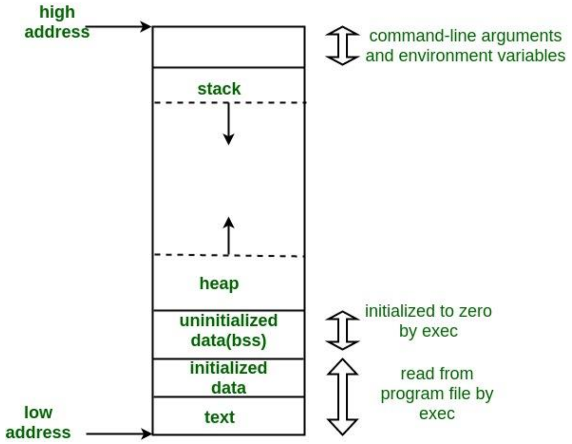
מבנה מחשב

תרגול 4

מבנה התוכנית

Introduction

- When we wrote: ‘int n = 10’; the compiler allocated the variable’s memory address and labeled it ‘n’.
- In Assembly, we’ll do it ourselves.
- Our program is comprised of 4 parts:
  - □ Data Segment
  - □ Text (=Code) Segment
  - □ Stack Segment
  - □ Heap


high address
command-line arguments and environment variables
stack
heap
uninitialized data(bss)
initialized to zero by exec
initialized data
read from program file by exec
text
low address

The Data Segment

- `.data` - start of data part of the code.
- We can give each data item a label by writing:

  `my_label:`

The Data Segment

- We must specify the type of each data item, e.g. byte, word, double, quad. For instance:

```
counter:        .word 15
```

- In memory, it will look like:

$$\underbrace{00001111}_{\text{counter}}\ \underbrace{00000000}_{\text{counter+1}}$$

- The compiler will translate ’15’ to binary.

The Data Segment

- Example:

```
            .data
vec:        .word          12089, -89, 130
avi:        .word          72
```

- So ‘vec’ is actually an array of words. Each item should be read as word, else it would have a different meaning.

The Data Segment

- Another Example:

```
              .data
alice:        .byte             3, 5 ,10, -7
bob:          .byte             9, 20
```

- In the memory it will be stored sequentially:

$$\underbrace{00000011}_{\text{alice}}\ \underbrace{00000101}_{\text{alice+1}}\ \underbrace{00001010}_{\text{alice+2}}\ \underbrace{11111001}_{\text{alice+3}}\ \underbrace{00001001}_{\text{bob}}\ \underbrace{00010100}_{\text{bob+1}}$$

The Data Segment

- Each variable type can store a **signed** or an **unsigned** value.
- Therefore, byte, for example can store a value in the range: -128 – 255.
- For simply allocating memory space we can use:

  ```
  .space      10*4
  ```

  for example.

The Data Segment

- `.section .rodata`

  means that from this point until `.data` or `.text` this section contains only read-only data.

The Text Segment

- `.text` – start of the text part of the code.

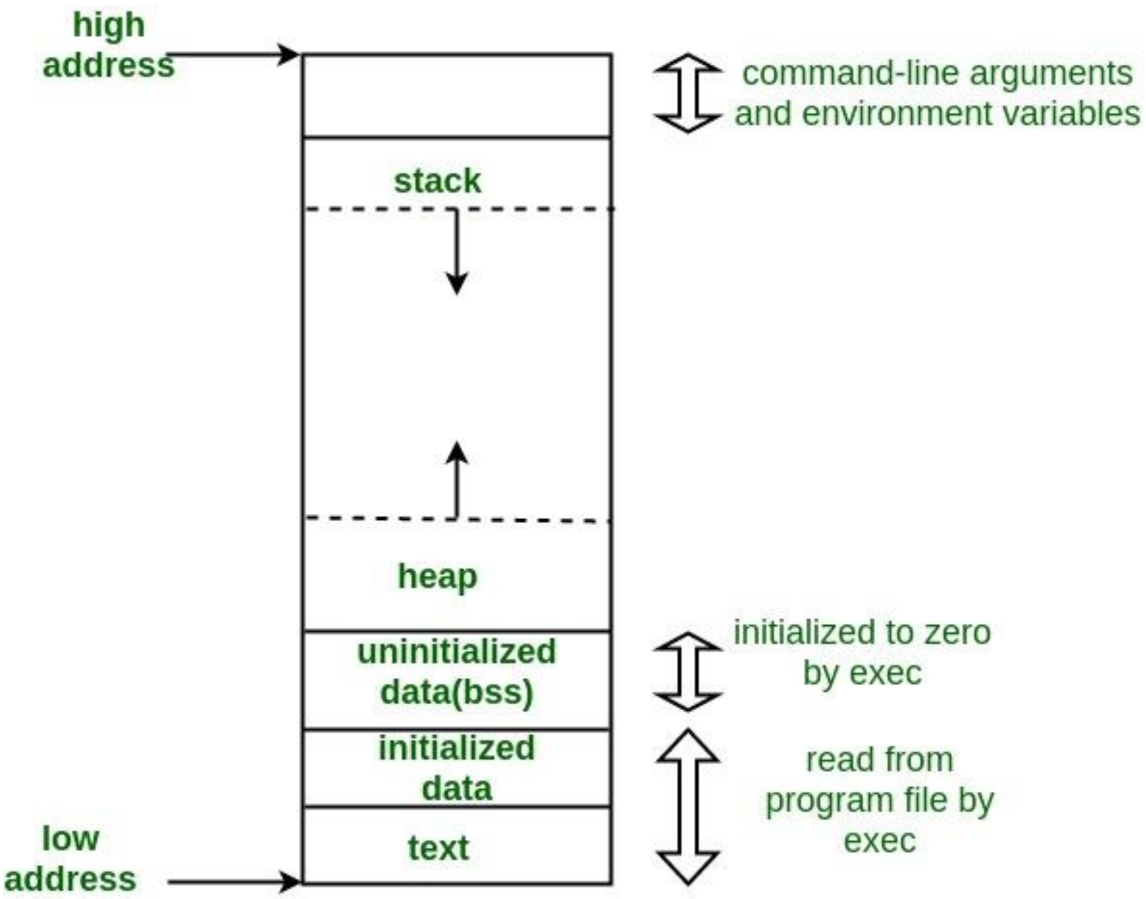
  for instance:

  ```
  .text
  addl  $20, %eax
  …
  ```

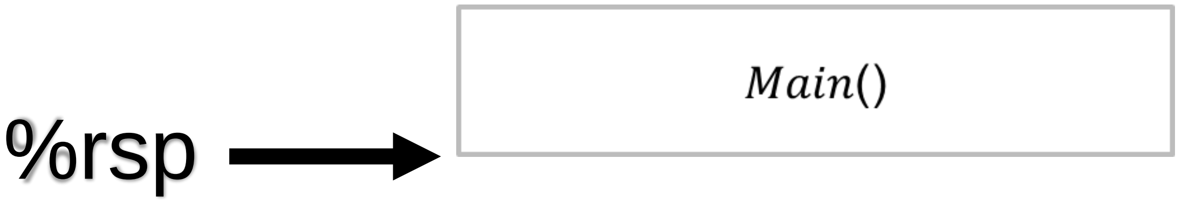
The Stack Segment

- We’ll use a stack, rather different then the “common” one.
- We can read and write data anywhere on the stack.
- But, for each procedure / function, we’ll add space in the stack in a LIFO manner.
- The Stack Pointer (%rsp) is a register pointing to the head of the stack.

The Stack Segment


high address
command-line arguments and environment variables
stack
heap
uninitialized data(bss)
initialized to zero by exec
initialized data
text
read from program file by exec
low address

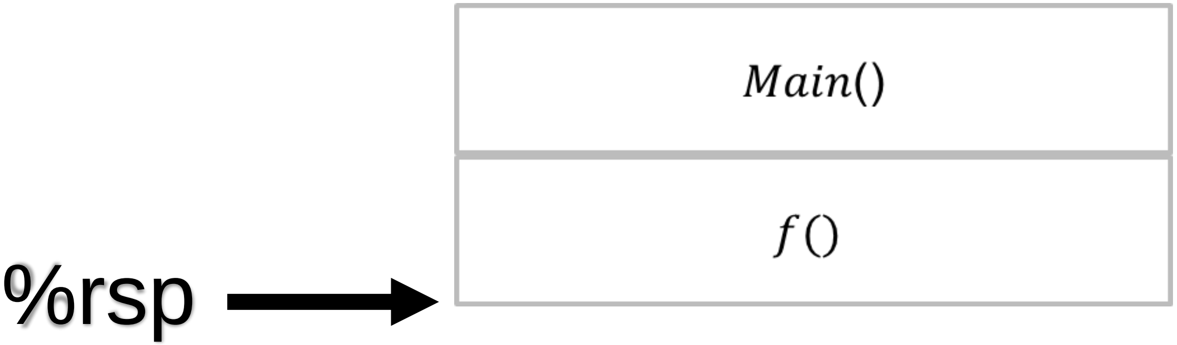
The Stack Segment



%rsp

$Main()$

The Stack Segment

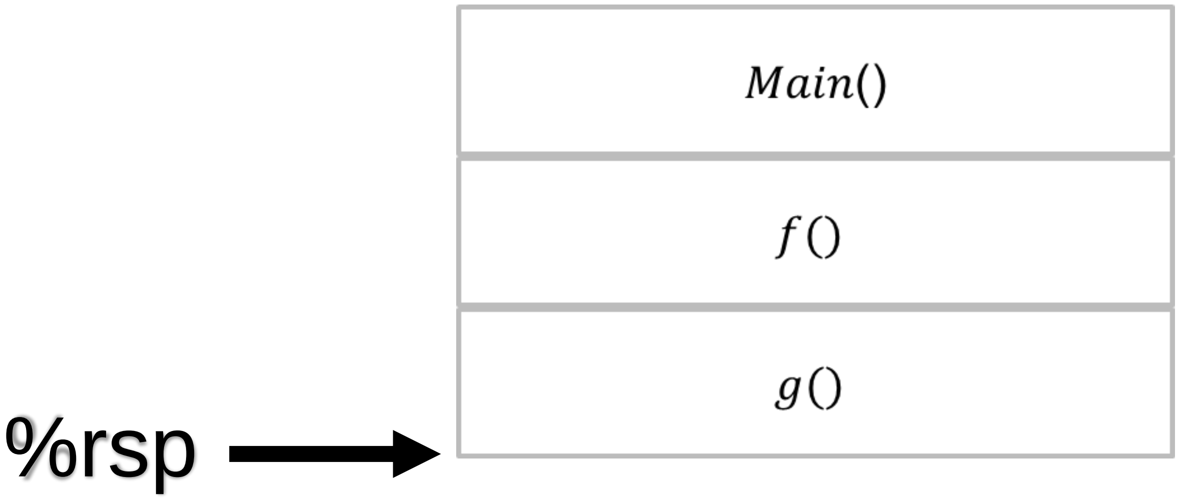


$Main()$

$f()$

%rsp

The Stack Segment



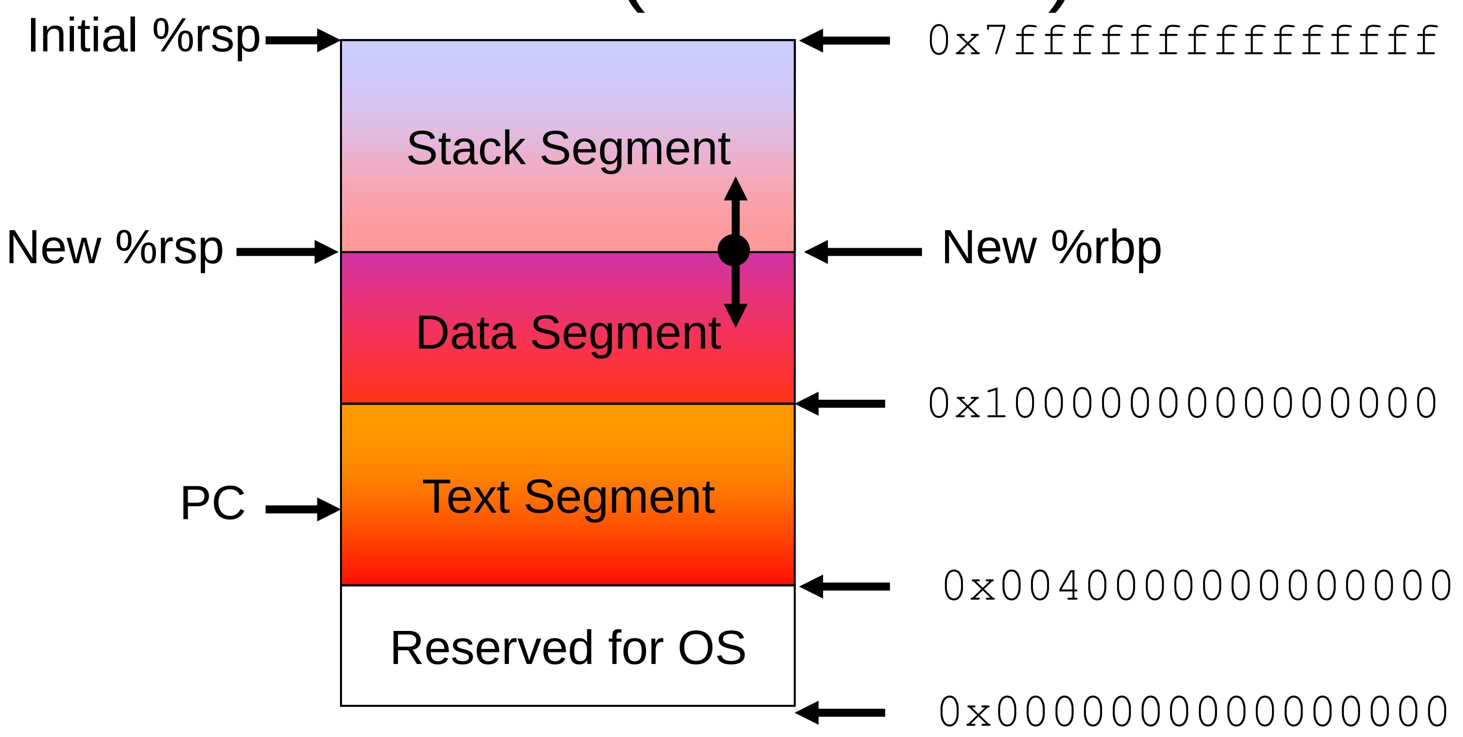
The Stack Segment

- For example:

```
.text
addq  $-64, %rsp
...               (the procedure itself)
addq  $64, %rsp
```

The Stack Segment

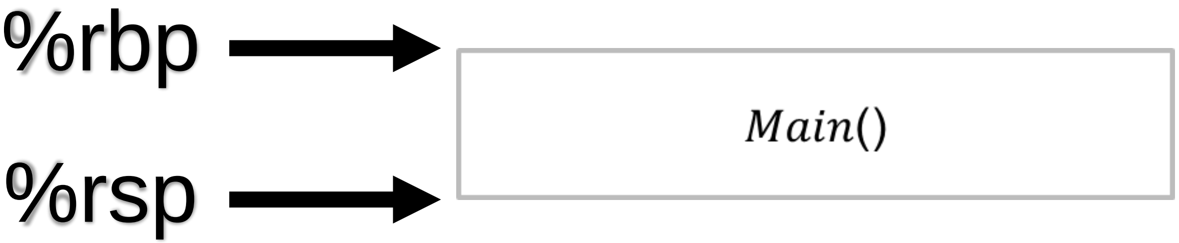
- Why did we add -64 (and not 64)?



The Stack Segment

- To manage a variable-size stack frame, x86-64 code uses %rbp as a Frame Pointer.
  - Note that if frame size is constant, %rbp is a general-purpose register
- The frame pointer is used to store the contents of the stack pointer at the beginning of the procedure.
- So if we’ll make an error managing the SP, we can recover its value, as it was when the procedure started.

The Stack Segment

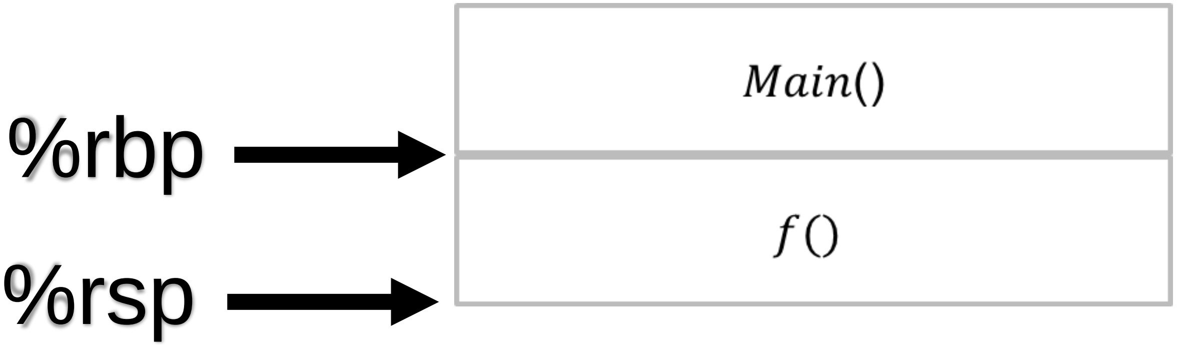


%rbp

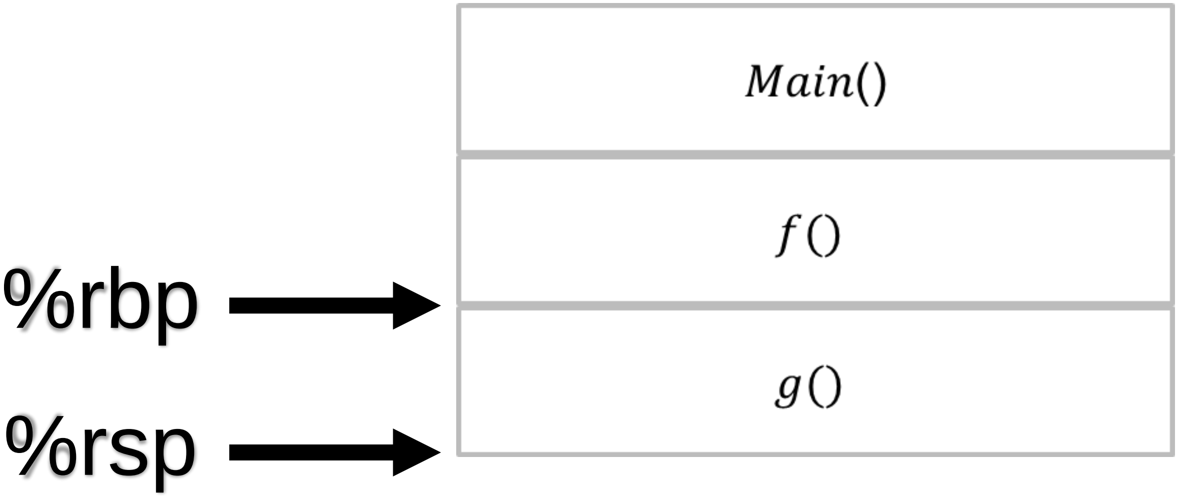
$Main()$

%rsp

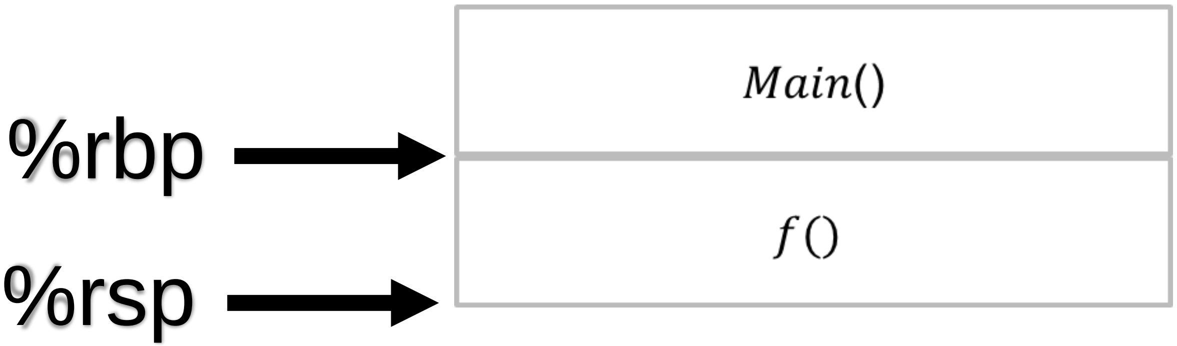
The Stack Segment



The Stack Segment

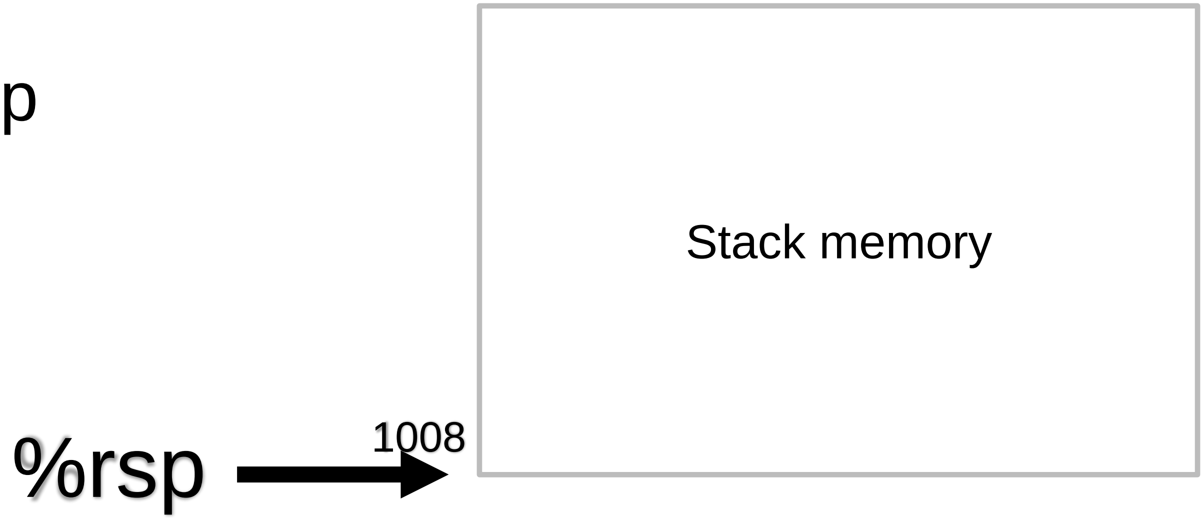


The Stack Segment



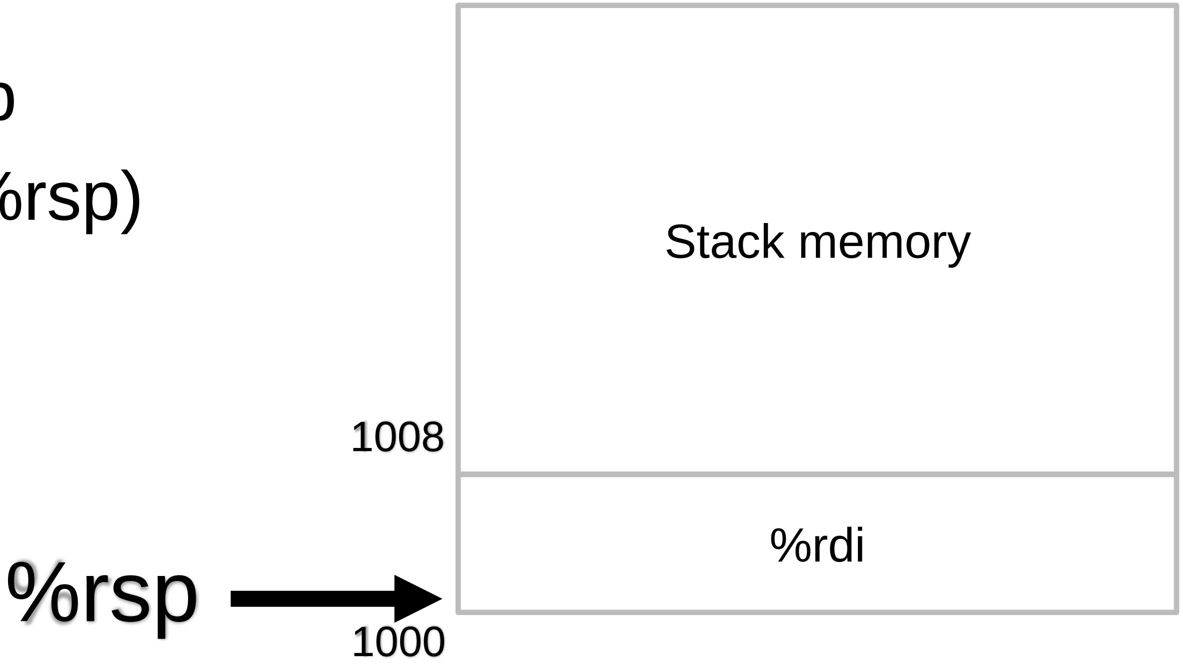
The Stack Segment

- Saving(push) quad value from %rdi into the stack:
  - subq $8, %rsp



The Stack Segment

- Saving(push) quad value from %rdi into the stack:
  - □ subq $8, %rsp
  - □ movq %rdi, (%rsp)



The Stack Segment

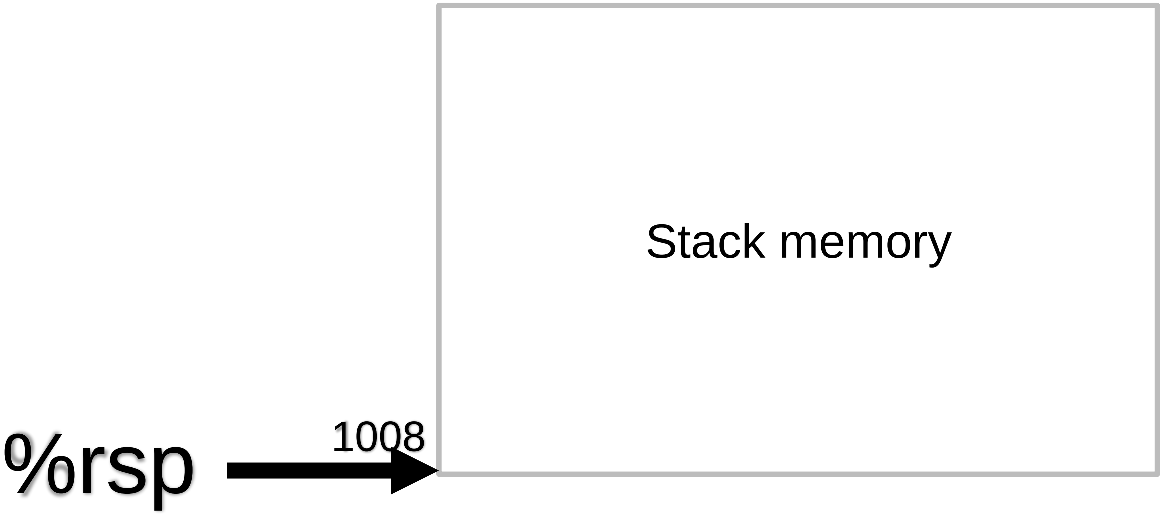
- Or simply use:
  - pushq %rdi
- For pop element from stack to %rdi:
  - popq %rdi

The Stack Segment

- Or simply use:
  - pushq %rdi
- For pop element from stack to %rdi:
  - popq %rdi

The Stack Segment

- Example:
  - □ pushq %rdi



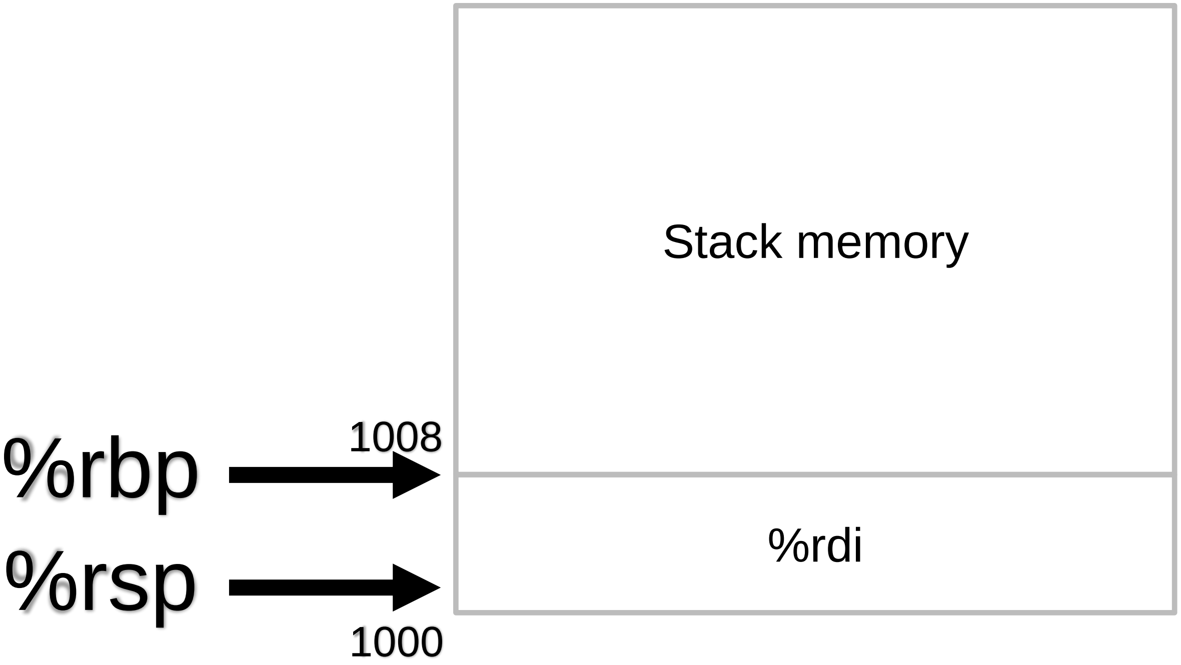
Stack memory

%rsp

1008

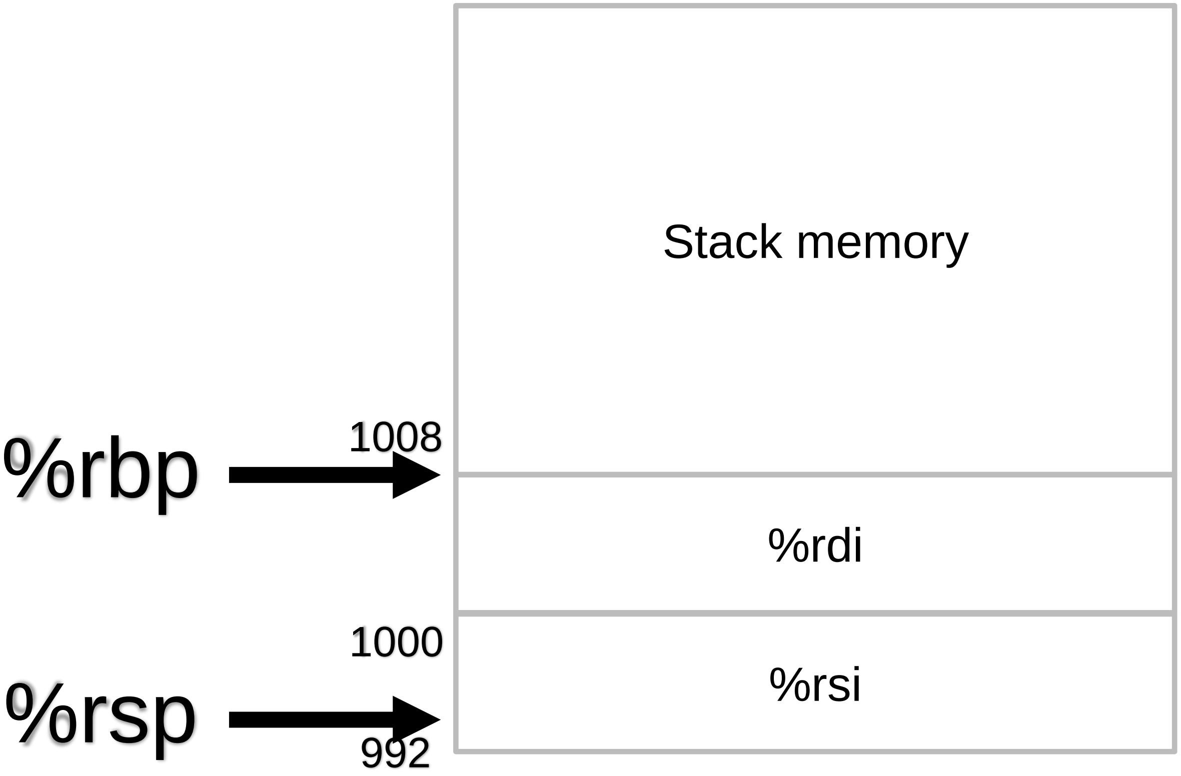
The Stack Segment

- Example:
  - □ pushq %rdi



The Stack Segment

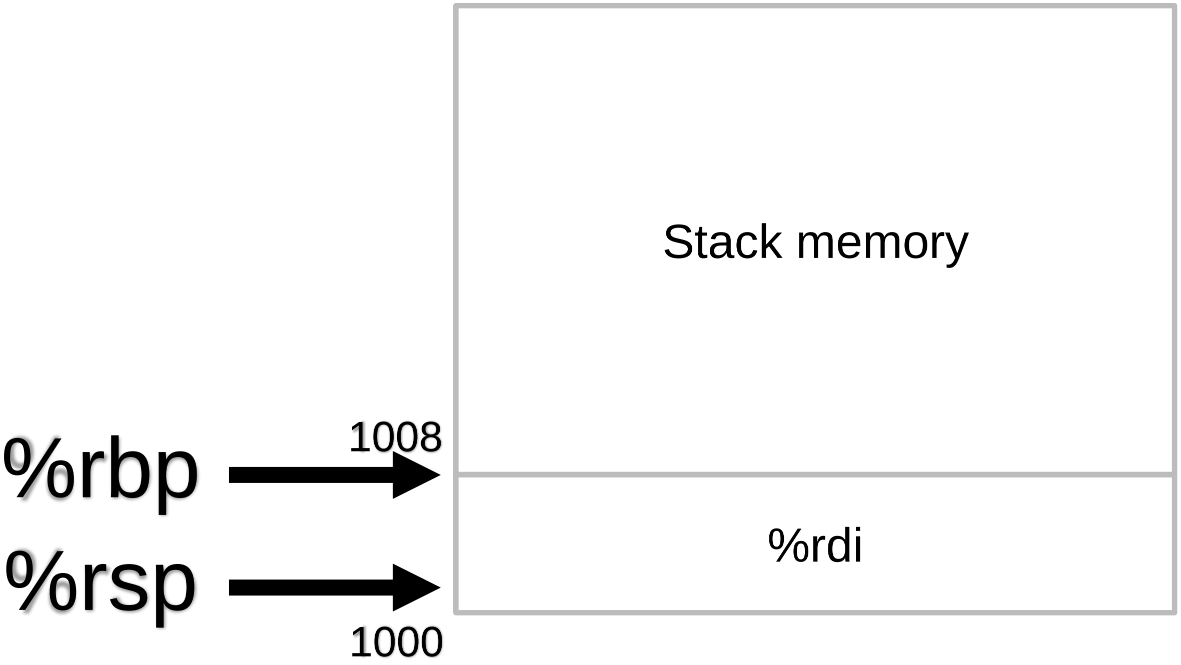
- Example:
  - □ pushq %rdi
  - □ pushq %rsi



%rbp remains the same

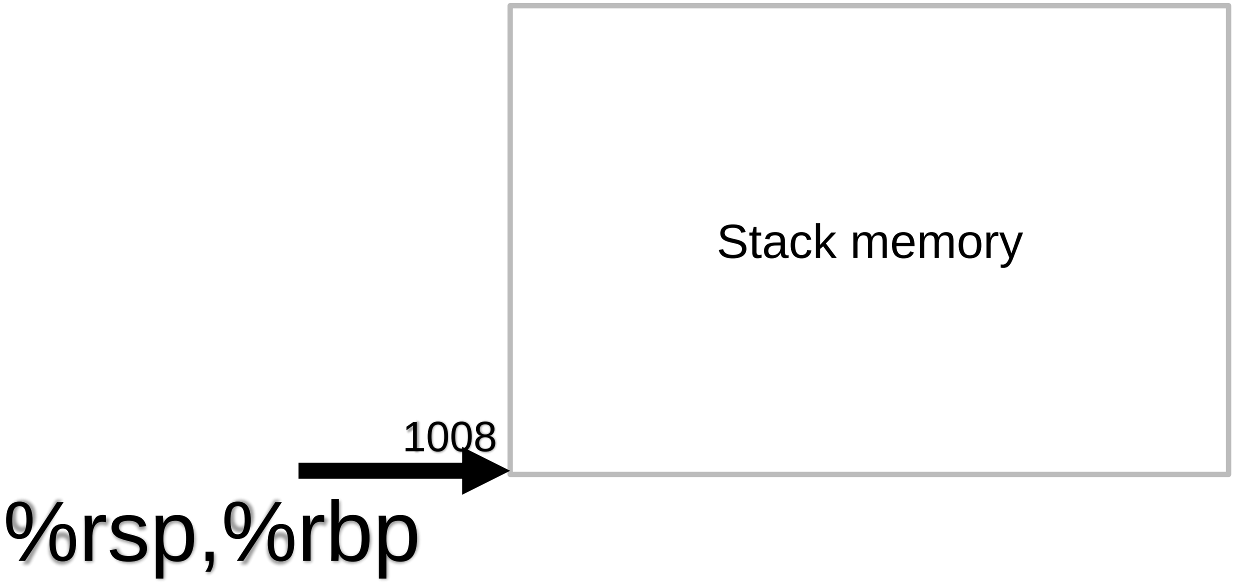
The Stack Segment

- Example:
  - pushq %rdi
  - pushq %rsi
  - popq %rax



The Stack Segment

- Example:
  - □ pushq %rdi
  - □ pushq %rsi
  - □ popq %rax
  - □ popq %rsi



Stack memory

1008

%rsp,%rbp

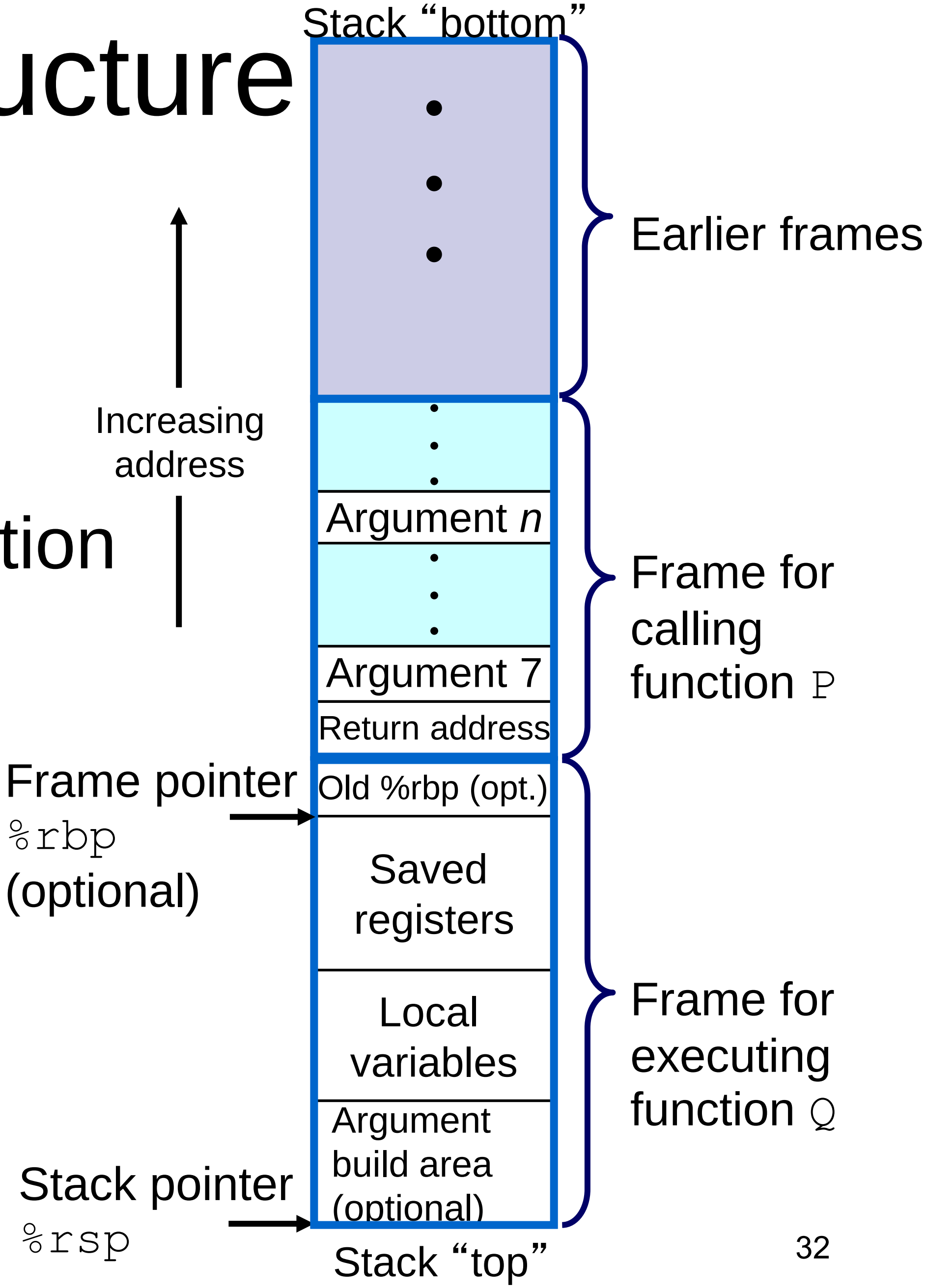
Program Structure

```
   .data
l: …            #all global data.
   …
   .section    .rodata
k: …            #all RO data such as string formats for printf.
   .text       #the beginning of the code
.globl  main   #defining the label “main” as the starting point.
   .type       main, @function       #defining “main” as function.
main:
                       # in case of a variable-size stack frame:
   pushq       %rbp    #saving the old frame pointer.
   movq        %rsp,   %rbp   #creating the new frame pointer.
   …                   #saving callee-save registers if needed

   …           #The program code

   …                   #restoring callee-save registers if needed
   movq        %rbp,   %rsp   #restoring the old stack pointer.
   popq        %rbp    #restoring the old frame pointer.
   ret         #returning to the function that called us.
```

Stack Frame Structure

The stack is used for:

- passing arguments
- storing return information
- saving registers
- local storage


Stack "bottom"
Increasing address
Earlier frames
Argument *n*
Argument 7
Return address
Frame for calling function P
Frame pointer %rbp (optional)
Old %rbp (opt.)
Saved registers
Local variables
Argument build area (optional)
Frame for executing function Q
Stack pointer %rsp
Stack "top"