TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Systemarchitektur, Professur für Betriebssysteme

OPERATING-SYSTEM CONSTRUCTION

Material based on slides by Olaf Spinczyk, Universität Osnabrück

Exercise 4: Task #4, Assembler Programming

https://tud.de/inf/os/studium/vorlesungen/betriebssystembau

HORST SCHIRMEIER

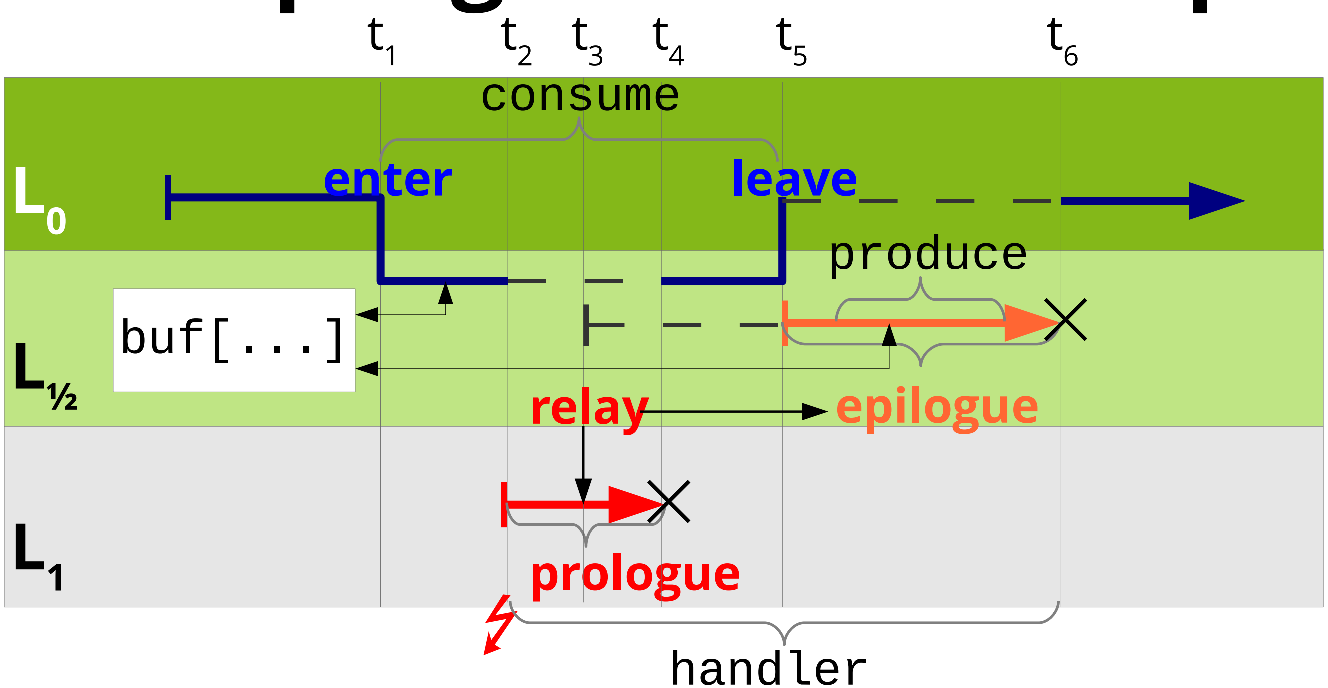
Overview

- Task #3: Tips & Tricks
- Task #4
  - Overview
  - x86-64 Assembler Programming
  - C / Assembler Interfacing

Overview

- **Task #3: Tips & Tricks**
- Task #4
  - Overview
  - x86-64 Assembler Programming
  - C / Assembler Interfacing

Pro/Epilogue Model – Sequence Example



L₁ interrupts are never disabled.

Interrupt-handler activation **latency is minimal**.

1 Application control flow enters epilogue level $L_{½}$ (**enter**).

2 Interrupt is signaled on level L_1, execute prologue.

3 Prologue requests epilogue for delayed execution (**relay**).

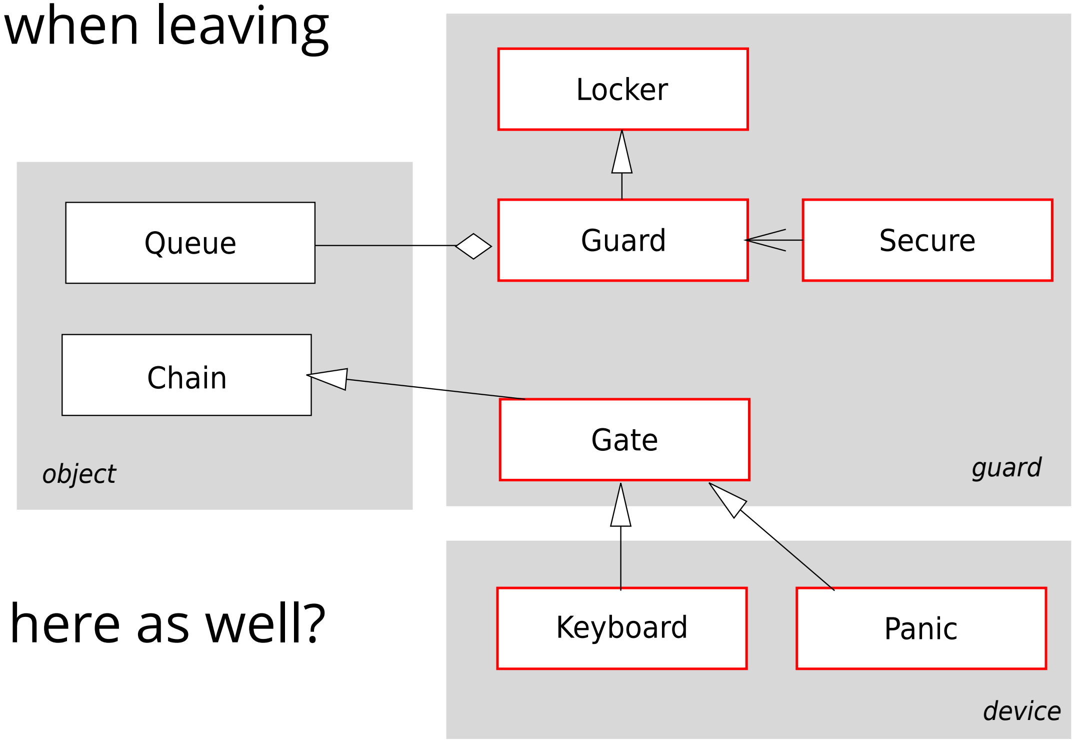
4 Prologue terminates, interrupted $L_{½}$ control flow (application) continues.

5 Application control flow leaves epilogue level $L_{½}$ (**leave**), process meanwhile accumulated epilogues.

6 Epilogue terminates, application control flow continues on L_0.

Task #3: Tips and Tricks

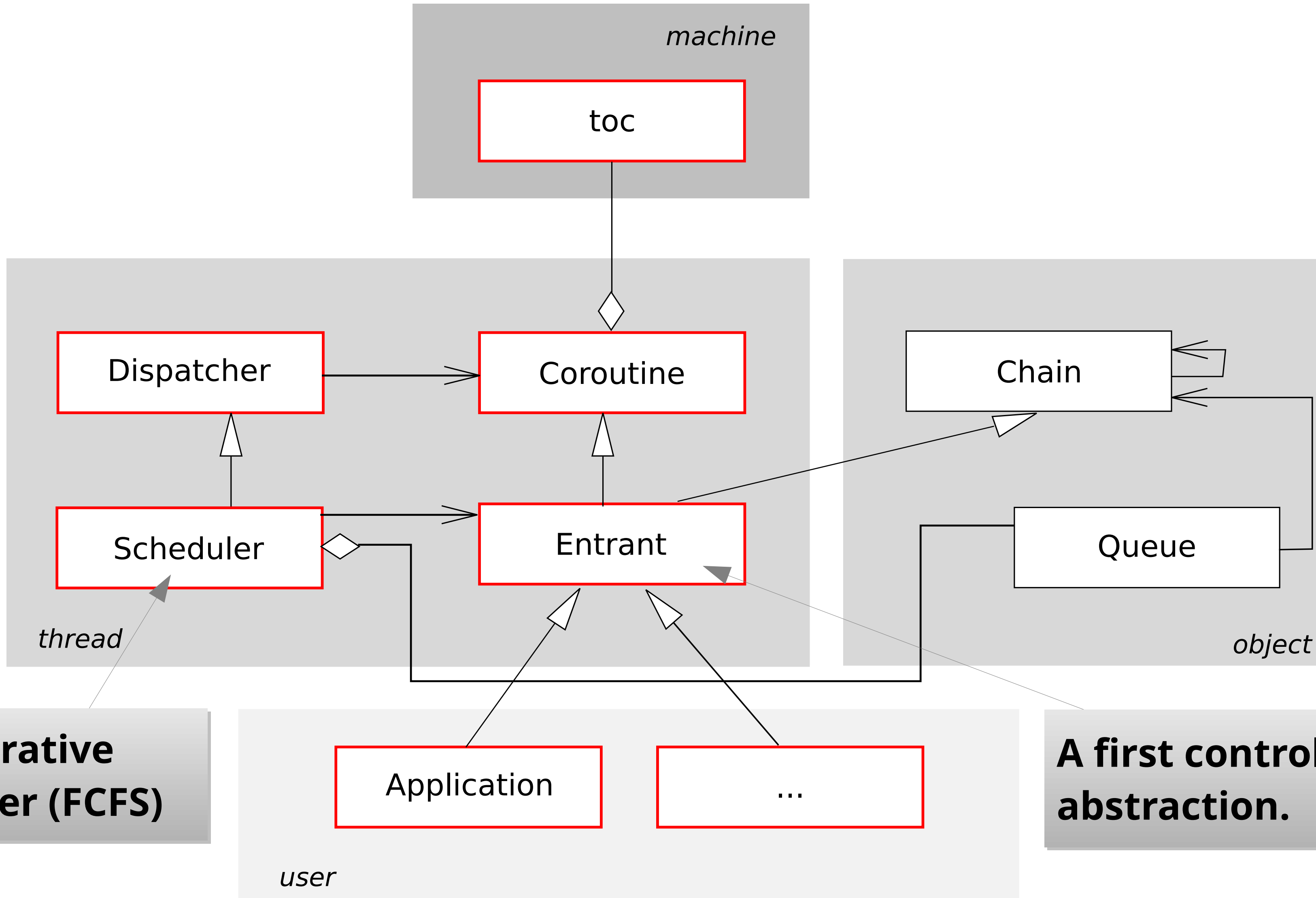
- Epilogue queue
  - Accesses must be synchronized! How?
- Guard::leave()
  - Which condition must hold when leaving this function?
- Gate::queued()
  - What's this there for?
- Interactions between prologue and epilogue
  - Do we need to synchronize here as well?



Overview

- Task #3: Tips & Tricks
- Task #4
  - **Overview**
  - x86-64 Assembler Programming
  - C / Assembler Interfacing

Task #4: Overview



A cooperative scheduler (FCFS)

A first control-flow abstraction.

Scheduler

Description

The scheduler manages the ready list (a private **Queue** member of this class), which is the list of processes of type **Entrant** that are ready to run. The list is processed from front to back. [...]

Public methods

void ready (Entrant& that)

This method registers the process that with the scheduler. It is appended to the end of the ready list.

void schedule ()

This method starts up scheduling by removing the first process from the ready list and activating it.

void exit ()

With this method a process can terminate itself. [...]

void kill (Entrant& that)

With this method a process can terminate another one (**that**). [...]

void resume ()

This method allows to trigger a context switch without the calling **Entrant** having to know which other **Entrant** objects exist in the system, and which of these should be activated. [...]

Overview

- Task #3: Tips & Tricks
- Task #4
  - Overview
  - **x86-64 Assembler Programming**
  - C / Assembler Interfacing

What is an Assembler?

- (Simple) compiler: transforms code of an assembler program → machine code
  - Assembler program = human-readable instructions
  - Machine code = binary representation of instructions (opcodes)
- More comfortable to write:
  - Instead of a bit string 01001000 00000101 11101000 00000011
    the programmer can write:
    `add rax, 1000`
- (Almost ...) bijective mapping:
  assembler instructions ⇔ binary machine-code instructions

Symbolic assembler instruction	Machine code
add rax	01001000 00000101
1000 (decimal)	00000011 11101000

- Each CPU architecture has its specific assembler.

What is an assembler capable of?

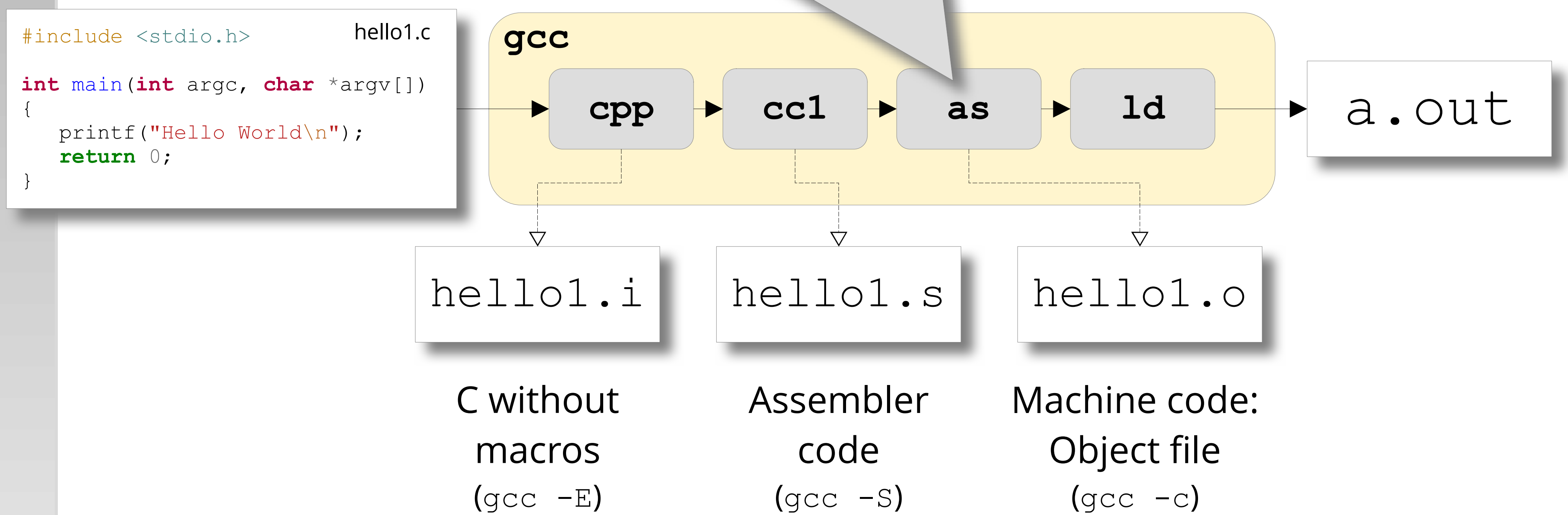
- Understands only a few complex expressions
  - Input language corresponds to **CPU instruction set**!
  - … sometimes additionally simple calculations and preprocessing at assembly time (see OOStuBS startup.asm, exercise #3)
- Constructs of higher programming languages are translated to simpler instructions by the compiler:
  - no complex statements
  - no comfortable loops – usually only “goto” equivalents
  - no structured data types
  - no subroutines with parameter passing

C/C++ Build Process

- Preprocessing,
  step: **gcc hel**
  - Generates file **a.**
    (name can be changed with para

Assembler: Component between compiler and linker

- Reads compiler-generated **assembler source code**
- Generates **object file**
  (binary machine instructions and data)



Example

- C statement: `sum = a + b + c + d;`
  - Too complex for the assembler, must be **broken down** to multiple steps!
- x86-64 assembler can only add *two* numbers and store the result in one of the two used “variables” (accumulator register)
- This C program is structurally closer to an assembler program:

```
sum = a;
sum = sum + b;
sum = sum + c;
sum = sum + d;
```

Example

- This program

```
sum = a;
sum = sum + b;
sum = sum + c;
sum = sum + d;
```

would look e.g. like this in x86-64 assembler:

```
mov     rax, [a]
add     rax, [b]
add     rax, [c]
add     rax, [d]
```

- An assembler …
  - supports only primitive operations
  - works in a line-oriented fashion (line = machine instruction)

Control structures: “if”

- Simple if-then-else constructs are already too complex for an assembler:

```
if ( a == 4711 )
{
    ...
} else {
    ...
}
```

- In x86-64 assembler, this looks as follows:

```
         cmp     rax, 4711    ; compare rax to 4711
         jne     unequal      ; unequal -> jump
equal:   ...                 ; else continue here
         jmp     cont        ; skip over else branch
unequal: ...                 ; else branch
cont:    ...                 ; continue with other stuff
```

Loops: Simple “for” Loop

- A simple counting loop is actually better supported:

```
for (i = 0; i < 100; i++) {
   sum = sum + a;
}
```

- … in x86-64 assembler:

```
          mov   rcx, 100
repeat:   add   rax, [a]
          loop  repeat
```

- `loop` instruction:
  - Implicitly decrements RCX register
  - Jumps only if RCX ≠ 0

What is a Register?

- Extremely fast, very small storage within the CPU that can (in x86-64 CPUs) store 64 bits
- Compiler: Mapping of high-level language variables to storage locations in the data/BSS segment of an object file
- Calculations with variables: Usually beforehand loading memory→register necessary
  - Not all variables fit into the low number of registers at the same time!
  - Mapping registers ⇔ variables changes over time

8086: Register File

Instruction and Stack Pointer

15	0
IP	
SP	

Flags register

15	0
FLAGS	

General-purpose registers

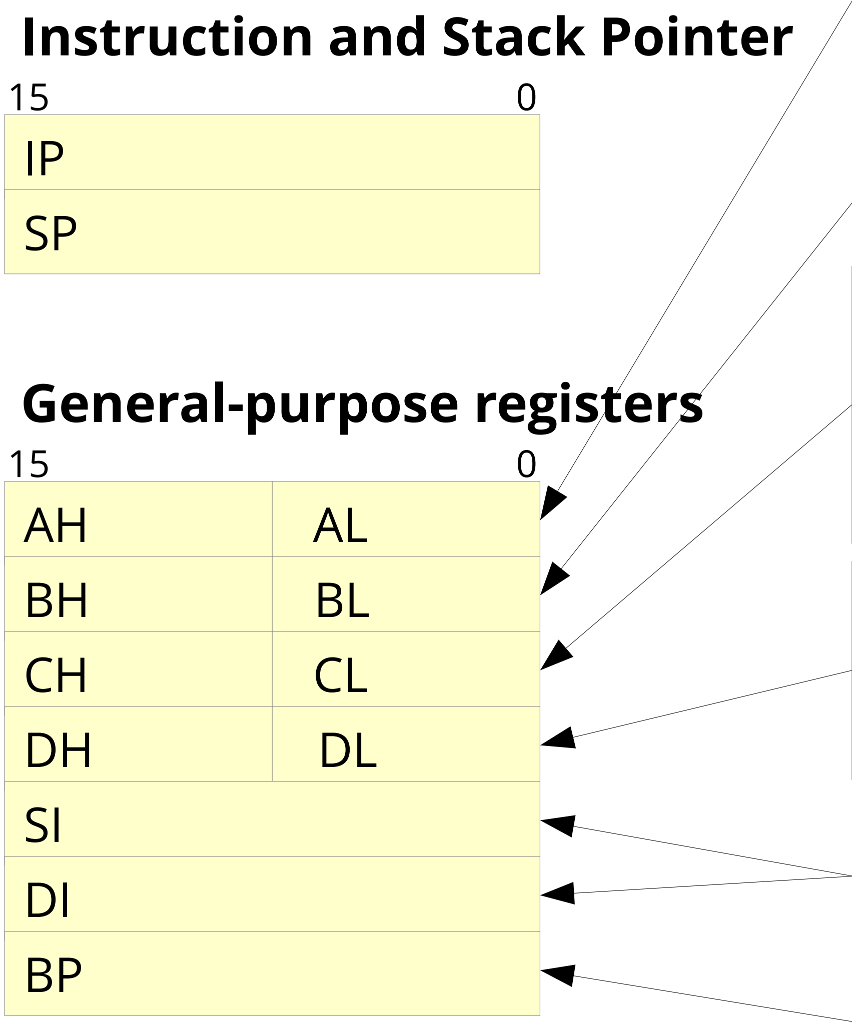
15	0
AH	AL
BH	BL
CH	CL
DH	DL
SI	
DI	
BP	

Segment registers

15	0	
CS		Code
SS		Stack
DS		Data
ES		Extra

Each **“general-purpose” register** fulfills a specific purpose

8086: Register File



AX: Accumulator Register
- arithmetic + logical operations
- I/O
- shortest machine code

BX: Base Address Register

CX: Count Register
- for LOOP instruction
- for string operations with REP
- for bit-shift and rotate

DX: Data Register
- DX:AX have 32 bits for MUL/DIV
- port number for IN and OUT

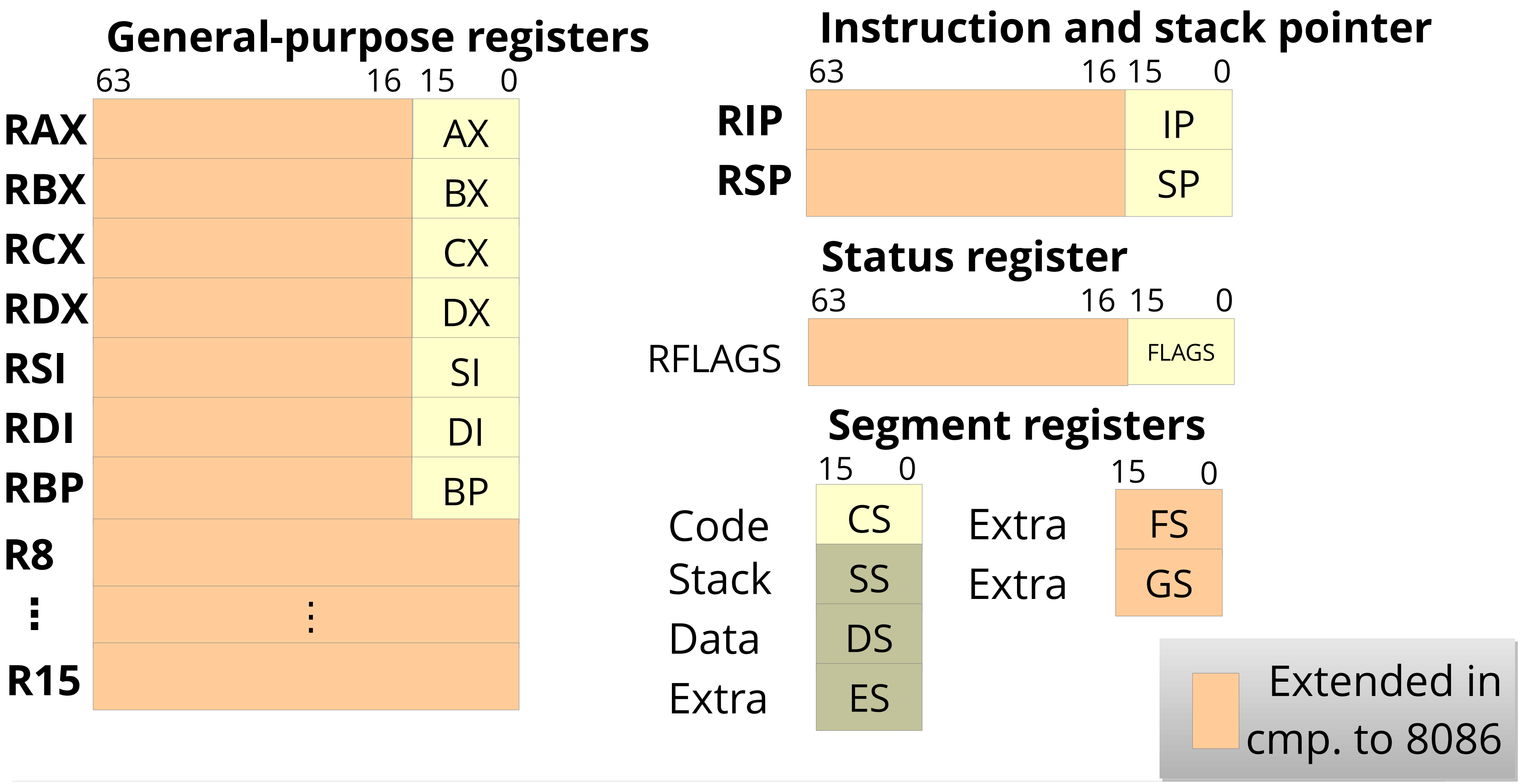
SI, DI: Index Register
- for array accesses (displacement)

BP: Base Pointer

TECHNISCHE UNIVERSITÄT DRESDEN

x86-64: Register File (Extensions)

- Extended registers prefixed with R… for compatibility



Memory

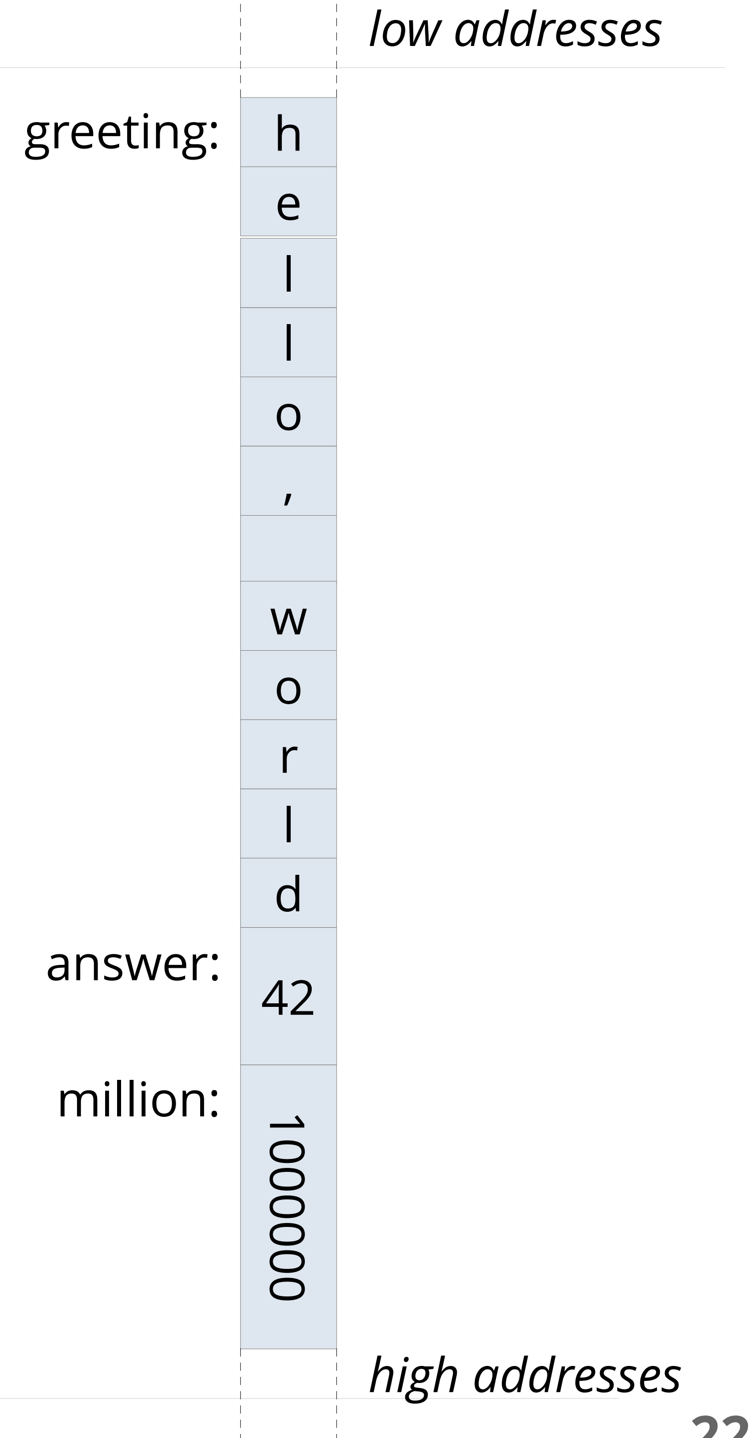
- In most of the cases, registers do not suffice to implement an algorithm
  - Memory access is necessary
- Main memory: Functionally like a gigantic array of registers, selectively 8, 16, 32 or 64 bits “wide”
  - smallest addressable unit: Byte
  - memory cells numbered consecutively → index
  - accesses are **several 100x slower** than to registers
- Access via **addresses**

Memory

- Example:

```
[SECTION .data]
greeting:    db 'hello,
world'
answer:      dw 42
million:     dd 1000000

[SECTION .text]
             mov ax,
[million]
```



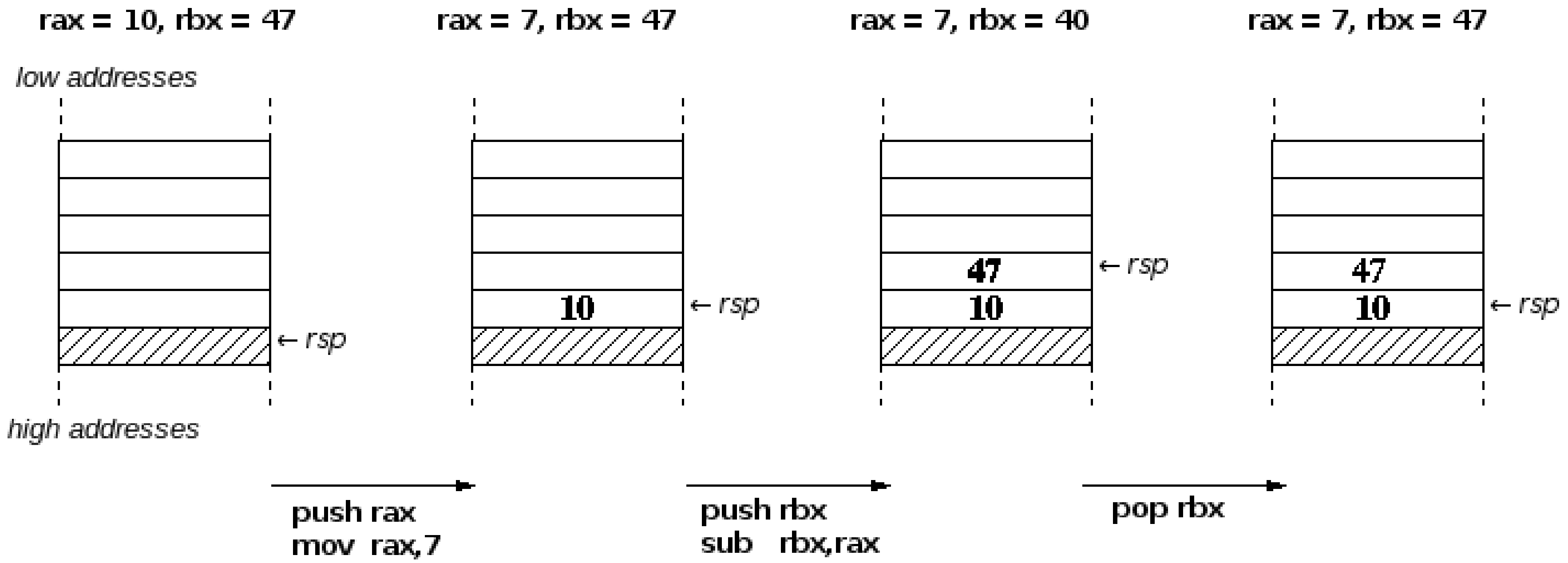
A bug hides here:
It should say `mov eax, [million]`

The Stack

- Variables stored at fixed memory addresses are accessible from all parts of the assembler program
  - via address or symbolic names (“labels”) → **global variables**
- However, for particular purposes we need non-global variables
  - Isolation between functions / objects
  - Recursively callable functions
- **Stack:** Temporary LIFO storage for values “as long as they are needed”
  - allows **dynamic allocation** of variables
  - addressed with **relative addresses**

The Stack

- **Push** operation: Store values “on top” of the stack (inverse: **Pop**)
  - memory address at which push/pop operate: special register, the so-called **stack pointer** (x86-64: **rsp**)
  - No need to care about concrete value of stack pointer; only remember **order** in which we pushed values!



Addressing Modes

- Most instructions can use **registers**, **memory**, or **constants** as operands
- The mov instruction allows the following modes (among others) (1st operand: target, 2nd operand: source):
  - **Register addressing** – transfer value of a reg. to another: `mov rbx, rdi`
  - **Immediate** – transfer a constant to a register: `mov rbx, 1000`
  - **Direct memory addressing** – transfer the value stored at the address (supplied **as a constant**) to a register: `mov rbx, [1000]`
  - **Register indirect** – transfer the value stored at the address (supplied **in a register**) to a register: `mov rbx, [rax]`
  - **Direct offset addressing** – transfer the value stored at the address (supplied **as a sum of a constant and an address**) to a register: `mov rax,[10+rsi]`

x86-64: Addressing Modes

- The CPU calculates **effective addresses** (EA) along a simple formula
  - all general-purpose registers can be used equally (!)

EA := Base-Reg. + (Index-Reg. * Scale) + Displacement

Scale: **1/2/4/8**

Displacement: **---**, **1/2/4 bytes**

- Example: `MOV RAX, array[RSI * 4]` (EA)
  - Read from array with 4-byte elements, using RSI as index
- New with x86-64: IP-relative addressing

EA := RIP + Displacement

Functions

- … known from higher-level programming languages …
  - Advantage compared to **goto**: Call from arbitrary location in your program, return/continue the calling program part
  - The function itself doesn't need to know where it was called from, and where to return afterwards (this happens automatically – how?)
- Not only data but also your program lies in main memory

  → each machine-code instruction has its own address
- Special **Instruction Pointer** register (**rip**) points to the next instruction to be executed

Functions

- Processor executes instruction, then usually increases **rip** by the length of the instruction

  → **rip** points to the next instruction
- **Jump instruction:** Changes **rip** to target address (absolute, or rip-relative)
- **Function call:** like a jump, plus saves the **return address**
  - old **rip** value (plus instruction length) is saved on the stack
- **Function return:** `ret` pops address from stack, jumps there

Functions

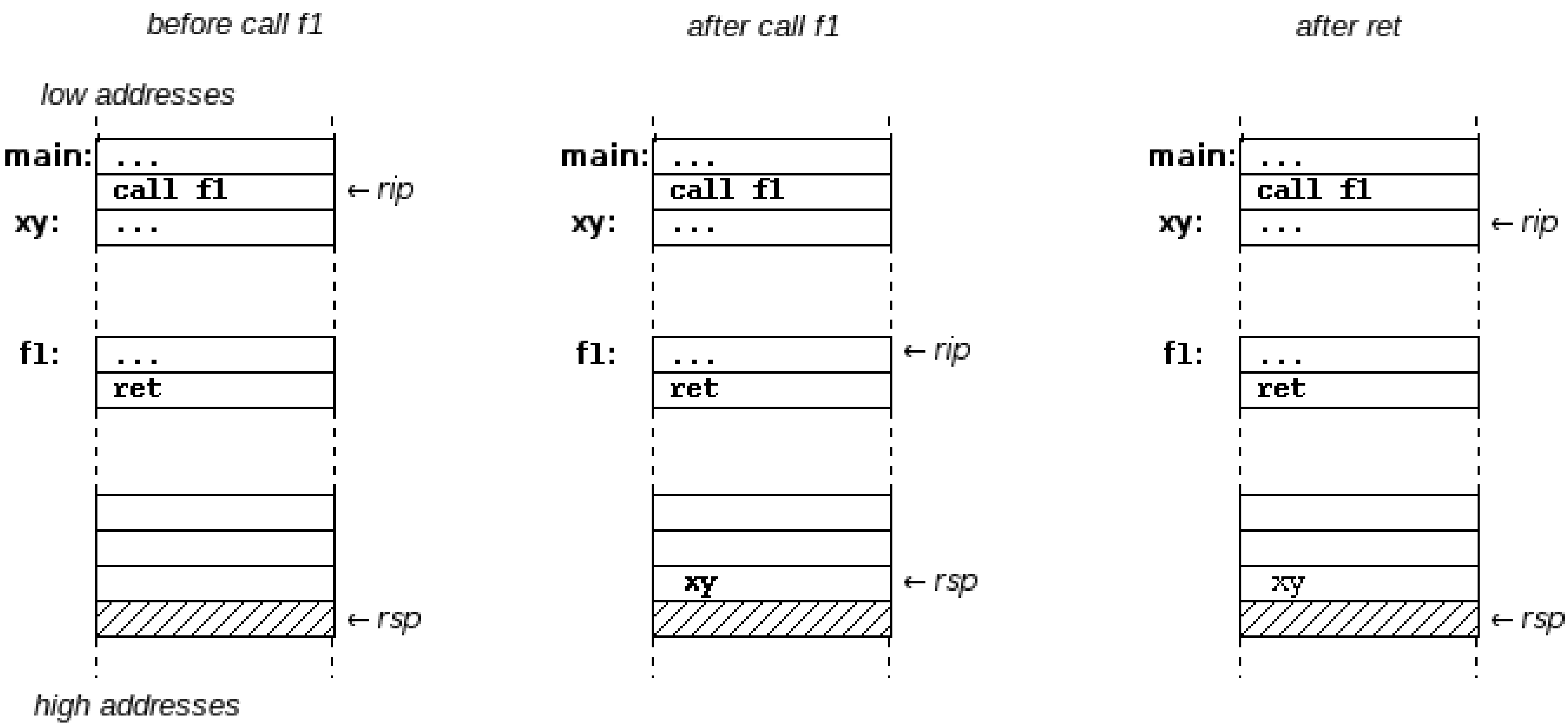
- x86-64: Implicitly save/restore the return address on the stack by using the **`call`** and **`ret`** instructions

```
; ----- Main program ----
;
main: ...
      call f1

xy:   ...

; ----- Function f1
f1:   ...
      ret
```

Functions


before call f1
after call f1
after ret
low addresses
main:
...
call f1
xy:
...
f1:
...
ret
← rip
← rsp
xy
high addresses

Functions

- Parameters: the first 6 in registers, further ones on the stack
- Parameters on the stack must be removed again afterwards (with `pop`, or by directly modifying **rsp**)

```
mov   rdi,rax   ; first parameter for f1 in rdi
mov   rsi,rbx   ; second parameter in rsi
mov   rdx,r13   ; third parameter in rdx
; …               …
push  r15       ; seventh parameter on the stack
call  f1
add   rsp, 8    ; remove seventh param. from stack
```

Functions

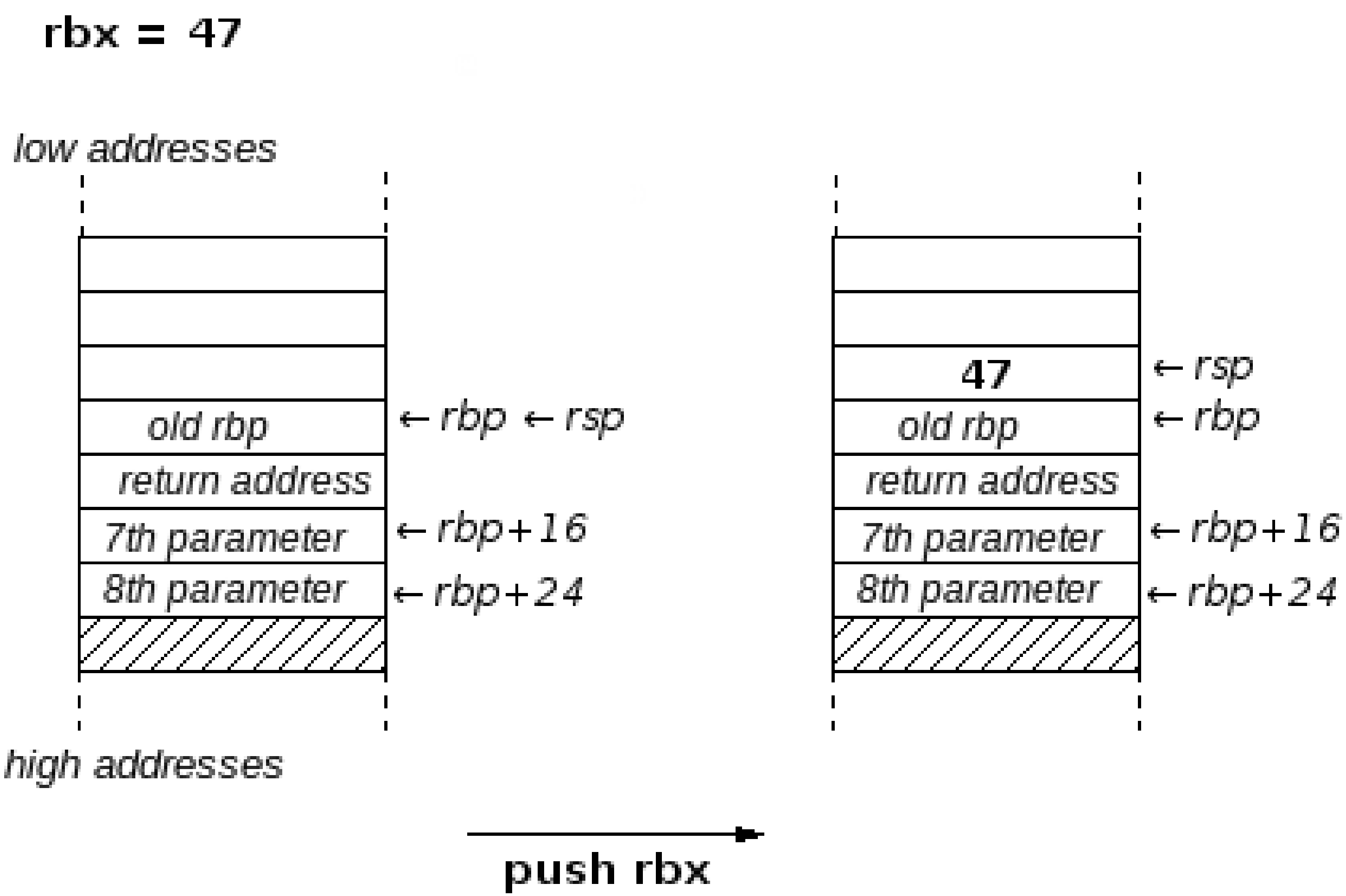
- Access to parameters *within the function*:
  - Simplified by using the **base pointer rbp**
  - Convention: Save **rbp** at the beginning of a function, set to **rsp**

    → access the 7th parameter via [rbp+16]

    → access the 8th parameter via [rbp+24] …
  - … independently from whether **rsp** was changed in the meantime (e.g. using **push** or **pop**)

```
f2:    push rbp
       mov  rbp,rsp
       ...
       mov rbx,[rbp+16]    ; load 7th parameter to rbx
       mov rax,[rbp+24]    ; load 8th parameter to rax
       ...
       pop rbp
       ret
```

Functions

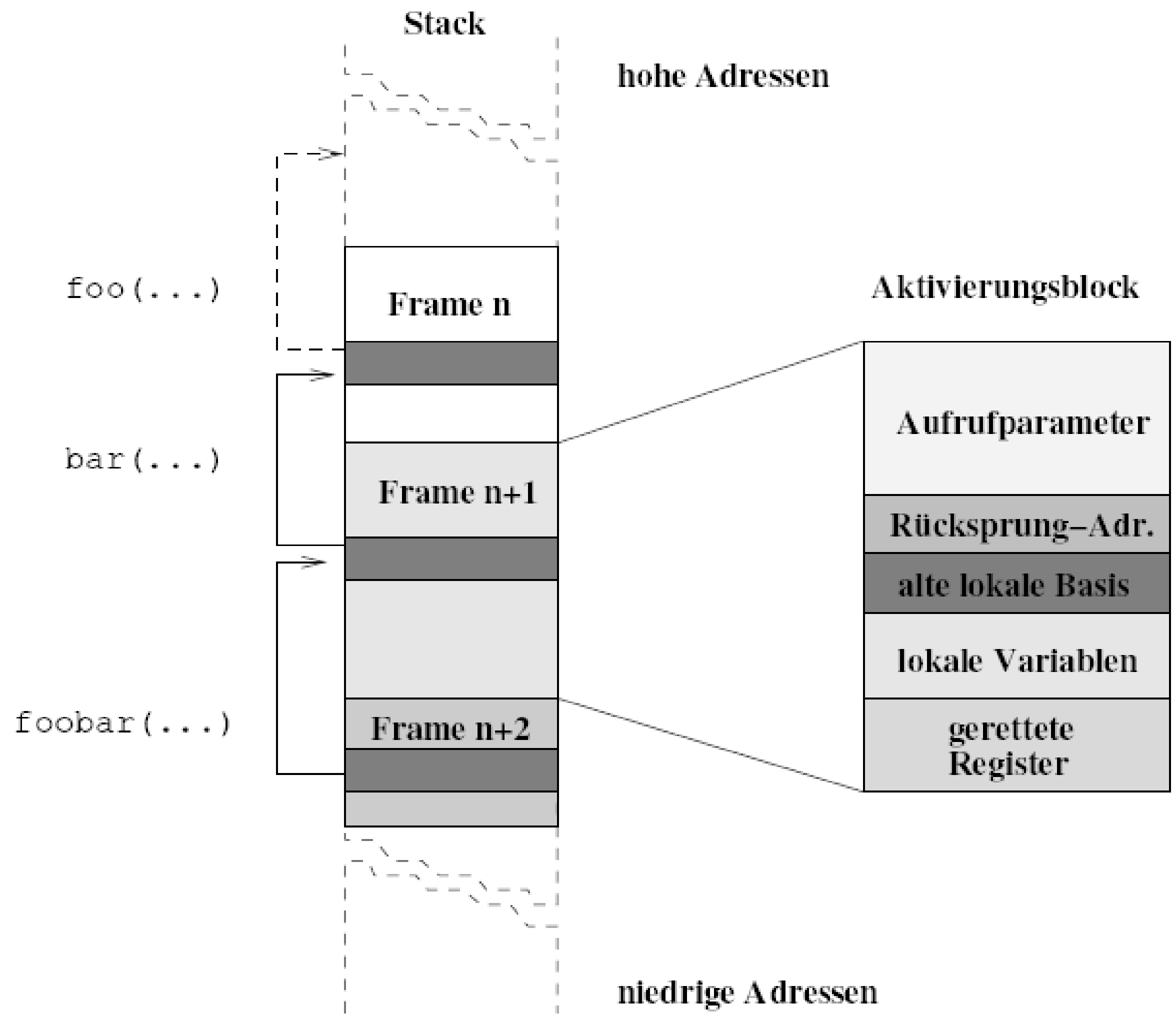

rbx = 47
low addresses
old rbp
return address
7th parameter
8th parameter
← rbp ← rsp
← rbp+16
← rbp+24
high addresses
push rbx
47
← rsp
old rbp
← rbp
return address
7th parameter
← rbp+16
8th parameter
← rbp+24

Nested Function Calls

```
void foobar(int x)
{
      ...
}

void bar(int x, int y, int z)
{
    int a, b;
    ...
    foobar(a+b);
    ...
}

void foo(int x, int y)
{
    int a, b, c;
    ...
    bar(a+b, x, c);
    ...
}
```



Overview

- Task #3: Tips & Tricks
- Task #4
  - Overview
  - x86-64 Assembler Programming
  - **C / Assembler Interfacing**

Calling Assembler Functions

- An assembler-code label can be exported to the linker – also a function address:

```
; EXPORTED FUNCTIONS
[GLOBAL toc_switch]
[GLOBAL toc_go]
toc_go: ...
```

- Now a C++ program can call the function
  - However, the compiler needs a (matching) declaration:

    ```
    extern "C" void toc_go(struct toc* regs);
    ```

- The assembler code can expect the parameter in **rdi**.
- Non-volatile registers may need to be saved/restored!