VRIJE UNIVERSITEIT BRUSSEL

Computersystemen

WPO: Exercise Session 3

David Blinder
Raees K. Muhamad

Procedures

How can we program procedures in x86 assembly?

- The x86 register set only provides us with a few bytes of memory
  → **this does not suffice beyond simple use cases**

- Using the data segment for parameter passing is unwieldy and error-prone; it means all variables in your program are global/static, which makes e.g. recursion nearly impossible → **impractical**

Procedure calling will be done via the stack, using the CDECL convention.

The C-declaration convention

C Declaration

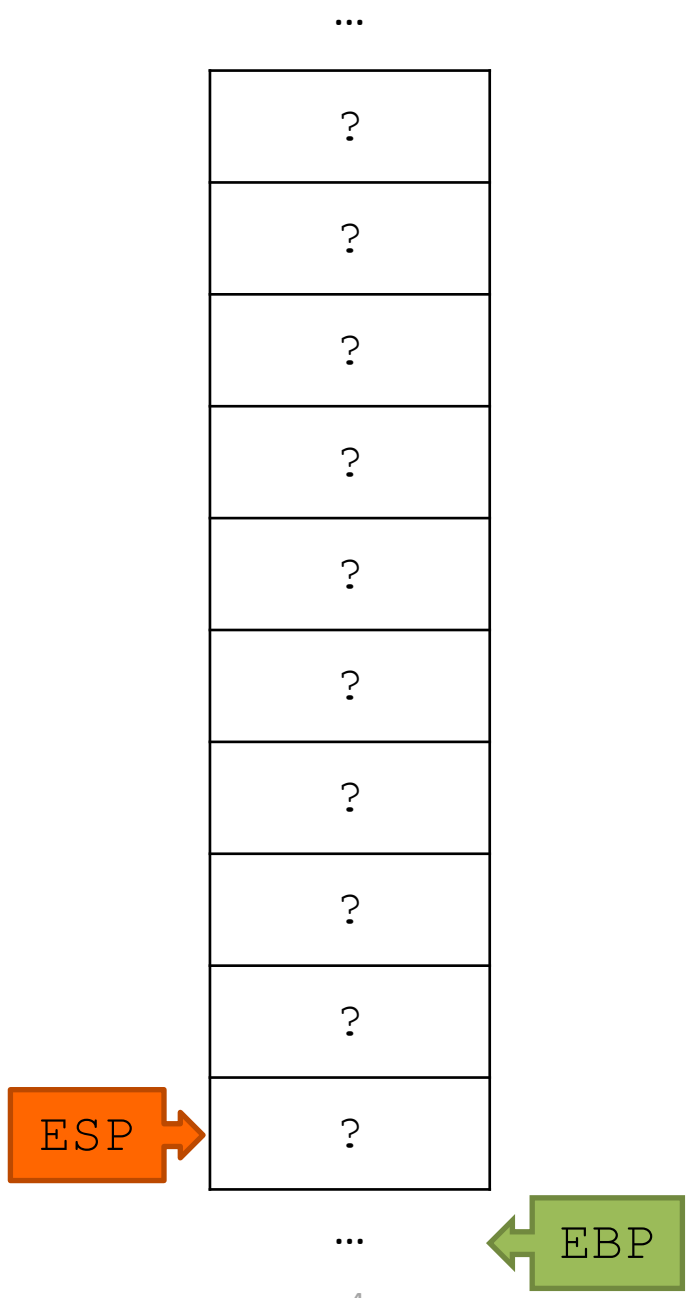
We are going to consider an example procedure:

```
int MyProc(int arg1, int arg2)
```

The stack pointer = `ESP`

The base pointer `EBP` points to a specific "base address" from which function input/output parameters can be retrieved from a fixed offset.

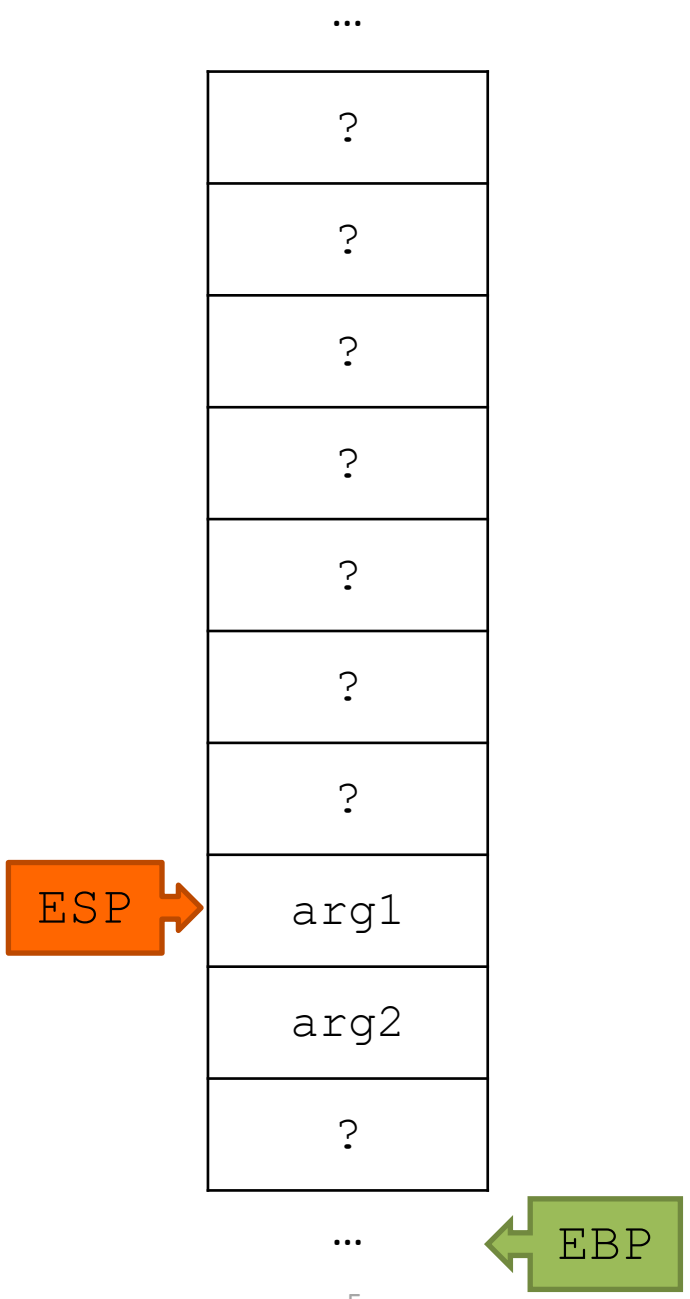
It now points to the base of the current procedure calling `MyProc`.



C Declaration

```
int MyProc(int arg1, int arg2)
```

First, the input argments are pushed on the stack in reverse order by the **caller** (before calling `MyProc`)



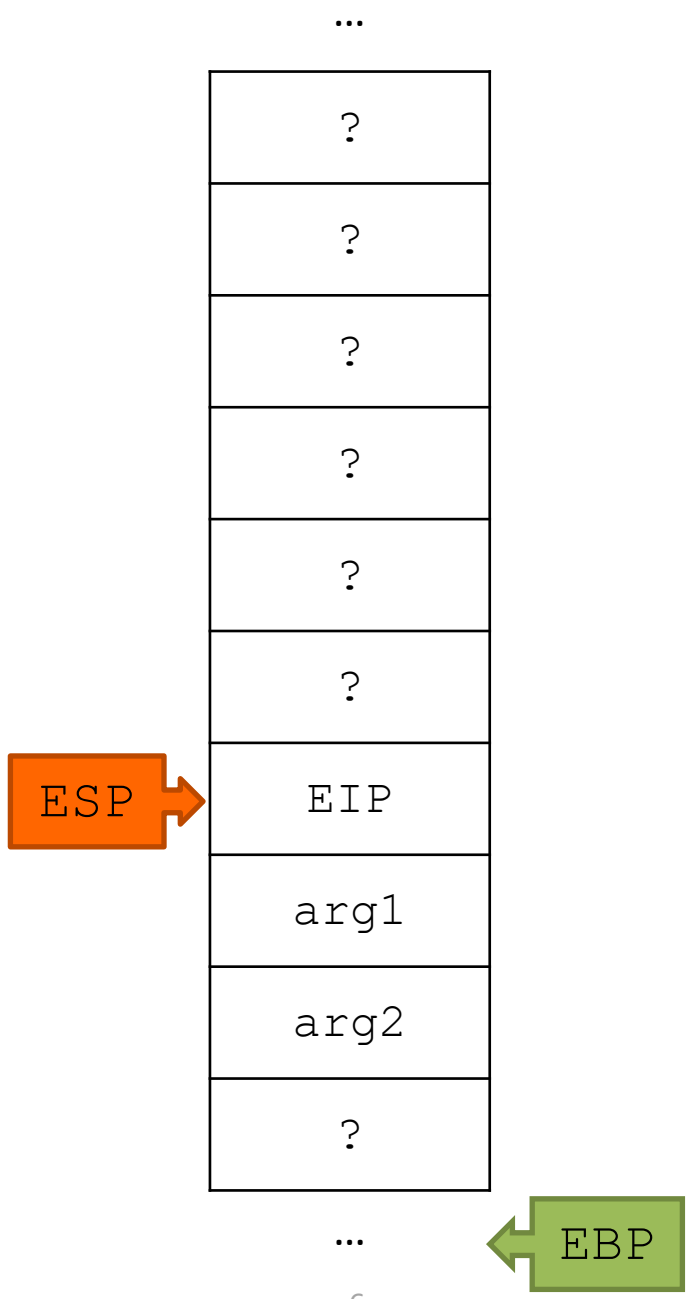
C Declaration

```
int MyProc(int arg1, int arg2)
```

Then, we invoke the instruction:

```
CALL MyProc
```

This will push the current instruction pointer register value `EIP` that will be used for returning to the code segment address after the procedure terminates.



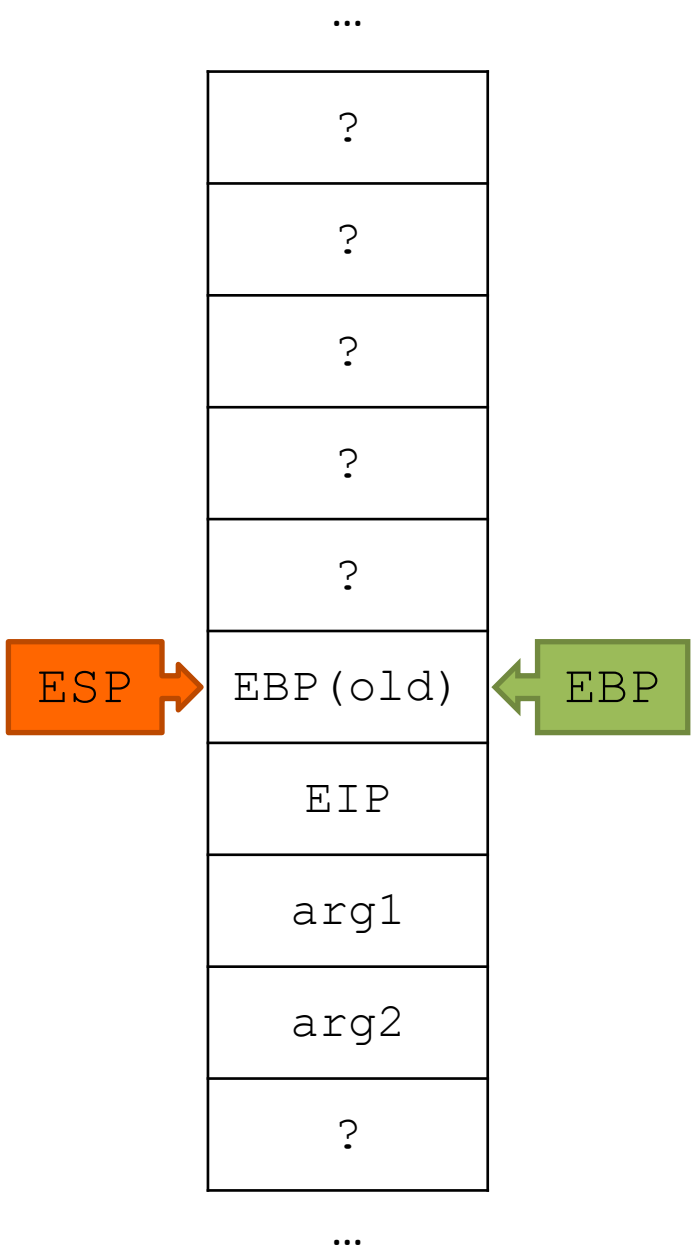
C Declaration

```
int MyProc(int arg1, int arg2)
```

The old base pointer is stored, and is `EBP` updated

```
push EBP
mov EBP, ESP
```

Afterwards, the old value of `EBP` can be restored.



C Declaration

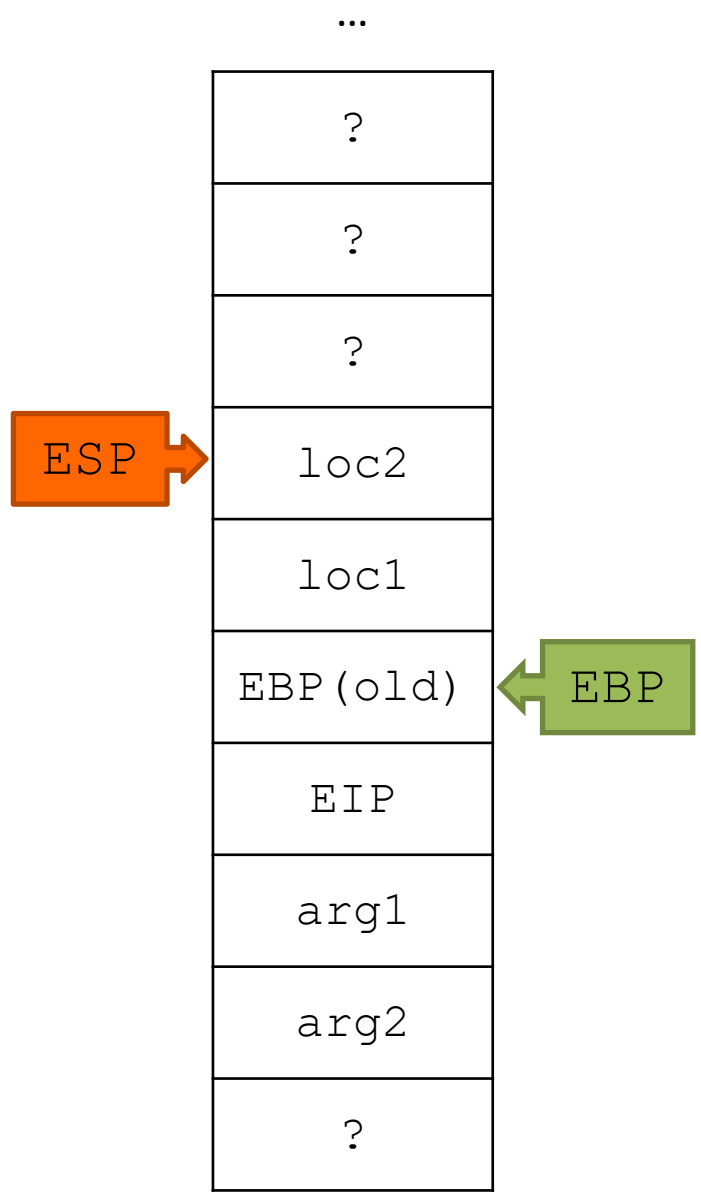
```
int MyProc(int arg1, int arg2)
```

When insufficient registers* are available, local variables of `MyProc` will be stored on the stack next

They are accessed via `EBP`:

```
mov EAX, [EBP-4]
mov EBX, [EBP-8]
```

The input arguments `arg1,arg2` are accessible via `[EBP+8],[EBP+12]`



*Prefer utilizing (generic) registers over memory accesses for performance whenever possible.

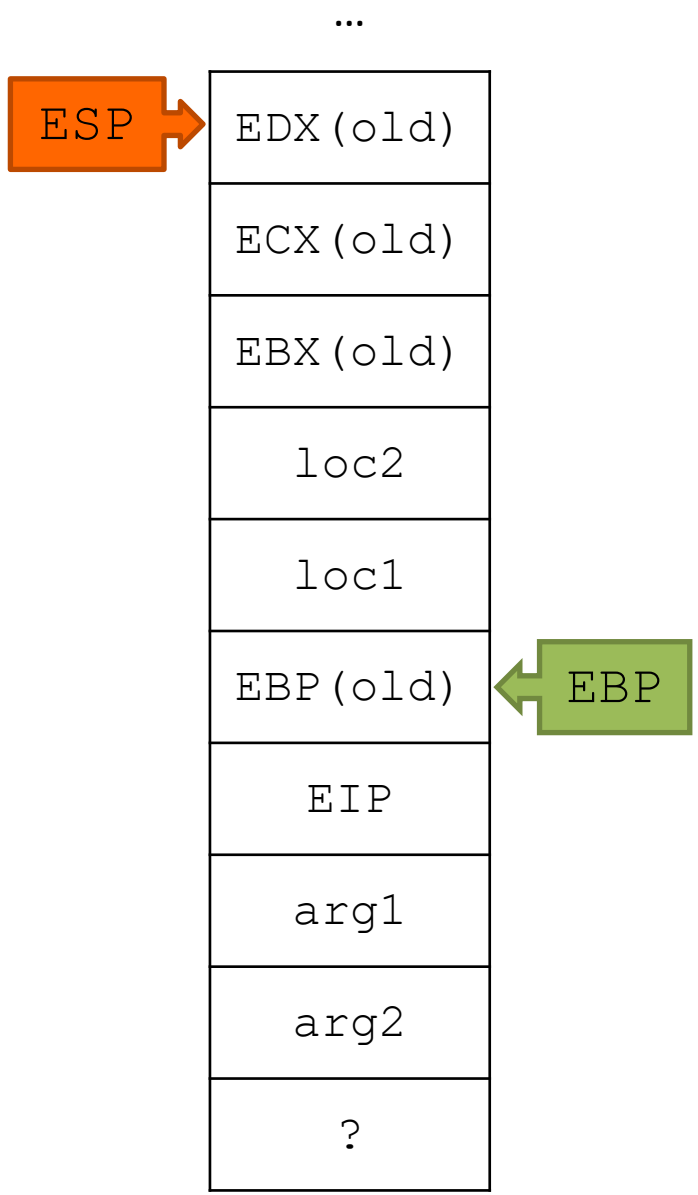
C Declaration

```
int MyProc(int arg1, int arg2)
```

Finally, used registers from the **caller** need to be stored as well. They are pushed on the stack before use, to be restored after the procedure finishes

In this example, we assume we need to preserve `EBX`, `ECX` and `EDX`.

Return values (for an integer, pointer) are returned in `EAX` by convention. Otherwise, the stack can be used as well.



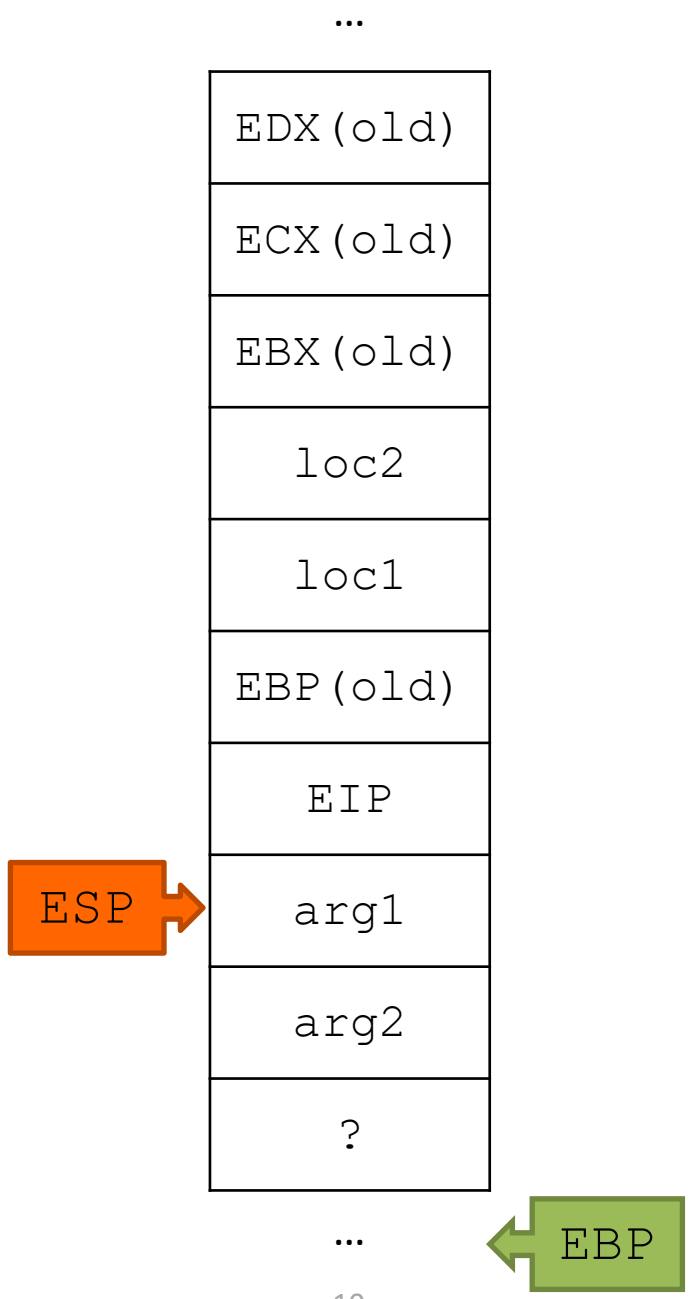
C Declaration

```
int MyProc(int arg1, int arg2)
```

Once finished, the old `EBX`, `ECX`, `EDX` are restored by popping them back.

`EBP`, `ESP` are restored too, and it returns from `MyProc` with:

```
mov ESP, EBP
pop EBP
ret
```



TASM IDEAL syntax

IDEAL Syntax

It can be tedious and error-prone to write correct procedure prologue code. That is why many assembler tools provide some help.

In this course, we rely on the IDEAL syntax of TASM. You are expected to know and understand the underlying principle of stack-based calling conventions! Questions will be asked during the exams.

IDEAL Syntax

```
PROC exampleProcedure
  ARG @@arg1:dword, @@arg2:dword RETURNS eax
  LOCAL @@var_x:word, @@var_y:byte
  USES ebx, ecx, edx

  mov ebx, [@@arg1]
  …
  ret
ENDP exampleProcedure
…

call exampleProcedure, offset data1, edx
```

VRIJE UNIVERSITEIT BRUSSEL

IDEAL Syntax

Input argument(s)

Optional return argument

```
PROC exampleProcedure
  ARG @@arg1:dword, @@arg2:dword RETURNS eax
  LOCAL @@var_x:word, @@var_y:byte
  USES ebx, ecx, edx

  mov ebx, [@@arg1]
  …
  ret
ENDP exampleProcedure
…
```

Local variables

Registers to be preserved

Use brackets to access content instead of pointer

Calling with input arguments

```
call exampleProcedure, offset data1, edx
```

size keywords

byte	8 bits
word	16 bits
dword	32 bits

Exercises

Exercises

Note: **download the template** for these exercises

1. Modify the unsigned number printing procedure to make full use of the IDEAL syntax. The number to be printed should be given via an input argument (not via a register).
2. Make a procedure that prints out an array of (32-bit) unsigned integers separated by commas, given the array offset as an input. The first value contains the array length (`arrlen`), followed by the array data itself (`arrdata`). They are guaranteed to be contiguous in memory.
3. Make a procedure that returns the maximum of array in `EAX`.
4. Program a bubble sort procedure that sorts an array in-place.
   Reference: `https://en.wikipedia.org/wiki/Bubble_sort`.