Section 1: Basic explanations
===========================

1. In x86 The convention is that eax is used for return value of functions
2. Register shorthand naming: eax - 4 bytes, ax - last two bytes of eax, al - last byte of eax (they are the same register physical on processor). Similar for edx, ecx, ebx, etc.

3. On x86-64 architectures you'll notice prefix "r".. Eg. rax represents the 64 bits extension of eax. If you need only the least significant 4 bytes, use eax. If you need all 8 bytes, use rax => they are the same register physical on processor.
4. 2. RSP / ESP are stack pointers and control function calls, allocation of parameters, temporary data.
5. Many instructions have a suffix of 4 possible letters like mov instr: mov[b,w,l,d]
   b - 1byte, w-2bytes, l-4bytes, d-8bytes
6. Addressing mode can be "immediate -imm" (**constants** - values or memory addresses), register value / a value in memory. You will notice these in River code along instructions since it needs to know type of operands.
   Displacement examples:
   A. mov 4(%edx), %eax => R[eax] = M[R[edx] + 4]] , where M is memory and R is register value table
   B. movl 80(%edx,%ecx,2),%eax => R[eax] = M[R[edx] + R[ecx]*2 + 80
   Uses only the last 4 bytes of eax.

7. Check you understanding on the code below please !!

```c
#include <stdio.h>

int sum(int *a, int n) {
    int total = 0;

    for (int i = 0; i < n; i++) {
      total += a[i];
    }

    return total;
}

int main() {
    int numbers[5] = {1, 2, 3, 4, 5};
    printf("%d\n", sum(numbers, 5));
    return 0;
}
```

```
sum:
        movl  $0, %edx
        movl  $0, %eax
        jmp   .L2
.L3:
        movslq %edx, %rcx
        addl  (%rdi,%rcx,4), %eax
        addl  $1, %edx
.L2:
        cmpl  %esi, %edx
        jl    .L3
        rep ret
.LC0:
        .string     "%d\n"
```

eax mapeaza pe total Avem 4 in dispacement pentru ca int are 4 bytes. Acolo accesam a[i] si-l adunam la total (eax)

```
main:
        subq  $40, %rsp
        movl  $1, (%rsp)
        movl  $2, 4(%rsp)
        movl  $3, 8(%rsp)
        movl  $4, 12(%rsp)
        movl  $5, 16(%rsp)
        movq  %rsp, %rdi
        movl  $5, %esi
        call  sum
        movl  %eax, %esi
        leaq  .LC0(%rip), %rdi
        movl  $0, %eax
        call  printf@PLT
        movl  $0, %eax
        addq  $40, %rsp
        ret
```

Punem parametrii pe stiva (rsp)

Lea calculeaza o adresa , in cazul asta pune in RDI adresa stringului pentru printf

Lea calculeaza o adresa , in cazul asta pune in RDI adresa stringului pentru printf

In eax se pune mereu adresa de return (0 in cazul asta)

Compile to assembly using: `gcc -S -Og array_sum.c`

- Try without `-Og`: What changes? Why?
- Note: use of 128 byte red zone after stack pointer.

8. Jumps and FLAGS

You'll notice the terms of flags in code: ZF, OF, SF, … associated with the jump instructions.
This is a very important topic to RIVER. PLEASE read slides 7-13 from the link below
https://fmiunibuc-my.sharepoint.com/:b:/g/personal/ciprian_paduraru_fmi_unibuc_ro/EWVfHBqDK4VJgAq3EuPNIsMBrEbj6i_fvfvPABug8gSC_w?e=PFvHlx

Section 2: Explanation about code
===

NOTE: Do not suppose any input when you analyze this code (e.g. "BBBB…"). The disassembled output is independent on any input.

	int x;
	void test_simple(const unsigned char *buf) {
	int i = 1;
	if (buf[0] == 'B') {

	if (buf[1] == 'A') {
	i = 2;
	}
	}
	x = i;
	}
00000530 <test_simple>:	
530: 8b 54 24 04 mov 0x4(%esp),%edx	ESP is the stack register. Remember that stack pointer grows from top to bottom. Function has a single parameter so at esp + 4 (stack pointer) - 0x4(%esp). - we have the address of "buf" parameter. This instruction copies buf address to EDX
534: b8 01 00 00 00 mov $0x1,%eax	We put constant 1 in register EAX (thus, eax is mapped to i)
539: 80 3a 42 cmpb $0x42,(%edx)	Compare constant 0x42 ('B') to the memory address referenced by EDX (buf[0]). The result will be put in ZF . Notice that cmp['b'] is used. B means to compare the least significant byte !
53c: 75 0c jne 54a <test_simple+0x1a>	If the result is not true, we jump to 54a. Not very important: In the instruction code (left), notice '0c' - this represents the offset to jump after this instruction which = 54a
53e: 31 c0 xor %eax,%eax	We 0 eax
540: 80 7a 01 41 cmpb $0x41,0x1(%edx)	Compare 'A" with buf[1] and sets the result in ZF. (Notice EDX address + 1 byte)
544: 0f 94 c0 sete %al	Sete works as follows: if ZF is 1, sets al to 1, else to 0. Remember that "al" register is the least significant part of eax (which maps to variable "I"). So if buf[1] = 'A' it will set eax to 1.
547: 83 c0 01 add $0x1,%eax	This will add 1 to eax (making i = 2 if both ifs are taken ! Seems redundant for you, but this is optimized code actually produced by compiler… :)).

54a: a3 00 00 00 00 mov %eax, 0x0	This is the jump explained above, when buf[0] != 'B', doing nothing - X is not used anywhere so it's optimized. Not important
54b: R_386_32 x	Relocation, he doesn't know the value of X, Not important.
54f: c3 ret	

Section 3: Advanced things, buffer overflows, types of attacks
==
WORK IN PROGRESS