University of Illinois at Urbana-Champaign
Dept. of Electrical and Computer Engineering

ECE 220: Computer Systems & Programming

Stack Frames Revisited

Recall Why ISAs Define Calling Conventions

A compiler must **systematically transform function calls into assembly instructions**.

Why systematically?

1. The compiler is a computer program: that's all it can do!
2. **Code generated by different compilers should interoperate**, so those compilers must make the same choices for subroutine call interfaces.

Recall the LC-3 Calling Convention

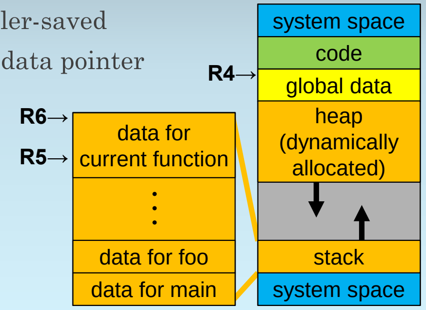
R0-**R3**: caller-saved

R4: global data pointer

R5: frame pointer

R6: stack pointer

R7: return address



R6→
R5→
data for current function
⋮
data for foo
data for main

system space
code
R4→
global data
heap (dynamically allocated)
stack
system space

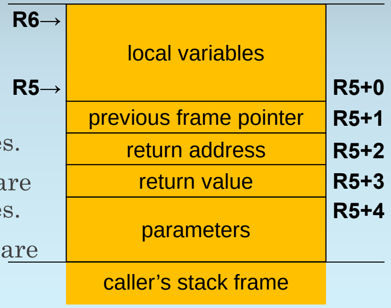
Recall the Structure of the LC-3 Stack Frame

R6 points to top of stack.

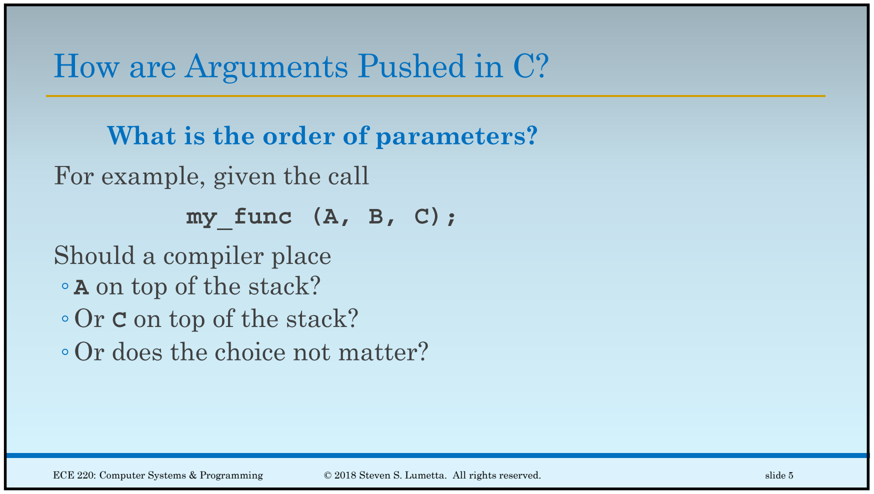
R5 points to bottom of local variables.

R5+0, -1, … are local variables.

R5+4, +5, … are parameters.



R6→ local variables
R5→ R5+0
previous frame pointer R5+1
return address R5+2
return value R5+3
parameters R5+4
caller's stack frame



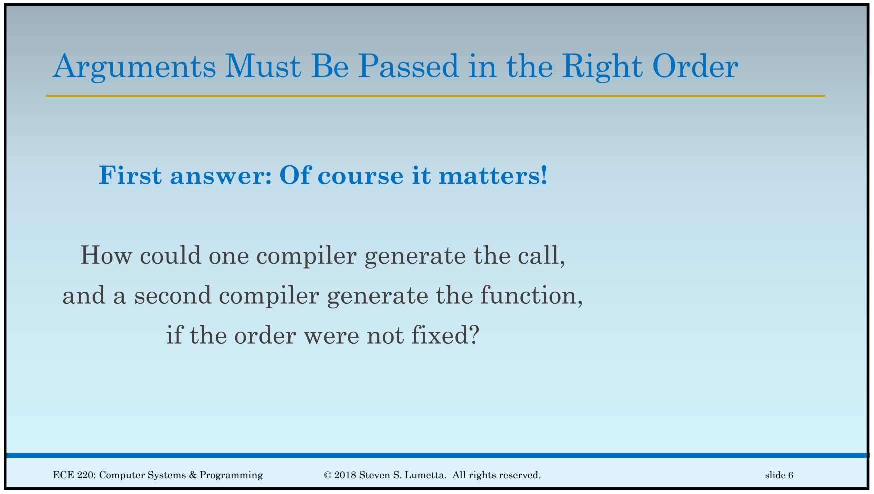
How are Arguments Pushed in C?

What is the order of parameters?

For example, given the call

```
my_func (A, B, C);
```

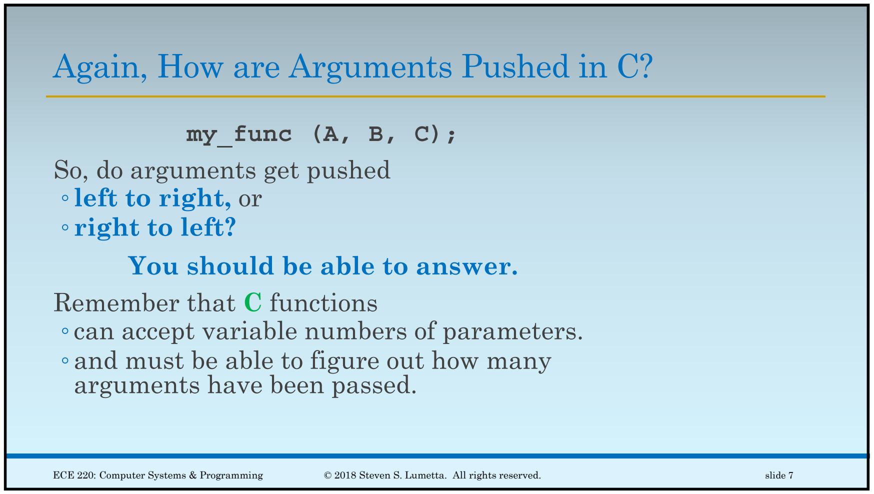
Should a compiler place
- `A` on top of the stack?
- Or `C` on top of the stack?
- Or does the choice not matter?



Arguments Must Be Passed in the Right Order

First answer: Of course it matters!

How could one compiler generate the call,
and a second compiler generate the function,
if the order were not fixed?



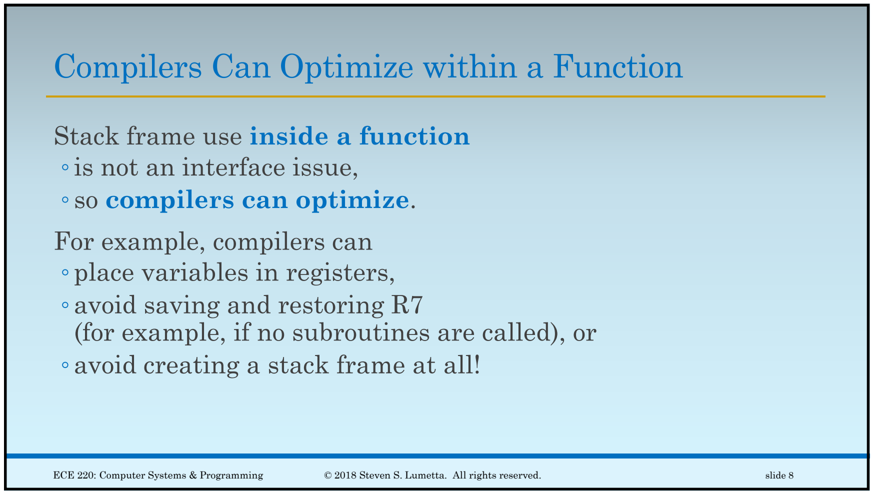
Again, How are Arguments Pushed in C?

```
my_func (A, B, C);
```

So, do arguments get pushed
- **left to right,** or
- **right to left?**

You should be able to answer.

Remember that **C** functions
- can accept variable numbers of parameters.
- and must be able to figure out how many arguments have been passed.



Compilers Can Optimize within a Function

Stack frame use **inside a function**
- is not an interface issue,
- so **compilers can optimize**.

For example, compilers can
- place variables in registers,
- avoid saving and restoring R7 (for example, if no subroutines are called), or
- avoid creating a stack frame at all!

Is R5 – R6 a Constant Inside a Function?

One common question:
- **why use both R5 and R6?***
- (Aren't R5 and R6 always the same distance apart?)

One answer:
- code adds/removes values from the stack
- (so, no, the **difference is not constant**).

*Note that the x86 (IA-32) ISA calling convention also uses two registers.

Compiler Does Know R5 – R6 Most of the Time

What kinds of things are pushed?
- callee-saved registers
- arguments to subroutines
- spilled values (when compiler runs out of registers for performing calculations)
- certain types of temporary allocation (not covered in our class—see `alloca`).

But—except for the last case—**the compiler KNOWS when R6 moves**, so it could still generate the right code…

Compiler Often Does Not Provide Such Information

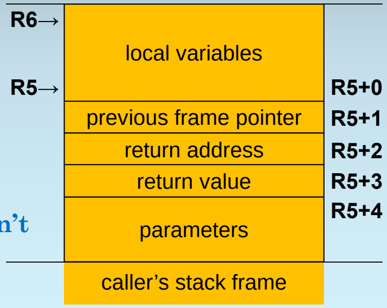
However, information about **R6's movement is often not passed to a debugger**.

So …
- you can turn on high levels of optimization
- and compilers (x86 compilers, for example) will reclaim the frame pointer,
- but good luck trying to debug (debugger will not be able to identify stack frames).

What is the Order of Local Variables?

What about the order of local variables?

Used only within the function, so choice doesn't matter.



Draw Stack Frames for a Program with Several Functions

Let’s **draw the stack frames for our prime number printing example**.

Here was our **main** function:

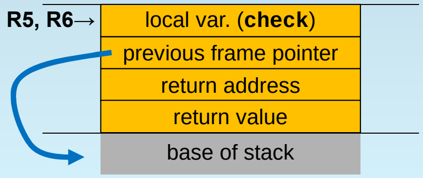
no parameters

```
int main ()
{
    int32_t check;
    // ... code doesn’t matter to us
}
```

one local variable

Stack Frame for **main** (During Execution of Code)

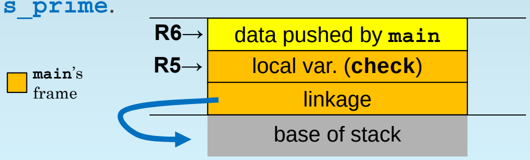
OS usually has data below **main**’s stack frame, but **from the program’s point of view, main’s stack frame starts at the base**.



Stack Frame for **main** (Just Before Call to **is_prime**)

Let’s
- collapse the linkage into one block, and
- add space for saved values.

Here’s what we might have **before calling is_prime**.



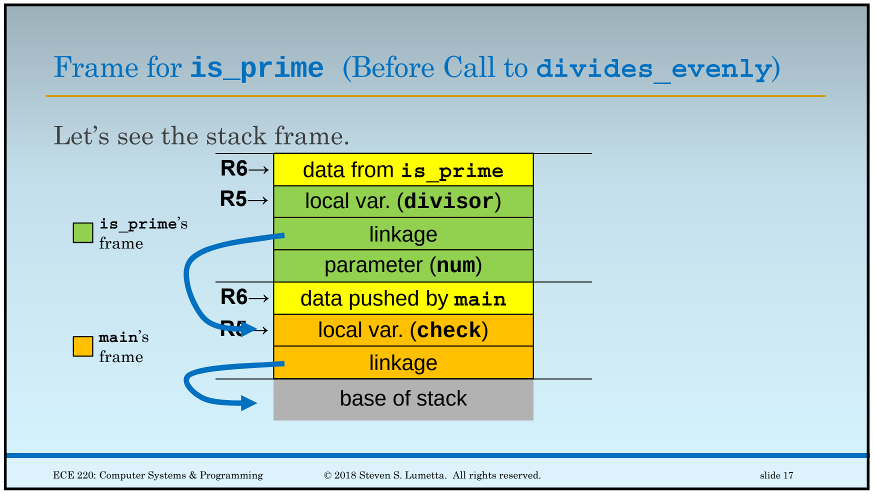
Our **is_prime** Function for Checking Primality

main calls **is_prime**:

one parameter

```
int32_t is_prime (int32_t num)
{
    int32_t divisor;
    // ... code doesn’t matter to us
}
```

one local variable



Frame for `is_prime` (Before Call to `divides_evenly`)

Let's see the stack frame.

R6→ data from `is_prime`

R5→ local var. (`divisor`)

linkage

parameter (`num`)

R6→ data pushed by `main`

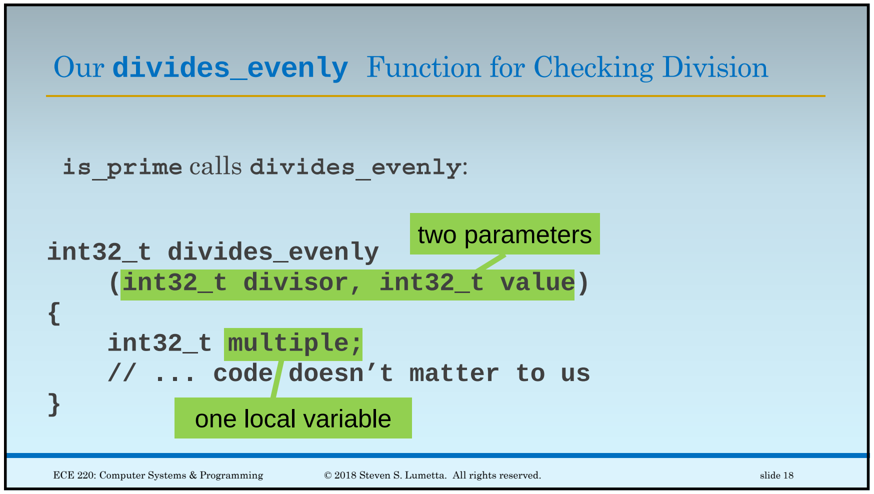
R5→ local var. (`check`)

linkage

base of stack

🟩 `is_prime`'s frame

🟧 `main`'s frame



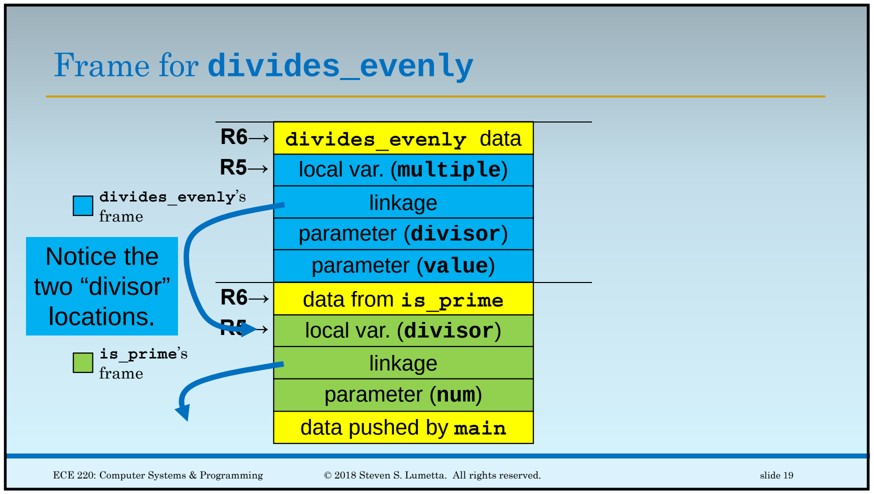
Our `divides_evenly` Function for Checking Division

`is_prime` calls `divides_evenly`:

```
int32_t divides_evenly
    (int32_t divisor, int32_t value)
{
    int32_t multiple;
    // ... code doesn't matter to us
}
```

two parameters

one local variable



Frame for `divides_evenly`

R6→ `divides_evenly` data

R5→ local var. (`multiple`)

linkage

parameter (`divisor`)

parameter (`value`)

R6→ data from `is_prime`

R5→ local var. (`divisor`)

linkage

parameter (`num`)

data pushed by `main`

🟦 `divides_evenly`'s frame

🟩 `is_prime`'s frame

Notice the two "divisor" locations.