

# Compiled and interpreted languages

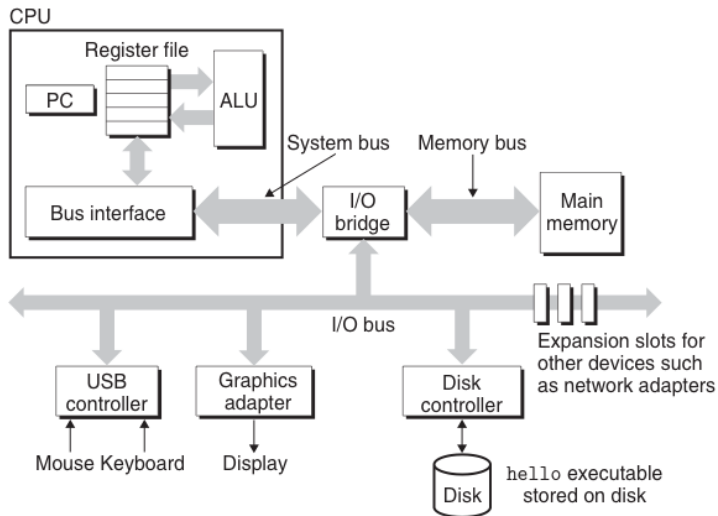
Troels Henriksen

How computers execute code

Compiled and interpreted languages

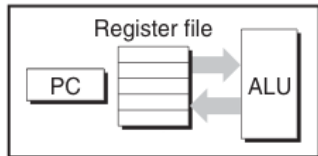
Tombstone diagrams

# The role of the processor



- Central processing unit (CPU).
- Reads data from memory, operates on it, writes it back.
- Also sends commands to other parts of the computer (e.g. IO).

# How the CPU works—conceptually and physically



- Data is stored in a small set of fixed-size *registers*.
  - ▶ Imagine a fixed collection of named variables.
  - ▶ x86-64 has 16 registers, each store 64 bits.
- Register contents sent to *Arithmetic-Logical Unit* to perform small operations.
  - ▶ “Interpret these two register contents as integers and add the second to the first.”
  - ▶ Others instructions copy data between registers and memory.
  - ▶ *Instructions modify the machine state.*

do forever:

```
instr = memory[PC]           # Fetch instruction
executeInstruction(instr)     # Execute instruction
PC += 1                      # Go to next instruction
```

- *Program counter* (PC) stores address of currently executing instruction.
- Implement control flow by manipulating PC.

## x86-64 registers

- Imagine programming where these are all the variables you have available.
- No function parameters either.
- x86-64 assembly for adding numbers stored in registers `r8`, `r9`, `r10`:

```
addq %r8, %r9 ; r8 += r9
addq %r8, %r10 ; r8 += r10
```

64-bit register	Bytes 0-3	Bytes 0-1	Bytes 0
%rax	%eax	%ax	%al
%rcx	%ecx	%cx	%cl
%rdx	%edx	%dx	%dl
%rbx	%ebx	%bx	%bl
%rsi	%esi	%si	%sil
%rdi	%edi	%di	%dil
%rsp	%esp	%sp	%spl
%rbp	%ebp	%bp	%bpl
%r8	%r8d	%r8w	%r8b
%r9	%r9d	%r9w	%r9b
%r10	%r10d	%r10w	%r10b
%r11	%r11d	%r11w	%r11b
%r12	%r12d	%r12w	%r12b
%r13	%r13d	%r13w	%r13b
%r14	%r14d	%r14w	%r14b
%r15	%r15d	%r15w	%r15b

# x86-64 instructions

There are thousands of x86-64 instructions.<sup>1</sup>

- All are encoded as bits and can be interpreted as numbers.
- Vary in size, but are always a whole number of bytes.

Instruction	Encoding in hex
<code>addq %r8, %r9</code>	4D 01 C8
<code>addq %r8, %r10</code>	4D 01 D0
<code>movq 0(%r9), %r8</code>	4D 8B 41 F8

- Different architectures have *different instructions*, *different registers*, and *different encodings*.
- Also two different syntaxes:
  - ▶ AT&T syntax (used for these slides, generated by default by GCC).
  - ▶ Intel syntax (probably more common among real assembly programmers).

<sup>1</sup>[https:](https://stefanheule.com/blog/how-many-x86-64-instructions-are-there-anyway/)

# Structured control flow as jumps

- Some instructions modify the program counter.
- These are used to implement control flow.

## C code

```
if (x < 0) {  
    x = 0;  
}
```

## x86-64 assembly

```
cmpq $0, %r8    ; Compare 0 and r8  
jns  .L2        ; Jump if r8 is greater  
movq $0, %r8    ; Set r8 to 0  
.L2:            ; Label  
...
```

- Instructions like `jns` set PC, which changes which instruction is read next.
- Labels like `L2` are just for humans.
  - ▶ The actual encoding of `jns` contains a byte offset to be added to PC.

# Loops as jumps

## C code

```
while (x > 0) {  
    x -= 1;  
    y += 1;  
}
```

## x86-64 assembly

```
jmp .L2          ; Jump to L2  
.L3:  
subq $1, %r8 ; x -= 1  
addq $1, %r9 ; y += 1  
.L2:  
cmpq $0, %r8 ; Compare 0 and x  
jg  .L3       ; Jump if x greater
```



# Loops as jumps

## C code

```
while (x > 0) {  
    x -= 1;  
    y += 1;  
}
```

## x86-64 assembly

```
jmp .L2          ; Jump to L2  
.L3:  
subq $1, %r8 ; x -= 1  
addq $1, %r9 ; y += 1  
.L2:  
cmpq $0, %r8 ; Compare 0 and x  
jg  .L3       ; Jump if x greater
```

We can also write this in C.

```
    goto L2;  
L3: x -= 1;  
    y += 1;  
L2: if (x > 0) goto L3;
```

# Function calls at the assembly level

**The CPU has no idea what a function is.**

- **How a function call works at assembly level.**

1. Jump somewhere by modifying PC.
2. Run instructions from there.
3. One of those will jump back to just after the instruction that did step 1.

- **Questions that must be answered:**

1. How does a function know where to “jump back”?
  - ▶ I.e. the address of the calling instruction.
2. How do we avoid clobbering the registers in use by the caller?
  - ▶ Registers are a bit like programming exclusively with global variables—all functions use the same ones.

# The stack to the rescue

## Conceptually

- A stack is a data structure on which we can *push* and *pop* values at the top.
- **Before calling a function:**
  - ▶ Push values of all registers to the stack.
  - ▶ Push address of calling instruction.
    - ▶ Now function will know where to jump back to!
- **After returning from function:**
  - ▶ Restore registers by popping them from stack.
- Important that function never pops more than it pushed.

## Actually

- The stack is just normal memory, with the *stack pointer* (SP) being a register dedicated to storing address of stack top.

- **Pushing  $x$  to the stack:**

`mem[SP] = x;`

`SP--;`

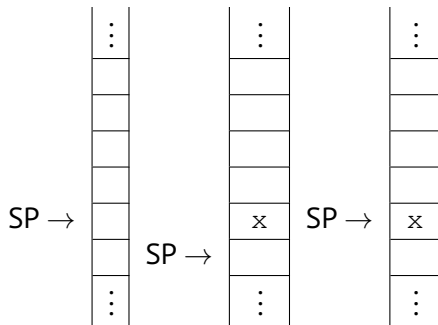
- **Popping  $x$  from stack:**

`SP++;`

`x = mem[SP];`

Original    After push    After pop

High addresses



Low addresses

# Takeaways

- CPUs execute instructions in sequence.
  - ▶ Instructions are stored in memory.

# Takeaways

- CPUs execute instructions in sequence.
  - ▶ Instructions are stored in memory.
- An instruction modifies the machine state.
  - ▶ May also cause execution to jump elsewhere.

# Takeaways

- CPUs execute instructions in sequence.
  - ▶ Instructions are stored in memory.
- An instruction modifies the machine state.
  - ▶ May also cause execution to jump elsewhere.
- Must move data from memory into registers to operate on them.
  - ▶ Registers are a very scarce resource.

# Takeaways

- CPUs execute instructions in sequence.
  - ▶ Instructions are stored in memory.
- An instruction modifies the machine state.
  - ▶ May also cause execution to jump elsewhere.
- Must move data from memory into registers to operate on them.
  - ▶ Registers are a very scarce resource.
- **Writing assembly code by hand is extremely tedious and completely impractical for all but the smallest programs.**
  - ▶ *Compilers* convert high-level languages to machine code.



How computers execute code

**Compiled and interpreted languages**

Tombstone diagrams

## Roughly two kinds of languages

Compiled languages are transformed to machine code before execution (e.g. C)

Interpreted languages are run directly by a software *interpreter* (e.g. Python)

# Roughly two kinds of languages

Compiled languages are transformed to machine code before execution (e.g. C)

Interpreted languages are run directly by a software *interpreter* (e.g. Python)

## Pedantic disclaimer

Compilation/interpretation is strictly a property of *implementations*, not *languages*.

- You could have a C interpreter or Python compiler
- But most (not all!) languages are built with a specific implementation technique in mind
- A few languages (Lisp, JavaScript) have lots of *very* different implementations...

**We teach you the big picture—the details are always more complicated in practice!**

# Tradeoffs

- 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

# Tradeoffs

- 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

# Tradeoffs

- 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

**Let us look at the scale of the overhead.**

# The Collatz conjecture

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

- **Conjecture:** if we apply this function to some number greater than 1, we will eventually reach 1
- To disprove this conjecture, we only need *a single counter-example* that goes into a cycle instead
- People write programs to investigate the behaviour of this sequence
  - ▶ Especially if you are not good at number theory

Listing 1: collatz.py

```
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

```
#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));
    }
}
```



## Benchmarking collatz.py

```
$ time python3 ./collatz.py 100000 >/dev/null
```

```
real    0m1.368s
```

```
user    0m1.361s
```

```
sys     0m0.007s
```

## Benchmarking collatz.py

```
$ time python3 ./collatz.py 100000 >/dev/null
```

```
real    0m1.368s
```

```
user    0m1.361s
```

```
sys     0m0.007s
```

```
$ gcc collatz.c -o collatz
```

## Benchmarking collatz.py

```
$ time python3 ./collatz.py 100000 >/dev/null
```

```
real    0m1.368s
```

```
user    0m1.361s
```

```
sys     0m0.007s
```

```
$ gcc collatz.c -o collatz
```

```
$ time ./collatz 100000 >/dev/null
```

```
real    0m0.032s
```

```
user    0m0.030s
```

```
sys     0m0.002s
```

# Benchmarking collatz.py

```
$ time python3 ./collatz.py 100000 >/dev/null
```

```
real    0m1.368s
```

```
user    0m1.361s
```

```
sys     0m0.007s
```

```
$ gcc collatz.c -o collatz
```

```
$ time ./collatz 100000 >/dev/null
```

```
real    0m0.032s
```

```
user    0m0.030s
```

```
sys     0m0.002s
```

$$\text{Speedup: } \frac{1.368}{0.032} = 42.75$$

# Combining interpretation and compilation

- Interpreted languages can be fast when
  - ▶ Most of the run-time is spent waiting data from files or network
  - ▶ They mostly call functions written in faster compiled languages
- **Best of both worlds:** flexibility of interpretation and speed of C

# Different ways to compile

# Different ways to compile

To executable program `collatz`

```
$ gcc collatz.c -o collatz
```

- Can be run directly

# Different ways to compile

To executable program `collatz`

```
$ gcc collatz.c -o collatz
```

- Can be run directly

To object file `collatz.o`

```
$ gcc collatz.c -c -o collatz.o
```

- Can be *linked* with other object files
- Can be processed further



# Different ways to compile

To executable program `collatz`

```
$ gcc collatz.c -o collatz
```

- Can be run directly

To object file `collatz.o`

```
$ gcc collatz.c -c -o collatz.o
```

- Can be *linked* with other object files
- Can be processed further

To shared object file `libcollatz.so`

```
$ gcc collatz.c -fPIC -shared -o libcollatz.so
```

- Can be linked *at run-time* by a running program
- How compiled programs support dynamic “plug-ins”

**All output files contain fully compiled machine code.**

# Calling C from Python

## Compiling C program to shared library

```
$ gcc collatz.c -fPIC -shared -o libcollatz.so
```

### Listing 3: collatz-ffi.py

```
import ctypes
import sys

c_lib = ctypes.CDLL('./libcollatz.so')

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

```
$ time python3 ./collatz-ffi.py 100000 >/dev/null
```

```
real    0m0.165s
```

```
user    0m0.163s
```

```
sys     0m0.003s
```

**Speedup:**  $\frac{1.368}{0.165} = 8.2$

```
$ time python3 ./collatz-ffi.py 100000 >/dev/null
```

```
real    0m0.165s
user    0m0.163s
sys     0m0.003s
```

$$\text{Speedup: } \frac{1.368}{0.165} = 8.2$$

- Slower than pure C by about  $5\times$
- 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!**

# NumPy performance

```
def f_python(v):  
    for i in range(len(v)):  
        v[i] = v[i]*2 + 3
```

```
def f_numpy(v):  
    return v * 2 + 3
```

Size of v	f_python	f_numpy	Difference
1	0.01ms	0.01ms	0.9×
10	0.01ms	0.01ms	1.4×
100	0.1ms	0.01ms	13.3×
1000	0.98ms	0.01ms	95.3×
10000	9.96ms	0.05ms	190.7×
100000	98.59ms	0.41ms	240.7×

How computers execute code

Compiled and interpreted languages

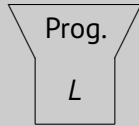
**Tombstone diagrams**

## Now a high-level view

- We've looked at some technical details of compilers and interpreters
- Do we also have a high-level model?

# Tombstone diagrams

A program written in  $L$



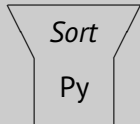


# Tombstone diagrams

A program written in  $L$



Example of program written in Python



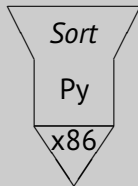
A machine that runs  $L$  programs



A machine that runs  $L$  programs



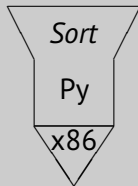
Example



A machine that runs  $L$  programs



Example



**Incorrect!** Languages (Python and x86) do not match!

An interpreter for  $F$ , written in  $T$

$F$

$T$

An interpreter for  $F$ , written in  $T$

$F$

$T$

Example

*Sort*

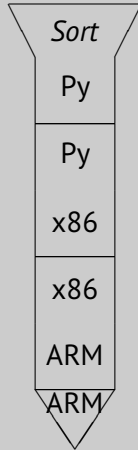
Py

Py

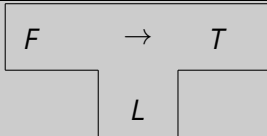
x86

x86

## Stacking interpreters

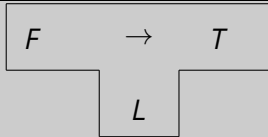


A compiler from  $F$  to  $T$ , written in  $L$

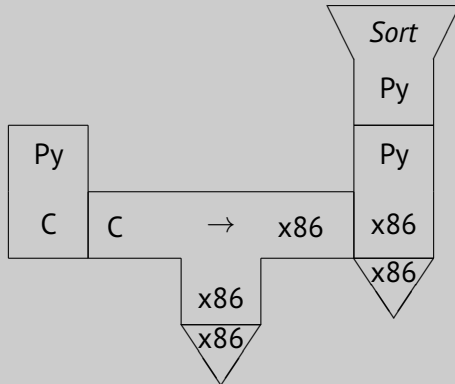




A compiler from  $F$  to  $T$ , written in  $L$



Example



# Compilers can be chained

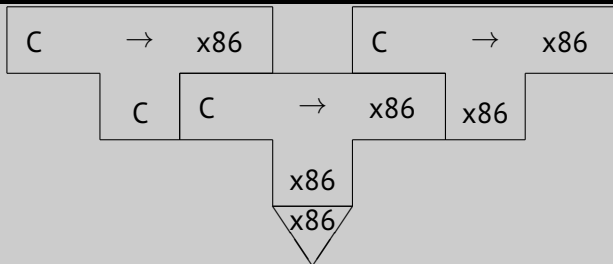
Futhark  $\rightarrow$  C  $\rightarrow$  machine code



# Compilers are also programs

- A C compiler is usually written in a high-level language, not in machine code
- Use old version of the compiler to compile the new version of the compiler

## Example



- All the way back to the first computers, where some primordial primitive compiler or assembler was written in machine code

# Advantages and limitations of tombstone diagrams

- + Abstracts away technical details of object files, compilation modes etc
- Cannot express more complex situations such as dynamic linking
- In practice mostly used for visualising **bootstrapping**—the process of writing compilers in the language they compile, or bringing up new hardware

# Conclusions

- Compiled languages tend to be fast, but less flexible
- Interpreted languages tend to be slower, but more flexible
- **Best of both worlds:** write computational primitives in fast languages, call them from slow languages
  - ▶ NumPy works like this
- Tombstone diagrams make the relationship between compiler, interpreter, and machine clear
  - ▶ Although in day-to-day work, we only use simple compositions