CSD1100

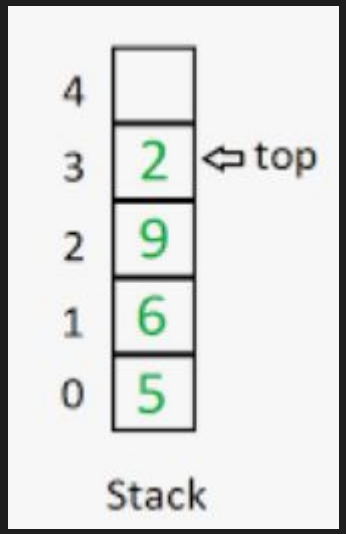
Assembler - Functions

Vadim Surov

Stack

- The stack is a special area in memory for placing temporarily data.
- Because the data is added and removed in a last-in-first-out manner, stack-based memory allocation is very simple and typically much faster than other methods of allocation.
- So far we used the stack to save/restore results of calculations temporarily in calculations and function calls.
- The following code is just an example how to work with the stack by pushing and popping values from and to registers.

```
8   section .data
9   fmt     db "%lld",10,0
10
11  section .text
12      global _start
13      extern printf
14
15  _start:
16      mov rax, 6
17      mov rbx, 9
18      mov rcx, 2
19      push rax
20      push rbx
21      push rcx
22      pop rax
23      pop rbx
24      pop rcx
25
26      PRINTF fmt, rax
27      EXIT
```



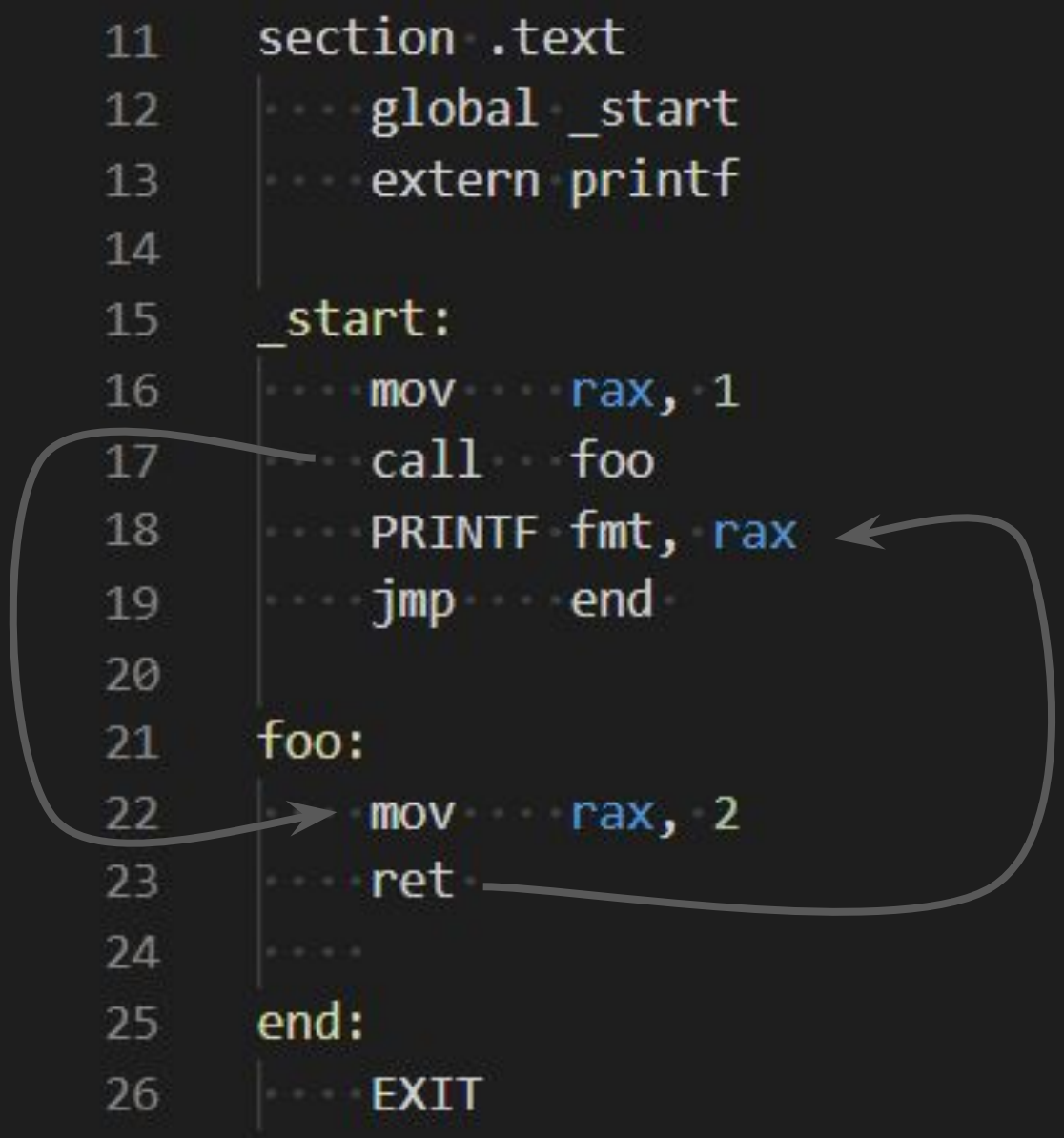
More about the stack

- The stack is reserved at the end of the memory area.
- As data is pushed to the stack, it grows toward smaller addresses.
- The **rsp** register contains the address of the top of the stack.
- The rsp register decrease or increase with each data element added or removed to the stack, pointing to the new top of the stack.
- **Function calls heavily relies on using the stack.**

Functions

- How to create and use functions in assembly language programs?

- The following example calls foo function to set 2 to the rax register.



call

- The **call** instruction is used to pass control to a function.
- When the call instruction is performed, the next instruction executed is the first instruction in the function.
- Continuing to step through the program, the next instruction in the function is performed, and so on until the **ret** instruction, which returns the control.
- The call instruction places the return address from the calling code onto the top of the stack, so the function knows where to return.

Calling side effects

- If you call a function developed by someone else there is no guarantee that registers will be in the same state when the function is finished as they were before the function was called.

- So, it is crucial that you save the current state of the registers before calling the function, and then restore them after the function returns.

Stack as a storage

- Obvious solution for this is to use the stack.
- The following code store and restore the rax register value in the stack before and after calling the foo function.

```
11  section .text
12      global _start
13      extern printf
14
15  _start:
16      mov     rax, 1
17      push    rax
18      call    foo
19      pop     rax
20      PRINTF fmt, rax
21      jmp     end
22
23  foo:
24      mov     rax, 2
25      ret
26
27  end:
28      EXIT
```

Stack as a storage

- We can use registers to pass parameters.
- There are 16 registers available **for everything.** What if registers not enough? Use stack to save them.
- The following code stacks 3 registers to use them for parameters and the returning result of a function:

```
15  _start:
16      push    rax
17      push    rbx
18      push    rcx
19
20      mov     rax, 1
21      mov     rbx, 2
22      call    add
23
24      PRINTF fmt, rcx
25
26      pop     rcx
27      pop     rbx
28      pop     rax
29
30      jmp     end
```

```
32  add:
33      add     rbx, rax
34      mov     rcx, rbx
35      ret
36
37  end:
38      EXIT
```

Comments

- Note that pop instructions go in the reverse order to push instructions.

- 32-bit assembler has pusha and popa instructions to save and restore all registers. Those instructions are not supported in 64-bit mode.

Stack as a storage for parameters

- Trying to keep track of which function uses which registers and memory, or which registers and memory are used to pass which parameters, can be a nightmare.
- The “C solution” for passing input values to functions is to use the stack.
- The stack is accessible from any functions used within the program.
- Using the stack creates a clean way to pass data between the main program and the functions sharing registers and memory, without having to worry about conflicts.

Parameters on the stack

- The following example uses the stack to pass parameters.

- All the parameters are located “underneath” the top of the stack.
- To retrieve the input parameters from the stack, efficient addressing is used:
    - [rsp+8] - the first parameter (Why +8, not +0?),
    - [rsp+16] - the second, and so on.

```
15  _start:
16      push    rax
17      push    rbx
18      push    rcx
19
20      push    1
21      push    2
22      call    add
23      add     rsp, 16  ; instead of 2 pops
24
25      PRINTF fmt, rcx
26
27      pop     rcx
28      pop     rbx
29      pop     rax
30
31      jmp     end
32
33  add:
34      mov     rax, [rsp+8]
35      add     rax, [rsp+16]
36      mov     rcx, rax
37      ret
38
39  end:
40      EXIT
```

Using rbp register

- There is a problem with this technique, however.
- If the function pushing/popping any data onto the stack it would change the stack pointer in rsp and thus throw off the efficient addressing values for accessing the parameters in the stack.
- To avoid this problem, it is common practice to copy the rsp register value to the **rbp** register when entering the function.

Function’s prologue and epilogue

- The following code demonstrates use of register rbp that always contains the correct pointer to the top of the stack when the function is called.
- The first two instructions at the top of the code save the original value of **rbp** to the top of the stack, and then copy the current **rsp** stack pointer (now pointing to the original value of rbp in the stack) to the rbp register.

```
15  _start:
16      push    rax
17      push    rbx
18      push    rcx
19
20      push    1
21      push    2
22      call    add
23      add     rsp, 16  ; instead of
24
25      PRINTF fmt, rcx
26
27      pop     rcx
28      pop     rbx
29      pop     rax
30
31      jmp     end
```

```
33  add:
34      ; Prologue
35      push rbp
36      mov  rbp, rsp
37
38      mov     rax, [rbp+16]
39      add     rax, [rbp+24]
40      mov     rcx, rax
41
42      ; Epilogue
43      mov rsp, rbp
44      pop rbp
45
46      ret
47
48  end:
49      EXIT
```

Function’s prologue and epilogue

- After the function processing completes, the last two instructions in the function retrieve the original value in the rsp register that was stored in the rbp register, and restore the original rbp register value.

- rbp is called **the frame pointer**.

Local memory on stack

- The stack is often used to store variables of fixed length local to the currently active functions.
- Memory on the stack is automatically, and very efficiently, reclaimed when the function exits, which can be convenient for the programmer if the data is no longer required.
- To get a local variable we can use the frame pointer and unique offset value for the variable.

System V

- Both Mac OS X and Linux follow the **System V** (pronounced: "System Five") calling convention for x86-64.
- Integer parameters to functions are passed in the registers rdi, rsi, rdx, rcx, r8, r9.
  - Further values are passed on the stack in reverse order.
  - Parameters passed on the stack may be modified by the called function.

System V

- Functions are called using the **call** instruction that pushes the address of the next instruction to the stack and jumps to the operand.
- Functions return to the caller using the **ret** instruction that pops a value from the stack and jump to it.
- Functions preserve the registers rbx, rsp, rbp, r12, r13, r14, and r15;
- Functions use rax, rdi, rsi, rdx, rcx, r8, r9, r10, r11 as scratch registers.
- The return value is stored in the **rax** register, or if it is a 128-bit value, then the higher 64-bits go in rdx.

References

1. Calling convention
   https://wiki.osdev.org/System_V_ABI