**** x86 ASSEMBLY BASICS ****



# The slides & practice programs can be found on github

github.com/susikohmelo/assembly_basics_presentation

# What *actually* is Assembly?

## Generic term

Languages which communicate directly with hardware.

Hardware specific.

Many different syntax can exist for the same language

```
***** EXAMPLES *****

x86   : intel & AMD

ARM   : mobile

RISCV : open-source
        alternative
```

# (Almost) everything is just memory

**Operations, variables...**
**It's all just data in memory.**

Fundamentally there is no difference between data and code.

Everything in your program is just a value stored in some address of RAM which you can freely access and edit.
**( Assuming the OS lets you )**

**You can even make self modifying code**

# Registers

**Super fast, temporary memory inside the CPU**

**Registers can be used to:**

- Store/assign values from/to other registers & RAM

- Do operations on the value they hold (add, multiply etc.)

# (Some) commonly used registers

***** GENERAL *****

ANYTHING GOES HERE
ax, bx, cx, dx

SRC/DEST POINTERS
si : source index
di : destination

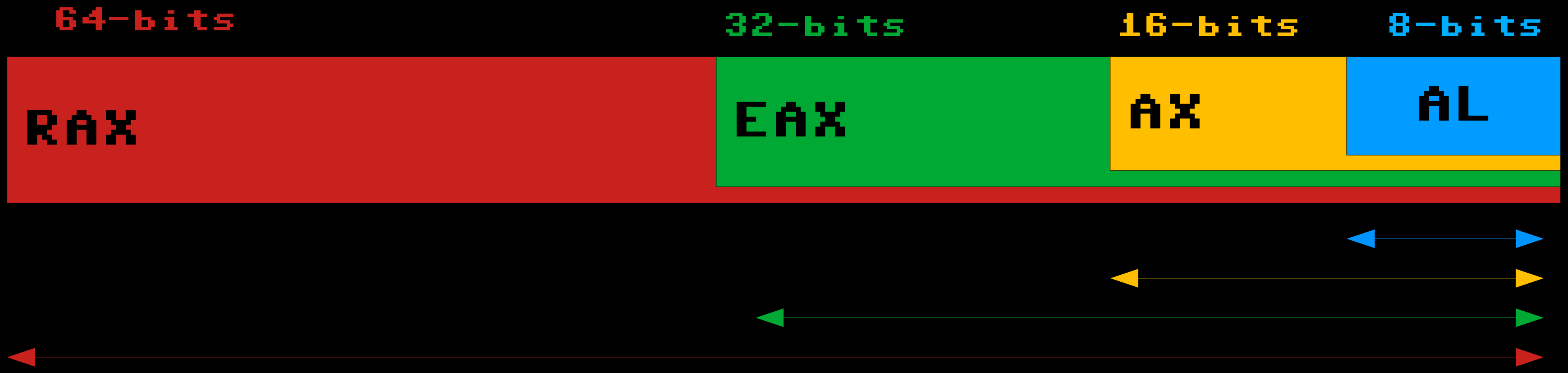
***** SPECIAL *****

STACK POINTERS
sp : Top pointer
bp : Bottom pointer

FLAG REGISTERS
bazillion of them

# Register inception

**Registers can be divided into multiple smaller registers**




# Instructions

**Instructions are the actual commands we can give the CPU to execute**

- Operations on registers (assign, add, multiply etc.)
- Move values in RAM & registers
- Jump to locations

```
***** EXAMPLES *****

MOV   ax, 42
--------------------
Move the value 42
into register ax

ADD   ax, bx
--------------------
Add value in bx to
the value in ax.
```

# Simple program example

```
*****   PROGRAM   *****

_start:                  ←

    MOV   ax, 42

    MOV   bx, ax

    ADD   ax, bx

    MOV   bx, 0x42
    MOV   [bx], ax
```

```
***** REGISTERS *****

ax  :  0

bx  :  0


*****     RAM     *****

0x42  =  0
```

# Simple program example

```
*****  PROGRAM  *****

_start:
    MOV   ax, 42      ←
    MOV   bx, ax
    ADD   ax, bx
    MOV   bx, 0x42
    MOV   [bx], ax
```

```
***** REGISTERS *****

ax : 42
bx : 0

*****    RAM    *****

0x42 = 0
```

# Simple program example

```
*****   PROGRAM   *****

_start:
    MOV   ax, 42
    MOV   bx, ax        ←
    ADD   ax, bx
    MOV   bx, 0x42
    MOV   [bx], ax
```

```
***** REGISTERS *****

ax : 42
bx : 42

*****    RAM    *****

0x42 = 0
```

# Simple program example

```
***** PROGRAM *****

_start:
    MOV   ax, 42
    MOV   bx, ax
    ADD   ax, bx       ←
    MOV   bx, 0x42
    MOV   [bx], ax
```

```
***** REGISTERS *****

ax : 84
bx : 42

*****   RAM   *****

0x42 = 0
```

# Simple program example

```
*****   PROGRAM   *****

_start:
    MOV   ax, 42
    MOV   bx, ax
    ADD   ax, bx
    MOV   bx, 0x42    ←
    MOV   [bx], ax
```

```
***** REGISTERS *****

ax  :  84
bx  :  0x42

*****    RAM    *****

0x42 = 0
```

# Simple program example

```
*****   PROGRAM   *****

_start:
    MOV   ax, 42
    MOV   bx, ax
    ADD   ax, bx
    MOV   bx, 0x42
    MOV   [bx], ax     ←
```

```
***** REGISTERS *****

ax : 84
bx : 42

*****    RAM    *****

0x42 = 84
```

**Let’s make a simple A+B function that can be called from a C program**

# How can we pass parameters/args?

**Many ways!**

1. Arbitrary RAM location (bad)
2. Registers
3. Stack

**In 32-bit C programs, all parameters are pushed onto stack.**
[ We'll look at this later ]

**In 64-bit C programs, parameters are put into specific registers, arguments > 6 get put onto stack.**

```
** 64-BIT LINUX **
CALLING CONVENTION

rdi   : param. 1
rsi   : param. 2
rdx   : param. 3
rcx   : param. 4
r8    : param. 5
r9    : param. 6

stack: param. 7..
```

Let’s add that

# System calls (syscall)

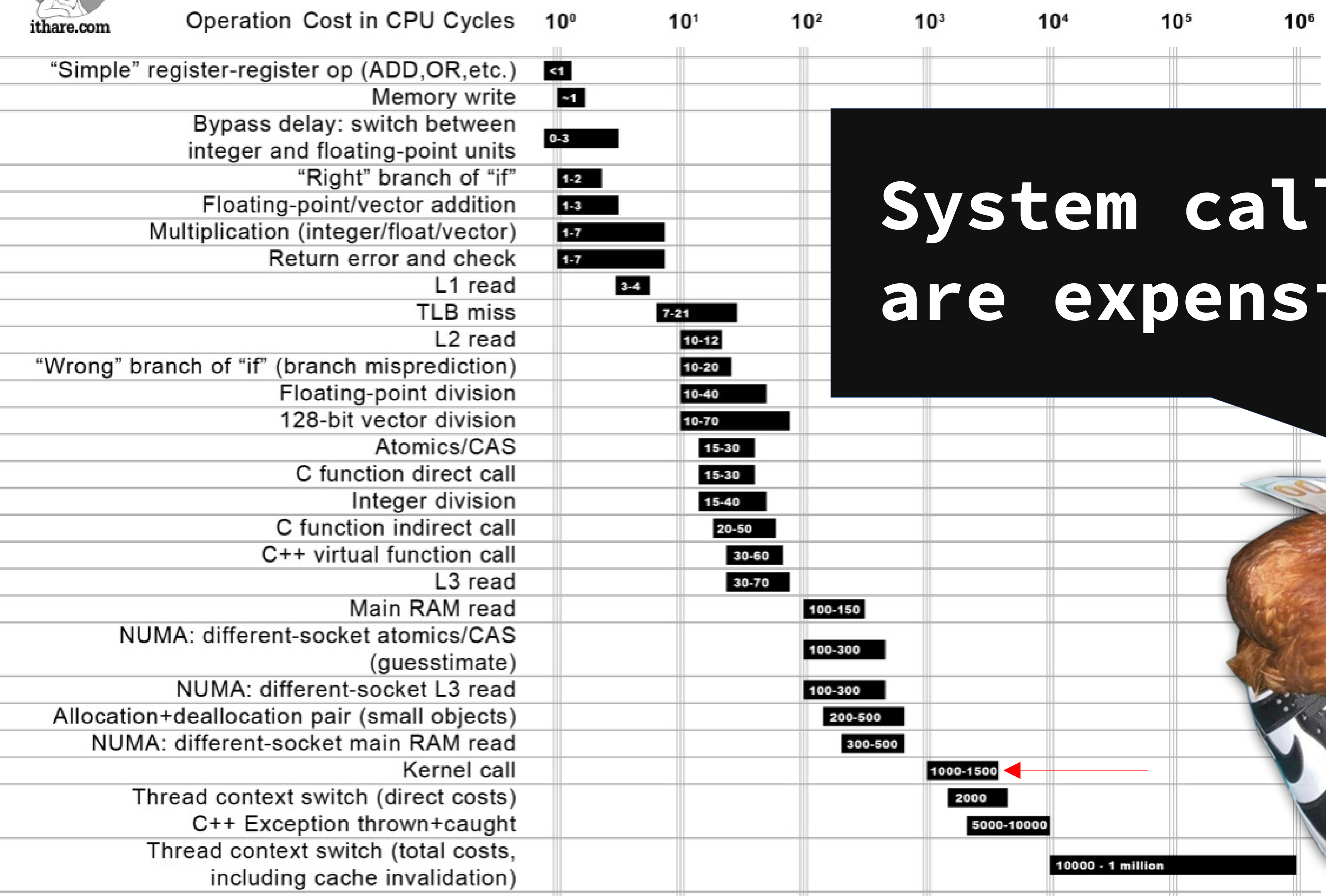
**System calls ask the OS to do something for us**

Usually done by sending a software interrupt signal.

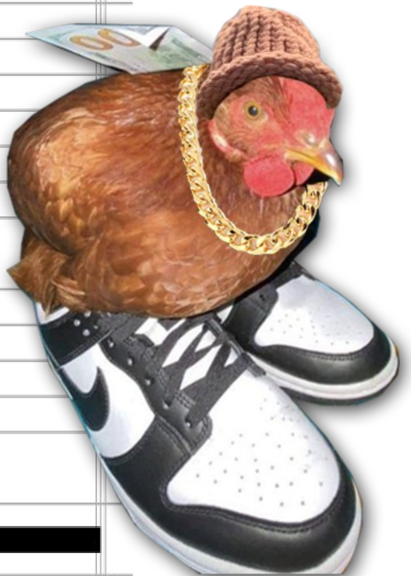
Registers are used as parameters for the syscall.

‘syscall’ sends the interrupt

```
***** EXAMPLE *****

/* syscall #60 */
MOV     rax, 60

/* exit code */
MOV     rdi, 42

syscall
```




# System calls are expensive



Let’s print something onto the screen with a system call

# The stack redpill

**sp/bp point to the top and bottom of the stack**

**You can access anything in the stack, not just the top!**

When you push a value, it gets put in to the address that the stack top is pointing to & the pointer is incremented.

The stack grows downwards. So as you add more things, the top of the stack becomes a smaller address.

**You can PUSH and POP registers and values onto the stack. But you can also just use the stack like a normal pointer!**

**NOTE**

**In the examples the push/pop operations add/remove 1 byte.**

**Unless you're on 8/16-bit hardware, the stack typically operates on 4+ bytes.**

The return address is also not being shown
[ That is coming later ]

# Simple program example

```
*****   PROGRAM   *****

_start:                ←

    PUSH 42
    PUSH 3

    MOV  al, [sp+1]
    POP  bl

    ADD  sp, 1
```

```
*****   STACK   *****

RAM
------------------
0x00 :  0
0x01 :  0
0x02 :  0
0x03 :  0    ← sp/bp

REGISTERS
------------------
sp  :  0x03
bp  :  0x03
al  :  0
bl  :  0
```

# Simple program example

```
*****   PROGRAM   *****

_start:

    PUSH 42           ←
    PUSH 3

    MOV  al, [sp+1]
    POP  bl

    ADD  sp, 1
```

```
*****   STACK   *****

RAM
------------------
0x00 :  0
0x01 :  0
0x02 :  42  ← sp
0x03 :  0   ← bp

REGISTERS
------------------
sp  :  0x02
bp  :  0x03
al  :  0
bl  :  0
```

# Simple program example

```
*****   PROGRAM   *****

_start:

    PUSH 42
    PUSH 3              ←

    MOV  al, [sp+1]
    POP  bl

    ADD  sp, 1
```

```
*****   STACK   *****

RAM
--------------------
0x00  :  0
0x01  :  3   ← sp
0x02  :  42
0x03  :  0   ← bp

REGISTERS
--------------------
sp  :  0x01
bp  :  0x03
al  :  0
bl  :  0
```

# Simple program example

```
*****   PROGRAM   *****

_start:

    PUSH 42
    PUSH 3

    MOV  al, [sp+1]←
    POP  bl

    ADD  sp, 1
```

```
*****   STACK   *****

RAM
------------------
0x00  :  0
0x01  :  3   ← sp
0x02  :  42
0x03  :  0   ← bp

REGISTERS
------------------
sp  :  0x01
bp  :  0x03
al  :  42
bl  :  0
```

# Simple program example

```
*****   PROGRAM   *****

_start:

    PUSH 42
    PUSH 3

    MOV  al, [sp+1]
    POP  bl              ←

    ADD  sp, 1
```

```
*****   STACK   *****

RAM
--------------------
0x00  :  0
0x01  :  3
0x02  :  42   ←  sp
0x03  :  0    ←  bp

REGISTERS
--------------------
sp  :  0x02
bp  :  0x03
al  :  42
bl  :  3
```

# Simple program example

```
*****   PROGRAM   *****

_start:

    PUSH 42
    PUSH 3

    MOV  al, [sp+1]
    POP  bl

    ADD  sp, 1         ←
```

```
*****   STACK   *****

RAM
--------------------
0x00  :  0
0x01  :  3
0x02  :  42
0x03  :  0    ← sp/bp

REGISTERS
--------------------
sp  :  0x03
bp  :  0x03
al  :  42
bl  :  3
```

# ‘Stack frames’

**Refers to a region of the stack “belonging” to a function**

On function calls, the bottom ptr is saved to the stack.
The bottom ptr is then set to point to top of the stack.

This effectively makes a new stack on top of the old one.

When the function returns, we pop the old bottom pointers location and now we are back in the original stack frame.

# Simple program example

```
*****   PROGRAM   *****

my_funct:
    PUSH bp
    MOV  bp, sp
    ...

    POP  bp
    RET

_start:                    ←
    PUSH 42
    CALL my_funct()
    ...
```

```
*****   STACK   *****

RAM
------------------
0x00  :  0
0x01  :  0
0x02  :  0
0x03  :  0
0x04  :  0
0x05  :  0       ← sp/bp

REGISTERS
------------------
sp  :  0x05
bp  :  0x05
```

# Simple program example

```
*****   PROGRAM   *****

my_funct:
    PUSH bp
    MOV  bp, sp
    ...

    POP  bp
    RET

_start:
    PUSH 42           ←
    CALL my_funct()
    ...
```

```
*****   STACK   *****

RAM
--------------------
0x00  :  0
0x01  :  0
0x02  :  0
0x03  :  0
0x04  :  42     ← sp
0x05  :  0      ← bp

REGISTERS
--------------------
sp  :  0x04
bp  :  0x05
```

# Simple program example

```
*****   PROGRAM   *****

my_funct:
      PUSH  bp
      MOV   bp, sp
      ...

      POP   bp
      RET

_start:
      PUSH  42
      CALL  my_funct()←
      ...
```

```
*****   STACK   *****

RAM
------------------
0x00  :  0
0x01  :  0
0x02  :  0
0x03  :  0
0x04  :  42     ← sp
0x05  :  0      ← bp

REGISTERS
------------------
sp  :  0x04
bp  :  0x05
```

# Simple program example

```
*****   PROGRAM   *****

my_funct:                    ←
    PUSH bp
    MOV  bp, sp
    ...

    POP  bp
    RET

_start:
    PUSH 42
    CALL my_funct()
    ...
```

```
*****   STACK   *****

RAM
------------------
0x00 : 0
0x01 : 0
0x02 : 0
0x03 : 0
0x04 : 42    ← sp
0x05 : 0     ← bp

REGISTERS
------------------
sp : 0x04
bp : 0x05
```

# Simple program example

```
*****   PROGRAM   *****

my_funct:
    PUSH  bp            ←
    MOV   bp, sp
    ...

    POP   bp
    RET

_start:
    PUSH  42
    CALL  my_funct()
    ...
```

```
*****   STACK   *****

RAM
------------------
0x00  :  0
0x01  :  0
0x02  :  0
0x03  :  0x05←  sp
0x04  :  42
0x05  :  0    ←  bp

REGISTERS
------------------
sp  :  0x03
bp  :  0x05
```

# Simple program example

```
*****   PROGRAM   *****

my_funct:
    PUSH bp
    MOV  bp, sp       ←
    ...

    POP  bp
    RET

_start:
    PUSH 42
    CALL my_funct()
    ...
```

```
*****   STACK   *****

RAM
------------------
0x00  :  0
0x01  :  0
0x02  :  0
0x03  :  0x05←  sp/bp
0x04  :  42
0x05  :  0

REGISTERS
------------------
sp  :  0x03
bp  :  0x03
```

# Simple program example

```
*****   PROGRAM   *****

my_funct:
      PUSH  bp
      MOV   bp, sp
      ...                  <-

      POP   bp
      RET

_start:
      PUSH  42
      CALL  my_funct()
      ...
```

```
*****   STACK   *****

RAM
------------------
0x00  :  0
0x01  :  0
0x02  :  0
0x03  :  0x05<-  sp/bp
0x04  :  42
0x05  :  0

REGISTERS
------------------
sp  :  0x03
bp  :  0x03
```

# Simple program example

```
*****   PROGRAM   *****

my_funct:
    PUSH bp
    MOV  bp, sp
    ...

    POP  bp
    RET

_start:
    PUSH 42
    CALL my_funct()
    ...
```

```
*****   STACK   *****

RAM
------------------
0x00 : 0
0x01 : 0
0x02 : 99      ← sp
0x03 : 0x05    ← bp
0x04 : 42
0x05 : 0

REGISTERS
------------------
sp : 0x02
bp : 0x03
```

# Simple program example

```
*****   PROGRAM   *****

my_funct:
    PUSH bp
    MOV  bp, sp
    ...                    ←

    POP  bp
    RET

_start:
    PUSH 42
    CALL my_funct()
    ...
```

```
*****   STACK   *****

RAM
------------------
0x00 : 0
0x01 : 22    ← sp
0x02 : 99
0x03 : 0x05  ← bp
0x04 : 42
0x05 : 0

REGISTERS
------------------
sp : 0x01
bp : 0x03
```

# Simple program example

```
*****   PROGRAM   *****

my_funct:
    PUSH bp
    MOV  bp, sp
    ...                  ←

    POP  bp
    RET

_start:
    PUSH 42
    CALL my_funct()
    ...
```

```
*****   STACK   *****

RAM
------------------
0x00 :  0
0x01 :  22
0x02 :  99     ← sp
0x03 :  0x05←  bp
0x04 :  42
0x05 :  0

REGISTERS
------------------
sp  :  0x02
bp  :  0x03
```

# Simple program example

```
*****  PROGRAM  *****

my_funct:
    PUSH bp
    MOV  bp, sp
    ...                  ←

    POP  bp
    RET

_start:
    PUSH 42
    CALL my_funct()
    ...
```

```
*****  STACK  *****

RAM
------------------
0x00 : 0
0x01 : 22
0x02 : 99
0x03 : 0x05← sp/bp
0x04 : 42
0x05 : 0

REGISTERS
------------------
sp : 0x03
bp : 0x03
```

# Simple program example

```
*****   PROGRAM   *****

my_funct:
    PUSH bp
    MOV  bp, sp
    ...

    POP  bp              ←
    RET

_start:
    PUSH 42
    CALL my_funct()
    ...
```

```
*****   STACK   *****

RAM
------------------
0x00  :  0
0x01  :  22
0x02  :  99
0x03  :  0x05
0x04  :  42     ← sp
0x05  :  0      ← bp

REGISTERS
------------------
sp  :  0x04
bp  :  0x05
```

# Simple program example

```
*****   PROGRAM   *****

my_funct:
    PUSH bp
    MOV  bp, sp
    ...

    POP  bp
    RET                 ←

_start:
    PUSH 42
    CALL my_funct()
    ...
```

```
*****   STACK   *****

RAM
------------------
0x00 : 0
0x01 : 22
0x02 : 99
0x03 : 0x05
0x04 : 42      ← sp
0x05 : 0       ← bp

REGISTERS
------------------
sp : 0x04
bp : 0x05
```

# Simple program example

```
*****   PROGRAM   *****

my_funct:
    PUSH bp
    MOV  bp, sp
    ...

    POP  bp
    RET

_start:
    PUSH 42
    CALL my_funct()
    ...
```

```
*****   STACK   *****

RAM
------------------
0x00 : 0
0x01 : 22
0x02 : 99
0x03 : 0x05
0x04 : 42      ← sp
0x05 : 0       ← bp

REGISTERS
------------------
sp : 0x04
bp : 0x05
```

Let’s look at some disassembled C code

We should now be able to understand quite a lot of what’s going on.

# How does the CPU know where to go

**Push, then pop a return address**

**A pointer to the current line of code being executed by the CPU is kept in the instruction pointer register (IP)**

**When you call a function:**

1. Pushes next line of code to the stack
2. Sets IP as the newly called address

**Returning from a function**

1. Pops the address back into IP

# Interacting with hardware

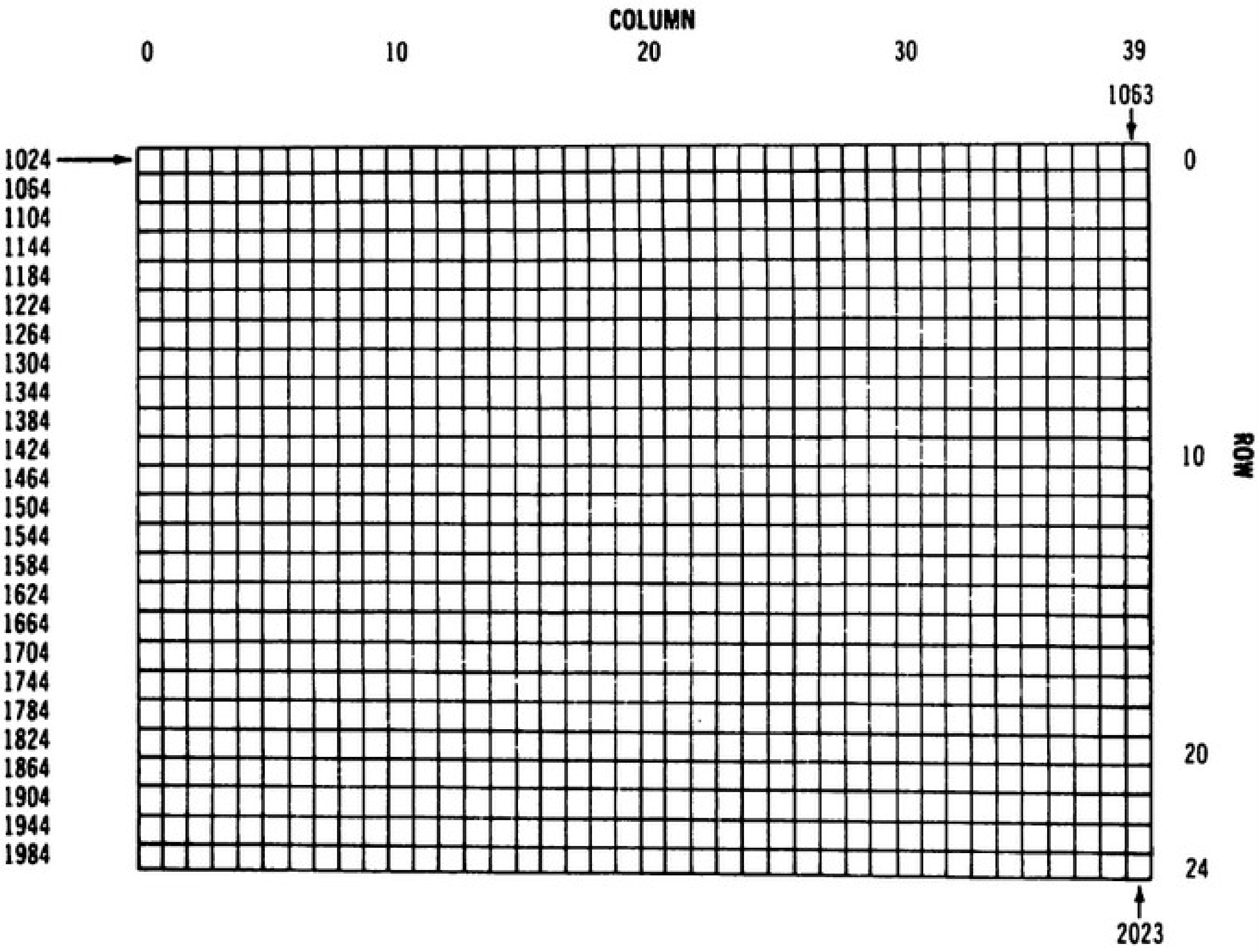
**The majority of devices are mapped somewhere in memory**

**You can communicate with devices by reading/writing some specific area of memory, usually in RAM.**

**In addition to RAM, there is I/O memory.**
You can think of it like a completely separate mini-RAM. The way you use it is the same as RAM. But instead of the ‘mov’ instruction, you use ‘in’ and ‘out’ to read/write.

I/O memory is often used to configure the hardware. Like telling the device which RAM address it should map to.

## SCREEN MEMORY MAP




Map of the screen from C64

To put a char on screen, move the value to the corresponding location in RAM.

Example:

```
MOV edi, 1024
MOV [edi], 13
```

Would display the character corresponding to the value 13 at the top left of the screen

**Though if you were actually using a C64, you would use BASIC command 'POKE'.

The slides & practice programs can be found on github

github.com/susikohmelo/assembly_basics_presentation