x86 branches & subroutines

CMSC 313

Raphael Elspas

Flags review

- The EFLAGS register is a 32 bit register that stores flags bit by bit.
- The relevant ones in this class are:
  - OF – overflow, 1 if the previous operation produces an overflow (POS+POS=NEG or NEG+NEG=POS). This flag is only triggered on signed operations
  - CF – carry, 1 if the previous operation produces a carry out bit = 1. This flag is only triggered on unsigned operations
  - ZF – zero, 1 if the previous operation produced a zero
  - SF – sign, 1 if the previous operation produced a negative number (first bit is zero).

Flags cont.

- Operations may set, clear, modify, or test (view) a flag
- Some operations do not affect any flags
- RFLAGS is the 64 bit version of EFLAGS (32 bit)

Branch/Jump instructions

- x86 refers to all jumps and branches as **jump** instructions
- Unconditional jump is **JMP**
- Conditional jumps are called **Jcc**, which stands for “Jump condition”**. cc** represents that there may be two letters as part of the conditional jump command such as **JGL** or **JLE**

x86 jump (jcc)

Inst		Sign
JL	<	Signed
JLE	<=	Signed
JB	<	Unsigned
JBE	<=	Unsigned
JG	>	Signed
JGE	>=	Signed
JA	>	Unsigned
JAE	>=	Unsigned

Inst	
JE	==
JNE	!=
JZ	Previous == 0
JNZ	Previous != 0
JS	Previous < 0
JNS	Preivious >= 0

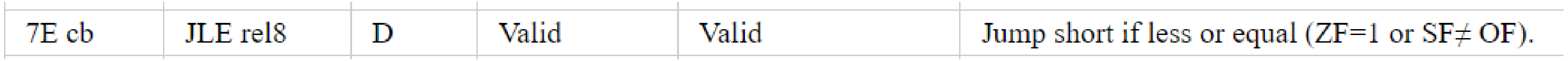
More located here:
https://www.felixcloutier.com/x86/jcc

Compare

- X86 provides a way of testing if values are greater, less than or equal to another without modifying either of the values. This is called a compare or **CMP**.
- CMP computes the difference between the operands, like a SUB instruction, but does not store the difference in a register. Instead, only the EFLAGS register is updated with new flag values.

CMP example

```
MOV eax, 10
MOV ebx, 20
CMP eax, ebx ; Compare the values in EAX and EBX
             ; eax – ebx is computed
             ; SF = 1, ZF = 0, CF = 0, OF = 0
jle happy    ; Jump to ‘happy’ if EAX <= EBX (ZF = 1 or SF != OF)
             ; branch taken
```



7E cb	JLE rel8	D	Valid	Valid	Jump short if less or equal (ZF=1 or SF≠ OF).

From https://www.felixcloutier.com/x86/jcc

JMP addressing

- Jumps can use **absolute** addressing or **relative** addressing
- An **absolute** address is a specific address
  - The value can be a label or in a register
  - Actual number is computed by the assembler
- A **relative** address is a displacement off of the value in the RIP register
  - Assembler computes address from offset

JMP addressing example

```
section .data
    absolute_data dd 42     ; Absolute data value stored in a 4-byte integer

section .text
main:
    ; Relative addressing
    mov eax, [ebp - 4]      ; Load the value stored at [ebp - 4] into eax
                            ; This is an example of relative addressing,
                            ; accessing data relative to the base pointer (ebp)

    ; Absolute addressing
    mov ebx, absolute_data  ; Load the address of absolute_data into ebx
    mov ecx, [ebx]          ; Load the value stored at the address in ebx into ecx
                            ; This is an example of absolute addressing,
                            ; accessing data directly via its memory address
```

if/else example

- Convert the following to

```
if ( x < y ) {
    columbus_sailed();
} else {
    the_ocean_blue();
}
```

```
section .data
      ;declare x and y
section .text
      MOV RAX, [x]
      MOV RBX, [y]
      CMP RAX, RBX
      JGE else
      columbus_sailed
      JMP done
else: the_ocean_blue
done:
```

Note: the “if” will run if x < y. The “else” will run if x >= y. therefore, since we want to jump to the else block, we will use the condition x >= y

While loop

- Convert the following to

```
while ( i > 0 ) {
    foo;
}
```

```
section .data
      ;declare i
section .text
loop: MOV RAX, [i]
      CMP RAX, 0
      JLE done
      foo
      JMP loop
done:
```

Note: We need an exit condition so that we can leave the loop. In the case of this program, although we have a jump instruction to done, we never reach it since “i” doesn’t change.

Loop over array

- You can declare an array of values by using commas in the .data section
- There is something wrong with this code. Use gdb to figure out what is wrong and suggest a fix
- **loop** keyword automatically decrements rcx by 1 and will jump to given address as long as rcx != 0

```
section .data
      arr dw 2,3,4,5
      len equ ($ - arr)/2
section .bss
      buffer resb 1
section .text
      global main
main: mov  rbx, arr      ; Load address of the message
      mov  ecx, len      ; load value of length
loop_print:
      ; convert from int to ascii
      mov  ax, word [rbx]
      add  al, '0'
      mov  [buffer], al

      ; setup syscall
      mov  rsi, buffer          ; address of ascii char
      mov  rdi, 1               ; File descriptor 1 (stdout)
      mov  rdx, 1               ; Length of the element
      mov  rax, 1               ; Syscall number for sys_write
      syscall                   ; Invoke syscall to write the message

      ; handle loop iteration
      add  rbx, 2
      loop loop_print

      ; exit syscall
      mov rax, 60
      mov rdi, 0
      syscall
```

Loop over array

- The rcx register gets **clobbered** (trash valued) by the write syscall
- We need to store rcx before syscall and then reintroduce it after.
- We can use **push** and **pop** to store rcx and then retrieve it later

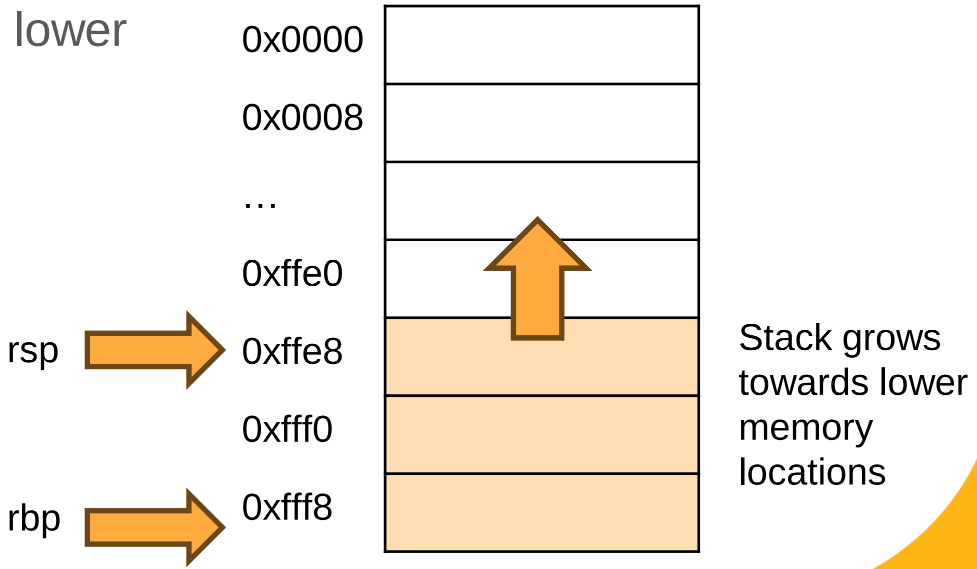
We'll see later that RAX, RCX, RDX, & R8-R11 can be modified by subroutine

```
section .data
      arr dw 2,3,4,5
      len equ ($ - arr)/2
section .bss
      buffer resb 1
section .text
      global main
main: mov  rbx, arr       ; Load address of the message
      mov  ecx, len       ; load value of length
loop_print :
      ; convert from int to ascii
      mov  ax, word [rbx]
      add  al, '0'
      mov  [buffer], al

      ; setup syscall
      push rcx
      mov  rsi, buffer            ; address of ascii char
      mov  rdi, 1                 ; File descriptor 1 (stdout)
      mov  rdx, 1                 ; Length of the element
      mov  rax, 1                 ; Syscall number for sys_write
      syscall                     ; Invoke syscall to write the message
      pop rcx

      ; handle loop iteration
      add  rbx, 2
      loop loop_print

      ; exit syscall
      mov rax, 60
      mov rdi, 0
      syscall
```

```
mov   rsi, buffer
mov   rdi, 1
mov   rdx, 1
mov   rax, 1
syscall

rcx ; ?? clobbered
```

Stack

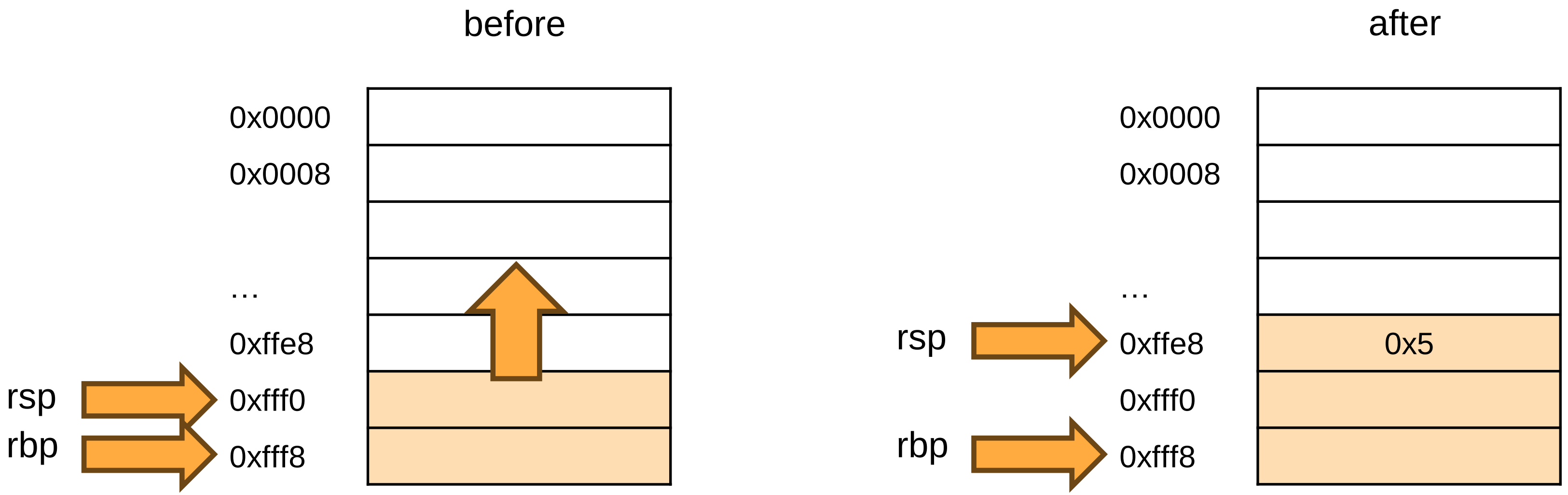
- The stack is a memory data structure to store data as needed
- Follows LIFO pattern: Last in, first out
- The top of the stack grows towards lower memory addresses
- RBP points to base of stack
- RSP points to top of stack



Push operation

- RSP gets decremented by the size of the operand
- Operand is copied into [RSP]

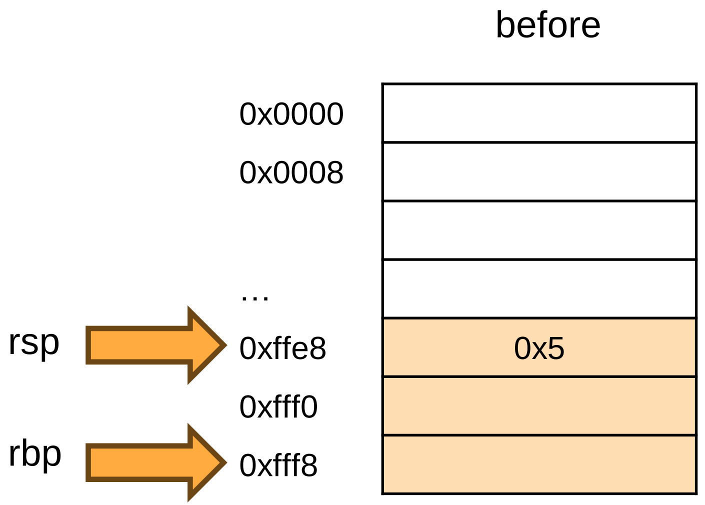
```
push rax ; rax = 0x5
```

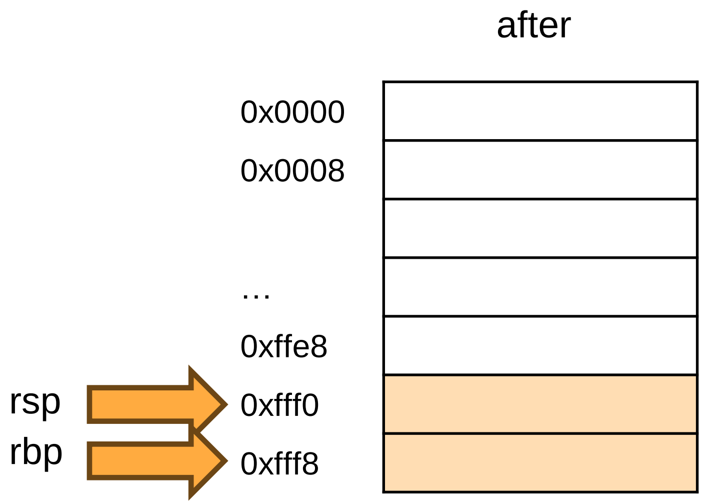


Pop operation

- Opposite of Push operation
- [RSP] is copied into operand
- RSP is incremented by the size of the operand

```
pop rax ; rax = <random>
; rax = 0x5
```

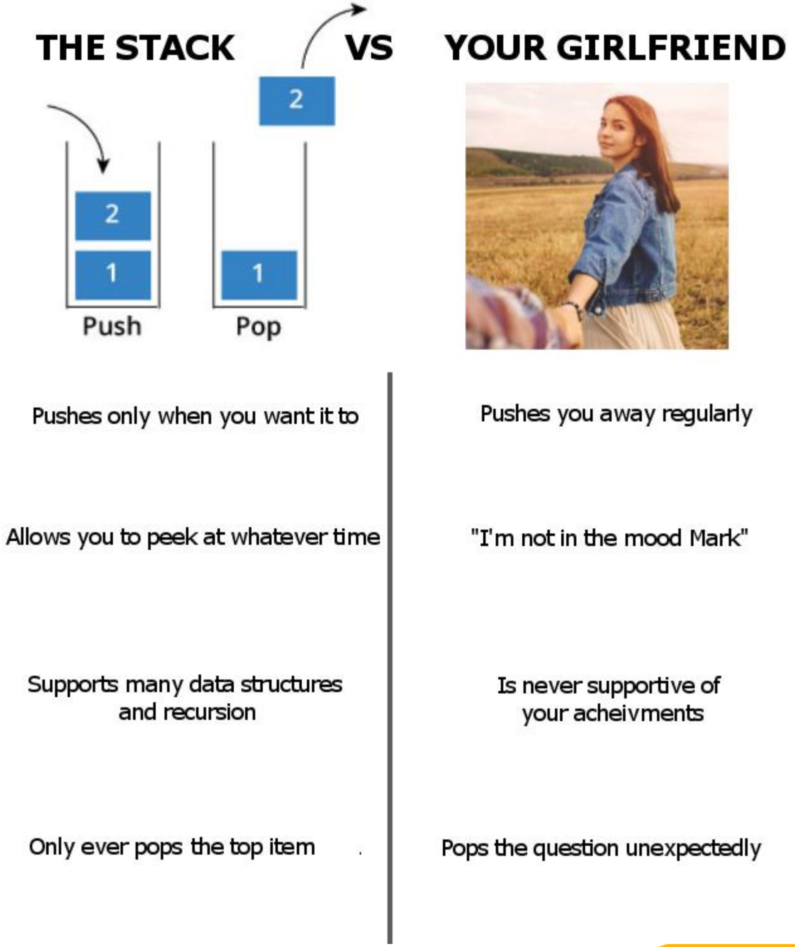




Push and Pop

- You don’t always have to push and pop values into the same register

```
push rax
; do something
pop rsi
```



THE STACK	VS	YOUR GIRLFRIEND
Pushes only when you want it to		Pushes you away regularly
Allows you to peek at whatever time		"I'm not in the mood Mark"
Supports many data structures and recursion		Is never supportive of your acheivments
Only ever pops the top item		Pops the question unexpectedly

Subroutine instructions

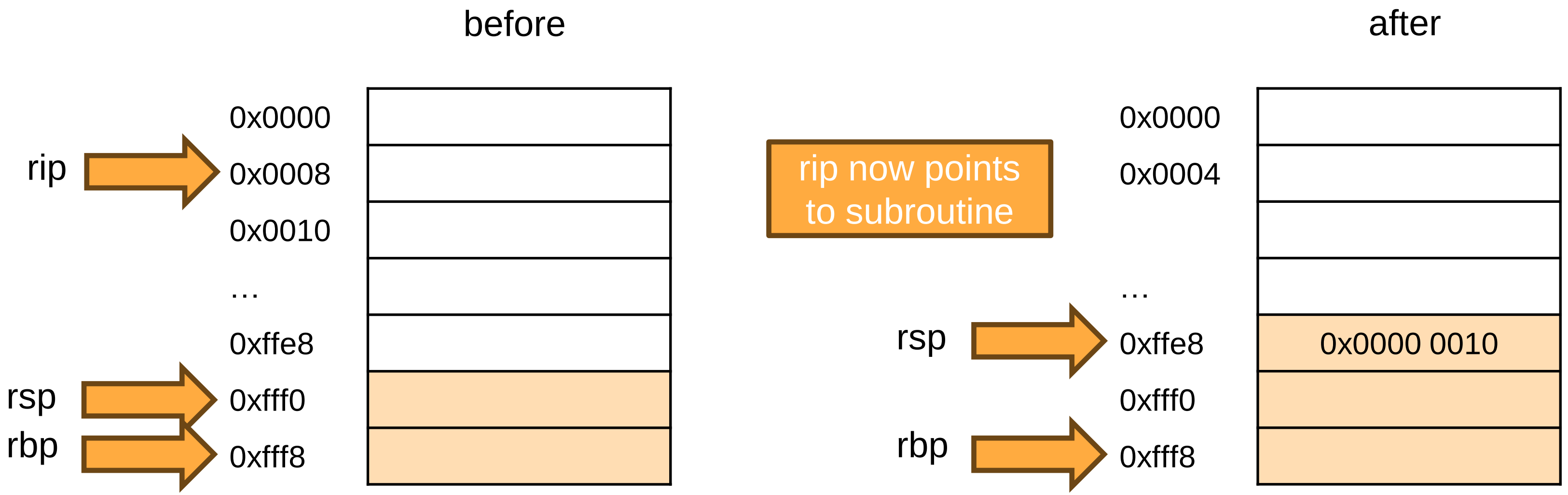
- **Subroutine** is the function equivalent in x86
- **CALL** label is used to call subroutine
  - Increments RIP
  - PUSHes RIP onto the stack
  - Jumps to the label
- **RET** returns from a subroutine
  - POPs the top of the stack into RIP
  - Execution proceeds from the location saved at RIP

A subroutine that is CALLed Always needs a RET!

```
foo: ; this is just a label
     mov rax, 5

bar: ; this is a subroutine
     mov rax, 5
     ret
```

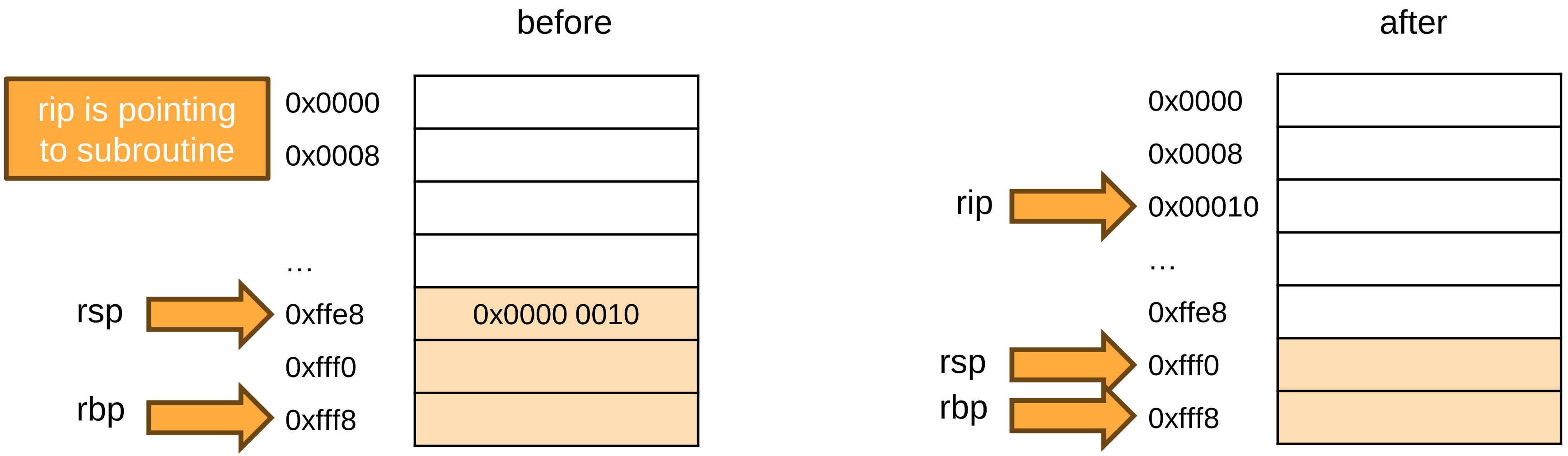
Subroutine stack during CALL

- Instruction after rip is saved on stack during a subroutine call
- Addresses are 64bit, so 8 bytes will be pushed on the stack



Subroutine stack during ret

- After RET, execution continues with where the head of the stack is pointing and rsp is incremented by the address size (64 bits)



Subroutines with Parameters and Return

- Functions can have parameters:

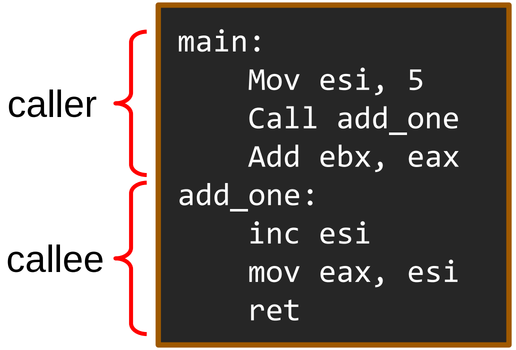
```
int func(int a, int b){

}
```

- In Assembly, CALL does not specify parameters and RET does not specify return value. The developer must handle allocating parameters and return value to either registers or the stack.
- The developer also must handle not overwriting registers during a function call.

Calling convention

- The **Caller** is the part of code that calls a subroutine.
- The **Callee** is the function that gets called
- The **calling convention** is a set of rules governing use of subroutines and which parts the caller is responsible for and which parts the callee is responsible for. These can include which registers are overwritten by whom, who sets up and takes down the stack, how parameters are setup, and how return values work.

Example:



```
main:
    Mov esi, 5
    Call add_one
    Add ebx, eax
add_one:
    inc esi
    mov eax, esi
    ret
```

Calling convention – saved regs

- **Caller-saved** registers must be saved by the caller before a subroutine call if they will be overwritten. The caller should expect those registers to be clobbered
- A caller can expect that **callee-saved** registers will not change after a subroutine returns. If a subroutine uses them, the previous value must be saved before that register is used, and then reset to the old value when the subroutine is ready to return.

Caller saved	Callee saved
RAX, RCX, RDX	RBX, RBP, RDI, RSI, RSP
R8, R9, R10, R11	R12, R13, R14, R15

Parameters and Return value

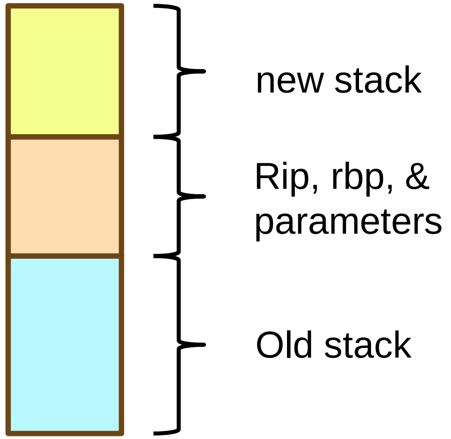
- The return value is always returned in the RAX register
- Parameters can be stored in 2 ways: the 32 bit method or the 64 bit method.

x86 user subroutine, x86-32 c declaration	**x86-64 c declaration**
The 32 bit method uses the stack to store parameters. The caller pushes values on to the stack, in reverse order, and the callee can access them by looking at values “below” the base pointer	The 64 bit method places the parameters 1-6 in the registers RDI, RSI, RDX, RCX, R8, R9, and all parameters after that get pushed to the stack.

Stack maintenance

- When a subroutine is called, a new stack is created for local variables to be created and only exist during the runtime of that stack.
- To create a new stack, the RBP and RSP must be updated.
- The following are done by the callee:
  - Push old RBP on to the stack
  - RBP is set to RSP

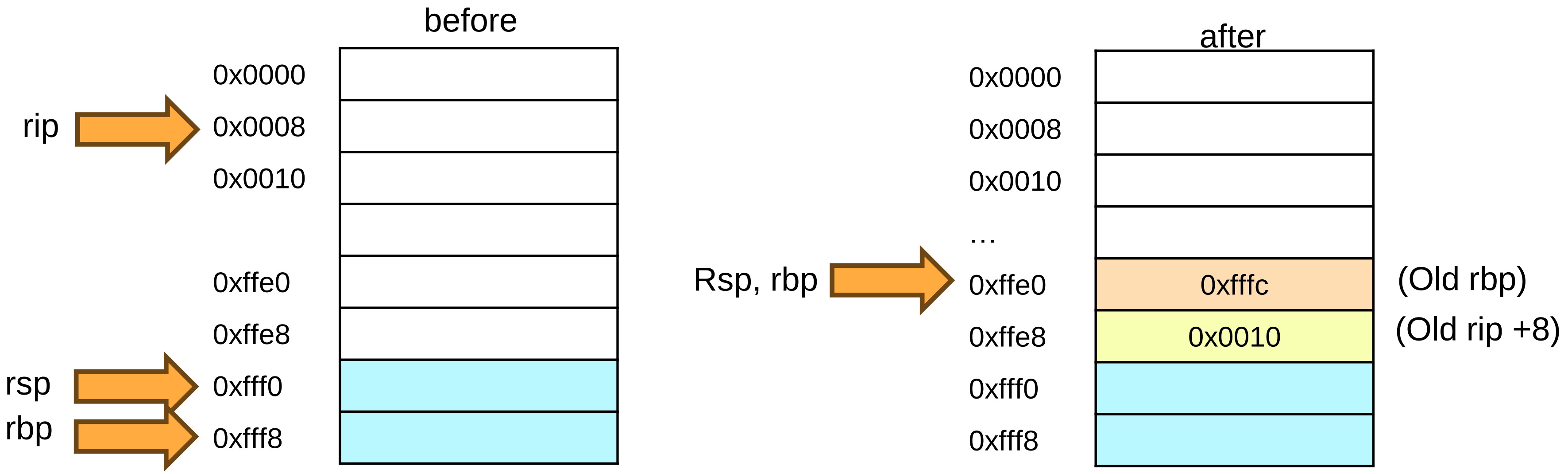
```
subroutine:
    push rbp     ; keep track of old rbp
    mov rbp, rsp ; update rbp to new stack base

    pop rbp      ; get back old rbp
    ret
```



Stack setup

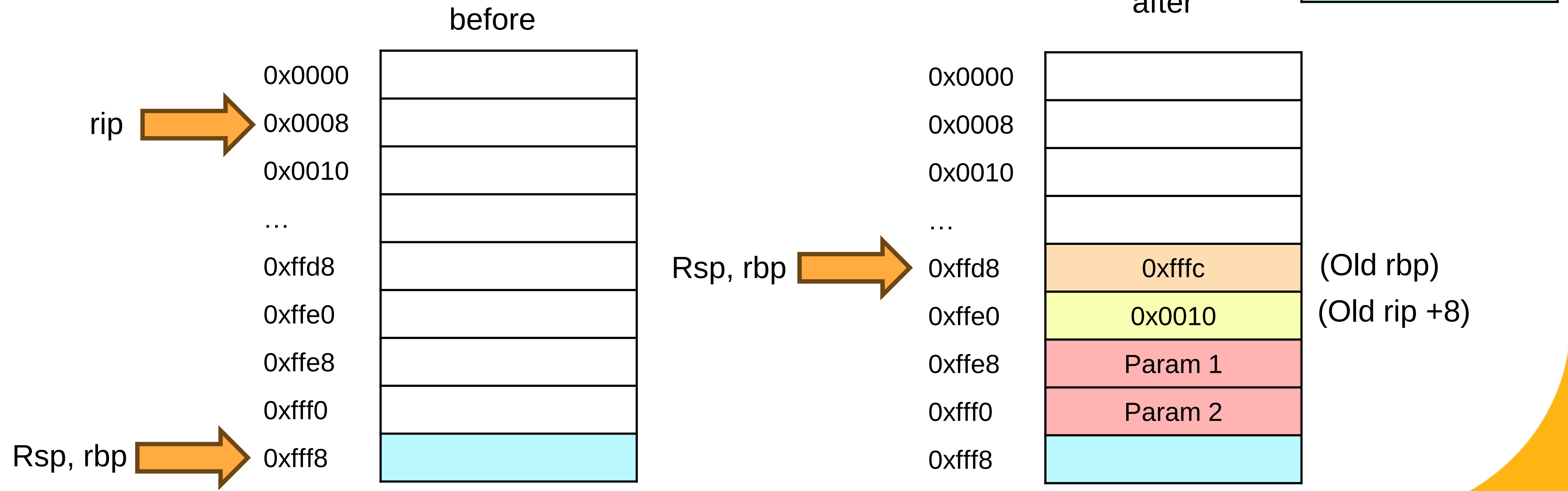
key

Done by callee
Automatic by “call”
Old stack



Stack setup with pushed parameters

- Let's say there are 2 parameters using the 32-bit parameter method

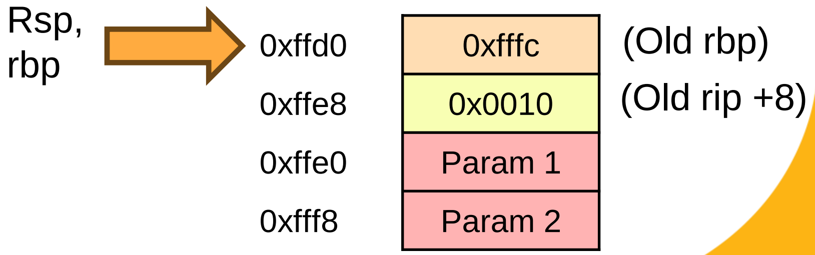


Passing and accessing parameters (pushed method)

- Although this convention is normally used on 32 bit, this strategy also works on 64 bit.

```
main:
    push rax            ;pass 2nd parameter
    push rbx            ;pass 1st parameter
    call subroutine
    pop rbx             ;clean up stack
    pop rax             ;clean up stack

subroutine:
    push rbp            ;save base pointer
    mov rbp, rsp        ;setup new base pointer

    mov rax, [rbp+16]   ;access 1st parameter
    mov rcx, [rbp+24]   ;access 2nd parameter

    pop rbp             ; restore old base pointer
    ret
```

Note: when you want to access parameters, you can offset from the base pointer:
RBP+16 (= RBP+0x10) will get the first parameter (skipping over stored RIP and RBP)



References

- Ivan Sekyonda’s slides
- https://en.wikipedia.org/wiki/X86_calling_conventions