if you have a choice, choose ARM
  - Significantly higher performance for the same cost
  - 64b ARM has some unique security features we will discuss later

What is x86?

- Complex instruction set computer (CISC) architecture
  - The opposite of reduced instruction set computer (RISC) architecture
  - There are a lot of instructions… The full ISA manual is over **5,000 pages long**!
  - We will not be teaching the full instruction set (obviously), just enough of the calling convention to be able to do Project 1
- Launched in 1978 with the Intel 8086 processor and eventually took over much of computing
  - Over 40 years ago! The ISA is very bloated because of this…
- 64-bit variant x86-64 launched in 1999 (but we won’t study it)



x86 Fact Sheet

- Little-endian
  - The least-significant byte of multi-byte numbers is placed at the first/lowest memory address
  - Same as RISC-V

storing the word 0x44332211 in memory:

0x11	0x22	0x33	0x44
address 0x00000000	address 0x00000001	address 0x00000002	address 0x00000003

```
int main(void) {
    uint32_t num = 0xdeadbeef;

    // This prints "deadbeef".
    printf("%x", num);

    // This prints "ef be ad de".
    uint8_t *bytes = (uint8_t *) &num;
    for (size_t i = 0; i < 4; i++) {
        printf("%x ", bytes[i]);
    }
}
```

0xef	0xbe	0xad	0xde

#include <stdint.h>
uint8_t （1 byte)
uint32_t （4 bytes)

x86 Registers

- Storage units as part of the CPU architecture (not part of memory)
- Only 8 main general-purpose registers:
  - EAX, EBX, ECX, EDX, ESI, EDI: General-purpose
  - ESP: Stack pointer (similar to **`sp`** in RISC-V)
  - EBP: Base pointer (similar to **`fp`** in RISC-V)
  - We will discuss ESP and EBP in more detail later
- Instruction pointer register: EIP
  - Similar to PC in RISC-V

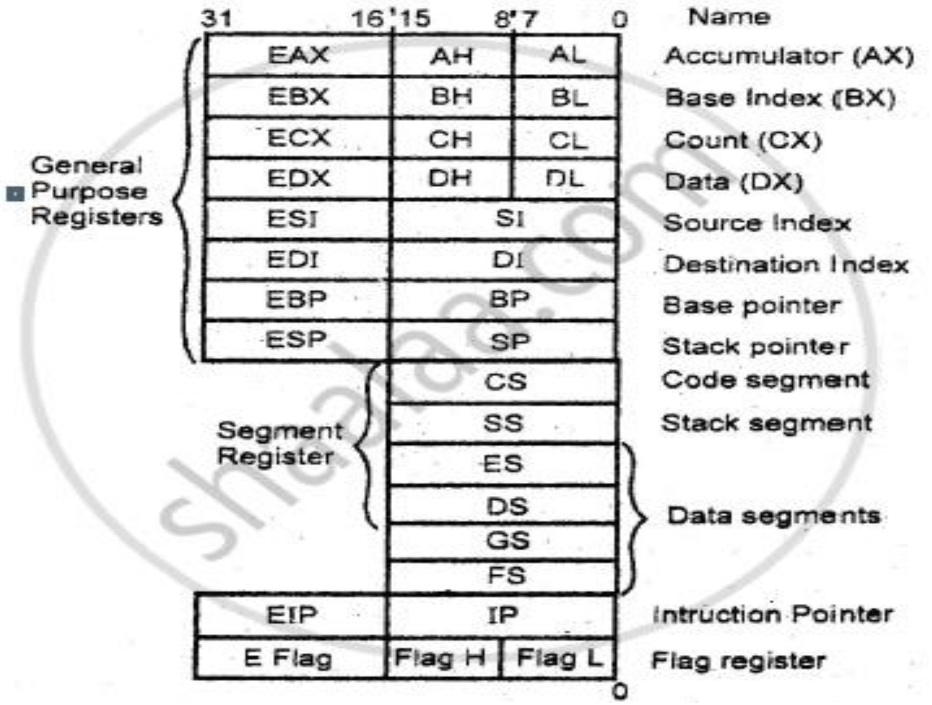
The main x86 registers…

- General purpose: EAX-EDX (E stands for Extended)
  - What you use for computing and other stuff, sorta…
- Indexes & Pointers
  - EBP: “Frame pointer”: points to the top/start of the current call frame on the stack
  - ESP: “Stack pointer”: points to the current stack
  - (Remember, stack grows down!)
    - PUSH and POP
      - Decrement the stack pointer and store something there
      - Load something and increment the stack pointer
    - Most operations are done with data on the stack…

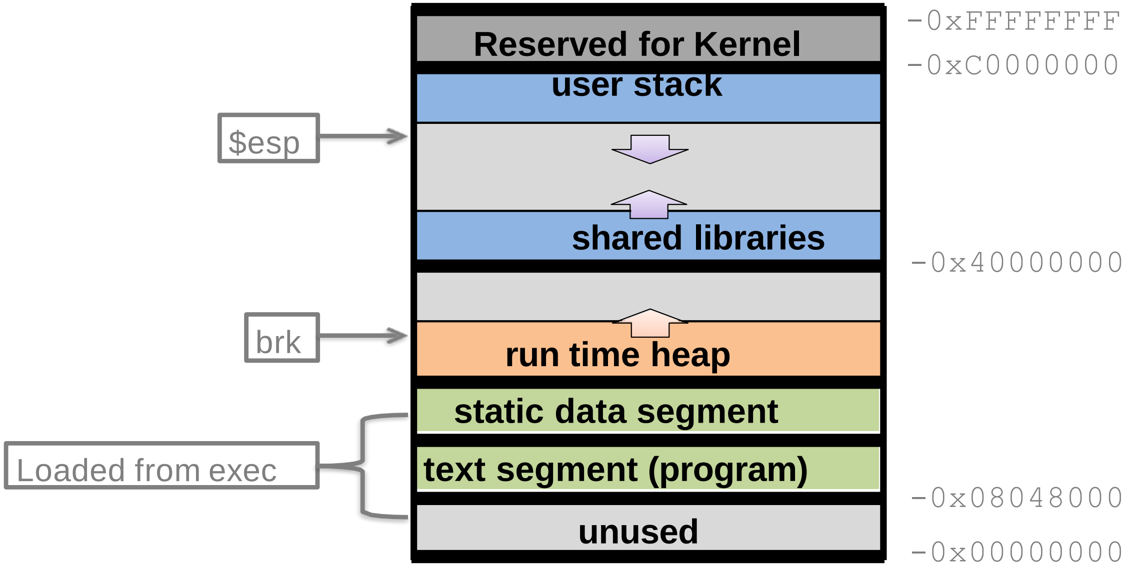
https://www.cs.rutgers.edu/~pxk/419/notes/frames.html

The main x86 registers…

- General purpose: EAX-EDX
  - What you use for computing and other stuff
- Indexes & Pointers
  - EBP: “Frame pointer”: points to the top/sta
    ESP: “Stack pointer”: points to the current
  - (Remember, stack grows down!)
    - PUSH and POP
      - Decrement the stack pointer and store something the
      - Load something and increment the stack pointer
    - Most operations are done with data on the stac



Linux (32-bit) process memory layout



x86 Syntax

- Register references are preceded with a percent sign **`%`**
  - Example: **`%eax`**, **`%esp`**, **`%edi`**
- Immediates are preceded with a dollar sign **`$`**
  - Example: **`$1`**, **`$161`**, **`$0x4`**
- Memory references use parentheses and can have immediate offsets
  - Example: **`8(%esp)`** dereferences memory 8 bytes above the address contained in ESP

x86 Assembly

- Instructions are composed of an opcode and zero or more operands.
- **`add $0x8, %ebx`**

Opcode | Source | Destination

- Pseudocode: **`EBX = EBX + 0x8`**
- The destination comes last
  - Contrast with RISC-V assembly, where the destination (RD) is first
- The **`add`** instruction only has two operands; and the destination is an input
  - Contrast with RISC-V, where the two source operands are separate (RS1 and RS2)
- This instruction uses a register and an immediate

x86 Assembly

- **`xor 4(%esi), %eax`**

Opcode | Source | Destination

- Pseudocode: **`EAX = EAX ^ *(ESI + 4)`**
- This is a memory reference, where the value at 4 bytes above the address in ESI is dereferenced, XOR’d with EAX, and stored back into EAX
  - Most instructions can be register-register, register-immediate, register-memory, or memory-immediate (but not memory-memory)
    - How can you achieve a memory-memory operation?

Stack Layout

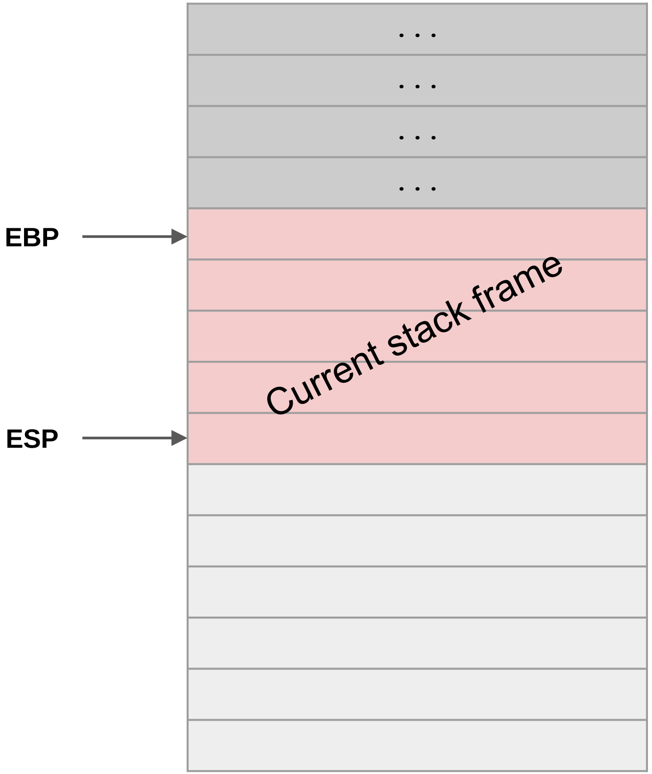
Textbook Chapter 2.6

Stack Frames

- When your code calls a function, space is made on the stack for local variables
  - This space is known as the **stack frame** for the function
  - The stack frame goes away once the function returns
- The stack starts at higher addresses. Every time your code calls a function, the stack makes extra space by growing down
  - Note: Data on the stack, such as a string, is still stored from lowest address to highest address. “Growing down” only happens when extra memory needs to be allocated.

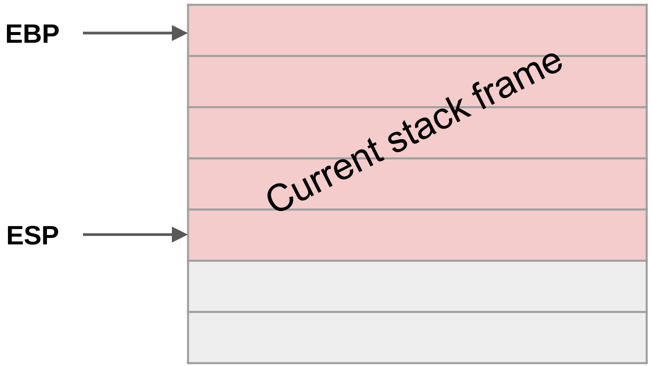
Stack Frames

- To keep track of the current stack frame, we store two pointers in registers
  - The EBP (base pointer) register points to the top of the current stack frame
    - Equivalent to RISC-V **`fp`**
  - The ESP (stack pointer) register points to the bottom of the current stack frame
    - Equivalent to RISC-V **`sp`** (but x86 moves the stack pointer up and down a lot more than RISC-V does)



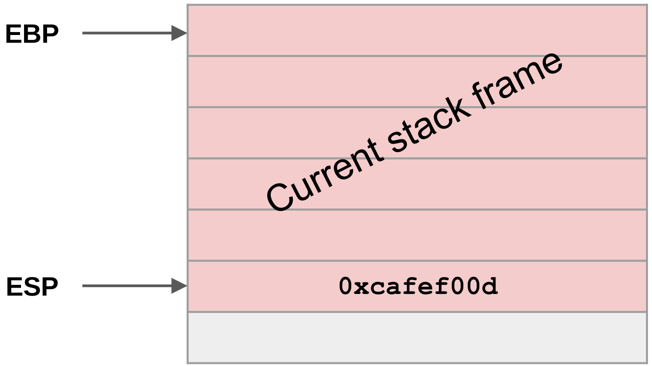
Pushing and Popping

- The **`push`** instruction adds an element to the stack
  - Decrement ESP to allocate more memory on the stack
  - Save the new value on the lowest value of the stack



EAX = `0xcafef00d`
EBX = ...

Before **`push %eax`**

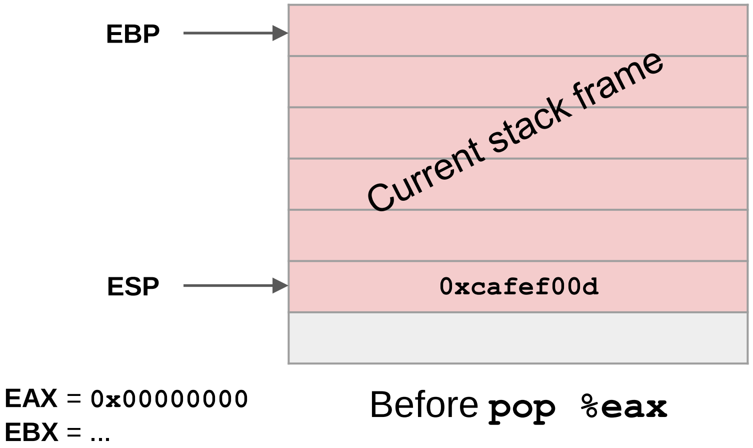


EAX = `0xcafef00d`
EBX = ...

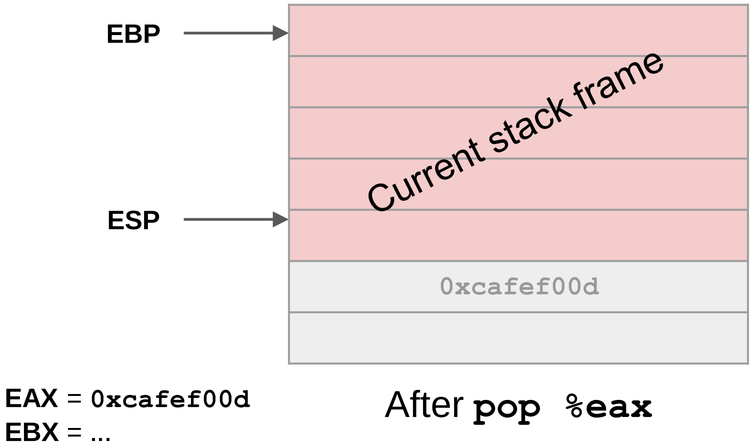
After **`push %eax`**

Pushing and Popping

- The **pop** instruction removes an element from the stack
  - Load the value from the lowest value on the stack and store it in a register
  - Increment ESP to deallocate the memory on the stack



Before **pop %eax**



After **pop %eax**

Why **push** and **pop**?

- This allows x86 to use “stack-machine”-style logic
  - Operations always operate on the top items on the stack and push the result back onto the stack
  - Example: *a + b * c + d*
    - `PUSH a // a`
    - `PUSH b // a, b`
    - `PUSH c // a, b, c`
    - `MUL    // a, (b*c)`
    - `ADD    // (a+(b*c))`
    - `PUSH   // (a+(b*c)), d`
    - `ADD    // (a+(b*c))+d`
    - Result is the last remaining item on the stack!

First convert the expression to postfix notation, than apply stack operations. https://www.geeksforgeeks.org/arithmetic-expression-evalution/

- Used to think this allowed for smaller code and simpler code generation
  - Reality? Not so much. RISC logic is equally efficient in code size in practice, especially with the compressed instructions for RISC-V.
  - But this misconception is why the Java VM is a stack machine too…
- (Not really a) Takeaway: Take 164 if you want to learn more about this

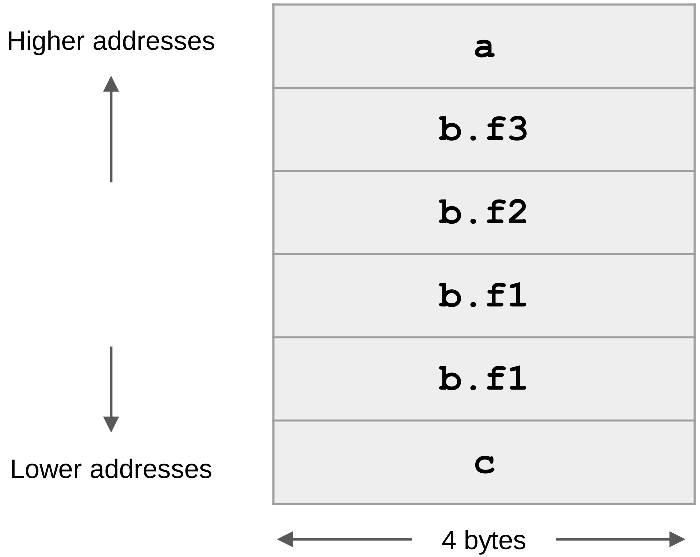
x86 Stack Layout

- Local variables are always allocated on the stack
  - Contrast with RISC-V, which has plenty of registers that can be used for variables
- Individual variables within a stack frame are stored with the first variable at the *highest* address
- Members of a struct are stored with the first member at the *lowest* address
- Global variables (not on the stack) are stored with the first variable at the *lowest* address

Stack Layout


```
struct foo {
    long long f1; // 8 bytes
    int f2;       // 4 bytes
    int f3;       // 4 bytes
};

void func(void) {
    int a;        // 4 bytes
    struct foo b;
    int c;        // 4 bytes
}
```



How would you fill out the boxes in this stack diagram?
Options:
`a` `b.f1` `b.f2` `b.f3` `c`