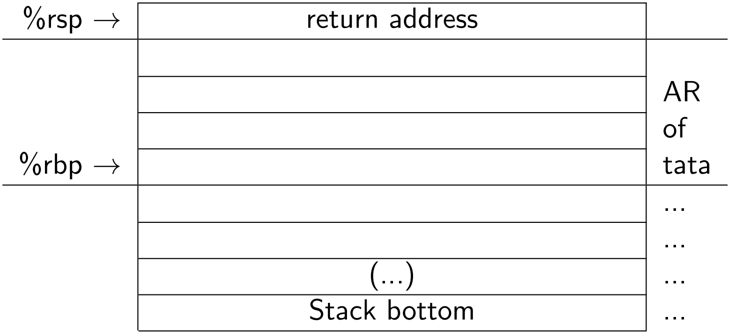
Before the `call toto`

- Drawing with the small addresses at the top
  - Reading order
  - Same view of memory as you see in gdb
- Descending stack, with pointer on last pushed value
- So stack grows up on the figure
- Warning: many figures you find in books or internet are upside down.
- All stack entries are 64-bit (this is ABI-dependent)

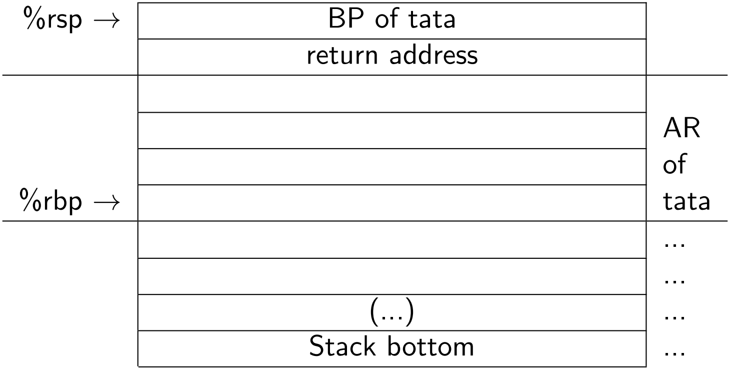
%rsp →	Top of stack	AR of tata
%rbp →		
		...
		...
	(...)	...
	Stack bottom	...

just after the call toto



%rsp →	return address	
		AR
		of
%rbp →		tata
		…
		…
	(…)	…
	Stack bottom	…

just after the push %rbp



%rsp →

BP of tata

return address

AR of tata

%rbp →

...

...

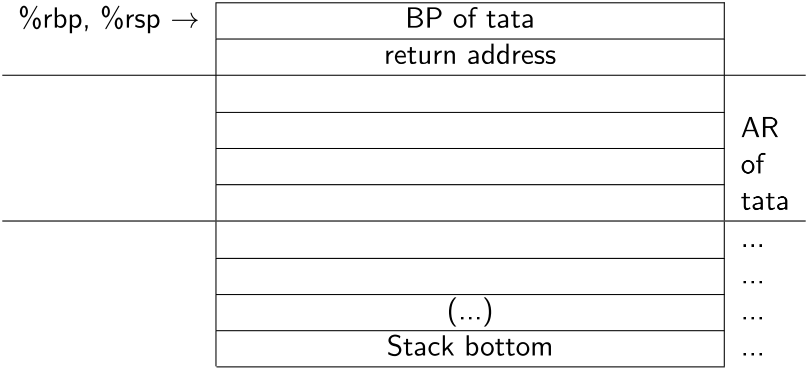
(...)

...

Stack bottom

...

just after the `movq %rsp, %rbp`



%rbp, %rsp →

BP of tata

return address

AR of tata

...

...

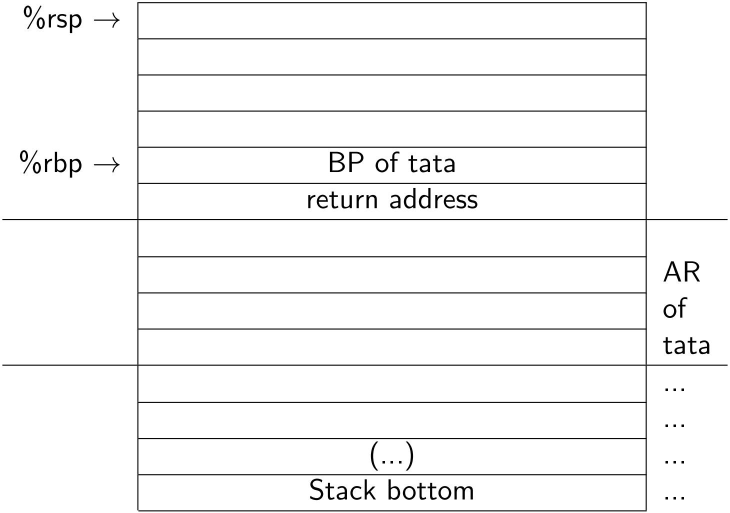
(...)

...

Stack bottom

...

just after the `subq $32, %rsp`



%rsp →

%rbp →

BP of tata

return address

AR of tata

(...)

Stack bottom

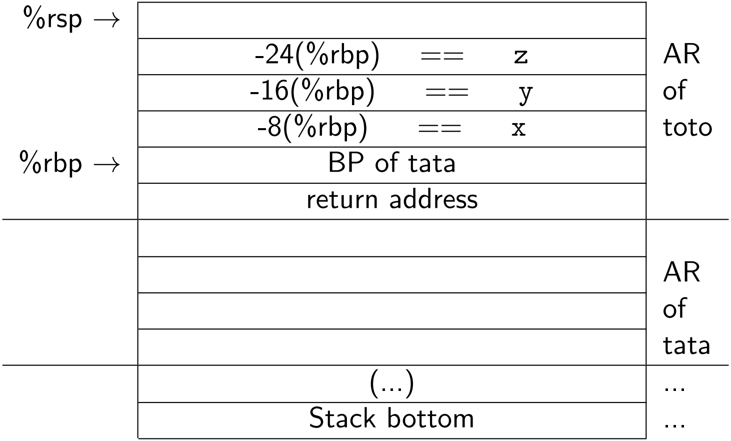
...

...

...

...

End of prologue, execution can begin



%rsp →

-24(%rbp) == z

-16(%rbp) == y

-8(%rbp) == x

%rbp → BP of tata

return address

AR of toto

AR of tata

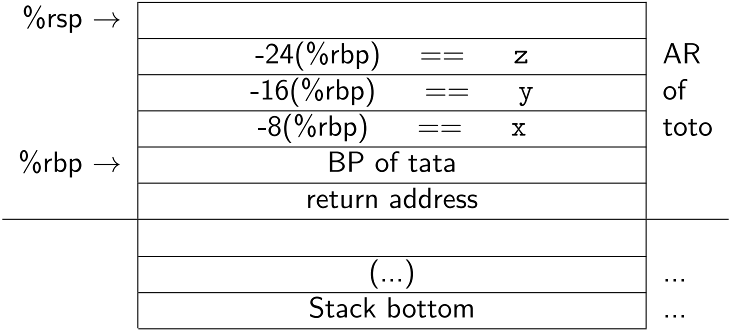
(...) ...

Stack bottom ...

We have an empty location at the top due to 16-byte alignment: the ABI states that *the value (%rsp + 8) is always a multiple of 16 when control is transferred to the function entry point.*

During execution of the procedure body

`toto` can push stuff on the stack, for instance when calling putchar



%rsp →

-24(%rbp) == `z`

-16(%rbp) == `y`

-8(%rbp) == `x`

%rbp → BP of tata

return address

(...)

Stack bottom

AR of toto

...

...

Epilogue

Please leave the stack in the same state as you found it when entering. 3 options:

```
leave
ret
```

```
movq %rbp, %rsp
pop %rbp
ret
```

```
addq $32, %rsp
pop %rbp
ret
```

The third is the most explicit but takes more space (requires to encode the constant 32).