AMD Opteron™
OSA146DAA5BN
CACJE 0607APBW
141885 4B61344

Introduction to x64

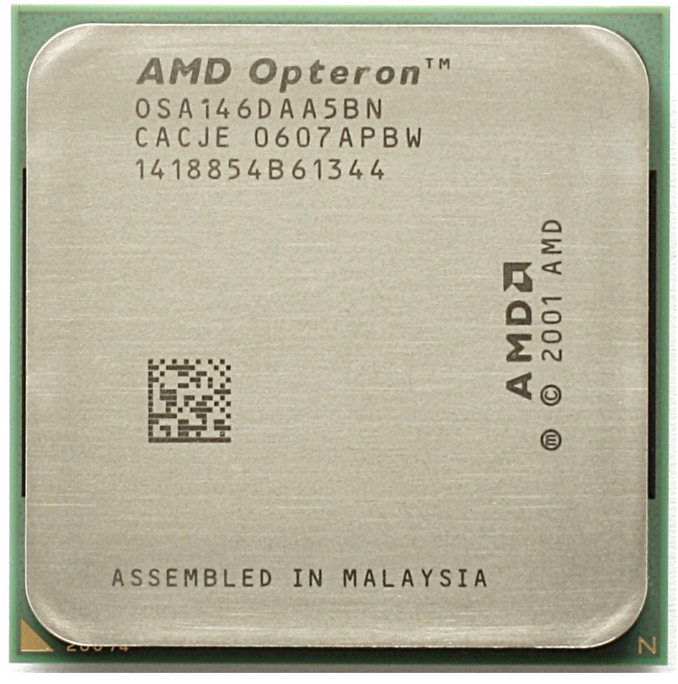
CSU34021 - Computer Architecture II

Some Background/History

- x86-64 is a backward-compatible extension of the x86 ISA.

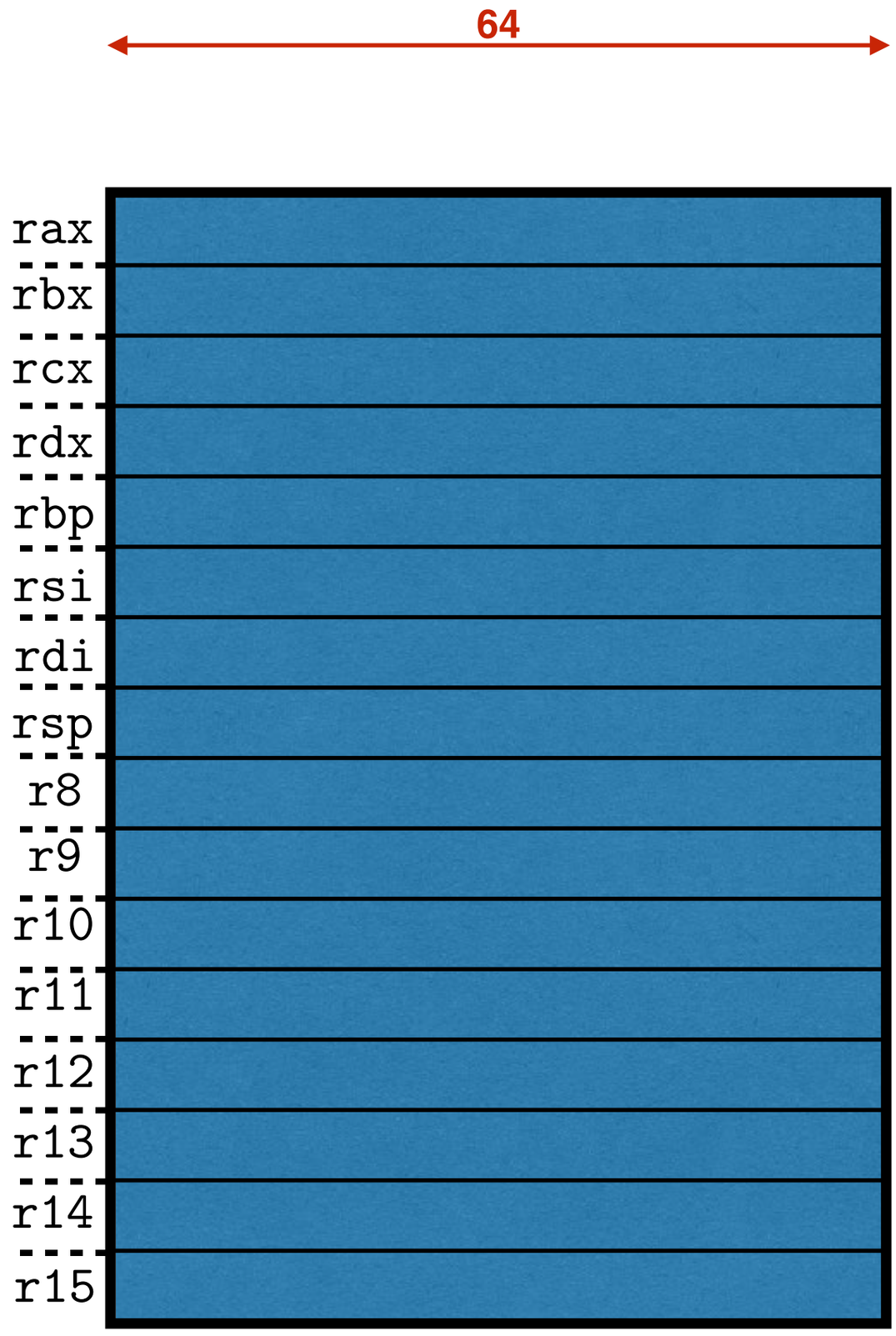
- Originally produced by AMD (Intel had developed a completely different ISA [called IA-64] for its servers and then had to backtrack and follow up AMD footsteps).

- It has additional registers, increased memory addressing space, and the same instruction set of IA-32.



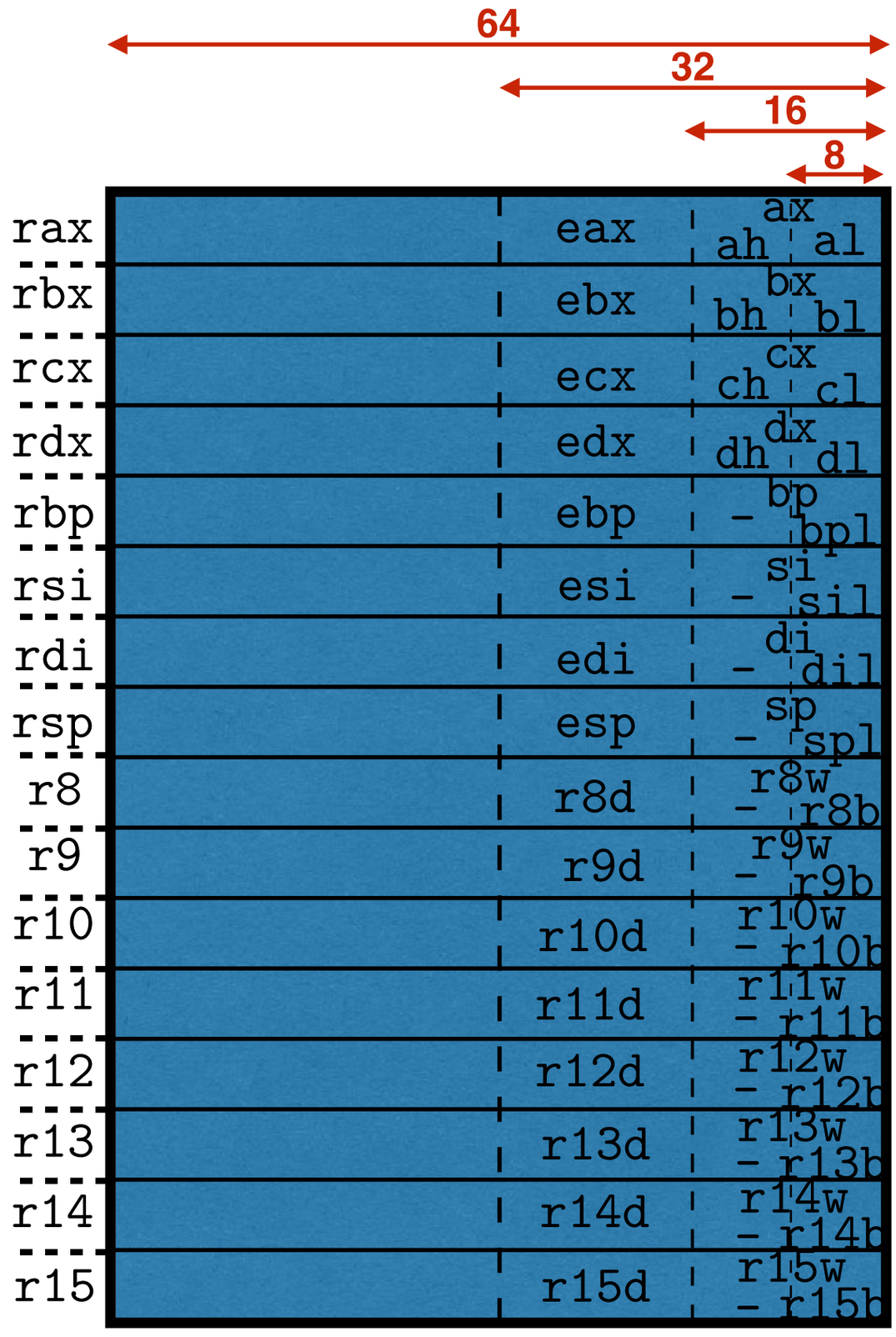
x64 Registers

- 8 more registers have been added.



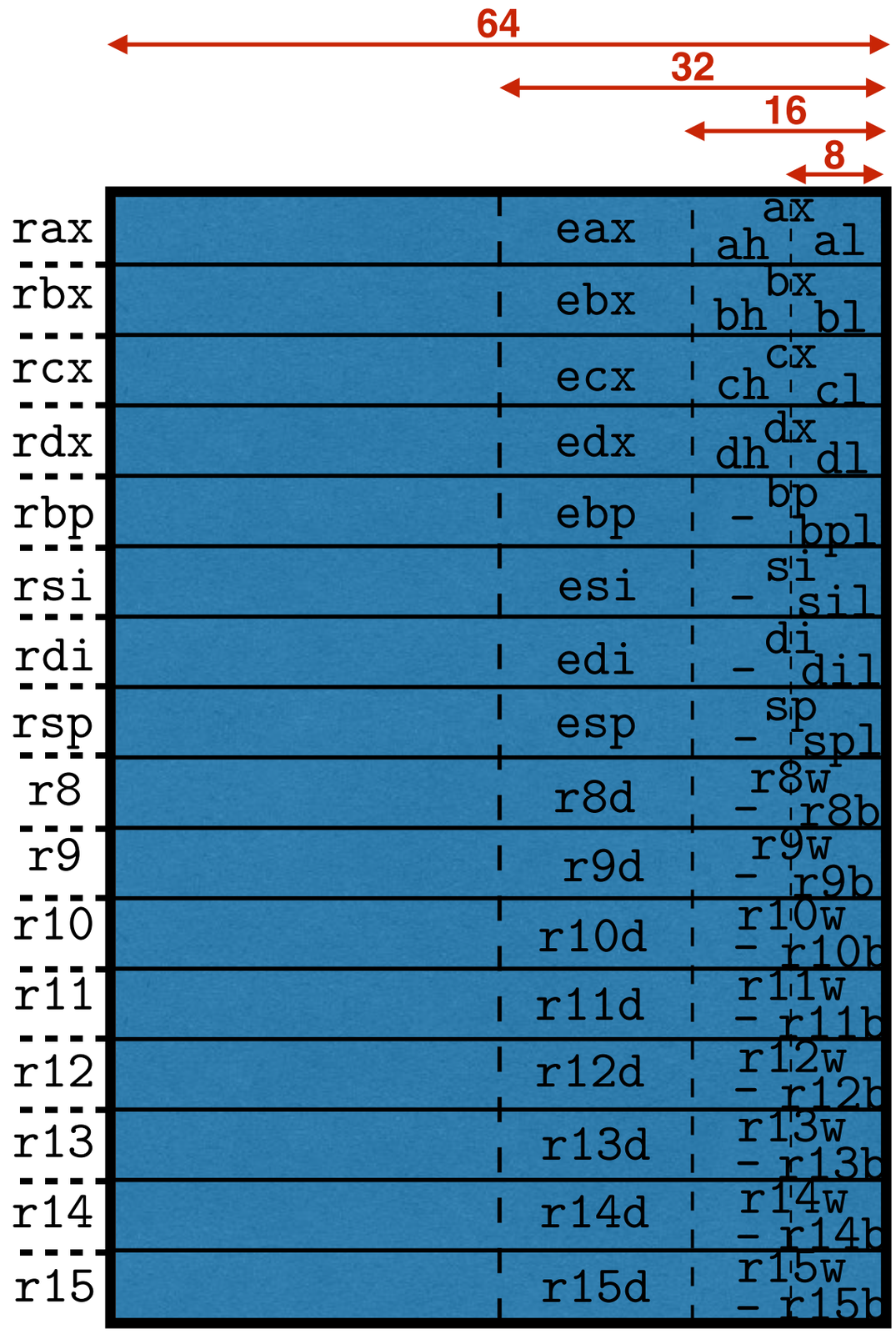
x64 Registers

- 8 more registers have been added.
- Lower bits are still accessible (not all the bits combination can be accessed at the same time…).
- All other registers extended as well.



x64 Registers

- 8 more registers have been added.
- Lower bits are still accessible (not all the bits combination can be accessed at the same time…).
- All other registers extended as well.
- Some implementations have additional registers for long integers and floating point operations (e.g. 16 XMM registers extending x87).



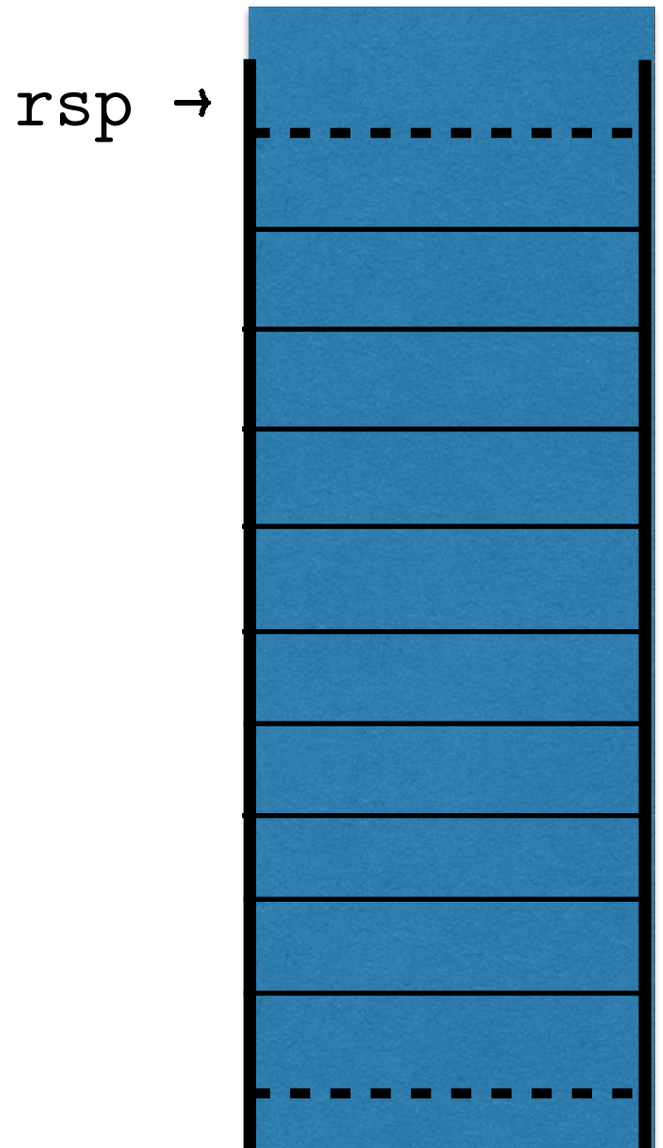
X64 Briefly…

- The instruction set is virtually the same of x86.
- Main difference in the calling convention for functions. Let's see the Microsoft one…
- The calling convention uses the fact that there are more registers to speed up the function calling.
- Plus, having more registers at ones disposal, often the frame pointer for the stack is not even used.

Microsoft Calling Convention

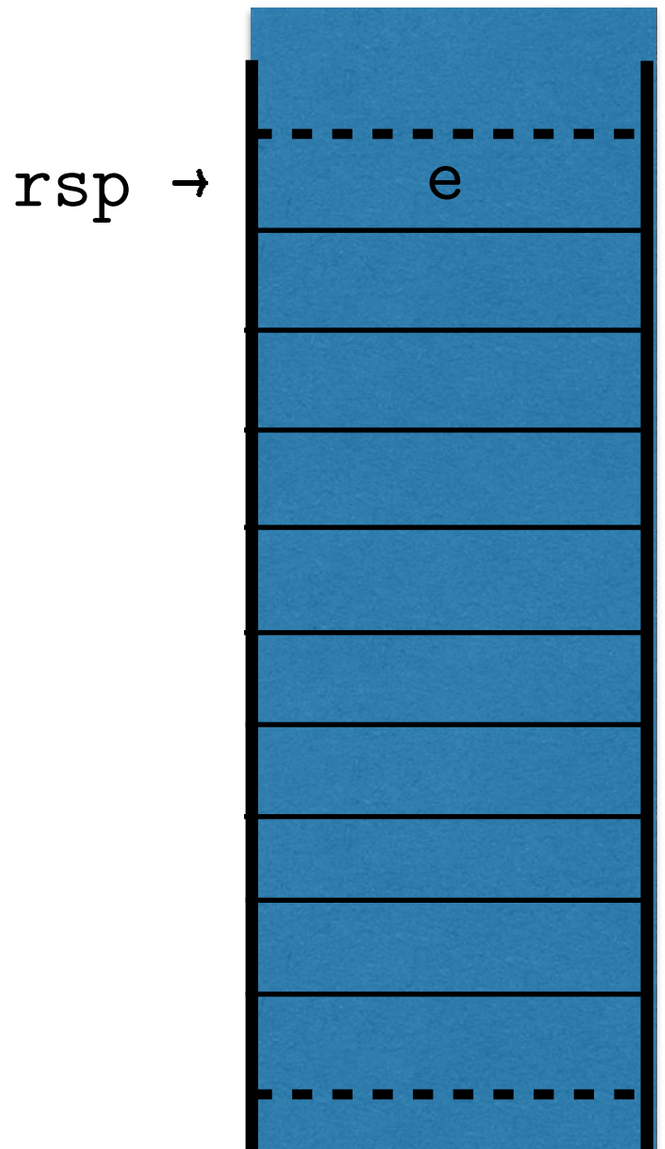
- `rcx, rdx, r8, r9` used for the first 4 integer or pointer arguments (**from left to right**).
- Additional parameters pushed on the stack (**right to left**).
- Caller to allocate 32-bytes of **shadow space** for storing the input registers. Caller also has to clean the stack.
- Integers returned to `rax`. For larger return values pointers are used.
- `rax, rcx, rdx, r8, r9, r10, r11` **considered volatile**.
- If other registers are used, the callee needs to preserve them.
- **The stack needs to be aligned on a 16-byte boundary!**

```
 _int64 my_fun(_int64 a, _int64 b, _int64 c, _int64 d, _int64 e)
{_int64 x; _int64 y; ... return x;}
```



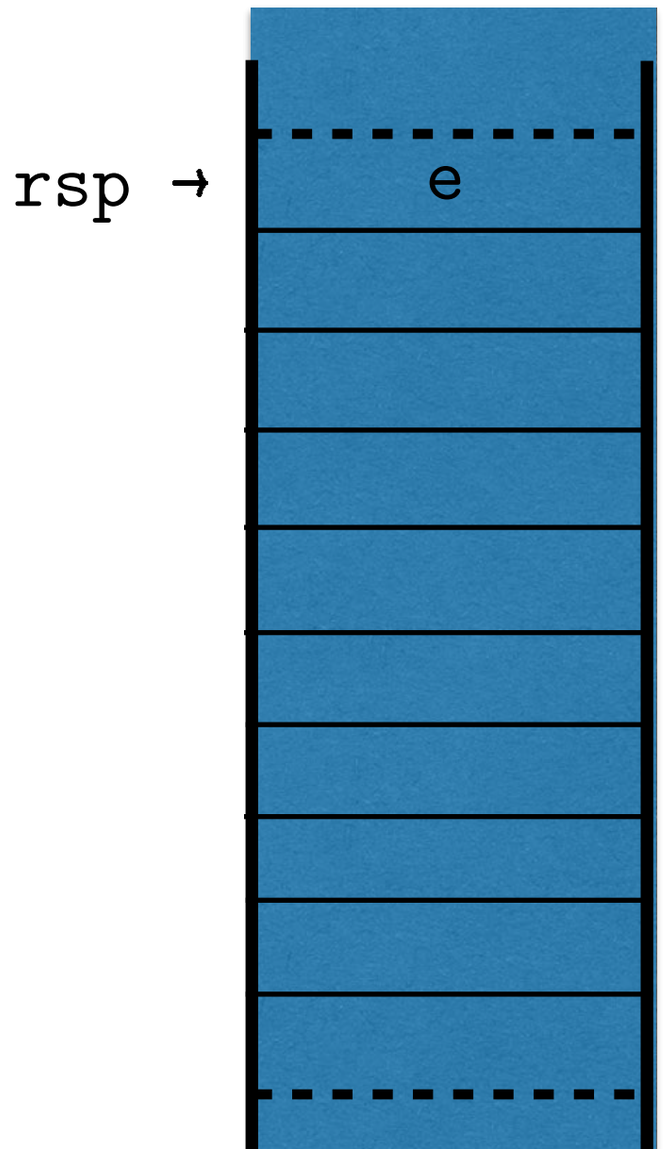
```
_int64 my_fun(_int64 a, _int64 b, _int64 c, _int64 d, _int64 e)
{_int64 x; _int64 y; ... return x;}
```

```
push e          ;5th arg in stack
```



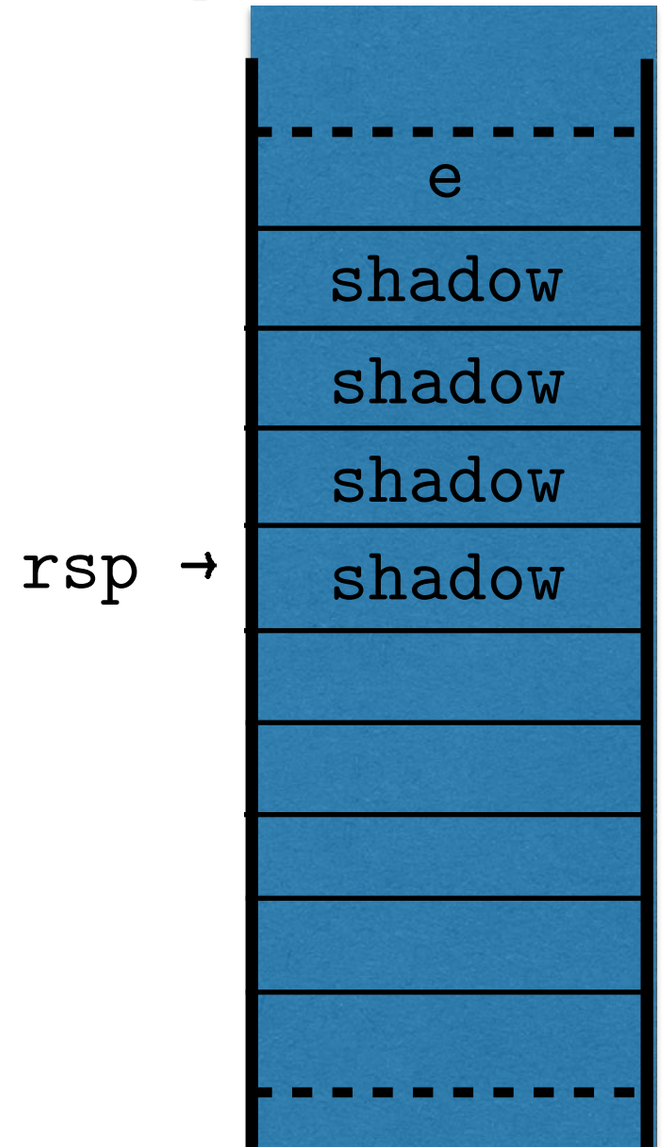
```
_int64 my_fun(_int64 a, _int64 b, _int64 c, _int64 d, _int64 e)
{_int64 x; _int64 y; ... return x;}
```

```
push e          ;5th arg in stack
mov  rcx, a     ;1st arg
mov  rdx, b     ;2nd arg
mov  r8 , c     ;3rd arg
mov  r9 , d     ;4th arg
```



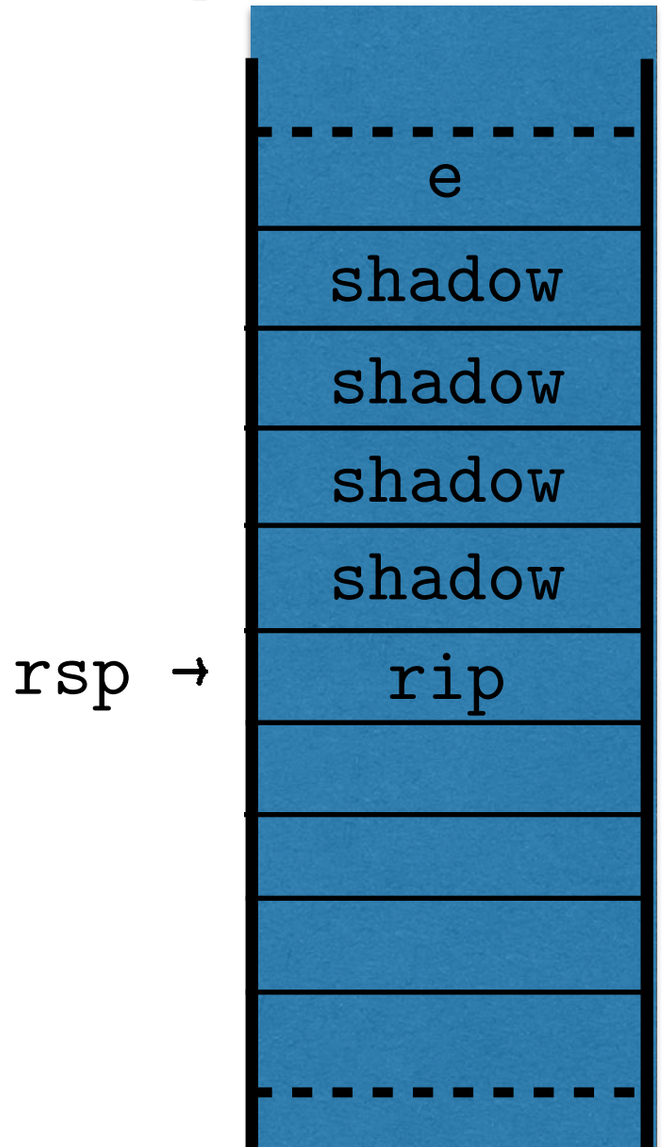
```
_int64 my_fun(_int64 a, _int64 b, _int64 c, _int64 d, _int64 e)
{_int64 x; _int64 y; ... return x;}
```

```
push e          ;5th arg in stack
mov  rcx, a     ;1st arg
mov  rdx, b     ;2nd arg
mov  r8 , c     ;3rd arg
mov  r9 , d     ;4th arg
sub  rsp, 32    ;shadow space
```



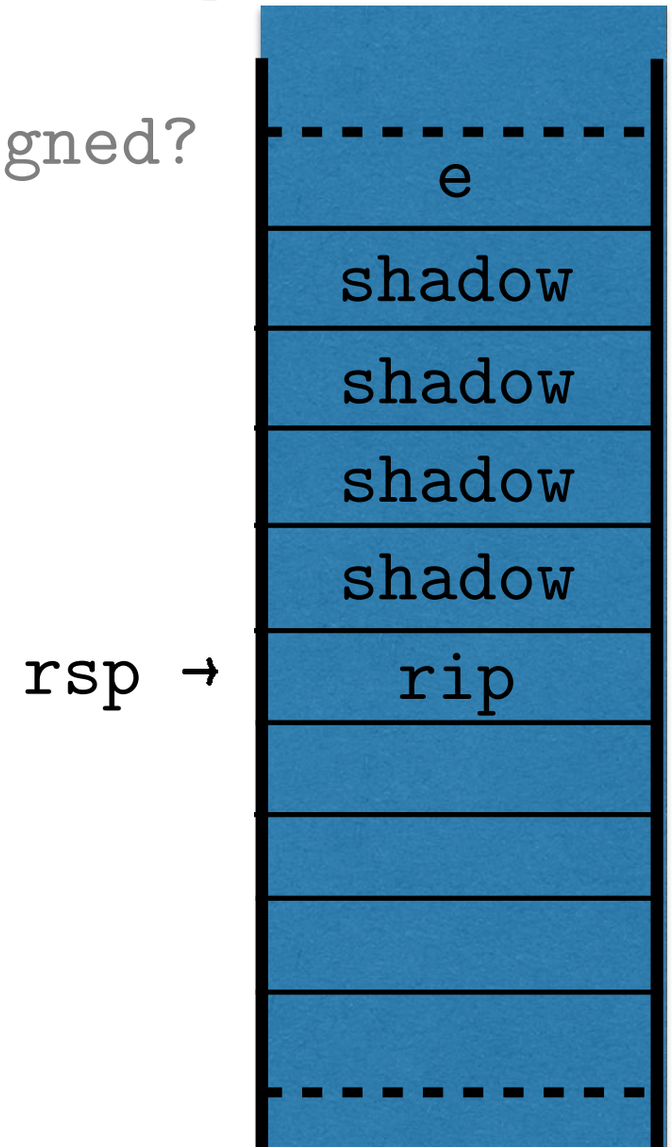
```
_int64 my_fun(_int64 a, _int64 b, _int64 c, _int64 d, _int64 e)
{_int64 x; _int64 y; ... return x;}
```

```
push e          ;5th arg in stack
mov  rcx, a     ;1st arg
mov  rdx, b     ;2nd arg
mov  r8 , c     ;3rd arg
mov  r9 , d     ;4th arg
sub  rsp, 32    ;shadow space
call my_fun
```



```
_int64 my_fun(_int64 a, _int64 b, _int64 c, _int64 d, _int64 e)
{_int64 x; _int64 y; ... return x;}
```

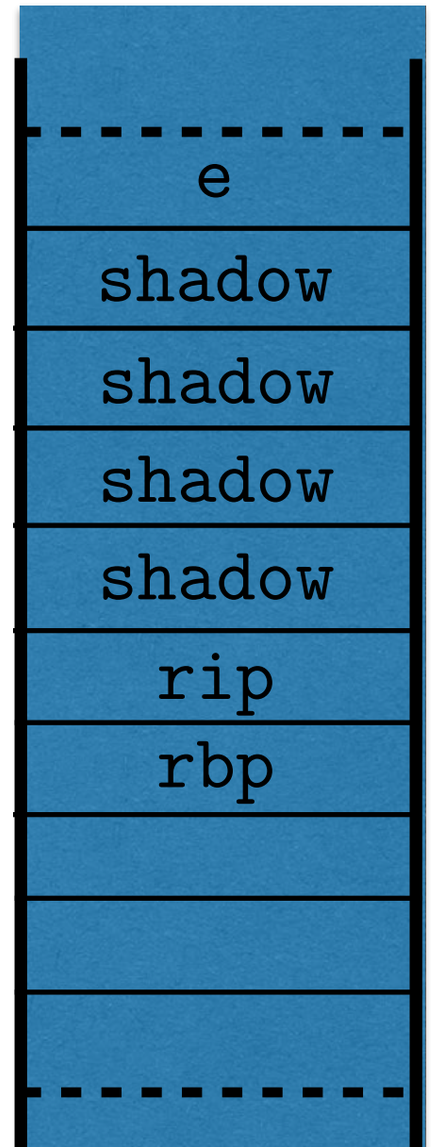
```
push e          ;5th arg in stack
mov  rcx, a     ;1st arg
mov  rdx, b     ;2nd arg
mov  r8 , c     ;3rd arg
mov  r9 , d     ;4th arg
sub  rsp, 32    ;shadow space
call my_fun
;is the stack aligned?
```



```
_int64 my_fun(_int64 a, _int64 b, _int64 c, _int64 d, _int64 e)
{_int64 x; _int64 y; ... return x;}
```

```
my_fun PROC:
  ;creating stack frame
  push rbp
  mov  rbp, rsp
```

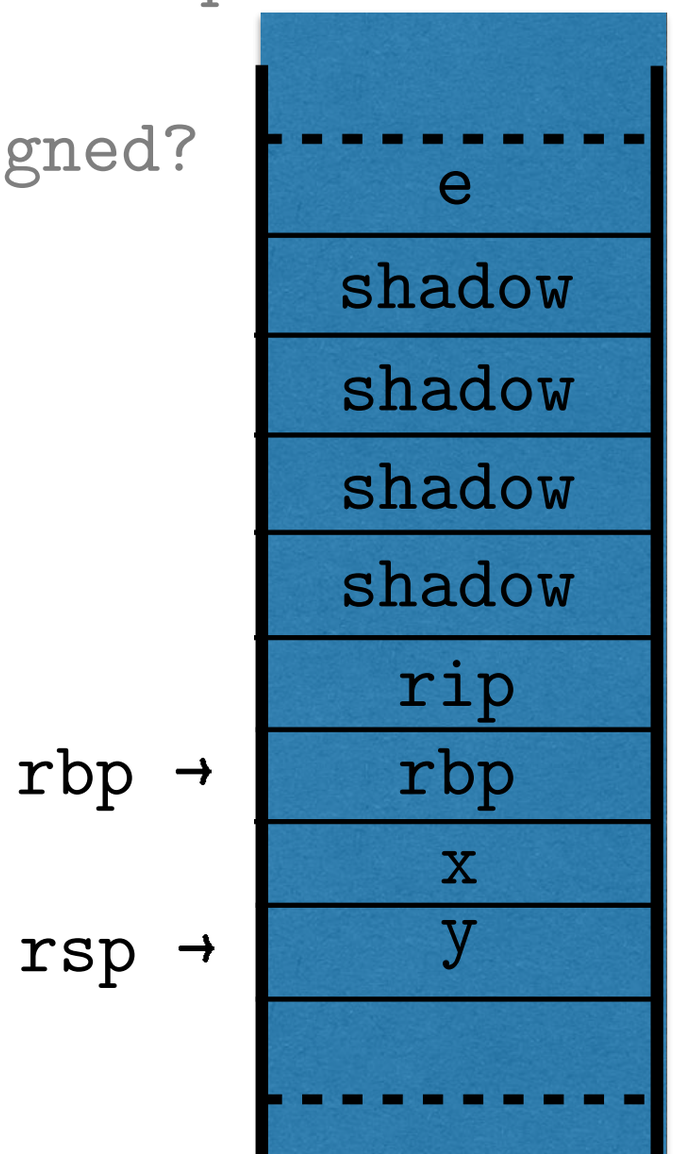
```
push e          ;5th arg in stack
mov  rcx, a     ;1st arg
mov  rdx, b     ;2nd arg
mov  r8 , c     ;3rd arg
mov  r9 , d     ;4th arg
sub  rsp, 32 ;shadow space
call my_fun
;is the stack aligned?
```



```
_int64 my_fun(_int64 a, _int64 b, _int64 c, _int64 d, _int64 e)
{_int64 x; _int64 y; ... return x;}
```

```
my_fun PROC:
  ;creating stack frame
  push rbp
  mov  rbp, rsp
  ;local variables
  sub  rsp, 16
```

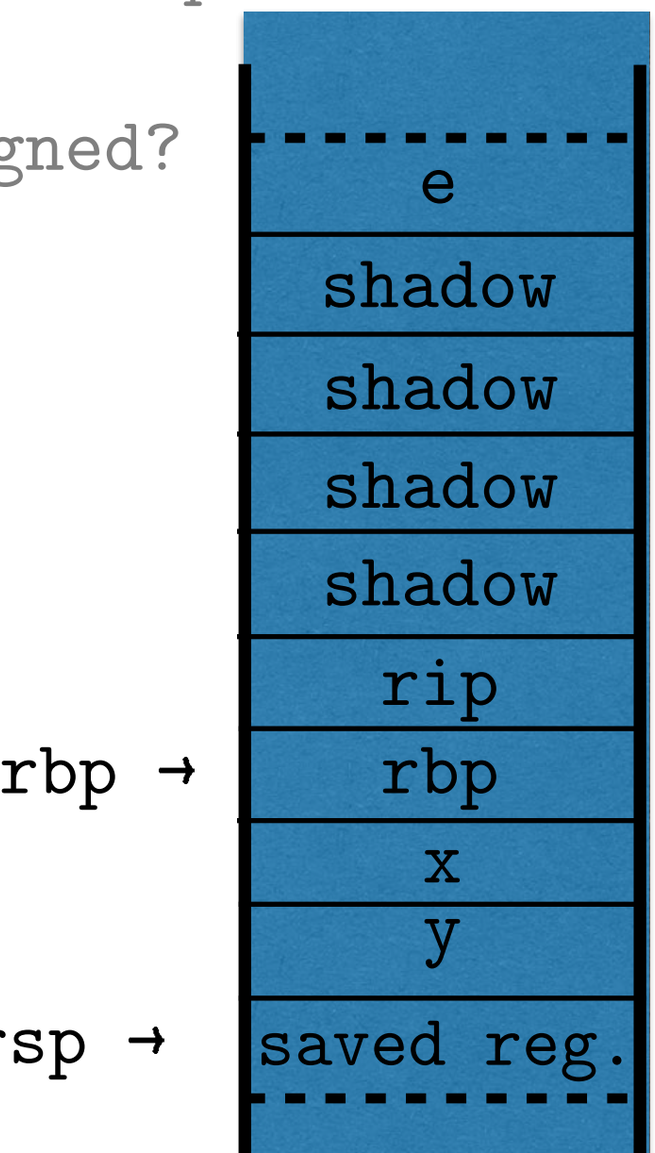
```
push e          ;5th arg in stack
mov  rcx, a     ;1st arg
mov  rdx, b     ;2nd arg
mov  r8 , c     ;3rd arg
mov  r9 , d     ;4th arg
sub  rsp, 32 ;shadow space
call my_fun
;is the stack aligned?
```



```
_int64 my_fun(_int64 a, _int64 b, _int64 c, _int64 d, _int64 e)
{_int64 x; _int64 y; ... return x;}
```

```
my_fun PROC:
  ;creating stack frame
  push rbp
  mov  rbp, rsp
  ;local variables
  sub  rsp, 16
  ;saving preserved regs
  ;(if necessary)
  push rbx
  ...
```

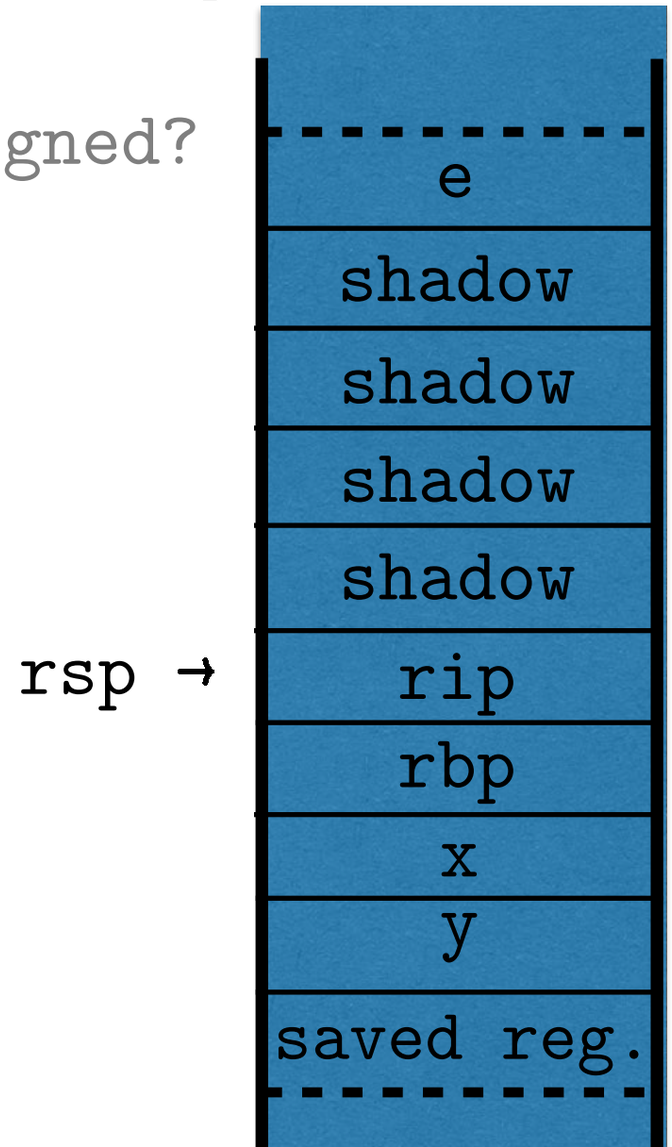
```
push e          ;5th arg in stack
mov  rcx, a     ;1st arg
mov  rdx, b     ;2nd arg
mov  r8 , c     ;3rd arg
mov  r9 , d     ;4th arg
sub  rsp, 32 ;shadow space
call my_fun
;is the stack aligned?
add  rsp, 40
```



```
_int64 my_fun(_int64 a, _int64 b, _int64 c, _int64 d, _int64 e)
{_int64 x; _int64 y; ... return x;}
```

```
my_fun PROC:
  ;creating stack frame
  push rbp
  mov  rbp, rsp
  ;local variables
  sub  rsp, 16
  ;saving preserved regs
  ;(if necessary)
  push rbx
  ...
  ;cleaning the stack
  pop  rbx
  add  rsp, 16
  pop  rbp
```

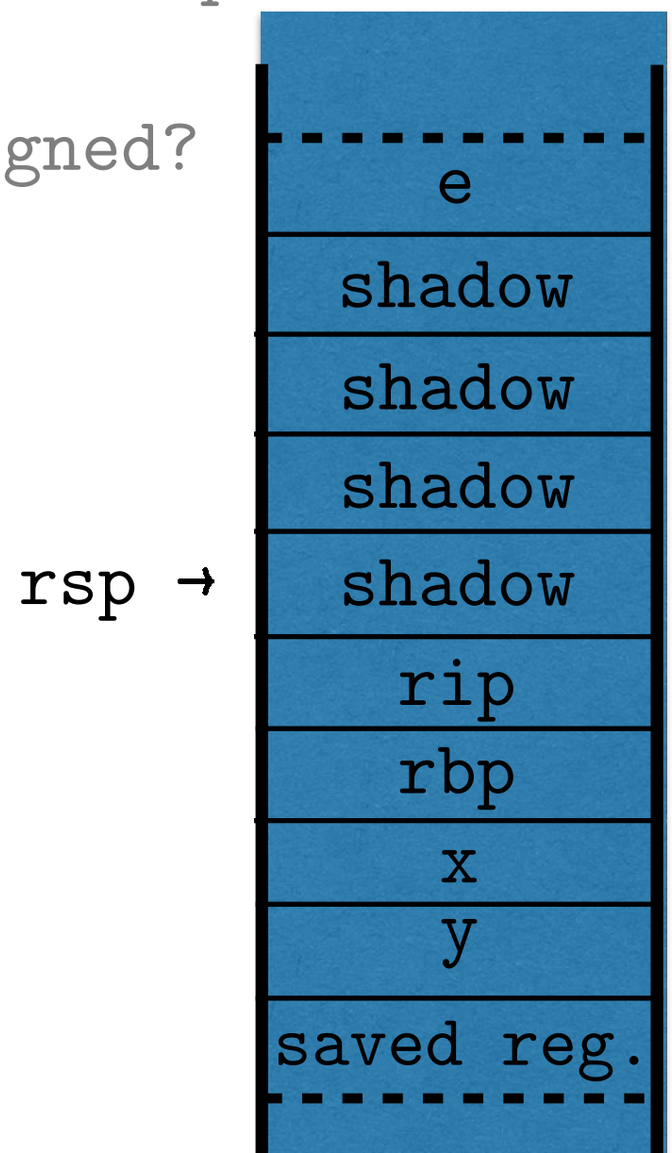
```
push e          ;5th arg in stack
mov  rcx, a     ;1st arg
mov  rdx, b     ;2nd arg
mov  r8 , c     ;3rd arg
mov  r9 , d     ;4th arg
sub  rsp, 32 ;shadow space
call my_fun
;is the stack aligned?
add  rsp, 40
```



```
_int64 my_fun(_int64 a, _int64 b, _int64 c, _int64 d, _int64 e)
{_int64 x; _int64 y; ... return x;}
```

```
my_fun PROC:
  ;creating stack frame
  push rbp
  mov  rbp, rsp
  ;local variables
  sub  rsp, 16
  ;saving preserved regs
  ;(if necessary)
  push rbx
  ...
  ;cleaning the stack
  pop  rbx
  add  rsp, 16
  pop  rbp
  ;returning
  mov  rax, [rbp+8]
  ret  0
my_fun ENDP
```

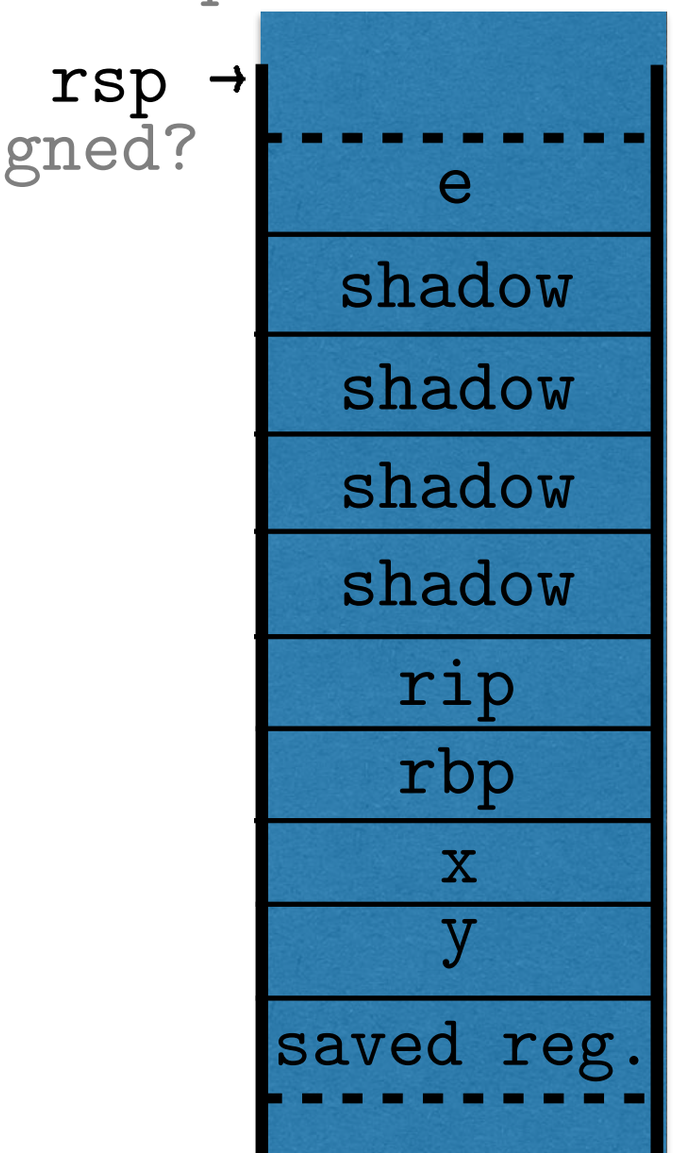
```
push e          ;5th arg in stack
mov  rcx, a     ;1st arg
mov  rdx, b     ;2nd arg
mov  r8 , c     ;3rd arg
mov  r9 , d     ;4th arg
sub  rsp, 32 ;shadow space
call my_fun
;is the stack aligned?
```



```
_int64 my_fun(_int64 a, _int64 b, _int64 c, _int64 d, _int64 e)
{_int64 x; _int64 y; ... return x;}
```

```
my_fun PROC:
  ;creating stack frame
  push rbp
  mov  rbp, rsp
  ;local variables
  sub  rsp, 16
  ;saving preserved regs
  ;(if necessary)
  push rbx
  ...
  ;cleaning the stack
  pop  rbx
  add  rsp, 16
  pop  rbp
  ;returning
  mov  rax, [rbp+8]
  ret  0
my_fun ENDP
```

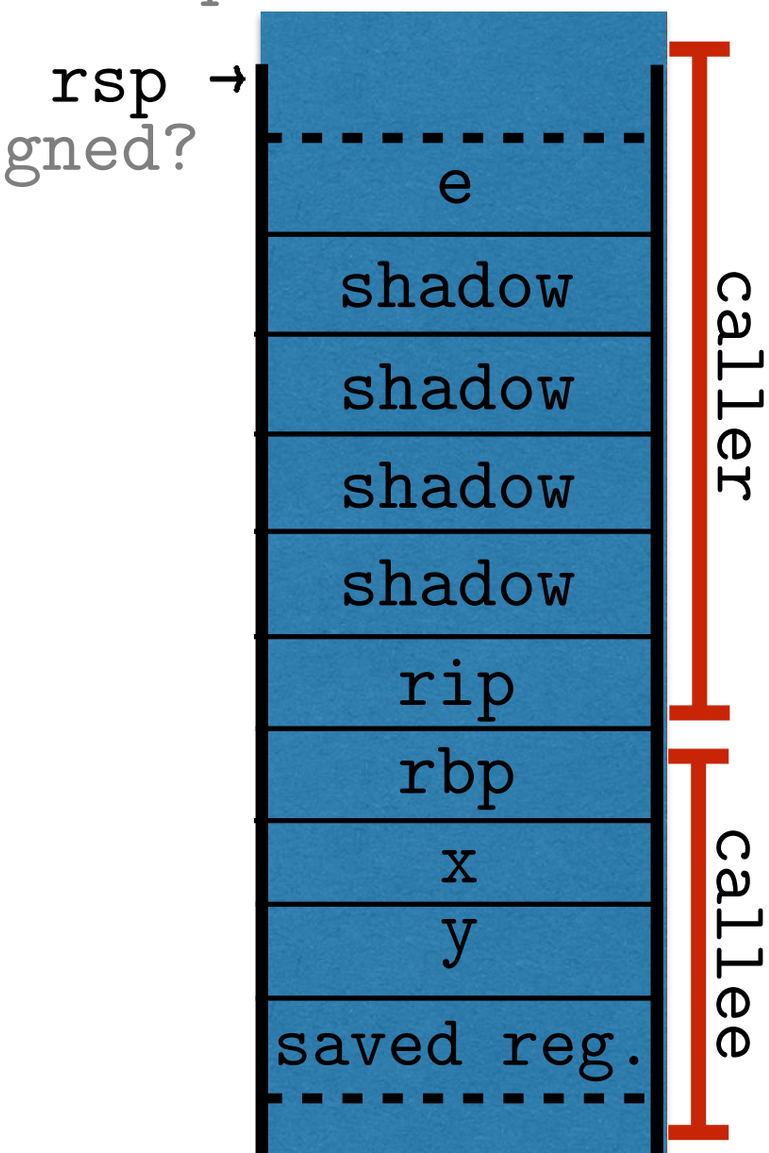
```
push e          ;5th arg in stack
mov  rcx, a     ;1st arg
mov  rdx, b     ;2nd arg
mov  r8 , c     ;3rd arg
mov  r9 , d     ;4th arg
sub  rsp, 32 ;shadow space
call my_fun
;is the stack aligned?
add  rsp, 40
```



```
_int64 my_fun(_int64 a, _int64 b, _int64 c, _int64 d, _int64 e)
{_int64 x; _int64 y; ... return x;}
```

```
my_fun PROC:
  ;creating stack frame
  push rbp
  mov  rbp, rsp
  ;local variables
  sub  rsp, 16
  ;saving preserved regs
  ;(if necessary)
  push rbx
  ...
  ;cleaning the stack
  pop  rbx
  add  rsp, 16
  pop  rbp
  ;returning
  mov  rax, [rbp+8]
  ret  0
my_fun ENDP
```

```
push e          ;5th arg in stack
mov  rcx, a     ;1st arg
mov  rdx, b     ;2nd arg
mov  r8 , c     ;3rd arg
mov  r9 , d     ;4th arg
sub  rsp, 32 ;shadow space
call my_fun
;is the stack aligned?
add  rsp, 40
```



Calling Convention and Shadow Space

- Advantage of x64 Microsoft Calling Convention vs. the x86 one?

- The **full shadow space** needs to be allocated **independently of number of input parameters**.

- It is used for debugging, for spilling input registers into the stack (useful for varargs functions), and for scratch space for the **callee function**.

- Shadow space doesn't need to be recreated for sequential calls!

Shadow Space

```
sub  rsp      , 48
```

```
int my_fun()
...
int main(){
...
my_fun(a,b,c,d,e)
my_fun(a,b,c,d,f,g)
...
}
```

Shadow Space

```
sub  rsp      , 48 52
```

```
int my_fun()
...
int main(){
...
my_fun(a,b,c,d,e)
my_fun(a,b,c,d,f,g)
...
}
```

Shadow Space

```
sub  rsp      , 48 52
mov  rax      , e
mov  [rsp+40], rax ;5th arg in stack
mov  rcx      , a   ;1st arg
mov  rdx      , b   ;2nd arg
mov  r8       , c   ;3rd arg
mov  r9       , d   ;4th arg
call my_fun
```

```
int my_fun()
...
int main(){
...
my_fun(a,b,c,d,e)
my_fun(a,b,c,d,f,g)
...
}
```

Shadow Space

```
sub  rsp      , 48 52
mov  rax      , e
mov  [rsp+40], rax ;5th arg in stack
mov  rcx      , a   ;1st arg
mov  rdx      , b   ;2nd arg
mov  r8       , c   ;3rd arg
mov  r9       , d   ;4th arg
call my_fun
mov  rax      , g
mov  [rsp+48], rax ;6th arg in stack
mov  rax      , f
mov  [rsp+40], rax ;5th arg in stack
call my_fun
```

```
int my_fun()
...
int main(){
...
my_fun(a,b,c,d,e)
my_fun(a,b,c,d,f,g)
...
}
```

Shadow Space

```
sub  rsp      , 48 52
mov  rax      , e
mov  [rsp+40], rax ;5th arg in stack
mov  rcx      , a   ;1st arg
mov  rdx      , b   ;2nd arg
mov  r8       , c   ;3rd arg
mov  r9       , d   ;4th arg
call my_fun
mov  rax      , g
mov  [rsp+48], rax ;6th arg in stack
mov  rax      , f
mov  [rsp+40], rax ;5th arg in stack
call my_fun
add  rsp      , 52
```

```
int my_fun()
...
int main(){
...
my_fun(a,b,c,d,e)
my_fun(a,b,c,d,f,g)
...
}
```

Code Generation Strategy for Non-leaf function x64

- Allocate enough stack space to support the call to the function with the most parameters [ONCE at the start of the function].
- Use the same stack space [and register, ofc] to pass parameters to ALL the functions called [Don’t push, but mov!]
- Be careful that the stack is 16-byte aligned upon entering a function, otherwise it could crash!

Summary

- Details on x86 Intel architecture.
- Introduction to x86 and x64 ISA and MASM assembler.
- Main conventions for function calling in Microsoft.
- Where next?
- x86 and x64 probably include the majority of personal devices used in real life. There is a plethora of information available on the web, and vibrant communities around them.
- If you want to read more:
  - “Introduction to Assembly language programming” (Dandamudi).
  - “The Intel 64 and IA-32 Architecture Software Developer Manuals”.