CS 24

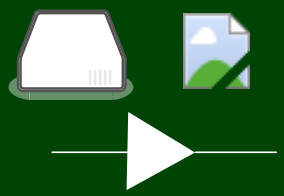
Introduction to Computing Systems

Memory and Fixed-Width Integers

Outline

Outline

Compilation Process



Hardware

```
01010101
11011011
01110111
```

Machine Code

```
movl %eax, -12(%rbp)
movl %esi, %eax
addq $16, %rsp
```

x86-64 Assembly

```
char *x = malloc(sizeof(char));
*x = 'a';
printf("%c\n", x);
free(x);
```

C Program

Java Virtual Machine

```
64 05 6c
```

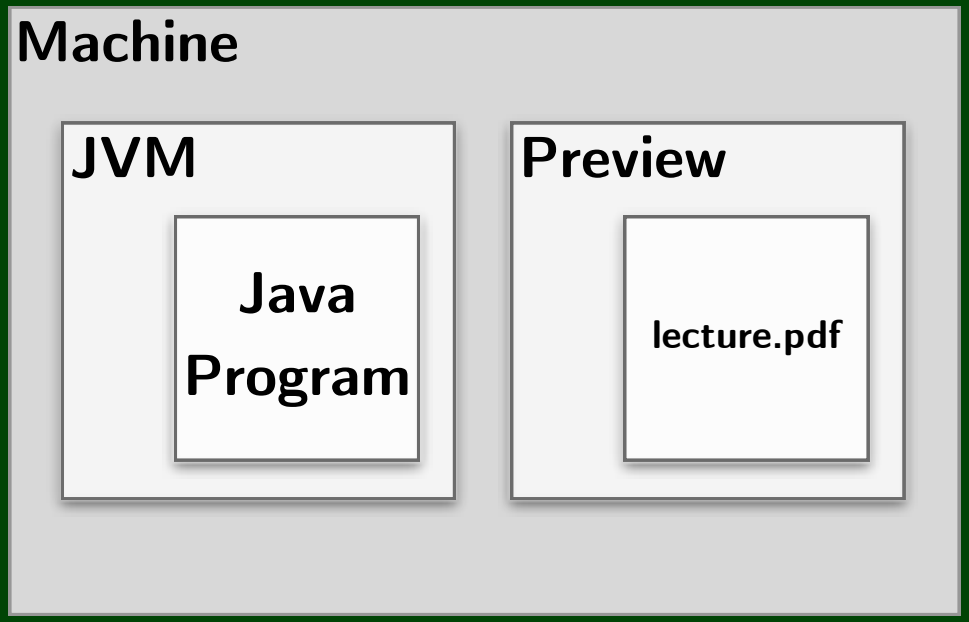
Java Bytecode

```
isub
iconst_2
idiv
```

Java Bytecode Instructions

```
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello");
    }
}
```

Java Program


Machine
JVM
Java
Program
Preview
lecture.pdf

Overview

In this project, you will implement all the integer JVM instructions. Your JVM will be able to run **real** compiled class files.

Learning Outcomes

- I can distinguish between how Java and C execute on a computer.
- I can identify the different levels of expressiveness between assembly/bytecode and statements in a high-level programming language.
- I can describe how code can be viewed as a type of data.
- I can write a virtual machine.

Outline

Memory, Addresses, and Pointers

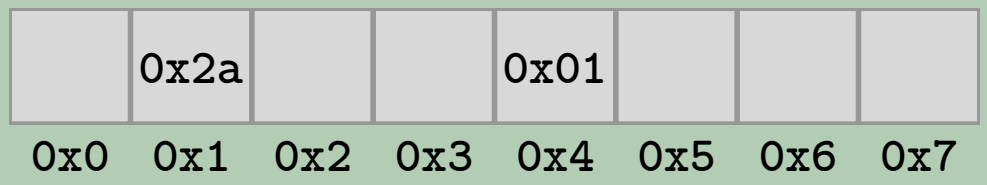
- **Memory** is (essentially) a large array of bytes.
- An **address** is an index into that array.
- A **pointer** is a variable that stores an address.

```
1 char *p = malloc(sizeof(char));
2 *p = 42;
3 printf("p = %p\n", p);
4 printf("*p = %p\n", *p);
5 printf("&p = %p\n", &p);
```

OUTPUT

```
>> p = 0x01
>> *p = 0x2a
>> &p = 0x04
```

A Picture of Memory

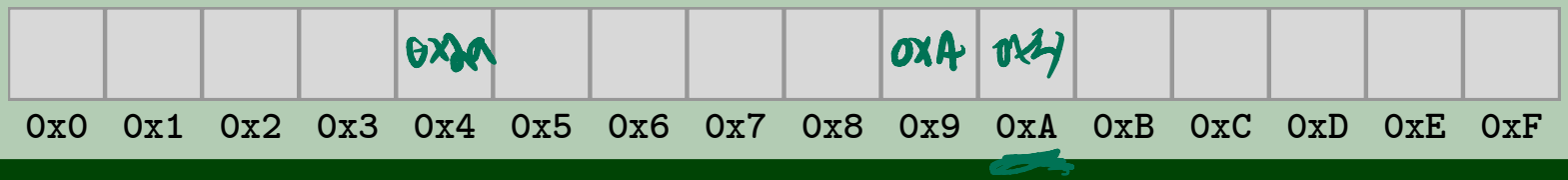


```
1  char **p = malloc(sizeof(char *));
2  *p = malloc(sizeof(char));
3  **p = 42;
4  printf("p = %p\n", p);
5  printf("*p = %p\n", *p);
6  printf("**p = %p\n", **p);
7  printf("&p = %p\n", &p);
8  printf("&*p = %p\n", &*p);
9  printf("*&p = %p\n", *&p);
```

OUTPUT

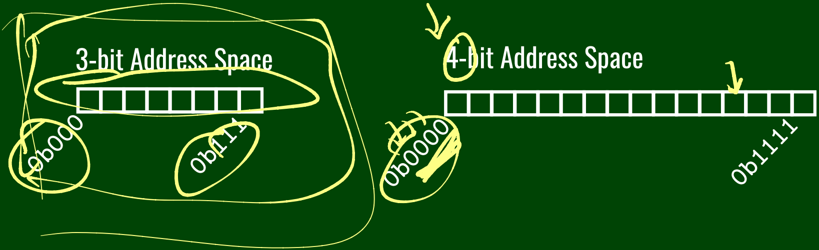
```
>> p = 0x0a
>> *p = 0x04
>> **p = 0x2a
>> &p = 0x09
>> &*p = 0x0a
>> *&p = 0x0a
```

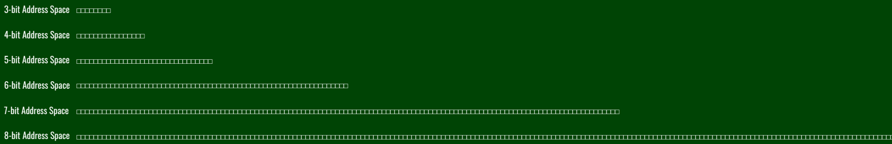
A Picture of Memory



Address Spaces

□ = 1 byte





Poll

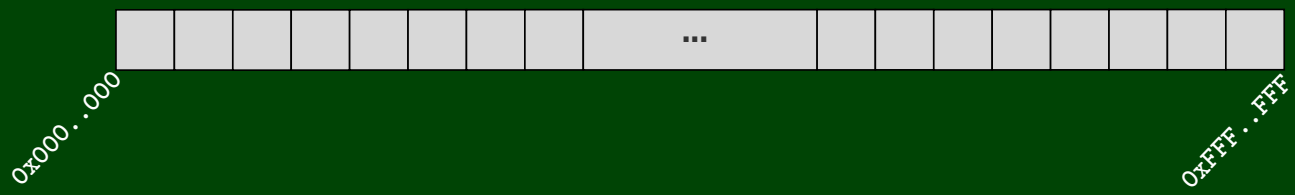
How many bits are necessary to represent an address in a tiny computer with only 8 addressable bytes?

a. 3
b. 4
c. 8
d. 64
e. ???

64-bit Address Space

The **word size** of a machine is the size of its **registers** and **addresses**.

`labradoodle` (and most other machines) have a 64-bit word size. This gives us 18 EB (exabytes) of addressable memory.

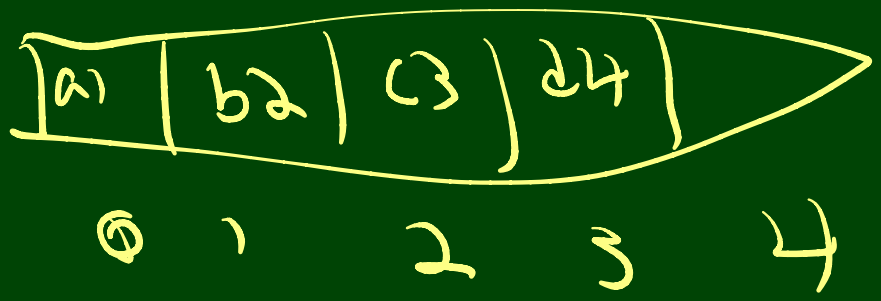


To reference a word, we use the address **of the first byte**. Thus, to move to the next word, we add **eight** (64-bit register = 8 bytes).

So, how are the bytes **within** a multi-byte word ordered in memory?

OUTPUT

```
>> x  = 0xa1b2c3d4
>> &x = 0x100
```



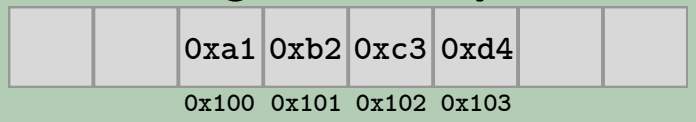
So, how are the bytes **within** a multi-byte word ordered in memory?

OUTPUT

```
>> x  = 0xa1b2c3d4
>> &x = 0x100
```

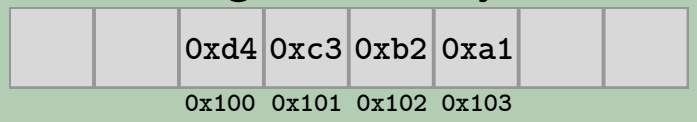
Big Endian (Internet, JVM)

Most Significant Byte First

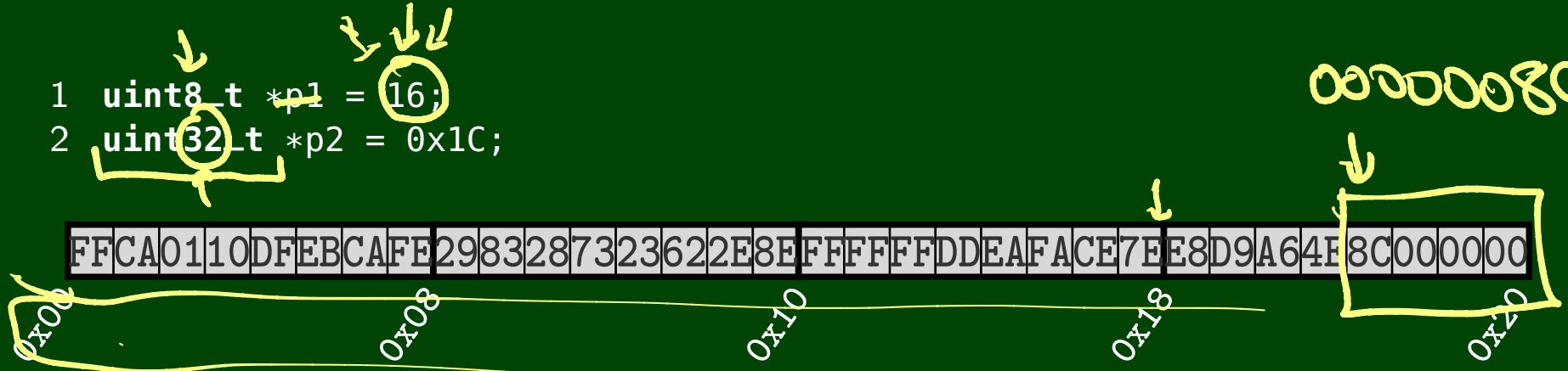


Little Endian (x86, ARM (most OSes))

Least Significant Byte First



```
1  uint8_t *p1 = 16;
2  uint32_t *p2 = 0x1C;
```



What are the values of *p1 and *p2 (in decimal) on a **little endian** machine?