LUND
UNIVERSITY

EDAP05: Concepts of Programming Languages

LECTURE 2: BACKGROUND

Christoph Reichenbach

Administrativa

- Course system is online now
  - Make sure to visit
  - We will sync against LADOK tomorrow
  - You can form a group, then select group slot preferences
  - After the deadline, we will assign you to groups
- Online discussions tbd: determined by poll today at (or shortly after) 18:00
- Starting today with a quick recap

Language Evaluation Summary

	Readability	Writability	Reliability
Simplicity	+	+	+
Orthogonality	+	+	+
Types	+	+	+
Syntax Design	+	+	+
Abstraction Support		+	+
Expressivity		+	+
Type Checking			+
Exception Handling			+
Restricted Aliasing			+

(this is Robert W. Sebesta, "Concepts of Programming Languages", Table 1.1)

- Separate dimension: **Cost**
- Alternative (more detailed) model: Green and Petre, "Cognitive Dimensions of Notation"

Restricted Aliasing

Java

```
public static <T> void
concat(List<T> lhs, List<T> rhs) {
    for (int i = 0; i < rhs.size(); i++) {
        lhs.add(rhs.get(i));
    }
}

concat(a, a);
```

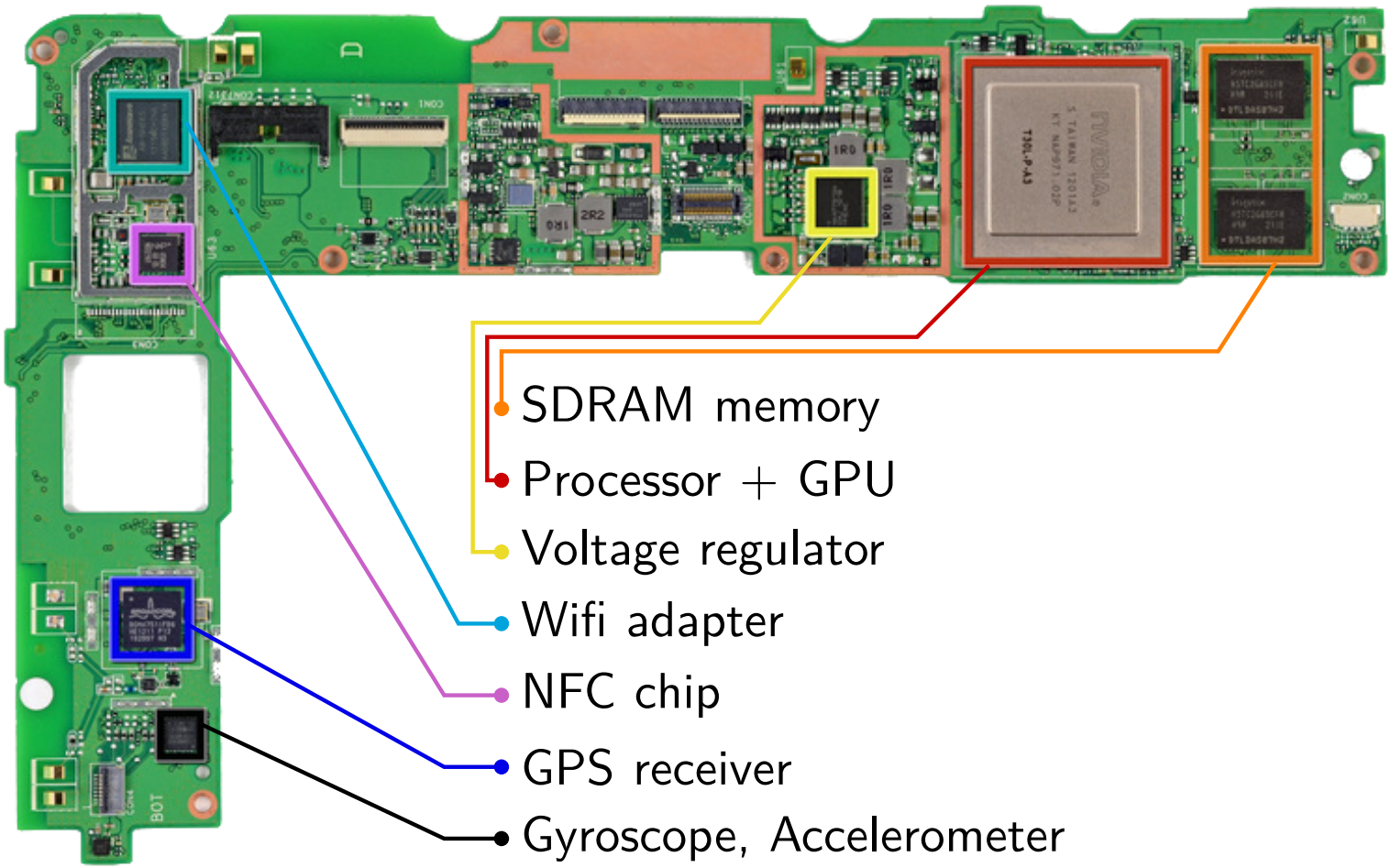
- Attach *rhs* to the end of *lhs*
- This code misbehaves (infinite loop) when passed the same list for both parameters
- **Aliasing**: two different names mean the same thing

Computers as Systems

- Programs run on CPU
- Use RAM
- Access hardware

. . . but how?

Nexus 7 Mainboard

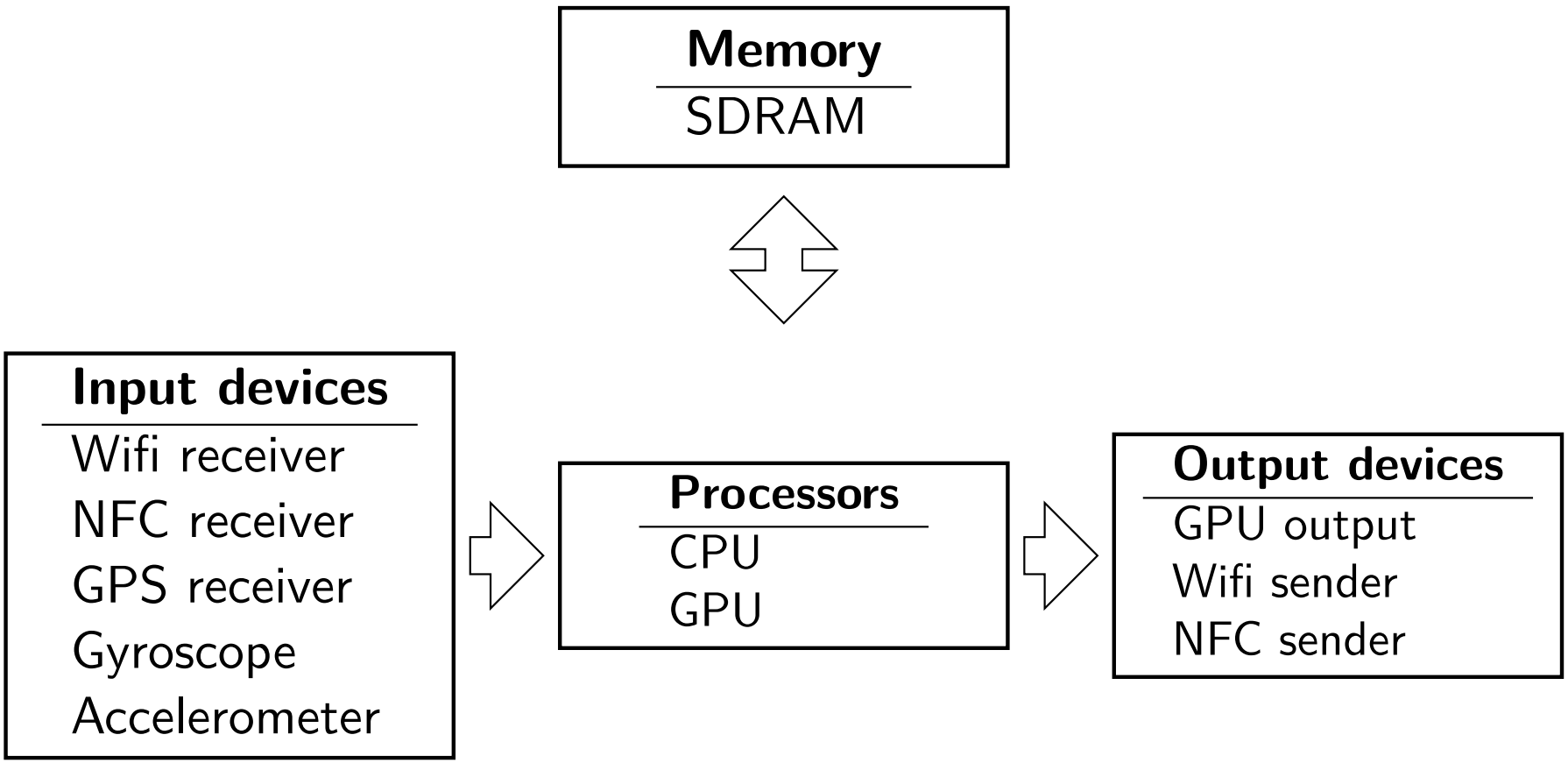


Nexus 7™Mainboard. (courtesy of ifixit.com).

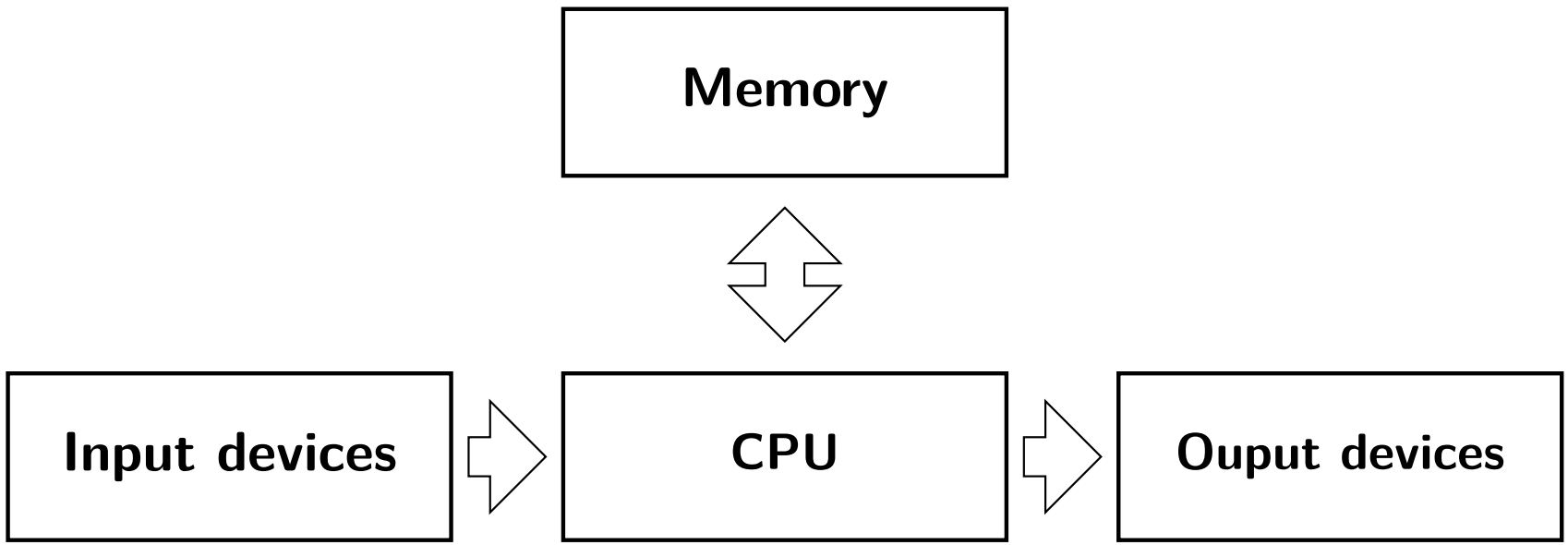
Functional Components (Nexus 7)

- SDRAM memory
- CPU
- GPU
  - GPU compute units
  - GPU graphics output
- Wifi adapter
  - Wifi sender
  - Wifi receiver
- NFC chip
  - NFC sender
  - NFC receiver
- GPS receiver
- Gyroscope
- Accelerometer

Computer Architecture

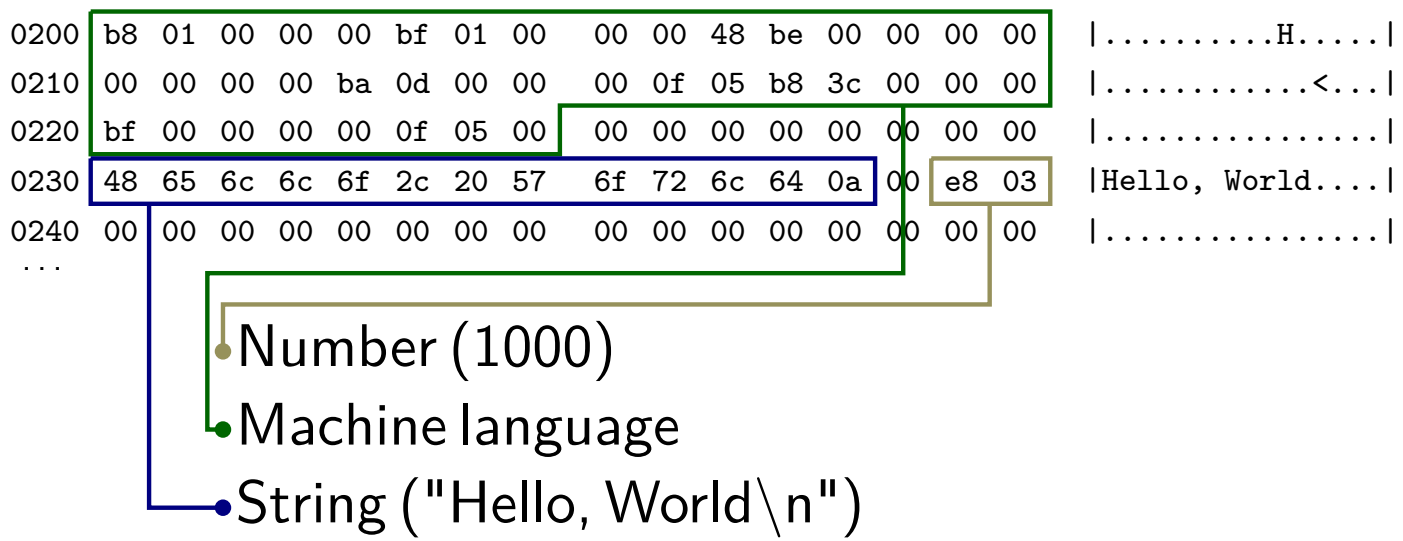


Computer Architecture: Abstracted



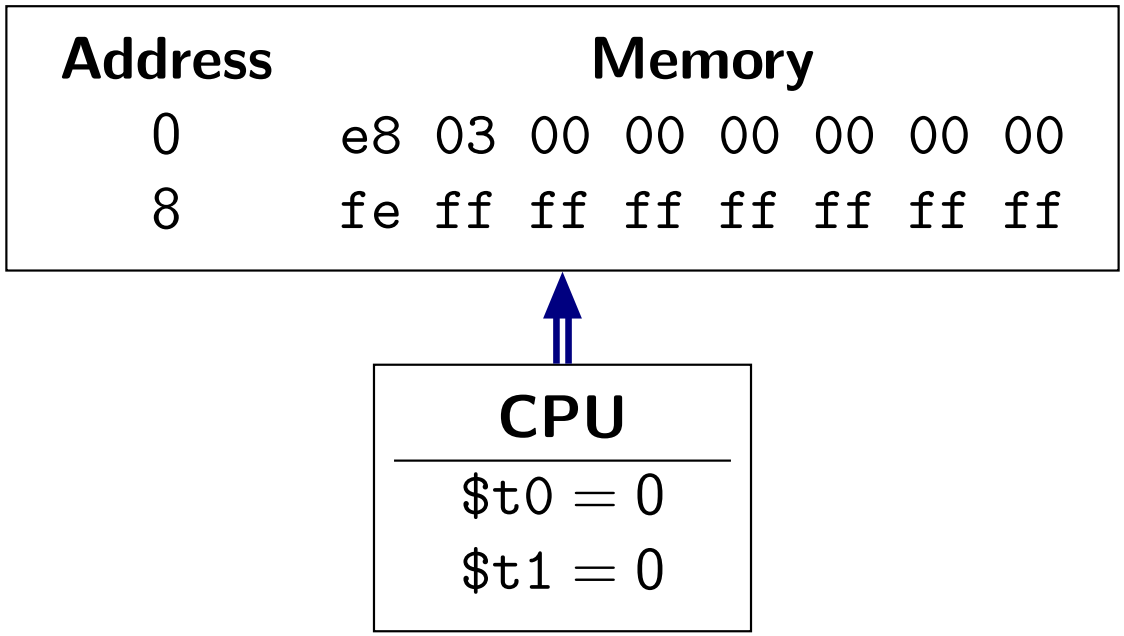
Data

```
user@host:~$ hexdump -C hello-world.o
 ...
0200  b8 01 00 00 00 bf 01 00   00 00 48 be 00 00 00 00  |..........H.....|
0210  00 00 00 00 ba 0d 00 00   00 0f 05 b8 3c 00 00 00  |............<...|
0220  bf 00 00 00 00 0f 05 00   00 00 00 00 00 00 00 00  |................|
0230  48 65 6c 6c 6f 2c 20 57   6f 72 6c 64 0a 00 e8 03  |Hello, World....|
0240  00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00  |................|
 ...
```



Memory can contain all sorts of data, often freely mixed

CPU + RAM Interaction



- RAM behaves like array: maps *addresses* to bytes
- To operate on memory:
  - CPU loads RAM contents into *registers* such as `$t0`, `$t1`
  - CPU operates on registers
  - CPU writes back registers into RAM
- Number of registers is very small, CPU can't do much without RAM!

Representing Numbers

- Numbers can be represented in a variety of ways
- Here: 64-bit little-endian two's complement numbers

In-Memory Representation	Hexadecimal	Decimal
00 00 00 00 00 00 00 00	0x0	0
01 00 00 00 00 00 00 00	0x1	1
10 00 00 00 00 00 00 00	0x10	16
0a 00 00 00 00 00 00 00	0xa	10
0f 00 00 00 00 00 00 00	0xf	15
00 01 00 00 00 00 00 00	0x100	256
ff ff ff ff ff ff ff ff	0xffffffffffffffff	−1
fe ff ff ff ff ff ff ff	0xfffffffffffffffe	−2

Summary

- All data is kept in RAM or in registers
  - RAM: lots of space, slow
  - Registers: very few, fast
- Code is data: CPU executes instructions from RAM
- Code can decide freely how to represent
  - Arrays
  - Data structures
  - Objects
  - Algebraic values

  ...
- Here, we work with 64-bit little-endian two's complement numbers

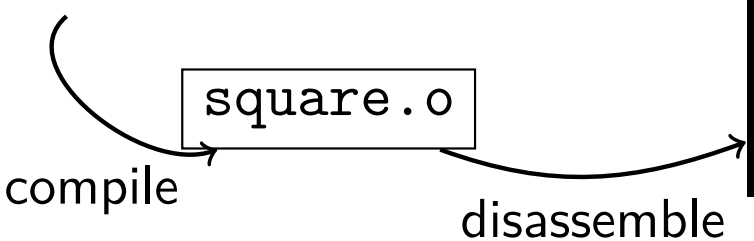
Demo

Summary

- **Machine Language**: The CPU's language
- **Assembly Language**: A "human-readable" encoding of machine language in text form
- Memory addresses:
  - Numbers!
  - Reference code, data
  - Point to different memory regions

C: square.c

```
int square(int x) {
  return x * x;
}
```




```
0000000000000000 <square>:
0:   0f af ff        imul edi,edi
3:   89 f8           mov eax,edi
5:   c3              ret
     Machine code    Assembly code
     (hexadecimal)
```

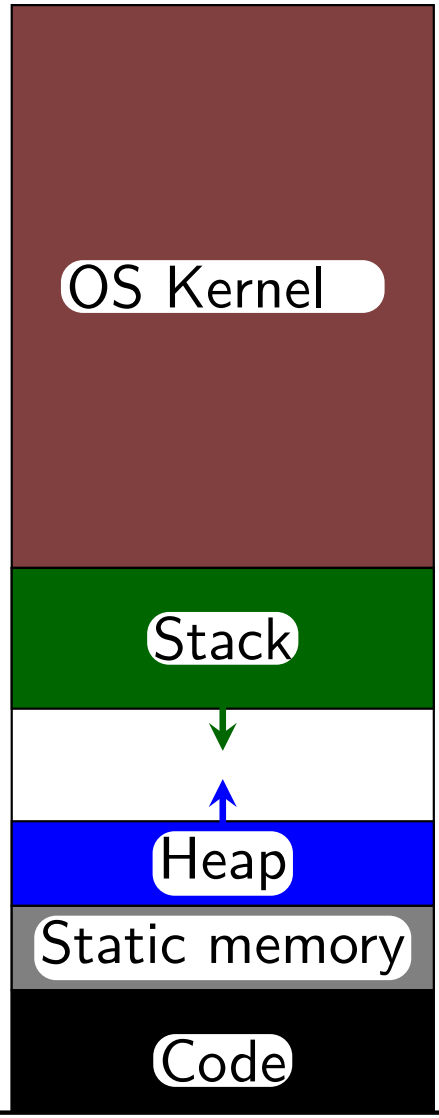
x86-64 memory addresses

- x86-64 uses 64-bit memory addresses
- Only lowest 48 bits are actually used
- At program start:
  - *Loader* allocates some addresses
  - Loads code, data into memory
  - Jumps into loaded code to start execution

Conventional memory layout in x86-64/Linux

Default allocation at program start:

- Operating system memory: not accessible to user-space code
- Stack: function calls, some temporary allocation
- Heap: temporary allocation
- Static memory: 'global' memory
- **Code** (also known as **text**): machine code



Layout requested by OS loader

Static Memory

- **Used for**:
  - Global variables (e.g., in C)
  - Constants (e.g., literal strings)
- **Size of region**:
  - fixed by loader

Stack Memory

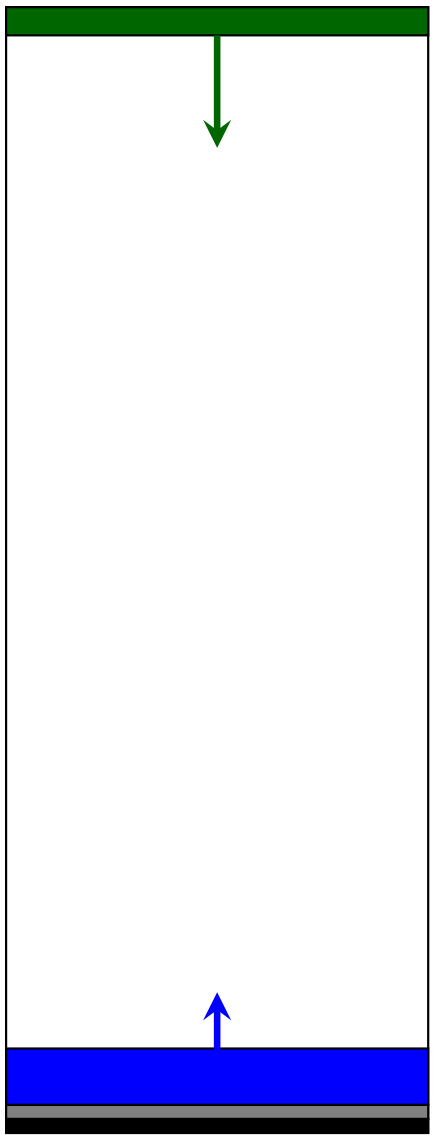
- **Used for**:
  - Local variables
  - Function calls, parameters
  - Enabling recursion
- **Size of region**:
  - On x86, stack begins at *top* of address space (by convention)
  - Grown automatically by operating system

Heap Memory

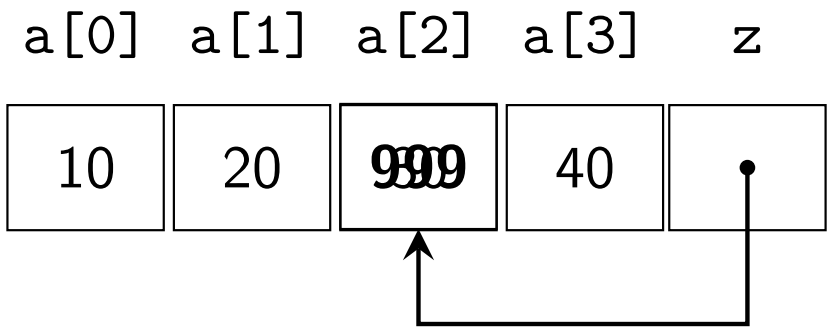
- **Used for**:
  - 'catch-all' when static/stack memory don't suffice
- **Region size**:
  - Arbitrary; grown on demand (explicit requests)
- **Access via**:
  - Pointer or reference variables (more later!)
- **Usage**: Program must *manage* heap:
  - Deallocate unused memory
  - Search for unused space on allocation
  - Grow heap (call operating system) if needed
  - Defragment memory (Garbage Collection)

Address Space Conventions

- Conventions simplify interaction with remainder of system
- Address space leaves *substantial* space for custom memory usage
  - Example here: we have mapped about 14 TiB
- Programs can freely allocate addresses for their own purposes
- Address space used e.g. by:
  - File access
  - Dynamic library loader
  - Threads



Pointers as in C, C++



- `int *z`
  z is a *pointer to* *`int`*
- `z = &a[2]`
  Address operator: `z` takes address of `a[2]`
- `*z`
  Dereference operator: accesses `int` value at memory location, e.g. to write
  `*z = 999`

Addresses are Just Numbers

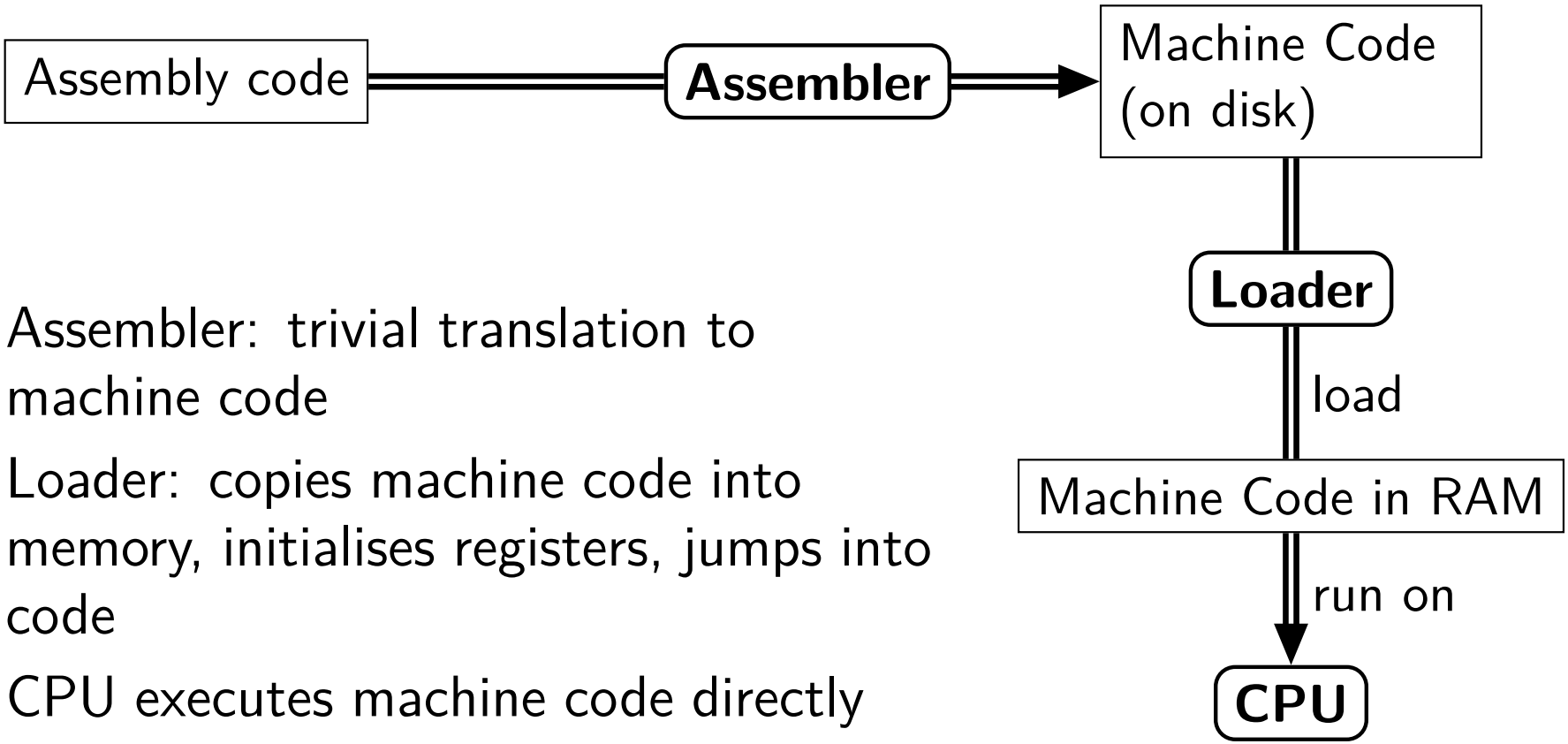
```
int a[3] = {10, 20, 30};

main() {
  int *a_ptr = &a[0];   // take address of a[0]
  a_ptr = a_ptr + 1;    // now: a_ptr = &a[1]
  *a_ptr = 0;           // now: a = {10, 0, 30}
}
```

- In systems languages like C, we can use RAM like a "map" / associative array
- Addresses are "just" a kind of number
- **Beware**: easy to break your own code in subtle ways...

Summary

- **Memory addresses** are numbers, indices into **address space**
- Address space split up into *regions*:
- Conventional regions (mostly pre-allocated by loader):
  - **Code** (‘`.text`’): executable code
  - **Static memory**: fixed-size read/write memory
  - **Stack**: dynamically FILO memory
    - Grows downwards on x86-64
  - **Heap**: catch-all
    - Explicit kernel requests needed to allocate, grow
    - Used by `malloc` (C), `new` (C++, Java, ...)

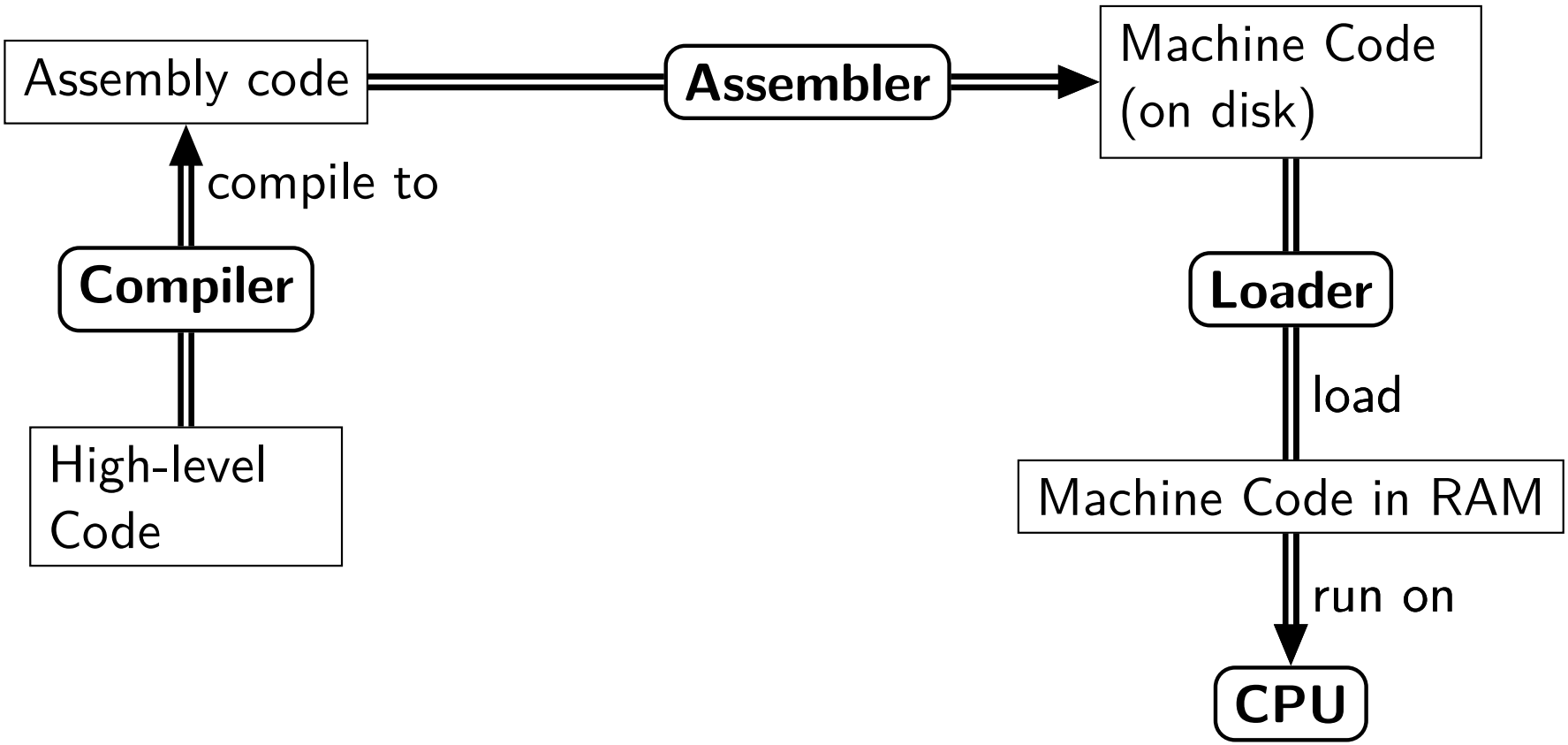
Program Execution



- Assembler: trivial translation to machine code
- Loader: copies machine code into memory, initialises registers, jumps into code
- CPU executes machine code directly

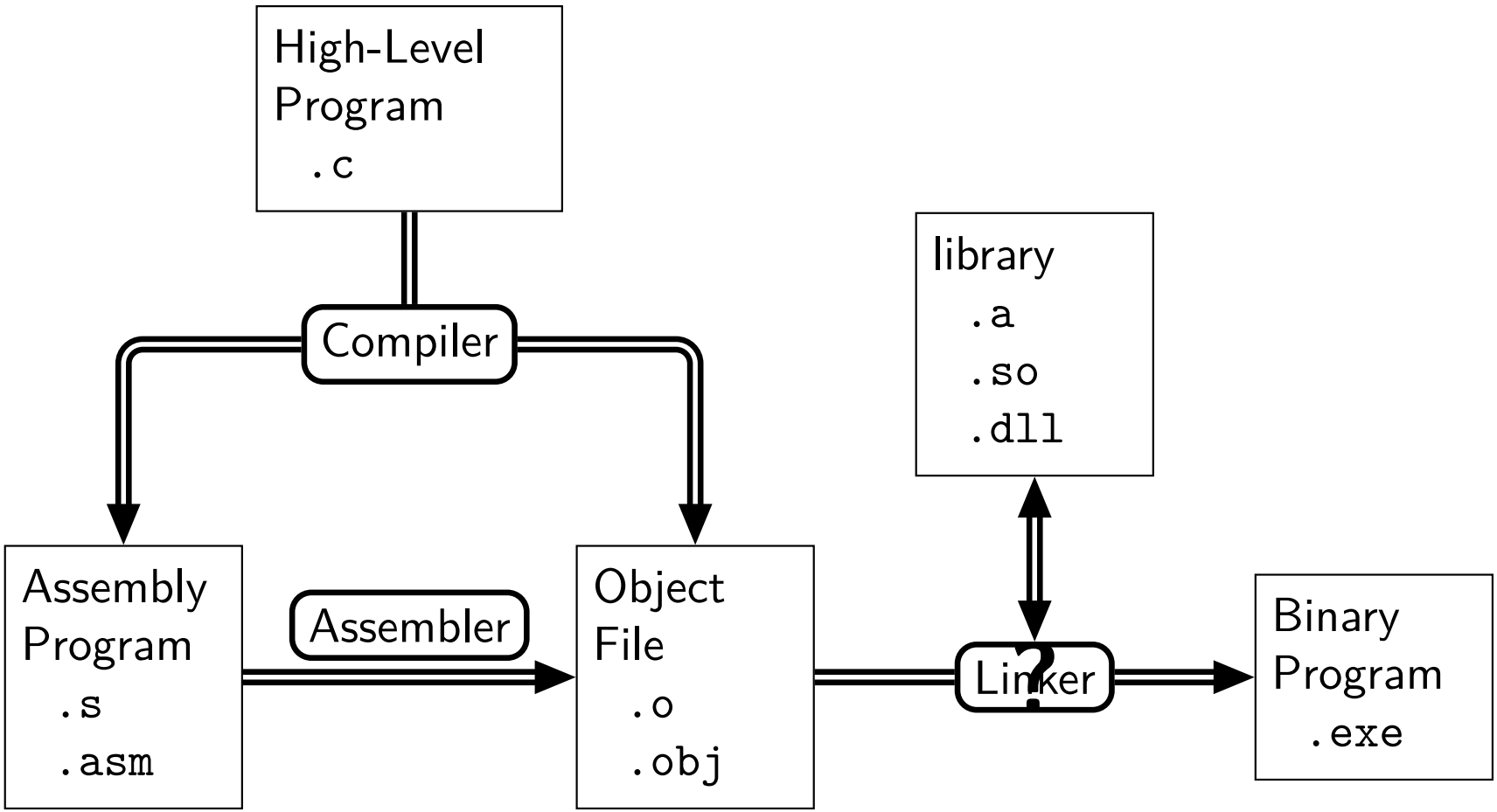
How about languages that the CPU can't execute directly?

Compilation



Examples: C, C++, SML, Haskell, FORTRAN, ...

Compiling and Linking in C

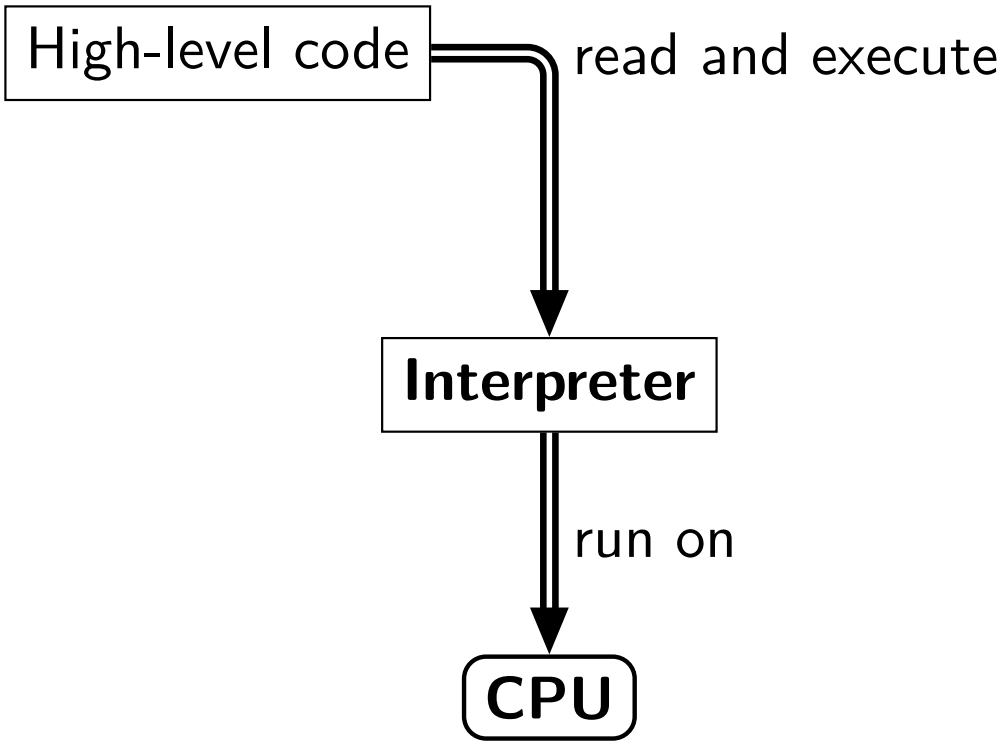


Binary program is machine code, can be run by CPU

Separate vs. Whole-Program Compilation

- *Separate compilation*:
  - Most compiled languages allow compiling components/libraries separately from each other
  - Allows us to avoid compiling code that hasn't changed
  - **Linker** combines these components
    - Often "hidden" in compiler and/or run-time system
- *Whole-program compilation*:
  - Compile everything at once
  - Can generate faster code
  - Can generate better error reports
  - Scales poorly
  - Makes **whole-program assumption** that we know all code that we need to execute

Interpretation



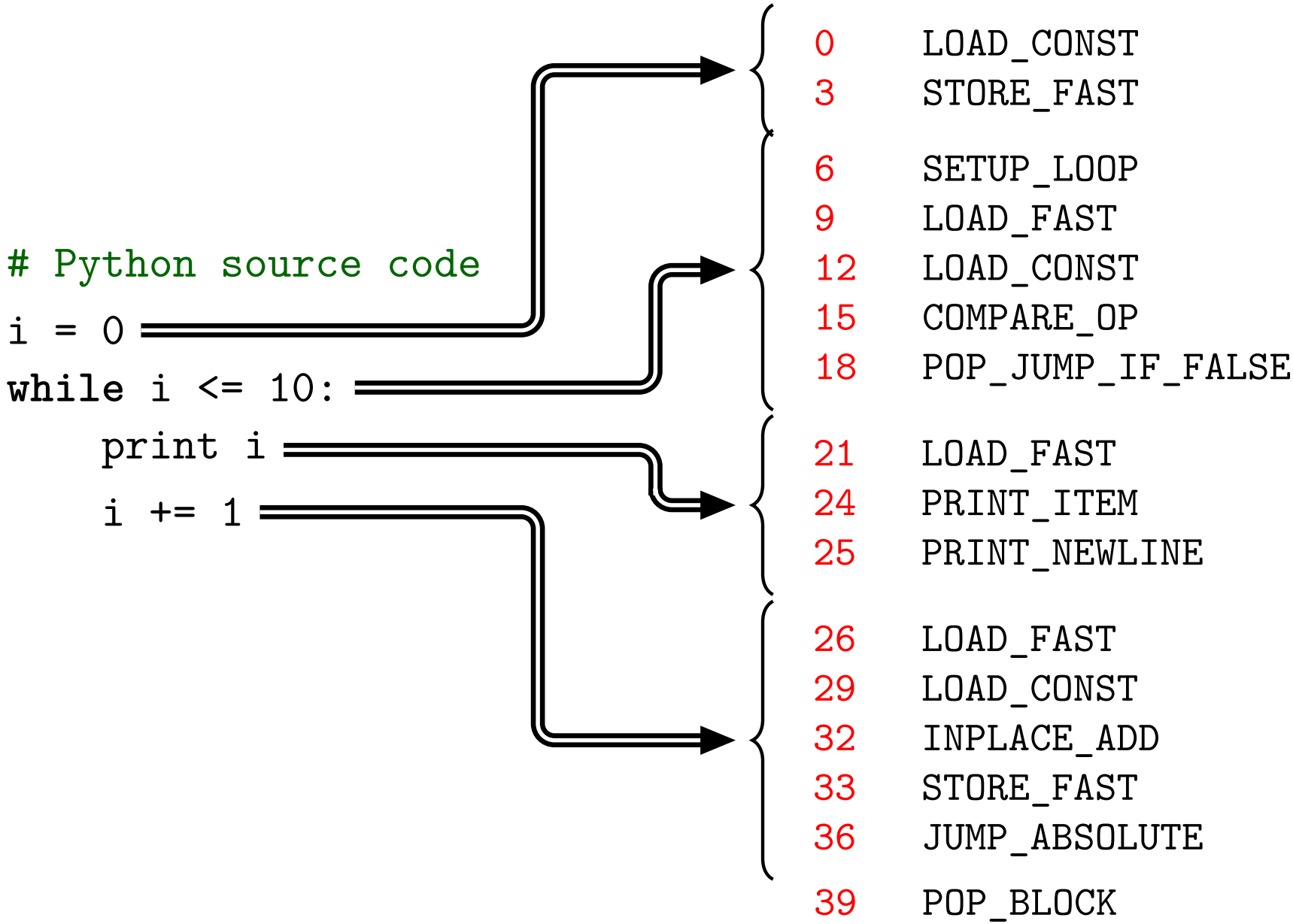
- Interpreter reads high-level code, then alternates:
  - Figure out next command
  - Execute command

Examples: Perl, Ruby, Bash, AWK, . . .

Hybrid Implementation

- Compilers compile to:
  - Machine code
  - **Bytecode** (Java, Python, C#, . . .)
  - Other high-level languages (“transpilers”)
- **Hybrid implementations** of languages use
  - One or more compilers
  - One or more intepreter

Example: CPython ('normal' Python)



Python execution (simplified)

- *Compile*: Python source code to **bytecode**
- *Interpret*:
  - Load next Python **bytecode** operation
  - Which instruction is it? Jump to specialised code that knows how to execute the instruction:
    - Load parameters to operation
    - Perform operation
    - Continue to next operation

Executing e.g. an addition in CPython takes dozens of assembly instructions

Comparison: Compilation vs Interpretation

Property	**Interpretation**	**Compilation**
Execution performance	slow	**fast**
Turnaround	**fast**	slow (compile & link)
Language flexibility	**high**	limited*

$\star$) Compiler Optimisation $\lightning$ Flexibility

Dynamic Compilation

- Idea: compile code *while executing*
- In theory: best of both worlds
- Practice:
  - Difficult to build
  - Memory usage tends to increase
  - Performance can be higher than pre-compiled code

Examples: Java, Scala, C#, JavaScript, ...

Summary

- Languages implemented via:
  - stand-alone **Compiler**
  - **Interpreter**
  - **Hybrid Implementation**
    - Part compiler, part interpreter (e.g., Python)
    - May include: **Dynamic Compiler** (e.g., JVM)
- Trade-off between:
  - Language flexibility
  - CPU time / RAM usage
- Languages may have multiple implementations
  - Example: CPython vs. Jython
  - `gcc` vs. `llvm/clang` vs. MSVC

Compilers and Run-Time Systems

Separate language implementations into:

- **Static** components:
  - Used *before program executes*
  - Examples:
    - Assemblers, linkers
    - Compilers: `gcc`, `javac`, `rustc`, `go build`
- **Dynamic** components:
  - Used *while program executes*
  - Form the **Run-Time System**
  - Examples:
    - Memory management (automatic or otherwise)
    - *dynamic* loading (e.g., `dlopen` in POSIX/C)
    - *dynamic* compilation (e.g., `eval` in JavaScript, Python)
    - *adaptive optimisation* (e.g., Java Virtual Machine)

Summary

- Languages $\neq$ language implementations
- Implementation types:
  - **Interpreter**, **Compiler**, **Hybrid Implementation**
  - Hybrid implementation can use **Dynamic Compilation**
  - Language implementation components:
    - **Static** (e.g., compilers): run before program runs
    - **Dynamic** (part of the **run-time system**): run while program runs
- **Linking**: connecting separately compiled program parts
- **Whole-Program Assumption**: we compile everything at once ($\Longrightarrow$ no linker needed)

Next Week

- Syntax
- Variables, Binding, Scope
- Semantics
- Basic Expressions
- Primitive Types

Read the listed material, bring your questions!