Functions

1

Functions

Functions are fundamentals in programming

They execute a piece of code to do a well-known task

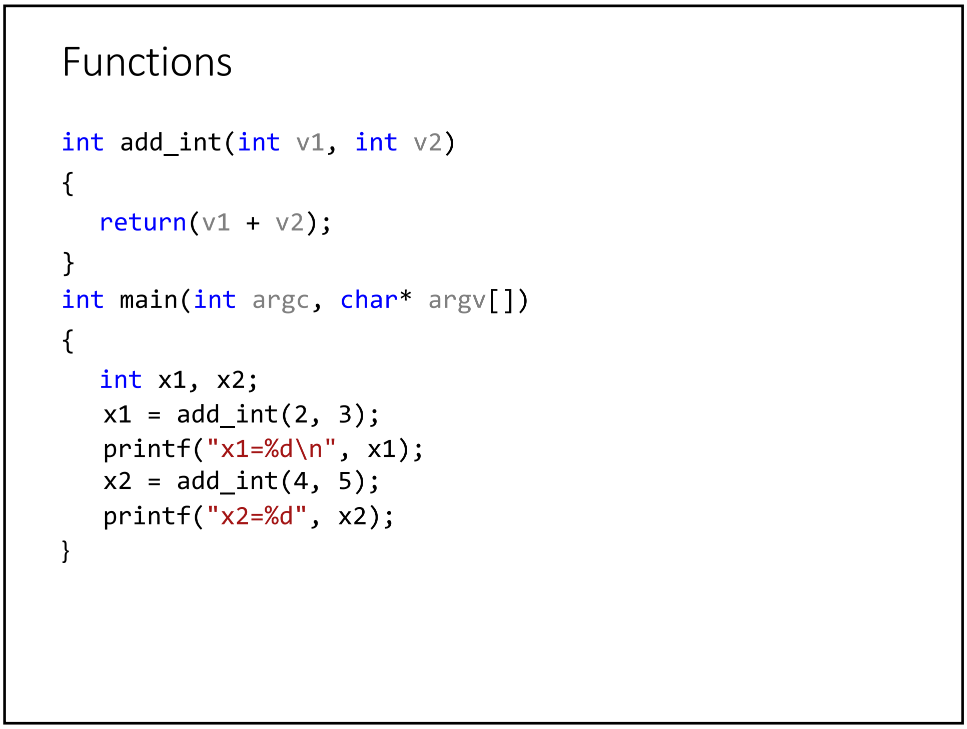
Functions can call any other function, or even call itself (recursive functions)

Programmers can share functions in different ways and languages

For example, the function printf(), used to print data in the display, can be used by any programmer, but no one knows who make it

Like many other functions…

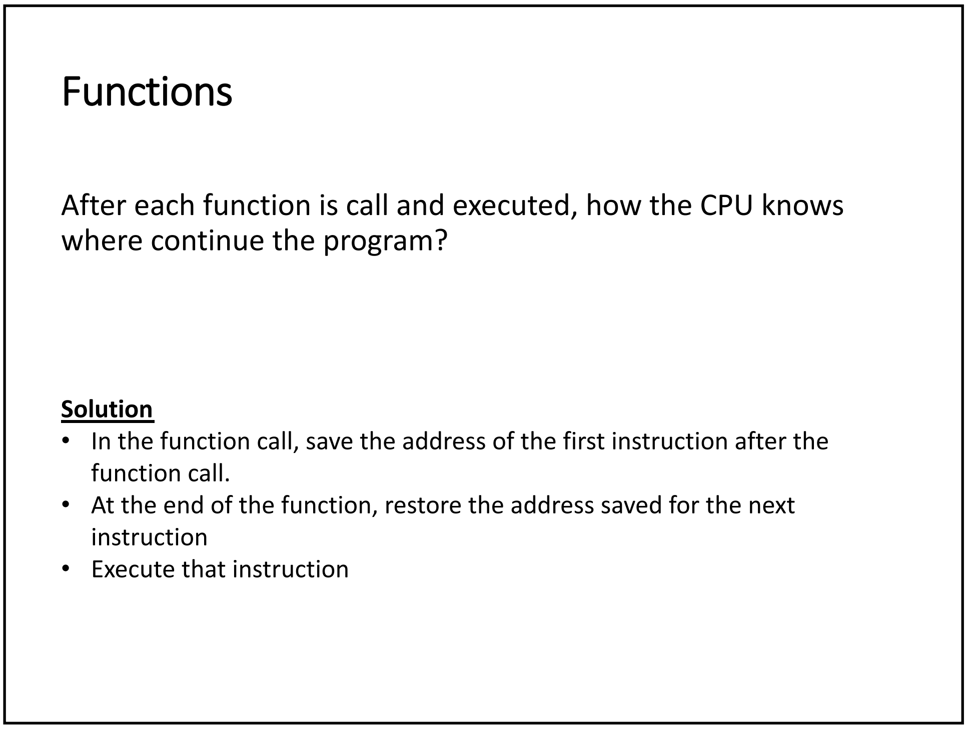
2



Functions

```
int add_int(int v1, int v2)
{
   return(v1 + v2);
}
int main(int argc, char* argv[])
{
   int x1, x2;
   x1 = add_int(2, 3);
   printf("x1=%d\n", x1);
   x2 = add_int(4, 5);
   printf("x2=%d", x2);
}
```

3



Functions

After each function is call and executed, how the CPU knows where continue the program?

Solution

- In the function call, save the address of the first instruction after the function call.
- At the end of the function, restore the address saved for the next instruction
- Execute that instruction

4

EIP Instruction Pointer Register

- The EIP register always contains the address of the next instruction to be executed.

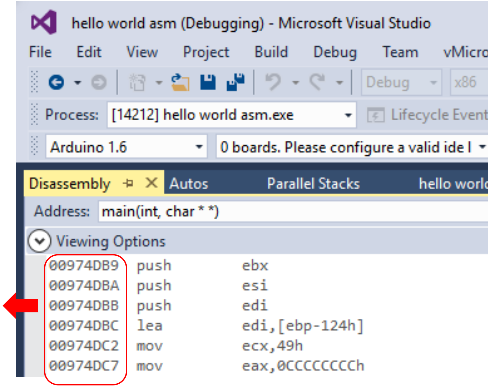
- You cannot directly access or change the instruction pointer.

- Instructions that control program flow, such as calls, jumps, loops, and interrupts, automatically change the instruction pointer.

Functions

EIP (Instruction Point) points, at any time, for the **next** instruction to be executed

Using the disassembly window, you can see the address of each instruction (number in the left of each instruction) and verify that, at any time, the EIP register points to the **next** instruction to be executed



Process: [14212] hello world asm.exe

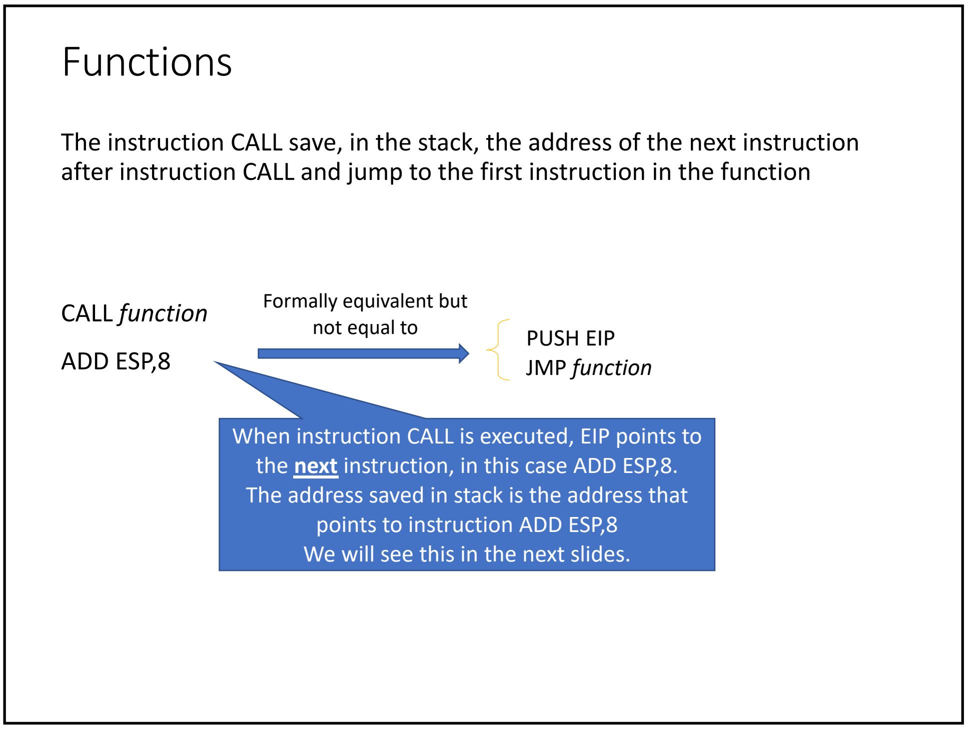
Arduino 1.6

0 boards. Please configure a valid ide l

Address: main(int, char * *)

```
00974DB9  push        ebx
00974DBA  push        esi
00974DBB  push        edi
00974DBC  lea         edi,[ebp-124h]
00974DC2  mov         ecx,49h
00974DC7  mov         eax,0CCCCCCCCh
```

Address of each instruction



Functions

The instruction CALL save, in the stack, the address of the next instruction after instruction CALL and jump to the first instruction in the function

CALL *function*

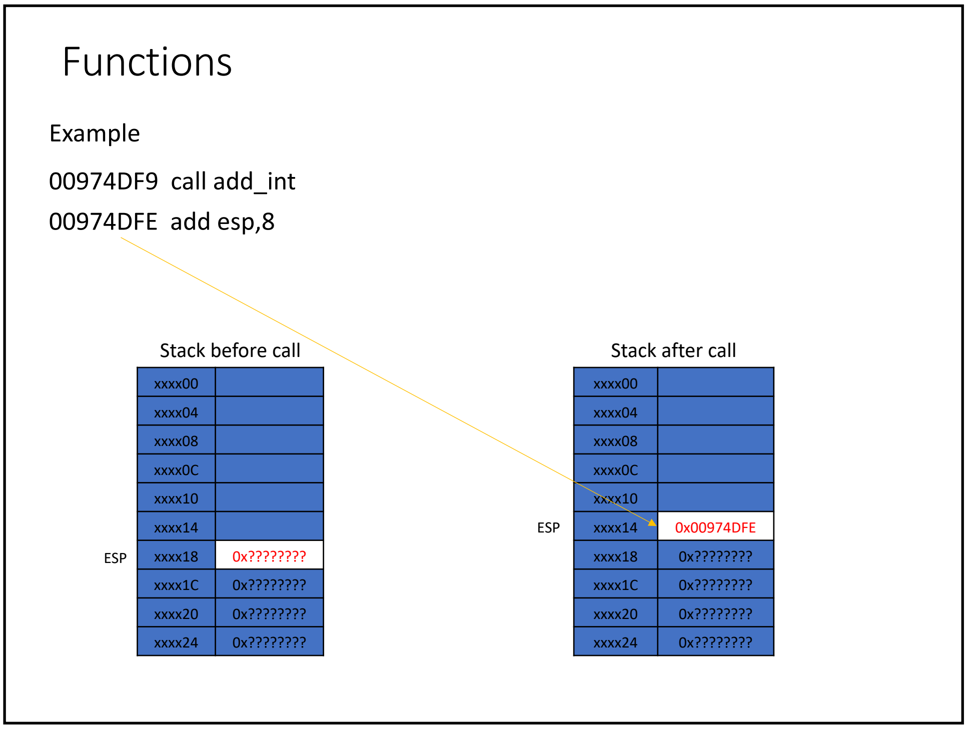
ADD ESP,8

Formally equivalent but not equal to

PUSH EIP
JMP *function*

When instruction CALL is executed, EIP points to the **next** instruction, in this case ADD ESP,8. The address saved in stack is the address that points to instruction ADD ESP,8
We will see this in the next slides.

7



Functions

Example

00974DF9 call add_int

00974DFE add esp,8

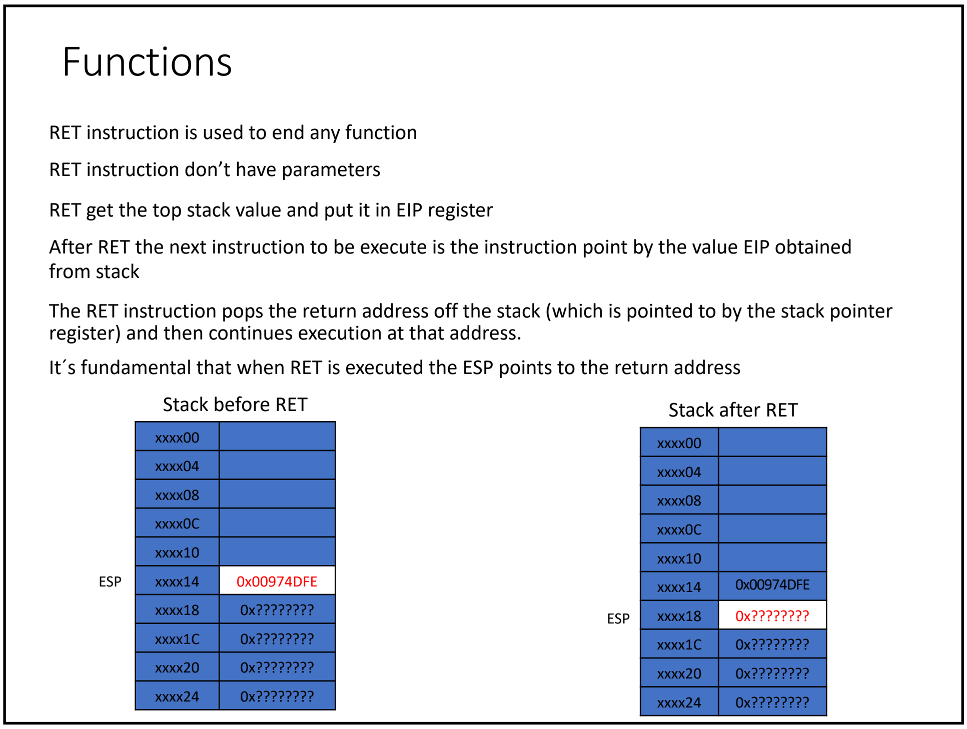
Stack before call

	xxxx00	
	xxxx04	
	xxxx08	
	xxxx0C	
	xxxx10	
	xxxx14	
ESP	xxxx18	0x????????
	xxxx1C	0x????????
	xxxx20	0x????????
	xxxx24	0x????????

Stack after call

	xxxx00	
	xxxx04	
	xxxx08	
	xxxx0C	
	xxxx10	
ESP	xxxx14	0x00974DFE
	xxxx18	0x????????
	xxxx1C	0x????????
	xxxx20	0x????????
	xxxx24	0x????????

8



Functions

RET instruction is used to end any function

RET instruction don’t have parameters

RET get the top stack value and put it in EIP register

After RET the next instruction to be execute is the instruction point by the value EIP obtained from stack

The RET instruction pops the return address off the stack (which is pointed to by the stack pointer register) and then continues execution at that address.

It´s fundamental that when RET is executed the ESP points to the return address

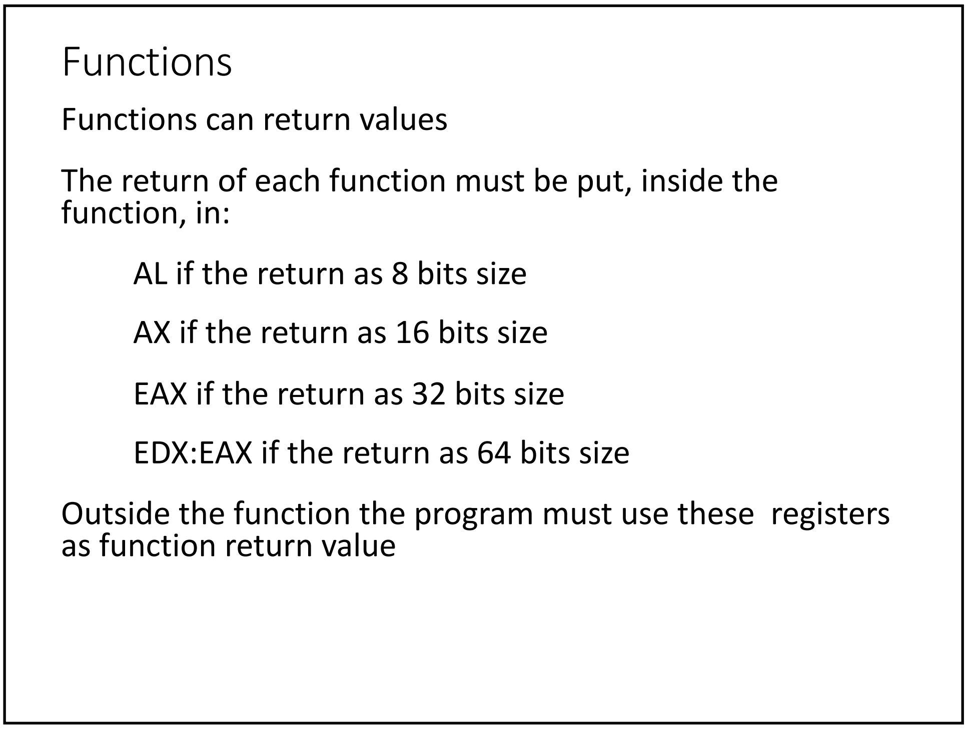
Stack before RET

	xxxx00	
	xxxx04	
	xxxx08	
	xxxx0C	
	xxxx10	
ESP	xxxx14	0x00974DFE
	xxxx18	0x????????
	xxxx1C	0x????????
	xxxx20	0x????????
	xxxx24	0x????????

Stack after RET

	xxxx00	
	xxxx04	
	xxxx08	
	xxxx0C	
	xxxx10	
	xxxx14	0x00974DFE
ESP	xxxx18	0x????????
	xxxx1C	0x????????
	xxxx20	0x????????
	xxxx24	0x????????

9



Functions

Functions can return values

The return of each function must be put, inside the function, in:

AL if the return as 8 bits size

AX if the return as 16 bits size

EAX if the return as 32 bits size

EDX:EAX if the return as 64 bits size

Outside the function the program must use these registers as function return value

10

Using parameters in functions

Functions can have parameters

Parameters can be used to customize the function task

Functions can do different tasks according the parameters

The values of parameters are defined by function caller

11

Using parameters in functions

At physical level parameters can be passed to functions by registers or by stack

By registers	By stack
Fast to pass parameters	Slow to pass parameters
Number of parameters limited by the number of registers	Number of parameters unlimited
Preferential used in assembly	Can be used in any language

Compilers can optimize functions to use registers to pass parameters, even if the programmer use stack for it

12

Parameters passed by registers

```
//C code example
int main(int argc, char* argv[])
{
    int x1;
    x1 = add_int(2, 3);
}

int add_int(int v1, int v2)
{
    return(v1 + v2);
}
```

Equivalent Assembly code for main

Programmer can choose the registers to pass parameters in this case EAX for the first and EBX for the second parameter

```
MOV EAX,2
MOV EBX,3
CALL add_int
MOV x1,EAX ; the return value is passed in  EAX
```

Assembly code for add_int

```
ADD EAX,EBX
RET   ; return to the next line after CALL
```

13

Parameters passed by stack

Parameters are put in stack from **right to left**

In the case x1 = add_int(2, 3);

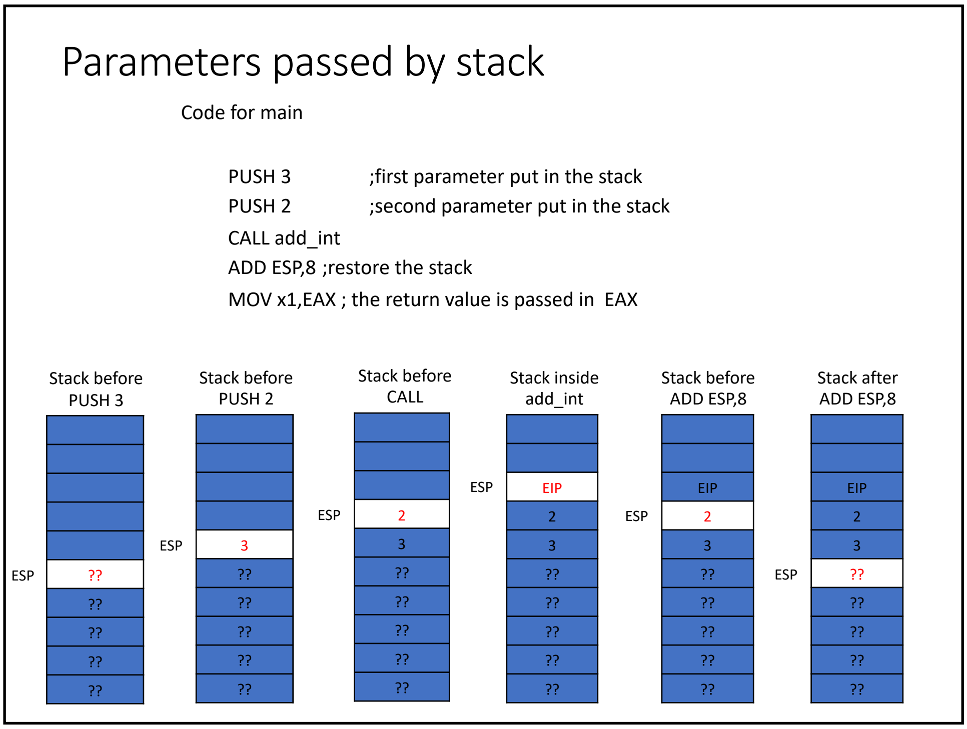
the first parameter put in the stack is 3 and the second is 2

Remember that **CALL also put in the stack the EIP**

Inside the function the code get the parameters from stack regarding they position

After finish the function the program must restore the stack to the state before the parameters passing

14



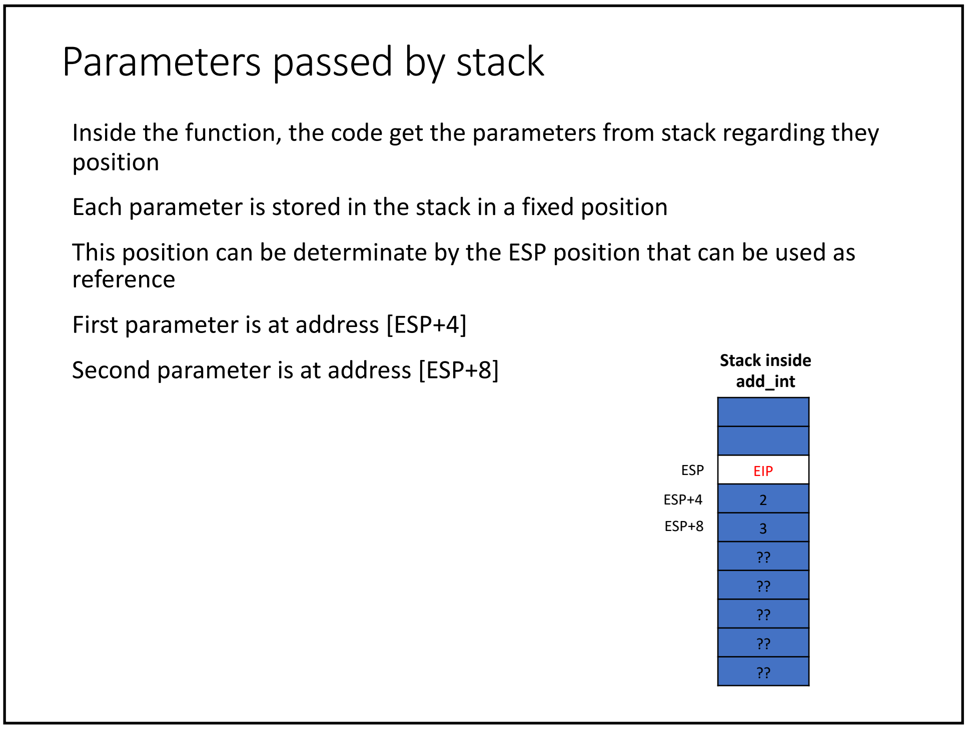
Parameters passed by stack

Code for main

```
PUSH 3          ;first parameter put in the stack
PUSH 2          ;second parameter put in the stack
CALL add_int
ADD ESP,8 ;restore the stack
MOV x1,EAX ; the return value is passed in  EAX
```

	Stack before PUSH 3		Stack before PUSH 2		Stack before CALL		Stack inside add_int		Stack before ADD ESP,8		Stack after ADD ESP,8
						ESP	EIP		EIP		EIP
				ESP	2		2	ESP	2		2
		ESP	3		3		3		3		3
ESP	??		??		??		??		??	ESP	??
	??		??		??		??		??		??
	??		??		??		??		??		??
	??		??		??		??		??		??
	??		??		??		??		??		??

15



Parameters passed by stack

Inside the function, the code get the parameters from stack regarding they position

Each parameter is stored in the stack in a fixed position

This position can be determinate by the ESP position that can be used as reference

First parameter is at address [ESP+4]

Second parameter is at address [ESP+8]

	Stack inside add_int
ESP	EIP
ESP+4	2
ESP+8	3
	??
	??
	??
	??
	??

16

Parameters passed by stack

If the function needs to do some push, the ESP is changed

For example, if the function needs PUSH ECX, the ESP is decremented by 4

Now the parameters referenced by ESP are different

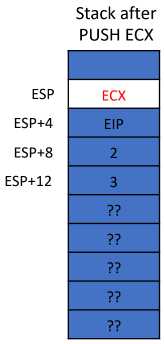
First parameter is at address [ESP+8]

Second parameter is at address [ESP+12]

If function do others PUSH, the ESP change again

This difficult the access to parameters

How to solve this???



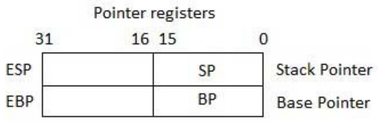
Stack after PUSH ECX

ESP	ECX
ESP+4	EIP
ESP+8	2
ESP+12	3
	??
	??
	??
	??
	??

BP

Base Pointer (BP) – The 16-bit BP register mainly helps in referencing the parameter variables passed to a subroutine. The address in SS register is combined with the offset in BP to get the location of the parameter. BP can also be combined with DI and SI as base register for special addressing.

EBP – 32 bit



Pointer registers

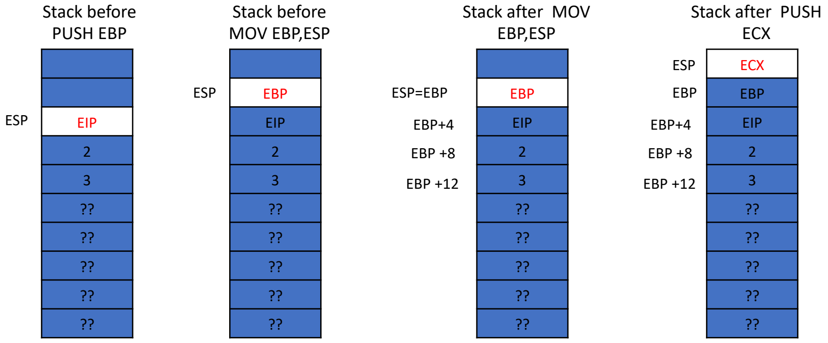
	31 16	15 0	
ESP		SP	Stack Pointer
EBP		BP	Base Pointer

Parameters passed by stack

To solve the problem of changing ESP, is used the register EBP as reference according the following code

```
PUSH EBP
MOV EBP,ESP
```

If the function do a new PUSH (PUSH ECX) the ESP is changed but the EBP remains unchanged



Stack before PUSH EBP

ESP EIP
2
3
??
??
??
??
??

Stack before MOV EBP,ESP

ESP EBP
EIP
2
3
??
??
??
??
??

Stack after MOV EBP,ESP

ESP=EBP EBP
EBP+4 EIP
EBP +8 2
EBP +12 3
??
??
??
??
??

Stack after PUSH ECX

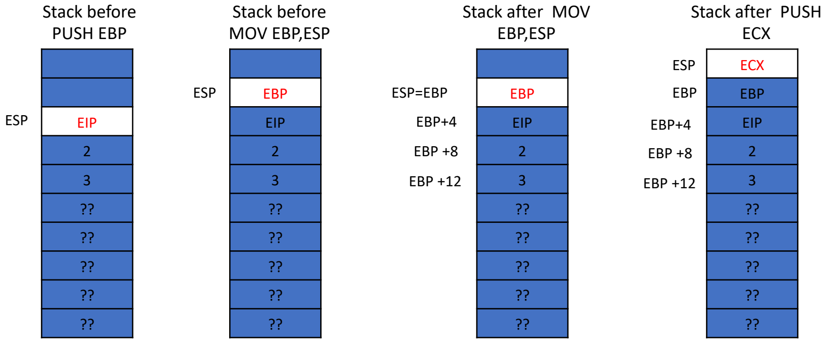
ESP ECX
EBP EBP
EBP+4 EIP
EBP +8 2
EBP +12 3
??
??
??
??
??

Parameters passed by stack

Now the parameters referenced by EBP are always in the same relative position, independent of the numbers of push instructions

First parameter is always at address [EBP+8]

Second parameter is always at address [EBP+12]



Stack before PUSH EBP

ESP EIP
2
3
??
??
??
??
??

Stack before MOV EBP,ESP

ESP EBP
EIP
2
3
??
??
??
??
??

Stack after MOV EBP,ESP

ESP=EBP EBP
EBP+4 EIP
EBP +8 2
EBP +12 3
??
??
??
??
??

Stack after PUSH ECX

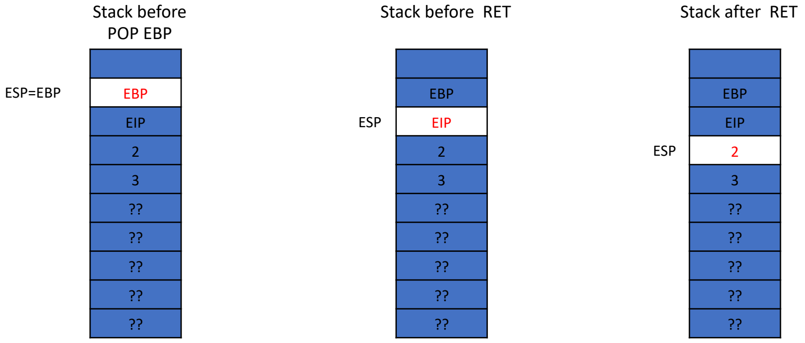
ESP ECX
EBP EBP
EBP+4 EIP
EBP +8 2
EBP +12 3
??
??
??
??
??

Parameters passed by stack

At the end of the function, EBP must be restored before the RET instruction

POP EBP

RET



21

Parameters passed by stack

Code for function add_int

```
PUSH EBP            ;save EBP in the stack
MOV EBP,ESP         ;make a stack reference using EBP
MOV EAX,[EBP+8]          ;get first parameter (2) in to EAX
ADD EAX,[EBP+12]    ;add first parameter with the second (3)
POP EBP             ;restore EBP
RET                 ;end the function
```

22

Using parameters in functions

As we see, at **physical level**, parameters can be passed to functions by registers or by stack

At **logical level**, parameters can be passed by value or by reference

23

Using parameters in functions

When parameters are passed by value, a copy of the original parameter is passed to the function

This means that function have access to the parameter value by a copy and not by the original parameter

If the function change the value passed, this change affect only the copy, but not the original parameter

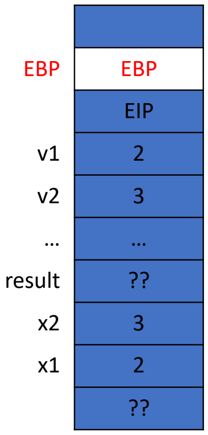
24

Using parameters in functions

```
//C code example
int add_int(int, int);
int main(int argc, char* argv[])
{
   int x1=2,x2=3,result;
   result = add_int(x1,x2);
}

int add_int(int v1, int v2)
{
   return(v1 + v2);
}
```

Stack inside the function



x1, x2, v1 and v2 are local variables

They are stored in the stack

v1 as the same value than x1, but they are sored at different address

The same is true for v2 and x2

Using parameters in functions

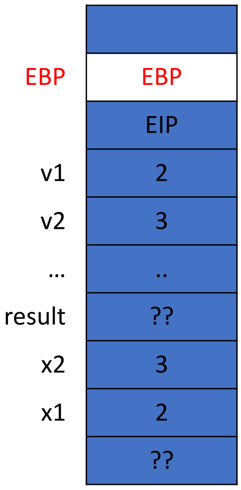
Assembly code for main

```
PUSH x2
PUSH x1
CALL add_int
ADD ESP,8
MOV result,EAX
```

Assembly code for add_int

```
PUSH EBP
MOV EBP,ESP
MOV EAX,[EBP+8]; v1
ADD EAX,[EBP+12]; add v1 with v2
POP EBP
RET
```

Stack inside the function



Using parameters in functions

When parameters are passed by reference, the address of the parameter is passed to the function

This means that function have access to the parameter since they know the exact parameter address

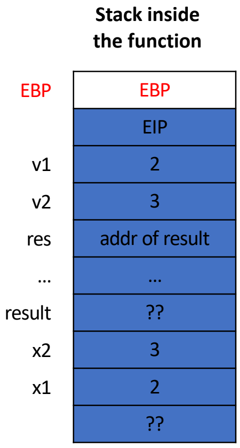
The function can write in the address of the parameter, and this write change the original parameter

27

Using parameters in functions

```
//C code example
void add_int(int, int , int *);
int main(int argc, char* argv[])
{
   int x1=2,x2=3,result;
    add_int(x1,x2,&result);
}

int add_int(int v1, int v2, int *res)
{
   *res=v1 + v2;
}
```



Stack inside the function

EBP	EBP
	EIP
v1	2
v2	3
res	addr of result
…	…
result	??
x2	3
x1	2
	??

x1 and x2 are passed by value

result is passed by reference

The address of result is stored in the stack as a parameter

28

Using parameters in functions

Assembly code for main

```
LEA EAX,result     ;put the address of result in EAX
PUSH EAX           ;put the address of result in the stack
PUSH x2            ;put the 2nd parameter in the stack
PUSH x1            ;put the 1st parameter in the stack
CALL add_int       ;call the function add_int
ADD ESP,12         ;restore the stack
```

Assembly code for add_int

```
PUSH EBP
MOV EBP,ESP
MOV EBX,[EBP+16]; put in EBX the address of result
MOV EAX,[EBP+8]; v1
ADD EAX,[EBP+12]; add v1 with v2
MOV [EBX],EAX; put the sum in the result
POP EBP
RET
```

Stack inside the function

EBP	EBP
	EIP
v1	2
v2	3
res	adr of result
…	…
result	??
x2	3
x1	2
	??