Lab 2. The Art of Assembly Language (I)

Dan Bruce, David Clark and Héctor D. Menéndez

Department of Computer Science
University College London

October 16, 2017

Modified from Xeno Kovah slides of Open Security Training

Introduction

This Lectures aims to teach assembly instructions from the basics.

We will start with the simplest instructions and, after, we will have a look to the control flow.

Instruction NOP

NOP: Does Nothing.

Just there to pad/align bytes, or to delay time.

It is usually used in obfuscation.

Instruction MOV

This instruction is use to **move**:

- **register** to **register**
- **memory** to **register**, **register** to **memory**
- **constant** to **register**, **constant** to **memory**
- **Never** use for memory to memory!

Memory addresses are given in **r/m32** form.

r/m32 Addressing Forms (I)

An r/m32 means it could be taking a value either from a register, or a memory address.

In Intel syntax, most of the time square brackets [] means to treat the value within as a memory address, and fetch the value at that address (like a pointer).

Most complicated form is:

```
[base + index*scale + disp]
```

r/m32 Addressing Forms (II)

```
1  mov eax, ebx
2  mov eax, [ebx]
3  mov eax, [ebx+ecx*X]  ;(X=1, 2, 4, 8)
4  mov eax, [ebx+ecx*X+Y]  ;(Y= one byte, 0-255 or 4 bytes,
5                           ; 0-2^32-1)
```

Instruction LEA

LEA means Load Effective Address.

Frequently used with pointer arithmetic, sometimes for just arithmetic in general.

Uses the r/m32 form but is the exception to the rule that the square brackets [ ] syntax means “value at”.

Example:
ebx = 0x2, edx = 0x1000

```
1   lea eax, [edx+ebx*2]
```

eax = 0x1004, not the value at 0x1004

Instructions ADD and SUB

Adds or Subtracts, just as expected:

- Destination operand can be r/m32 or register
- Source operand can be r/m32 or register or immediate
- No source and destination as r/m32s, because that could produce a memory-to-memory transfer, which is not allowed on x86.

Evaluates the operation as if it were on signed AND unsigned data, and sets flags as appropriate.

They modify OF, SF, ZF, AF, PF, and CF flags

```
1  add esp, 8
2  sub eax, [ebx*2]
```

Instructions MUL and DIV

Multiplies or Divides, just as expected:

- In this case the first operand is in eax (and edx for the division).
- The other operand can be r/m32 or register or immediate

```
1  mul dword ebx
2  div dword ebx
```

The result is stored in eax (also in edx for the multiplication).

The Stack

Here, we are interested to check how the stack works and how the system call function works.

We will see the relationships between the stack and the calls.

Instruction PUSH

PUSH - Push Word, Doubleword or Quadword onto the Stack.

For our examples, it will always be a DWORD (4 bytes).

Can either be an immediate (a numeric constant), or the value in a register.

The push instruction automatically decrements the stack pointer, esp, by 4.

Instruction POP

POP - Pop a Value from the Stack.

Take a DWORD off the stack, put it in a register, and increment esp by 4

PUSH example

eax	0x00000003
esp	0x0012FF8C

eax	0x00000003
esp	0x0012FF88

Stack Before

	0x0012FF90	0x00000001
esp⟶	0x0012FF8C	0x00000002
	0x0012FF88	undefined
	0x0012FF84	undefined
	0x0012FF80	undefined

Stack After

	0x00000001
	0x00000002
esp⟶	0x00000003
	undefined
	undefined

POP example

eax	0xFFFFFFFF
esp	0x0012FF88

eax	0x00000003
esp	0x0012FF8C

Stack Before

0x0012FF90	0x00000001	
0x0012FF8C	0x00000002	
esp⟶ 0x0012FF88	0x00000003	
0x0012FF84	undefined	
0x0012FF80	undefined	

Stack After

	0x00000001	
esp⟶	0x00000002	
	undefined	
	undefined	
	undefined	

Calling Conventions

How code calls a subroutine is compiler-dependent and configurable.
But there are a few conventions.

We will only deal with the “cdecl” and “stdcall” conventions.

Cdecl

"C declaration" - most common calling convention.

Function parameters pushed onto stack right to left.

Saves the old stack frame pointer and sets up a new stack frame.

eax or edx:eax returns the result for primitive data types.

Caller is responsible for cleaning up the stack

Stdcall

This convention is used by Microsoft C++ code - e.g. Win32 API.

Function parameters pushed onto stack right to left.

Saves the old stack frame pointer and sets up a new stack frame.

eax or edx:eax returns the result for primitive data types.

Callee responsible for cleaning up any stack parameters it takes.

Instruction CALL

CALL's job is to transfer control to a different function, in a way that control can later be resumed where it left off.

First it pushes the address of the next instruction onto the stack.

It will be used by RET for the moment when the procedure is done.

It changes eip to the address given in the instruction.

Destination address can be specified in multiple ways: Absolute address and Relative address (relative to the end of the instruction).

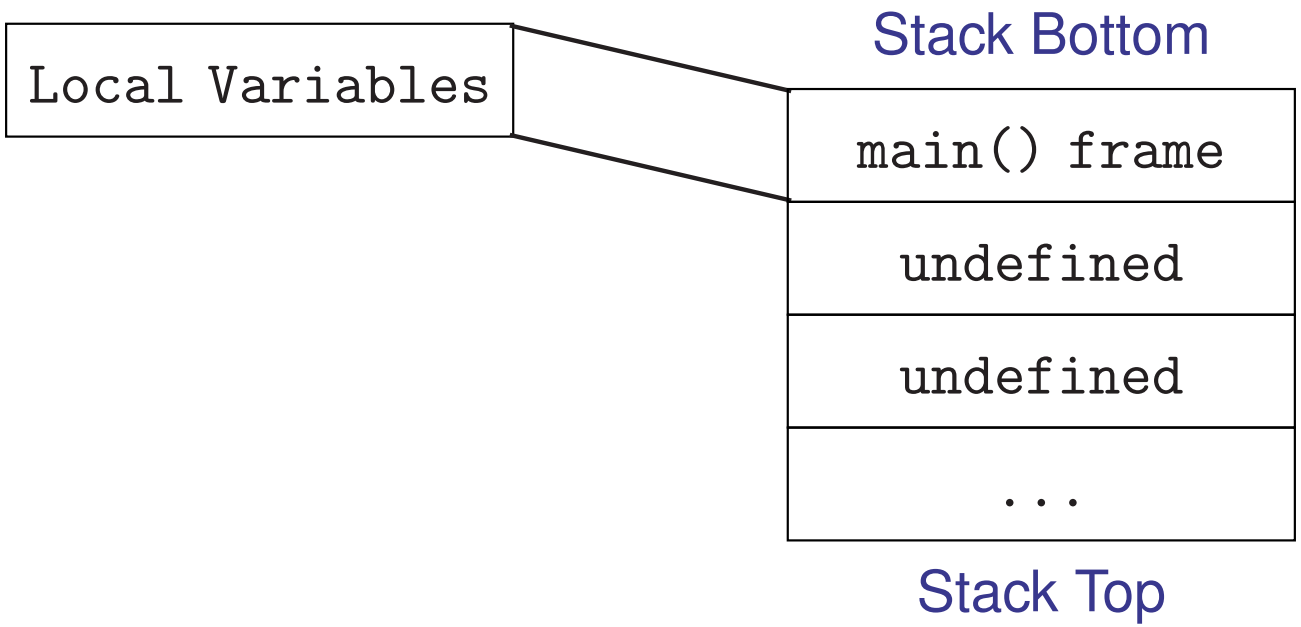
Instruction RET

There are two types:

- Pop the top of the stack into eip (remember pop increments stack pointer). In this form, the instruction is just written as “ret” (typically used by cdecl functions)
- Pop the top of the stack into eip and add a constant number of bytes to esp. In this form, the instruction is written as “ret 0x8”, or “ret 0x20”, etc (typically used by stdcall functions)

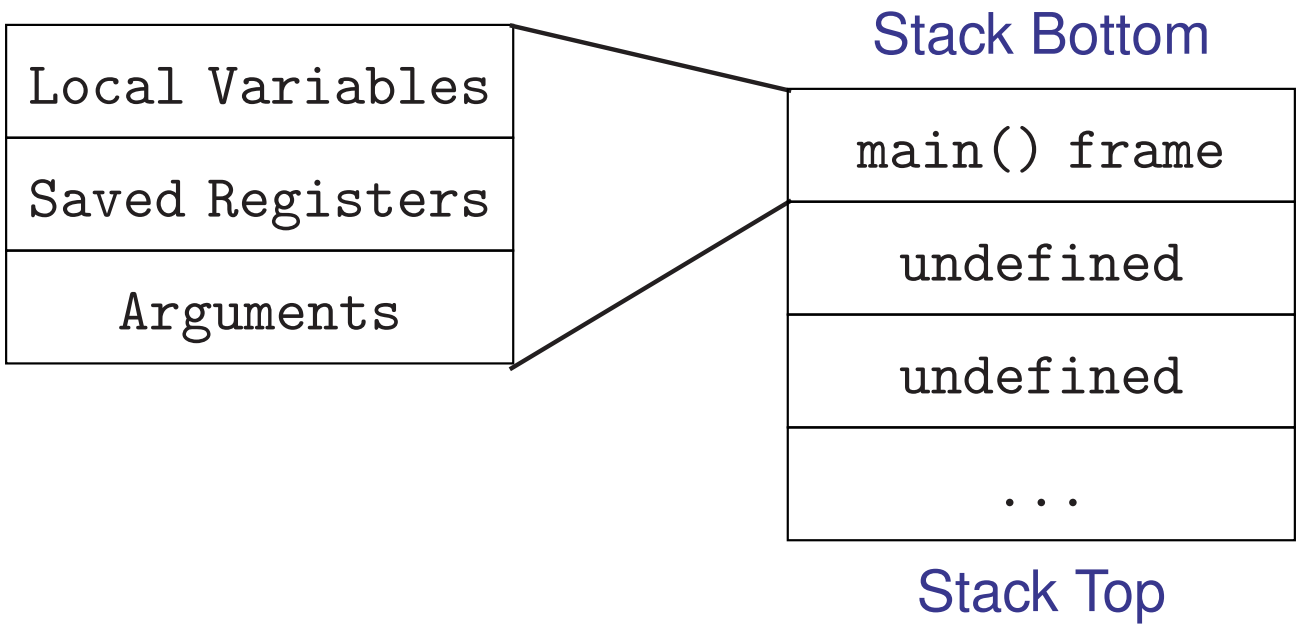
General Stack Frame Operation (I)

We are going to pretend that main() is the very first function being executed in a program. This is what its stack looks like to start with (assuming it has any local variables).



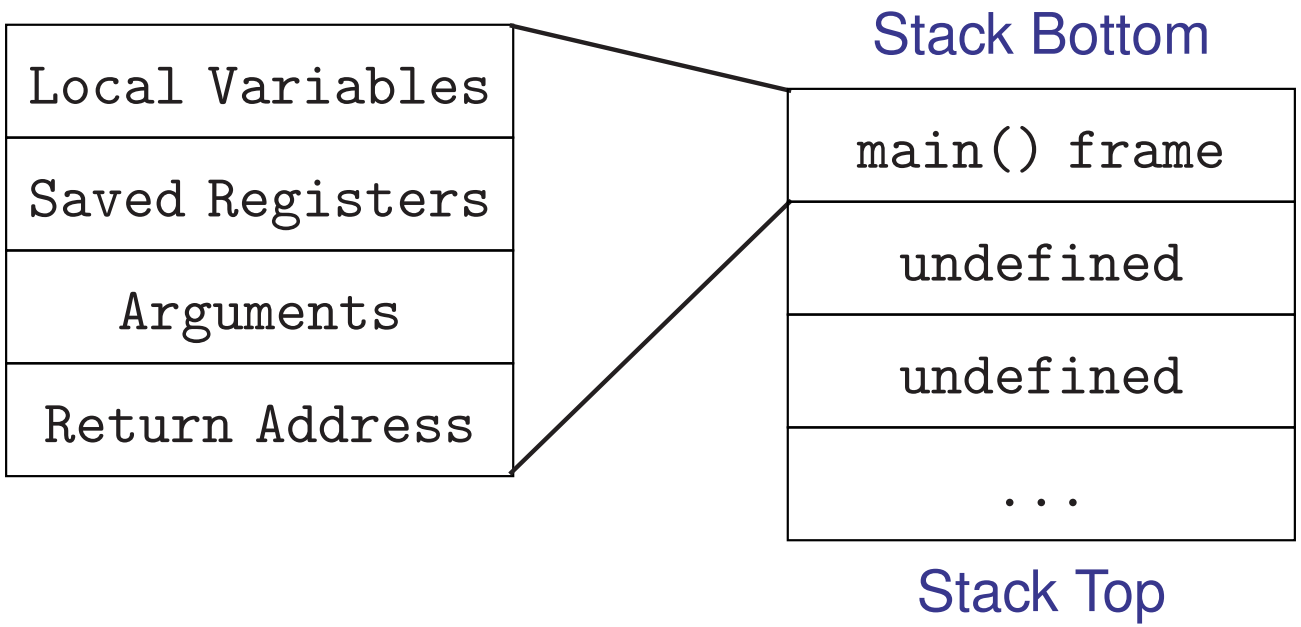
General Stack Frame Operation (II)

When main() decides to call a subroutine, main() becomes “the caller”. We will assume main() has some registers it would like to remain the same, so it will save them. We will also assume that the callee function takes some input arguments.



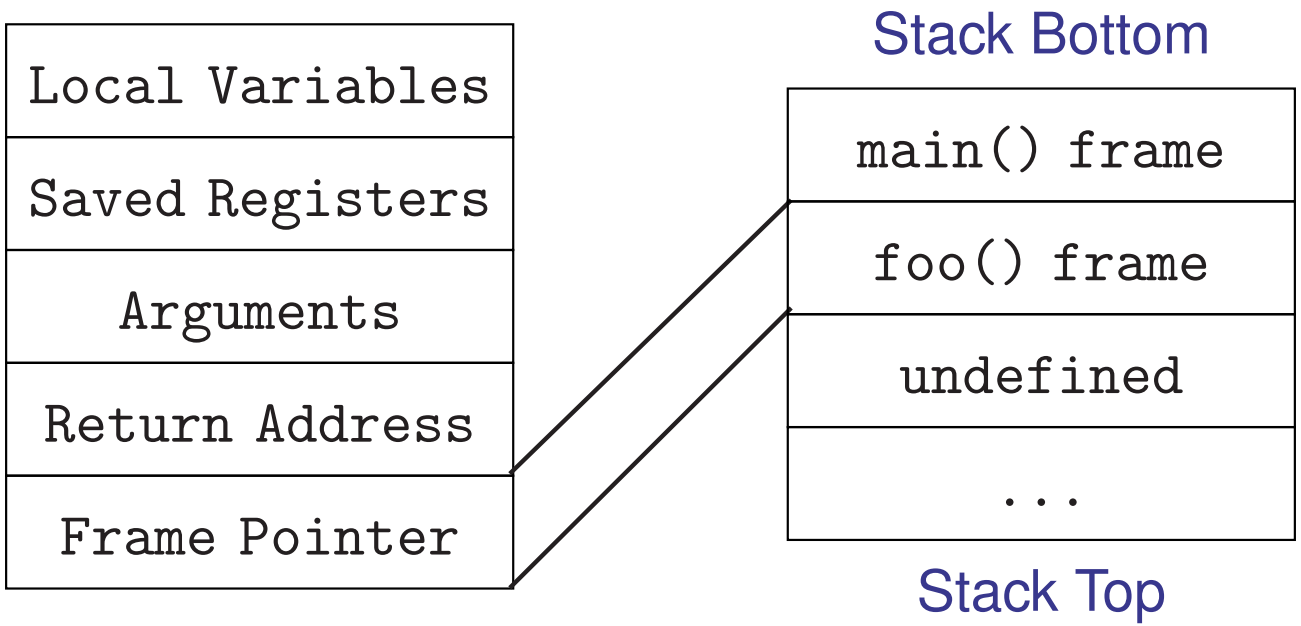
General Stack Frame Operation (III)

When main() actually issues the CALL instruction, the return address gets saved onto the stack, and because the next instruction after the call will be the beginning of the called function, we consider the frame to have changed to the callee.



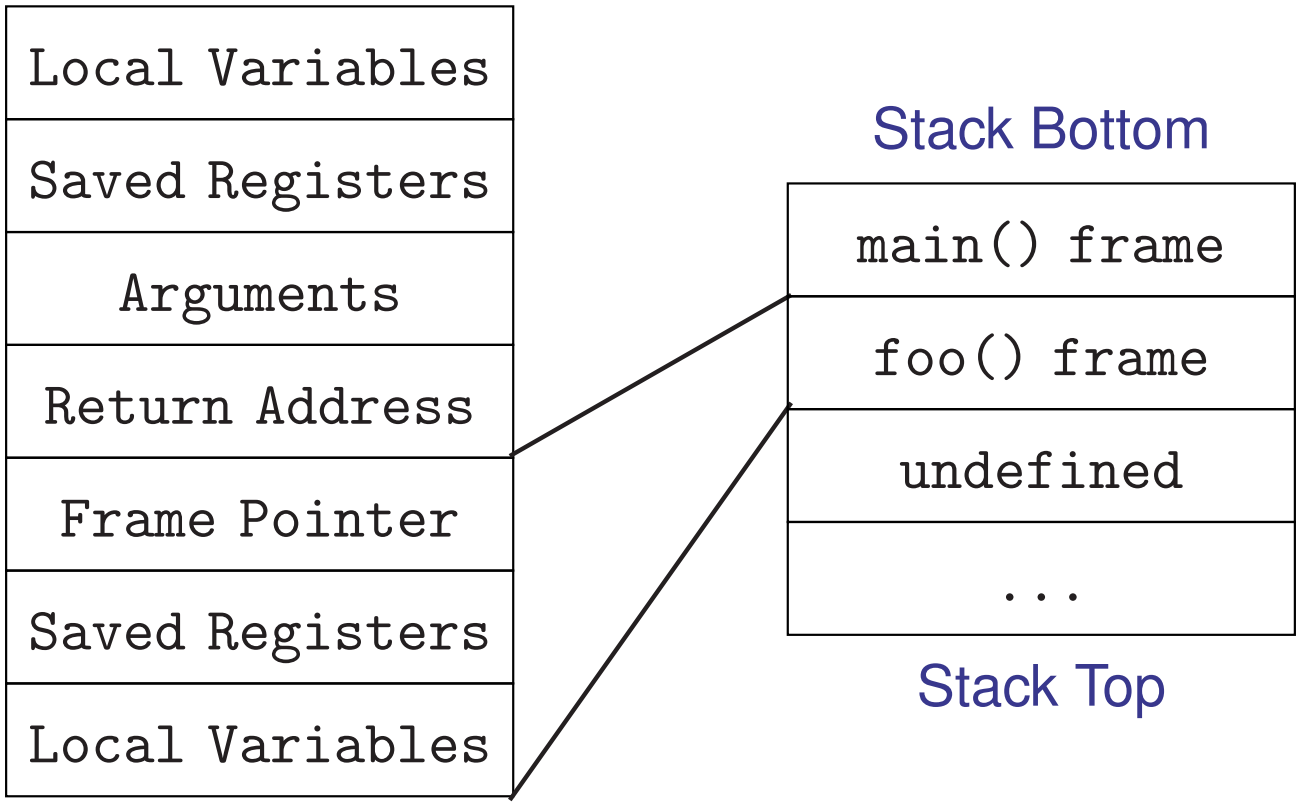
General Stack Frame Operation (IV)

When foo() starts, the frame pointer (ebp) still points to main()'s frame. So the first thing it does is to save the old frame pointer on the stack and set the new value to point to its own frame.



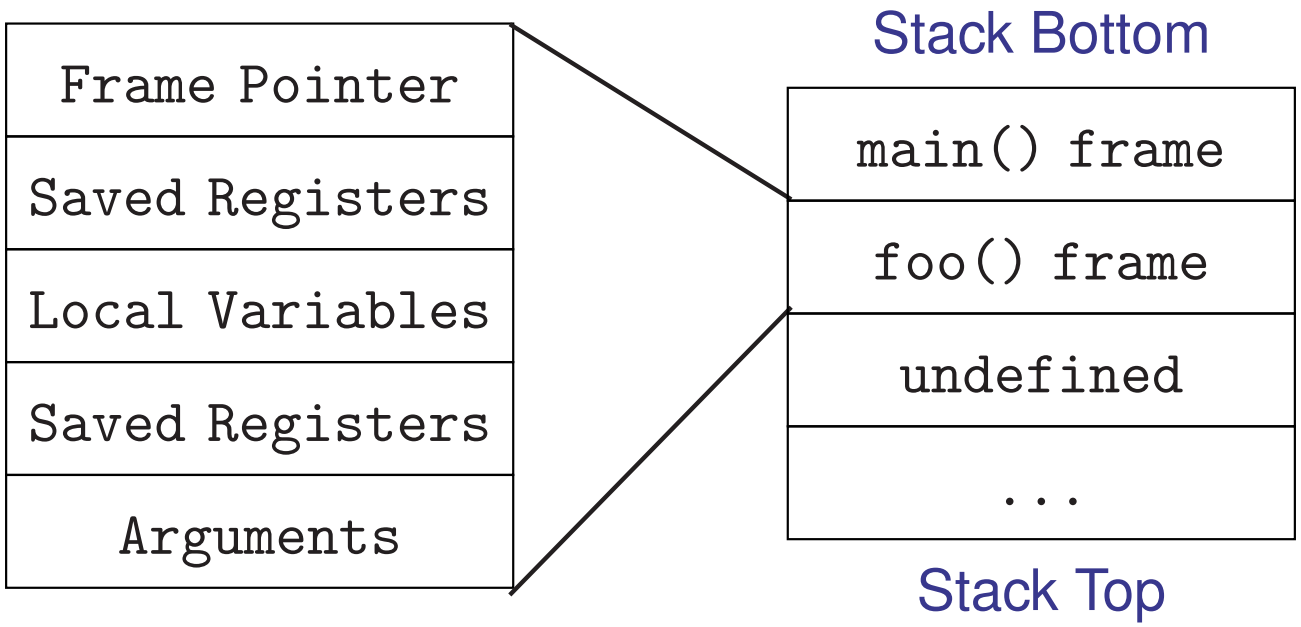
General Stack Frame Operation (V)

Next, we'll assume the callee foo() would like to use all the registers, and must therefore save the callee-save registers. Then it will allocate space for its local variables.



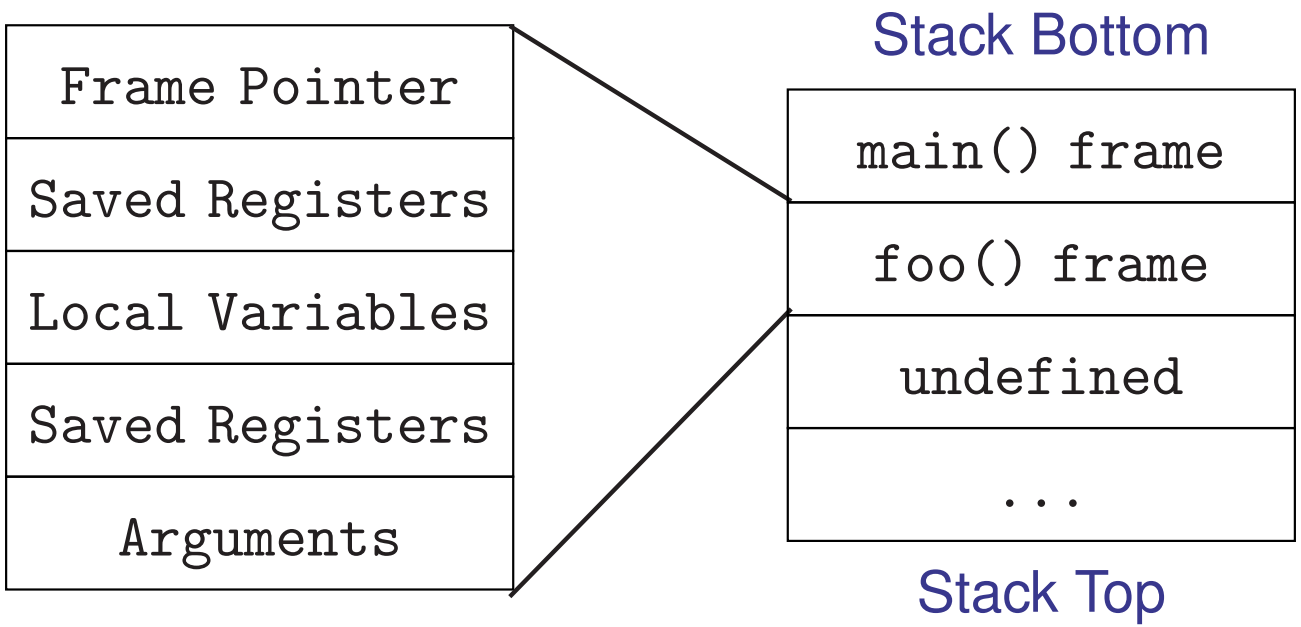
General Stack Frame Operation (VI)

At this point, foo() decides it wants to call bar(). It is still the callee-of-main(), but it will now be the caller-of-bar. So it saves any caller-save registers that it needs to. It then puts the function arguments on the stack as well



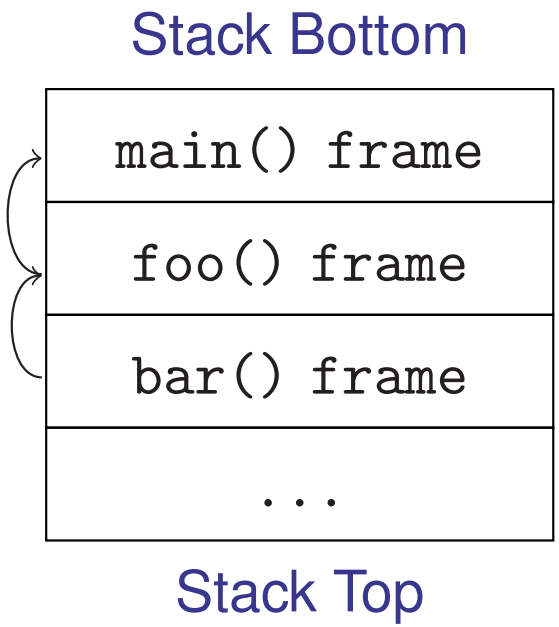
General Stack Frame Operation (VII)

Every part of the stack frame is technically optional (that is, you can hand code asm without following the conventions.) But compilers generate code which uses portions if they are needed. Which pieces are used can sometimes be manipulated with compiler options. (E.g. omit frame pointers, changing calling convention to pass arguments in registers, etc.)



Stack Frames are a Linked List!

The ebp in the current frame points at the saved ebp of the previous frame.



Example

EIP = 00401010, but no instruction yet executed

eax	0x003435C0	ebp	0x0012FFB8	esp	0x0012FF6C

```
sub:
00401000   push ebp
00401001   mov ebp,esp
00401003   mov eax,0BEEFh
00401008   pop ebp
00401009   ret
main:
00401010   push ebp
00401011   mov ebp,esp
00401013   call sub (401000h)
00401018   mov eax,0F00Dh
0040101D   pop ebp
0040101E   ret
```

0x0012FF6C	0x004012E8
0x0012FF68	undefined
0x0012FF64	undefined
0x0012FF60	undefined
0x0012FF5C	undefined
0x0012FF58	undefined

Example

The first instruction introduces frame register in the stack

eax	0x003435C0	ebp	0x0012FFB8	esp	0x0012FF68

```
sub:
00401000   push ebp
00401001   mov ebp,esp
00401003   mov eax,0BEEFh
00401008   pop ebp
00401009   ret
main:
00401010   push ebp
00401011   mov ebp,esp
00401013   call sub (401000h)
00401018   mov eax,0F00Dh
0040101D   pop ebp
0040101E   ret
```

0x0012FF6C	0x004012E8
0x0012FF68	0x0012FFB8
0x0012FF64	undefined
0x0012FF60	undefined
0x0012FF5C	undefined
0x0012FF58	undefined

Example

We update the frame register with the stack register

eax	0x003435C0	ebp	0x0012FF68	esp	0x0012FF68

```
sub:
00401000   push ebp
00401001   mov ebp,esp
00401003   mov eax,0BEEFh
00401008   pop ebp
00401009   ret
main:
00401010   push ebp
00401011   mov ebp, esp
00401013   call sub (401000h)
00401018   mov eax,0F00Dh
0040101D   pop ebp
0040101E   ret
```

0x0012FF6C	0x004012E8
0x0012FF68	0x0012FFB8
0x0012FF64	undefined
0x0012FF60	undefined
0x0012FF5C	undefined
0x0012FF58	undefined

Example

We call the subroutine

eax	0x003435C0	ebp	0x0012FF68	esp	0x0012FF64

```
sub:
00401000   push ebp
00401001   mov ebp,esp
00401003   mov eax,0BEEFh
00401008   pop ebp
00401009   ret
main:
00401010   push ebp
00401011   mov ebp, esp
00401013   call sub (401000h)
00401018   mov eax,0F00Dh
0040101D   pop ebp
0040101E   ret
```

0x0012FF6C	0x004012E8
0x0012FF68	0x0012FFB8
0x0012FF64	0x00401018
0x0012FF60	undefined
0x0012FF5C	undefined
0x0012FF58	undefined

Example

Again, we save the frame register in the stack

eax	0x003435C0	ebp	0x0012FF68	esp	0x0012FF60

```
sub:
00401000   push ebp
00401001   mov ebp,esp
00401003   mov eax,0BEEFh
00401008   pop ebp
00401009   ret
main:
00401010   push ebp
00401011   mov ebp, esp
00401013   call sub (401000h)
00401018   mov eax,0F00Dh
0040101D   pop ebp
0040101E   ret
```

0x0012FF6C	0x004012E8
0x0012FF68	0x0012FFB8
0x0012FF64	0x00401018
0x0012FF60	0x0012FF68
0x0012FF5C	undefined
0x0012FF58	undefined

Example

We update it with the stack register

eax	0x003435C0	ebp	0x0012FF60	esp	0x0012FF60

```
sub:
00401000   push ebp
00401001   mov ebp,esp
00401003   mov eax,0BEEFh
00401008   pop ebp
00401009   ret
main:
00401010   push ebp
00401011   mov ebp, esp
00401013   call sub (401000h)
00401018   mov eax,0F00Dh
0040101D   pop ebp
0040101E   ret
```

0x0012FF6C	0x004012E8
0x0012FF68	0x0012FFB8
0x0012FF64	0x00401018
0x0012FF60	0x0012FF68
0x0012FF5C	undefined
0x0012FF58	undefined

Example

We change the value for eax register

eax	0x0000BEEF	ebp	0x0012FF60	esp	0x0012FF60

```
sub:
00401000   push ebp
00401001   mov ebp,esp
00401003   mov eax,0BEEFh
00401008   pop ebp
00401009   ret
main:
00401010   push ebp
00401011   mov ebp, esp
00401013   call sub (401000h)
00401018   mov eax,0F00Dh
0040101D   pop ebp
0040101E   ret
```

0x0012FF6C	0x004012E8
0x0012FF68	0x0012FFB8
0x0012FF64	0x00401018
0x0012FF60	0x0012FF68
0x0012FF5C	undefined
0x0012FF58	undefined

Example

We recover the frame pointer

eax	0x0000BEEF	ebp	0x0012FF68	esp	0x0012FF64

```
sub:
00401000   push ebp
00401001   mov ebp,esp
00401003   mov eax,0BEEFh
00401008   pop ebp
00401009   ret
main:
00401010   push ebp
00401011   mov ebp, esp
00401013   call sub (401000h)
00401018   mov eax,0F00Dh
0040101D   pop ebp
0040101E   ret
```

0x0012FF6C	0x004012E8
0x0012FF68	0x0012FFB8
0x0012FF64	0x00401018
0x0012FF60	undefined
0x0012FF5C	undefined
0x0012FF58	undefined

Example

We return from the subroutine

eax	0x0000BEEF	ebp	0x0012FF68	esp	0x0012FF68

```
sub:
00401000   push ebp
00401001   mov ebp,esp
00401003   mov eax,0BEEFh
00401008   pop ebp
00401009   ret
main:
00401010   push ebp
00401011   mov ebp, esp
00401013   call sub (401000h)
00401018   mov eax,0F00Dh
0040101D   pop ebp
0040101E   ret
```

0x0012FF6C	0x004012E8
0x0012FF68	0x0012FFB8
0x0012FF64	undefined
0x0012FF60	undefined
0x0012FF5C	undefined
0x0012FF58	undefined

Example

We change the value of eax

eax	0x0000F00D	ebp	0x0012FF68	esp	0x0012FF68

```
sub:
00401000   push ebp
00401001   mov ebp,esp
00401003   mov eax,0BEEFh
00401008   pop ebp
00401009   ret
main:
00401010   push ebp
00401011   mov ebp, esp
00401013   call sub (401000h)
00401018   mov eax,0F00Dh
0040101D   pop ebp
0040101E   ret
```

0x0012FF6C	0x004012E8
0x0012FF68	0x0012FFB8
0x0012FF64	undefined
0x0012FF60	undefined
0x0012FF5C	undefined
0x0012FF58	undefined

Example

We recover ebp register

eax	0x0000F00D	ebp	0x0012FFB8	esp	0x0012FF6C

```
sub:
00401000   push ebp
00401001   mov ebp,esp
00401003   mov eax,0BEEFh
00401008   pop ebp
00401009   ret
main:
00401010   push ebp
00401011   mov ebp, esp
00401013   call sub (401000h)
00401018   mov eax,0F00Dh
0040101D   pop ebp
0040101E   ret
```

0x0012FF6C	0x004012E8
0x0012FF68	undefined
0x0012FF64	undefined
0x0012FF60	undefined
0x0012FF5C	undefined
0x0012FF58	undefined

Example

We finish the main function

eax	0x0000F00D	ebp	0x0012FFB8	esp	0x0012FF70

```
sub:
00401000   push ebp
00401001   mov ebp,esp
00401003   mov eax,0BEEFh
00401008   pop ebp
00401009   ret
main:
00401010   push ebp
00401011   mov ebp, esp
00401013   call sub (401000h)
00401018   mov eax,0F00Dh
0040101D   pop ebp
0040101E   ret
```

0x0012FF6C	undefined
0x0012FF68	undefined
0x0012FF64	undefined
0x0012FF60	undefined
0x0012FF5C	undefined
0x0012FF58	undefined

Linux System Calls

System calls are the interface between user programs and the Linux Kernel.
In C there are several libraries that directly do a system call and made it transparent to the user.
Here we will focused on system calls that are perform using system interruption:

```
1   int     0x80
```

When we want to do a system call, there are some specifications we need to follow.

System Call Parameters

The registers are used to set the parameters of the system calls:

Syscall#	Par 1	Par 2	Par 3	Par 4	Par 5	Par 6
eax	ebx	ecx	edx	esi	edi	ebp

The return value is set on eax.

Possible System Calls

Some relevant system calls are:

eax	Name	ebx	ecx	edx
1	sys_exit	int *(code)*	-	-
2	sys_fork	struct pt_regs	-	-
3	sys_read	unsigned int *(input)*	char * *(buffer)*	size_t *(len)*
4	sys_write	unsigned int *(output)*	const char * *(msg)*	size_t *(len)*
5	sys_open	const char *	int	int
6	sys_close	unsigned int	-	-

Possible System Calls

The values of ebx in write and read are:

- 0: Stdin.
- 1: Stdout.
- 3: Stderr.

Hello World

Let's start with the first assembly program: Hello World

```
1  section     .text
2  global      main                    ;this is the entry point
3
4  main:                               ;start of the program
5
6      mov     eax,4                   ;system sys_write
7      mov     ebx,1                   ;file descriptor: stdout
8      mov     ecx,msg                 ;message
9      mov     edx,len                 ;message length
10     int     0x80                    ;call kernel
11
12     mov     eax,1                   ;system sys_exit
13     mov     ebx,0                   ;error code 0
14     int     0x80                    ;call kernel
15
16 section     .data
17
18 msg     db  'Hello, world!',0xa     ;our dear string
19 len     equ $ - msg                 ;length of our dear string
```

Program Description

The program is divided in two sections: text and data.

Text is related to the instructions while data is related to the variables that are going to be used during the program execution.

When you want to add uninitialized variables used section bss.

Text Section

The text section is read-only and starts with the entry point declaration. This specifies the first assembly label. After the label the program logic is as follows:

1. We specify the system call using the register eax (in this case write).
2. We specify the file descriptor using the register ebx
3. We move the message pointer to the register ecx
4. We move the message length the the register edx.
5. We call the kernel.
6. We specify a new system call (halt).
7. We call the kernel.

Data Section

The data section is read-write and defines the data used by the program. There are two variables

1. The message that we want to print
2. The message length.

Data Section

In the data section we can include variables, constants and strings.
Examples:

- Variable: var db 10
- Constant: var equ 100
- String: string db 'Hello',0

Bss Section

In the bss section we can include uninitialized variables. Examples:

- Undefined Variable: var: resb 1
- Undefined String with 100 length: string: resb 100

Compiling and Linking

Once the program has been created, we need to compile and link it.

For this task we use NASM and GCC (which links with ld).

NASM

The Netwide Assembler, NASM, is an 80x86 and x86-64 assembler.

It was designed for portability and modularity.

LD

The ld command combines several object files and libraries, resolves references, and produces an ouput file.

ld can produce a final linked image (executable, dylib, or bundle), or another object file.

We will use it through gcc compiler.

A compilation with ld:

```
$ nasm -f elf64 ex1.asm
$ ld ex1.o -o ex1
```

Or in our case:

```
$ nasm -f elf64 ex1.asm
$ gcc -o ex1 ex1.o
```

Adapting to 64 bit Architecture

There are specific differences in 64 bits architectures.
The most relevant are related to:

- The word size 32 to 64
- The registers
- The use of the stack

Registers in 64 bit Architecture

The Registers are:
The 16 integer registers are 64 bits wide and are called:

R0 R1 R2 R3 R4 R5 R6 R7 R8 R9 R10 R11 R12
R13 R14 R15 RAX RCX RDX RBX RSP RBP RSI RDI

(Note that 8 of the registers have alternate names.)
You can treat the lowest 32-bits of each register as a register itself but using these names:

R0D R1D R2D R3D R4D R5D R6D R7D R8D R9D R10D R11D R12D
R13D R14D R15D EAX ECX EDX EBX ESP EBP ESI EDI

Registers in 64 bit Architecture

You can treat the lowest 16-bits of each register as a register itself but using these names:

R0W R1W R2W R3W R4W R5W R6W R7W R8W R9W R10W R11W R12W
R13W R14W R15W AX CX DX BX SP BP SI DI

You can treat the lowest 8-bits of each register as a register itself but using these names:

R0B R1B R2B R3B R4B R5B R6B R7B R8B R9B R10B R11B R12B
R13B R14B R15B AL CL DL BL SPL BPL SIL DIL

64 bits: Calling C Functions

A program can also call C functions.

Using the 64 libraries, we can also compile the program with C libraries as stdio, stdlib, string, etc.

The way of calling a function changes from 32 bits to 64 bits.

Normally, 32 is more focused on passing the arguments using the stack while 64 uses the registers.

64 bits: Calling C Functions

```
1  global  main
2  extern  puts
3
4  section .text
5  main:                       ; C standard start
6    mov     rdi, message      ; Pointer to message
7    call    puts              ; puts(message)
8    ret                       ; Return from main
9  section .data
10 message:
11   db      "Hello again!", 0
```

64 bits: Calling C Functions

```
1  global main
2  extern printf
3
4  section .text
5  main:
6     mov rdi,msg          ;Message
7     mov rax,0          ;No special Registers
8     call printf          ;printf("%s",msg)
9     ret
10
11 section .data
12    msg: db "More random Strings",0
```

```
1  global main
2  extern printf
3  extern scanf
4  extern exit
5  section .text
6  main:
7          mov rax, 0
8          mov rdi, msg1
9          call printf
10
11         mov rax, 0
12         mov rdi, input
13         mov rsi, num
14         call scanf
15
16         mov rax, 0
17         mov rdi, msg2
18         mov rsi, [num]
19         call printf
```

```
1
2         mov rax, 0
3         mov rdi, 0
4         call exit
5   section .data
6   msg1 db "Introduce a number: ", 0
7   msg2 db "Your number is %d.", 0xa, 0
8   input db "%d", 0
9   num dq 0
```

Exercises

1. Create a program that prints a string introduced by the user. Do not use C functions.
2. Repeat this program using printf and scanf.
3. Create a program that asks the user for two integer numbers and calculates: the sum, substraction (first-second), division (first/second) and multiplication of them.
4. Separate the operations of the previous program in four functions and call them. Use registers to pass the parameters.