CMPT 295

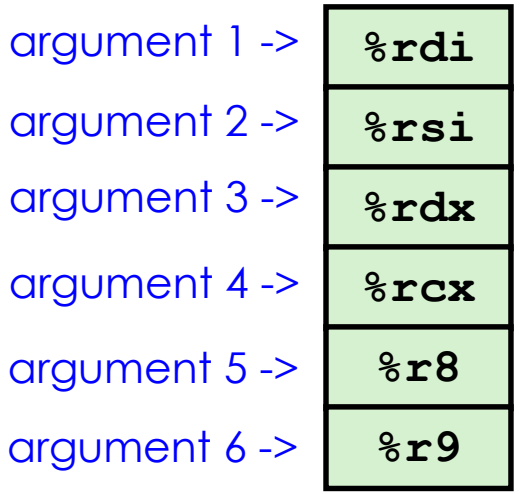
Unit - Machine-Level Programming

Lecture 19 – Assembly language – Program Control – Function Call and Stack – Managing Local Data

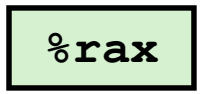
Last lecture

- Passing data mechanism
  - x86-64 function call convention:

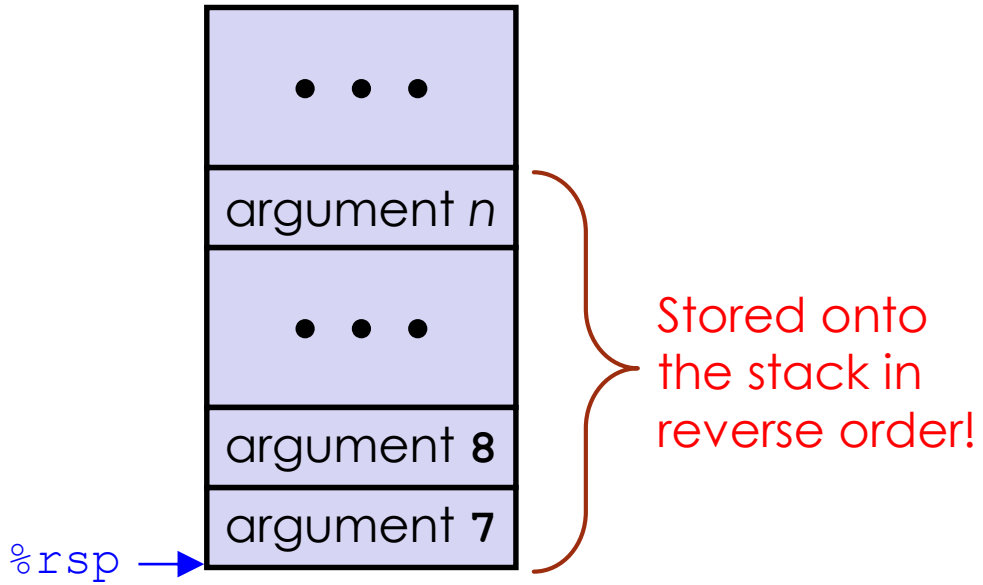
First 6 arguments



return value



Stack



Today's Menu

- Introduction
  - C program -> assembly code -> machine level code
- Assembly language basics: data, `move` operation
  - Memory addressing modes
- Operation `leaq` and Arithmetic & logical operations
- Conditional Statement – Condition Code + `cmovX`
- Loops
- Function call – Stack
  - Overview of Function Call
  - Memory Layout and Stack - x86-64 instructions and registers
  - Passing control
  - Passing data – Calling Conventions
  - Managing local data
  - Recursion
- Array
- Buffer Overflow
- Floating-point operations

To recap …

- Overview of ***Function Call*** mechanisms:
  - What happens when a function (***caller***) calls another function (***callee***)?

Last lecture:

1. **Control** is passed …
   - To the beginning of the code in ***callee*** function
   - Back to where ***callee*** function was called in ***caller*** function
2. **Data** is passed …
   - To ***callee*** function via *function parameter(s)*
   - Back to ***caller*** function via *return value*
3. **Memory** is …
   - Allocated when ***callee*** function starts executing
   - Deallocated when ***callee*** function stops executing

… allocated a stack frame on the stack, but what can be stored on this stack frame?

3. Managing local data

- When writing assembly programs, what can we use when we need space for our local data?
  - **We can use registers!**
    - Yes! Registers are our **first choice** as they are the fastest storage location on a computer.
  - OK! but, since **registers are shared by all functions** in x86-64 assembly language, we need to follow some convention, otherwise … :

```
who:
    • • •
    movq $15213, %rbx
    call amI
    addq %rbx, %rax
    • • •
    ret
```

```
amI:
    • • •
    subq $18213, %rbx
    • • •
    ret
```

Register Table	
%rbx	

3. Managing local data - "register saving" convention => *callee saved* registers

- When we need space for our local data …

1. **Registers**
   - A function can utilise unused registers (only when needed)
   - Some registers are referred to as ***callee saved* registers**:
     - **`%rbx`**, **`%rbp`**, **`%r12`** to **`%r15`** (and **`%ebx`**, **`%bx`**, **`%bl`**, …)
   - ***Callee saved* registers** means that …
     - the **callee** function must preserve the values of these registers before using them,
     - then restore their values before the control is returned (through the execution of **`ret`** instruction) to the **caller** function

"register saving" convention:

1) ***callee saved* registers**

3. Managing local data - "register saving" convention => *callee saved* registers

- How can **callee** preserve the values of these ***callee saved*** **registers** before using them?
  - Example of a scenario:
    - **Caller** uses `%r13`
    - **Caller** calls **callee**
    - At the start of **callee**, **callee** `pushq %r13`
    - Then **callee** uses `%r13`
    - Then before execution flow returns from **callee** to **caller** (via `ret`), **callee** `popq %r13`
    - The execution flow returns to **caller** which continues using `%r13`

If **callee** `pushq` more than 1 register, then **callee** `popq` them in reverse order

callee saved registers
Upon return from **callee**, **caller** can always assume that these registers still contain the values **caller** stored in them before calling **callee**!

3. Managing local data - "register saving" convention => *caller saved* registers

1. Registers (cont'd)

- Some registers are referred to as ***caller saved* registers**:
  - **`%r10`**, **`%r11`**, **`%rax`** and all 6 registers used for passing data as arguments to **callee** (and **`%r10d`**, **`%r10w`**, **`%r10b`**, …)
- ***Caller saved* registers** means that …
  - the **caller** function must preserve the values of these registers before …
    - setting up the **callee**'s argument(s) into the appropriate "data passing as argument" register(s) and
    - calling the **callee**
  - then once the control is returned to the **caller**, the **caller** must restore their values before using them

"register saving" convention:

2) *caller saved* registers

3. Managing local data - "register saving" convention => *caller saved* registers

- How can **caller** preserve the values of these ***caller saved* registers** before using them?
  - Example of a scenario:
    - **Caller** uses `%r10`
    - Before calling **callee**, **caller** `pushq %r10` then calls **callee**
    - **Callee** uses `%r10`
    - Then after the execution flow has returned from **callee** to **caller** (via `ret`), **caller** `popq %r10`
    - **Caller** continues using `%r10`

If **caller** `pushq` more than 1 register, then **caller** `popq` them in reverse order

***caller saved* registers**
Callee can always assume that **caller** has saved the content of these registers, so it is "safe" for **callee** to use them!

x86-64 "register saving" convention

- Solution 1:

```
who:
    • • •
    movq $15213, %rbx
    call amI
    addq %rbx, %rax
    • • •
    ret
```

```
amI:


    subq $18213, %rbx

    ret
```

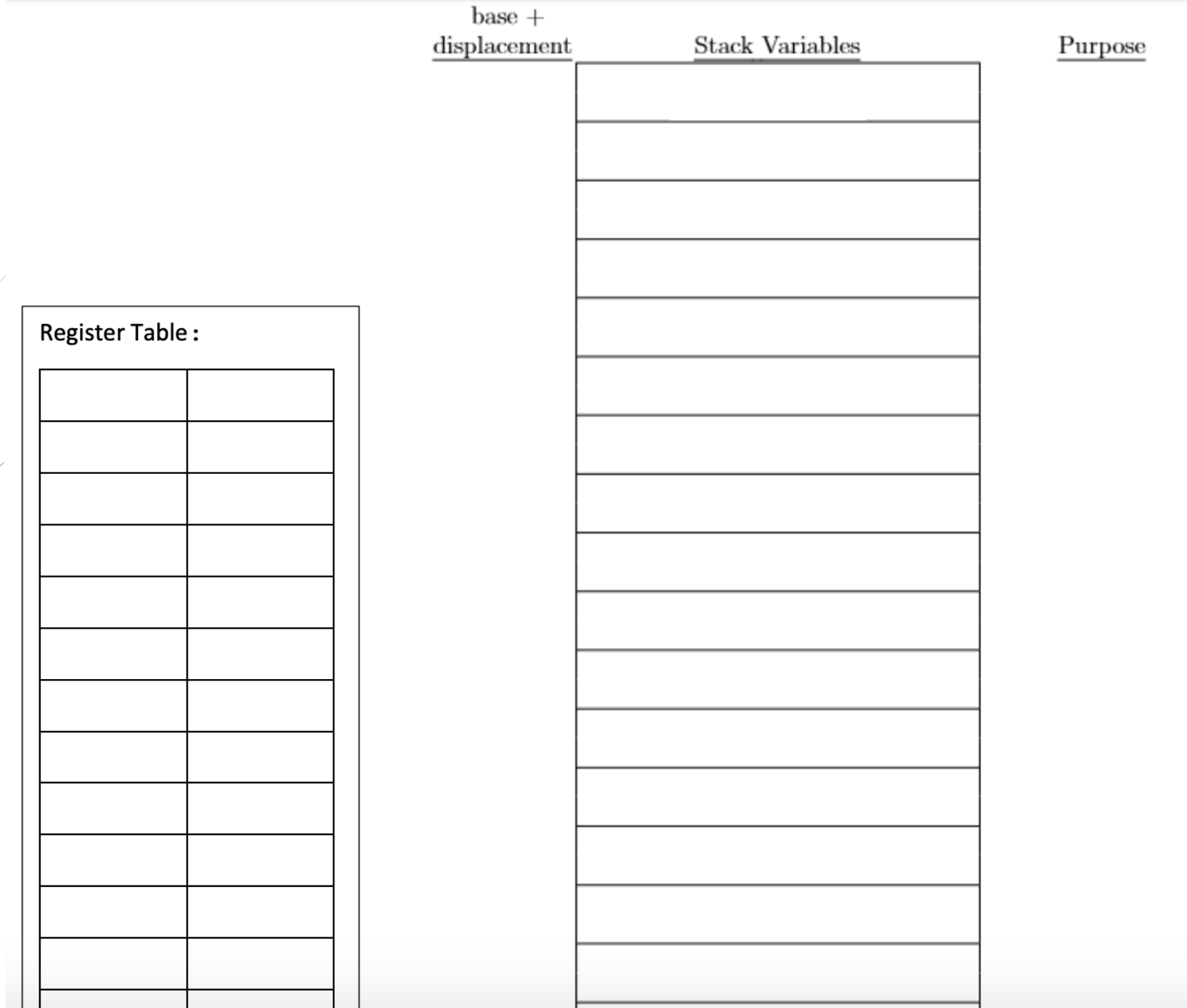
- Solution 2:

```
who:
    • • •
    movq $15213, %r10

    call amI

    addq %r10, %rax
    • • •
    ret
```

```
amI:
    • • •
    subq $18213, %r10
    • • •
    ret
```


base +
displacement
Stack Variables
Purpose
Register Table :

3. Managing local data => spilling

- When writing assembly programs, what can we use when we need space for our local data?
  - **We can use stack!**

If we run out of registers!

2. Stack
   - A function can use the stack to store the values of its local variables and for temporary space
   - *Set-up* and *Clean-up* code:
     - Example: `subq $16, %rsp` and `addq $16, %rsp`
   - To spill onto the stack:
     - Example: `movq %rax, 56(%rsp)`

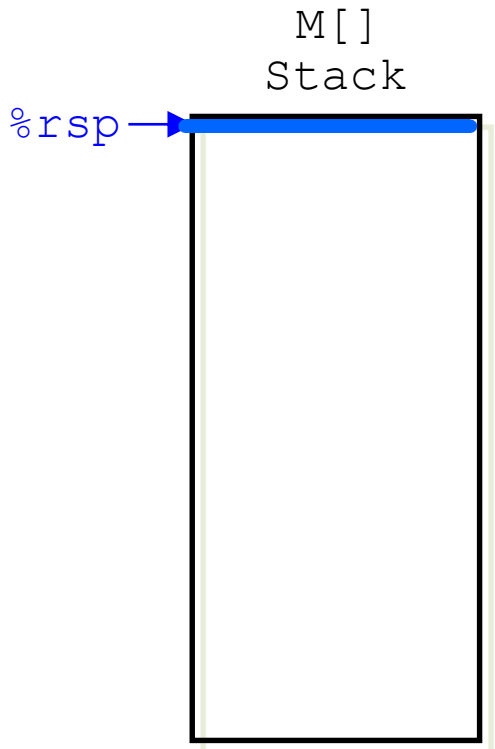
Must remember to clean-up the stack before returning to **caller**!

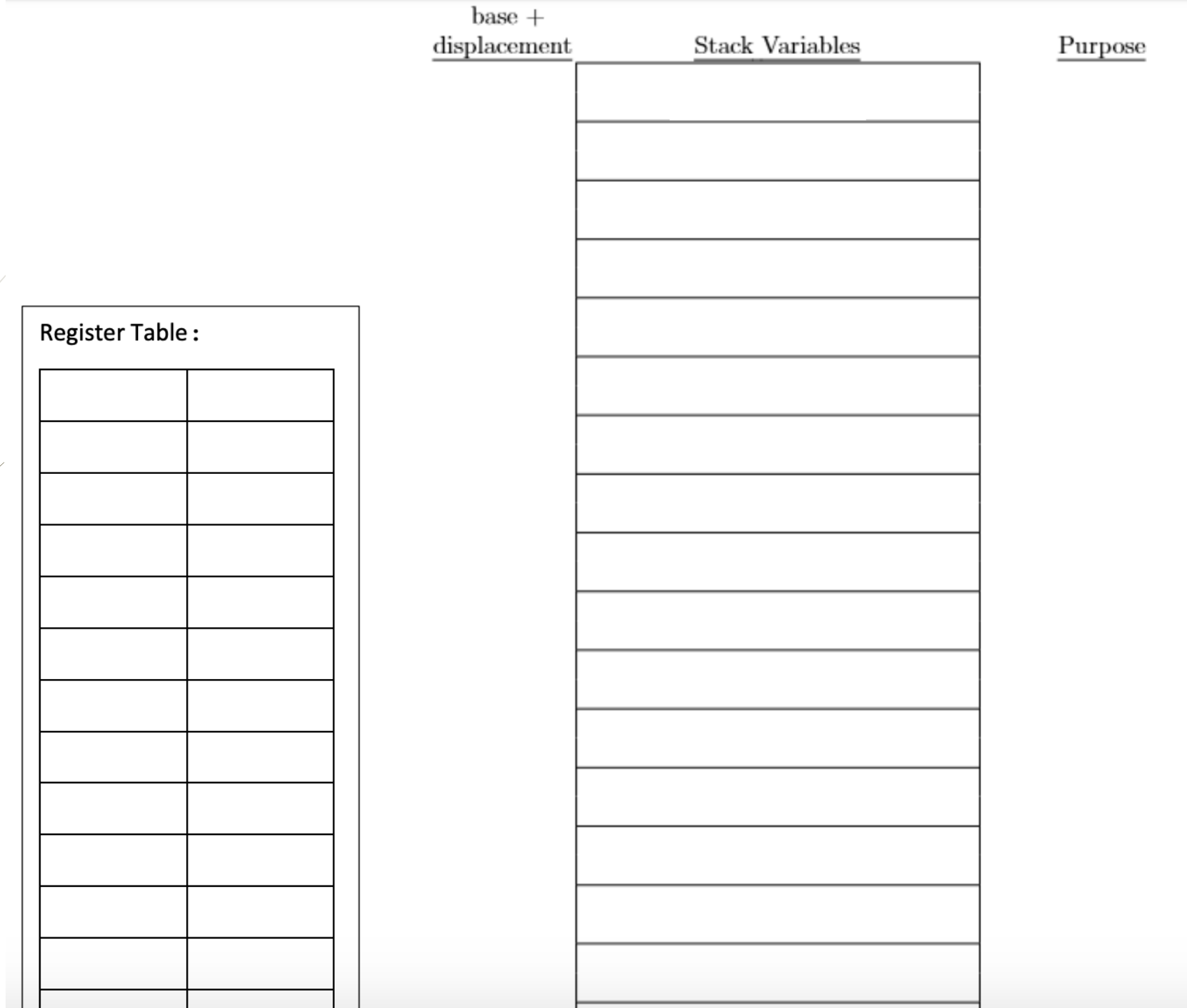
Local variables on Stack – Example

```
long incr(long *p, long val)
{
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

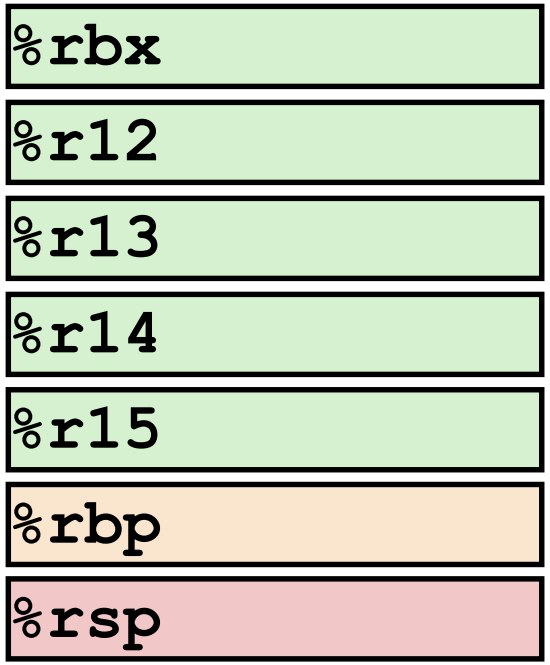



base +
displacement
Stack Variables
Purpose
Register Table :

Summary - x86-64 "register saving" convention

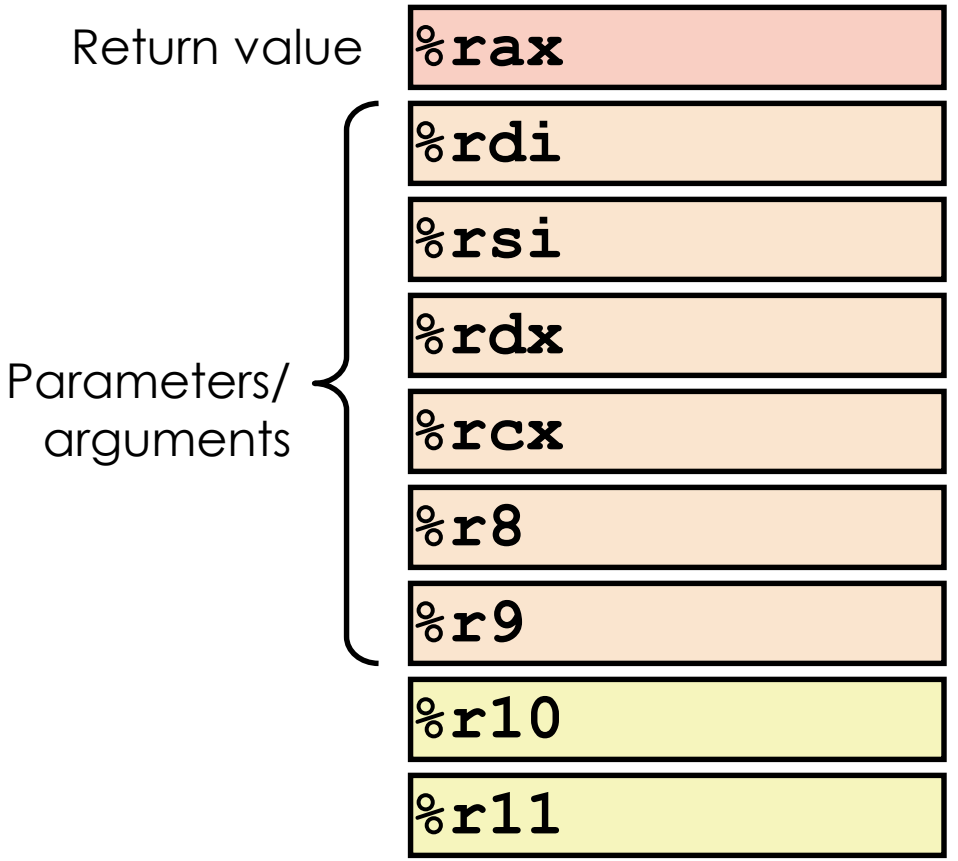
callee saved registers:

- **Callee** must save & restore before modifying



caller saved registers:

- **Caller** must save & restore
- Can be modified by **callee**

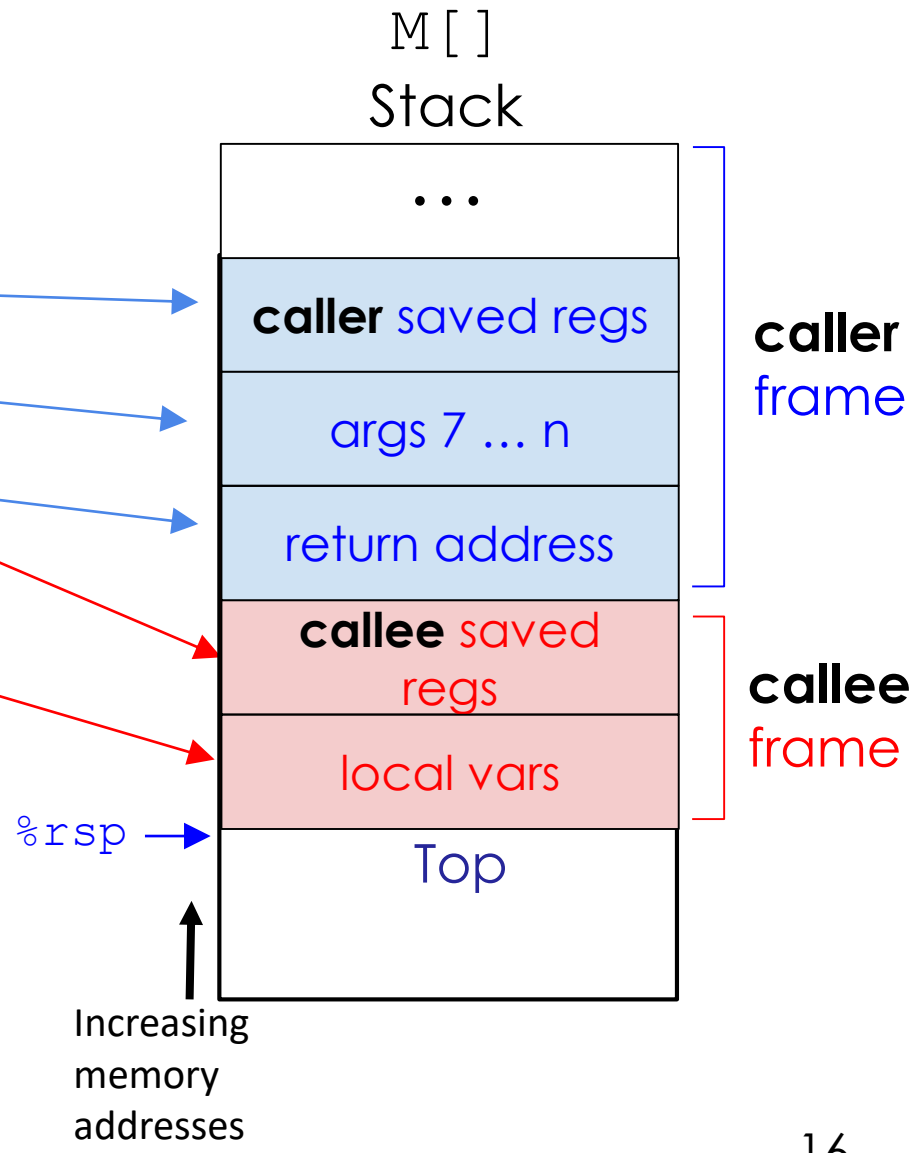


Summary - x86-64 conventions and stack frame

caller preserves ***caller saved* registers**
caller passes arguments
caller calls **callee**

callee preserves ***callee saved* registers**
callee constructs local vars (get stack space)
callee performs function
callee recycles local vars (restore stack space)
callee restores ***callee saved* registers**
callee returns to **caller**

caller pops arguments
caller restores ***caller saved* registers**



16

Next lecture

- Introduction
  - C program -> assembly code -> machine level code
- Assembly language basics: data, `move` operation
  - Memory addressing modes
- Operation `leaq` and Arithmetic & logical operations
- Conditional Statement – Condition Code + `cmovX`
- Loops
- Function call – Stack
  - Overview of Function Call
  - Memory Layout and Stack - x86-64 instructions and registers
  - Passing control
  - Passing data – Calling Conventions
  - Managing local data
  - Recursion
- Array
- Buffer Overflow
- Floating-point operations