VRIJE UNIVERSITEIT BRUSSEL

Computersystemen

WPO: Exercise Session 1

David Blinder
Raees K. Muhamad

Introduction to the course

Overview

- WPO’s: Raees K. Muhamad, David Blinder

- Main goal: learn how to program in assembly, using the x86 instruction set
- Project: make a video game (on Canvas: “project.pdf”) **with 2 people**

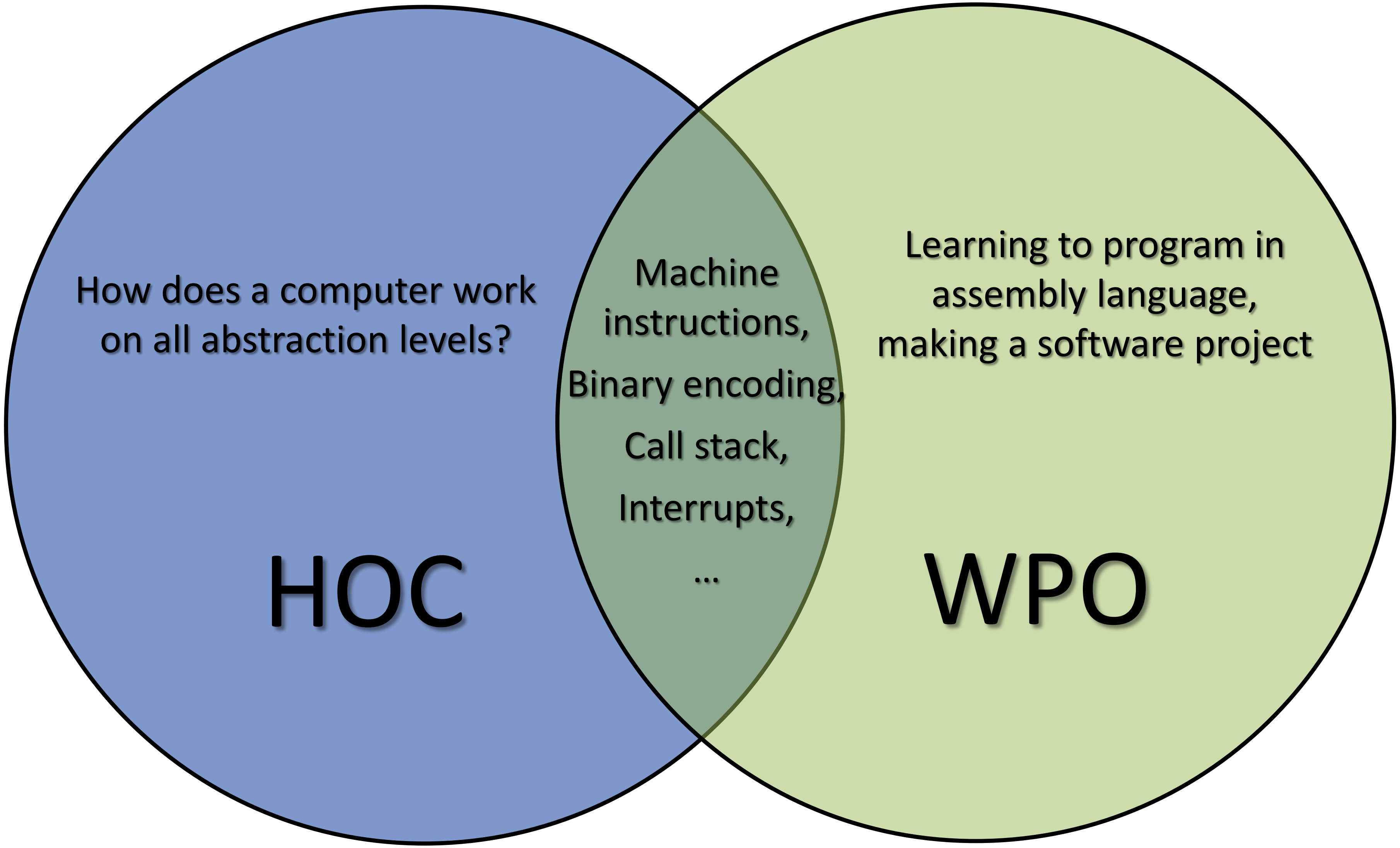
  → deadline: December 26th, 2020

- 2h WPO per week
- Exercise sessions until November
- Project follow-up

- Intermediate defense

Overview

Goals



Project

Tasks

- Goal: make an <u>interactive</u> program with <u>graphics</u> in 80386 x86 ASM.
  Examples: **Games**, physics simulation, (3D) graphics, paint app, interactive JPEG/MP3 encoder/decoder, etc.
- <u>Note</u>: no (clones of) **snake, pong, pacman, space invaders or tetris!**
- **Groups of 2 people**. Deliverables: report + source code. (see “project.pdf”)
- Since this is a programming project, you will primarily be evaluated on <u>code quality.</u>
- General metric: **efficiency** and **functionality**. (<u>min req</u>: it must run w/o errors)
  - (code efficiency) minimize redundancy, efficient use of instructions, no needless overhead
  - (algorithmic efficiency) low memory and computing requirements
  - (functionality) features of your game, complexity, game modes, AI, etc.

VRIJE UNIVERSITEIT BRUSSEL

Project

What (not) to do

- **Rule #1: no redundancy!** (copy-pasting in code is almost always a bad idea)
  - **Same rules that apply as for ‘higher’ programming languages.**
- You can use existing code, but **mention it clearly**
- Emphasis is on programming functionality rather than level content (e.g. procedurally generated levels > 100 hard-coded levels), for all team members

- Potential pitfalls:
  - @Engineers: remember to think about data structures and global program design. Once your code works, don’t leave it be, clean it up a make it more compact/efficient/… before proceeding
  - @Computer scientists: do not spend (too much) time on implementing high level abstractions (inheritance, virtual functions, …), focus on algorithmic efficiency and game functionality.

Assembly programming

What is **Assembly** Language?

- Native language of the machine
- Processor understands only machine code
  - Machine code is a sequence of one or more machine instructions
  - A machine instruction represents a single machine operation code (**opcode**) and its **operands** (some opcodes have no operands)
  - Opcodes and operands are represented as specific combinations of binary values (0’s & 1’s)

- Assembly Language helps writing this machine code via use of **mnemonics**
  - mnemonics are English-like words that map to the various machine instructions
- The Assembler tool converts such mnemonics into the 0’s and 1’s (or bytes) (and does much more)

Assembly Code vs **Machine** Code

Assembly Code	Machine Code (in base2 and base16)	
`mov ax, 1`	10111000 00000001 00000000	B8 01 00
`mov bx, ax`	10001011 11011000	8B D8
`mov ah, 9`	10110100 00001001	B4 09
`mov ax, 4c00h`	10111000 00000000 01001100	B8 00 4C
`int 21h`	11001101 00100001	CD 21

Note: It is perfectly possible to write machine code directly in binary code, but obviously, doing so would be needlessly difficult and error-prone. Hence, the use of an assembler and assembly code facilitate writing machine code.

Assembly Code vs **Machine** Code

Opcode and operand example

```
mov ax, 4c00h
```

Opcode	Operand 1	Operand 2
Move instruction	Destination	Source

Moves the 16-bit value 4c00h (=19456) into register AX.
The Assembler tool converts this to 3 bytes:

```
B8 00 4c
```

B8: mov ax

Relation to **high-level** programming

Opcode and operand example

- Most code is written in high-level languages, like Python, C++, Java, ...
- Typically, a compiler translates high-level code to machine code (possibly via intermediate assembler code)

- Example with C code:

C code

```c
int getSix() {


  int a = 1;
  a = a + 5;

  return a;
}
```

Assembler code

```
GETSIX PROC NEAR
  push bp
  mov  bp, sp
  mov  ax, 1
  add  ax, 5
  pop  bp
  ret
GETSIX ENDP
```

The Intel 80386 processor

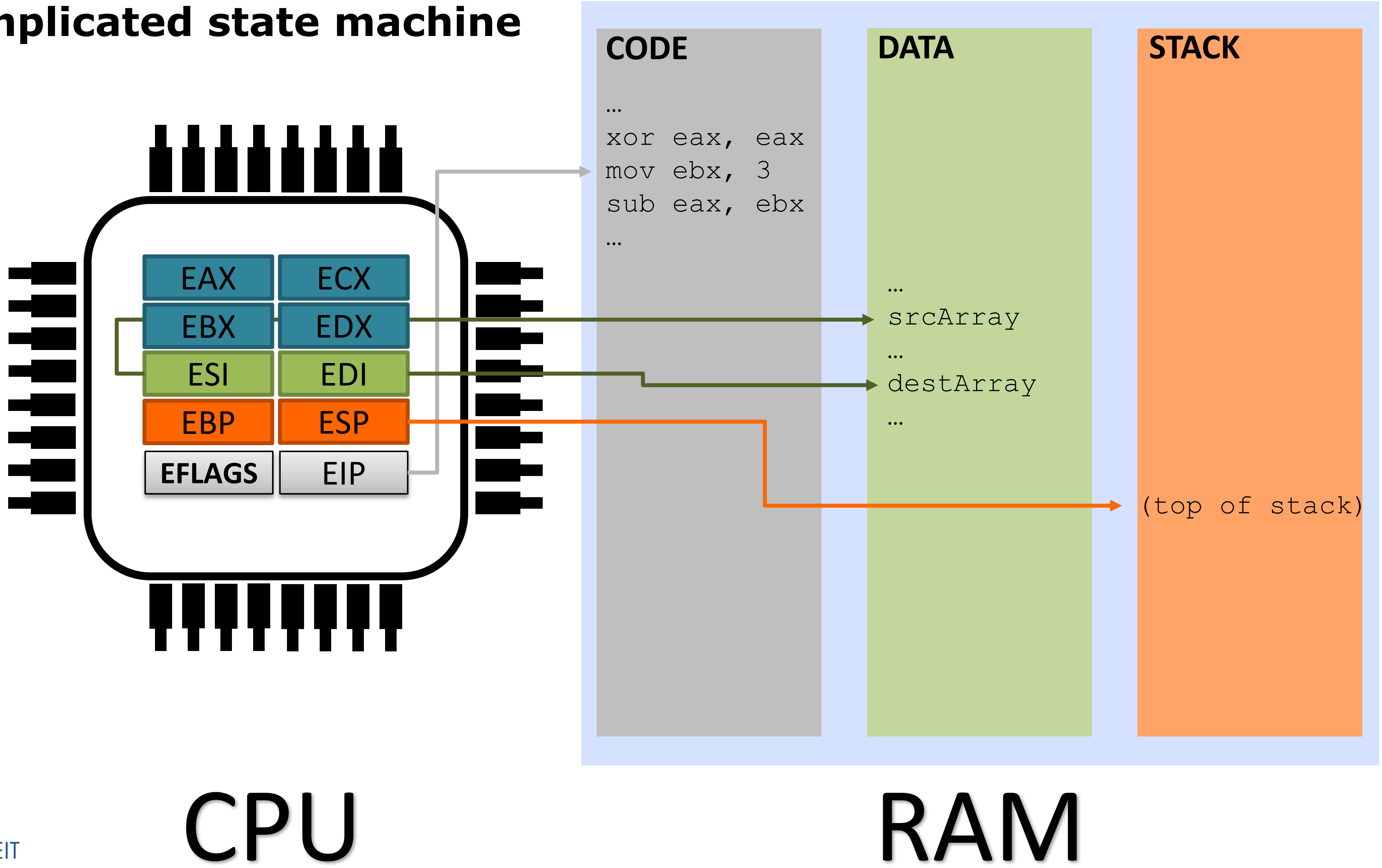
The Intel **80386**

One of the first 32-bit processors

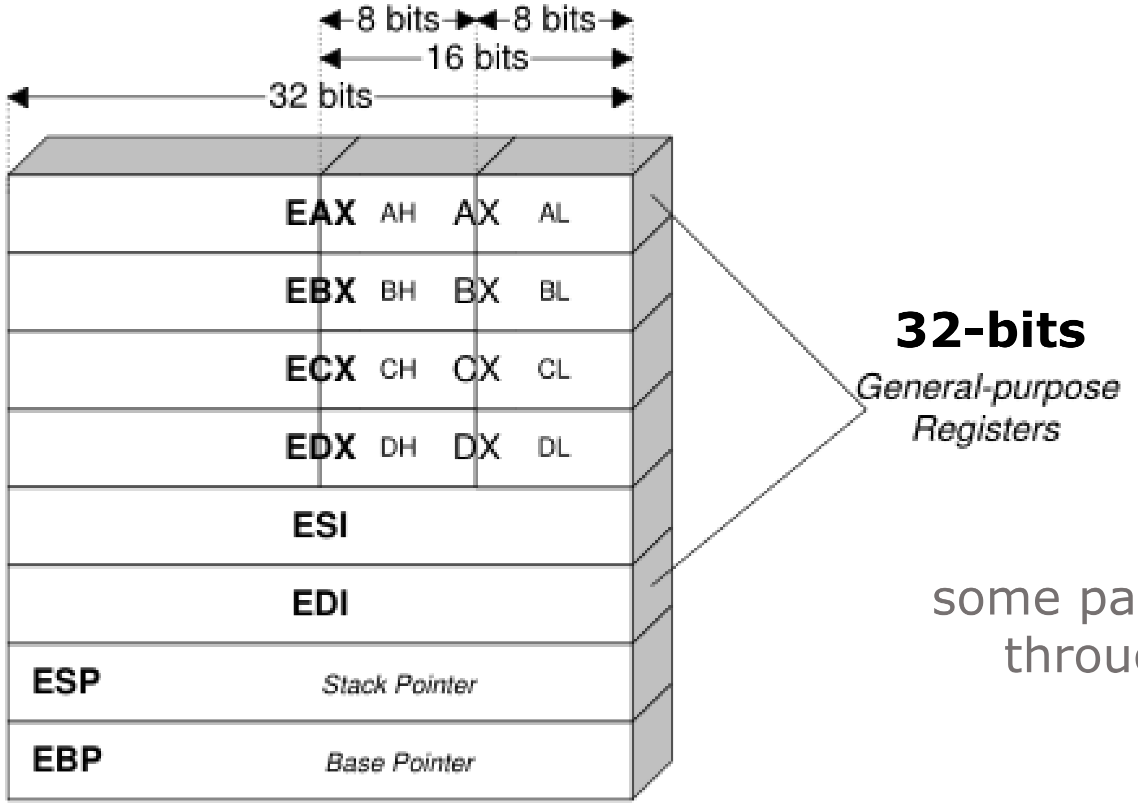
- Typically runs at 33Mhz
- Supports up to 4GB memory address space
- Floating point instructions are provided by the (optional) **80387** co-processor

High-level diagram of a CPU + memory mapping

A complicated state machine



The Intel **80386** registers



EIP, **EFLAGS** (read-only)

some parts are accessible through 16-bit and 8-bit parts

Instruction types

- Data Movement instructions
- Arithmetic & Logic instructions
- (un)conditional Flow instructions (i.e. Branching)
- (Interrupts)

Data movement instructions

Moving data between registers and between registers and memory

00000000 00000000 00000000 00000101

```
mov eax, 5                  ; move constant 5 to eax
mov ebx, eax                ; copy register eax to ebx
mov ecx, [var_x]            ; copy content of var_x to ecx
mov edx, offset var_x       ; copy address of var_x to edx

DATASEG:
      var_x dd 01234567h
```

Remember: "mov ax, 15" will affect the value in "eax", they cover the same register bits!

Arithmetic and logic instructions

Almost all calculations can only be applied on registers

```
add eax, 4           ; add 4 to eax
sub ebx, eax         ; subtract eax from ebx
xor ecx, ecx         ; xor ecx with itself (ecx=0)
imul ecx, eax        ; signed multiply of ecx with eax
add eax, [2*ebp+4]   ; (pointer arithmetic)
shl eax, 3           ; bitshift left (3 bits)
sar ebx, 2           ; arithmetic bitshift right
```

Conditional Flow instructions

Jump (un)conditionally based on **EFLAGS** register.

Some important examples:

- Bit 0 CF : Carry Flag
- Bit 2 PF : Parity Flag
- Bit 6 ZF : Zero Flag.
- Bit 7 SF : Sign Flag.
- Bit 9 IF : Interruption Flag (set by **sti** instruction)
- Bit 10 DF : Direction Flag (cleared by **cld** instruction)
- Bit 11 OF : Overflow Flag

VRIJE UNIVERSITEIT BRUSSEL

Conditional Flow instructions

Jump (un)conditionally based on **EFLAGS** register.

```
jlabel1:         ; label (doesn't create any machine code)
jmp jlabel2      ; jump unconditionally
jlabel2:
cmp eax, ebx     ; (sub), but eax unchanged
jge jlabel1      ; jump if greater or equal
test ecx, edx    ; (and), but ecx unchanged
jl jlabel1       ; jump if smaller (less than)
sub ecx, 5
jz jlabel2       ; jump if zero; same as (je)
jc jlabel2       ; jump if carry
```

Interrupts

Saves state, executes depending typically on codewords in (parts of) **eax.**

Examples:

- int 21h (if AH = 09h) → prints string
- int 21h (if AH = 02h) → print character
- int 21h (if AX = 4C00h) → terminate program
- int 10h (if AH = 03h) → VGA graphics mode
- int 16h (if AH = 01h) → test keyboard press

These are mostly OS-dependent software routines,
the chosen numerical interrupt values have no real "meaning"

Exercises

Exercises

1. Write a simple "Hello World!" program. Use function `09h` (in `AH`) of `int 21h`.
   (Hint: look in C:\EXERCISES\HELLO)
   - Reference: http://stanislavs.org/helppc/int_21-9.html.

2. Write an if-then-else construct. Print to the screen a message that depends on the value of `EAX`, which can 0, 1 or neither.
   - Reference: https://en.wikibooks.org/wiki/X86_Disassembly/Branches

3. Write a program that prints 10 times "HelloWorld!". Make use of branching and the `ECX` register for counting.
   - Reference: https://en.wikibooks.org/wiki/X86_Disassembly/Loops

Exercises

4. Draw a pyramid of '*' symbols, given a height in `ebx`:

```
    *
   ***
  *****     for height = 5
 *******
*********
```

Use function `02h` (in `AH`) of `int 21h` to print a single symbol (you can print a newline with the two successive symbols `0Dh` and `0Ah`)

Hint: you can directly use a character symbol: **`mov dl,'*'`**
Reference: http://stanislavs.org/helppc/int_21-2.html

VRIJE UNIVERSITEIT BRUSSEL