Allocating Local Variables
Saving Callee-Saved Registers

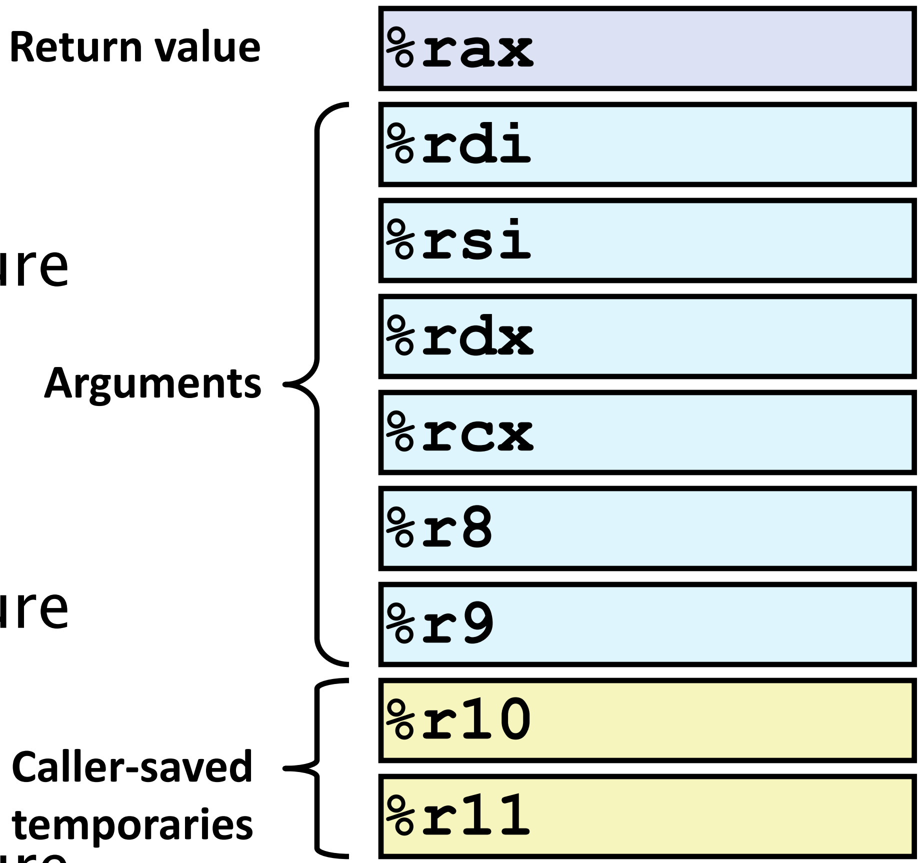
In assembler

Quick Review – Register Usage

- Basic Categories
  - Caller-saved
  - Callee-saved
- Other Categories / Purposes
  - Return value
  - Parameter
  - Stack frame

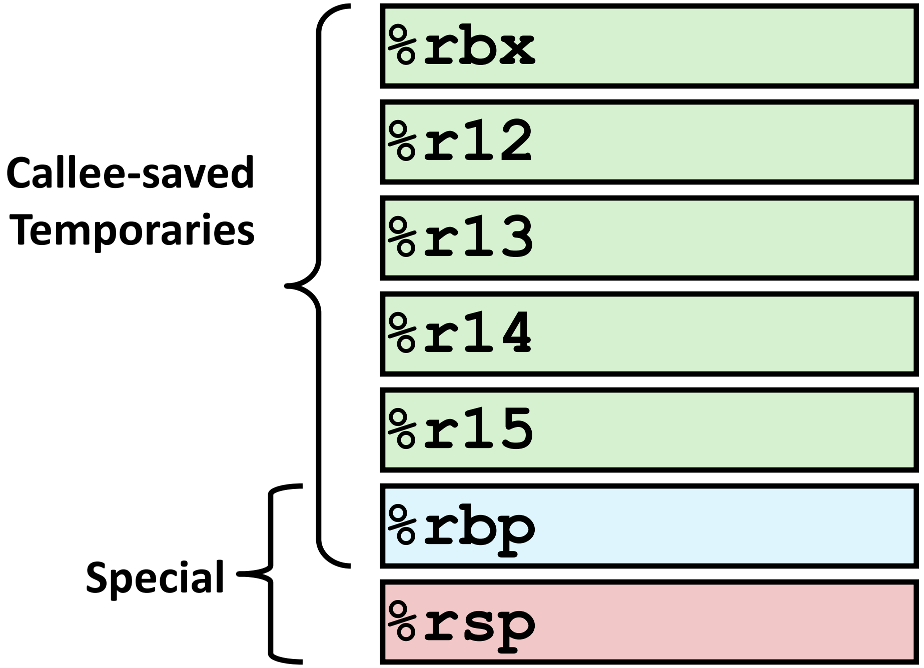
x86-64 Linux Register Usage #1

- `%rax`
  - Return value
  - Also caller-saved
  - Can be modified by called procedure
- `%rdi, ..., %r9`
  - Arguments
  - Also caller-saved
  - Can be modified by called procedure
- `%r10, %r11`
  - Caller-saved
  - Can be modified by called procedure



x86-64 Linux Register Usage #2

- `%rbx`, `%r12`, `%r13`, `%r14` , `%r15`
  - Callee-saved
  - Callee must save & restore
- `%rbp`
  - Callee-saved
  - Callee must save & restore
  - May be used as frame pointer
  - Can mix & match
- `%rsp`
  - Special form of callee save
  - Restored to original value upon exit from procedure



How do we save registers and allocate space?

- In C, automatic storage local variables are supposed to wind up on the stack
- In assembler, callee-saved registers are more efficient to use in functions that call other functions
- Let us take a look at how that is done in assembler and what the stack winds up looking like

Local variables on the stack

```
Func()
{
    long array[5];

/* other code deals with
x and i*/

    x += array[i];
/* end of function code
not shown */
```

C code

```
# do stack frame stuff here-not shown
subq     $40, %rsp     #allocate array
movq     %rsp, %rdi    #save the pointer

# other code goes here. i is in %rsi
# x is in %rax

addq (%rdi, %rsi, 8), %rax  #add array[i] to x
# longs are 8 bytes, so we scale i by 8
```

Assembler

Getting to stack variables

- Allocate space *after* setting up the stack frame
- We can subtract from %rsp to allocate space on the stack – this grows the stack
- After allocating, we can save %rsp to another register to have a pointer to the space
- We can also do math based on %rbp to find things we put on the bottom of the stack frame

Callee saved registers in the stack

- We generally take care of rbp first thing
- After allocating locals, we can push any other callee-saved registers we want to use
- The other callee-saved registers are rbx and r12-r15
- It is more efficient to use callee-saved registers in functions that call other functions

An example stack

Current function is marked in green

Address	Contents	Commentary
Higher Addresses	stuff that belongs to the calling function	
	stack-based parameters in reverse order	Used for parameters that are too large to fit in a register or when there are more than 6 parameters
	return address	rsp points here when this function starts
	old rbp	This function pushed old rbp and set **rbp** to point here
	local variables	Allocate locals - subtract from rsp
	callee saved registers	This function pushed these early on in the code
	Stack-based parameters	Then it used the stack in other ways, such as getting ready to call a function
	that this function	that has parameters in the stack
Lower	put on the stack for	
Addresses	a call it is about to make	**rsp** points here

Pay attention to what goes where (1)

- In the previous slide the tan part of the stack is not part of the current function's stack frame
- The white background area is not in the current stack frame, but the current function knows that it can find any stack-based parameters there
- The return address is needed by the ret instruction to resume execution in the calling function. The stack pointer starts here when the current function begins execution.
- Typically, the next thing on the stack is the old base pointer. The current base pointer is set to this address.

Pay attention to what goes where (2)

- Next on the stack are local variables, if any. The example array of 5 longs given earlier goes here
- If any callee-saved registers are going to be saved, they go next into the stack
- After that comes any arbitrary uses of the stack, such as getting ready to make a function call that has stack-based parameters
- The stack pointer always hold the lowest address, which is where the last thing put on the stack is stored
- Note that a push decrements the stack pointer first and then stores data at the new address