CS 24

Introduction to Computing Systems

Memory and Fixed-Width Integers

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

0x00 0x08 0x10 0x18 0x20

Outline

Outline

hello.c

./a.out

```
01010101
11011011
01110111
```

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

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

Hardware

Machine Code

x86-64 Assembly

C Program

Java

```
64 05 6c
```

```
isub
iconst_2
idiv
```

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

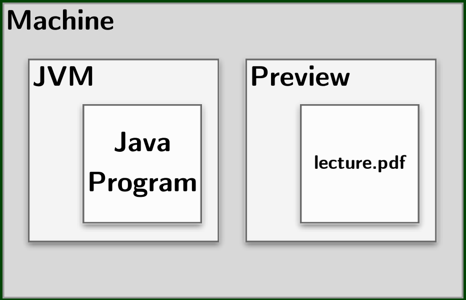
Java Virtual Machine

Java Bytecode

Java Bytecode Instructions

Java Program

java Hello


Machine
JVM
Java Program
Preview
lecture.pdf

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.

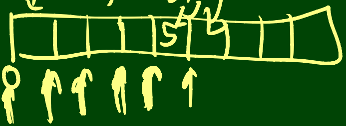
Outline

int x = 5;
(int*)x

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.

a single byte




```
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
```

p → 42

A Picture of Memory

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

Memory Abstraction

```
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);
```

char *p2 = *p

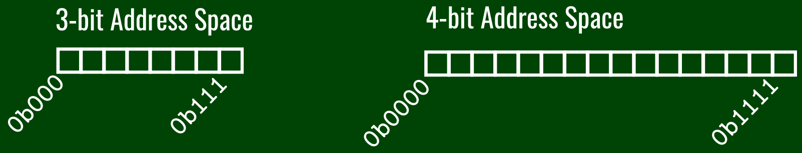
OUTPUT

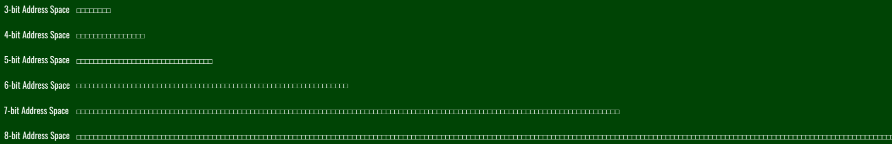
```
>> p = 0x0a
>> *p = 0x04
>> **p = 0x2a
>> &p = 0x09
>> &*p = 0x0a
>> *&p = 0x0a
```

A Picture of Memory

0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF
				2a					A	4					

□ = 1 byte


3-bit Address Space
0b000
0b111
4-bit Address Space
0b0000
0b1111


3-bit Address Space
4-bit Address Space
5-bit Address Space
6-bit Address Space
7-bit Address Space
8-bit Address Space

Poll

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

a 3

b 4

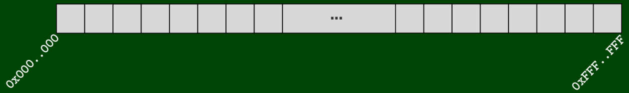
c 8

d 64

e ???

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

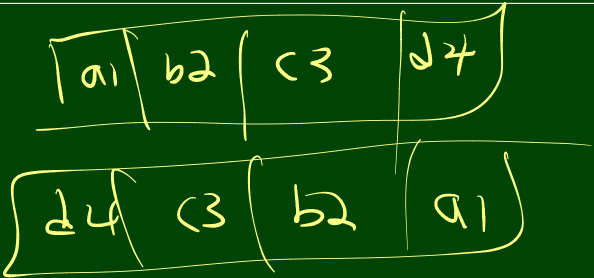
`labradoodle` (and most other machines) have a 64-bit word size. This gives us 18 EB (exabytes) of addressable memory.



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
```



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 (most OSes))

Least Significant Byte First

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

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

char *p = 16

0x00								0x08								0x10								0x18								0x20
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	

*p1 = 16

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

printf("%d", *p1)

*p1 = 255

*p2 = 0x8C

8C000000

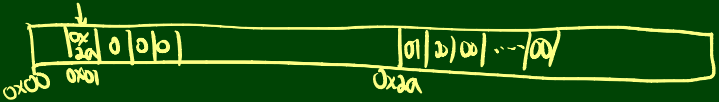
8C

DO THIS ON YOUR INDEX CARD!

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?



int *p = 1;

*p = 0x2a

↳ 4 bytes

Outline

Idealized integers can be an **unbounded** number of bits. But, instruction sets work over specific numbers of bytes (e.g., the word size). For example, the `uint8_t` representation of 4 is `0b00000100`.

In general, if the word length is w, then $\boxed{(b_{w-1} \cdots b_0)_2 = \sum_{i=0}^{w-1} b_i 2^i.}$

Poll

What is the largest number representable by 4 bits?

- a 16
- b 15
- c 8
- d 7
- e ???

size_t x = 0;

printf("%lu", x-1);

This takes care of **unsigned** integers, but how do we represent **signed integers**?

$\boxed{\text{uint}\square\text{\_t}}$ int32_t

In general, if the word length is w, then

$\sum b_i \cdot 2^i$

$$(b_{w-1} \cdots b_0)_2 = -b_{w-1} 2^{w-1} + \boxed{\left(\sum_{i=0}^{w-2} b_i 2^i \right)}$$

In general, if the word length is w, then

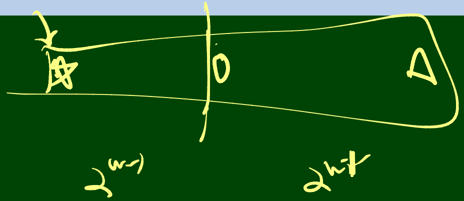
$$(b_{w-1} \cdots b_0)_2 = -b_{w-1}2^{w-1} + \left(\sum_{i=0}^{w-2} b_i 2^i\right)$$

$2^{w-1} - 1$

Poll

Which of these is the 8-bit two's complement representation of -1?

- a 0b11111111
- b 0b01111111
- c 0b10000000
- d 0b00010000
- e ???



In general, if the word length is w, then

$$(b_{w-1} \cdots b_0)_2 = -b_{w-1} 2^{w-1} + \left(\sum_{i=0}^{w-\mathbf{2}} b_i 2^i \right)$$

In general, if the word length is w, then

$$(b_{w-1} \cdots b_0)_2 = -b_{w-1} 2^{w-1} + \left(\sum_{i=0}^{w-2} b_i 2^i \right)$$

Poll

Which of these is the 16-bit two's complement representation of -1?

- a 0x1000
- b 0xF000
- c 0xFFFF
- d 0xEFFF
- e ???

0xF = 0b1111

0x0FFF

0b0111 = 0x7

A Program in x86-64


```
mystery:
   test %edi, %edi
   je   L2
L1:
   imul %edi, %esi
   add  $0xffffffff, %edi
   jne  L1
L2:
   mov  %esi, %eax
   retq
```

w bits!

Base 16	**Unsigned**	**Signed**
Min		
Max		
-1		

Base 10	**Unsigned**	**Signed**
Min		
Max		

Special Integer Values

Base 16	**Unsigned**	**Signed**
Min	`0x0000...`	`0x8000...`
Max	`0xFFFF...`	`0x7FFF...`
-1	Not representable	`0xFFFF...`

Base 10	**Unsigned**	**Signed**
Min	0	-2^{w-1}
Max	$2^w - 1$	$2^{w-1} - 1$

a && b <- boolean expr.

a & b <- bitwise expr

0010

a = 1011
& & & &
b = 0100

0000

1011
1101
001
0001

a << 1

1011 << 1

0110

unsigned right shift

Signed >>

0b00001111 << 1

0b00001111 >> 1

Special Integer Values

Base 16	**Unsigned**	**Signed**
Min	`0x0000...`	`0x8000...`
Max	`0xFFFF...`	`0x7FFF...`
-1	Not representable	`0xFFFF...`

Base 10	**Unsigned**	**Signed**
Min	0	-2^{w-1}
Max	$2^w - 1$	$2^{w-1} - 1$