VRIJE UNIVERSITEIT BRUSSEL

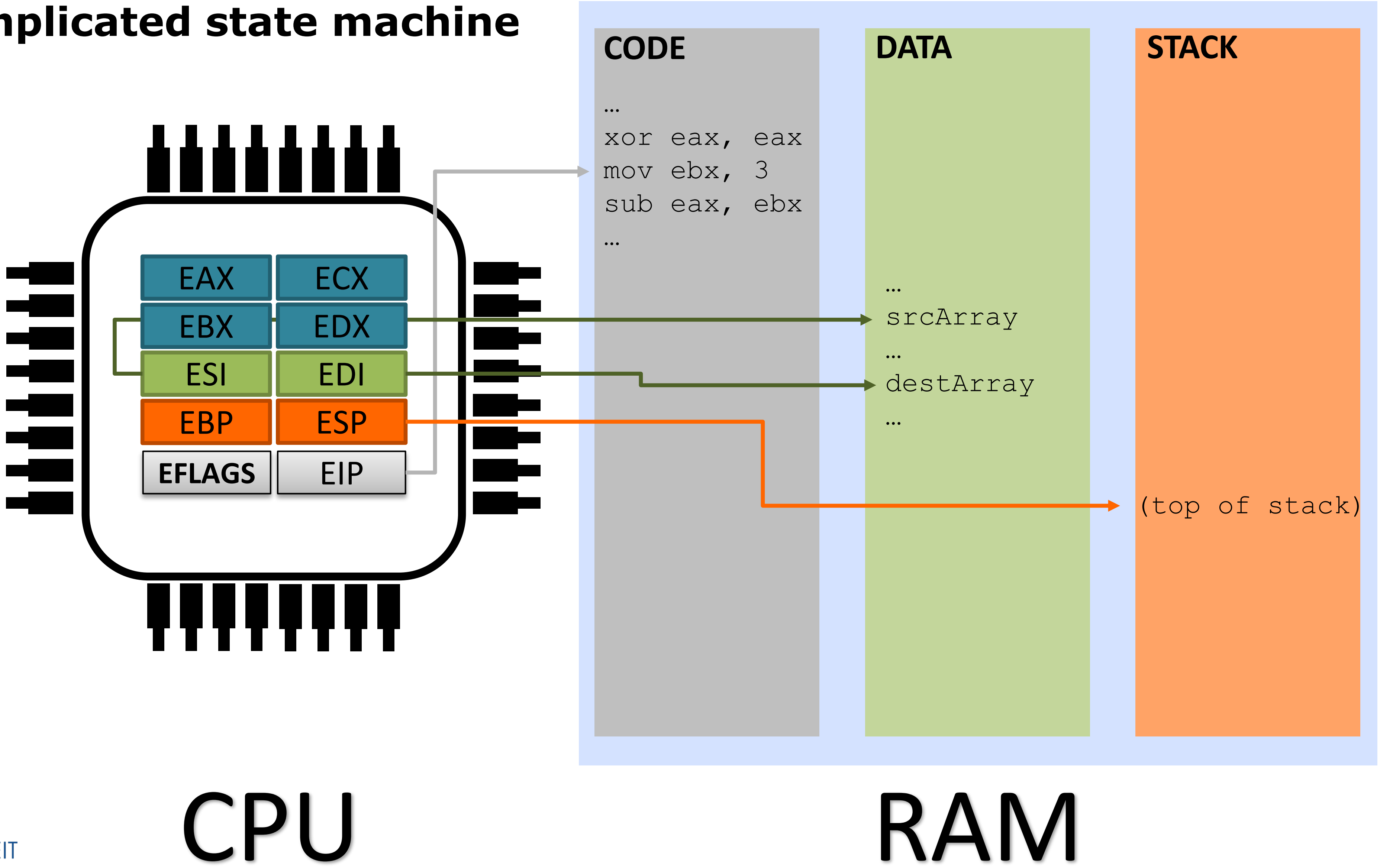
Computersystemen

WPO: Exercise Session 2

David Blinder
Raees K. Muhamad

High-level diagram of a CPU + memory mapping

A complicated state machine



The Stack

The Stack

The stack is a memory segment which is accessed as a LIFO data structure. The top of the stack is pointed to by the register ESP.

With the push instruction, you can push registers (or variables in memory) to the stack segment. This will decrement the stack pointer.

Conversely pop will load elements from the top of stack and increment the stack pointer.

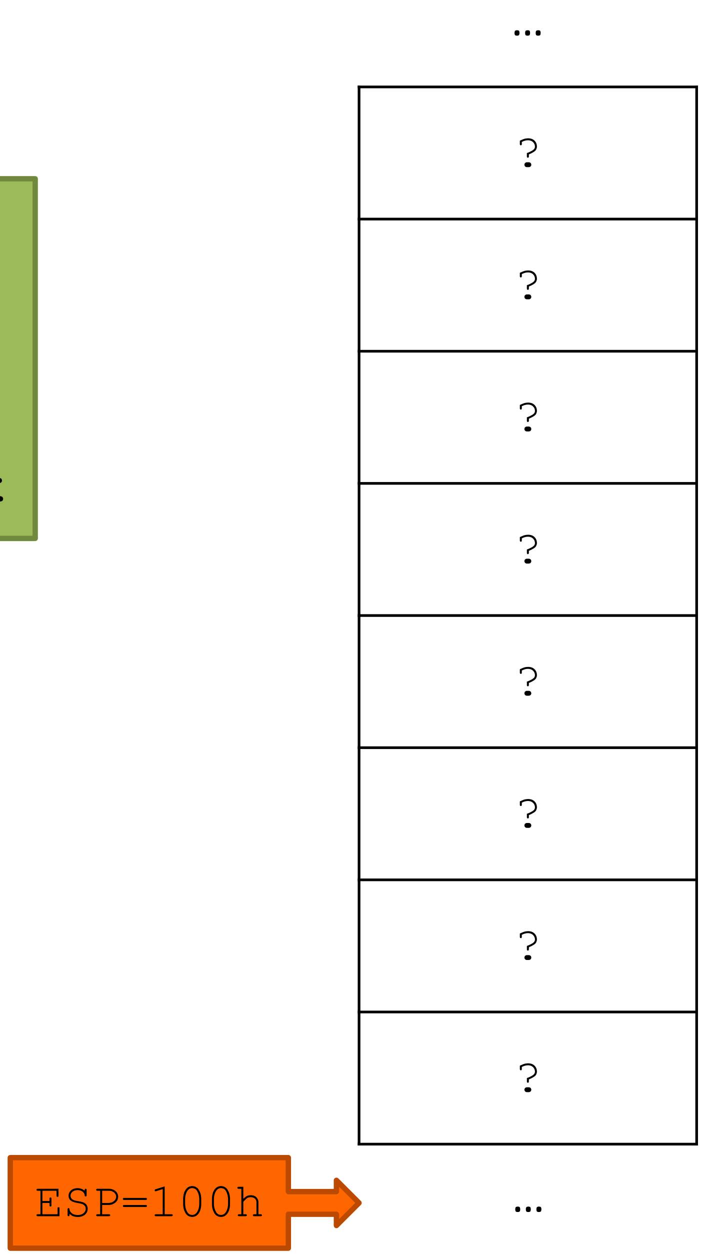
Note: you can push either 16-bit or 32-bit elements, but please be careful when mixing both!

The Stack

```
CODESEG
xor ebx, ebx
push ebx
push [var1]
push [var2]
pop  eax
...
DATASEG
var1 dd 01234567h
var2 dd 089ABCDEh

STACK 100h
```

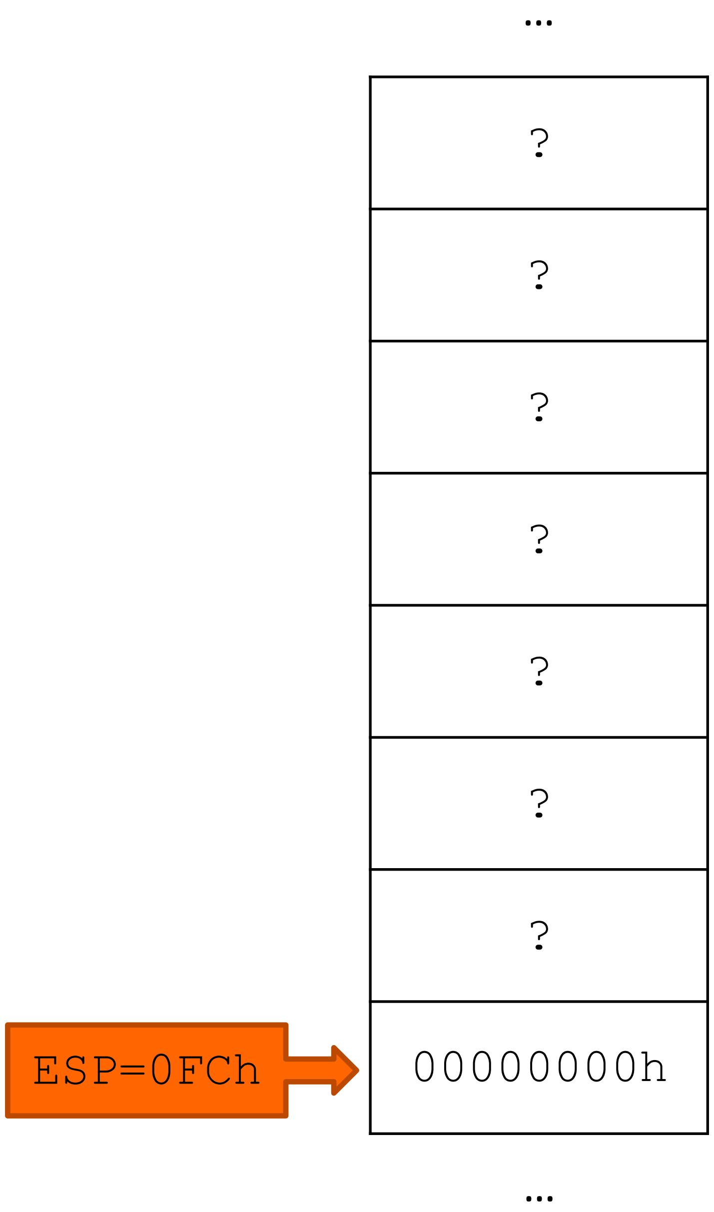
push ebx is equivalent to:

sub esp, 4
mov [esp], ebx



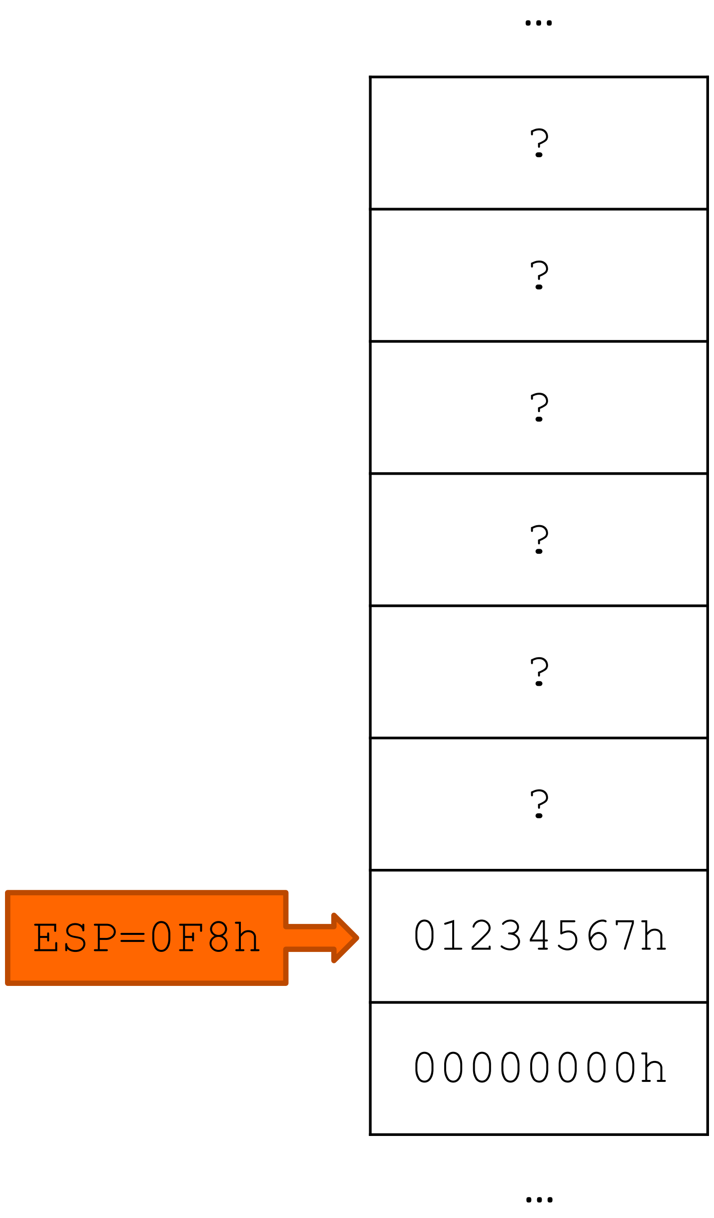
The Stack

```
CODESEG
xor ebx, ebx
push ebx
push [var1]
push [var2]
pop  eax
...
DATASEG
var1 dd 01234567h
var2 dd 089ABCDEh

STACK 100h
```



The Stack

```
CODESEG
xor ebx, ebx
push ebx
push [var1]
push [var2]
pop  eax
...
DATASEG
var1 dd 01234567h
var2 dd 089ABCDEh

STACK 100h
```



The Stack

```
CODESEG
xor ebx, ebx
push ebx
push [var1]
push [var2]
pop  eax
...
DATASEG
var1 dd 01234567h
var2 dd 089ABCDEh

STACK 100h
```

pop eax is equivalent to:

```
mov eax, [esp]
add esp, 4
```

ESP=0F4h

…
?
?
?
?
?
089ABCDEh
01234567h
00000000h
…

VRIJE UNIVERSITEIT BRUSSEL

The Stack

```
CODESEG
xor ebx, ebx
push ebx
push [var1]
push [var2]
pop  eax
...
DATASEG
var1 dd 01234567h
var2 dd 089ABCDEh

STACK 100h
```

EAX = 089ABCDEh

...
?
?
?
?
?
089ABCDEh
01234567h
00000000h
...

ESP=0F8h

The Stack
Examples

Preserving register values

```
push eax
push edx
mov ah, 09h
mov edx, offset msg
int 21h
pop edx
pop eax
```

Local procedure variables

…

Nested loops

```
outerloop:
push ecx
…
mov ecx, 20 ;overwrite
innerloop:
…
loop innerloop
pop ecx
loop outerloop
```

VRIJE UNIVERSITEIT BRUSSEL

Multiplication and Division

Examples

Multiplication (MUL) is done with implied operand `EAX`. The 64-bit result is stored jointly in `EDX:EAX`. (use IMUL for signed division)

```
mov eax, 8   ; set EAX to 8
mul ebx      ; multiply EAX by EBX, store in EDX:EAX
imul ecx     ; signed multiply of EAX by ECX
```

Division (DIV) is with the implied operand `EDX:EAX`. Stores result in `EAX` and modulo in `EDX`. (use IDIV for signed division). If quotient doesn't fit in (32-bit) `EAX` → ERROR!

```
mov eax, 246 ; set EAX to 246
xor edx, edx ; set EDX to zero (so EDX:EAX = EAX)
div ebx       ; divide EDX:EAX by EBX: EAX=246/EBX, EDX=246%EBX
```

For efficient multiplication and division by powers of two, use bit-shifting instead!

```
sal eax, 5   ; equiv. to signed multiply by 2^5 = 32
sar ebx, 3   ; equiv. to signed division by 2^3 = 8
```

Exercises

Exercises

1. Write a program that takes a positive 32-bit number in EAX and prints it to the screen in decimal notation.

→ print the digits out character-by-character with `int 21, AH=02h`

2. Write a program that takes a **two's complement** 32-bit number in EAX and prints it to the screen in decimal notation.
   (Use the code from exercise 1)

Exercises

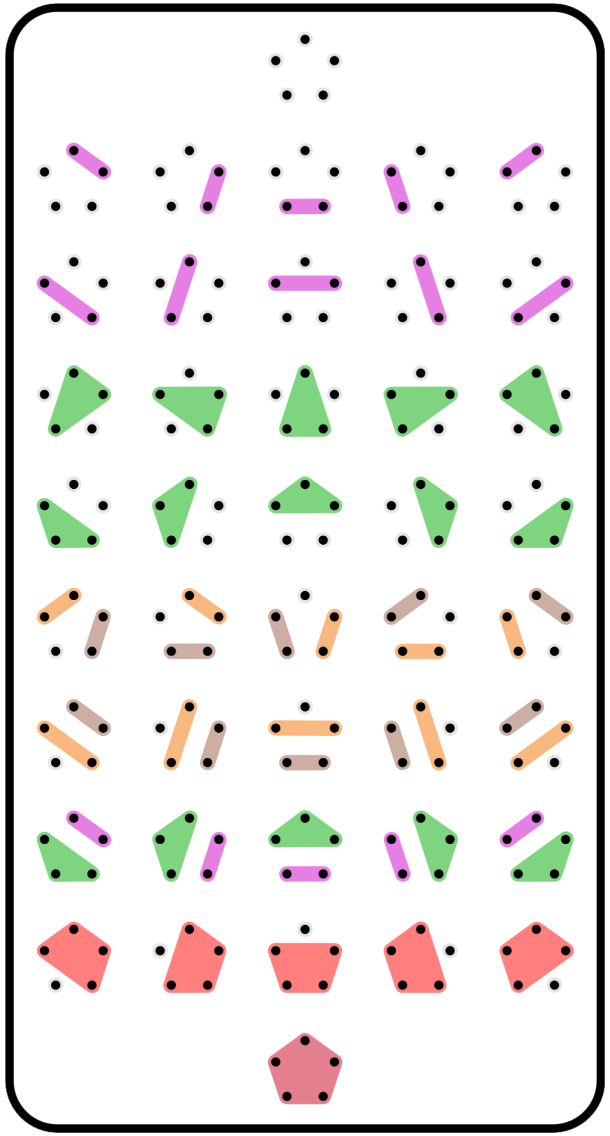
3. Write a program that prints out the nth Catalan number. The Catalan numbers C_n are a sequence of natural numbers indexed by n.
They have various applications in mathematics:

- # of possible binary search trees with n different keys,
- # of ways a convex polygon can be triangulated
- # of noncrossing partitions of an *n*-element set (see figure)
- …

They can be computed with the formula, $\forall n \in \mathbb{N}$:

$$C_0 = 1;\ C_{n+1} = \sum_{i=0}^{n} C_i \cdot C_{n-i}$$

The Catalan numbers are, starting from $C_0,\ C_1,\ C_2, \ldots$
1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, …



The $C_5 = 42$ noncrossing partitions of a 5-element set