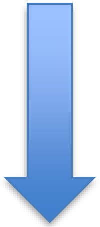
X86 Assembly Review

University of Illinois

ECE 422/CS 461

Review: Assembly

High-level programming language
(e.g., C, C++)



Compiler

Machine code (bit strings)

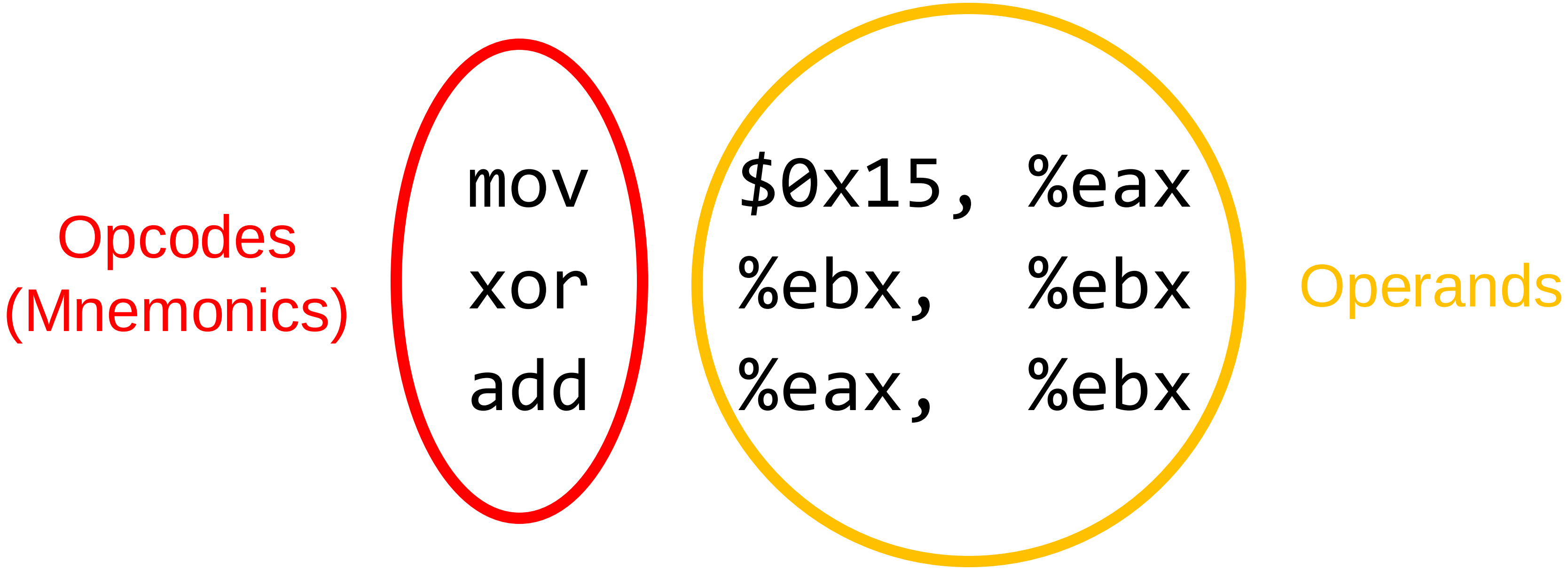
Assembly: a human-readable
encoding of machine code

Review: x86 Assembly

```
Machine code: b8 0f 00 00 00 31 db 01 c3
```

```
mov    $0x15, %eax
xor    %ebx,  %ebx
add    %eax,  %ebx
```

Review: x86 Assembly



Opcodes
(Mnemonics)

```
mov    $0x15, %eax
xor    %ebx,  %ebx
add    %eax,  %ebx
```

Operands

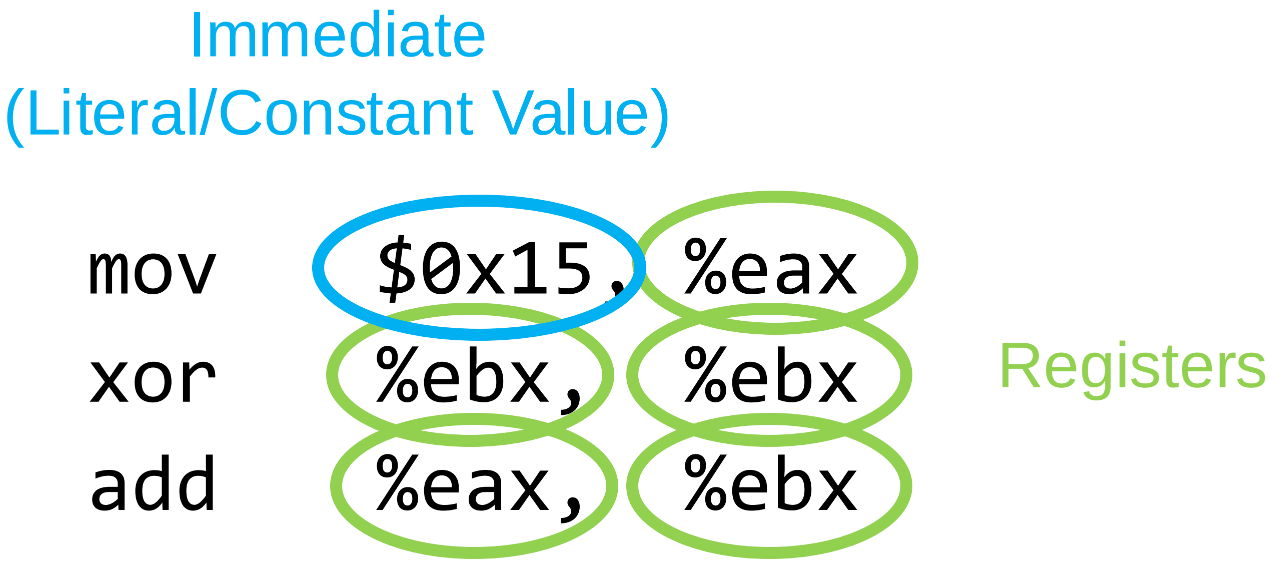
Review: x86 Assembly

Immediate
(Literal/Constant Value)

```
mov    $0x15, %eax
xor    %ebx,  %ebx
add    %eax,  %ebx
```

Registers

Review: x86 Assembly



Immediate
(Literal/Constant Value)

```
mov    $0x15, %eax
xor    %ebx,  %ebx
add    %eax,  %ebx
```

Registers

Also, memory addresses (more on these in a moment)

Commonly Used x86 Registers

General purpose registers

- EAX - Return value
- EBX
- ECX - Loop counter
- EDX
- EDI - Repeated destination
- ESI - Repeated source

Special Registers

- EBP – Frame pointer/Base pointer
- ESP - Stack pointer
- EIP - Program counter
- EFLAGS - Status of previous operations (used in conditionals)

x86 Assembly Syntax: Intel vs. AT&T

There are two main variants of x86 syntax.
Suppose we want to move a value from EBX to EAX.

Intel

- `mov eax, ebx`
- Destination operand first
- % for registers optional
- ……

AT&T (GAS)

- `mov %ebx, %eax`
- Destination operand last
- % for registers required
- ……

In this course, we use AT&T (GAS) syntax exclusively.

x86 Assembly Syntax: Quick Quiz

What does `add %eax, %ebx` do?

Intel

- `mov eax, ebx`
- Destination operand first
- % for registers optional
- ……

AT&T (GAS)

- `mov %ebx, %eax`
- Destination operand last
- % for registers required
- ……

In this course, we use AT&T (GAS) syntax exclusively.

Memory Operations

- What if we want to use a value from memory, rather than a register or constant value?

Example: `Load Mem[%ebp + (8 * %ecx) + 4] into %eax`

- x86 Assembly syntax

```
mov 4(%ebp,%ecx,8), %eax
```

GAS/AT&T Memory Address Calculation

Write it in assembly:

```
displacement (base_reg, offset_reg, multiplier)
```

Calculate it:

```
base_reg + (offset_reg*multiplier) + displacement
```

```
mov   8(%ebp), %eax           # Mem[EBP+8] to eax
mov   %eax, 12(,%edx,4)       # eax to Mem[EDX*4+12]
```

Notice that not all fields are required!

GAS/AT&T Memory Syntax Example

```
typedef struct {
  int a, b, c, d;
} foo_t;
foo_t my_foos[10];

my_foos[5].c = 461;
```

GAS/AT&T Memory Syntax Example

```
typedef struct {
  int a, b, c, d;
} foo_t;
foo_t my_foos[10];

my_foos[5].c = 461;
```

```
# Assume %ebx points to my_foos
mov $5, %ecx
movl $461, 8(%ebx, %ecx, 16)
```

Common x86 Instructions (1)

Arithmetic Operations

- `add`, `sub` - add/subtract first operand to/from second
- `inc`, `dec` - increment/decrement operand
- `neg` - change sign of operand

Logical Operations

- `and`, `or`, `xor` - bitwise and/or/xor
- `not` - flip all of the bit values
- `shl`, `shr` - shift bits left/right

Common x86 Instructions (2)

Data Transfer Instructions

- `mov` - copy data from first operand to second operand
- `lea` - compute address and store it in second operand (does NOT access memory)

- `push` - push the operand onto the stack
- `pop` - pop the value at the top of the stack into the operand (more on stack push/pop later)

Common x86 Instructions (3)

Control Flow Instructions

- `jmp` – jump to label or address specified by operand
- `je` - jump if equal
- `jne` - jump if not equal
- `jz` - jump if zero
- `jl/jg` - jump if less than / greater than
- `jle/jge` - jump if equal to or less than / greater than

For conditional jumps, the EFLAGS register is used. EFLAGS is set by CMP, TEST, and arithmetic/logical instructions

32-bit x86 ISA

- 1 byte = 8 bits
- char: 1 byte
- Integer: 4 bytes
- Memory address width: 4 bytes
- Pointer: 4 bytes
- Registers: 4 bytes

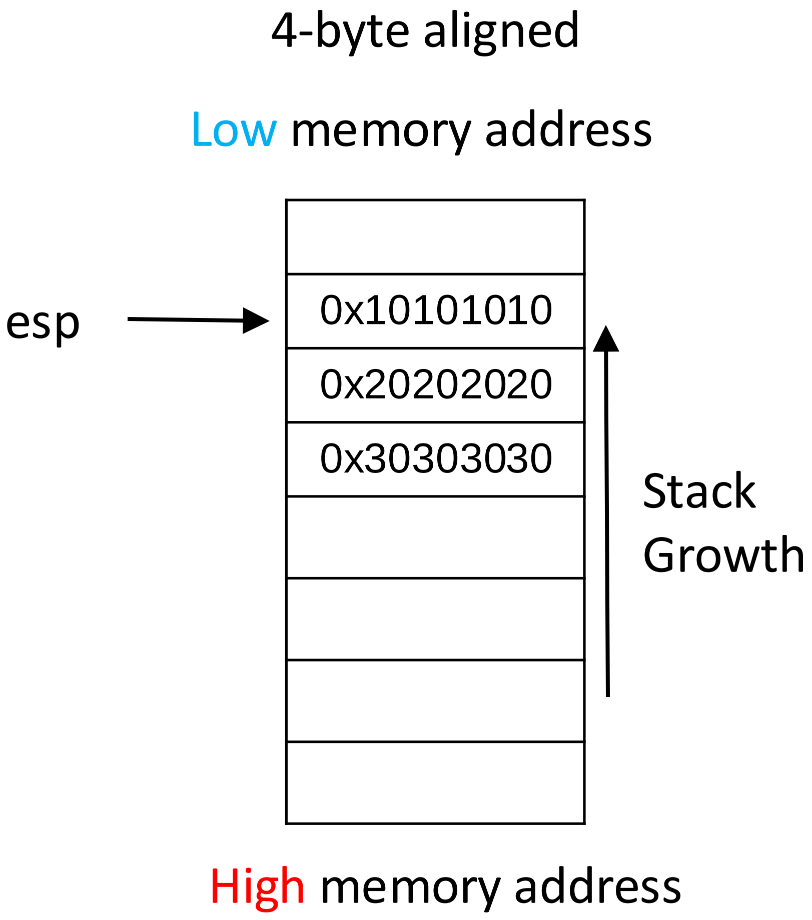
- **Each memory location: 1 byte**

The Stack

- Stores working data (local variables, function arguments, return addresses, etc.)
- Last-in First-out (LIFO) structure
- Grows towards lower memory addresses (upwards)
- Manipulated with `push` and `pop` instructions

The Stack

- ESP (stack pointer) points to the top of the stack
- push instruction subtracts 4 from ESP and then writes to the top of the stack



4-byte aligned

Low memory address

esp

0x10101010

0x20202020

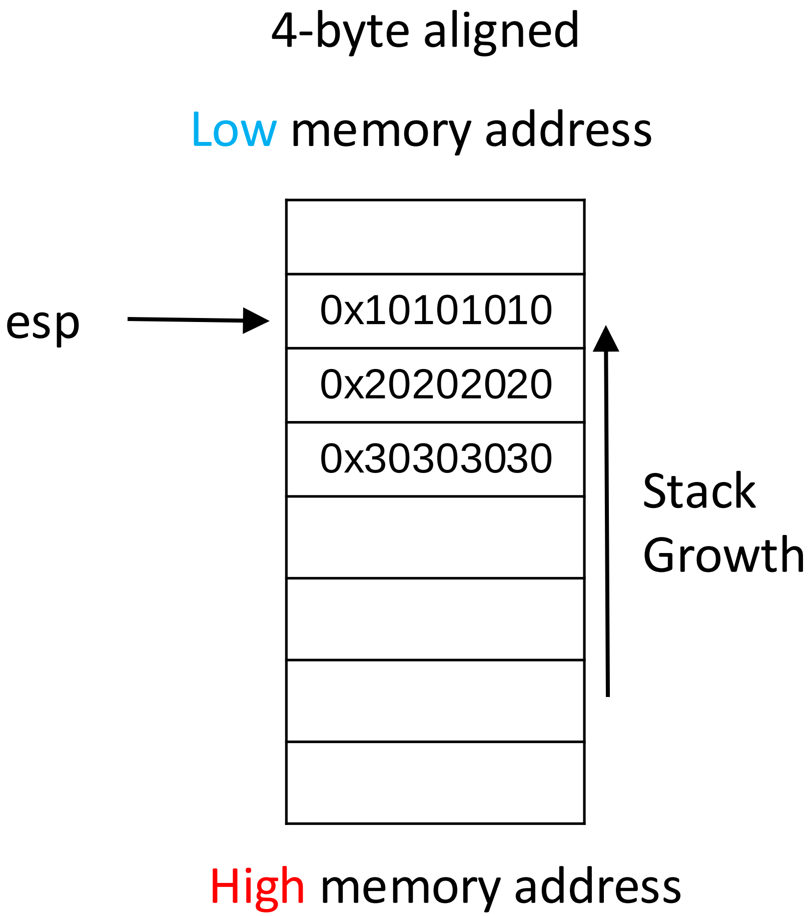
0x30303030

Stack Growth

High memory address

The Stack

- ESP (stack pointer) points to the top of the stack
- push instruction subtracts 4 from ESP and then writes to the top of the stack
  - Example: `push 0x40404040`



4-byte aligned

Low memory address

esp

0x10101010

0x20202020

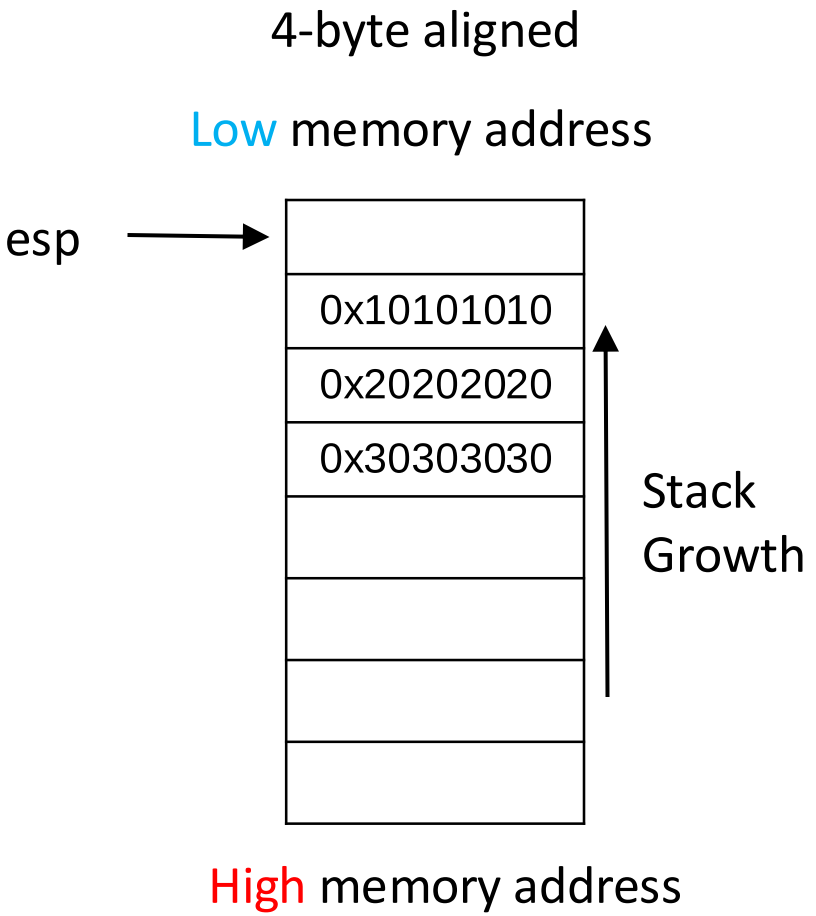
0x30303030

Stack Growth

High memory address

The Stack

- ESP (stack pointer) points to the top of the stack
- push instruction subtracts 4 from ESP and then writes to the top of the stack
    - Example: `push 0x40404040`



4-byte aligned

Low memory address

esp

0x10101010

0x20202020

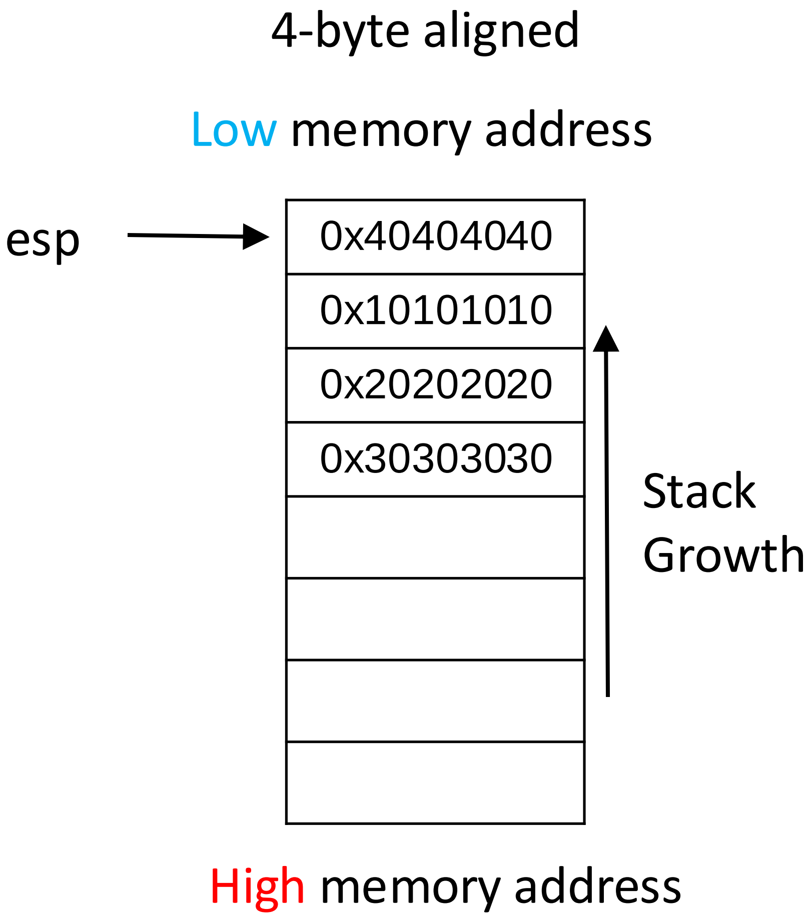
0x30303030

Stack Growth

High memory address

The Stack

- ESP (stack pointer) points to the top of the stack
- push instruction subtracts 4 from ESP and then writes to the top of the stack
    - Example: `push 0x40404040`



4-byte aligned

Low memory address

esp

0x40404040

0x10101010

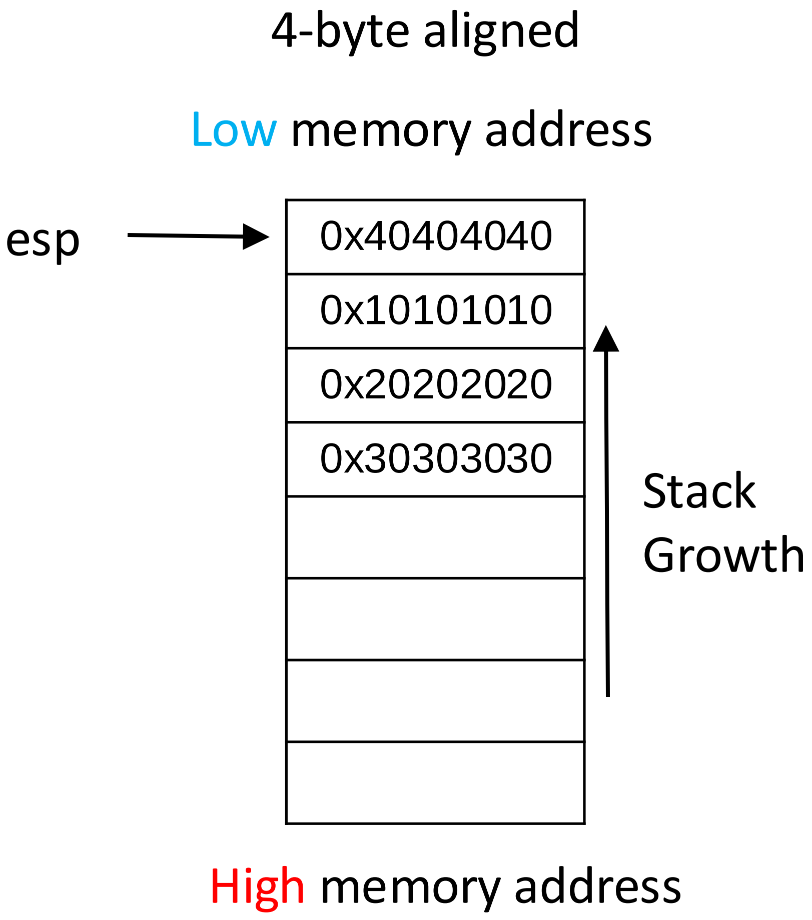
0x20202020

0x30303030

High memory address

The Stack

- ESP (stack pointer) points to the top of the stack
- push instruction subtracts 4 from ESP and then writes to the top of the stack
- pop instruction reads the value on the top of the stack and then adds 4 to ESP
  - ○ `Example: pop %eax`



4-byte aligned

Low memory address

esp

0x40404040

0x10101010

0x20202020

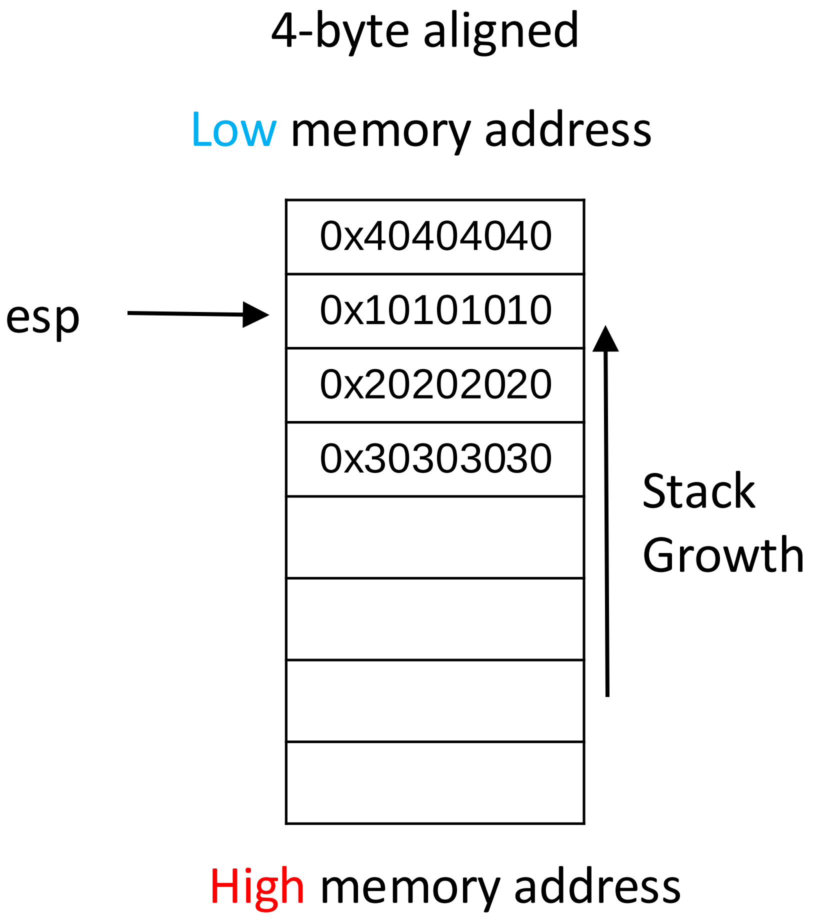
0x30303030

Stack Growth

High memory address

The Stack

- ESP (stack pointer) points to the top of the stack
- push instruction subtracts 4 from ESP and then writes to the top of the stack
- pop instruction reads the value on the top of the stack and then adds 4 to ESP
    - `Example: pop %eax`
    - `%eax <=== 0x40404040`



4-byte aligned

Low memory address

esp

0x40404040

0x10101010

0x20202020

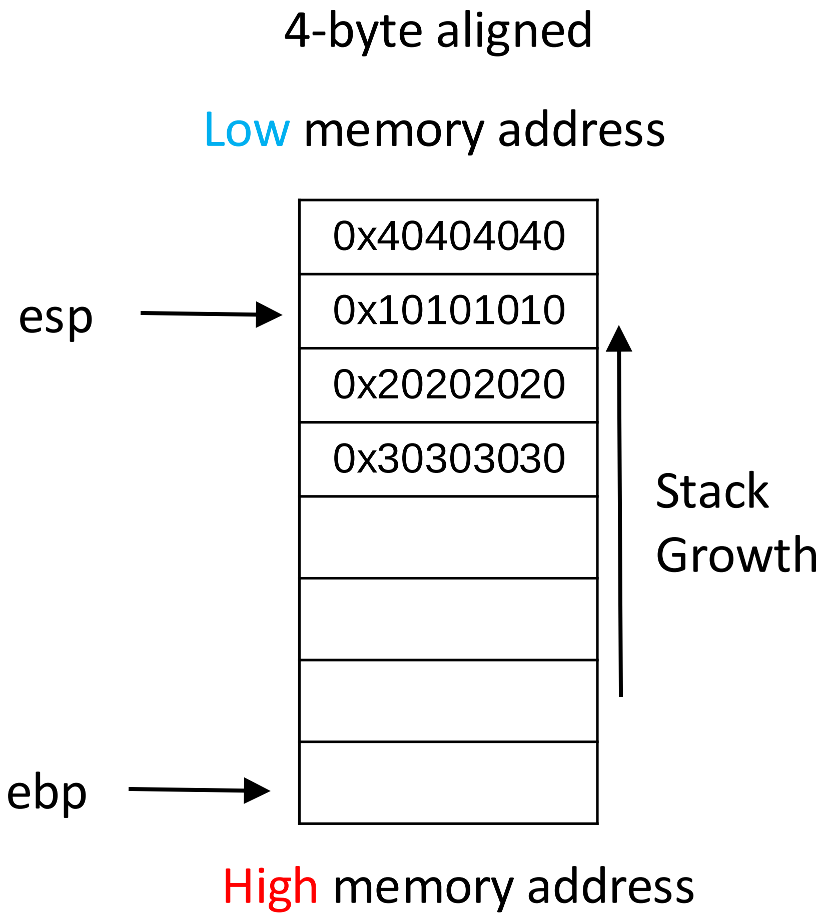
0x30303030

Stack Growth

High memory address

The Stack

- ESP (stack pointer) points to the top of the stack
- push instruction subtracts 4 from ESP and then writes to the top of the stack
- pop instruction reads the value on the top of the stack and then adds 4 to ESP
- EBP (base pointer / frame pointer) points to the bottom of the **current** stack



4-byte aligned

Low memory address

0x40404040

esp → 0x10101010

0x20202020

0x30303030

ebp →

Stack Growth

High memory address

Summary

- 32-bit x86 (AT&T/GAS syntax)
- Memory access syntax
- Stack: stack / base pointers, push/pop