Program Representation

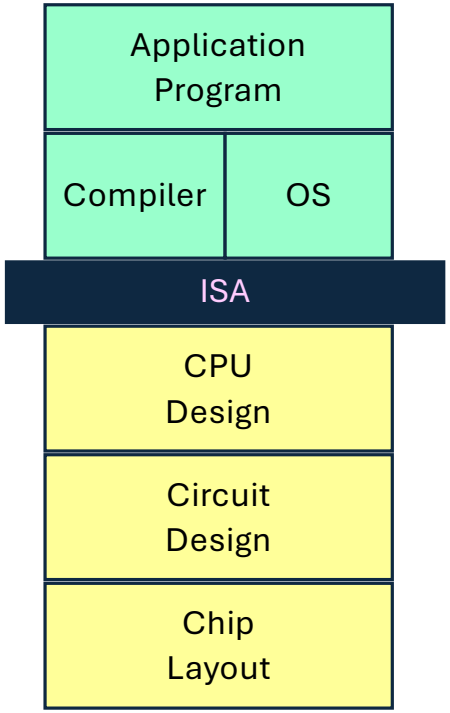
Philippe Bonnet, bonnet@di.ku.dk
HPPS 2025 – 3a

Outline

- Modern processors
- Assembly
- Compilers & Interpreters
- Experiments

Instruction Set Architecture

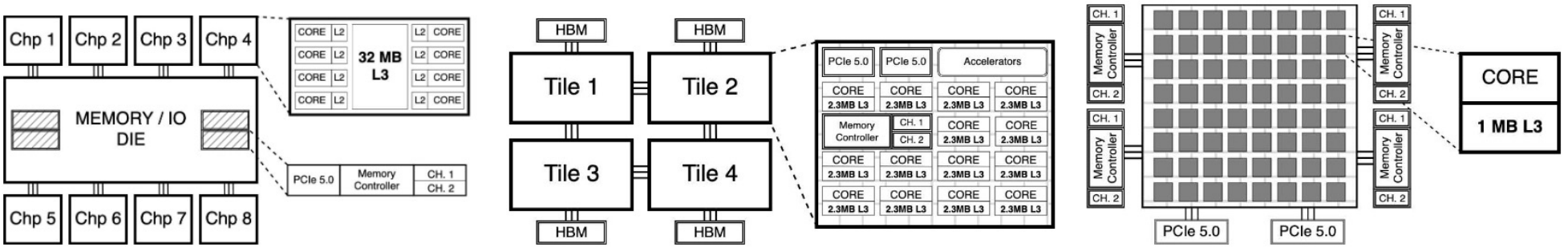
- Assembly Language View
  - Processor state
    - Registers, memory, …
  - Instructions
    - `addq`, `pushq`, `ret`, …
    - How instructions are encoded as bytes
- Layer of Abstraction
  - Above: how to program machine
    - Processor executes instructions in a sequence
  - Below: what needs to be built
    - Use variety of tricks to make it run fast
    - E.g., execute multiple instructions simultaneously



https://www.tele-task.de/lecture/video/8963/

Modern processors

Chiplet-based architecture



(a) AMD EPYC Milan (b) Intel Sapphire Rapids (c) ARM Graviton 3

Parallelism

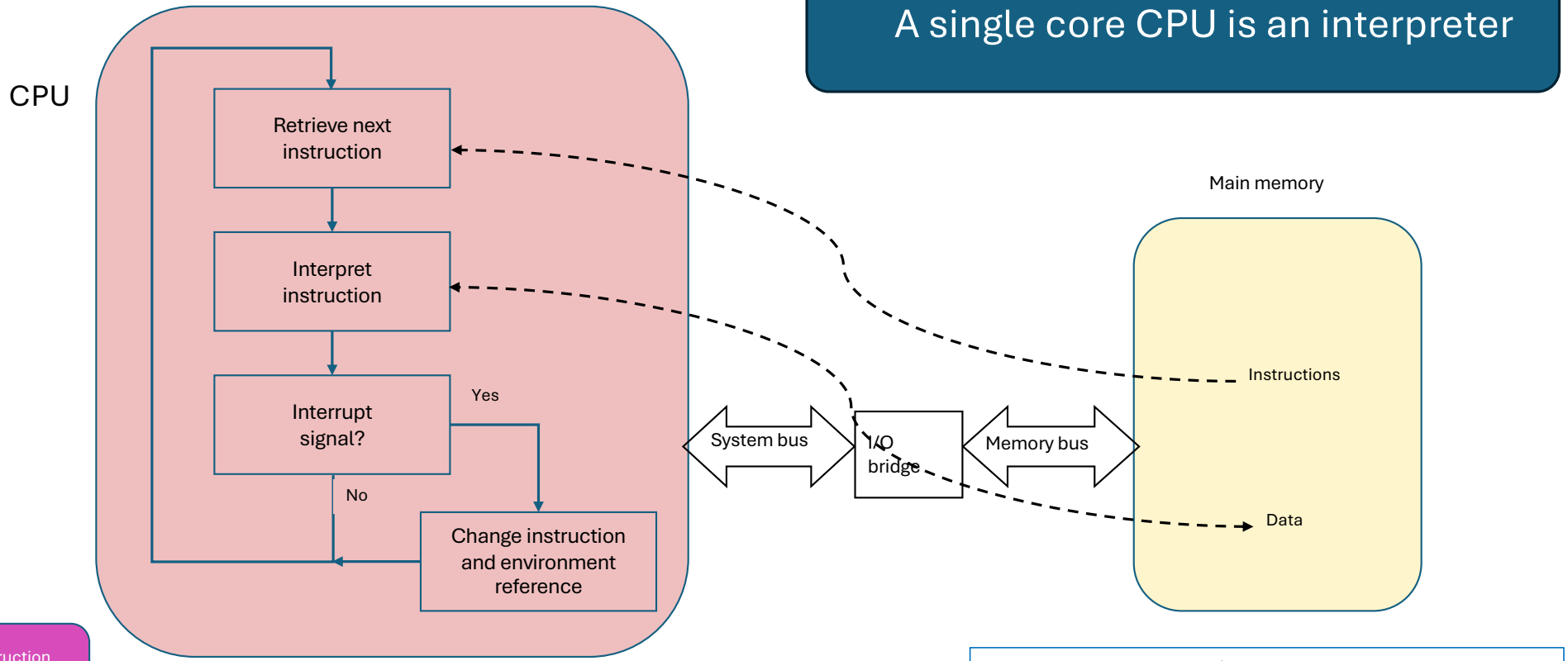
- Multiple cores => multiple parallel threads of executions
- Out of order execution, speculation, multi-issue => multiple parallel instructions (at the cost of non-determinism)
- Single-instruction multiple-data (SIMD) => multiple parallel operations on data

Locality:

- Cache hierarchy per core and across cores within a processor

Processor as Interpreter

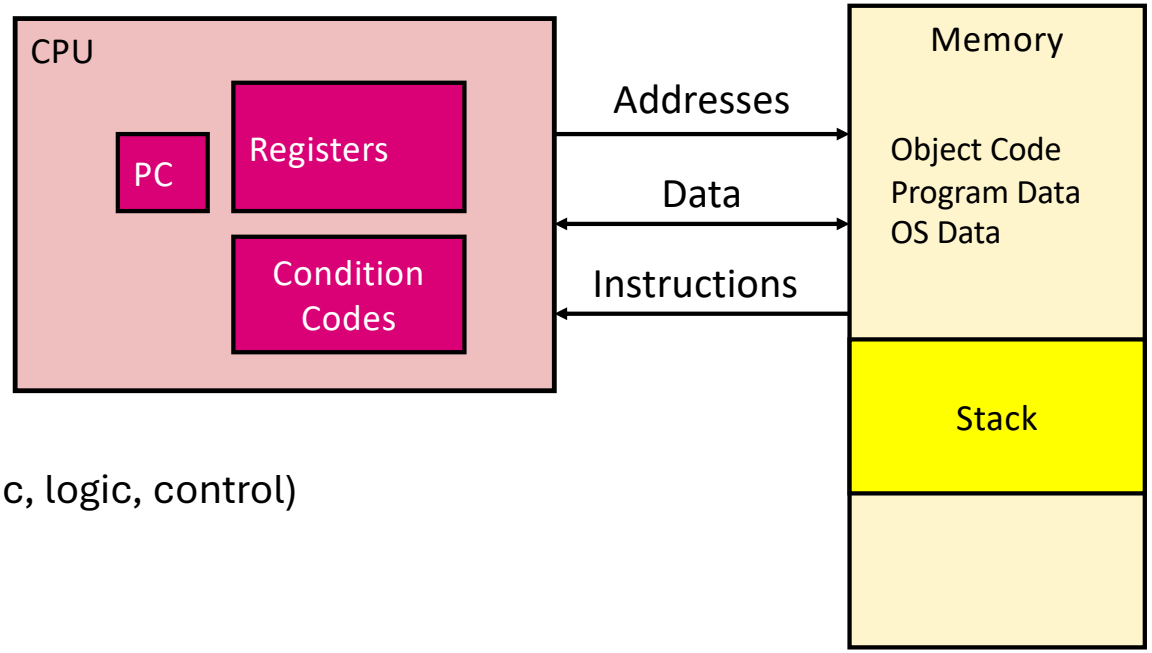
A single core CPU is an interpreter



Instruction repertoire: CISC / RISC

Von Neumann Architecture:
Data and instructions in main memory

Processor abstraction



Programmer-Visible State

PC: Program counter

- Address of next instruction, 8B
- Called “RIP” (x86-64)

Registers

- Data (to be) processed (arithmetic, logic, control)
- Each register contains 8B

Condition codes

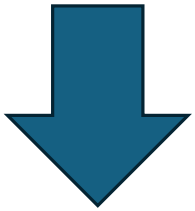
- Store status information about most recent arithmetic operation
- Used for conditional branching

https://en.wikichip.org/wiki/x86

Intel x86-64

Little endian
variable length
instruction set

Binary representation



Textual representation

Assembly language

For historical reasons, r0-r7 are called original registers. They have the following names:

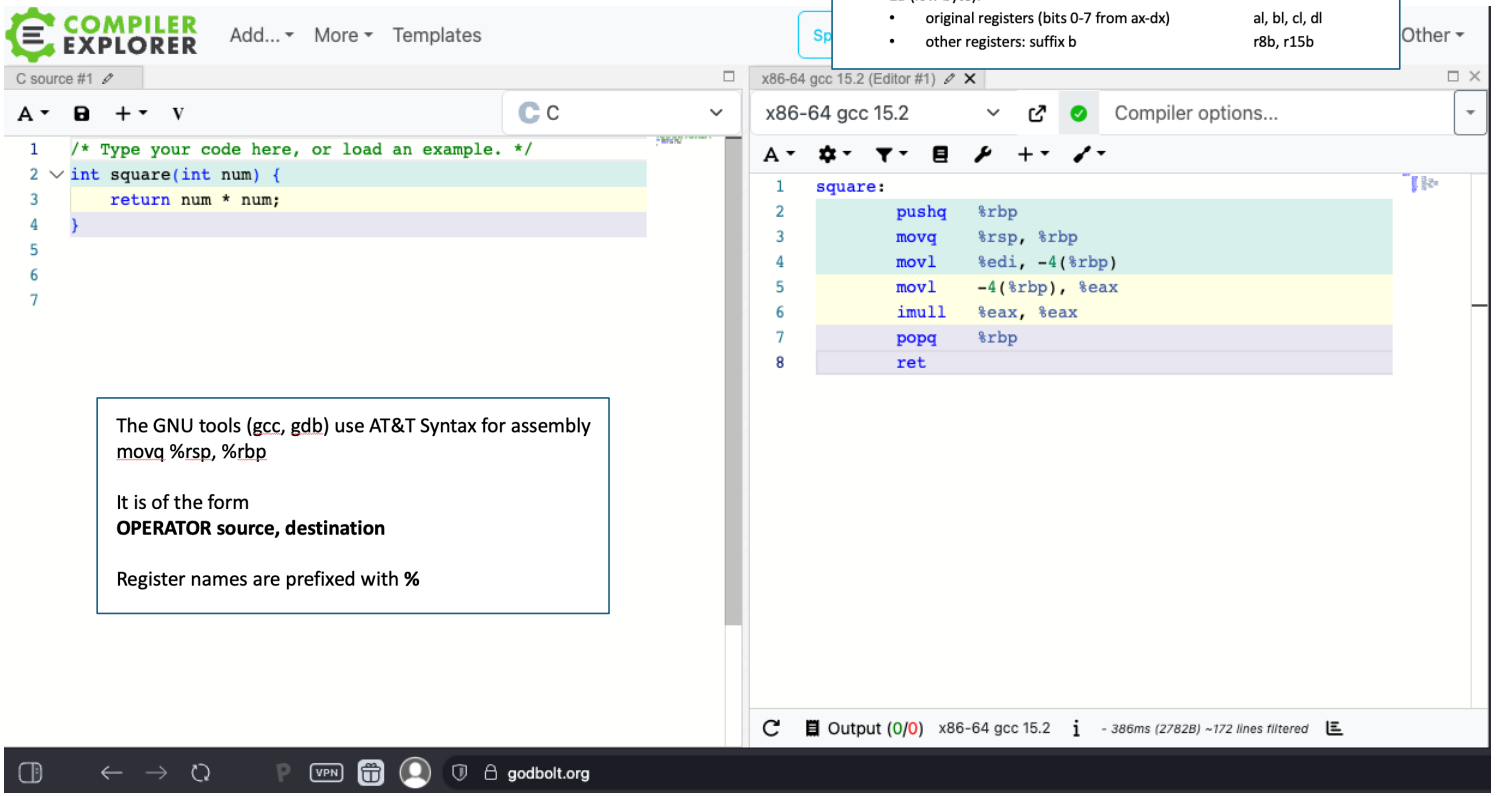
- ax: register a
- bx: register b
- cx: register c
- dx: register d

- bp: register base pointer (start of stack)
- sp: register stack pointer (current location in stack, grow downwards)

- si: register source index (source for data copies)
- di: register destination index (destination for data copies)

Register values can be accessed as:

- **8B**:
  - original registers: **prefix r** — rax, rsp, rsi
  - Other registers: no suffix — r8, r15
- **4B**:
  - original registers: **prefix e** — eax, esp, esi
  - other registers: **suffix d** — r8d, r15d
- **2B**:
  - original registers: **No prefix** — ax, sp, si
  - other registers: **suffix w** — r8w, r15w
- **1B** (high byte):
  - original registers (bits 8-15 from ax-dx) — ah, bh, ch, dh
- **1B** (low byte):
  - original registers (bits 0-7 from ax-dx) — al, bl, cl, dl
  - other registers: suffix b — r8b, r15b



The GNU tools (gcc, gdb) use AT&T Syntax for assembly
movq %rsp, %rbp

It is of the form
OPERATOR source, destination

Register names are prefixed with **%**

Assembly language

Three classes of instructions:

1. Transfer between memory and register
   - Load/store data: register <-> memory
   - Push/pop: register <-> stack

How are registers organized?
How is memory addressed?

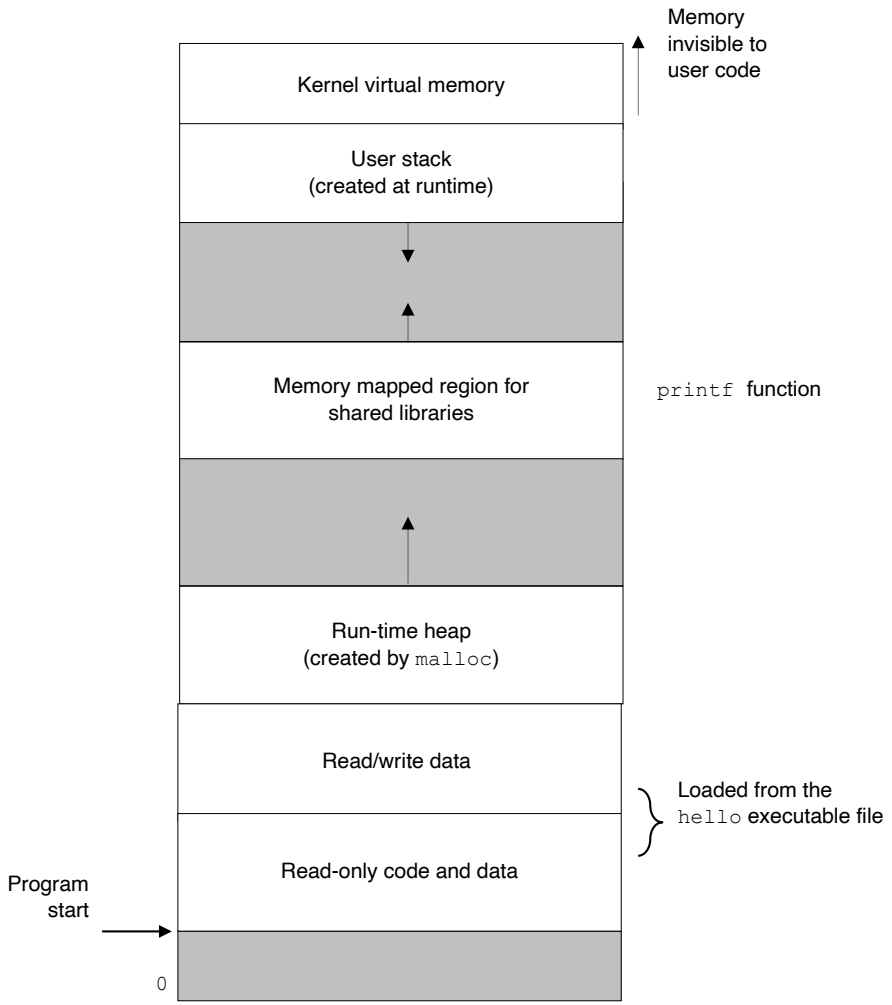
2. Arithmetic and comparison functions

3. Transfer control
   - Jumps to/from procedures
   - Conditional branches

How are procedure calls organized?

Virtual memory

- Every process in Linux has a similar view of virtual memory.
- Assembly manipulates virtual addresses in load/store.
- Within the CPU, a memory management unit translates virtual addresses into physical addresses.



Structured programming

Functions transform inputs into output

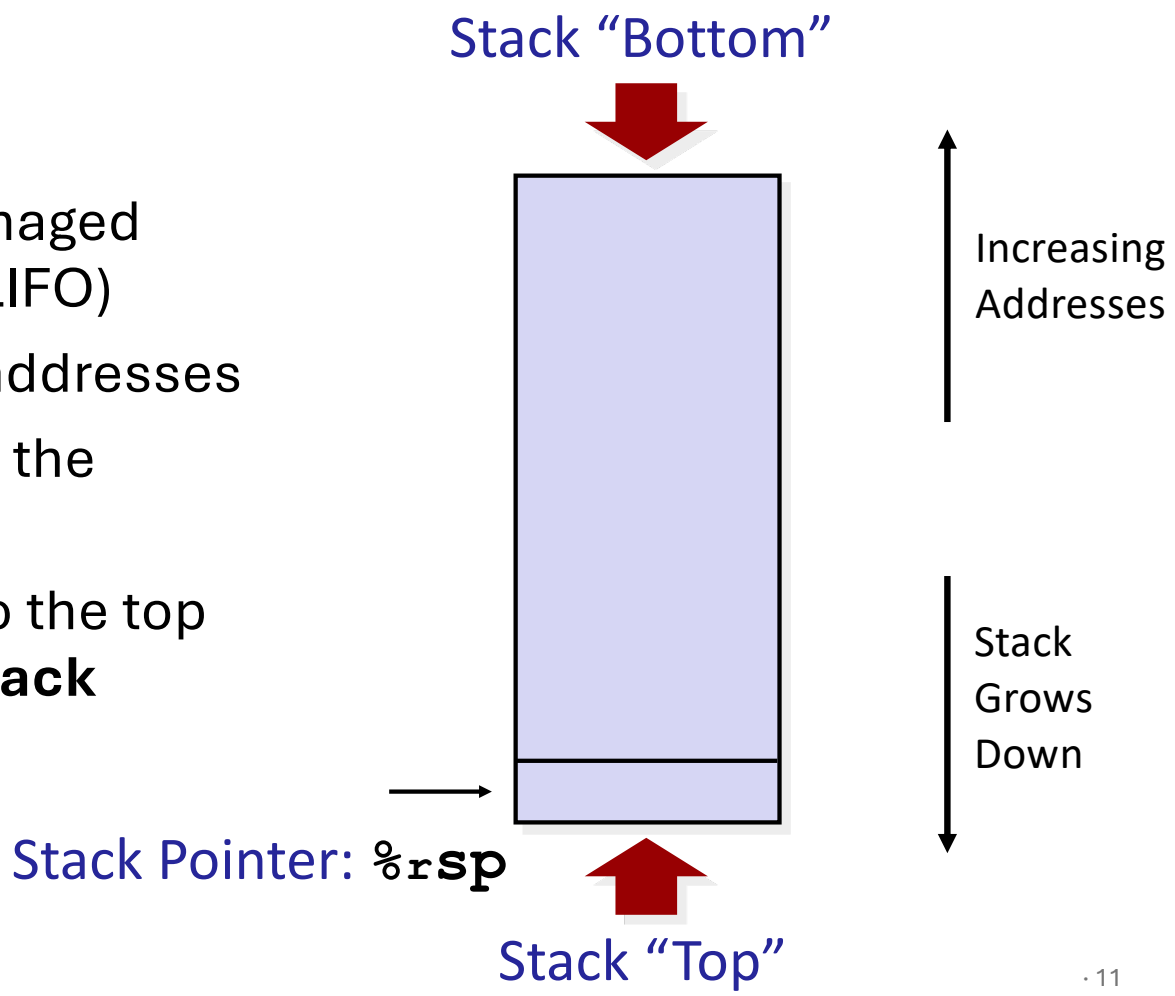
⇒ Memory allocation for input and output parameters
Memory allocation for local variables (defined within the function)

⇒ Control flow (which instruction to execute next)
when calling a function
when returning from a function

These two problems are solved with the **stack**

The Stack

- Region of memory managed with stack discipline (LIFO)
- Grows towards lower addresses
- Regiser %rbp points to the bottom of the stack
- Register %rsp points to the top of the stack, it is the **stack pointer**

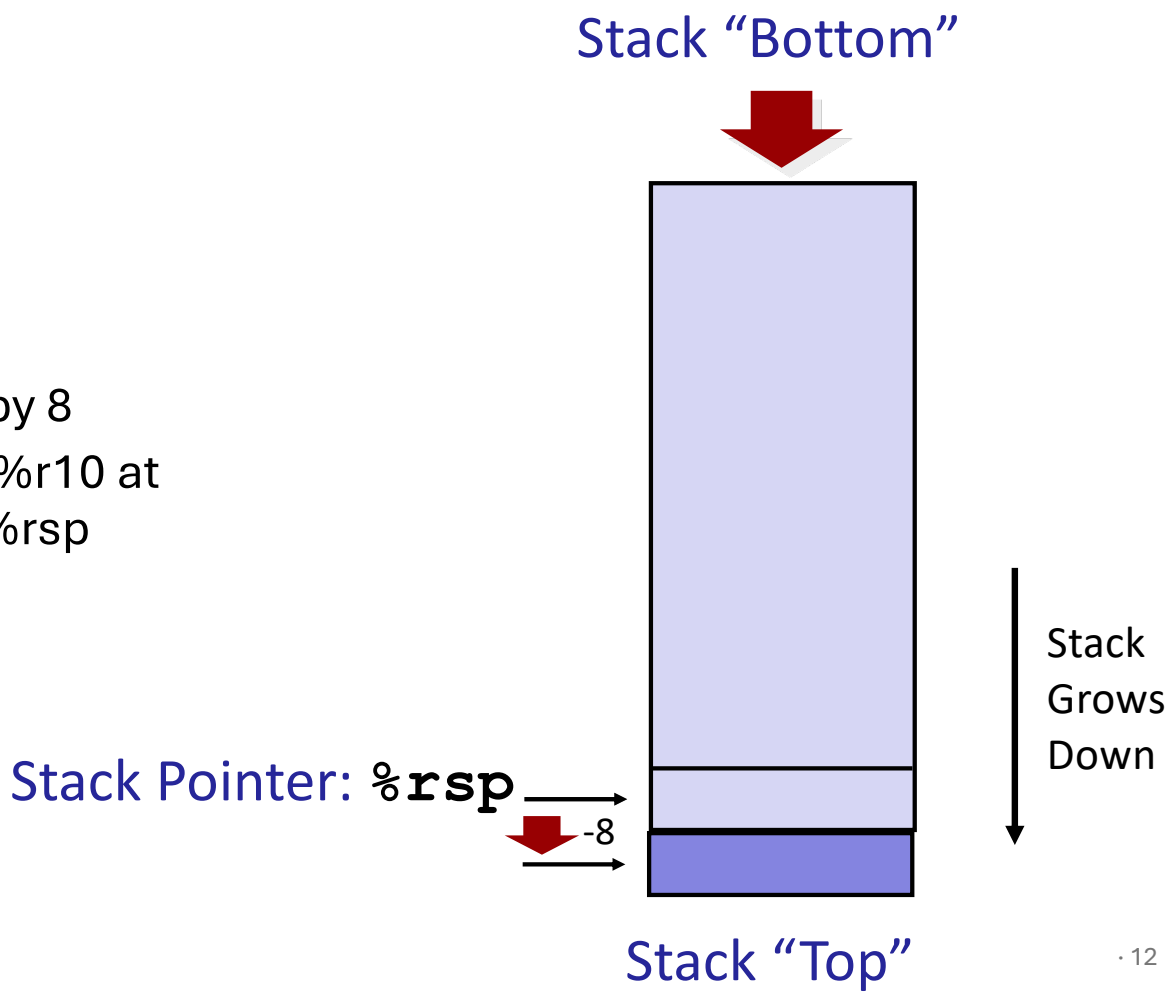


Stack - Push

Operator push

- pushq %r10

1. Decrement %rsp by 8
2. Write contents of %r10 at address given by %rsp

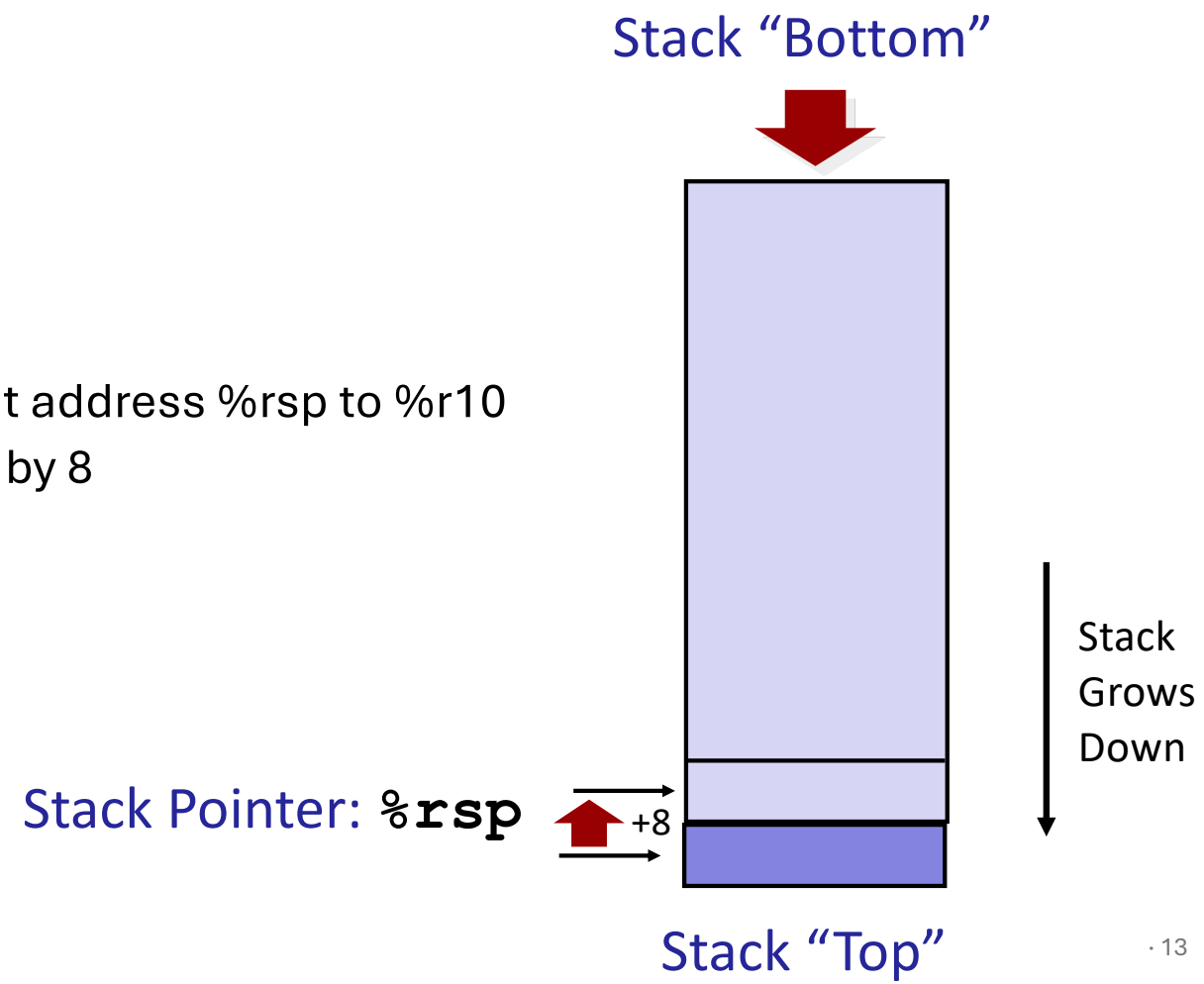


Stack - Pop

Operator pop

- popq %r10

1. Copy contents at address %rsp to %r10
2. Increment %rsp by 8



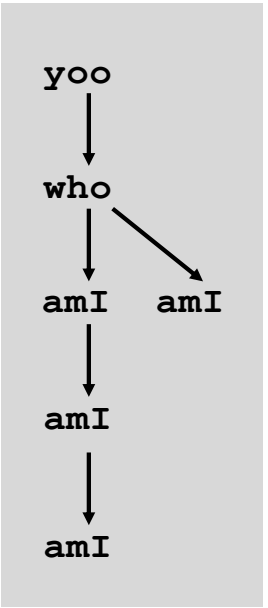
Example

```
yoo(…)
{
   •
   •
   who();
   •
   •
}
```

```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

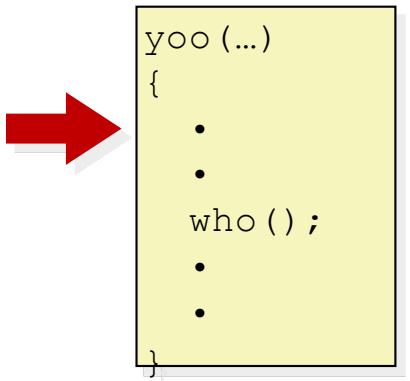
```
amI(…)
{
   •
   •
   amI();
   •
   •
}
```

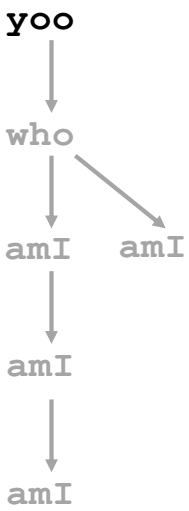
Example
Call Chain

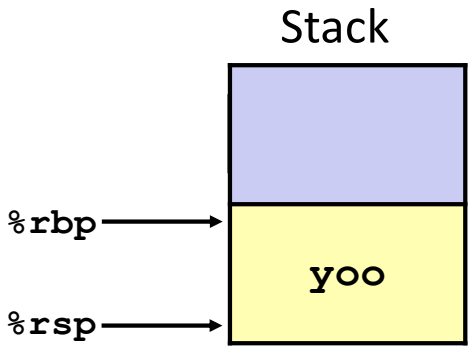


Procedure **amI()** is recursive

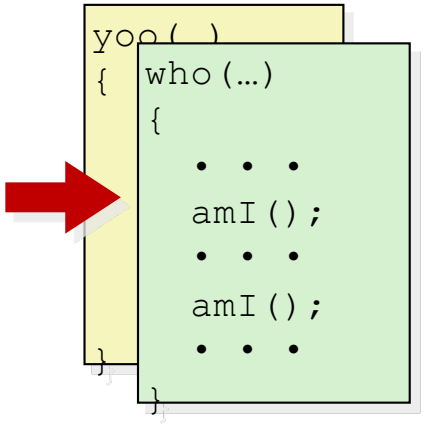
Example

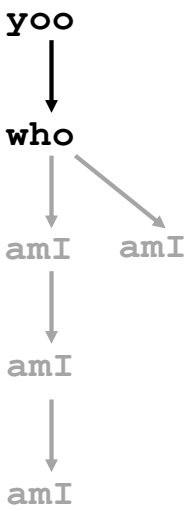


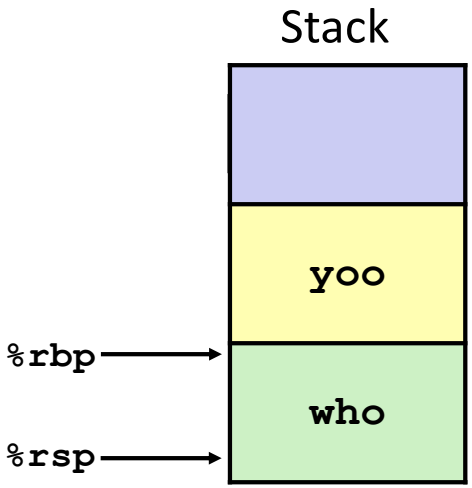




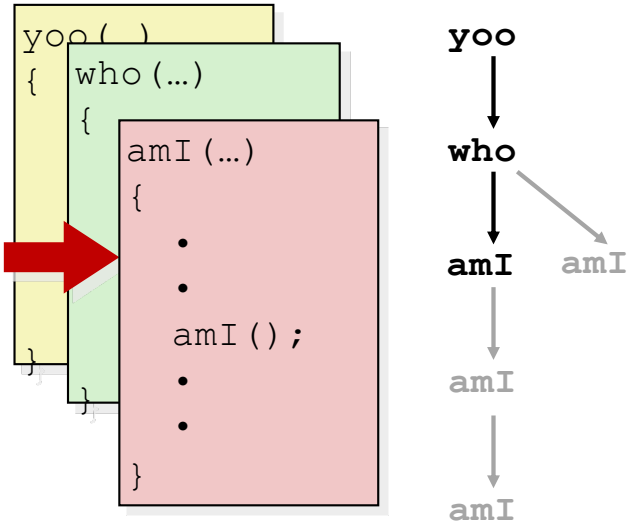
Example



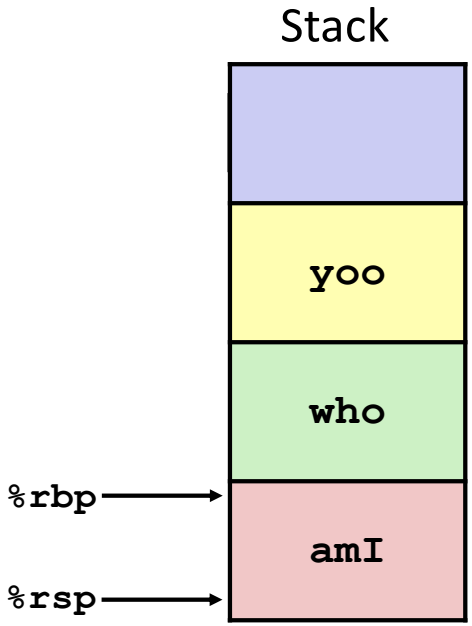




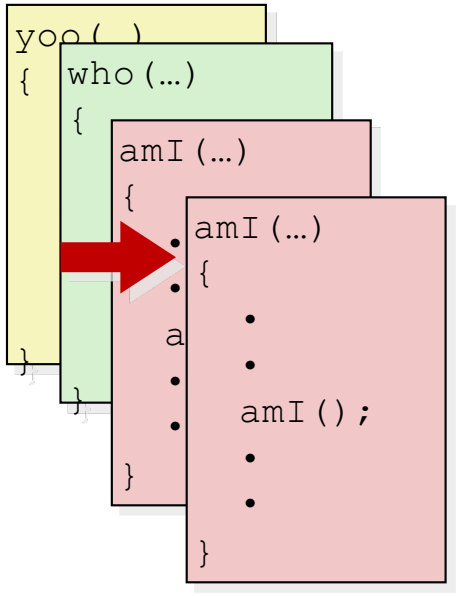
Example

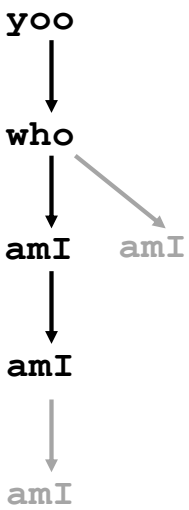

yoo( )
{
who(…)
{
amI(…)
{
•
•
amI();
•
•
}
}
}

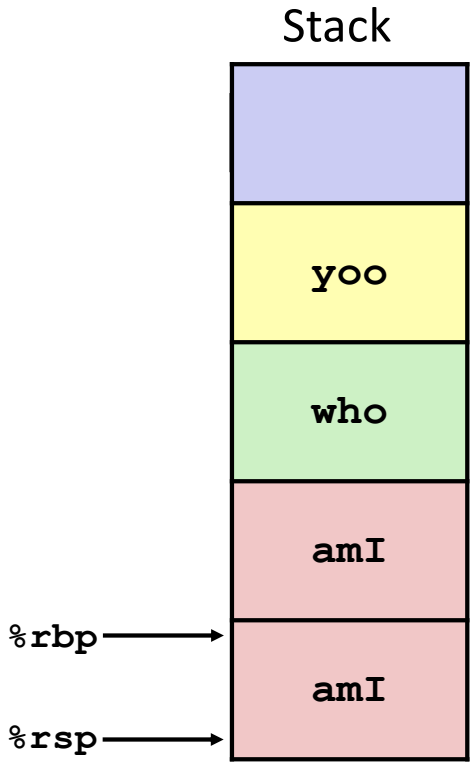
yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

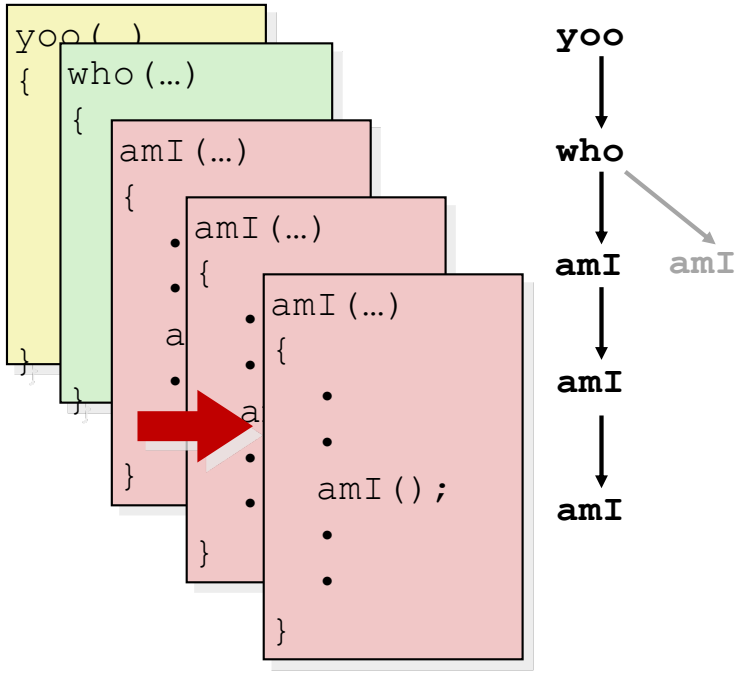
Example


who(…)
{
amI(…)
{
amI(…)
{
amI();
}

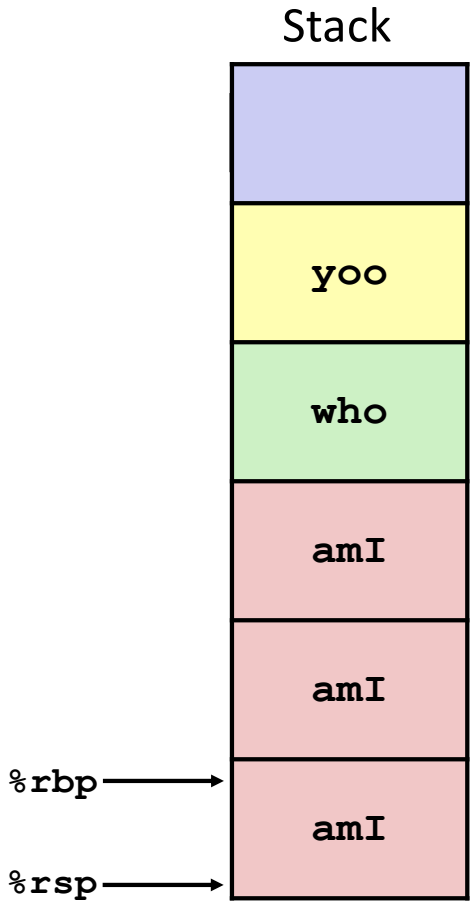

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
%rbp
%rsp

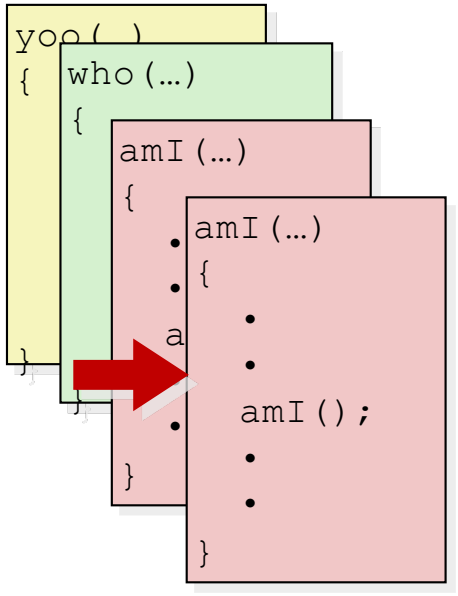
Example

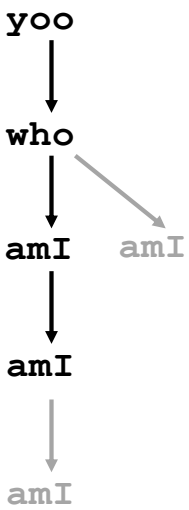

who(…)
amI(…)
amI(…)
amI();

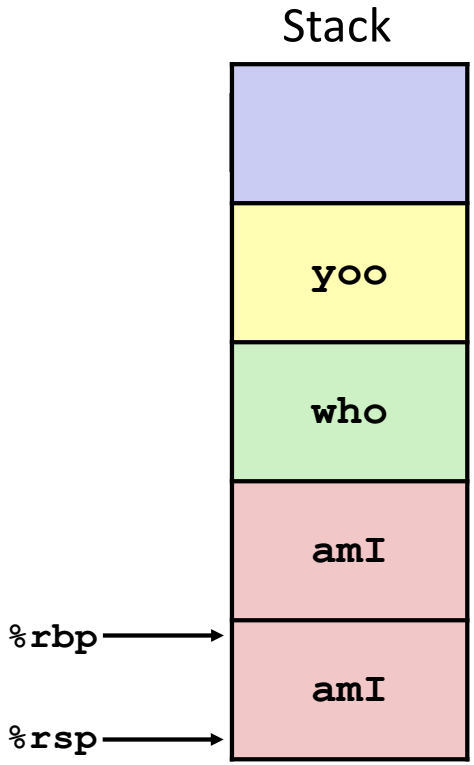
yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
amI
%rbp
%rsp

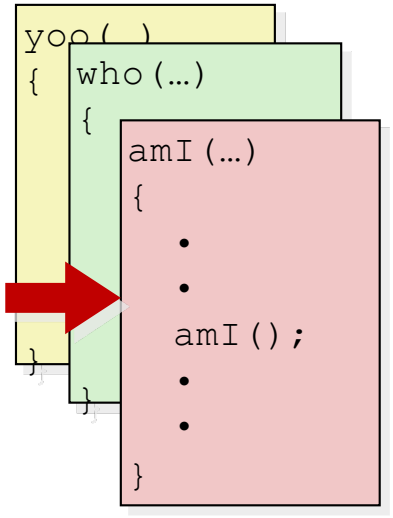
Example

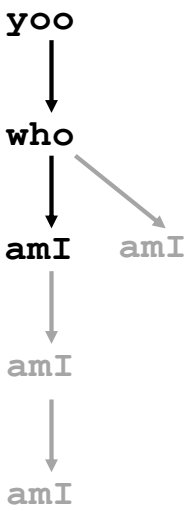

who(…)
{
amI(…)
{
amI(…)
{
amI();
}

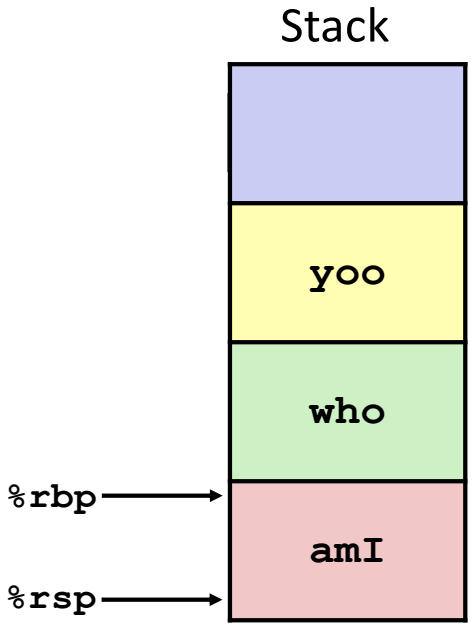

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
%rbp
%rsp

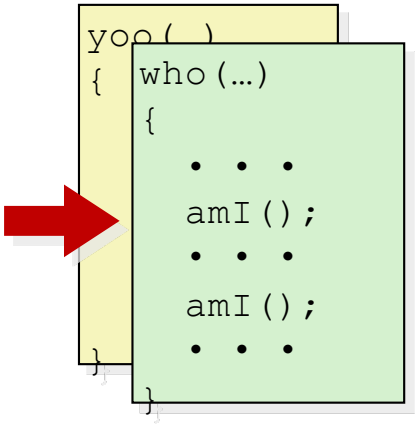
Example

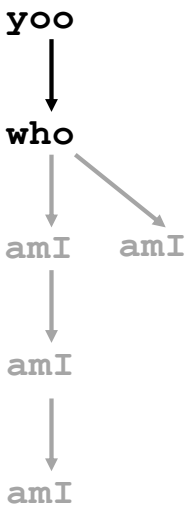

yoo( )
{
who(…)
{
amI(…)
{
•
•
amI();
•
•
}
}
}

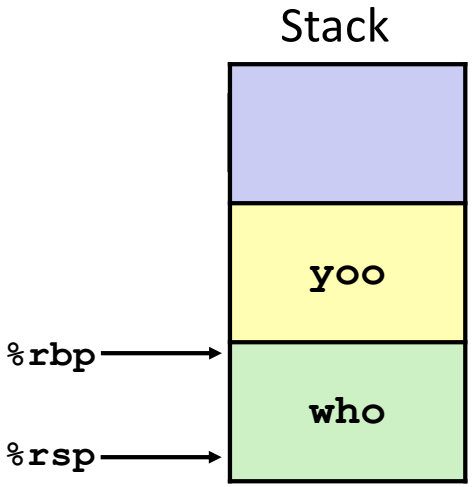

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

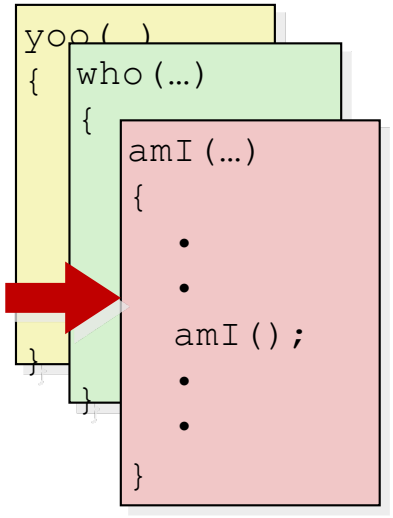
Example

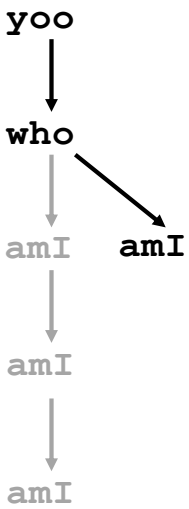


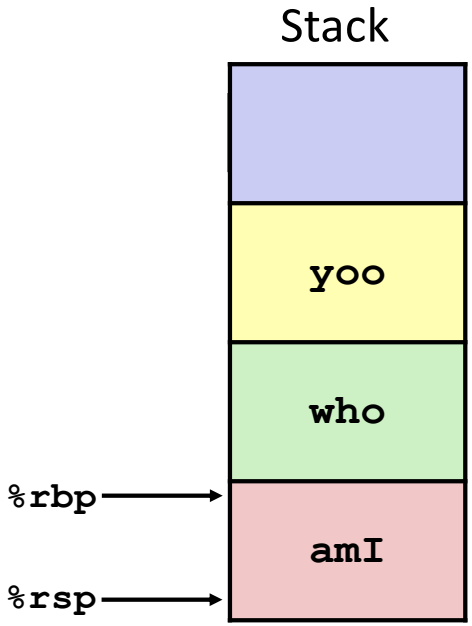




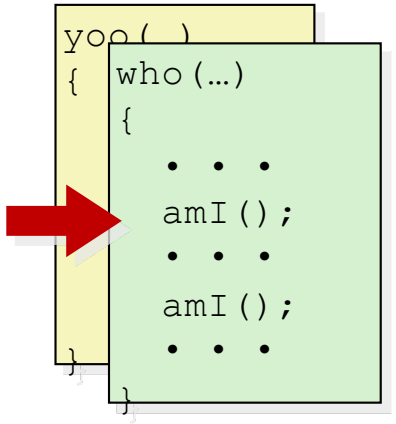
Example

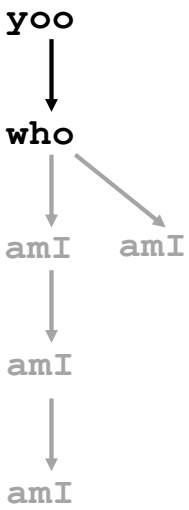

yoo( )
{
who(…)
{
amI(…)
{
•
•
amI();
•
•
}
}
}

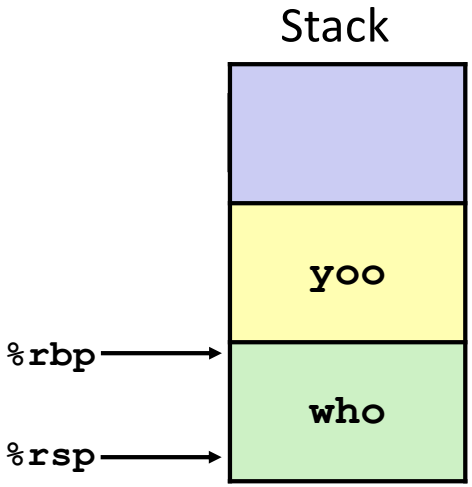

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

Example





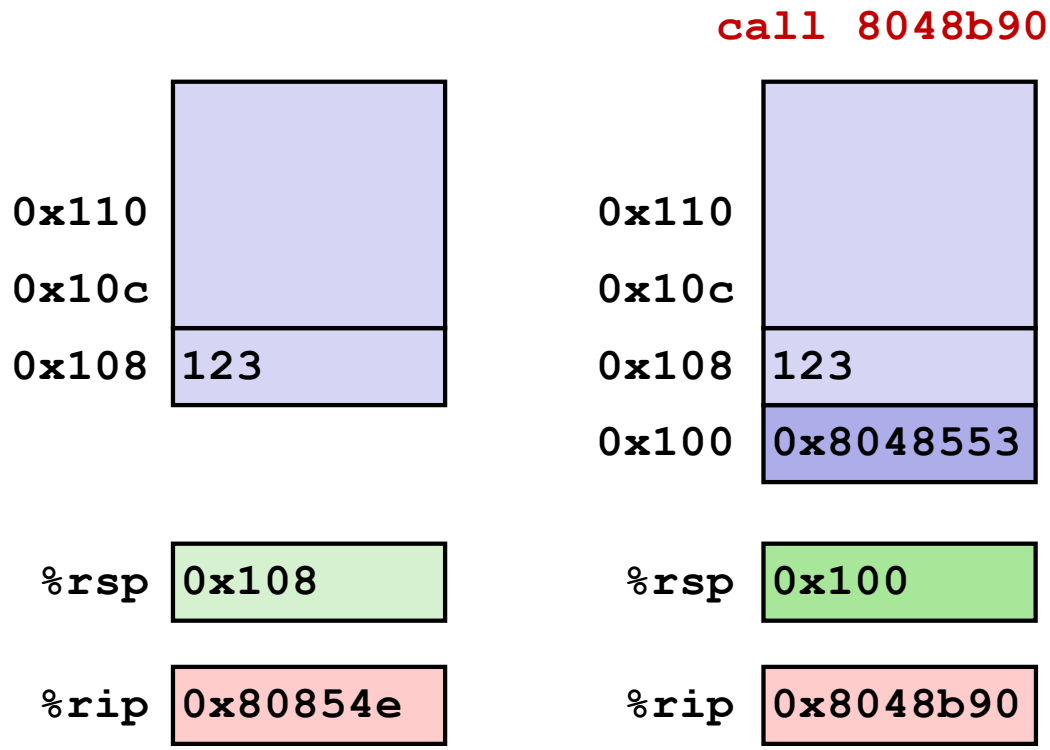


Procedure Call

- Instructions:
  - **call**: push **return address** on stack; jump to label/address
    Return address is address of instruction right after call instruction
  - **ret**: pop return address from stack into instruction pointer

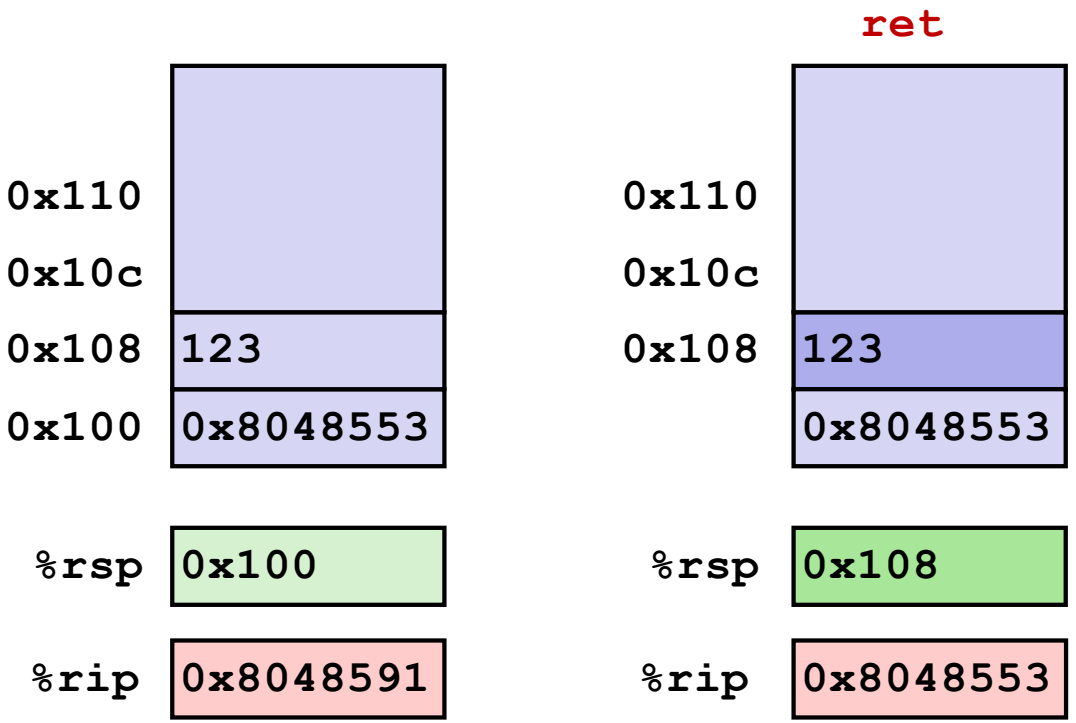
Procedure Call Example

```
804854e:   e8 3d 06 00 00   call   8048b90 <main>
8048553:   50               pushl  %eax
```



Procedure Call Example

```
8048591:   c3                  ret
```



Calling a function (x86-64)

To call a function:

1. **Place the first six integer or pointer parameters in %rdi, %rsi, %rdx, %rcx, %r8 and %r9**
2. Push onto the stack subsequent parameters and parameters larger than 8B (in order).
3. Execute the call instruction, which:
   - Pushes the return address onto the stack
   - Jumps to the start of the specified function (i.e., update instruction pointer)

Executing a function

- The C run-time system introduces instruction to set-up and clean-up the stack in each procedure.

- Set-up consists in allocation and initialization of a stack-frame. Clean-up consists in deallocating a stack frame.

- A stack-frame is the space needed on the stack by a procedure for storing:
  - The return address
  - (some) parameters
  - Local variables

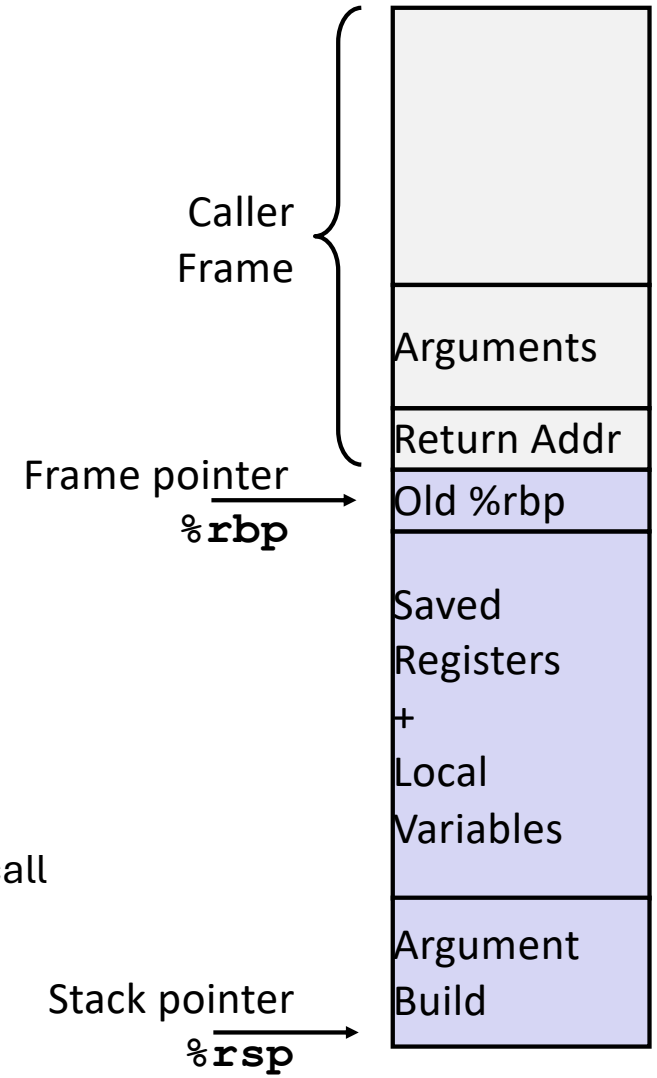
Stack Frame

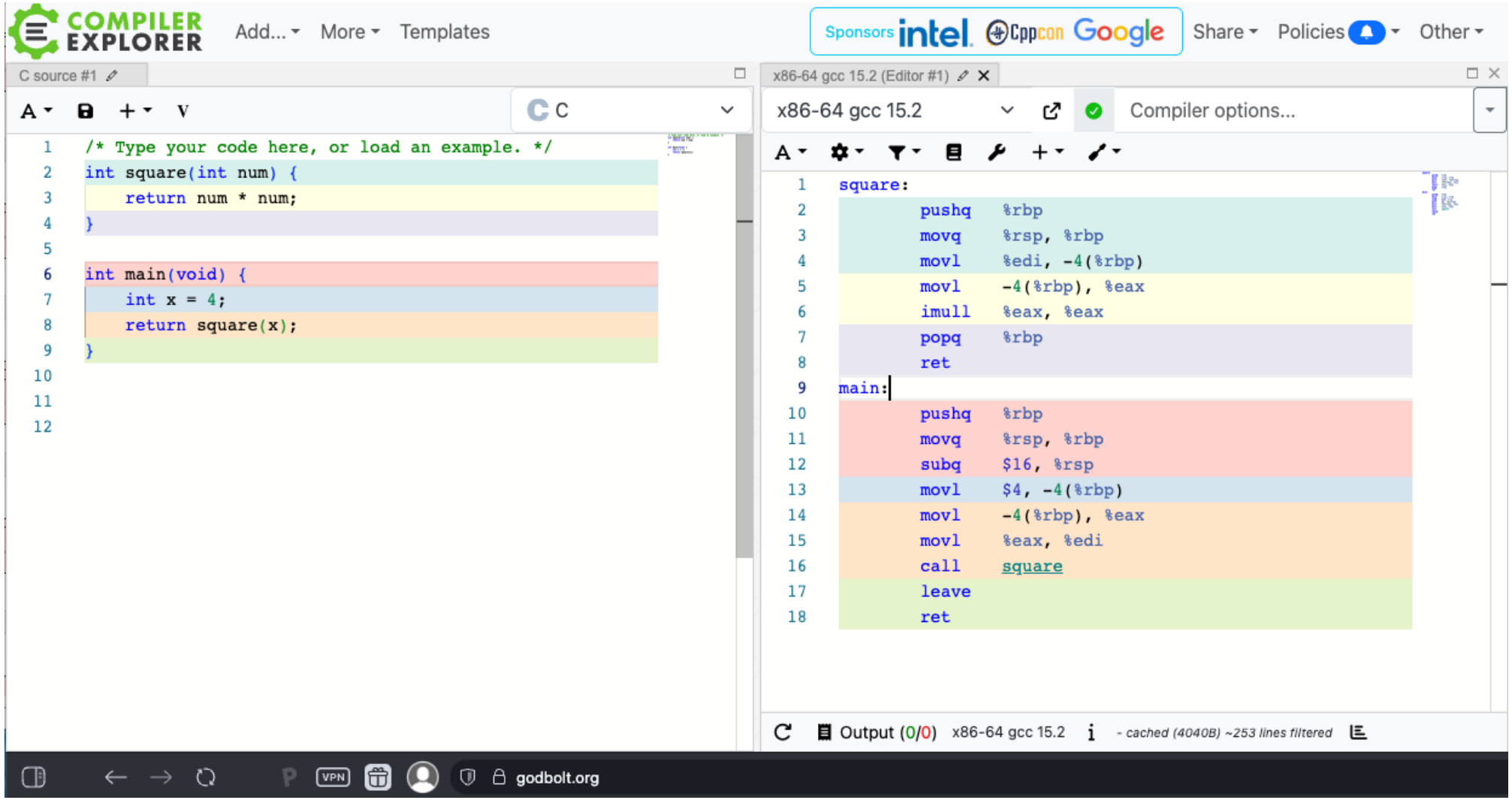
Caller:

- Arguments
  - pushed by program (if needed)
- Return address
  - pushed by call

Callee:

- Previous frame pointer (%rbp)
- Other callee-save registers (%rbx, %r12-15)
- Space for local variables
- Arguments for next function (when about to call another function)




COMPILER EXPLORER
Add...
More
Templates
Sponsors intel CppCon Google
Share
Policies
Other
C source #1
C
x86-64 gcc 15.2 (Editor #1)
x86-64 gcc 15.2
Compiler options...
/* Type your code here, or load an example. */
int square(int num) {
return num * num;
}
int main(void) {
int x = 4;
return square(x);
}
square:
pushq %rbp
movq %rsp, %rbp
movl %edi, -4(%rbp)
movl -4(%rbp), %eax
imull %eax, %eax
popq %rbp
ret
main:
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $4, -4(%rbp)
movl -4(%rbp), %eax
movl %eax, %edi
call square
leave
ret
Output (0/0)
x86-64 gcc 15.2
- cached (4040B) ~253 lines filtered
VPN
godbolt.org

Compilers and interpreters

- Compiled languages
  - \+ Almost always faster.
  - \- Require compilation after every change.
  - \- Usually cannot run program fragments in isolation.
  - \- Tend to have more restrictions (e.g. static typing).
  - \- Much more difficult to implement.
- Interpreted languages
  - \- Usually slow.
  - \+ Can run immediately.
  - \+ Can easily run fragments (e.g. single functions) in isolation.
  - \+ Much easier to implement.

(source T.Henriksen)

The Collatz conjecture

$$f(n) = \begin{cases} \frac{n}{2} & \text{if } n \text{ is even} \\ 3n+1 & \text{if } n \text{ is odd} \end{cases}$$

***Conjecture**: if we apply this function to some number greater than 1, we will eventually reach 1.*

Listing 1: `collatz.py`

```python
import sys

def collatz(n):
    i = 0
    while n != 1:
        if n % 2 == 0:
            n = n // 2
        else:
            n = 3 * n + 1
        i = i + 1
    return i

k = int(sys.argv[1])
for n in range(1, k):
    print(n, collatz(n))
```

Listing 2: `collatz.c`

```c
#include <stdio.h>
#include <stdlib.h>

int collatz(int n) {
  int i = 0;
  while (n != 1) {
    if (n % 2 == 0) {
      n = n / 2;
    } else {
      n = 3 * n + 1;
    }
    i++;
  }
  return i;
}

int main(int argc, char** argv) {
  int k = atoi(argv[1]);
  for (int n = 1; n < k; n++) {
    printf("%d %d\n", n, collatz(n));
  }
}
```

```
htl719@kumac ~/D/C/H/h/w/lec-tue> make python-run
time python3 ./collatz.py 100000 >/dev/null

real    0m1.268s
user    0m1.247s
sys     0m0.009s
htl719@kumac ~/D/C/H/h/w/lec-tue> make run
time ./collatz 100000 >/dev/null

real    0m0.041s
user    0m0.036s
sys     0m0.001s
htl719@kumac ~/D/C/H/h/w/lec-tue> make optim-run
time ./o-collatz 100000 >/dev/null

real    0m0.015s
user    0m0.012s
sys     0m0.001s
```

```
plain: collatz.c
    gcc collatz.c -o collatz

optim: collatz.c
    gcc -O3 -ffast-math collatz.c -o o-collatz
```

$$\text{Speedup} = \frac{1.268}{0.015} = 84.5$$

1st run

```
htl719@kumac ~/D/C/H/h/w/lec-tue> make python-run
time python3 ./collatz.py 100000 >/dev/null

real    0m1.449s
user    0m1.289s
sys     0m0.020s
htl719@kumac ~/D/C/H/h/w/lec-tue> make run
time ./collatz 100000 >/dev/null

real    0m0.224s
user    0m0.041s
sys     0m0.004s
htl719@kumac ~/D/C/H/h/w/lec-tue> make optim-run
time ./o-collatz 100000 >/dev/null

real    0m0.144s
user    0m0.014s
sys     0m0.004s
```

Mixing C and Python

Compiling C programs into object file that can be linked statically (at compile time)

or

shared object file that can be linked from Python (at run-time)

```
object: collatz.c
    gcc -c collatz.c -o collatz.o

reloc: collatz.c
    gcc -fPIC -shared collatz.c -o libcollatz.so
```

Linking libraries from Python

```
1  import sys
2  import ctypes
3
4  c_lib = ctypes.CDLL('./libcollatz.so')
5
6  if __name__ == '__main__':
7      k = int(sys.argv[1])
8      sum = 0
9      for i in range(1, k):
10         sum += c_lib.collatz(i)
11     print(sum)
~
```

Mixing Python and C

```
htl719@kumac ~/D/C/H/h/w/lec-tue> time python3 collatz-ffi.py 100000
10753712
python3 collatz-ffi.py 100000  0.07s user 0.01s system 22% cpu 0.374 total
```

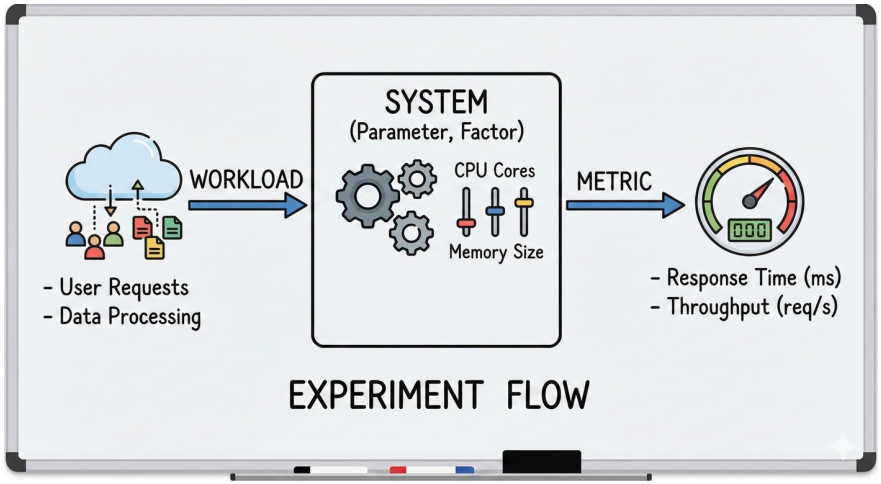
- x5 slower compared to C
- X20 faster compared to Python

- Faster if we made fewer “foreign” calls, but each took more time.
- Ideal case is single foreign function call that operates on many values.
- **This is exactly how NumPy works!**

Experiments

Experimental framework

- System
  - Parameters (fixed)
  - Factors (take different values in an experiment)
- Workload
- Metric
  - Time / space
  - Throughput / latency
- Experiments
  - Studying the impact of modifying one factor at a time



(source: gemini)

Experimental results:

1. What you expect
2. What you observe
3. How you explain the discrepancies