Lecture #4 - Computer Architecture

AMath 483/583 - Spring 2016

Announcements

- Homework #1 Technical Issues —> Extension to **Monday, 25 April 5:00pm**
- Good Homework #1 Strategy
    - [1] Jupyter Notebook —> Requested functions / scripts / modules
    - [2] Write directly to target. Write many tests.
- Primary Resources for this lecture on Syllabus

Binary

- Decimal numeral system (base 10)

$$31.415 = 3 \times (10^1) + 1 \times (10^0) + 4 \times (10^{-1}) + 1 \times (10^{-2}) + 5 \times (10^{-3})$$

- Binary system (base 2)

$$\begin{aligned} 110.110_2 &= 1 \times (2^2) + 1 \times (2^1) + 0 \times (2^0) + 1 \times (2^{-1}) + 1 \times (2^{-2}) + 0 \times (2^{-3}) \\ &= 6.75_{10} \end{aligned}$$

- Hexadecimal system (base 16)

$$2a4.8f_{16} = 2 \times (16^2) + 10 \times (16^1) + 4 \times (16^0) + 8 \times (16^{-1}) + 15 \times (16^{-2})$$

Storing Data on a Computer

- Computer memory divided into “bytes”: 1 byte = 8 bits
- 1 byte = 256 different values
- `int` = 4 bytes (32 bits)
- `long` = 8 bytes (64 bits)
- 0 + 0 = 0
  0 + 1 = 1 + 0 = 1
  1 + 1 = 10

8-bit integers

`00000000` = 0

`00000001` = 1

`00000010` = 2

`00000011` = 3

…

`11111111` = 255

Storing Data on a Computer

- Computer memory divided into “bytes”: 1 byte = 8 bits
- 1 byte = 256 different values
- `int` = 4 bytes (32 bits)
- `long` = 8 bytes (64 bits)
- 0 + 0 = 0
  0 + 1 = 1 + 0 = 1
  1 + 1 = 10

8-bit signed integers

`10000000`	= -128
`10000001`	= -127
`10000010`	= -126
	…
`11111110`	= -2
`11111111`	= -1
`00000000`	= 0
`00000001`	= 1
`00000010`	= 2
	…
`01111111`	= 127

Storing Data on a Computer

- What about floating point numbers?
- Base 10 proposal: integer part + fractional part

```
00003.14159  (pi)
00000.000314 (pi / 10000)
31415.90000  (pi * 10000)
```

- Disadvantages:
  - precision depends on size of number
  - many wasted bits
  - limited range (science requires large and small numbers)

Storing Data on a Computer

- Solution: use scientific notation

  `0.2345e-18` = 0.2345 x 10^(18) =
  0.0000000000000000002345

- Mantissa = 0.2345,   Exponent = -18

- Mantissa = 0.10110,   Exponent = -11011

$$\begin{aligned} 0.101101 &= 1(2^{-1}) + 0(2^{-2}) + 1(2^{-3}) + 1(2^{-4}) + 0(2^{-5}) + 1(2^{-6}) \\ &= 0.703125_{10} \\ -11011 &= -1(2^4) + 1(2^3) + 0(2^2) + 1(2^1) + 1(2^0) \\ &= -27_{10} \end{aligned}$$

- So the number is:

$$0.703125 \times 10^{-27}$$

Storing Data on a Computer

- double = 8 bytes (64 bits)
  - 53 bit (signed)mantissa, 11 bit (signed)exponent
  - 52 bits of significant figures / precision

$$2^{-52} \approx 2.2 \times 10^{-16}$$

roughly 15 digits of precision

Floating Point Operations

- Often, digits of precision are lost in operations (add, mul). Two reasons:
  - non-exact binary representation

    `0.1 (base 10) = 0.0001100110011…`

  - numbers with different scales (exponents)

$$0.123 \times 10^{18} + 0.456 \times 10^{-12}$$

Quick Python Demo

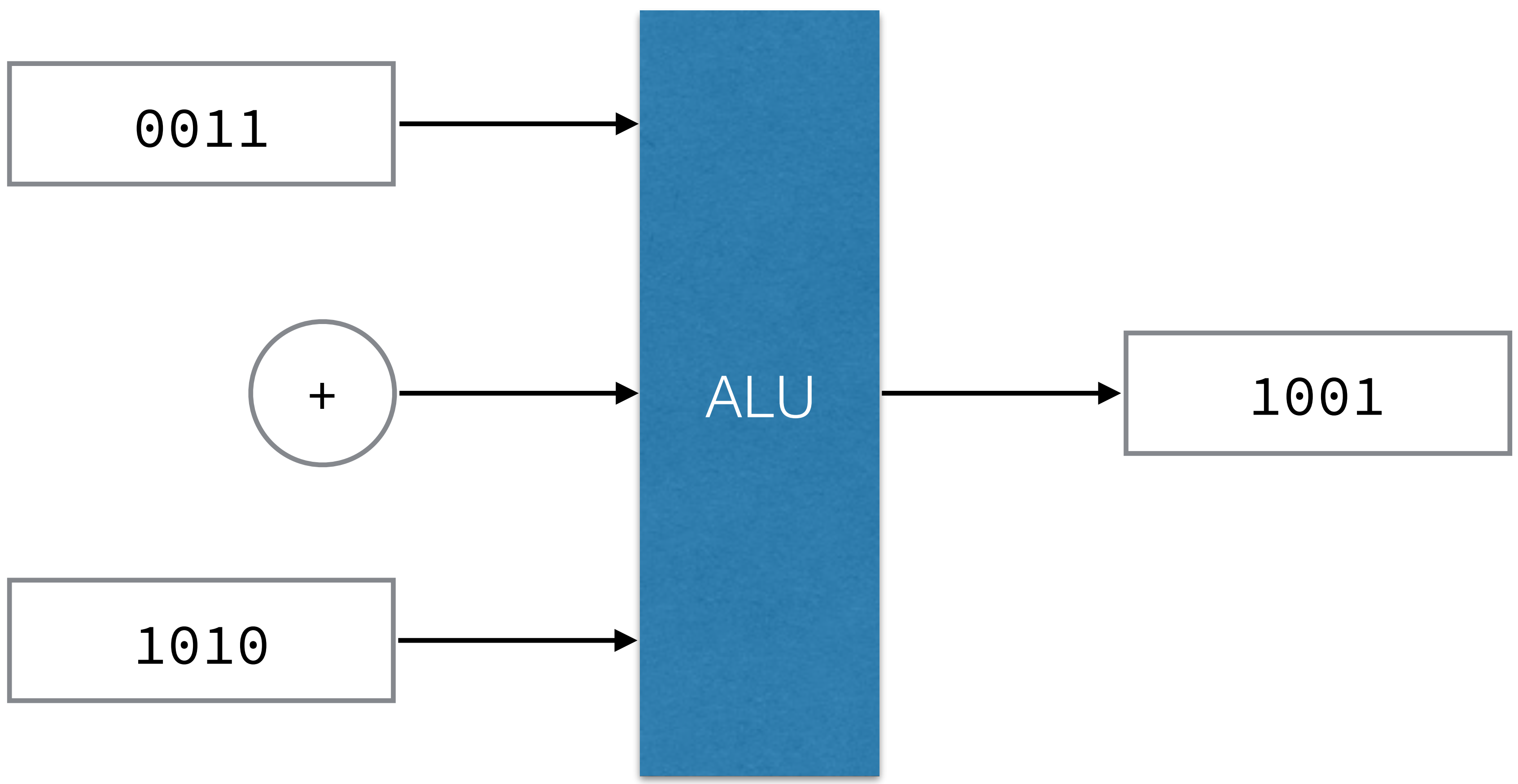
Floating Point Operations Error

CPU

- Carries out arithmetic instructions on input data
- Components:
  - control unit - directs CPU to carry out instructions
  - arithmetic logic unit - bitwise operations
  - memory management unit - maps data to location in RAM, cache, etc.

CPU

- arithmetic logic unit - two data inputs and an operation input with corresponding output



- "Execution pipeline" - control unit and ALU managing tasks

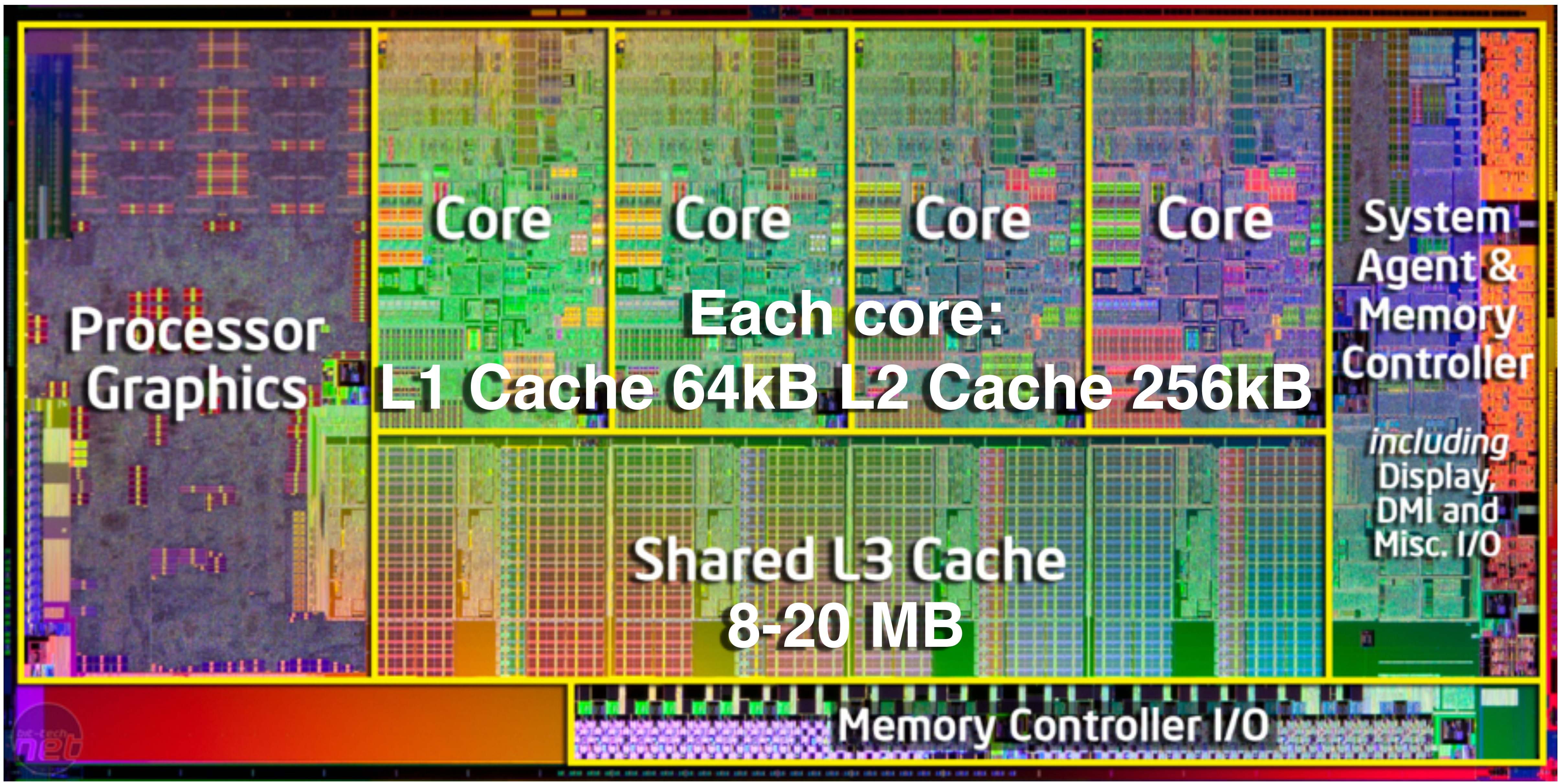
CPU

- memory management unit - manages registers and RAM addresses
  - registers - 32-bit / 64-bit storage units for the processor / ALU to retrieve and store data
    - data, address, instruction, stack pointer, …

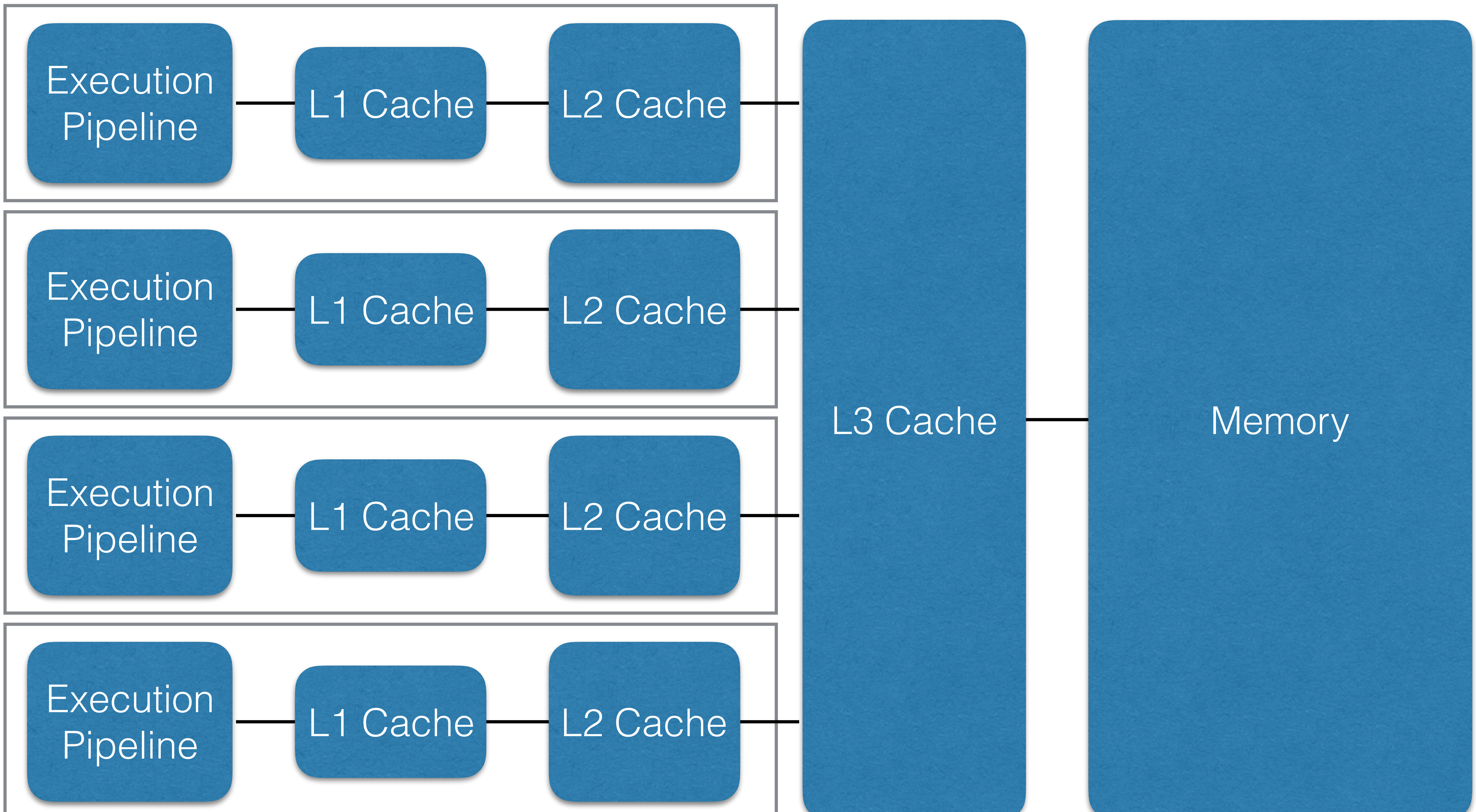
Memory

- RAM - Random Access Memory
  - where your program and data live
- Registers - where your CPU can interact with data
- Cache - an “in-between” space where data from RAM is moved “closer” to the CPU for performance

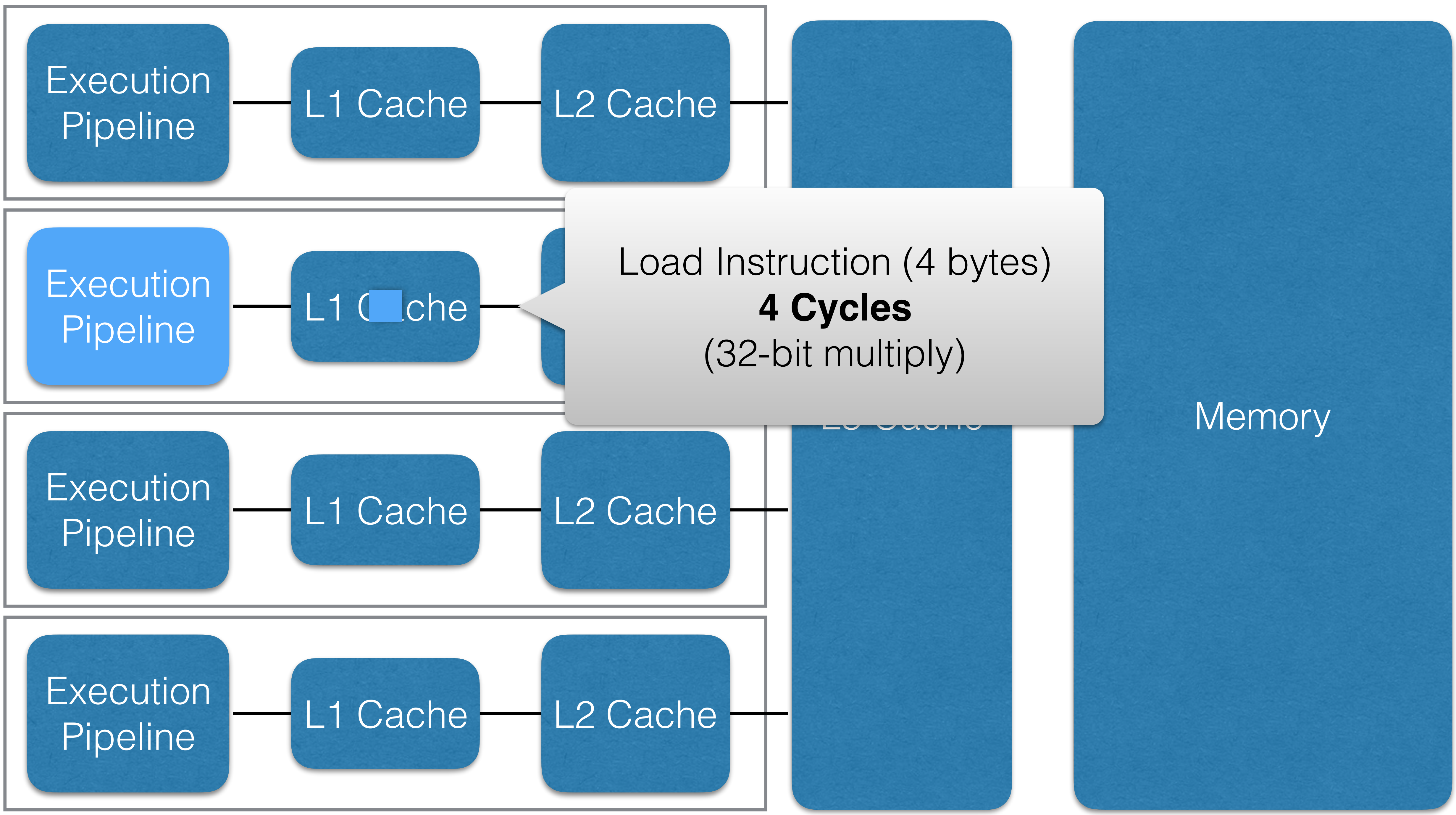
Intel Sandy Bridge CPU



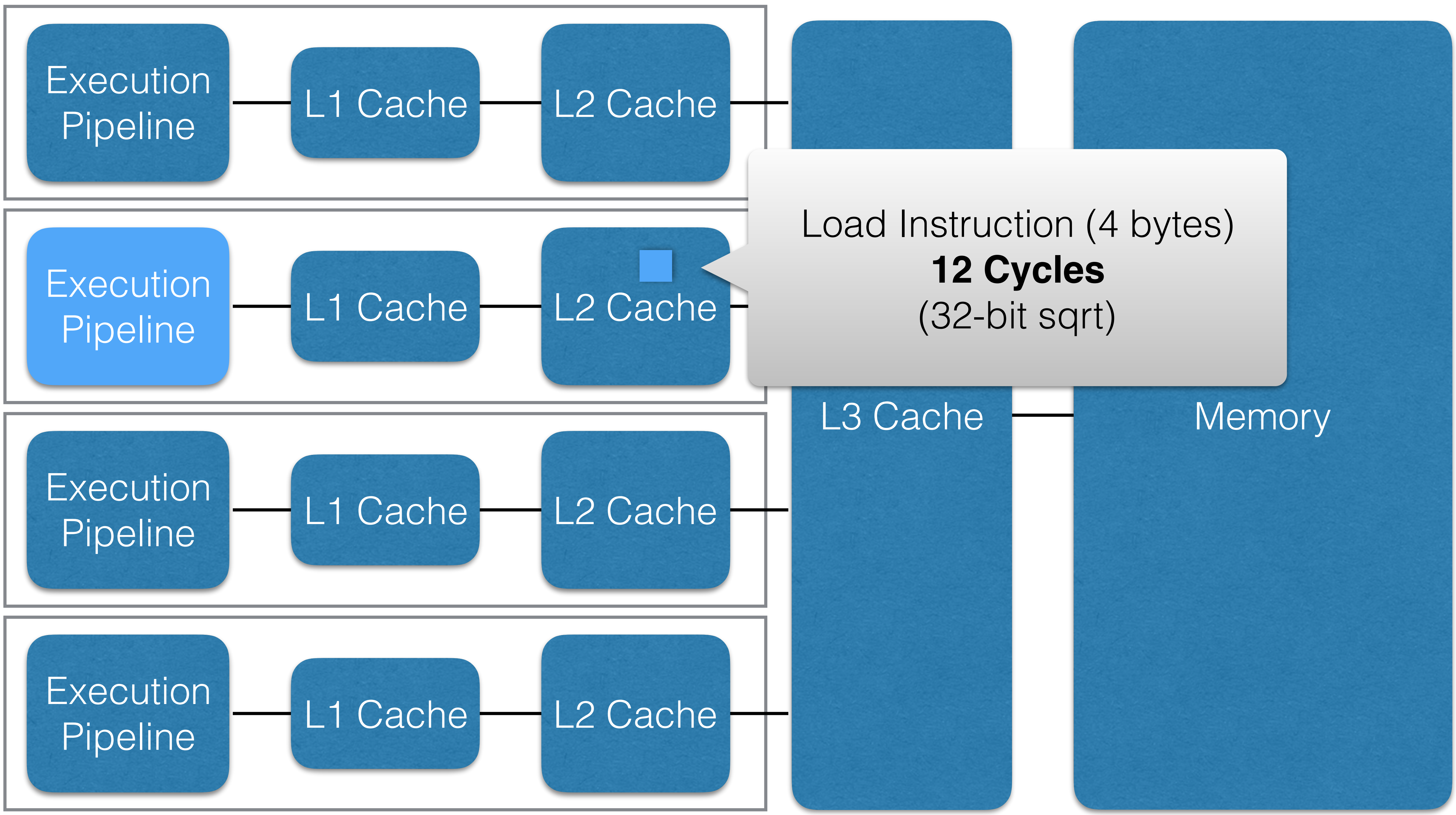
Memory Layout



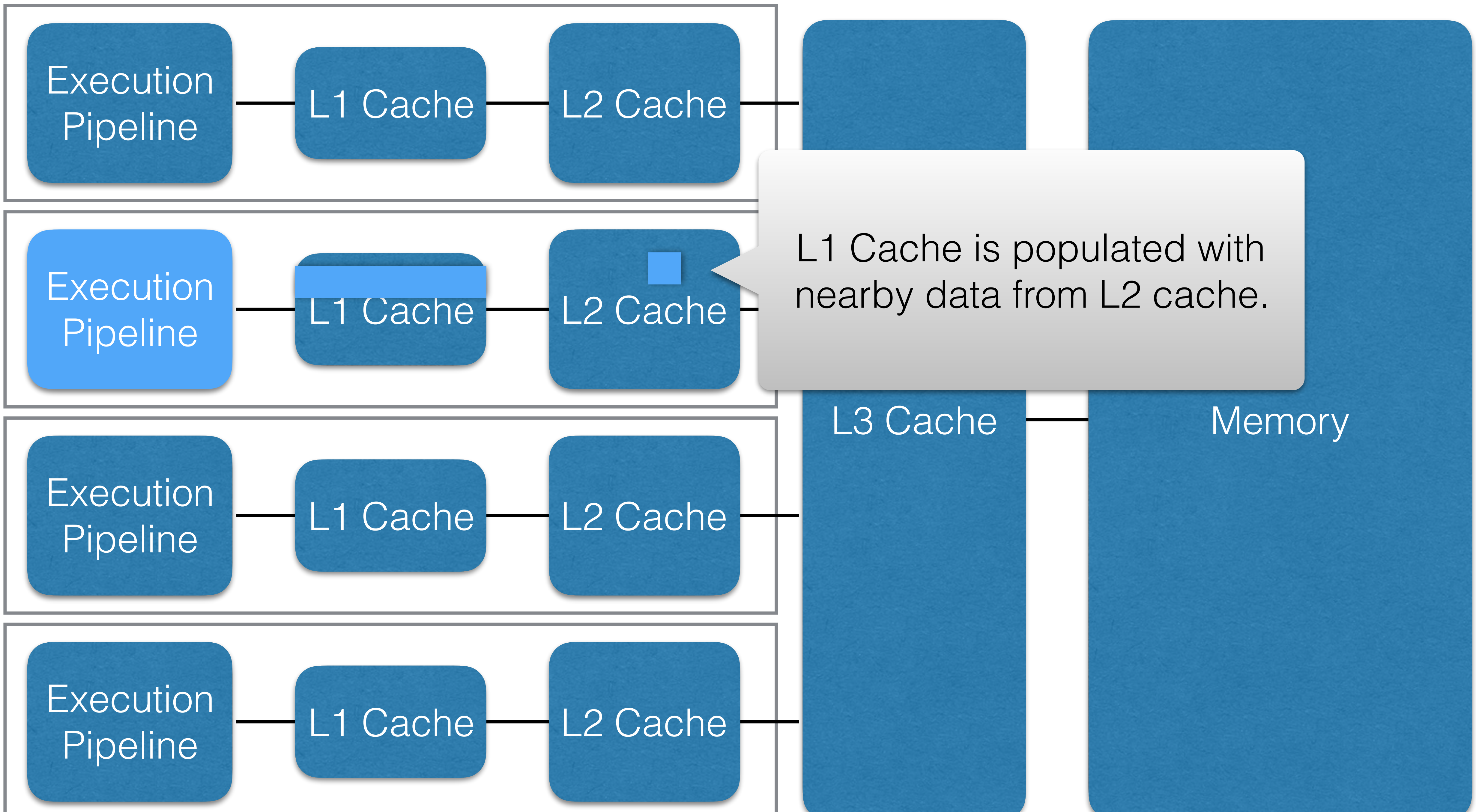
Memory Layout



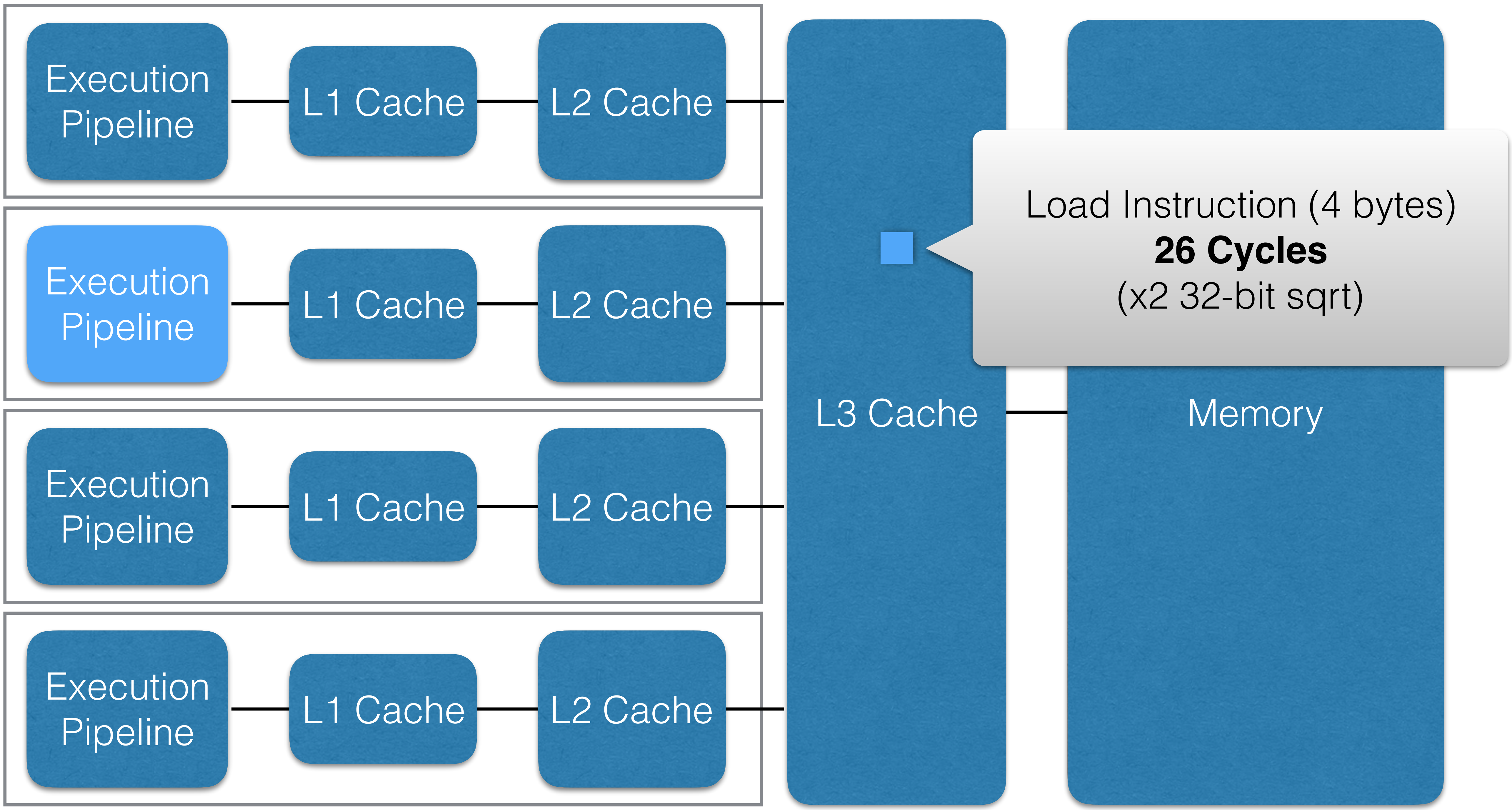
Memory Layout



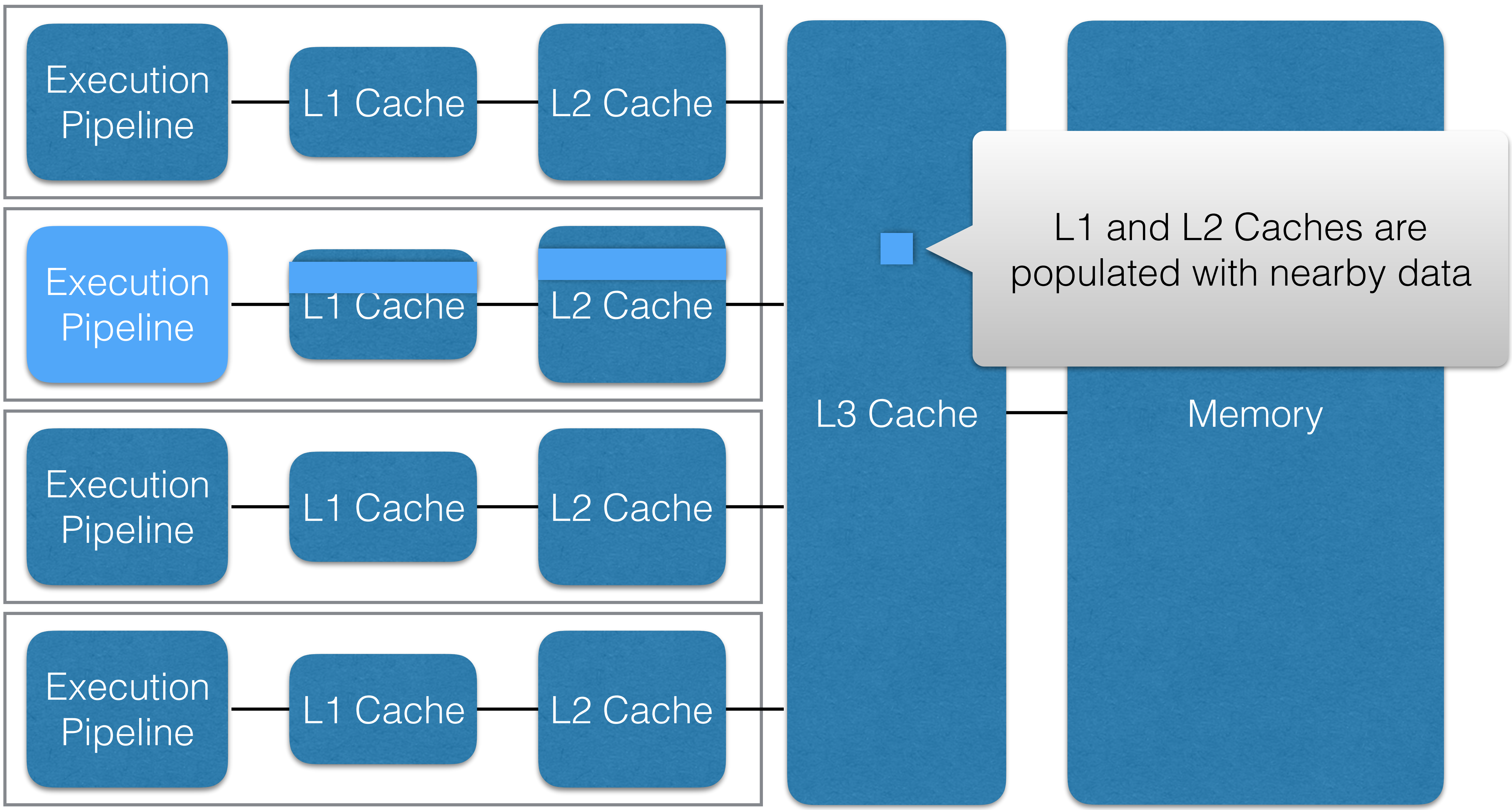
Memory Layout



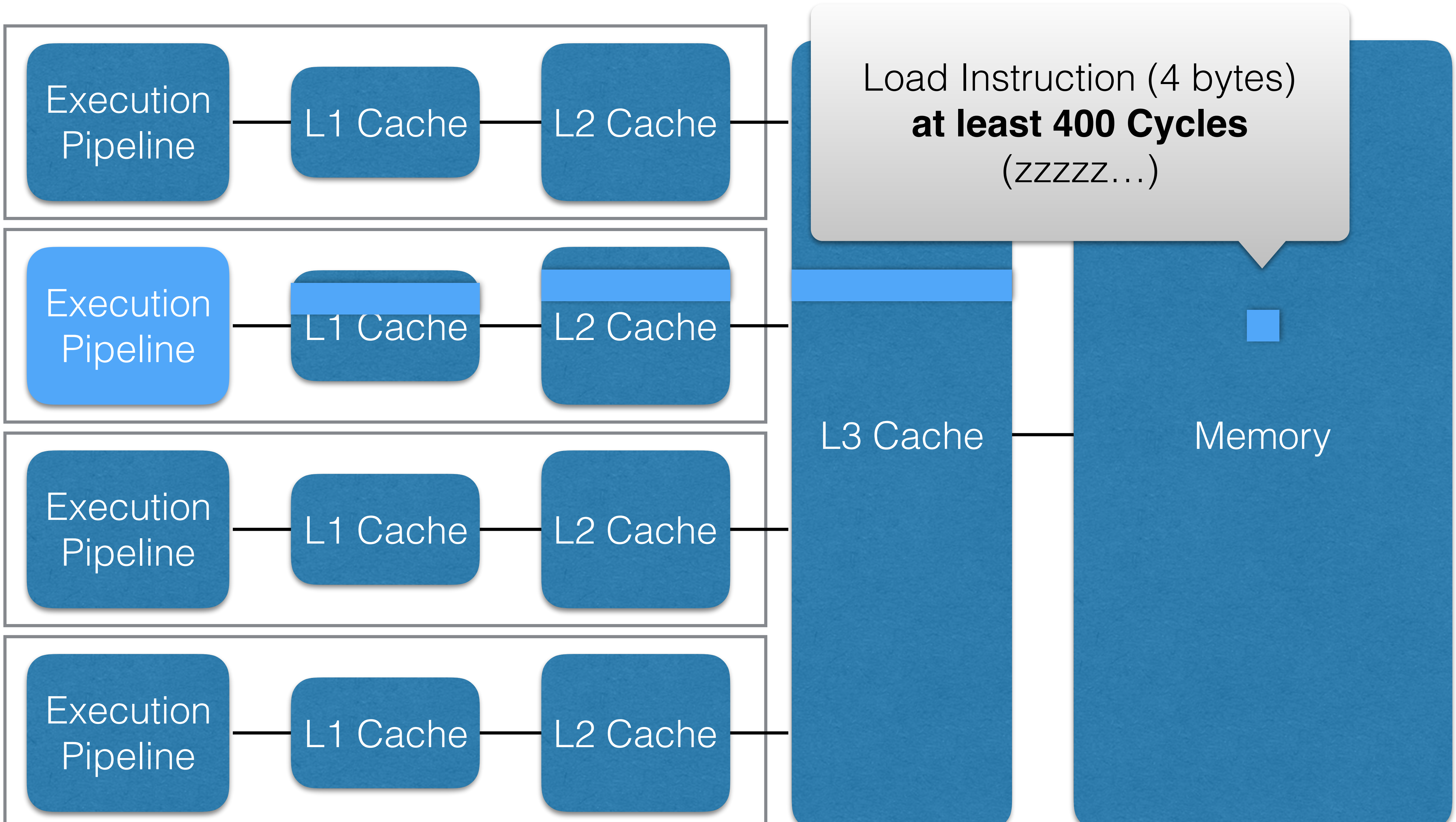
Memory Layout



Memory Layout



Memory Layout



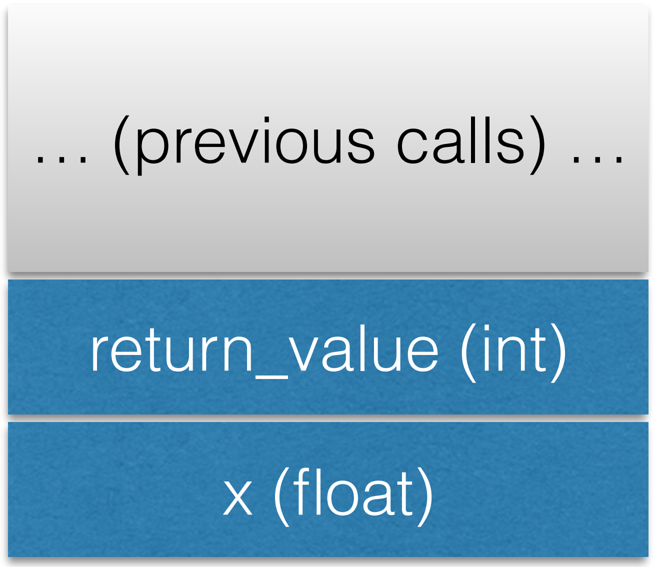
Moral of the Story

Try to keep the data you need as close as possible.

The Call Stack

- Stores info about the active functions and subroutines in a program.

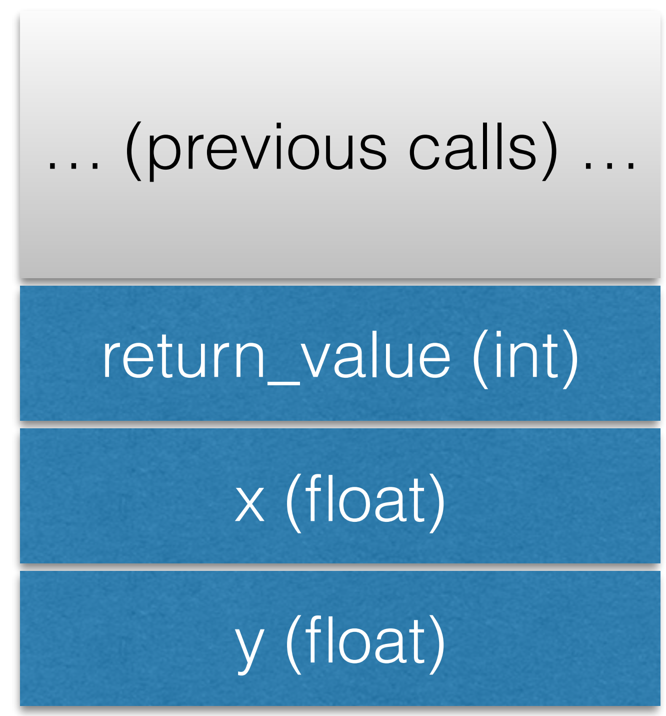
```
int foo(float x)
{
    float y = 2;
    float a = bar(x,y);
    int b = floor(a);
    return b;
}

float bar(float x, float y)
{
    double z = x + y;
    return z;
}
```



The Call Stack

- Stores info about the active functions and subroutines in a program.

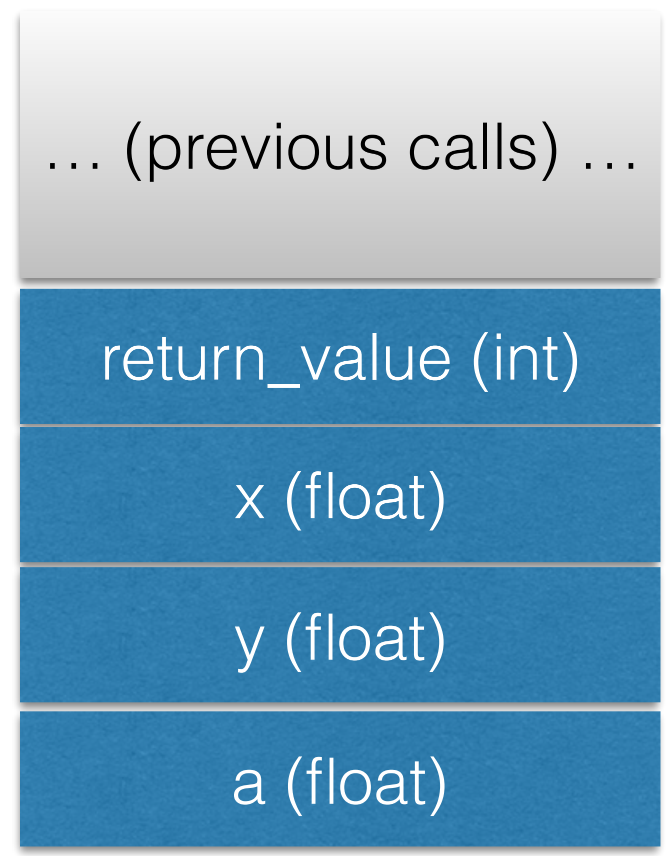
```
int foo(float x)
{
    float y = 2;
    float a = bar(x,y);
    int b = floor(a);
    return b;
}

float bar(float x, float y)
{
    double z = x + y;
    return z;
}
```



The Call Stack

- Stores info about the active functions and subroutines in a program.

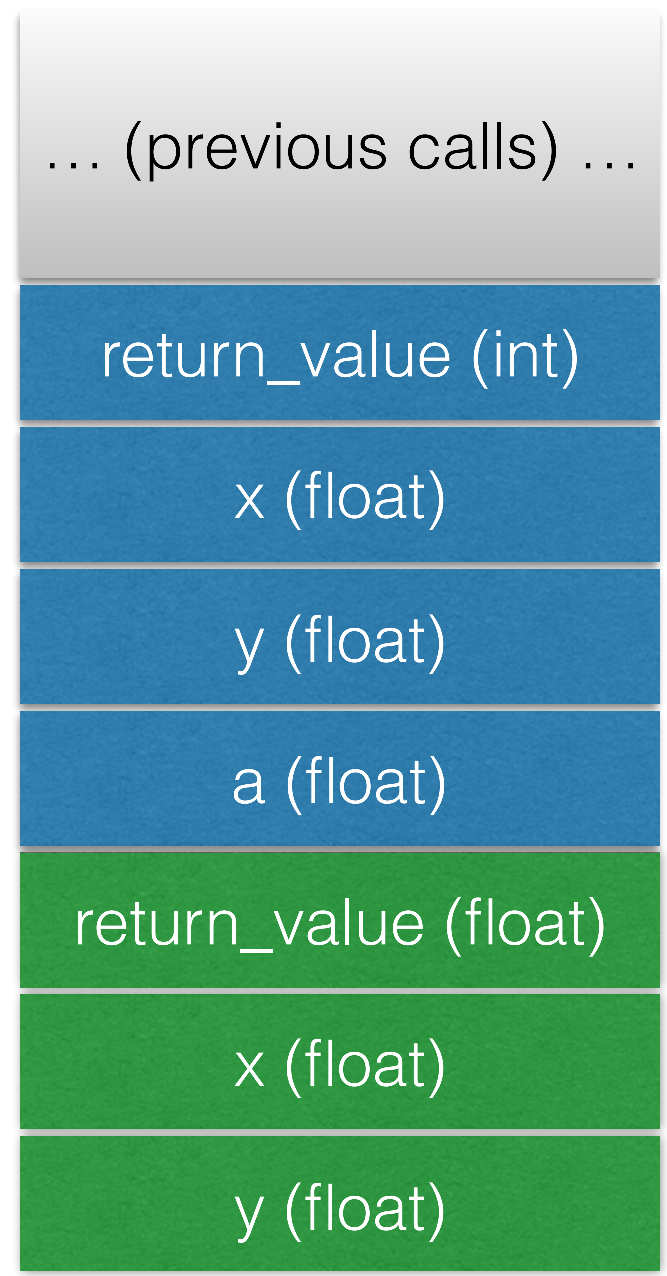
```
int foo(float x)
{
    float y = 2;
    float a = bar(x,y);
    int b = floor(a);
    return b;
}

float bar(float x, float y)
{
    double z = x + y;
    return z;
}
```



The Call Stack

- Stores info about the active functions and subroutines in a program.

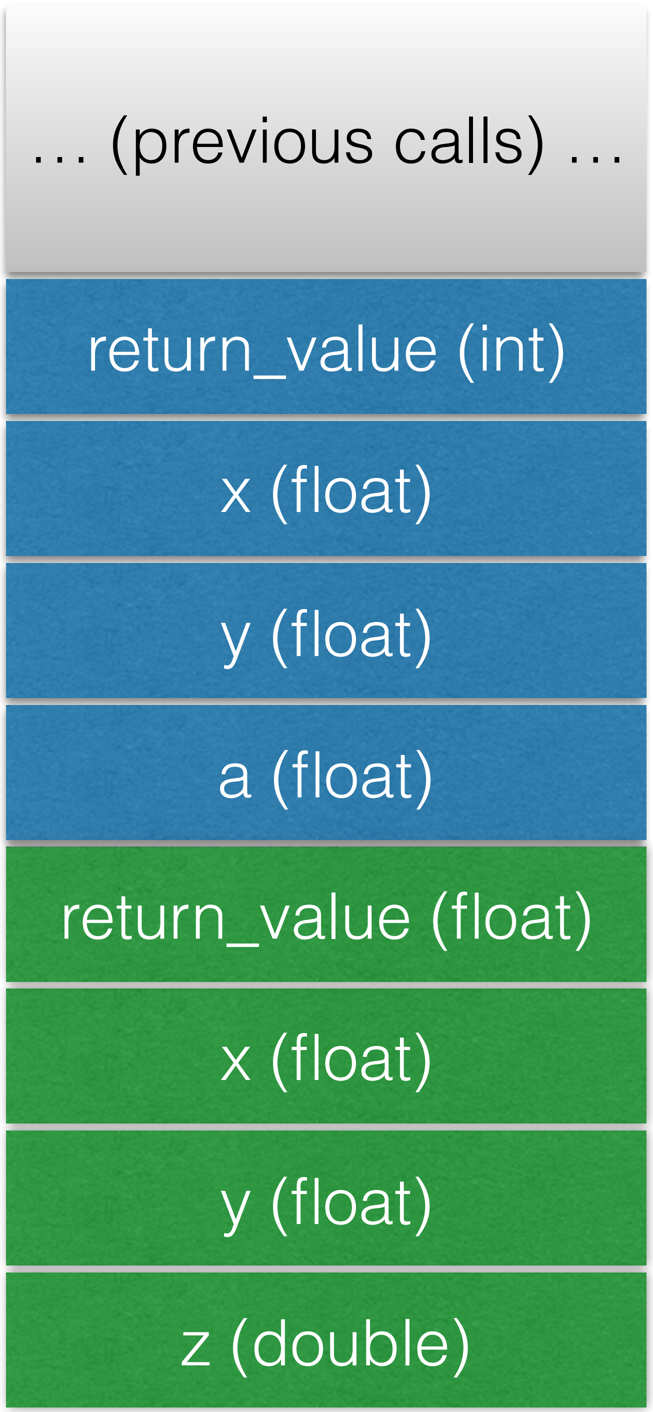
```
int foo(float x)
{
    float y = 2;
    float a = bar(x,y);
    int b = floor(a);
    return b;
}

float bar(float x, float y)
{
    double z = x + y;
    return z;
}
```



The Call Stack

- Stores info about the active functions and subroutines in a program.

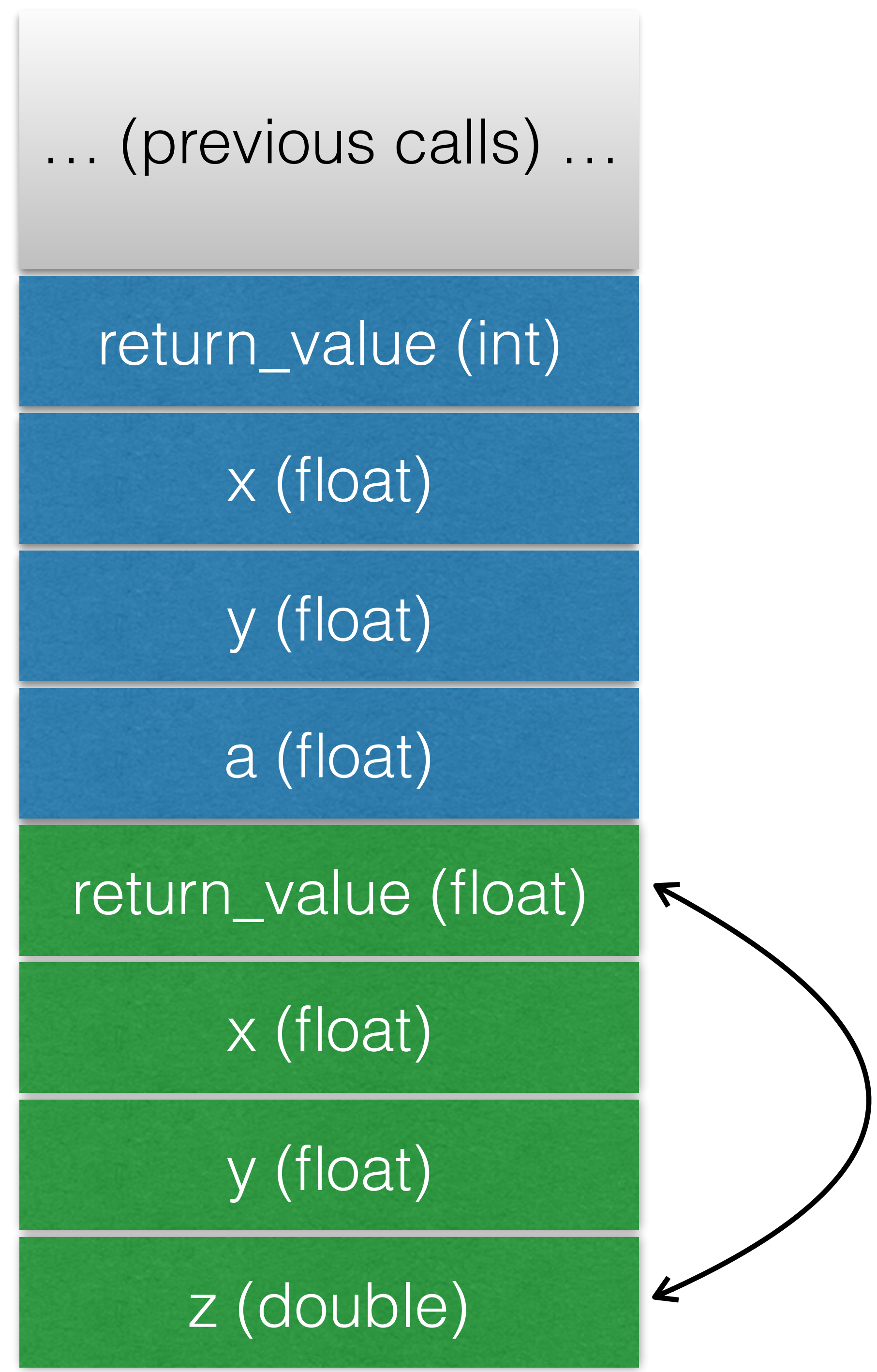
```
int foo(float x)
{
    float y = 2;
    float a = bar(x,y);
    int b = floor(a);
    return b;
}

float bar(float x, float y)
{
    double z = x + y;
    return z;
}
```



The Call Stack

- Stores info about the active functions and subroutines in a program.

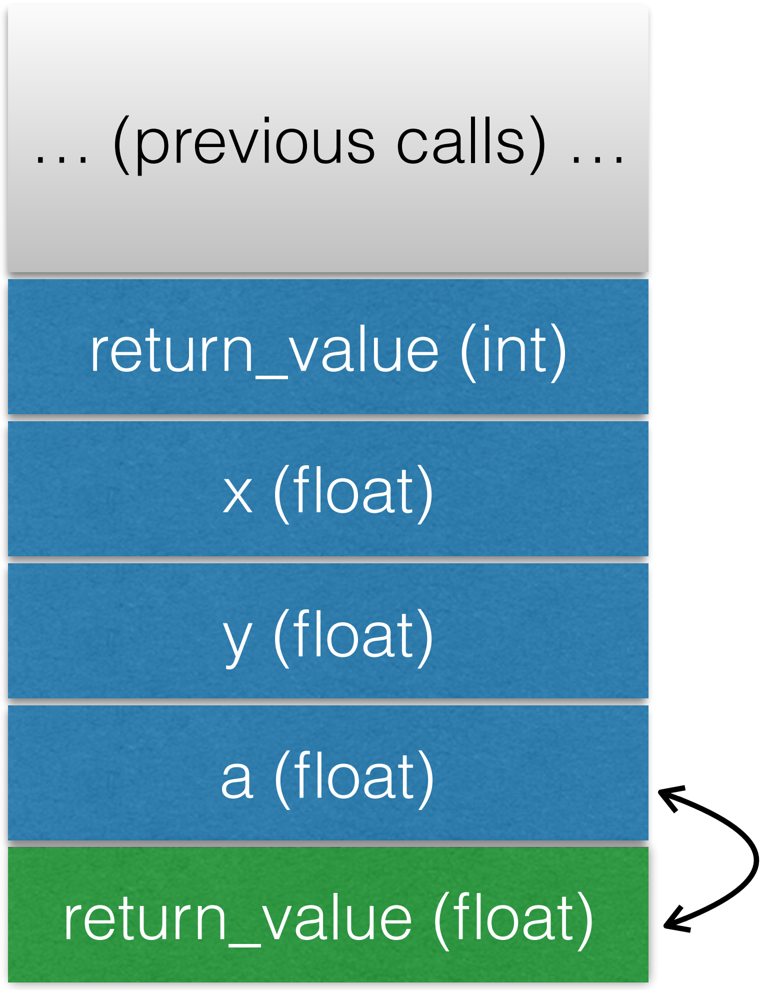
```
int foo(float x)
{
    float y = 2;
    float a = bar(x,y);
    int b = floor(a);
    return b;
}

float bar(float x, float y)
{
    double z = x + y;
    return z;
}
```



The Call Stack

- Stores info about the active functions and subroutines in a program.

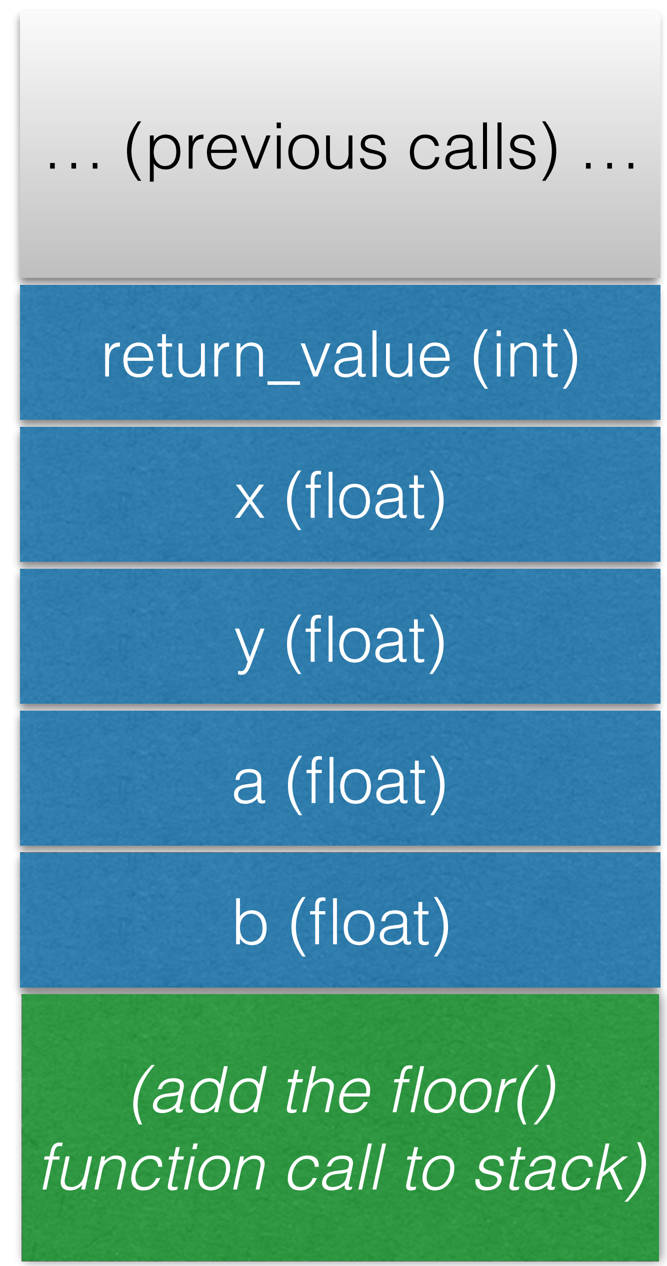
```
int foo(float x)
{
    float y = 2;
    float a = bar(x,y);
    int b = floor(a);
    return b;
}

float bar(float x, float y)
{
    double z = x + y;
    return z;
}
```



The Call Stack

- Stores info about the active functions and subroutines in a program.

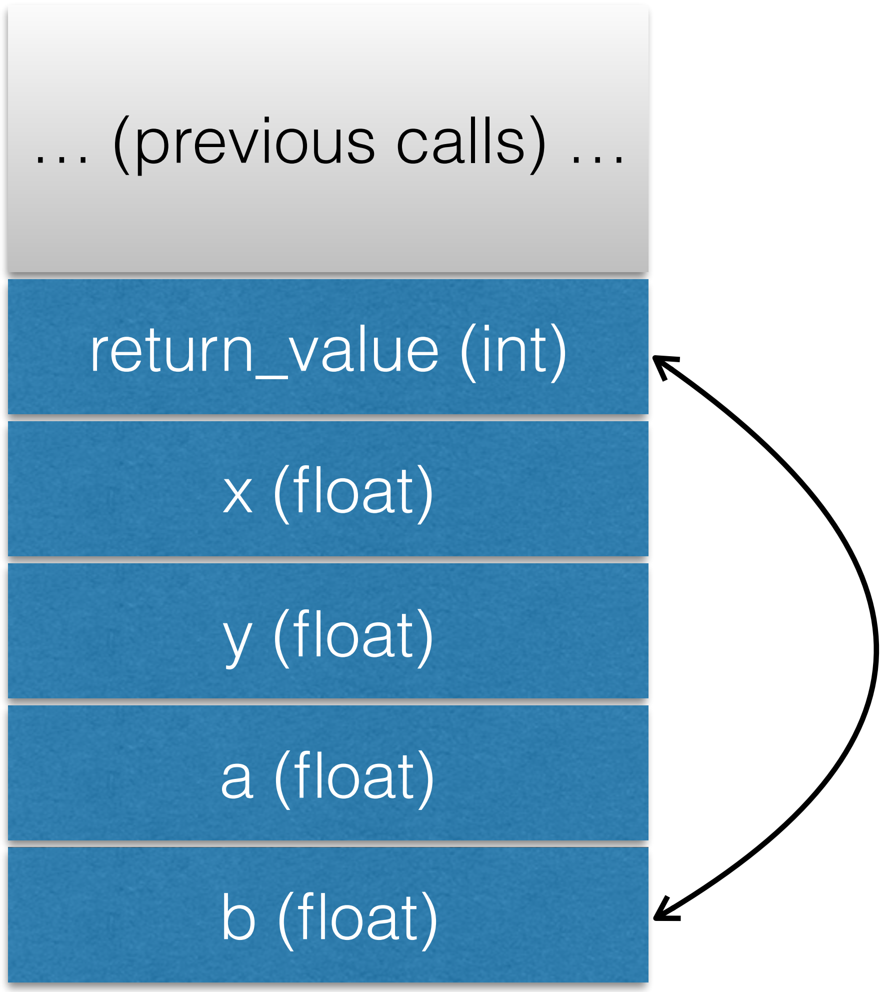
```
int foo(float x)
{
    float y = 2;
    float a = bar(x,y);
    int b = floor(a);
    return b;
}

float bar(float x, float y)
{
    double z = x + y;
    return z;
}
```



The Call Stack

- Stores info about the active functions and subroutines in a program.

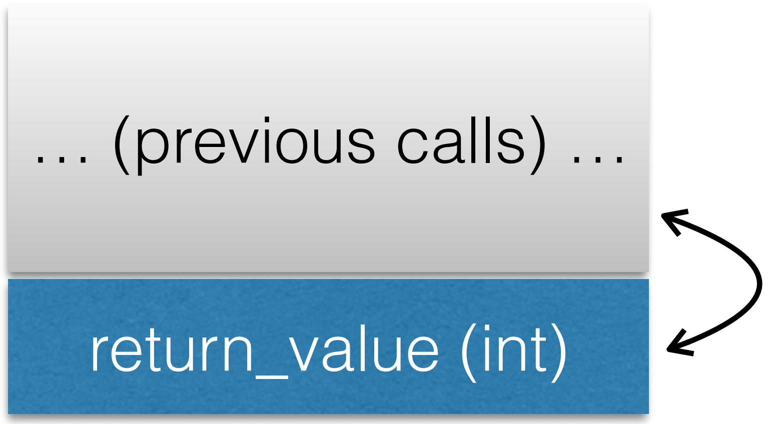
```
int foo(float x)
{
    float y = 2;
    float a = bar(x,y);
    int b = floor(a);
    return b;
}

float bar(float x, float y)
{
    double z = x + y;
    return z;
}
```



The Call Stack

- Stores info about the active functions and subroutines in a program.

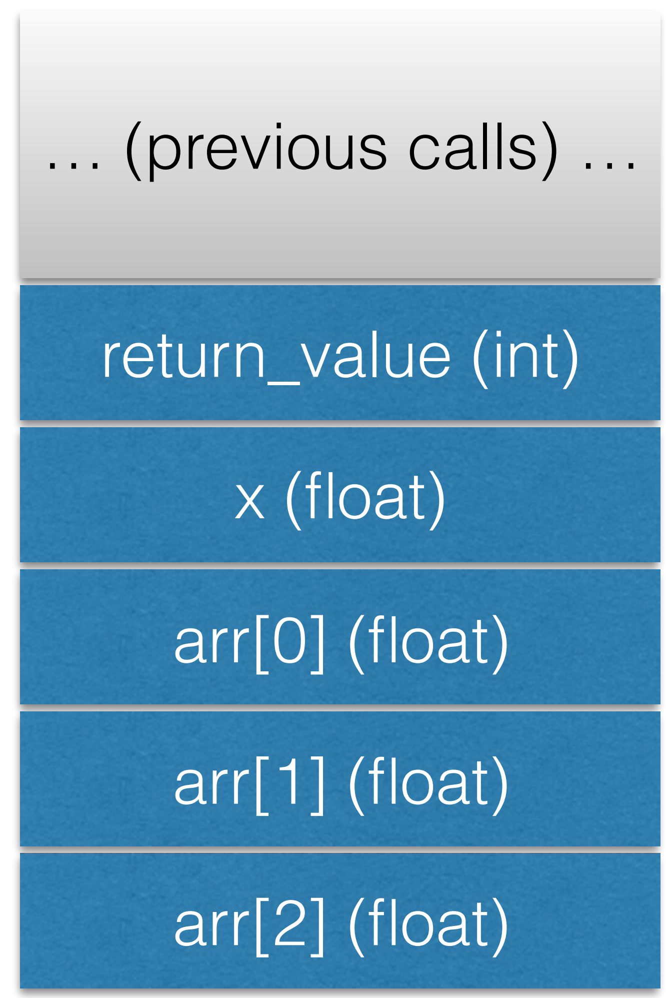
```
int foo(float x)
{
    float y = 2;
    float a = bar(x,y);
    int b = floor(a);
    return b;
}

float bar(float x, float y)
{
    double z = x + y;
    return z;
}
```



The Heap

- Allocating space for arrays on the stack:
  - if the size is known at compile time then the complier knows how much space to make on the stack
  - deleted once function returns

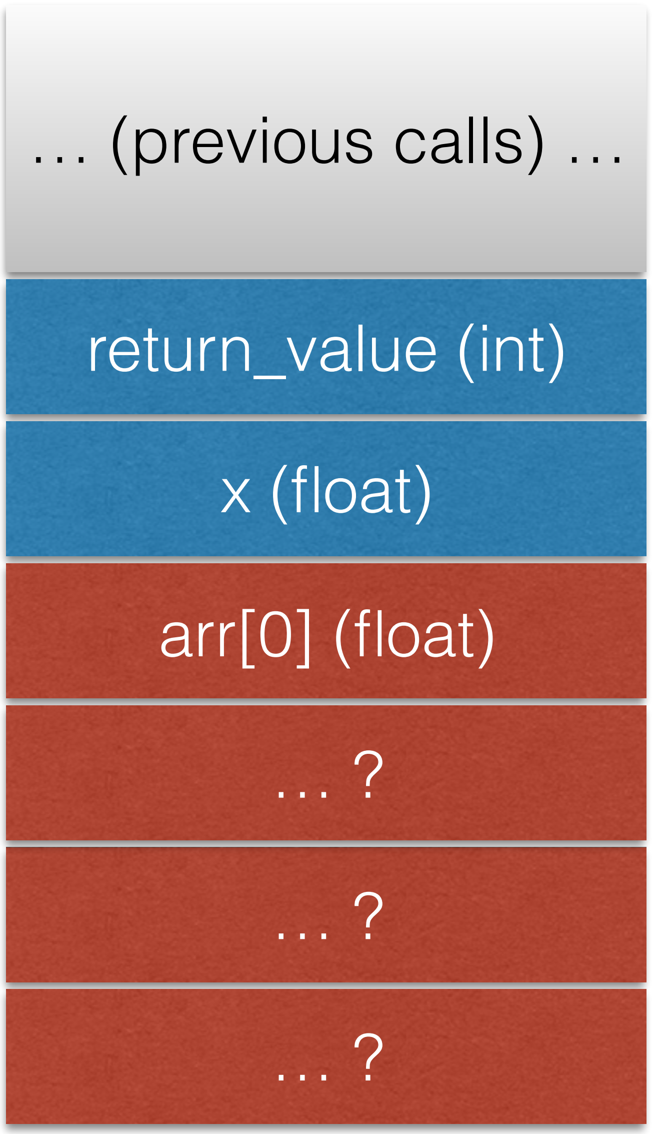
```
int foo(float x)
{
    float arr[3];
    ...
}
```



The Heap

- Allocating space for arrays on the stack:
    - but what if the array size is not known at compile time?

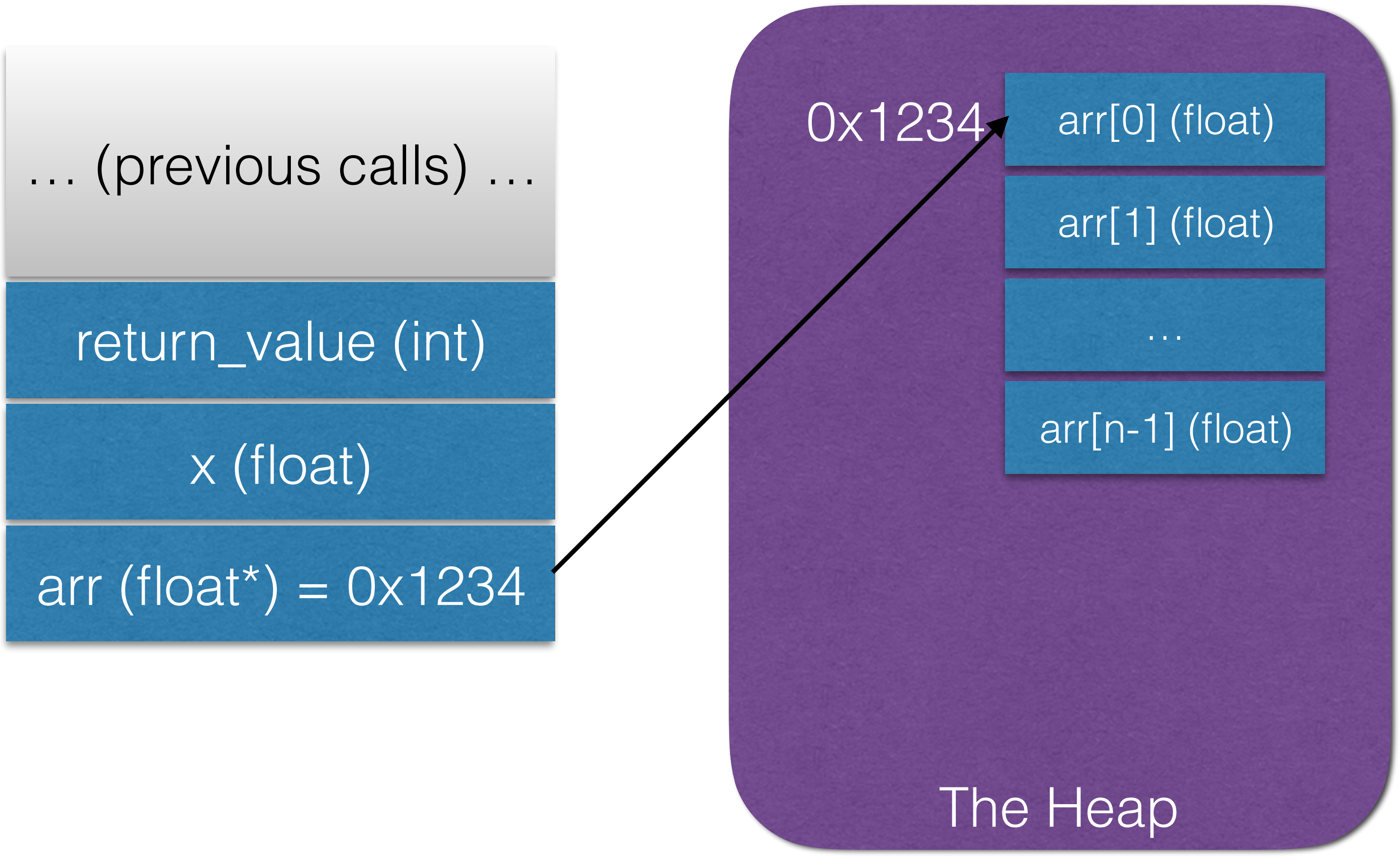
```
int foo(float x, size_t n)
{
    float arr[n]; // error
    ...
}
```



The Heap

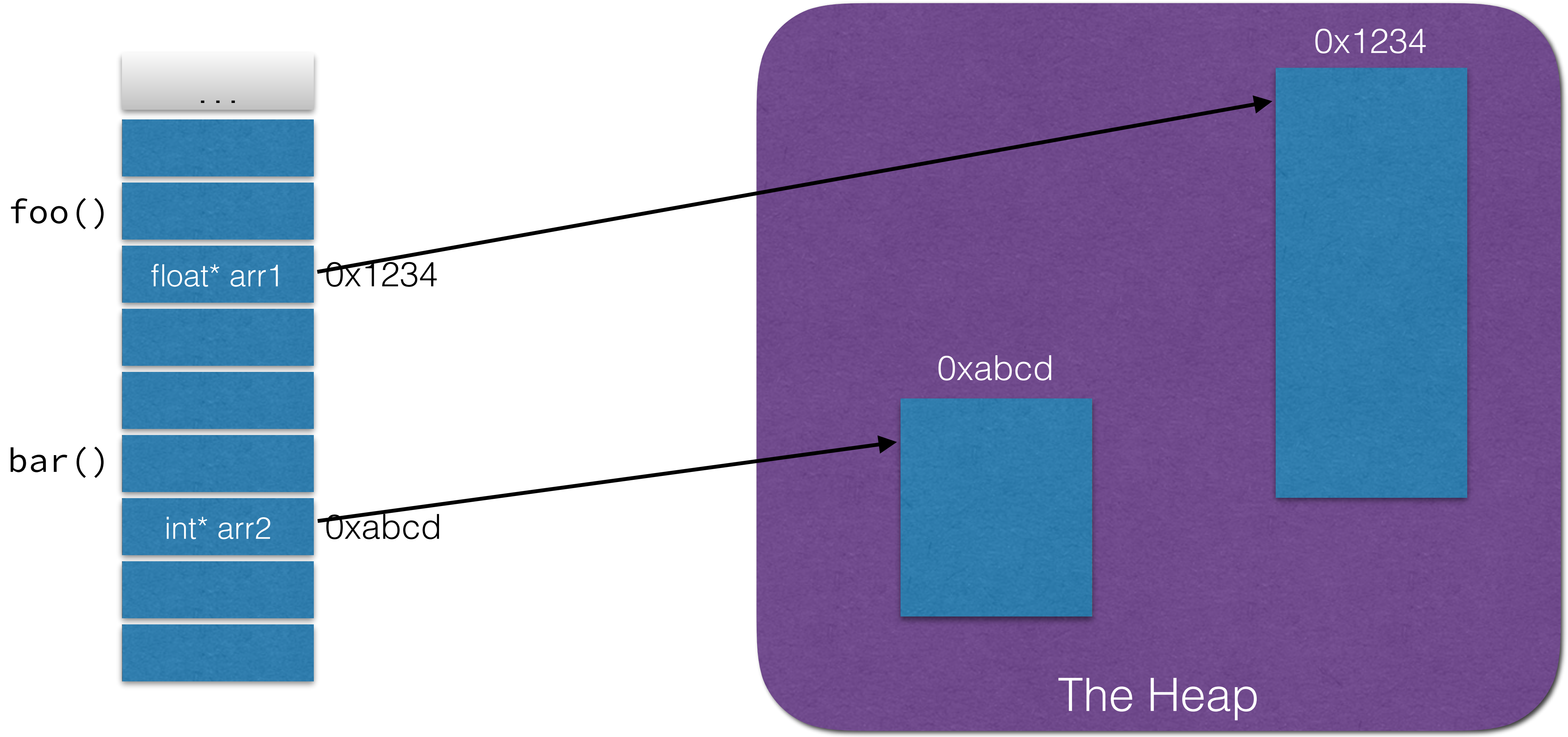
- Instead, program allocates memory separate area called *the heap*.
    - The stack contains the “address” of that location in memory
    - OS tries to find requested *contiguous* memory during runtime

```
int foo(float x, size_t n)
{
    float* arr = malloc(
      n*sizeof(float));
}
```



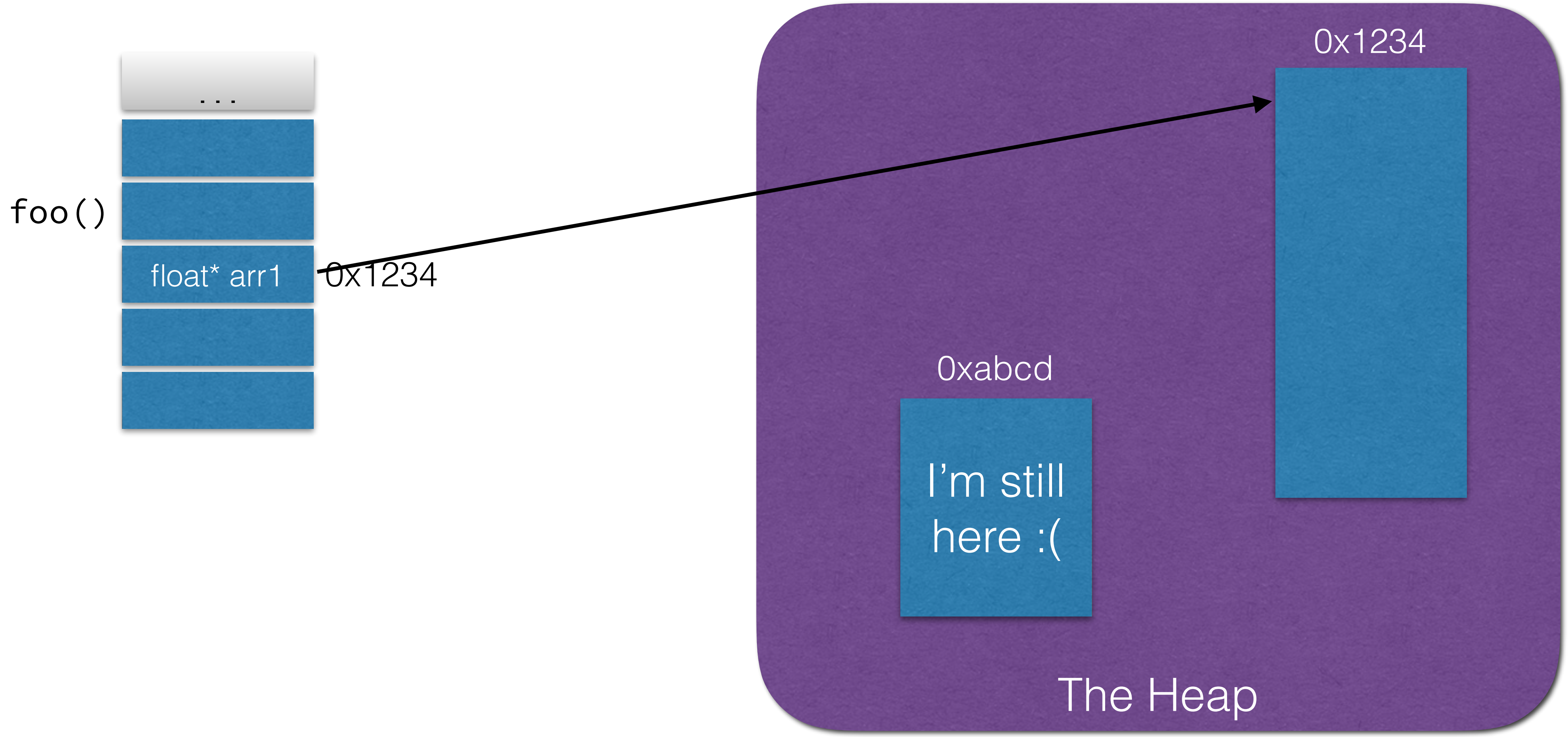
The Heap

- Caution: heap allocations can stick around after function calls (unlike stack allocations)
  - "Garbage collection" = automatic heap cleaning
  - C does not do garbage collecting - manually "free" heap allocations (Python does GC)



The Heap

- Caution: heap allocations can stick around after function calls (unlike stack allocations)
  - "Garbage collection" = automatic heap cleaning
  - C does not do garbage collecting - manually "free" heap allocations (Python does GC)



Stack and Heap

- Next week dive into C
    - spend almost two weeks on it —> plenty of time
- Memory management more direct than in Python

Demo

Navigating the stack in Python using `pdb`