Assembly Functions

Systems Programming

Michael Sonntag
Institute of Networks and Security

What is a „function“?

- **Name**
  - Symbol representing the address where the function begins
- **Parameters**
  - Input; arbitrary number and type should be supported
  - Sometimes also output or input&output parameters desirable
- **Local variables**
    - Often called “automatic variables”
  - Private data storage
  - Not accessible from outside
  - “Thrown away” when function ends
- **Global variables**
  - Public data storage
  - Accessible from inside and outside
  - Survive end of function (and already exist before it starts)

What is a „function“?

- **Return address**
  - Invisible “parameter”
  - Tells program where to resume executing after function completed
  - In most progr. languages, return address is handled automatically
    - Even in most assembly languages, e.g. X86-*
- **Return value**
  - Transfer result back to caller
  - Most programming languages allow only a **single** return value
    - Most also require that it fits in a **single register** (or use one of the circumventions listed below themselves to emulate this)
  - More than one/larger return value needed?
    1. Use pointers to data (i.e. memory addresses of data)
    2. In the function change/set (parameter) values that a pointers point to
    3. Read values in caller after function returns
    - Pointer to data as direct return value
      - Careful: who reserved the memory, who frees it?
  - Can be “extended” through in+out parameters

Stack

- Where do we store return addresses, parameters, local variables…?
  - Sometime also large return values (structures)
- In a special memory area called the "Stack"
- We don't know how "deep" functions will be called (recursion!), so we cannot set an upper limit for the stack size
  - Practically, most OS **do** set a limit → if reached, the program is terminated (to prevent e.g. runaway recursion). But until then only as much memory is provided as is actually used!
- Dangerous from the security point of view
  - Attackers can manipulate it: arbitrary data at arbitrary position (within the stack)
  - Might be executed (but see modern precautions!)

Stack

- Region at the "top" addresses of memory (of current process)
  - Stack is separate for each process
  - Multiple threads: separate stack for each thread
    - Obviously not at the very top any more, as threads share memory, so stack size limits are more stringent (relocating a stack is impossible!)
    - On 64 Bit computers the logical address space is very large, so this is less a problem
- Stack **grows down** in memory (from high towards low addresses)
- Limited in size: everything larger than a few kB should be put on Heap
- Used to **implement functions**
  - Save state of the caller
  - Pass parameters to the callee
  - Store return address
  - Store local variables
  - Store large return data (reserved by caller)

Stack

- **RSP register** always points to the “top” of the stack
  - =**Lowest** used address
    - For a quadword (=8 bytes) it is the “first” byte (=lowest address of the 8)
- Push data (**`push`**)
  - “Decrement” RSP register to create space
    - How much? As many bytes as the new data item is long!
  - Write data item at new top
- Pop data (**`pop`**)
  - Read data from top (and store it in some place, typ. some register)
  - “Increment” RSP register
    - How much? As many bytes as we remove – this need **NOT** be the same as was used on push!
- RSP can also be modified by **`SUB/ADD`**, e.g. for creating/destroying local variables (remember: stack grows down, so **`SUB`** **creates** space!)
  - Then the “new content” is **NOT** initialized (=old values)!

JKU INSTITUTE OF NETWORKS AND SECURITY

Calling conventions

- How to call a functions:
  - How do we pass parameters? Any metadata (=type) for them?
    - Which registers? Or stack only?
    - Where do we put the “this” pointer of object-oriented languages?
  - What about local variables?
    - And who cleans up after them?
  - What about the return value (“A” register, stack…)?
  - Which register may be used in the function (=who saves them)?
  - Return address (x86 → hardware restrictions!), frames…
- This is a „**calling convention**“
  - „Convention“ because there are few technical restrictions
  - Every programming language (or programmer) can decide on her own what to do/how to do it
  - You can even mix them in a single program
  - BUT: However a function is programmed, this function must be called in an exactly matching way!

Calling conventions: cdecl

- C declaration → Original C programming language
  - All parameters are passed on the stack
  - Stored in reverse order (last parameter is pushed first)
  - Return value in A (=EAX) register
  - Registers A, C, D (=EAX, ECX, EDX) are caller saved, rest is callee saved
  - EBP register used for frame pointer
  - Caller cleans up stack
  - Linux modification: stack must be 16-Byte aligned on function call
  - Used in Linux-x86-**32** Bit
  - Typical C declaration: `void _cdecl funct();`

Specify the calling convention to use for this function

Calling conventions: Others

- Pascal: Pascal programming language
  - Similar to cdecl, but parameters are pushed on the stack in normal order, which prevents functions with variable count of parameters
  - Callee has to clean up stack
  - Used by Windows 16 Bit (Windows 3.x)
- Stdcall: Similar to Pascal
  - Parameters are pushed in reverse order, like in cdecl
  - Standard calling convention for Windows 32 Bit
- Microsoft fastcall: `__fastcall`
  - First two argument are passed in ECX and EDX, rest on stack in reverse order
  - Windows 32 Bit (depending on compiler; used for optimization)

Calling conventions: Microsoft X64

- RCX, RDX, R8, R9 are used for the first four parameters, the rest are pushed on the stack in reverse order
  - Smaller values are right-justified in registers (=lower bits)
- Return value is in RAX (smaller values do **not** set upper bits to 0!)
- RAX, RCX, RDX, R8-11 are caller-saved
- RBP, RBX, RSI, RDI, R12-15 are callee-saved
- Caller must always reserve 32 Byte of space on stack (shadow space) for the first four parameters (even if less used!)
  - Note that only space is provided, the parameter values are **not** written there by the caller – they are **only** in the registers!
- Stack pointer must be aligned to 16 Bytes
- Used on Windows 64 Bit

Calling conv.: SystemV AMD64 ABI

- SystemV (old Unix version; 1983), AMD64 (“original” 64 Bit version of IA32), ABI (“Application Binary Interface” ≈ calling convention+…)
- First six parameters are passed in RDI, RSI, RDX, RCX, R8, and R9, the rest are pushed on the stack in reverse order
    - Fewer parameters? Function can use them freely (caller-saved!)
  - Linux Kernel calls: RCX replaced by R10; max. 6 parameters; no stack ever used (except as memory for where pointers point to)
  - Syscalls may (will) always destroy RCX and R11
- Return value is stored in RAX (+ potentially RDX for long values)
- RAX, RCX, RDX, RSI, RDI, R8-R11 are caller-saved
  - So if it is a procedure and not a function (=no return value), the function can use RAX freely
- RBP, RBX, R12-R15 are callee-saved

Matz, Hubička, Jaeger, Mitchell (Eds.), "System V Application Binary Interface: AMD64 Architecture Processor Supplement“, http://chamilo2.grenet.fr/inp/courses/ENSIMAG3MM1LDB/document/doc_abi_ia64.pdf

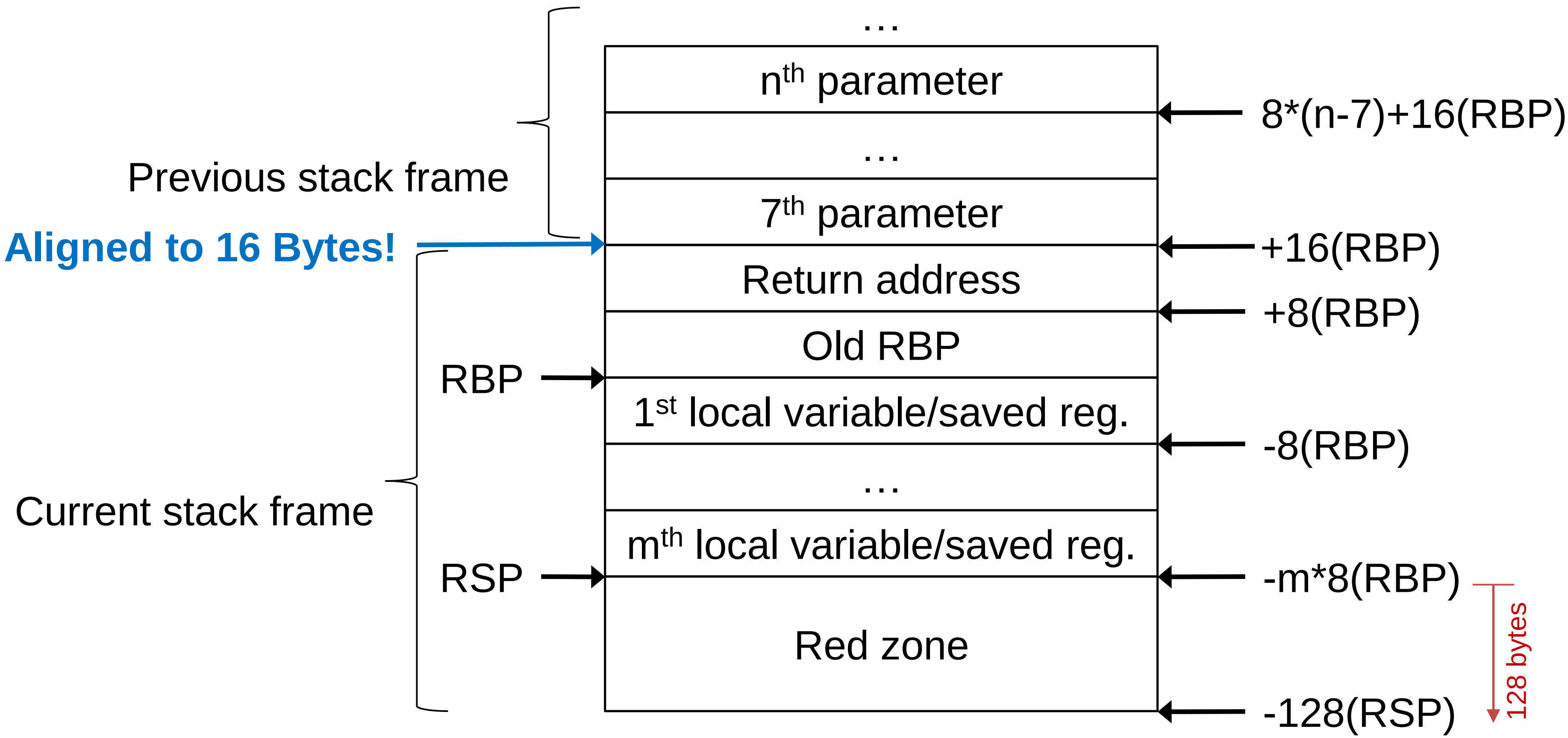
Calling conv.: SystemV AMD64 ABI

- Stack pointer must be aligned to 16 Bytes (=end of parameters)
- 128 Bytes below RSP are guaranteed to exist and can be used by function for e.g. local variables without RSP adjustment ("Red zone")
  - Not used by signals and interrupt handlers
  - Will be overwritten by function calls → useful for "leaf" functions
    - "Leaf" function = function that doesn't call **any** other function
  - Optimization purpose: use without adjusting RSP
- Direction Flag (DF) must be cleared (=forward) on entry and exit
- Used on **Linux 64 Bit**, MacOS…

Matz, Hubička, Jaeger, Mitchell (Eds.), "System V Application Binary Interface: AMD64 Architecture Processor Supplement", https://uclibc.org/docs/psABI-x86_64.pdf

Stack frame

- How does the stack look like when a function is called?
  - For each function (without optimizations used!) that is called, a "frame" is reserved



Calling a function – Caller

- Caller adjusts stack pointer so it will end up at 16-Byte alignment after the next 3 bullets

  ↕ **Can be exchanged**

- Caller saves (typically on stack) all caller-save registers - **if needed**
- Caller pushes parameters N to 7 on stack in reverse order
- Caller puts parameters 1 to 6 in the appropriate registers
- Caller executes the **`call ...`** instruction
  - Which pushes address of the next instruction (=RIP) on the stack
    - This is the "return address", where execution resumes after the function returns
  - The RIP register is modified to contain the address specified in the call instruction – which is the start of the function

Calling a function – Callee - Prologue

- Callee saves the base pointer on the stack
  - "Trace back" to the previous function; used e.g. by debuggers
  - **`push %rbp`**
- Callee copies the stack pointer into the base pointer
  - **`movq %rsp,%rbp`**
- Callee subtracts amount of bytes needed for local variables from RSP
  - **`subq $???,%rsp`**
    - If less than 128 needed (with registers!), the red zone can be used
- Callee pushes all callee-save registers to be used on the stack

- Now the actual function begins
  - Parameters: RBP+16 (7th parameter) and up
  - Local variables: RBP-8 (1st local variable) and down
  - Callee stores return value in RAX (if function and not procedure)

Calling a function – Callee - Epilogue

- Callee restores all callee-saved registers
- Callee resets the stack to remove the local variables
  - Should be done even if the red zone was used
  - Sometimes used to restore stack in case of exceptions
    - Scanning the code for the epilogue instructions to correctly unwind it
  - **`movq %rbp,%rsp`**
- Callee restores the old frame pointer
  - **`popq %rbp`**
- Callee returns to calling program
  - **`ret`**

Calling a function – Back at Caller

- All local variables have been destroyed
  - **Never** attempt to return a pointer to a local variable!
- Future stack pushes will overwrite the values
  - Some might still be accessible, as the red zone of the current stack frame might cover them (partially)
  - The red zone is purely an optimization!
    - Need not set RBP to RSP and need not subtract from RSP
    - **RSP** is used as base for all parameters and local variables
    - RBP is no longer needed and can be used as normal register (but: callee-saved!)
    - Can be used for temporary data and local variables
    - **Careful! Further function calls start from RSP and NOT from the end of the red zone, so calling a function will destroy data in it!**
- Return value is in RAX (plus potentially RDX)
- Caller has to remove parameters 7 to N from stack
  - `addq $`(N-6)*8`,%rsp` or N-6 times `popq %`<some register>
- Caller restores any saved caller-save registers – **if done**
- Caller re-adjusts stack if any adjustment for alignment was made (or ignores this and just wastes the space until it is cleaned up when it returns itself and resets the stack)

↕ **Can be exchanged**

Program start

- When Linux starts a program, how does the CPU content look like, where does it begin, how are parameters provided…?
  - Do not assume specific content of registers, unless noted below
    - Flags do have defined content, but for security set them explicitly
  - RSP points to the end of the stack
    - (%RSP) → Number of arguments
    - 8(%RSP) → Pointer to first argument (= program name)
      - This is a **pointer**. This is **not** the string, but the **address** of the **first character** of the string!
    - 16(%RSP) → Pointer to second argument (= first parameter), if present
    - Higher on stack: more parameters, process environment and other data
      - Not used in this course!
  - RDX: function pointer to specify an exit procedure
    - Not used in this course! Simply ignore it (and use the register)
  - Program entry point: "`_start`"
    - Exactly this name, cannot be changed

Calling a function – Example

- We will call the following function:
  - ```
    int doSomething(int p1, int p2, int p3,
                    int p4, int p5, int p6,
                    int p7, int p8, int p9)
    ```
  - 9 parameters: p1, p2, …, p9 (64-bit integer each)
  - 1 return value (64-bit integer)
- Example for calling this function:
  - `if (doSomething(1,2,3,4,5,6,7,8,9) != 0) { ... }`
  - Calling the function puts the return address on the stack
  - Caller is responsible for passing parameters in the right place
    ```

Calling a function – Example

- We will call the following function:
  - `int doSomething(int p1, int p2, int p3, int p4, int p5, int p6, int p7, int p8, int p9)`
  - 9 parameters: p1, p2, …, p9 (64-bit integer each)
  - 1 return value (64-bit integer)
- Example for calling this function:
  - `if (doSomething(1,2,3,4,5,6,7,8,9) != 0) { ... }`
  - Calling the function puts the return address on the stack
  - Caller is responsible for passing parameters in the right place

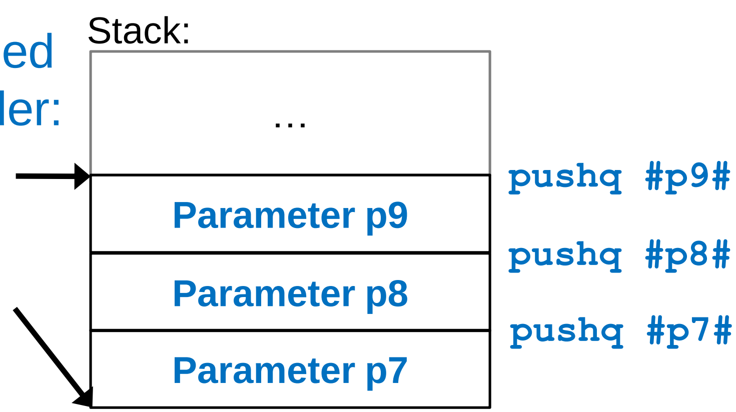
The first 6 parameters are stored in **registers**:

- p1 → RDI
- p2 → RSI
- p3 → RDX
- p4 → RCX
- p5 → R8
- p6 → R9

Calling a function – Example

- We will call the following function:
  - `int doSomething(int p1, int p2, int p3,`
    `int p4, int p5, int p6,`
    `int p7, int p8, int p9)`
  - 9 parameters: p1, p2, …, p9 (64-bit integer each)
  - 1 return value (64-bit integer)
- Example for calling this function:
  - `if (doSomething(1,2,3,4,5,6,7,8,9) != 0) { ... }`
  - Calling the function puts the return address on the stack
  - Caller is responsible for passing parameters in the right place

Further parameters are pushed onto the **stack** in reverse order:



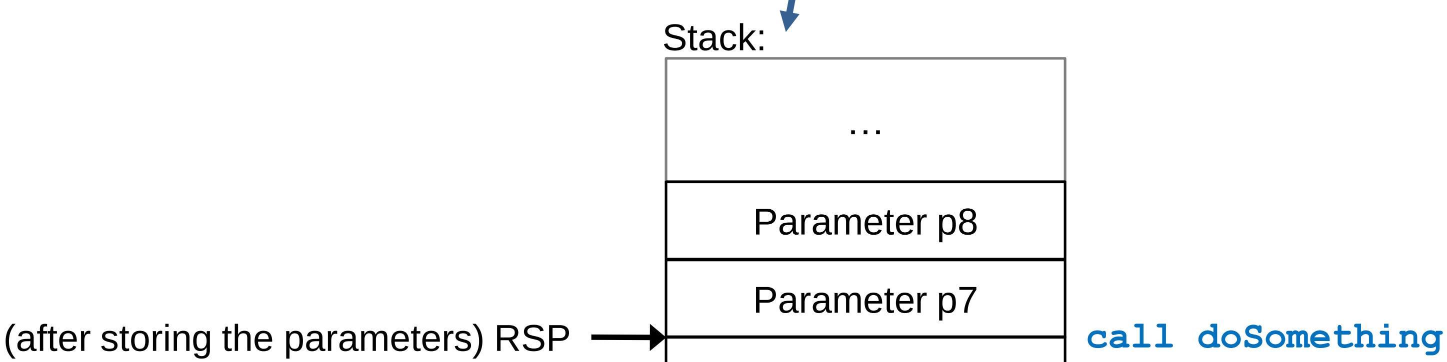
Calling a function – Example

- We will call the following function:
  - `int doSomething(int p1, int p2, int p3,`
    `int p4, int p5, int p6,`
    `int p7, int p8, int p9)`
  - 9 parameters: p1, p2, …, p9 (64-bit integer each)
  - 1 return value (64-bit integer)
- Example for calling this function:
  - `if (doSomething(1,2,3,4,5,6,7,8,9) != 0) { ... }`
  - Calling the function puts the return address on the stack
  - Caller is responsible for passing parameters in the right place

Return value will be stored in **register** RAX (by the function)

Calling a function – Example

- We will call the following function:
  - `int doSomething(int p1, int p2, int p3,`
    `int p4, int p5, int p6,`
    `int p7, int p8, int p9)`
  - 9 parameters: p1, p2, …, p9 (64-bit integer each)
  - 1 return value (64-bit integer)
- Example for calling this function:
  - `if (doSomething(1,2,3,4,5,6,7,8,9) != 0) { ... }`
  - Calling the function puts the return address on the **stack**
  - Caller is responsible for passing parameters in the right place



Calling a function – Example

- Internally, the function will also need:
  - Registers: RBX, R10, R11, R12
  - Two 8-byte values as local variables

Calling a function – Example

- Internally, the function will also need:
  - Registers: RBX, R10, R11, R12
  - Two 8-byte values as local variables

Callee-saved

➔ Caller does not need to do anything

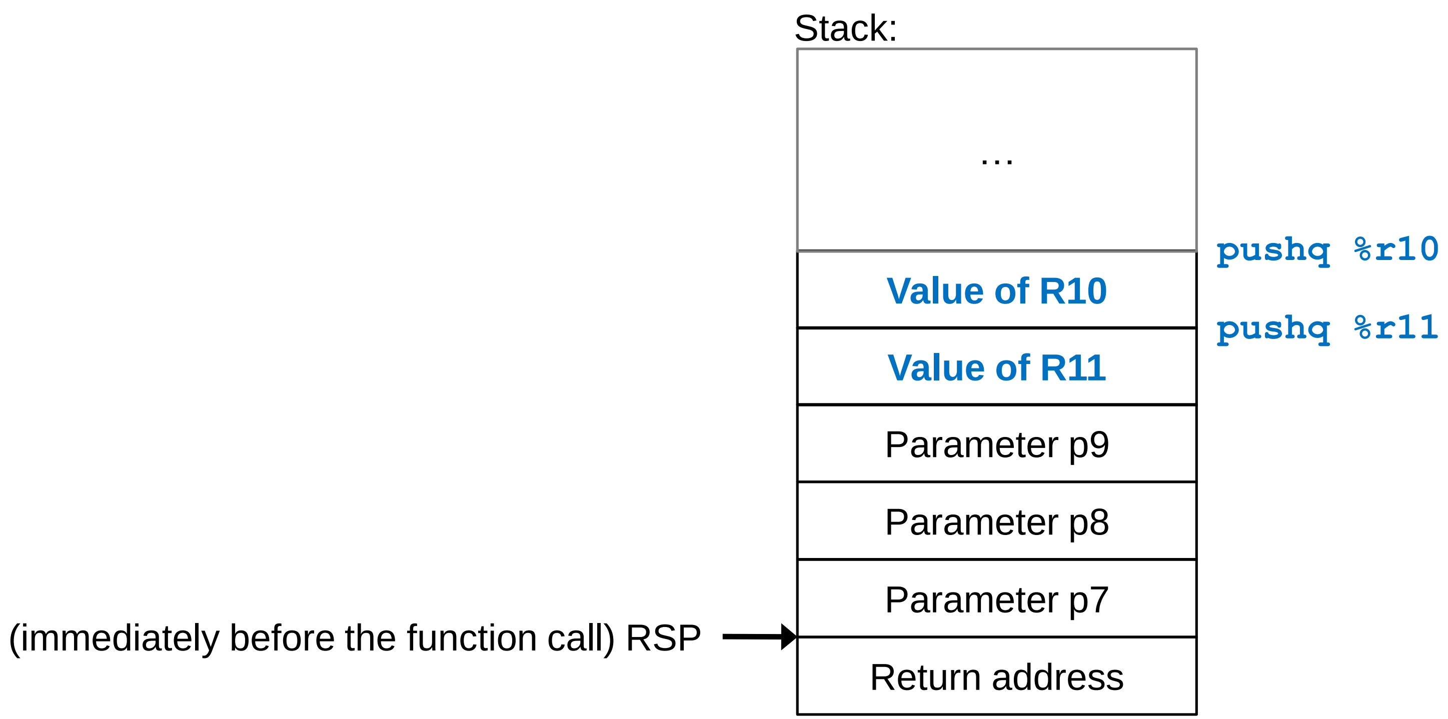
➔ Must be preserved by the function itself (see later)

Calling a function – Example

- Internally, the function will also need:
  - Registers: RBX, R10, R11, R12
  - Two 8-byte values as local variables

Caller-saved

➔ Caller must store these values on **stack:**

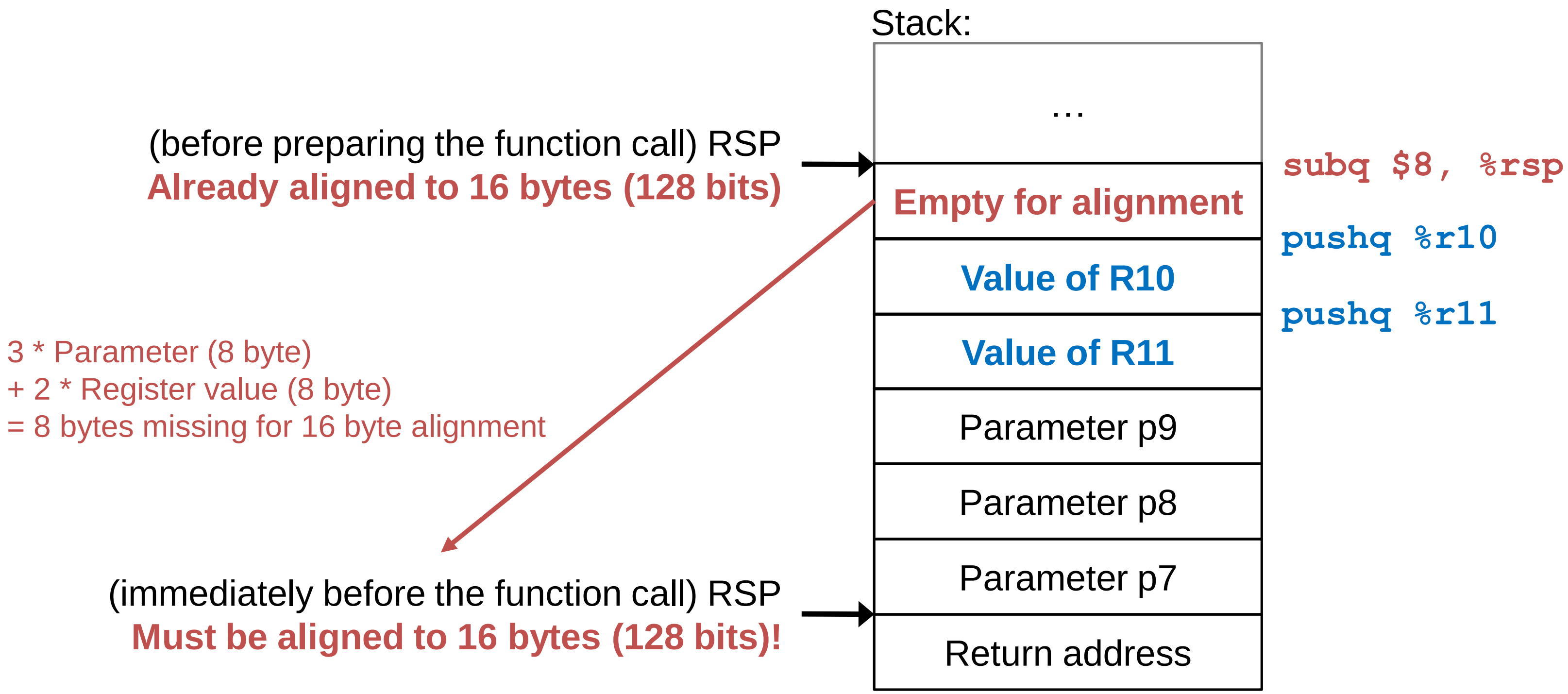


Calling a function – Example

- Internally, the function will also need:
  - Registers: RBX, R10, R11, R12
  - Two 8-byte values as local variables

Caller-saved

➔ Caller must store these values on **stack:**



Calling a function – Example

- Internally, the function will also need:
  - Registers: RBX, R10, R11, R12
  - Two 8-byte values as local variables

➔ The function is responsible for this (see later)

Calling a function – Example (caller)

```
alignment          { subq $8,%rsp          # Ensure stack alignment (we push 24 bytes; if
                   {                        # aligned before we need to "add" 8 bytes more)
caller-saved       { pushq %r10            # Save caller-safe registers
registers          { pushq %r11
parameters         { movq $1,%rdi          # Store first parameter in register
in registers       { movq $2,%rsi          # Note: No parameters names appear in assembler!
                   { movq $3,%rdx
                   { movq $4,%rcx
                   { movq $5,%r8
                   { movq $6,%r9           # Store sixth parameter in register
parameters         { pushq $9              # Further parameters are pushed on stack
on stack           { pushq $8              # in reverse order!
                   { pushq $7
                     call doSomething
cleanup            { addq $24,%rsp         # Clean up parameters from stack (equal to 3*popq)
                   { popq %r11             # Restore caller-safe registers
                   { popq %r10
                   { addq $8,%rsp          # Clean up alignment space
                     cmpq $0,%rax          # Now check the return value
                     je ...                # If zero, jump over the next block
```

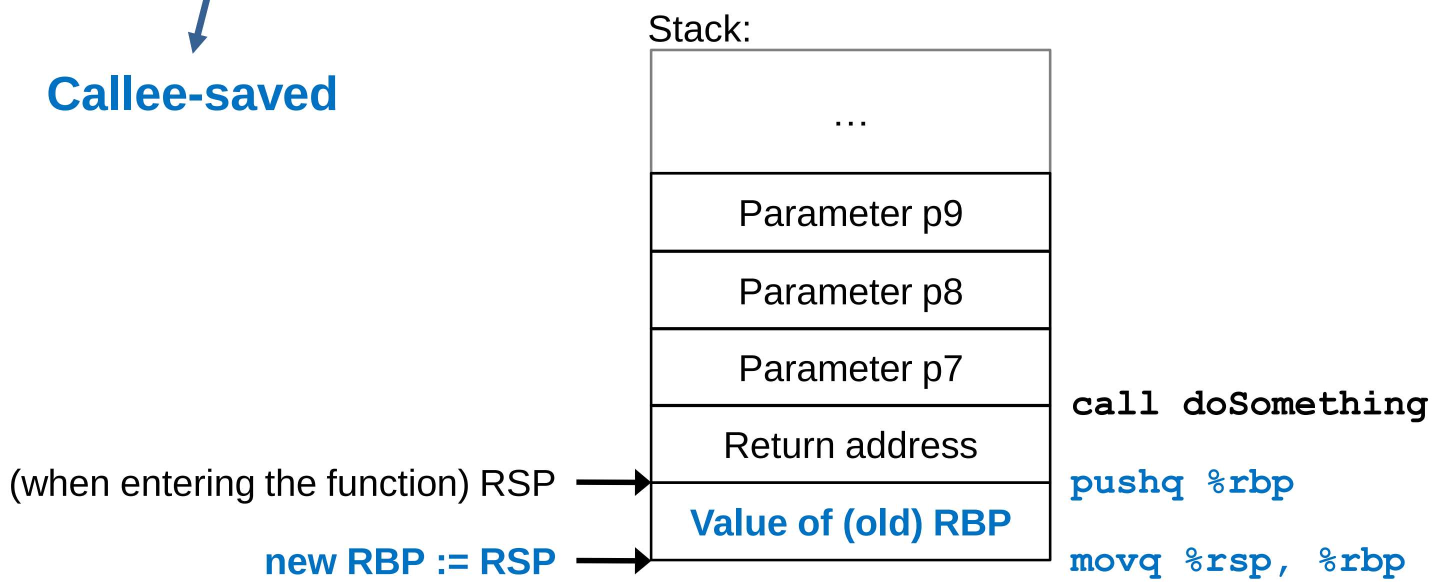
- Here we have to perform the alignment at the beginning
  - Or we would not know where exactly parameter 7 is on the stack!

Calling a function – Example

- The function needs to access its parameters and variables on the stack
  - Stack pointer changes when pushing to/popping from the stack
    → Cannot be used (or only with lots of difficulties → Compilers do this)
  - Base pointer RBP is used to store that stack position

Calling a function – Example

- The function needs to access its parameters and variables on the stack
  - Stack pointer changes when pushing to/popping from the stack
    ➔ Cannot be used (or only with lots of difficulties → Compilers do this)
  - Base pointer RBP is used to store that stack position



Calling a function – Example

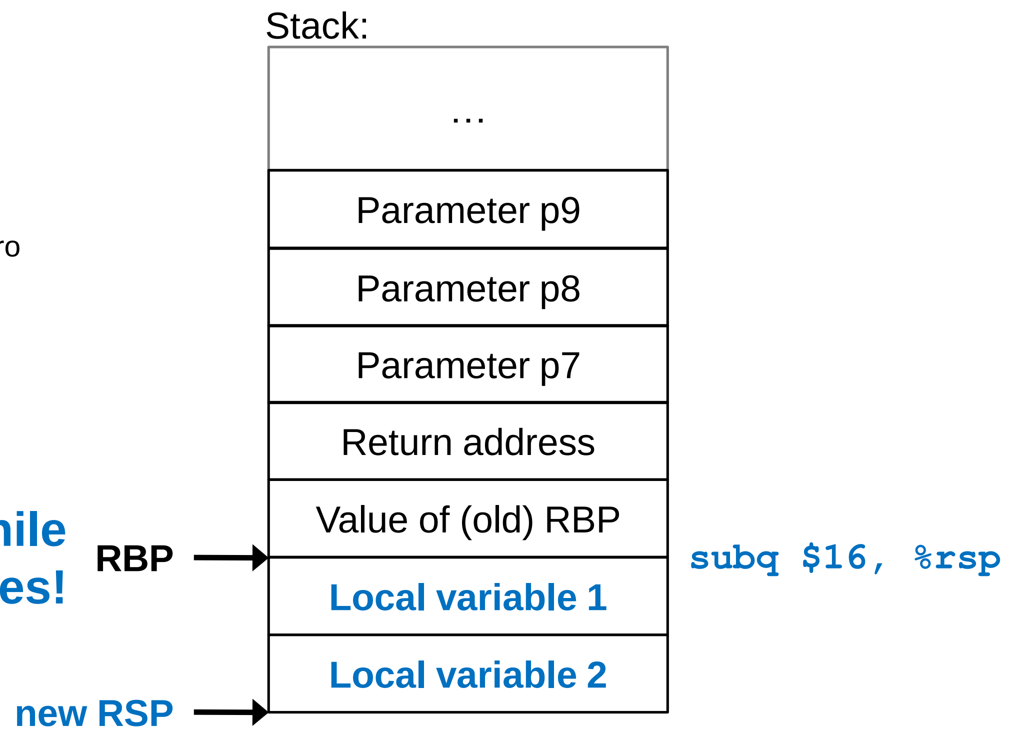
- Internally, the function will also need:
  - Registers: RBX, R10, R11, R12
  - Two 8-byte values as local variables

Note:
`subq $16, %rsp`
creates uninitialized space on the stack
We could use

```
  pushq $0
  pushq $0
```

instead to create the same space initialized to zero
(or any other value we need).

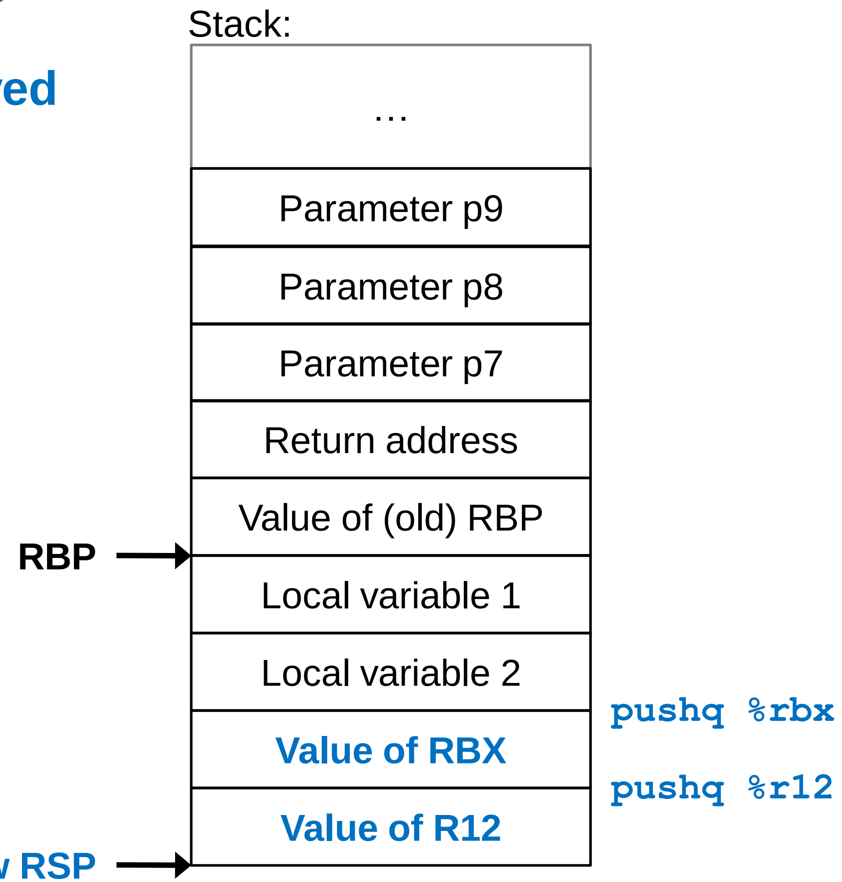
RBP stays the same while the function executes!



Calling a function – Example

- Internally, the function will also need:
  - Registers: RBX, R10, R11, R12
  - Two 8-byte values as local variables

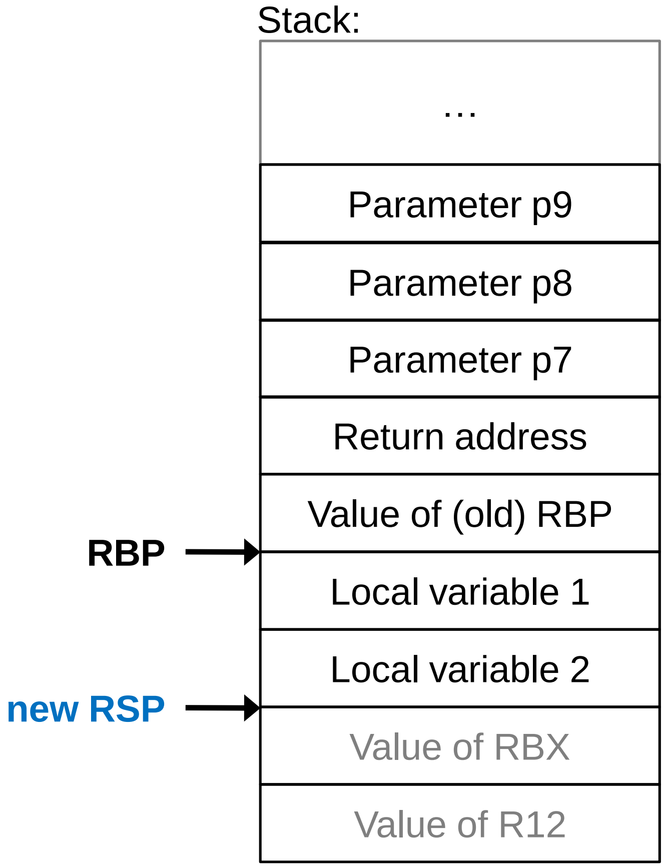
Callee-saved



Calling a function – Example

- At the end of the function, RAX is somehow set to the desired return value and the function has to clean up the stack
  - Restore saved Registers
    - In reverse order!

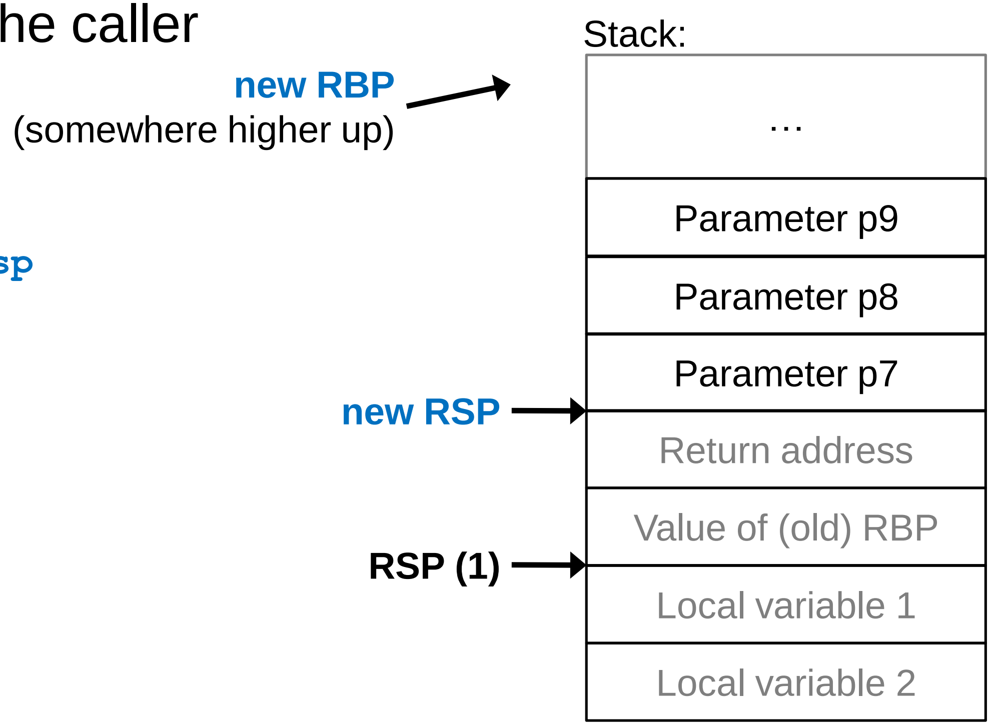
```
popq %r12
popq %rbx
```



Calling a function – Example

- Remove all local variables
  - Doesn't matter how many there are: RSP := RBP always removes all
- Restore the old RBP
- Return to the caller

(1)
```
movq %rbp,%rsp
popq %rbp
ret
```



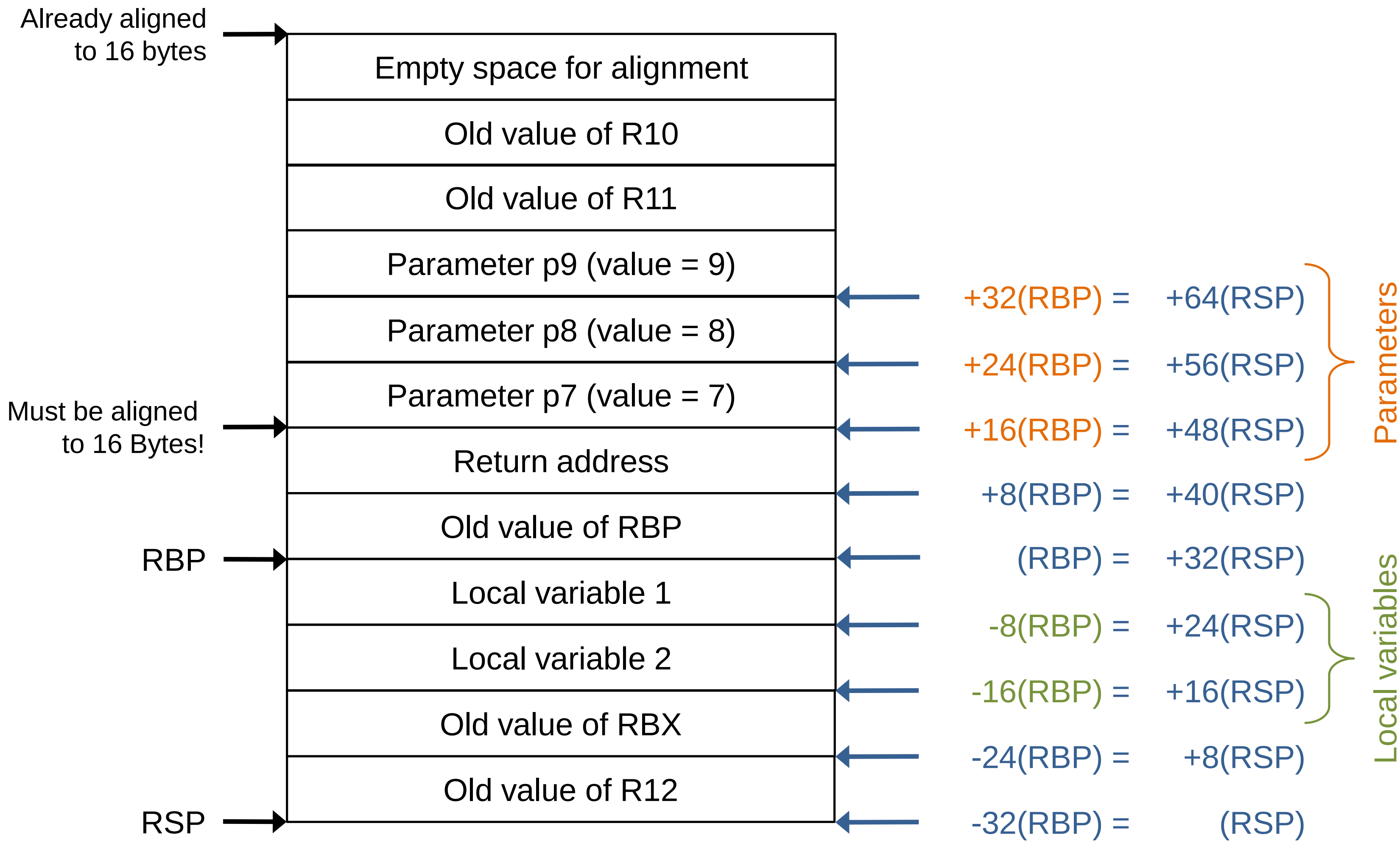
Calling a function – Example (callee)

```
doSomething:
        pushq %rbp          # Store old base pointer
        movq %rsp,%rbp      # Create new base pointer
        subq $16,%rsp       # Reserve space for 2 local variables
        pushq %rbx          # Save old value on stack
        pushq %r12          # R10 and R11 are caller-save!
        ...
        movq 16(%rbp),%r12          # Access parameter 7
        movq %r12,-16(%rbp)         # Store it in local variable 2
        ...
        movq %rdi,%rax              # Set return value
        ...
        addq $10,%r10       # Change the registers we "use"
        addq $10,%r11
        addq $10,%r12
        addq $10,%rbx
        ...
        popq %r12           # Restore old register values
        popq %rbx
        movq %rbp,%rsp      # Destroy local variables
        popq %rbp           # Restore old base pointer
        ret                 # Return to calling function
```

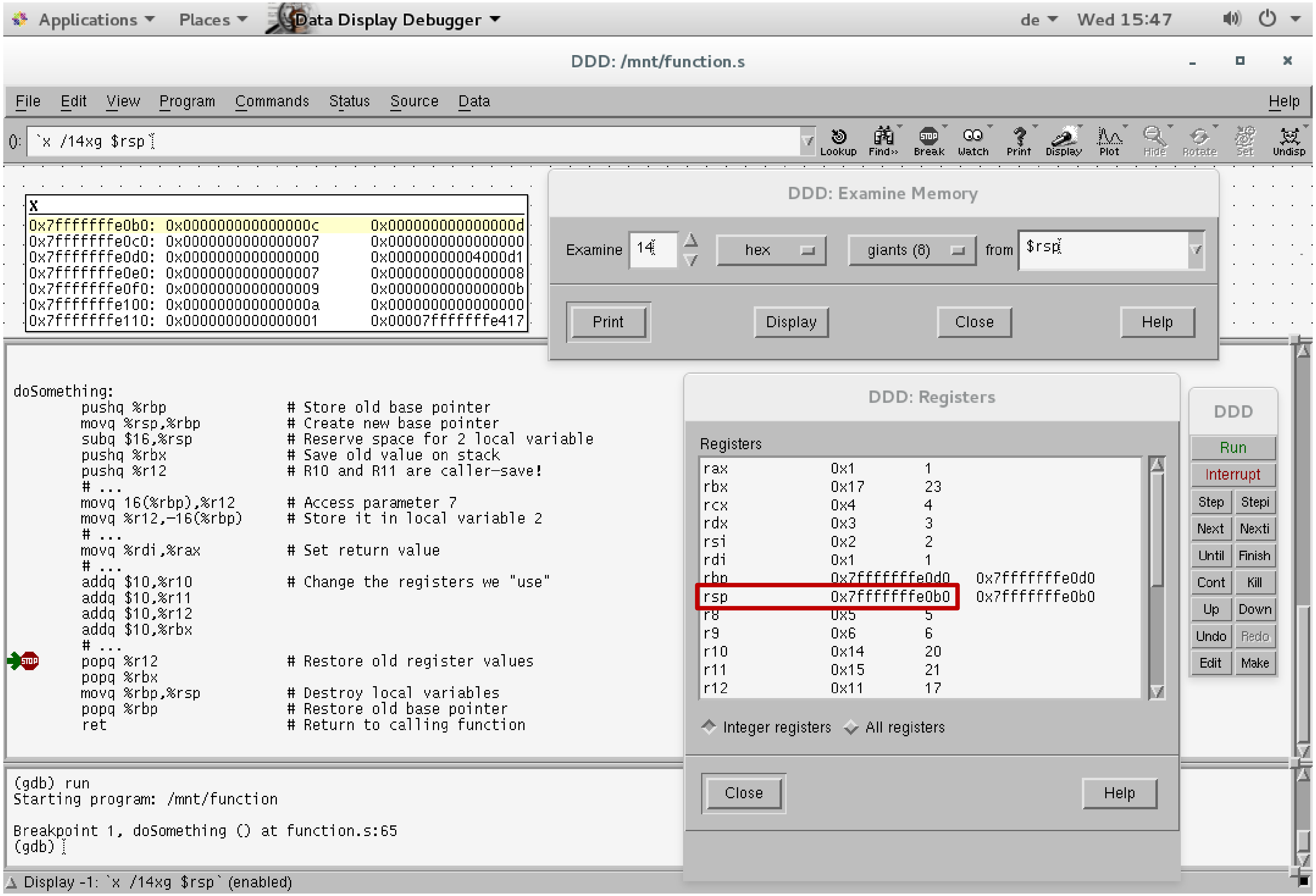
Prologue (first five instructions: pushq %rbp … pushq %r12)

Epilogue (last five instructions: popq %r12 … ret)

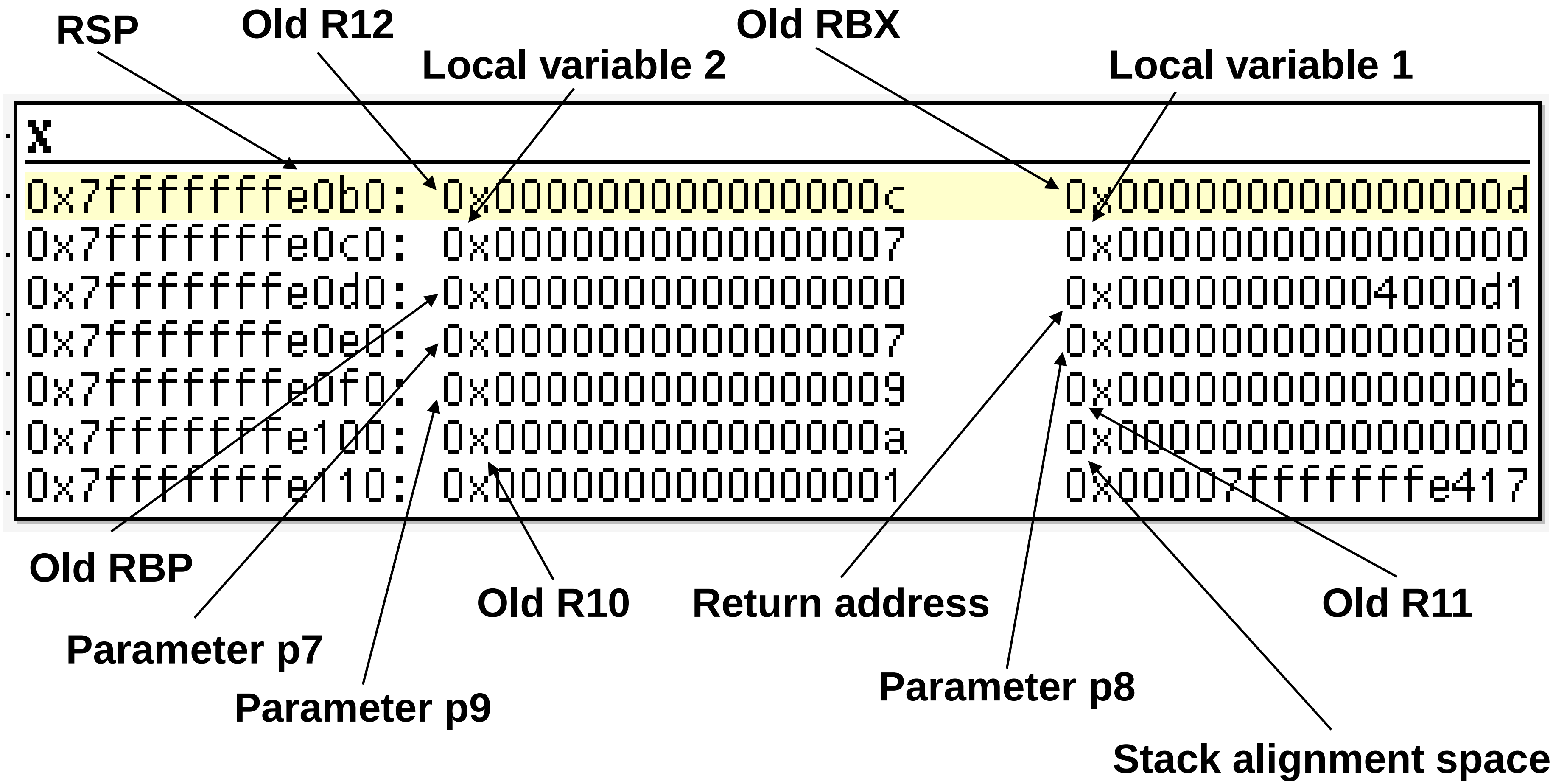
Complete stack of example program



Stack of example program in ddd



Stack of example program analyzed



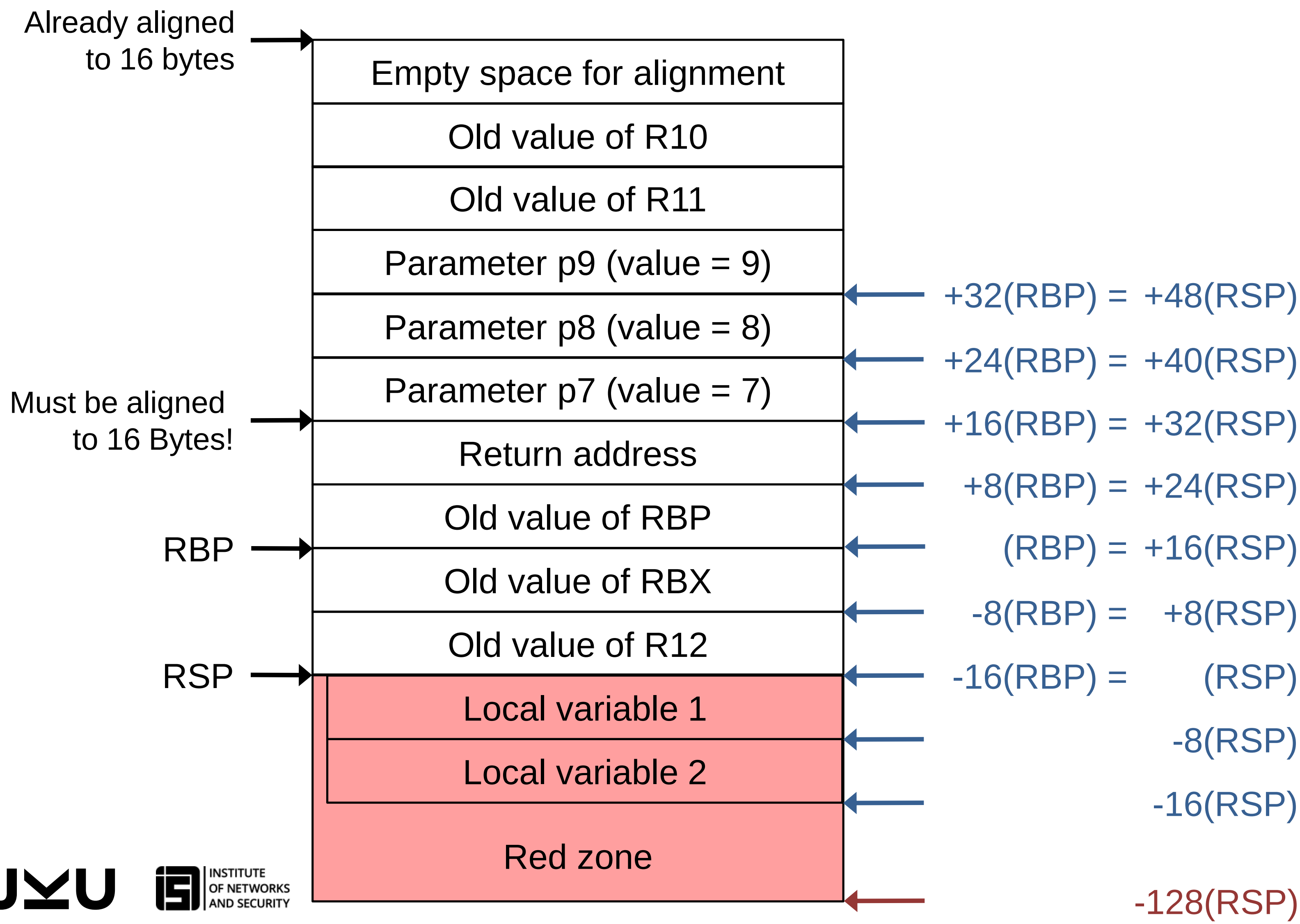
*

Calling a function – Example (callee)

```
doSomething:
        pushq %rbp          # Store old base pointer
        movq %rsp,%rbp      # Create new base pointer
                            # No need for RSP adjust., as less than 128 bytes
        pushq %rbx          # Save old value on stack
        pushq %r12          # R10 and R11 are caller-save!
        ...
        movq 16(%rbp),%r12          # Access parameter 7
        movq %r12,-16(%rsp)         # Store it in local variable 2
        ...
        movq %rdi,%rax              # Set return value
        ...
        popq %r12           # Restore old register values
        popq %rbx
        movq %rbp,%rsp      # Reset stack pointer always, even if unnecessary!
        popq %rbp           # Restore old base pointer
        ret                 # Return to calling function
```

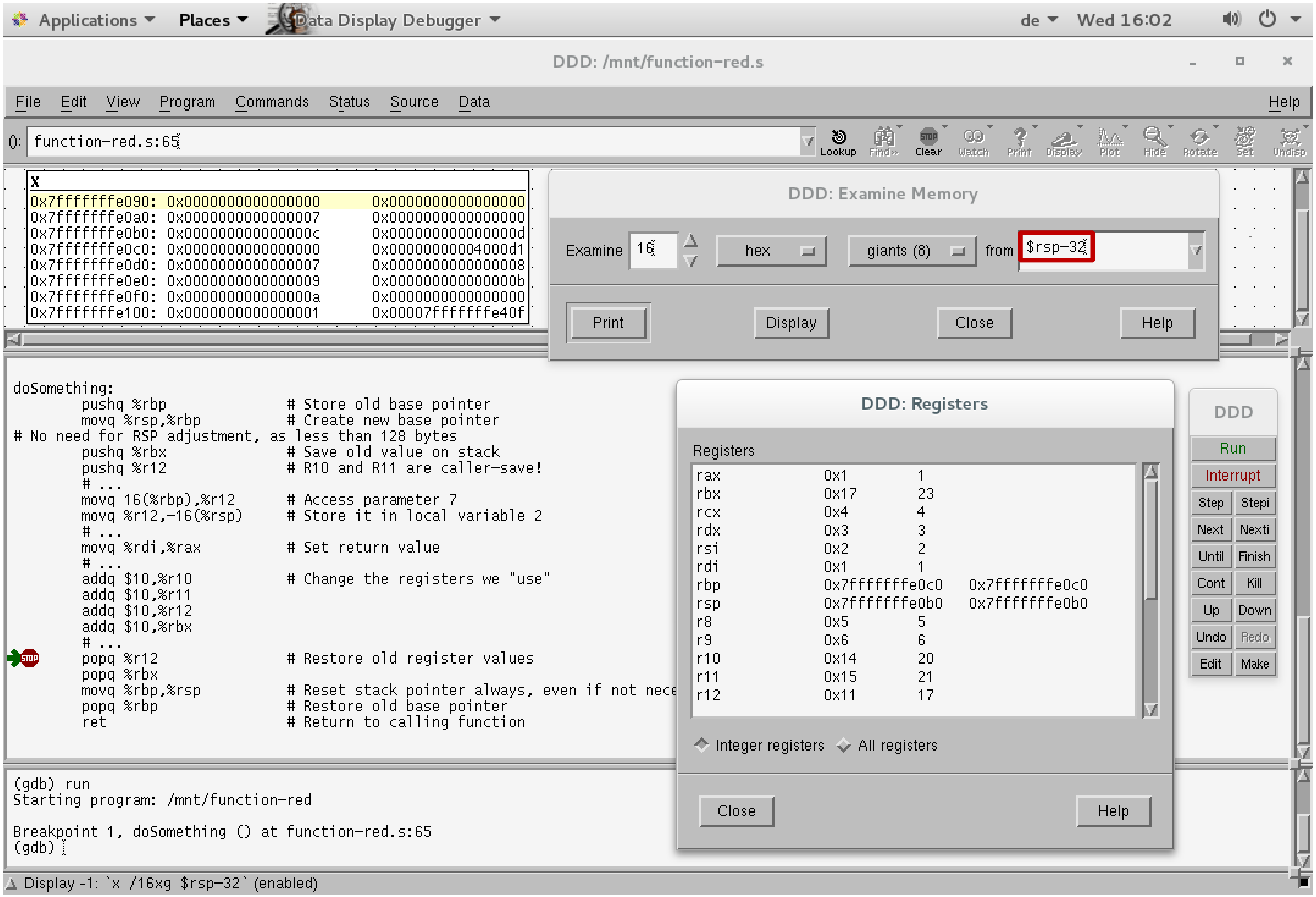
- Variant: the function does not use explicit local variables, but uses the red zone (max. 128 bytes below RSP) instead
  - Still resets the base pointer

Stack of example program (variant)



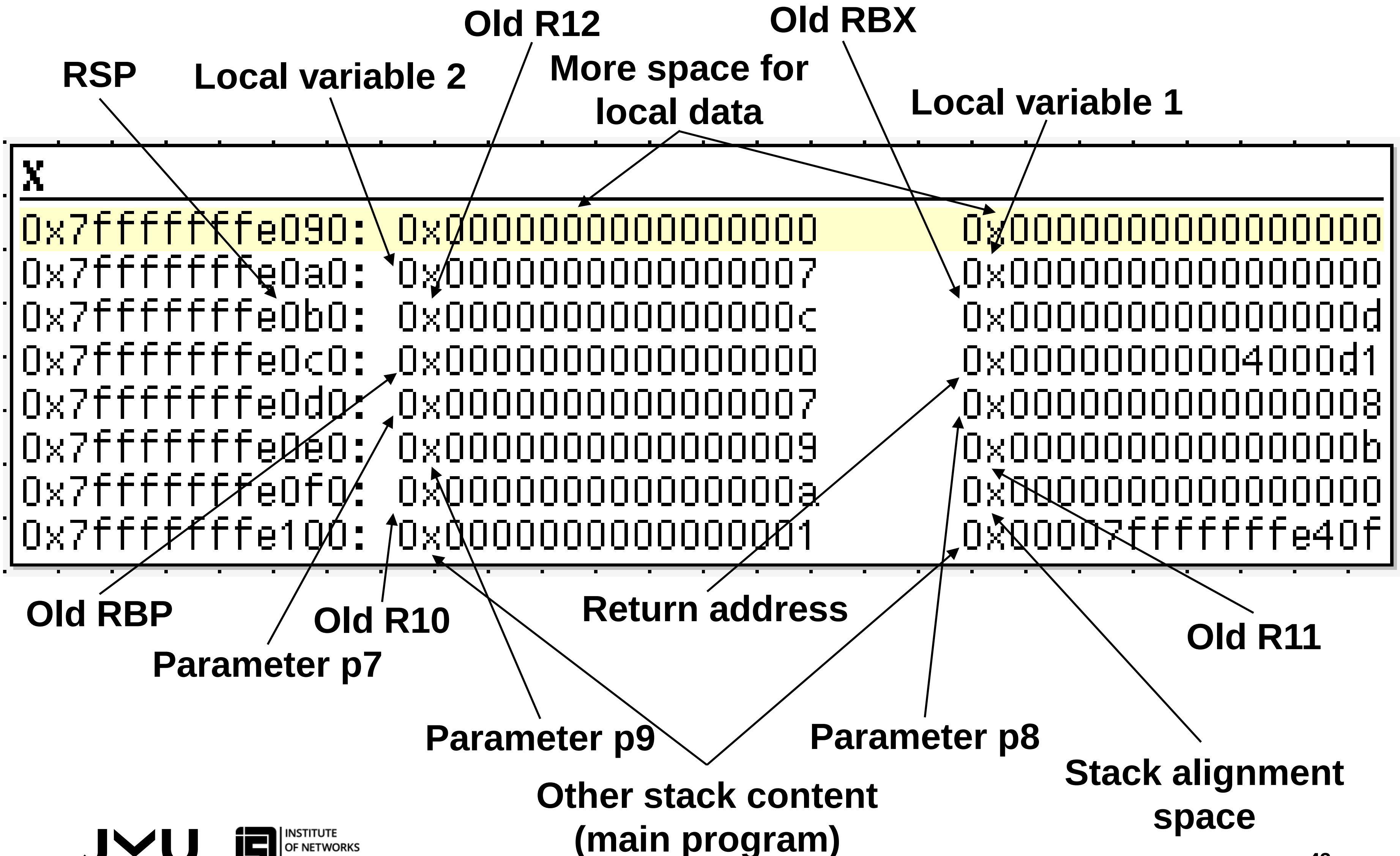
*

Stack of red zone variant in ddd



*

Stack of red zone variant analyzed



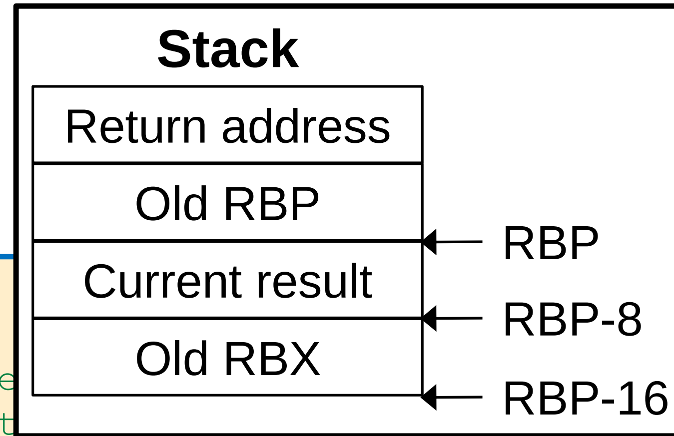
power1.s

```
    # PURPOSE:  Program to illustrate how functions work
    #           This program will compute the value of 2^3 + 5^2
...
_start:
...
    movq $2,%rdi              # Store first argument
    movq $3,%rsi              # Store second argument
    call  power               # Call the function
    movq  %rax,%r12           # Save first result into temporary register

    movq $5,%rdi              # Store first argument
    movq $2,%rsi              # Store second argument
    call power                # Call the function
    movq %rax,%rdi            # Save second result into temporary register
    addq %r12,%rdi            # The second result is in %r12
                              # Add the first one and store in %rdi

    movq $60,%rax             # Exit (%rdi is returned)
    syscall
```

power1.s




```
    .type power, @function
power:
    pushq %rbp              # Save old base pointe
    movq  %rsp,%rbp         # Make stack pointer t
    subq  $8,%rsp           # Get room for our local storage
    pushq %rbx              # Preserve callee-safe register
    movq  %rdi,%rbx         # Put first argument in %rbx
    movq  %rsi,%rcx         # Put second argument in %rcx
    movq  %rbx,-8(%rbp)     # Store current result
power_loop_start:
    cmpq  $1,%rcx           # If the power is 1, we are done
    je    end_power
    movq  -8(%rbp),%rax     # Move the current result into %rax
    imulq %rbx,%rax         # Multiply the current result by the base number
    movq  %rax,-8(%rbp)     # Store the current result
    decq  %rcx              # Decrease the power
    jmp   power_loop_start  # Run for the next power
end_power:
    movq -8(%rbp),%rax      # Return value goes in %rax
    popq %rbx               # Restore callee-safe registers
    movq %rbp,%rsp          # Restore the stack pointer
    popq %rbp               # Restore the base pointer
    ret                     # Return to caller
```

Notes on power1.s

- **`.type power,@function`**
  - Tells the linker that **`power`** should be treated as a function
- Difference between **`jmp`** and **`call`**
  - **`jmp`** modifies the RIP register to point to the new code location
  - **`call`** additionally pushes the return address on the stack
- The algorithm uses a local variable to **temporarily store** the result
  - Also a register would be possible (if available, e.g. R12)
  - But a register is not possible if the function calls another function and wants to pass a pointer to this variable as a parameter, as there is **no pointer to a register**
    - Registers do not have memory addresses!
- This program does not work if the parameter power is zero
  - See improved version power2.s in later slides

*

power2.s

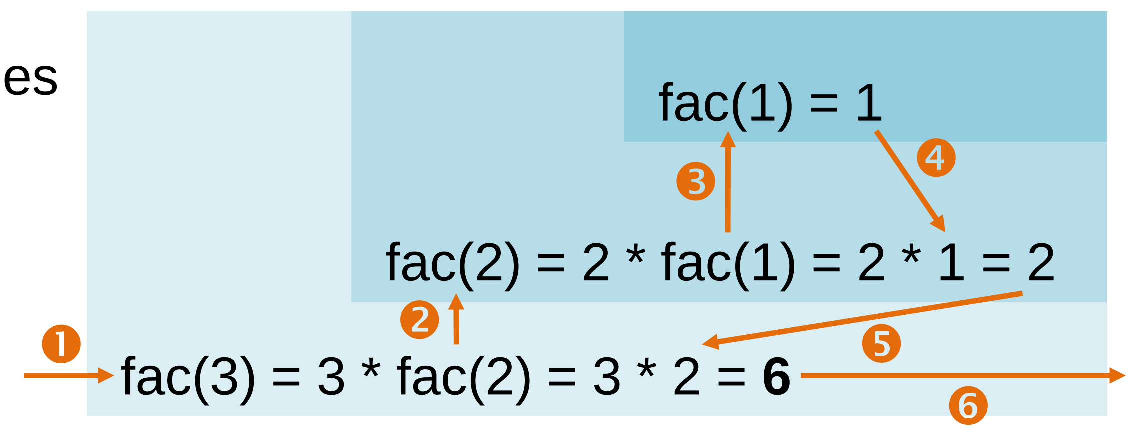
Stack
Return address

```
power:
   movq  $1,%rax
   cmpq  $0,%rsi            # If the power is 0, we return 1
   je    end_power
   movq  %rdi,%rax          # Prepare local variable for first round
power_loop_start:
   cmpq  $1,%rsi            # If the power is 1, we are done
   je    end_power
   imulq %rdi,%rax          # Multiply the current result by the base number
   decq  %rsi               # Decrease the power
   jmp   power_loop_start   # Run for the next power
end_power:
   ret                      # Return to caller
```

- Optimized version: As this is a “leaf function” (it does not call any other functions itself), we can skip everything about the stack
  - No prologue, no epilogue → Sole stack content is return address
- We do not use any callee-safe registers, so we don’t have to save anything on the stack either
- Additionally check for “exponent 0” and return 1

Factorial – Recursion example

- **Factorial of a number n**
  - Product of all numbers between 1 and number n
  - Factorial of 7 = 1 * 2 * 3 * 4 * 5 * 6 * 7
- Observation
  - Factorial of 7 = factorial of 6 * 7
  - Generalized: **fac(n) = fac(n – 1) * n**
  - Base case: **fac(1) = 1**
- Recursive definition
- Implementation as a **recursive function**
  - Function calls itself
  - Returns when it reaches the base case



factorial.s

```
        .section .text
        .globl _start
        .globl factorial # this is not needed unless we want to share
                         # this function among other programs
_start:
        movq  $4,%rdi    # The factorial takes one argument - the
                         # number we want a factorial of (4 -> 24).
        call  factorial  # run the factorial function
        movq  %rax,%rdi  # factorial returns the answer in %rax, but
                         # we want it in %rdi to send it as our exit status
        movq  $60,%rax   # call the kernel's exit function
        syscall

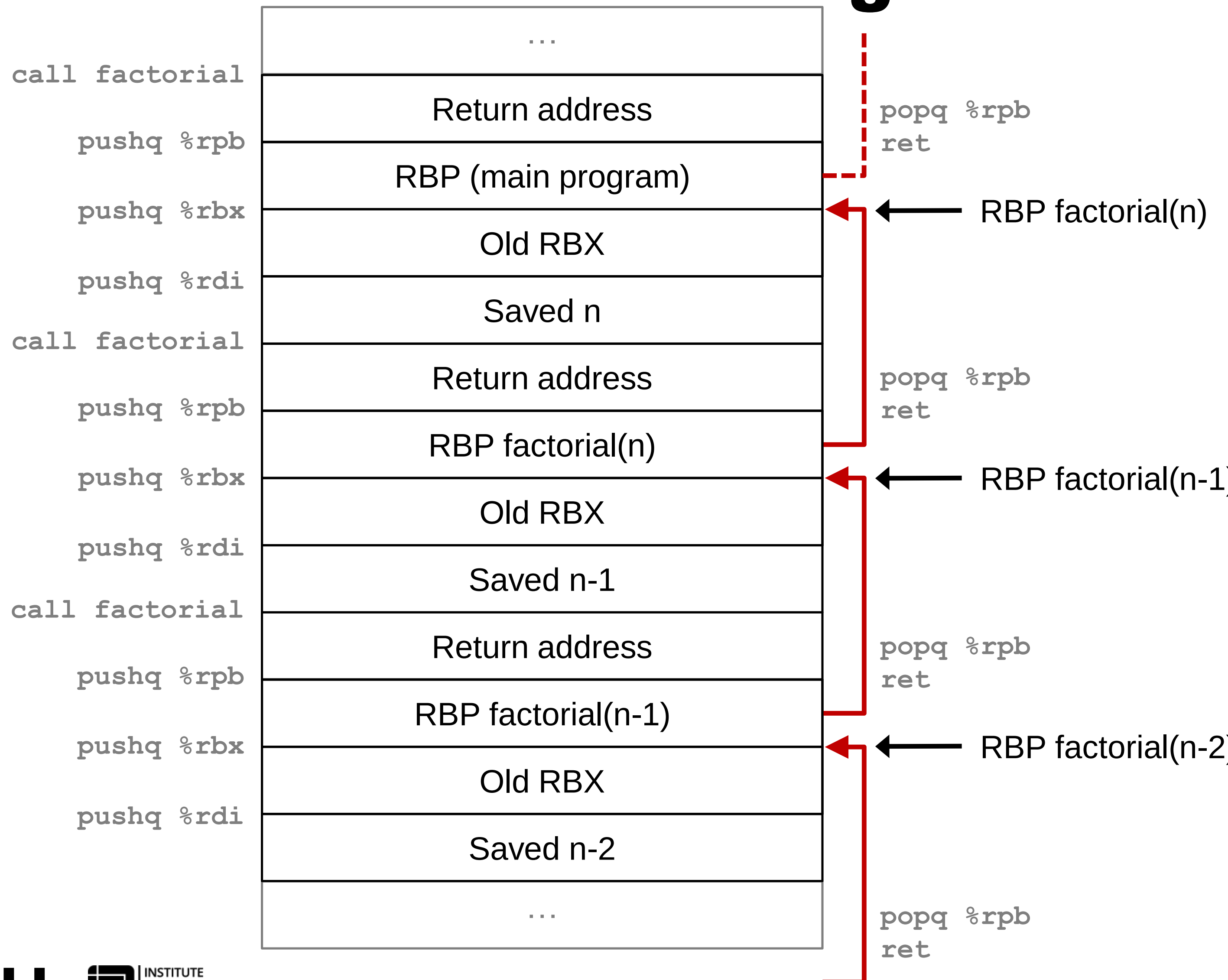
        .type factorial,@function
factorial:
        pushq %rbp       # standard function stuff - we have to
                         # restore %rbp to its prior state before
                         # returning, so we have to push it
        movq  %rsp,%rbp  # This is because we don't want to modify
                         # the stack pointer, so we use %rbp.
        pushq %rbx       # Save RBX (used for multiplication)
                         # Note: We could easily use e.g. R11 to
                         # avoid needing the stack!
```

factorial.s

```
check_base_case0:
        movq  $1,%rax
        cmpq  $0,%rdi     # If the number is 0, we return 1
        je end_factorial
check_base_case1:
        cmpq  $1,%rdi     # If the number is 1, that is our base
        je end_factorial  # case, and we simply return (1 is
                          # already in %rax as the return value)

        pushq %rdi        # save our own parameter for later
        decq  %rdi        # decrease the value
        call  factorial   # call factorial
        popq  %rbx        # retrieve our own parameter
        imulq %rbx,%rax   # multiply it by the result of the last
                          # call to factorial (in %rax); the answer
                          # is stored in %rax, which is good since
                          # that's where return values go.
end_factorial:
        popq  %rbx        # restore old value
        movq  %rbp,%rsp   # standard function return stuff - we
        popq  %rbp        # have to restore %rbp and %rsp to where
                          # they were before the function started
        ret               # return from the function
```

Stack of factorial.s during recursion



Note: 3 * pushq + return address → stack remains aligned to 16 bytes!

*

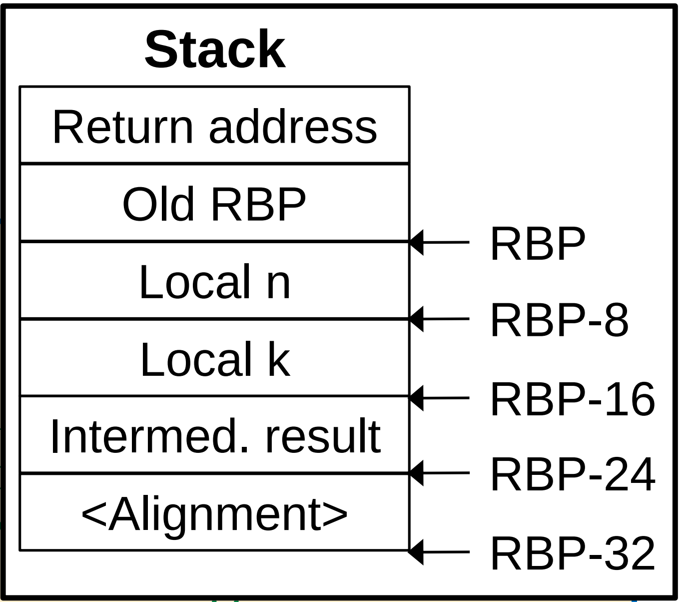
binom.s

- Binomial coefficient “n over k”
  - **(n over 0) = (n over n) = 1** (base case)
  - **(n over k) = (n - 1 over k - 1) + (n - 1 over k)** (recursive case)
- Function binom(n, k)
  - **binom(n, 0) = binom(n, n) = 1**
  - **binom(n, k) = binom(n-1, k-1) + binom(n-1, k)**
- Differences to factorial.s
  - 2 base cases
  - 2 recursive calls in general case
  - Need to save intermediate result of first call
- Note: Code does not check the parameters for validity
- Note: Return address + RBP → Stack is again correctly aligned

binom.s

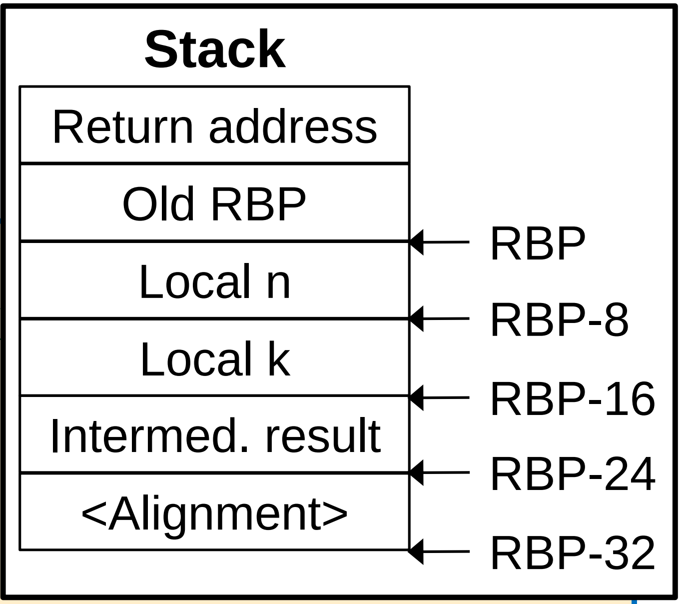
```
        .type binom,@function
                                # RDI = n, RSI = k
binom:
        pushq %rbp              # standard function
                                # restore %rbp to i
                                # returning, so we
        movq  %rsp,%rbp         # This is because w
                                # the stack pointer
                                # 8(%rbp) holds the return address
        subq $32,%rsp           # get room for local n, local k and
                                # result of first recursive call
                                # Additional 8 Bytes for stack alignment
                                # in recursive calls
check_base_case1:
        cmpq $0,%rsi            # If k is 0, we return 1
        jne check_base_case2
        movq $1,%rax
        jmp end_binom

check_base_case2:
        cmpq  %rdi,%rsi         # If n = k, we return 1
        jne general_case
        movq $1,%rax
        jmp end_binom
```



*

binom.s




```
general_case:
        # Note: Parameters are passed in registe
        # so we do not have a "backup copy" on o
        movq  %rdi, -8(%rbp) # save n
        movq  %rsi,-16(%rbp) # save k
        decq  %rdi           # decrease n
        decq  %rsi           # decrease k
        # first recursive call: (n - 1 over k -
        call  binom          # recursive call
        movq %rax,-24(%rbp)  # save value of first recursive call
        movq  -8(%rbp),%rdi  # restore n
        movq -16(%rbp),%rsi  # restore k
        decq %rdi            # decrease n
        # second recursive call: (n - 1 over k)
        call  binom          # recursive call
        # %rax holds result of second recursive call
        addq -24(%rbp),%rax  # compute sum of recursive calls = result
        # %rax holds result

end_binom:
        movq  %rbp,%rsp      # standard function return stuff - we
        popq  %rbp           # have to restore %ebp and %esp to where
                             # they were before the function started
        ret                  # return from the function
```

JOHANNES KEPLER
UNIVERSITÄT LINZ

INSTITUTE
OF NETWORKS
AND SECURITY

https://www.ins.jku.at

THANK YOU FOR YOUR ATTENTION!

Slides by: Michael Sonntag
michael.sonntag@ins.jku.at
+43 (732) 2468 - 4137
S3 235 (Science park 3, 2nd floor)