CS 24

Introduction to Computing Systems

Memory and Fixed-Width Integers

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

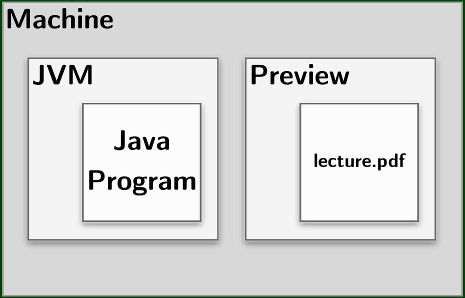
```
isub
iconst_2
idiv
```

Java Bytecode Instructions

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

Java Program

Executing Java Code



Project01: Build A TeenyJVM

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

1. Compilation and JVM
2. Memory
3. Integers
4. Adding and Removing Bits
5. Bit Operations

Memory Abstraction

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.

```c
char *p = malloc(sizeof(char));
*p = 42;
printf("p = %p\n", p);
printf("*p = %p\n", *p);
printf("&p = %p\n", &p);
```

OUTPUT

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

A Picture of Memory

	0x2a			0x01			
0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7

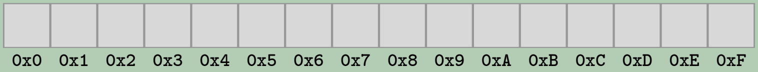
Memory Abstraction

```c
char **p = malloc(sizeof(char *));
*p = malloc(sizeof(char));
**p = 42;
printf("p = %p\n", p);
printf("*p = %p\n", *p);
printf("**p = %p\n", **p);
printf("&p = %p\n", &p);
printf("&*p = %p\n", &*p);
printf("*&p = %p\n", *&p);
```

OUTPUT

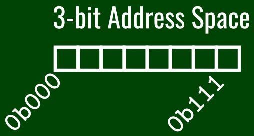
```
>> p = 0x0a
>> *p = 0x04
>> **p = 0x2a
>> &p = 0x09
>> &*p = 0x0a
>> *&p = 0x0a
```

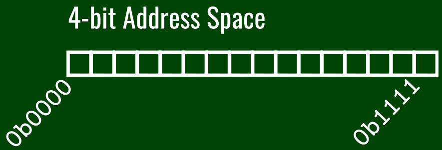
A Picture of Memory

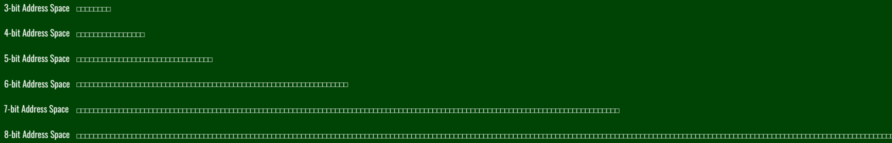


Address Spaces

□ = 1 byte







Address Spaces

Poll

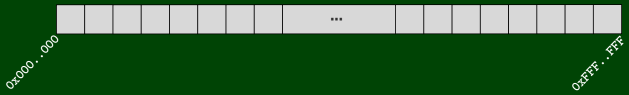
How many bits are necessary to represent an address in a tiny computer with only 8 addressable bytes?

- 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**.

`compute-cpu2` (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).

Reading/Storing Multiple Bytes: Endianness

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

OUTPUT

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

Reading/Storing Multiple Bytes: Endianness

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

		`0xa1`	`0xb2`	`0xc3`	`0xd4`		
		`0x100`	`0x101`	`0x102`	`0x103`		

Little Endian (x86, ARM $^{\text{(most OSes)}}$)

Least Significant Byte First

		`0xd4`	`0xc3`	`0xb2`	`0xa1`		
		`0x100`	`0x101`	`0x102`	`0x103`		

Memory and Endianness

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

FF	CA	01	10	DF	EB	CA	FE
29	83	28	73	23	62	2E	8E
FF	FF	FF	DD	EA	FA	CE	7E
E8	D9	A6	4E	8C	00	00	00

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

Putting It All Together

Suppose we declare `uint32_t *p;` on a 64-bit **little endian** machine. Also, suppose the following:

OUTPUT

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

Which memory locations do we know the values of and what are they?