Basic Assembly

Calling Conventions

Assembly language programming
By xorpd

xorpd.net

Objectives

- We explore different methods for communicating with functions.
- We present conventions for communicating with functions.

Passing arguments

- Arguments are chunks of information that we pass into a function as input.
- So far we used registers to pass arguments, and we used registers to pass the result.
- We want to explore some different ways of passing arguments.

Method 1: Registers

- Values are passed on some of the registers:

```
    mov     ecx,5     ; argument
    mov     edx,2     ; argument
    call    my_func
```

```
my_func:
    mov     eax,ecx
    sub     eax,edx
    ret
```

- Sometimes referred to as FASTCALL.
- Very common in 64 bit long mode.
  - There are more registers.

Method 2: Global data

- Values are passed through a global memory location:

```
section '.bss' readable writeable
    arg1        dd      ?
    arg2        dd      ?
```

```
    mov     dword [arg1],5      ; argument
    mov     dword [arg2],2      ; argument
    call    my_func
```

```
my_func:
    mov     eax,dword [arg1]
    sub     eax,dword [arg2]
    ret
```

- Ugly, but works.

Method 3: The stack

- We pass arguments over the stack:

```
    push    5       ; argument
    push    2       ; argument
    call    my_func
    add     esp,8 ; clean stack.
```

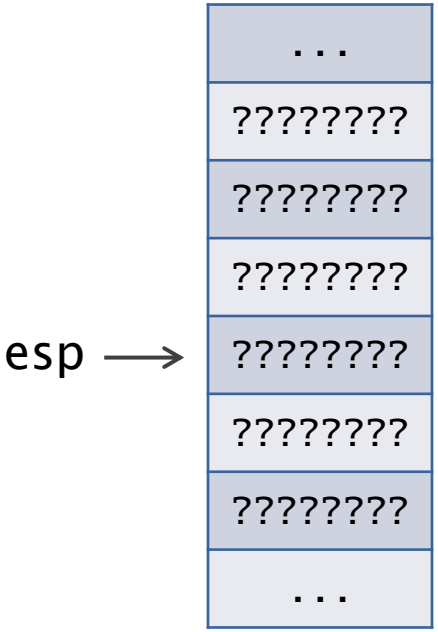
```
my_func:
    mov     eax,dword [esp + 8]
    sub     eax,dword [esp + 4]
    ret
```

Method 3: The stack

- We pass arguments over the stack:

```
push    5       ; argument
push    2       ; argument
call    my_func
add     esp,8 ; clean stack.
```

```
my_func:
    mov     eax,dword [esp + 8]
    sub     eax,dword [esp + 4]
    ret
```

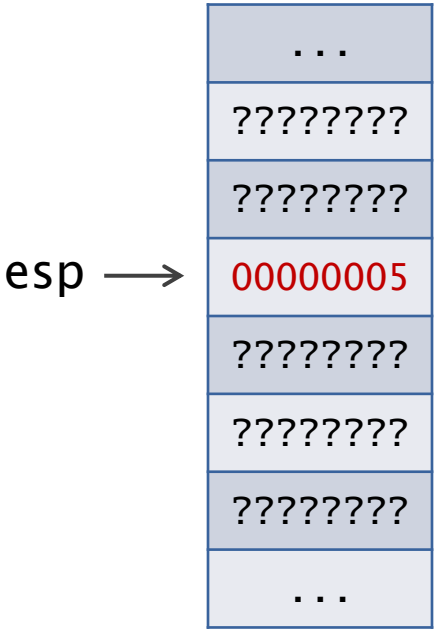


Method 3: The stack

- We pass arguments over the stack:

```
    push    5       ; argument
→   push    2       ; argument
    call    my_func
    add     esp,8 ; clean stack.
```

```
my_func:
    mov     eax,dword [esp + 8]
    sub     eax,dword [esp + 4]
    ret
```

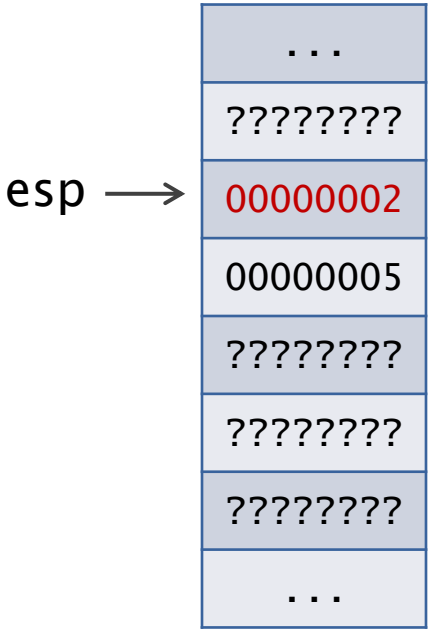


Method 3: The stack

- We pass arguments over the stack:

```
    push    5       ; argument
    push    2       ; argument
→   call    my_func
    add     esp,8 ; clean stack.
```

```
my_func:
    mov     eax,dword [esp + 8]
    sub     eax,dword [esp + 4]
    ret
```

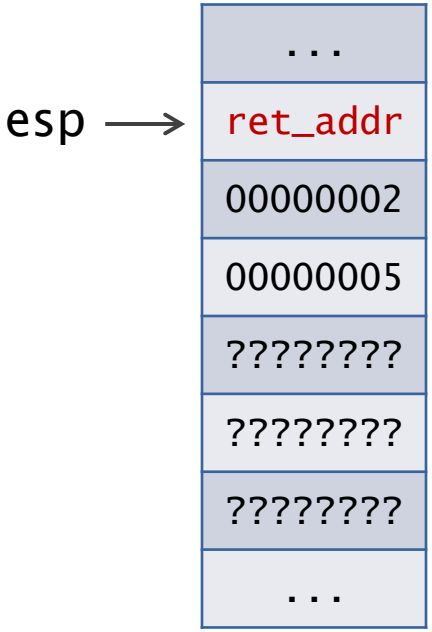


Method 3: The stack

- We pass arguments over the stack:

```
    push    5       ; argument
    push    2       ; argument
    call    my_func
    add     esp,8 ; clean stack.
```

```
my_func:
    mov     eax,dword [esp + 8]
    sub     eax,dword [esp + 4]
    ret
```



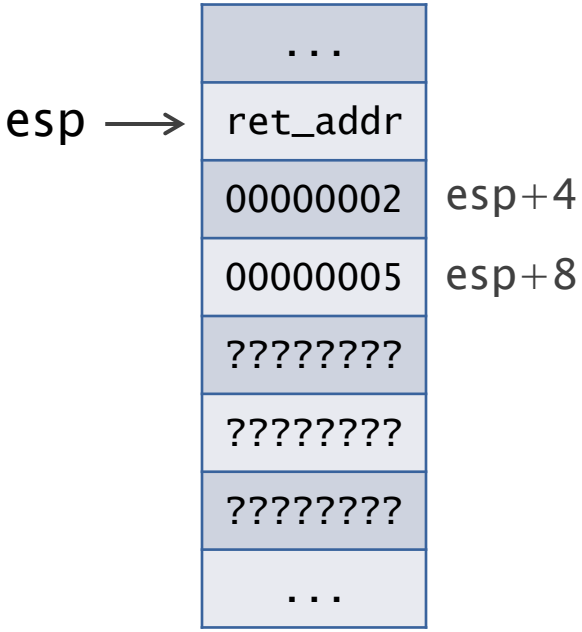
Method 3: The stack

- We pass arguments over the stack:

```
    push    5       ; argument
    push    2       ; argument
    call    my_func
    add     esp,8 ; clean stack.
```

```
my_func:
→   mov     eax,dword [esp + 8]
    sub     eax,dword [esp + 4]
    ret
```

eax	????????



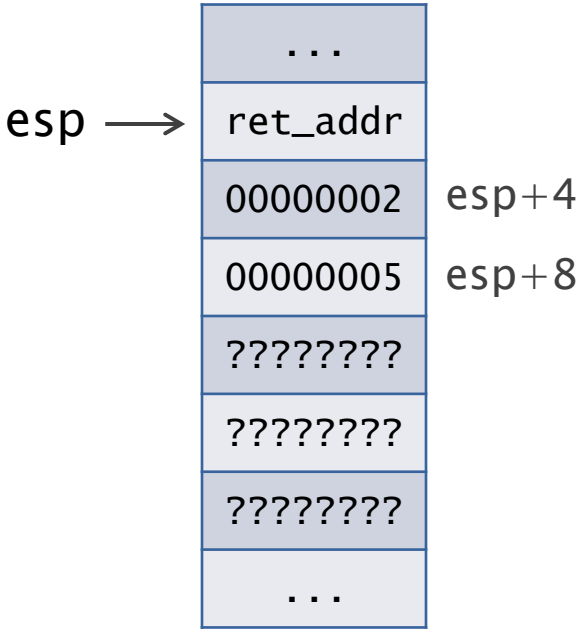
Method 3: The stack

- We pass arguments over the stack:

```
    push    5       ; argument
    push    2       ; argument
    call    my_func
    add     esp,8 ; clean stack.
```

```
my_func:
    mov     eax,dword [esp + 8]
→   sub     eax,dword [esp + 4]
    ret
```

eax	00000005



Method 3: The stack

- We pass arguments over the stack:

```
    push    5       ; argument
    push    2       ; argument
    call    my_func
    add     esp,8 ; clean stack.
```

```
my_func:
    mov     eax,dword [esp + 8]
    sub     eax,dword [esp + 4]
→   ret
```

eax	00000003

	...
esp →	ret_addr
	00000002
	00000005
	????????
	????????
	????????
	...

Method 3: The stack

- We pass arguments over the stack:

```
    push    5       ; argument
    push    2       ; argument
    call    my_func
→   add     esp,8 ; clean stack.
```

```
my_func:
    mov     eax,dword [esp + 8]
    sub     eax,dword [esp + 4]
    ret
```

eax	00000003

	...
	ret_addr
esp →	00000002
	00000005
	????????
	????????
	????????
	...

Method 3: The stack

- We pass arguments over the stack:

```
    push    5       ; argument
    push    2       ; argument
    call    my_func
    add     esp,8 ; clean stack.
→
```

```
my_func:
    mov     eax,dword [esp + 8]
    sub     eax,dword [esp + 4]
    ret
```

eax	00000003

	...
	ret_addr
	00000002
	00000005
esp →	????????
	????????
	????????
	...

Calling Conventions

- Every function has an interface with the external world
  - Input, Output.
- We want to be able to call other people's functions (And vice versa)
  - Maybe written in a different language?
  - Maybe compiled using a different compiler?
- Assuming that we chose the **stack** to pass arguments, there are some more decisions to be made:
  - Who cleans the stack? (Caller or Callee)
  - How to pass output from the function?
  - …

Cleaning the stack

- Who should clean the stack? Caller or callee?

Caller

```
    push    5       ; argument
    push    2       ; argument
    call    my_func
    add     esp,8 ; clean stack.
```

```
my_func:
    mov     eax,dword [esp + 8]
    sub     eax,dword [esp + 4]
    ret
```

Callee

```
    push    5       ; argument
    push    2       ; argument
    call    my_func
```

```
my_func:
    mov     eax,dword [esp + 8]
    sub     eax,dword [esp + 4]
    ret     8       ; clean stack.
```

Cleaning the stack

- Who should clean the stack? Caller or callee?

Caller

```
    push    5       ; argument
    push    2       ; argument
    call    my_func
    add     esp,8 ; clean stack.
```

```
my_func:
    mov     eax,dword [esp + 8]
    sub     eax,dword [esp + 4]
    ret
```

Callee

```
    push    5       ; argument
    push    2       ; argument
    call    my_func
```

```
my_func:
    mov     eax,dword [esp + 8]
    sub     eax,dword [esp + 4]
    ret     8       ; clean stack.
```

- Pop dword x from stack.
- $eip \leftarrow x$
- Increase esp by 8

Cleaning the stack

- Who should clean the stack? Caller or callee?

Caller

```
    push    5       ; argument
    push    2       ; argument
    call    my_func
    add     esp,8 ; clean stack.
```

```
my_func:
    mov     eax,dword [esp + 8]
    sub     eax,dword [esp + 4]
    ret
```

CDECL
The C language

Callee

```
    push    5       ; argument
    push    2       ; argument
    call    my_func
```

```
my_func:
    mov     eax,dword [esp + 8]
    sub     eax,dword [esp + 4]
    ret     8       ; clean stack.
```

STDCALL
Microsoft API

Return value

- The output of a function is also called the “return value”.
- Both CDECL and STDCALL conventions require that functions return value in EAX.

```
my_func:
    mov     eax,dword [esp + 8]
    sub     eax,dword [esp + 4]
    ret
```

“Order” of arguments

- In higher level languages, function arguments are sometimes said to have **order.**
  - First argument, second argument etc.

- With the CDECL and STDCALL conventions, the last pushed value is the “first” argument.

```
push   1      ; (3) Third argument
push   9      ; (2) Second argument
push   2      ; (1) First argument
call   some_func
add    esp,0ch ; clean stack.
```

- some_func(2,9,1)

Summary

- Three methods for passing arguments:
  - Registers.
  - Global memory.
  - **The Stack.**

- Calling conventions help connect different pieces of code.

- Two major calling conventions using the **stack**:

	CDECL	STDCALL
Origin	C language	Microsoft API
Who cleans stack	Caller	Callee
Return value	eax	eax
Order	Last value pushed is “first argument”	

Exercises

- Fill in code
- Read code