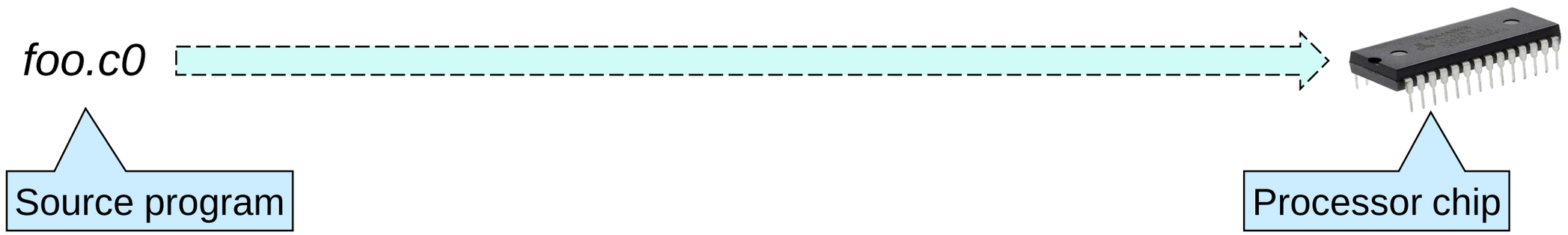
Program Execution

Execution Models

How are Programs Executed?

- Ultimately, the instructions of a program run on the hardware

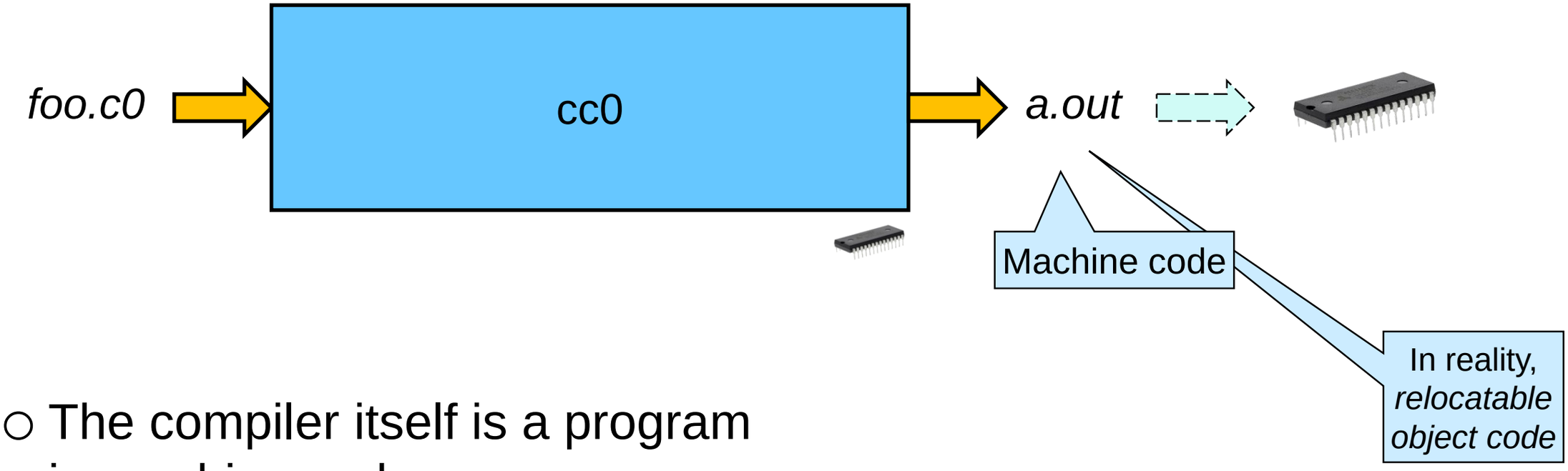


  - But the hardware does not understand C0

- Two main ways to bridge the gap
  - through a *compiler*
  - through an *interpreter*

Compilation

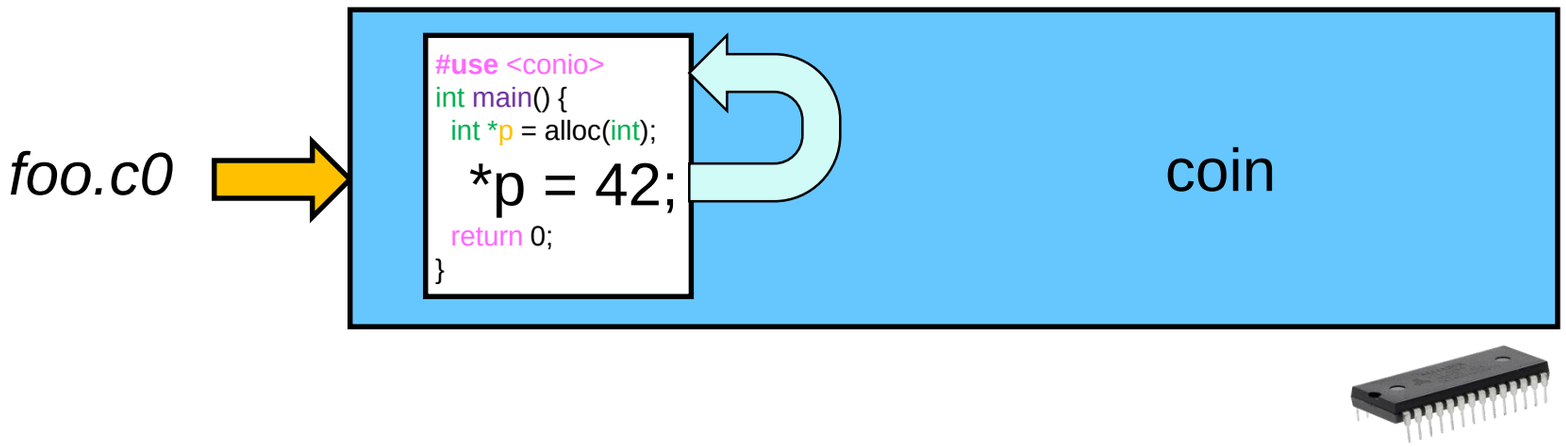
- A **compiler** translates the source program into **machine code**
  - an equivalent program in the language that the processor understands and can execute directly
    - with the help of the OS



  - The compiler itself is a program in machine code
    - when we execute it

Interpreters

- An **interpreter** reads each line in the source program and **simulates** it on the hardware

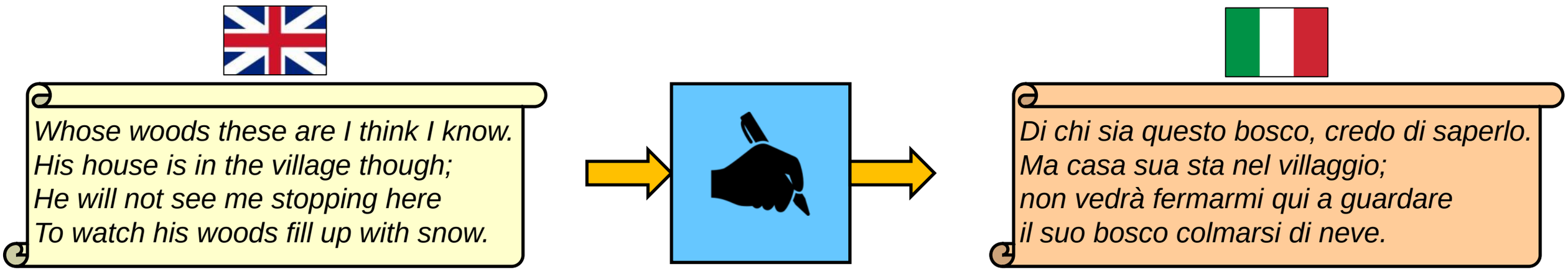


  - The interpreter itself is a program in machine code
    - when we execute it

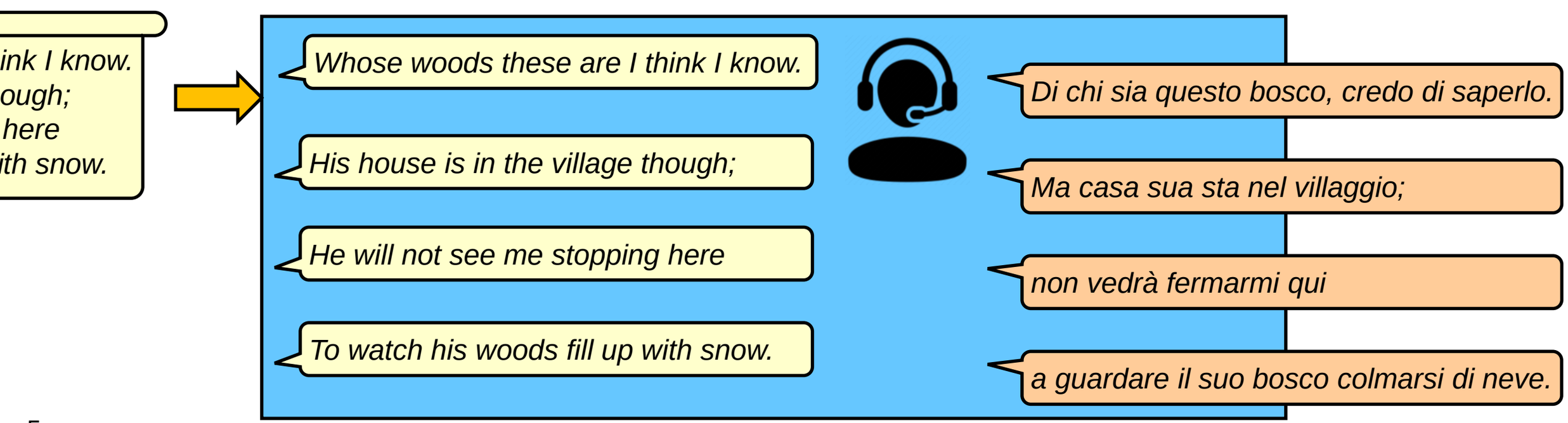
  - The interpreter acts like a *virtual processor* for the source language

Compilation vs. Interpretation

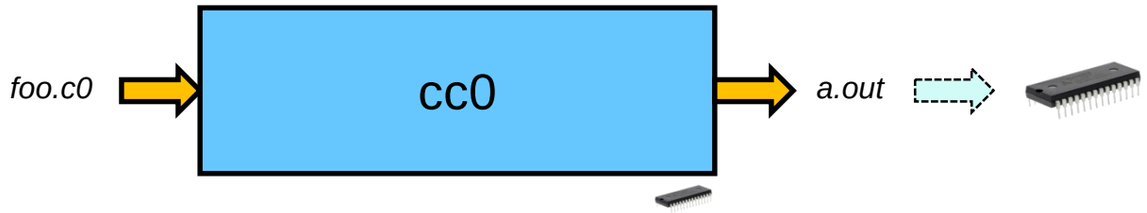
- Compilation is like translating a text in full offline



- Interpretation is like translating a text line by line in real time



Compilation



- To run a program, all we need is the executable
  - ➢ on the same hardware and with the same OS
  - ○ distribute the executable, not the source program

- The (executable) code runs very fast
  - ○ The compiler can perform lots of optimizations

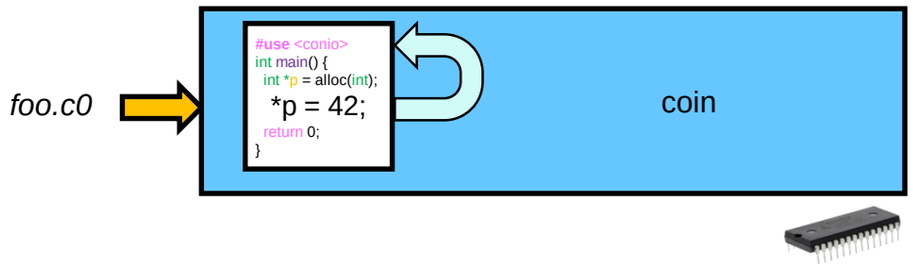
- Recompiling a large program takes time

- Running a program on new hardware requires a new compiler
  - ○ Writing a compiler is hard if we want the code to be fast

- Languages that are typically compiled:
  - ○ languages where performance is paramount

C, …

Interpretation



- To run a program, we need the source code *and the interpreter*

- Each source instruction is simulated
  - this slows down execution
  - but the instructions can easily be screened for safety

- Running a program on new hardware requires a new interpreter

- Languages that are typically interpreted:
  - Shell scripts, make, …
  - languages used to write small programs where performance is not critical

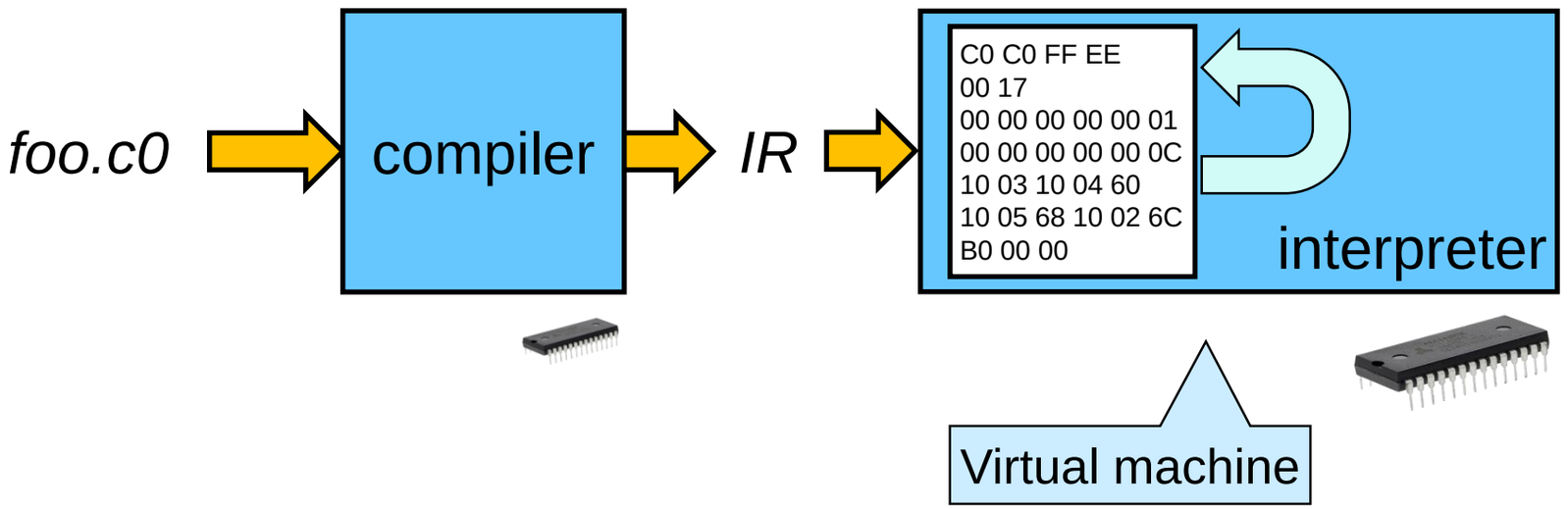
Compilation vs. Interpretation

	Compilation	**Interpretation**
Pro	• Code is very fast • Perform complex optimizations • Just executable required to run	• Instructions can be screened • Can be use interactively
Cons	• Lengthy recompilation • No safety checks • Not portable	• Interpreter and source code are needed for running • No optimizations • Execution is slower

The Best of Both Worlds

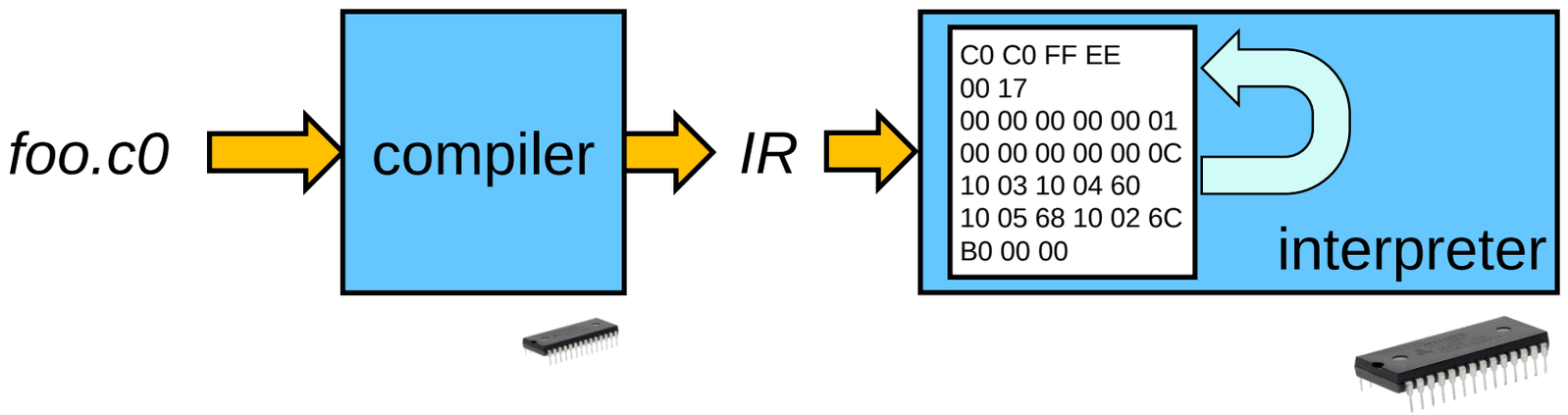
1. Compile the high-level source program to a lower level **intermediate representation**

2. Interpret the intermediate representation
   - This interpreter is called a **virtual machine** (VM)



- This is called **two-stage execution**

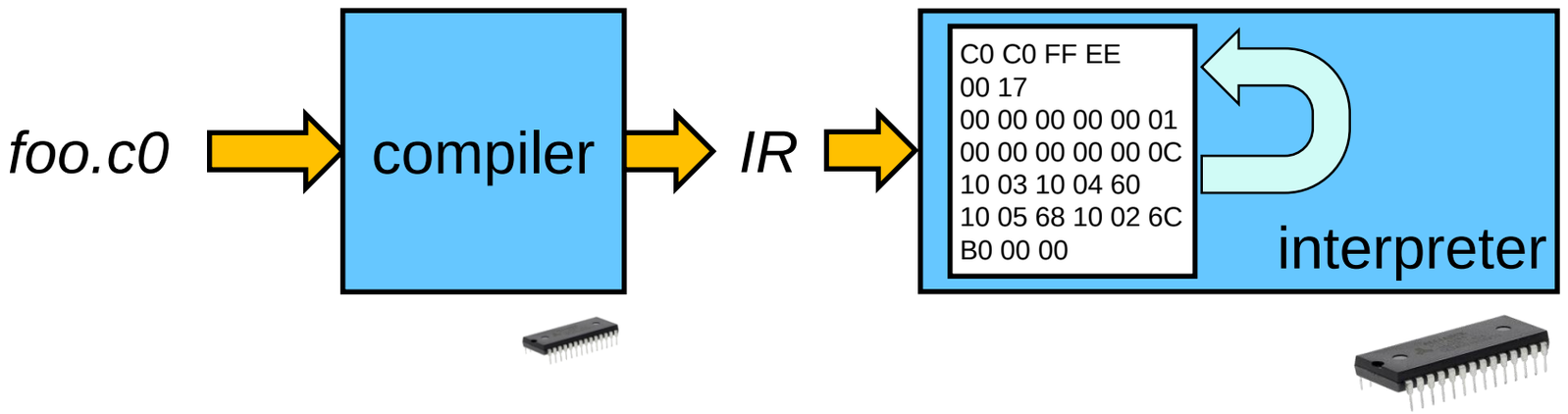
Two-stage Execution



- We gain benefits if the intermediate representation language is much simpler than the source language
  - the VM can be lightweight
    - very little simulation overhead
  - the compiler can perform complex optimizations

- An intermediate language where each instruction fits in one byte is called a **bytecode**

Two-stage Execution



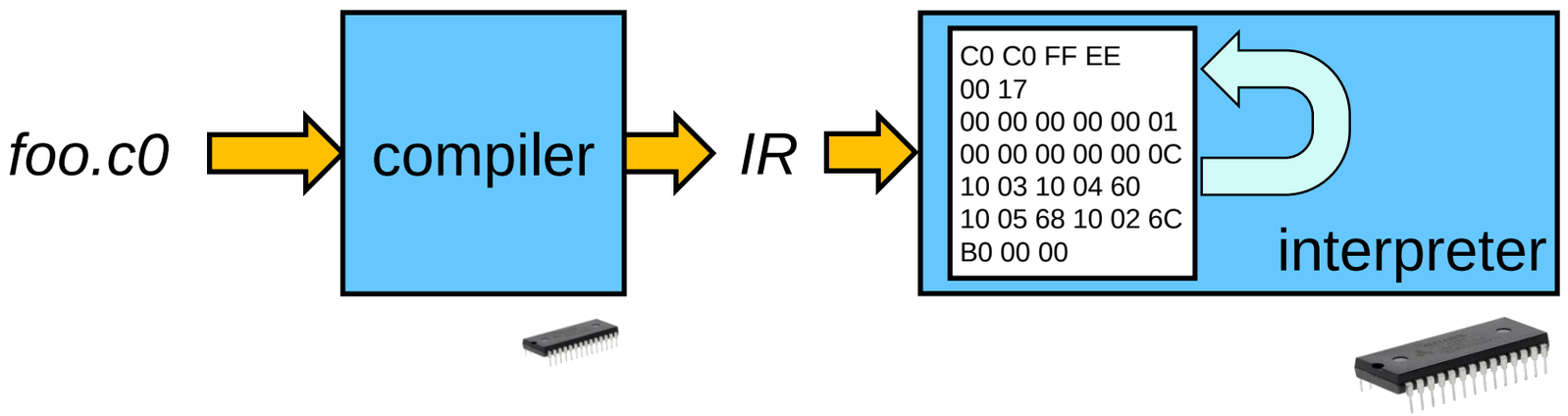
- To run a program, all we need is the bytecode and the VM

- To run a program on new hardware we need
  - a new VM
    - easy to implement because the compiler does the heavy lifting
  - We can compile the source program on different hardware, or
  - if the compiler is written in the source language, it can **compile itself** to bytecode and then run on the new VM

Write the compiler once and for all

Chicken and egg problem?
Solved through **bootstrapping**

Two-stage Execution



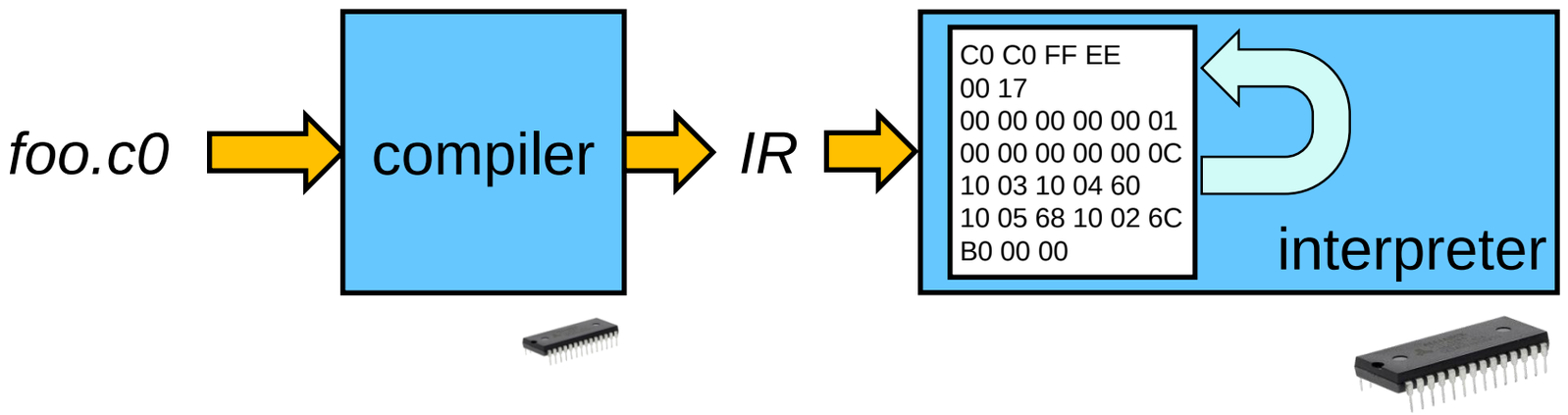
- Most modern languages use this two-stage approach
  - a Python program is first compiled to *Python bytecode* and then executed in the *Python VM*

A data structure in memory

  - Javascript, PHP and many others are compiled to a common bytecode called the WebAssembly
  - Other languages are compiled to LLVM
    - Implementations of gcc based on Clang do that too

Two-stage Execution

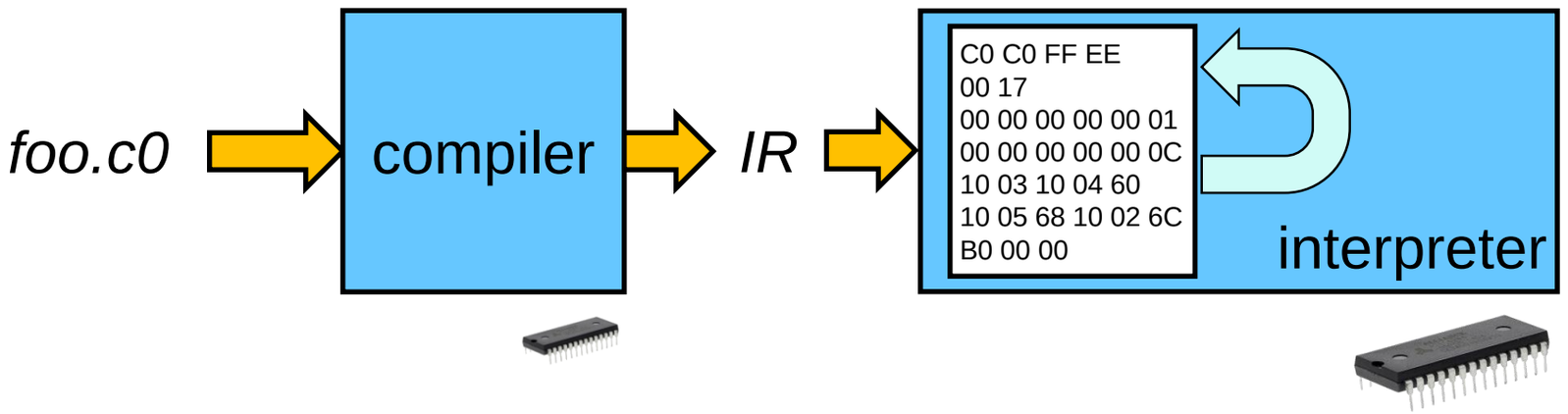
Historical note



- The first mainstream language to use this two-stage approach was Pascal in 1970
  - the goal was portability
    - have programs run in a uniform way across hardware
    - have an efficient way to get them running on new hardware

Historical note

Two-stage Execution



- The language that popularized it was **Java** in 1995
  - the IR language is called Java bytecode
  - the virtual machine is called the JVM

The contents of a .class file

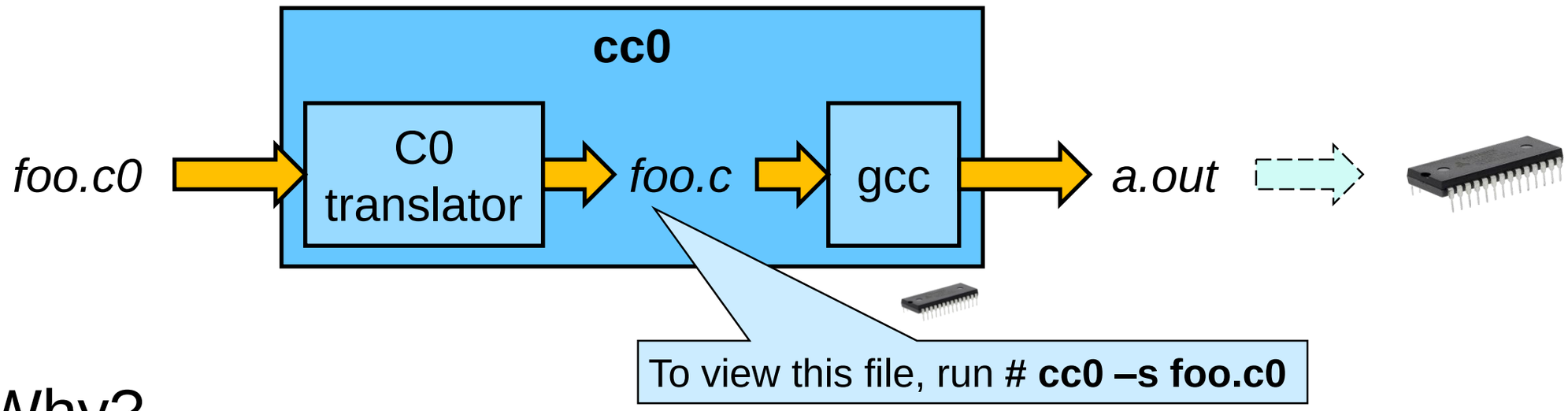
  - the goal was supporting mobile code on the nascent Web
    - a browser downloaded an *applet* and ran it
      - the bytecode was compact to minimize download time and cost
      - the JVM ran it (relatively) fast
    - the bytecode was untrusted
      - it was typechecked for statically unsafe operations
      - it was screened at run-time for unsafe operations

Mainly security concerns

C0 Execution Models

Compiling a C0 Program with cc0

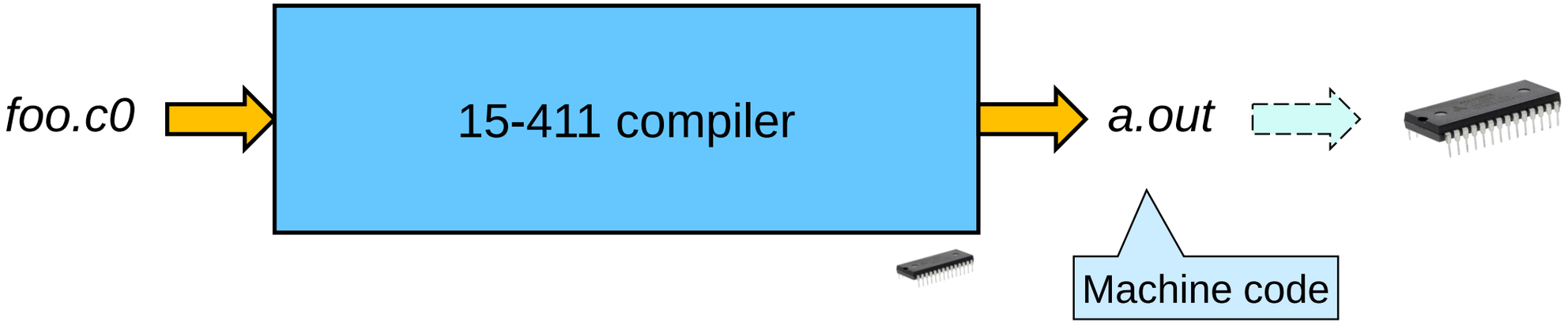
- Under the hood, **cc0** translates a C0 program to C and then runs **gcc** to compile it



- Why?
  - Writing a C0-to-C translator is relatively **easy**
    - the most complicated part is dealing with C's undefined behaviors
  - The resulting executable is extremely **fast**
    - the gcc compiler is really good
  - This makes cc0 very **portable**
    - there is a gcc compiler for almost every hardware

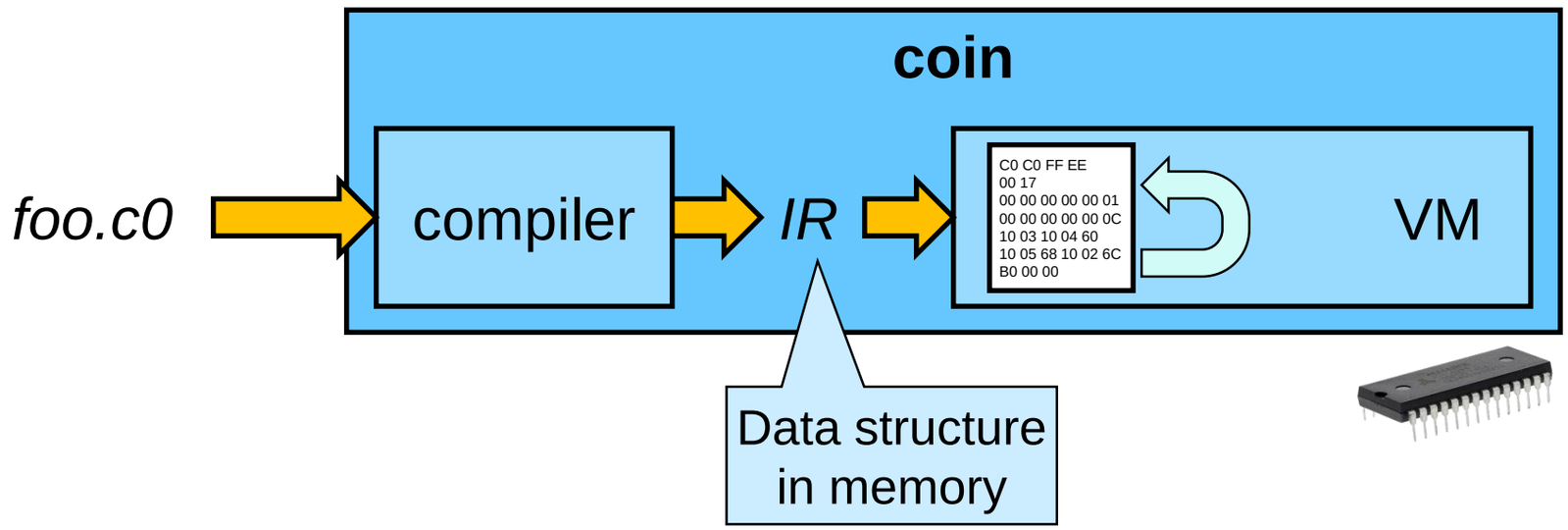
Compiling a C0 Program *without* cc0

- CMU's compiler course (15-411) teaches how to write a standalone compiler for C0



Interpreting a C0 Program in coin

- Under the hood, **coin** compiles a C0 program to a bytecode data structure in memory and then runs a virtual machine



Two-stage Execution of a C0 Program

- A C0 program can be compiled to **C0VM bytecode** with

Linux Terminal

```
# cc0 -b foo.c0
```

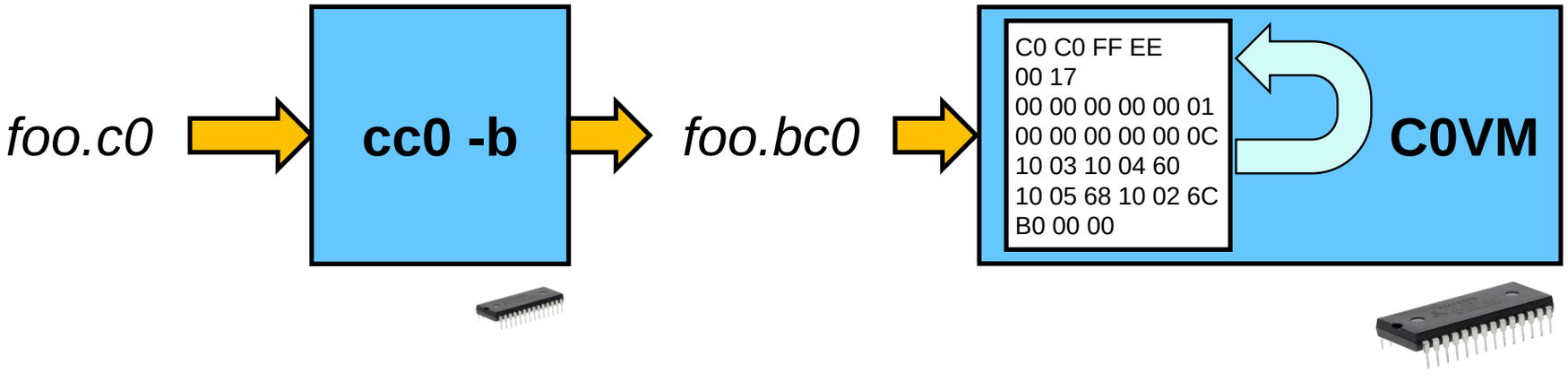
This produces the *C0VM bytecode file* foo.bc0

- The bytecode file is then executed using the **C0 virtual machine**

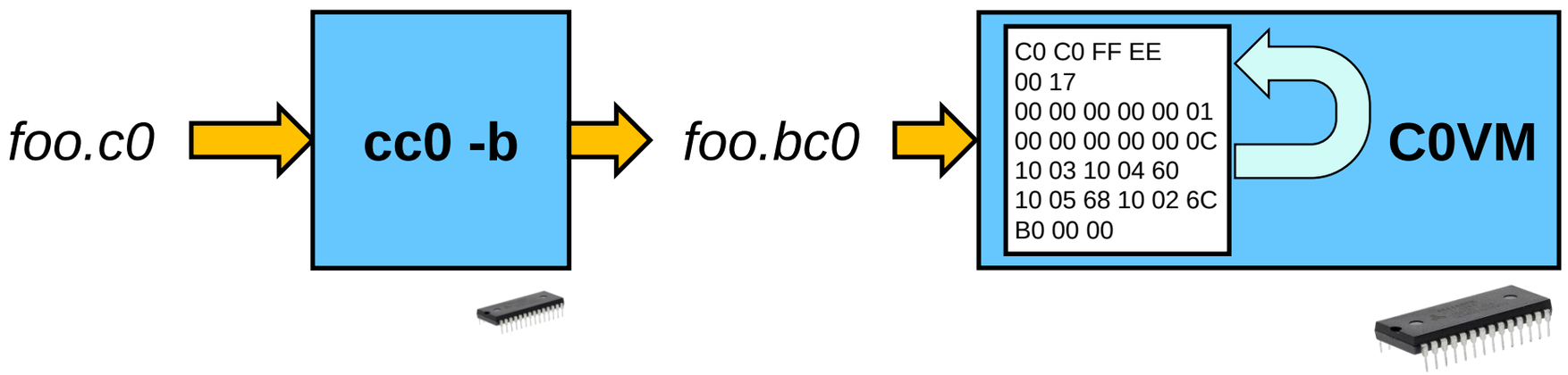
Linux Terminal

```
# c0vm foo.bc0
```

This runs foo.bc0 in the C0VM



Two-stage Execution of a C0 Program



- Compiling to C0VM bytecode takes some effort …

  … but implementing the C0VM is relatively easy

- We will now examine what this involves
  - understand the structure of the C0VM bytecode
  - describe how to execute C0VM bytecode instructions
  - outline what it takes to implement the C0VM

C0 Bytecode

Compiling a Simple C0 Program

- Consider this C0 program
  - in file ex1.c0

```
int main() {
  return (3 + 4) * 5 / 2;
}
```

- We compile it to bytecode with

Linux Terminal

```
# cc0 -b ex1.c0
```

If we had contracts, we also could pass the -d flag

- Let’s look at the bytecode file ex1.bc0

A C0VM Bytecode File

```
int main() {
  return (3 + 4) * 5 / 2;
}
```

- This is a **text** file
  - This is because C0VM is a pedagogical architecture

to learn how virtual machines work

  - An actual bytecode file would be raw binary

That's what a Java .class file is

- It would be easy to produce **binary** instead

```
C0 C0 FF EE     # magic number
00 17           # version 11, arch = 1 (64 bits)

00 00           # int pool count
# int pool

00 00           # string pool total size
# string pool

00 01           # function count
# function_pool

#<main>
00              # number of arguments = 0
00              # number of local variables = 0
00 0C           # code length = 12 bytes
10 03   # bipush 3      # 3
10 04   # bipush 4      # 4
60      # iadd          # (3 + 4)
10 05   # bipush 5      # 5
68      # imul          # ((3 + 4) * 5)
10 02   # bipush 2      # 2
6C      # idiv          # (((3 + 4) * 5) / 2)
B0      # return        #

00 00           # native count
# native pool
```

A C0VM Bytecode File

```
int main() {
  return (3 + 4) * 5 / 2;
}
```

- The (ASCII representation of the) bytes in hexadecimal are on the left
  - ➢ two hex digits represent 1 byte
  - ○ Everything after a # is a comment
  - ○ Spaces and new lines are for readability

- The actual bytecode is

  C0C0FFEE0017000000000001000000C10031004601005681002 6CB00000

  - ➢ as a bit sequence

```
C0 C0 FF EE     # magic number
00 17           # version 11, arch = 1 (64 bits)

00 00           # int pool count
# int pool

00 00           # string pool total size
# string pool

00 01           # function count
# function_pool

#<main>
00              # number of arguments = 0
00              # number of local variables = 0
00 0C           # code length = 12 bytes
10 03   # bipush 3      # 3
10 04   # bipush 4      # 4
60      # iadd          # (3 + 4)
10 05   # bipush 5      # 5
68      # imul          # ((3 + 4) * 5)
10 02   # bipush 2      # 2
6C      # idiv          # (((3 + 4) * 5) / 2)
B0      # return        #

00 00           # native count
# native pool
```

```
int main() {
  return (3 + 4) * 5 / 2;
}
```

A C0VM Bytecode File

- The bytecode consists of five **segments**
  - We will examine them in detail later
  - For now we only consider this portion, which is how

    return (3 + 4) * 5 / 2;

    gets compiled

```
C0 C0 FF EE     # magic number
00 17           # version 11, arch = 1 (64 bits)

00 00           # int pool count
# int pool

00 00           # string pool total size
# string pool

00 01           # function count
# function_pool

#<main>
00              # number of arguments = 0
00              # number of local variables = 0
00 0C           # code length = 12 bytes
10 03   # bipush 3      # 3
10 04   # bipush 4      # 4
60      # iadd          # (3 + 4)
10 05   # bipush 5      # 5
68      # imul          # ((3 + 4) * 5)
10 02   # bipush 2      # 2
6C      # idiv          # (((3 + 4) * 5) / 2)
B0      # return        #

00 00           # native count
# native pool
```

Bytecode Instructions

Postfix Notation

- Arithmetic operators are normally written infix

  (3 + 4) * 5 / 2 — The operators are written between their operands

  - They are given a precedence, and we use parentheses to override it

- **Postfix notation** places the operator ***after*** the operand

  3 4 + 5 * 2 / — This is also called Polish reverse notation

  - Parentheses are not needed any more
  - A postfix expression can be executed with the help of a **stack**
    - when seeing a number, push it on the stack
    - when seeing an operator, apply it to the topmost numbers on the stack and replace them with the result
    - The final result is the one number on the stack

Stack	Expression
(empty)	3 4 + 5 * 2 /
3	4 + 5 * 2 /
3 4	+ 5 * 2 /
7	5 * 2 /
7 5	* 2 /
35	2 /
35 2	/
17	(done)

```
…
  return (3 + 4) * 5 / 2;
…
```

Arithmetic Expressions

- The middle column of the bytecode for

  (3 + 4) * 5 / 2

  is just like the postfix notation for it

  3 4 + 5 * 2 /

```
  …
10 03     # bipush 3     # 3
10 04     # bipush 4     # 4
60        # iadd         # (3 + 4)
10 05     # bipush 5     # 5
68        # imul         # ((3 + 4) * 5)
10 02     # bipush 2     # 2
6C        # idiv         # (((3 + 4) * 5) / 2)
B0        # return       #
  …
```

- Rather than having numbers to be pushed on the stack
  - like 3

  we have **instructions** to push numbers on the stack
  - like bipush 3
    - otherwise, we would not know whether 3 4 is the two numbers 3 and 4, or the number 34
      - Recall the spaces are for readability only
        - they don't exist in the actual bytecode

```
…
  return (3 + 4) * 5 / 2;
…
```

Arithmetic Expressions

- Every item in the postfix notation of

  3 4 + 5 * 2 /

  is turned into an **instruction**
  - ➢ bipush 3 pushes 3 on the stack
    - ❑ really, that's 10 03
  - ➢ iadd adds the two topmost stack elements

```
…
10 03    # bipush 3    # 3
10 04    # bipush 4    # 4
60       # iadd        # (3 + 4)
10 05    # bipush 5    # 5
68       # imul        # ((3 + 4) * 5)
10 02    # bipush 2    # 2
6C       # idiv        # (((3 + 4) * 5) / 2)
B0       # return      #
…
```

C0VM bytecode instructions

Mnemonic read-out

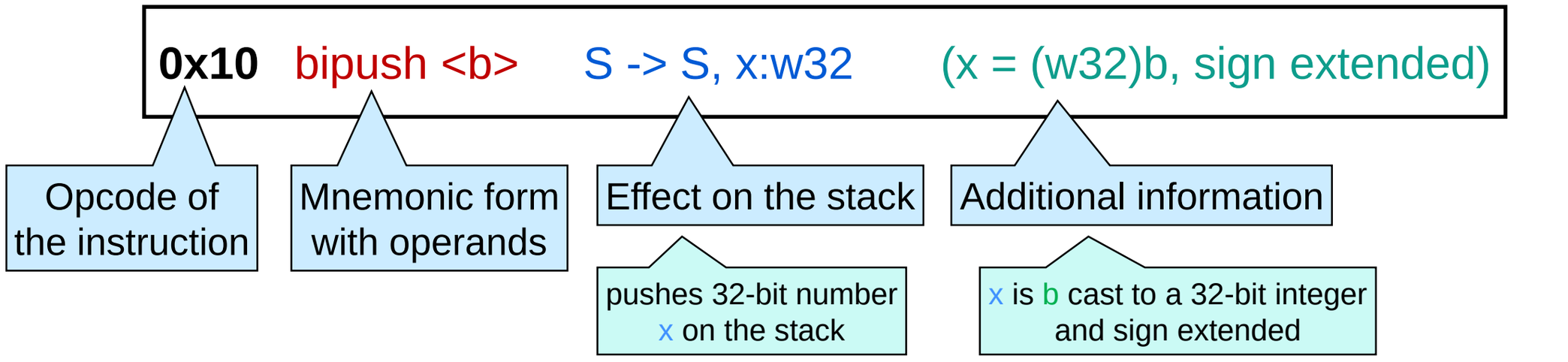
Top of the stack after executing the instruction

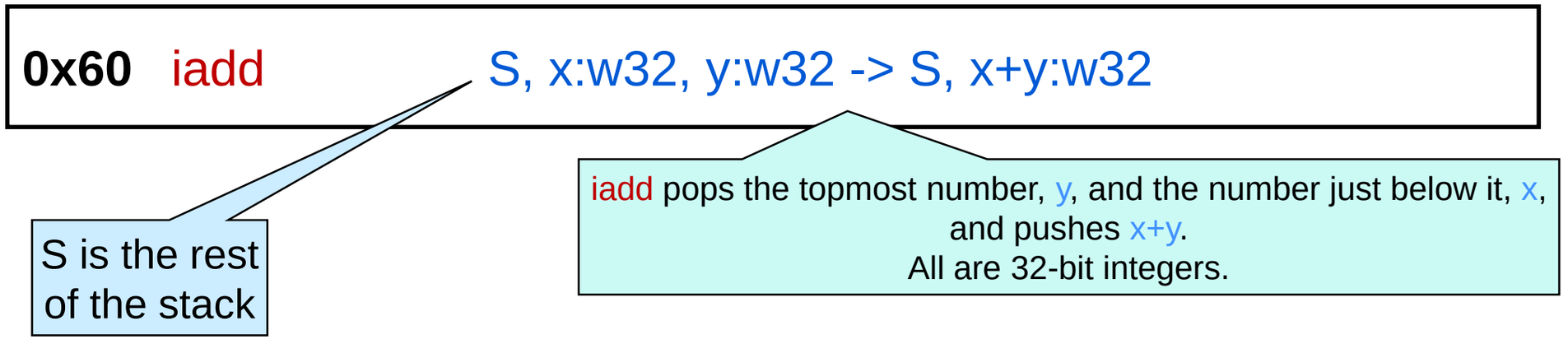
- Each instructions starts with one byte called its **opcode**
  - ○ the opcode of bipush is 0x10
  - ○ the opcode of iadd is 0x60
- Some instructions take operands
  - ○ bipush takes a 1-byte operand (the number to push on the stack)
  - ○ iadd takes no operand

bipush can only push numbers in the range [-128, 127]

Describing Instructions

- C0VM instructions are uniformly described by **rules** that spell out their effect on the stack and other run-time data structures





C0VM Instructions so far

0x10	bipush <b>	S -> S, x:w32	(x = (w32)b, sign extended)
0x60	iadd	S, x:w32, y:w32 -> S, x+y:w32	
0x68	imul	S, x:w32, y:w32 -> S, x*y:w32	
0x6C	idiv	S, x:w32, y:w32 -> S, x/y:w32	
0xB0	return	., v -> .	(return v to caller)

return expects a single value v
on the stack and returns it to the caller
“.” denotes the empty stack

Divides the second topmost element of the stack
by the topmost element

- In a valid bytecode file, the stack always contains enough operands to execute any instructions

The Current Instruction

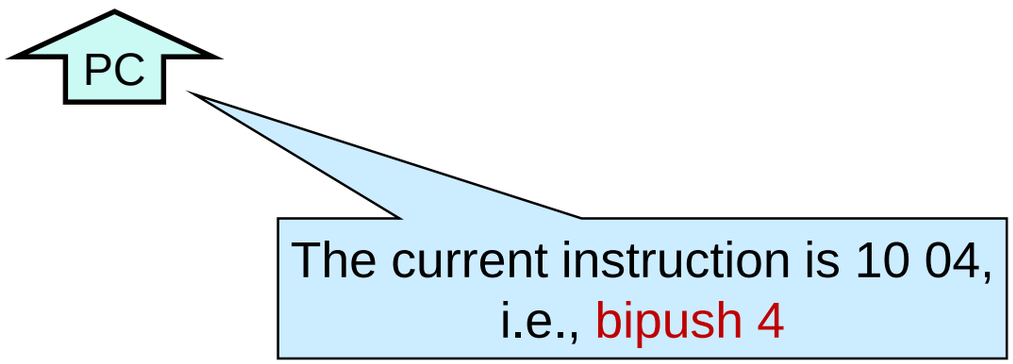
- Once the **current** instruction has been executed, the C0VM executes the one after it
  - but how to tell which one is the current instruction?

    1003100460100568100 26CB0

```
 …
10 03    # bipush 3    # 3
10 04    # bipush 4    # 4
60       # iadd        # (3 + 4)
10 05    # bipush 5    # 5
68       # imul        # ((3 + 4) * 5)
10 02    # bipush 2    # 2
6C       # idiv        # (((3 + 4) * 5) / 2)
B0       # return      #
 …
```

  - The C0VM has a **program counter** that points to the opcode of the current instruction

    1003**1004**60100568100 26CB0



The Next Instruction

- Once the **current** instruction has been executed, the PC is updated to point to the next instruction

```
1003100460100568 10026CB0
          ^
          PC
```

```
 …
10 03    # bipush 3    # 3
10 04    # bipush 4    # 4
60       # iadd        # (3 + 4)
10 05    # bipush 5    # 5
68       # imul        # ((3 + 4) * 5)
10 02    # bipush 2    # 2
6C       # idiv        # (((3 + 4) * 5) / 2)
B0       # return      #
 …
```

  - By how much to update it depends on the number of operands of the current instruction
    - we move it two byte over after executing bipush 4

```
1003100460 10056810026CB0
          ^
          PC
```

    - then, we move one byte over after executing iadd

```
1003100460100 56810026CB0
          ^
          PC
```

We need to be careful not to get lost in the bytecode

Run-time Data Structures … so far

- To run a C0VM bytecode program, the C0VM needs to maintain some **data structures**
  - the bytecode itself
  - the operand stack S
  - the program counter PC
  - …

We will see how later

more to come

Local Variables

Another Example

- Next, let's compile

```
int mid(int lo, int hi) {
  int mid = lo + (hi - lo)/2;
  return mid;
}

int main() {
  return mid(3, 6);
}
```

Midpoint of an array segment

- Two novelties
  - function headers
  - local variables

We will look at how to call a function later

```
C0 C0 FF EE       # magic number
00 17             # version 11, arch = 1 (64 bits)

00 00             # int pool count
# int pool

00 00             # string pool total size
# string pool

00 02             # function count
# function_pool

#<main>
00                # number of arguments = 0
00                # number of local variables = 0
00 08             # code length = 8 bytes
10 03       # bipush 3          # 3
10 06       # bipush 6          # 6
B8 00 01    # invokestatic 1    # mid(3, 6)
B0          # return            #

#<mid>
02                # number of arguments = 2
03                # number of local variables = 3
00 10             # code length = 16 bytes
15 00       # vload 0           # lo
15 01       # vload 1           # hi
15 00       # vload 0           # lo
64          # isub              # (hi - lo)
10 02       # bipush 2          # 2
6C          # idiv              # ((hi - lo) / 2)
60          # iadd              # (lo + ((hi - lo) / 2))
36 02       # vstore 2          # mid = (lo + ((hi - lo) / 2));
15 02       # vload 2           # mid
B0          # return            #

00 00             # native count
# native pool
```

Function Headers

```
int mid(int lo, int hi) {
  int mid = lo + (hi - lo)/2;
  return mid;
}
```

- Each function starts with a 4-bytes **header**
  - 1 byte for the number of function arguments
    - there can be at most 2^8 arguments
  - 1 byte for the number of local variables
    - the function arguments count among the local variables
    - there are at most 2^8 local variables
  - 2 bytes for the number of bytes in the bytecode of the function
    - each function can be compiled in at most 2^{16} bytes
      - cc0 does not have this restriction

```
  …
#<mid>
02              # number of arguments = 2
03              # number of local variables = 3
00 10           # code length = 16 bytes
15 00       # vload 0         # lo
15 01       # vload 1         # hi
15 00       # vload 0         # lo
64          # isub            # (hi - lo)
10 02       # bipush 2        # 2
6C          # idiv            # ((hi - lo) / 2)
60          # iadd            # (lo + ((hi - lo) / 2))
36 02       # vstore 2        # mid = (lo + ((hi - lo) / 2));
15 02       # vload 2         # mid
B0          # return          #
  …
```

Local Variables

```
int mid(int lo, int hi) {
  int mid = lo + (hi - lo)/2;
  return mid;
}
```

```
  …
#<mid>
02              # number of arguments = 2
03              # number of local variables = 3
00 10           # code length = 16 bytes
15 00       # vload 0         # lo
15 01       # vload 1         # hi
15 00       # vload 0         # lo
64          # isub            # (hi - lo)
10 02       # bipush 2        # 2
6C          # idiv            # ((hi - lo) / 2)
60          # iadd            # (lo + ((hi - lo) / 2))
36 02       # vstore 2        # mid = (lo + ((hi - lo) / 2));
15 02       # vload 2         # mid
B0          # return          #
  …
```

- The local variables are held in a new run-time data structure,
  - the **local variable array**, V
- Two bytecode instructions operate on V
  - vload i pushes the i-th value of V onto the operand stack
  - vstore i pops the operand stack and saves this value in the i-th position of V

0x15	vload <i>	S -> S, v	(v = V[i])
0x36	vstore <i>	S, v -> S	(V[i] = v)

i is 1 byte unsigned:
V contains at most 256 values

  - When a function is called, V is preloaded with its arguments

Local Variables

```
int mid(int lo, int hi) {
  int mid = lo + (hi - lo)/2;
  return mid;
}
```

- What this code does:
  - push **lo** – that's V[0]
  - push **hi**, **lo** and compute **(hi-lo)/2**
  - add them, getting **lo + (hi-lo)/2**
  - save to **mid** – that's V[2]
  - load **mid** on the stack
  - return to caller

```
 …
#<mid>
02              # number of arguments = 2
03              # number of local variables = 3
00 10           # code length = 16 bytes
15 00       # vload 0         # lo
15 01       # vload 1         # hi
15 00       # vload 0         # lo
64          # isub            # (hi - lo)
10 02       # bipush 2        # 2
6C          # idiv            # ((hi - lo) / 2)
60          # iadd            # (lo + ((hi - lo) / 2))
36 02       # vstore 2        # mid = (lo + ((hi - lo) / 2));
15 02       # vload 2         # mid
B0          # return          #
 …
```

- Note that
  - ➢ vstore 2 pops **mid** from the stack
  - ➢ vload 2 pushes **mid** back on the stack
  - These two instruction could be optimized away

For didactic reasons,
the compiler does not perform any optimization

Run-time Data Structures

- To run a C0VM bytecode program, the C0VM needs to maintain some **data structures**
  - the bytecode itself
  - the operand stack S
  - the program counter PC
  - the local variables array V
  - …

We will see how later

Added

more to come

Functions

Another Example

- Next, let's compile

```
int next_rand(int last) {
  return last * 1664525 + 1013904223;
}

int main() {
  return next_rand(0xdeadbeef);
}
```

Part of the linear congruential generator

```
C0 C0 FF EE       # magic number
00 17             # version 11, arch = 1 (64 bits)

00 03             # int pool count
# int pool
00 19 66 0D
3C 6E F3 5F
DE AD BE EF

00 00             # string pool total size
# string pool

00 02             # function count
# function_pool

#<main>
00                # number of arguments = 0
00                # number of local variables = 0
00 07             # code length = 7 bytes
13 00 02    # ildc 2          # c[2] = -559038737
B8 00 01    # invokestatic 1  # next_rand(-559038737)
B0          # return          #

#<next_rand>
01                # number of arguments = 1
01                # number of local variables = 1
00 1B             # code length = 11 bytes
15 00       # vload 0   # last
13 00 00    # ildc 0    # c[0] = 1664525
68          # imul      # (last * 1664525)
13 00 01    # ildc 1    # c[1] = 1013904223
60          # iadd      # ((last * 1664525) + 1013904223)
B0          # return    #

00 00               # native count
# native pool
```

- Two novelties
  - large numerical constants
  - function calls

Large Numerical Constants

- bipush only handles constants in the range [-128, 127]
- How to deal with bigger constants?
  - e.g., 0xdeadbeef

  Lots of options
  - have a bytecode instruction that takes 4 bytes as operands
    - e.g., large_push de ad be ef — Not a real C0VM instruction
  - replace the large constant with an expression that evaluates to it
    - e.g., 0xdeadbeef = (0xde << 24) | (0xea << 16) | (0xbe << 8) | 0xef
  - …
- C0VM writes large constants in the **integer pool** and provides an instruction to access them

Integer Pool

- The integer pool is the second segment of a C0VM bytecode
  - it records the number of integers
  - and the integers themselves

- The instruction ildc i pushes the i-th integer on the stack
  - i is given as two bytes
    - there can be up to 2^{16} constants

0x13 ildc <c1,c2> S -> S, x:w32
(x = int_pool[(c1<<8)|c2])

Read the two bytes c1,c2
as a 16-bit unsigned integer
and use that to index the int_pool

```
C0 C0 FF EE       # magic number
00 17             # version 11, arch = 1 (64 bits)

00 03             # int pool count
# int pool
00 19 66 0D
3C 6E F3 5F
DE AD BE EF

00 00             # string pool total size
# string pool

00 02             # function count
# function_pool

#<main>
00                # number of arguments = 0
00                # number of local variables = 0
00 07             # code length = 7 bytes
13 00 02  # ildc 2          # c[2] = -559038737
B8 00 01  # invokestatic 1  # next_rand(-559038737)
B0        # return          #

#<next_rand>
01                # number of arguments = 1
01                # number of local variables = 1
00 1B             # code length = 11 bytes
15 00     # vload 0   # last
13 00 00  # ildc 0    # c[0] = 1664525
68        # imul      # (last * 1664525)
13 00 01  # ildc 1    # c[1] = 1013904223
60        # iadd      # ((last * 1664525) + 1013904223)
B0        # return    #

00 00             # native count
# native pool
```

Integer Pool

- ildc i pushes the i-th integer on the stack

Access the integer at index 2
that's 0xDEADBEEF == -559038737

Access the integer at index 0
that's 0x0019660D == 1664525

Access the integer at index 1
that's 0x3C6EF35F == 1013904223

```
C0 C0 FF EE       # magic number
00 17             # version 11, arch = 1 (64 bits)

00 03             # int pool count
# int pool
00 19 66 0D
3C 6E F3 5F
DE AD BE EF

00 00             # string pool total size
# string pool

00 02             # function count
# function_pool

#<main>
00                # number of arguments = 0
00                # number of local variables = 0
00 07             # code length = 7 bytes
13 00 02    # ildc 2          # c[2] = -559038737
B8 00 01    # invokestatic 1  # next_rand(-559038737)
B0          # return          #

#<next_rand>
01                # number of arguments = 1
01                # number of local variables = 1
00 1B             # code length = 11 bytes
15 00       # vload 0   # last
13 00 00    # ildc 0    # c[0] = 1664525
68          # imul      # (last * 1664525)
13 00 01    # ildc 1    # c[1] = 1013904223
60          # iadd      # ((last * 1664525) + 1013904223)
B0          # return    #

00 00               # native count
# native pool
```

Function Pool

- Functions live in the fourth segment of a C0VM bytecode, the **function pool**
  - it records the number of functions
  - and the functions themselves
    - each function contains the information needed to know where the next function starts

- By convention, main is always the first function

```
C0 C0 FF EE       # magic number
00 17             # version 11, arch = 1 (64 bits)

00 03             # int pool count
# int pool
00 19 66 0D
3C 6E F3 5F
DE AD BE EF

00 00             # string pool total size
# string pool

00 02             # function count
# function_pool

#<main>
00                # number of arguments = 0
00                # number of local variables = 0
00 07             # code length = 7 bytes
13 00 02    # ildc 2          # c[2] = -559038737
B8 00 01    # invokestatic 1  # next_rand(-559038737)
B0          # return          #

#<next_rand>
01                # number of arguments = 1
01                # number of local variables = 1
00 1B             # code length = 11 bytes
15 00       # vload 0   # last
13 00 00    # ildc 0    # c[0] = 1664525
68          # imul      # (last * 1664525)
13 00 01    # ildc 1    # c[1] = 1013904223
60          # iadd      # ((last * 1664525) + 1013904223)
B0          # return    #

00 00             # native count
# native pool
```

Calling Functions

- We call the i-th function in the program with the instruction invokestatic i
  - ➢ say this function is g
  - ○ the arguments of g need to be on the stack
  - ○ when g returns, its returned value is pushed onto the stack in place of the arguments

0xB8 invokestatic <c1,c2> S, v1, v2, …, vn -> S, v
(function_pool[c1<<8|c2] => g, g(v1,...,vn) = v)

0xB0 return ., v -> . (return v to caller)

return expects a single value v on the stack and returns it to the caller
"." denotes the empty stack

Read the two bytes c1,c2 as a 16-bit unsigned integer and use that to index the function_pool

g(v1, … vn) returns v

Calling Functions

- Before calling a function, we need to do some bookkeeping so the caller can resume the execution when it returns
  - save the caller's stack
  - save the caller's local variable array
  - save the caller's program counter
    - specifically the PC of the next instruction to execute
  - save who the caller was

- For this we need a new run-time data structure, the **call stack**
  - the call stack contains a **frame** for each function currently being called
    - each frame contains the above information for this function

Returning from a Functions

- Upon returning from a function, we need restore the contents of the caller's frame
  - its stack
  - its local variable array
  - its program counter
    - specifically the PC of the next instruction to execute
  - which function the caller was

Added

Depending of the implementation, we can either use the caller's index in the function pool, or the caller's bytecode

Bytecode as a Data Structure

Other Segments

- The 5 segment sof a C0VM bytecode file are

1. The **header** contains
   - a 4-byte **magic number**
     - an identifier for C0VM bytecode files
     - a quick way to reject an obviously incorrect tile
   - the **version** of the bytecode and the target **architecture**
     - so that the C0VM implementation matches the bytecode it is executing

The header is largely fixed

```
C0 C0 FF EE       # magic number
00 17             # version 11, arch = 1 (64 bits)

00 03             # int pool count
# int pool
00 19 66 0D
3C 6E F3 5F
DE AD BE EF

00 00             # string pool total size
# string pool

00 02             # function count
# function_pool

#<main>
00                # number of arguments = 0
01                # number of local variables = 1
00 07             # code length = 7 bytes
13 00 02    # ildc 2              # c[2] = -559038737
B8 00 01    # invokestatic 1      # next_rand(-559038737)
B0          # return              #

#<next_rand>
01                # number of arguments = 1
01                # number of local variables = 1
00 1B             # code length = 11 bytes
15 00       # vload 0     # last
13 00 00    # ildc 0      # c[0] = 1664525
68          # imul        # (last * 1664525)
13 00 01    # ildc 1      # c[1] = 1013904223
60          # iadd        # ((last * 1664525) + 1013904223)
B0          # return      #

00 00               # native count
# native pool
```

Other Segments

- The 5 segment of a C0VM bytecode file are

2. The integer pool — see earlier
3. The **string pool**
   - like the integer pool but for strings
4. The function pool — see earlier
5. The **native pool**
   - similar to the function pool but for library functions
     - e.g., print

```
C0 C0 FF EE       # magic number
00 17             # version 11, arch = 1 (64 bits)

00 03             # int pool count
# int pool
00 19 66 0D
3C 6E F3 5F
DE AD BE EF

00 00             # string pool total size
# string pool

00 02             # function count
# function_pool

#<main>
00                # number of arguments = 0
01                # number of local variables = 1
00 07             # code length = 7 bytes
13 00 02    # ildc 2          # c[2] = -559038737
B8 00 01    # invokestatic 1  # next_rand(-559038737)
B0          # return          #

#<next_rand>
01                # number of arguments = 1
01                # number of local variables = 1
00 1B             # code length = 11 bytes
15 00       # vload 0   # last
13 00 00    # ildc 0    # c[0] = 1664525
68          # imul      # (last * 1664525)
13 00 01    # ildc 1    # c[1] = 1013904223
60          # iadd      # ((last * 1664525) + 1013904223)
B0          # return    #

00 00             # native count
# native pool
```

The Bytecode Data Structure

- We can represent a bytecode file in the C0VM as
  - an array of bytes

    C0C0FFEE0017000000000001000000C10031004601005681002 6CB00000

    - accessing specific parts is delicate
      - the 1st function or the 3rd constant in the integer pool
    - easy to get wrong
  - a data structure that reflects the logical organization of a bytecode file
    - segments
    - the various pools
    - …

The Bytecode Data Structure

- *A data structure that reflects the logical organization of a bytecode file*

The int_pool is an array of int_count 32-bit signed integers

The string_pool is an array of string_count chars

The function_pool is an array of function_count struct function_info*s

The native_pool is an array of native_count struct native_info*s

```
struct bc0_file {
  /* header */
  uint32_t magic;
  uint16_t version;

  /* integer constant pool */
  uint16_t int_count;
  int32_t *int_pool;      // \length(int_pool) == int_count

  /* string literal pool */
  /* stores all strings consecutively with NUL terminators */
  uint16_t string_count;
  char *string_pool;      // \length(string_pool) == string_count

  /* function pool */
  uint16_t function_count;
  struct function_info *function_pool; // \length(function_pool) == function_count

  /* native function tables */
  uint16_t native_count;
  struct native_info *native_pool; // \length(native_pool) == native_count
};
```

The Bytecode Data Structure

- *A data structure that reflects the logical organization of a bytecode file*
  - Functions

The code of a function is an array of code_length unsigned bytes

```
struct function_info {
  uint8_t num_args;
  uint8_t num_vars;
  uint16_t code_length;
  ubyte *code;          // \length(code) == code_length
};
```

    - ubyte is defined as uint8_t
  - Native functions
    - we only need to know the number of arguments and how to pass control to it

```
struct native_info {
  uint16_t num_args;
  uint16_t function_table_index;
};
```

The Bytecode Data Structure

- *A data structure that reflects the logical organization of a bytecode file*
- Observe the use of **fixed-size integers**
  - we need to represent specific numbers of bits
    - to match the bytecode file
  - the number of bits of implementation-defined integers may vary

```
struct bc0_file {
  /* header */
  uint32_t magic;
  uint16_t version;

  /* integer constant pool */
  uint16_t int_count;
  int32_t *int_pool;      // \length(int_pool) == int_count

  /* string literal pool */
  /* stores all strings consecutively with NUL terminators *
  uint16_t string_count;
  char *string_pool;      // \length(string_pool) == string_co

  /* function
  uint16_t f
  struct fun

  /* native
  uint16_t
  struct nat
};
```

```
struct function_info {
  uint8_t nu
  uint8_t nu
  uint16_t c
  ubyte *co
};
```

```
struct native_info {
  uint16_t num_args;
  uint16_t function_table_index;
};
```

Jumps

Another Example

- Next, let's compile

```
int main() {
  int sum = 0;
  for (int i = 1; i < 100; i += 2)
    sum += i;
  return sum;
}
```

```
  ...
#<main>
00                  # number of arguments = 0
02                  # number of local variables = 2
00 26               # code length = 38 bytes
10 00         # bipush 0          # 0
36 00         # vstore 0          # sum = 0;
10 01         # bipush 1          # 1
36 01         # vstore 1          # i = 1;
# <00:loop>
15 01         # vload 1           # i
10 64         # bipush 100        # 100
A1 00 06      # if_icmplt +6      # if (i < 100) goto <01:body>
A7 00 14      # goto +20          # goto <02:exit>
# <01:body>
15 00         # vload 0           # sum
15 01         # vload 1           # i
60            # iadd              #
36 00         # vstore 0          # sum += i;
15 01         # vload 1           # i
10 02         # bipush 2          # 2
60            # iadd              #
36 01         # vstore 1          # i += 2;
A7 FF E8      # goto -24          # goto <00:loop>
# <02:exit>
15 00         # vload 0           # sum
B0            # return            #
  ...
```

- Novelty: loops
  - conditionals are handled similarly

Branch Instructions

```
int main() {
  int sum = 0;
  for (int i = 1; i < 100; i += 2)
    sum += i;
  return sum;
}
```

- Conditionals and loops are transformed into branch instructions
  - **Conditional branch instructions**
    - ➢ jump to a specific point in the bytecode if the top values of the stack satisfy a condition
      (go to the next instruction otherwise)
    - ➢ e.g., if_cmpeq +9
      - ❑ jump 9 bytes forward if the top two values on the stack are equal
      - ❑ go to the next instruction otherwise
  - **Unconditional branch instruction**
    - ➢ always jump to a specific point in the bytecode
    - ➢ e.g., goto -24
      - ❑ jump 24 bytes backward

```
  …
#<main>
00                  # number of arguments = 0
02                  # number of local variables = 2
00 26               # code length = 38 bytes
10 00       # bipush 0          # 0
36 00       # vstore 0          # sum = 0;
10 01       # bipush 1          # 1
36 01       # vstore 1          # i = 1;
# <00:loop>
15 01       # vload 1           # i
10 64       # bipush 100        # 100
A1 00 06    # if_icmplt +6      # if (i < 100) goto <01:body>
A7 00 14    # goto +20          # goto <02:exit>
# <01:body>
15 00       # vload 0           # sum
15 01       # vload 1           # i
60          # iadd              #
36 00       # vstore 0          # sum += i;
15 01       # vload 1           # i
10 02       # bipush 2          # 2
60          # iadd              #
36 01       # vstore 1          # i += 2;
A7 FF E8    # goto -24          # goto <00:loop>
# <02:exit>
15 00       # vload 0           # sum
B0          # return            #
  …
```

Branch Instructions

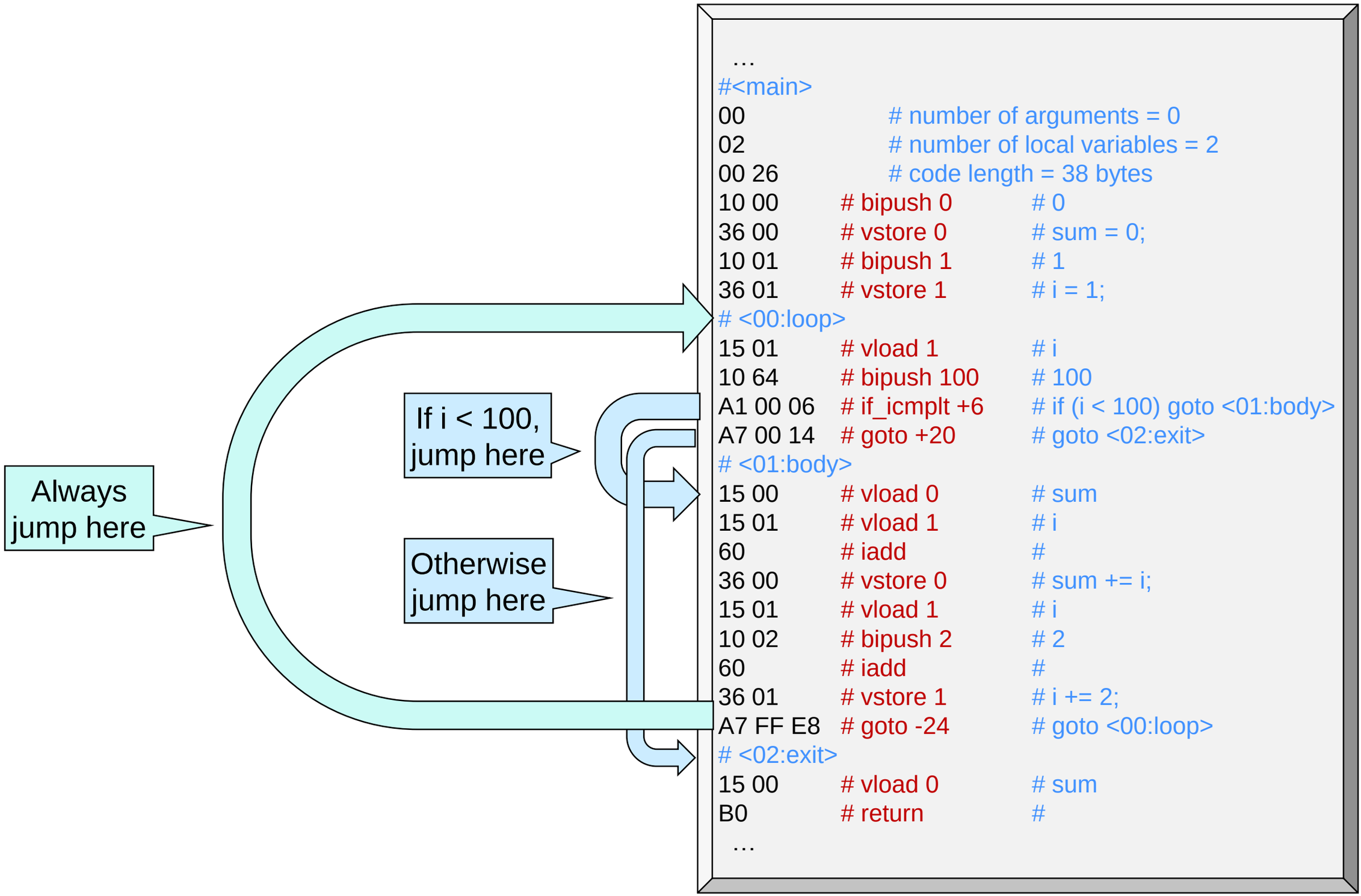
- Examples

0xA1	if_cmplt <o1,o2>	S, x:w32, y:w32 -> S	(pc = pc + (o1<<8|o2) if x < y)
0xA7	goto <o1,o2>	S -> S	(pc = pc + (o1<<8|o2))

  - <o1,o2> is a 16-bit signed **offset**
    - it specifies by how many bytes to jump
      - forward – if positive
      - backward – if negative
    - it jumps **bytes**, not instructions

Branch Instructions

```
int main() {
  int sum = 0;
  for (int i = 1; i < 100; i += 2)
    sum += i;
  return sum;
}
```



Structs

Another Example

- Next, let's compile

```
typedef struct list_node list;
struct list_node {
  char data;
  list* next;
};

list* prepend(list* l, char c) {
  list* res = alloc(list);
  res->data = c;
  res->next = l;
  return res;
}

int main() {
  list* l = prepend(NULL, 'a');
  return 0;
}
```

```
  …
#<main>
00              # number of arguments = 0
03              # number of local variables = 3
00 0B           # code length = 11 bytes
01          # aconst_null       # NULL
10 61       # bipush 97         # 'a'
B8 00 01    # invokestatic 1    # prepend(NULL, 'a')
36 00       # vstore 0          # l = prepend(NULL, 'a');
10 00       # bipush 0          # 0
B0          # return            #

#<prepend>
02              # number of arguments = 2
03              # number of local variables = 3
00 15           # code length = 21 bytes
BB 10       # new 16            # alloc(list)
36 02       # vstore 2          # res = alloc(list);
15 02       # vload 2           # res
62 00       # aaddf 0           # &res->data
15 01       # vload 1           # c
55          # cmstore           # res->data = c;
15 02       # vload 2           # res
62 08       # aaddf 8           # &res->next
15 00       # vload 0           # l
4F          # amstore           # res->next = l;
15 02       # vload 2           # res
B0          # return            #
  …
```

- Novelty: allocated memory
  - pointers
  - structs
- Also: characters

Characters

- Characters are represented as their ASCII value

The ASCII value of 'a' is 97 in decimal (0x61 in hex)

```
  …
#<main>
00              # number of arguments = 0
03              # number of local variables = 3
00 0B           # code length = 11 bytes
01          # aconst_null       # NULL
10 61       # bipush 97         # 'a'
B8 00 01    # invokestatic 1    # prepend(NULL, 'a')
36 00       # vstore 0          # l = prepend(NULL, 'a');
10 00       # bipush 0          # 0
B0          # return            #

#<prepend>
02              # number of arguments = 2
03              # number of local variables = 3
00 15
BB 10
36 02
15 02
62 00
15 01
55
15 02       # vload 2           # res
62 08       # aaddf 8           # &res->next
15 00       # vload 0           # l
4F          # amstore           # res->next = l;
15 02       # vload 2           # res
B0          # return            #
  …
```

```
…
int main() {
  list* l = prepend(NULL, 'a');
  return 0;
}
```

- On the operand stack, they are treated as 32-bit integers
  - even if a char is just 1 byte long

- Booleans too are treated as 32-bit integers
  - true as 1
  - false as 0
  - even if a bool is just 1 bit long

Pointers

- Pointers are represented as 8 bytes (unsigned)
  - NULL is 0x0000000000000000
- The instruction aconst_null loads NULL on the stack

0x01 aconst_null S -> S, null:*

- The operand stack can contain
  - 64-bit pointers — denoted x:*
  - 32-bit integers — denoted x:w32

```
  …
#<main>
00              # number of arguments = 0
03              # number of local variables = 3
00 0B           # code length = 11 bytes
01          # aconst_null       # NULL
10 61       # bipush 97         # 'a'
B8 00 01    # invokestatic 1    # prepend(NULL, 'a')
36 00       # vstore 0          # l = prepend(NULL, 'a');
10 00       # bipush 0          # 0
B0          # return            #

#<prepend>
02              # number of arguments = 2
03              # number of local variables = 3
00 15
BB 10
36 02
15 02
62 00
15 01
55
15 02       # vload 2           # res
62 08       # aaddf 8           # &res->next
15 00       # vload 0           # l
4F          # amstore           # res->next = l;
15 02       # vload 2           # res
B0          # return            #
  …
```

```
…
int main() {
  list* l = prepend(NULL, 'a');
  return 0;
}
```

Recall

0x60 iadd S, x:w32, y:w32 -> S, x+y:w32

c0_value

- The operand stack can contain
  - 64-bit pointers
  - 32-bit integers
- As an abstraction, we write **c0_value** as the type of stack elements
  - a union type discriminated by an enum type
  - four coercion functions allow us to go back and forth

```
c0_value int2val(int32_t i);
int32_t val2int(c0_value v);
c0_value ptr2val(void *p);
void *val2ptr(c0_value v);
```

fails if, in reality, v is not a 32-bit integer

fails if, in reality, v is not a pointer

Memory Allocation

- The C0 instruction alloc(tp) is compiled into new s
  - where s is the number of bytes needed to represent type tp

This is determined by the compiler

```
…
#<main>
00
03
00 0B
01
10 61
B8 00 01
36 00
10 00
B0

#<preper
02
03                      # number of local variables = 3
00 15                   # code length = 21 bytes
BB 10     # new 16      # alloc(list)
36 02     # vstore 2    # res = alloc(list);
15 02     # vload 2     # res
62 00     # aaddf 0     # &res->data
15 01     # vload 1     # c
4F        # amp         # res->data = c;
15 02     # vload 2     # res
62 08     # aaddf 8     # &res->next
15 00     # vload 0     # l
4F        # amp         # res->next = l;
15 02     # vload 2     # res
B0        # return      #
…
```

```
typedef struct list_node list;
struct list_node {
  char data;
  list* next;
};

list* prepend(list* l, char c) {
  list* res = alloc(list);
  …
```

0xBB new <s> S -> S , a:* (*a is now allocated, size <s>)

- the address of the new memory is pushed onto the stack

Why 16 and not 9 or even 12?
The code to access a node can be more efficient in this way

Fields

- A field in a struct is compiled into an **offset** relative to the start of the struct
  - ➢ that's the number of bytes to skip over before we find that field

- aaddf f pops an address a from the stack and pushes the address that is f bytes after a

```
typedef struct list_node list;
struct list_node {
  char data;
  list* next;
};

list* prepend(list* l, char c) {
  list* res = alloc(list);
  res->data = c;
  res->next = l;
…
```

```
…
#<main>
00
03                                                    3
00 0B
01
10 61                                                 'a')
B8 00 01                                              LL, 'a');
36 00
10 00
B0

#<prepe
02
03                                                    3
00 15          # code length = 21 bytes
BB 10          # new 16              # alloc(list)
36 02          # vstore 2            # res = alloc(list);
15 02          # vload 2             # res
62 00          # aaddf 0             # &res->data
15 01          # vload 1             # c
55             # smstore             # res->data = c;
15 02          # vload 2             # res
62 08          # aaddf 8             # &res->next
15 00          # vload 0             # l
…
```

0x62 aaddf <f> S, a:* -> S, (a+f):* (a != NULL; f field offset in bytes)

The offset f is determined by the compiler

The **data** field is 0 bytes inside the struct

The **next** field is 8 bytes inside the struct

The compiler decided it is best to use 8 bytes for the data field (even if it's a char)

Manipulating Heap Values

- Heap values need to be
  - read onto the stack
  - written into the heap
- What C0VM instruction to use depends on the size of the value *on the heap*
  - a pointer is 8 bytes
  - an int is 4 bytes
  - a char or bool is 1 byte
    - when stored in the heap
    - on the stack they take up 32 bits

```
...
#<main>                                                    3
00
03
00 0B
01
10 61                                                      'a')
B8 00 01                                                   LL, 'a');
36 00
10 00
B0

#<prepe                                                    3
02
03
00 15          # code length = 21 bytes
BB 10          # new 16              # alloc(list)
36 02          # vstore 2            # res = alloc(list);
15 02          # vload 2             # res
62 00          # aaddf 0             # &res->data
15 01          # vload 1             # c
55             # cmstore             # res->data = c;
15 02          # vload 2             # res
62 08          # aaddf 8             # &res->next
15 00          # vload 0             # l
4F             # amstore             # res->next = l;
15 02          # vload 2             # res
B0             # return              #
...
```

```
typedef struct list_node list;
struct list_node {
  char data;
  list* next;
};

list* prepend(list* l, char c) {
  list* res = alloc(list);
  res->data = c;
  res->next = l;
...
```

Manipulating Heap Values

- *What C0VM instruction to use depends on the size of the value on the heap*
  - an int is 4 bytes

Read **4 bytes** from address a and
push them onto the stack as a 32-bit integer x

int i = *p;

0x2E	imload	S, a:* -> S, x:w32	(x = *a, a != NULL, load 4 bytes)
0x4E	imstore	S, a:*, x:w32 -> S	(*a = x, a != NULL, store 4 bytes)

Pop the 32-bit integer x on the top of the stack
and write it as **4 bytes** at address a

*p = 15122;

Manipulating Heap Values

- *What C0VM instruction to use depends on the size of the value on the heap*
  - a char or bool is 1 byte

Read **1 byte** from address a and
push it onto the stack as a 32-bit integer x

char c = res->data;

0x34	cmload	S, a:* -> S, x:w32	(x = (w32)(*a), a != NULL, load 1 byte)
0x55	cmstore	S, a:*, x:w32 -> S	(*a = x & 0x7f, a != NULL, store 1 byte)

Pop the 32-bit integer x on the top of the stack
and write it as **1 byte** at address a

res->data = c;

Keep just the
7 rightmost bits

Because the range of
C0 chars is [0, 128)

Manipulating Heap Values

- *What C0VM instruction to use depends on the size of the value on the heap*
  - a pointer is 8 bytes

Read **8 bytes** from address a and
push it onto the stack as a 64-bit pointer b

list* l = res->next;

0x2F	amload	S, a:* -> S, b:*	(b = *a, a != NULL, load address)
0x4F	amstore	S, a:*, b:* -> S	(*a = b, a != NULL, store address)

Pop the 64-bit pointer b on the top of the stack
and write it as **8 bytes** at address a

res->next = l;

Compiled Example

```
…
list* prepend(list* l, char c) {
  list* res = alloc(list);
  res->data = c;
  res->next = l;
  return l;
}
…
```

Procure 16 bytes

Go to its 1st byte

Store the character on the top of the stack there

Go to its 9th byte

Store the pointer on the top of the stack there

```
  …
#<main>
00              # number of arguments = 0
03              # number of local variables = 3
00 0B           # code length = 11 bytes
01          # aconst_null       # NULL
10 61       # bipush 97         # 'a'
B8 00 01    # invokestatic 1    # prepend(NULL, 'a')
36 00       # vstore 0          # l = prepend(NULL, 'a');
10 00       # bipush 0          # 0
B0          # return            #

#<prepend>
02              # number of arguments = 2
03              # number of local variables = 3
00 15           # code length = 21 bytes
BB 10       # new 16            # alloc(list)
36 02       # vstore 2          # res = alloc(list);
15 02       # vload 2           # res
62 00       # aaddf 0           # &res->data
15 01       # vload 1           # c
55          # cmstore           # res->data = c;
15 02       # vload 2           # res
62 08       # aaddf 8           # &res->next
15 00       # vload 0           # l
4F          # amstore           # res->next = l;
15 02       # vload 2           # res
B0          # return            #
  …
```

Arrays

Another Example

- Next, let's compile

```
int main() {
  int[] A = alloc_array(int, 100);
  for (int i = 0; i < 100; i++)
    A[i] = i;
  return A[99];
}
```

- Novelty: arrays
  - alloc_array
  - array accesses

```
 …
#<main>
00                  # number of arguments = 0
02                  # number of local variables = 2
00 2D               # code length = 45 bytes
10 64         # bipush 100    # 100
BC 04         # newarray 4    # alloc_array(int, 100)
36 00         # vstore 0      # A = alloc_array(int, 100);
10 00         # bipush 0      # 0
36 01         # vstore 1      # i = 0;
# <00:loop>
15 01         # vload 1       # i
10 64         # bipush 100    # 100
A1 00 06      # if_icmplt +6  # if (i < 100) goto <01:body>
A7 00 15      # goto +21      # goto <02:exit>
# <01:body>
15 00         # vload 0       # A
15 01         # vload 1       # i
63            # aadds         # &A[i]
15 01         # vload 1       # i
4E            # imstore       # A[i] = i;
15 01         # vload 1       # i
10 01         # bipush 1      # 1
60            # iadd          #
36 01         # vstore 1      # i += 1;
A7 FF E7      # goto -25      # goto <00:loop>
# <02:exit>
15 00         # vload 0       # A
10 63         # bipush 99     # 99
63            # aadds         # &A[99]
2E            # imload        # A[99]
B0            # return        #
 …
```

Allocating Arrays

- The instruction alloc_array(tp, n) is compiled into newarray s
  - where s is the number of bytes needed to represent type tp

This is determined by the compiler

  - the number of elements of the array is at the top of the stack

```
  …
#<main>
00              # number of arguments = 0
02              # number of local variables = 2
00 2D           # code length = 45 bytes
10 64       # bipush 100    # 100
BC 04       # newarray 4    # alloc_array(int, 100)
36 00       # vstore 0      # A = alloc_array(int, 100);
10 00       # bipush 0      # 0
36 01
# <00:l
15 01
10 64
A1 00
A7 00 15    # goto +21      # goto <02:exit>
# <01:body>
15 00       # vload 0       # A
15 01       # vload 1       # i
63          # aadds         # &A[i]
15 01       # vload 1       # i
4E          # imstore       # A[i] = i;
15 01       # vload 1       # i
10 01       # bipush 1      # 1
60          # iadd          #
36 01       # vstore 1      # i += 1;
63          # aadds         # &A[99]
2E          # imload        # A[99]
B0          # return        #
  …
```

```
…
int[] A = alloc_array(int, 100);
…
```

0xBC newarray <s> S, n:w32 -> S, a:* (a[0..n) now allocated, each array element has size <s>)

Accessing Arrays

- Array elements are accessed with aadds s
  - where s is the size (in bytes) of each array element
    - ➢ determined by the compiler
  - and the index of the element is at the top of the stack

```
 …
#<main>
00                  # number of arguments = 0
02                  # number of local variables = 2
00 2D               # code length = 45 bytes
10 64          # bipush 100    # 100
BC 04                                          (int, 100)
36 00                                          rray(int, 100);
10 00
36 01
# <00:l
15 01
10 64
A1 00 06       # if_icmplt +6  # if (i < 100) goto <01:body>
A7 00 15       # goto +21      # goto <02:exit>
# <01:body>
15 00          # vload 0       # A
15 01          # vload 1       # i
63             # aadds         # &A[i]
15 01          # vload 1       # i
A7 FF E7       # goto -25      # goto <00:loop>
# <02:exit>
15 00          # vload 0       # A
10 63          # bipush 99     # 99
63             # aadds         # &A[99]
2E             # imload        # A[99]
B0             # return        #
 …
```

```
 …
for (int i = 0; i < 100; i++)
  A[i] = i;
 return A[99];
 …
```

0x63 aadds S, a:*, i:w32 -> S, (elems(a)+s*i):*

(a != NULL, 0 <= i < \length(a))

a may not be where the elements are stored because we need to store the length of the array

s is the size of a's elements

Accessing Arrays

- The elements themselves are
  - read with cmload, imload, amload
  - written with cmstore, imstore, amstore

  depending on their size

```
  …
#<main>
00                  # number of arguments = 0
02                  # number of local variables = 2
00 2D               # code length = 45 bytes
10 64         # bipush 100    # 100
BC 04                                          (int, 100)
36 00                                          rray(int, 100);
10 00
36 01
# <00:l
15 01
10 64
A1 00 06      # if_icmplt +6  # if (i < 100) goto <01:body>
A7 00 15      # goto +21      # goto <02:exit>
# <01:body>
15 00         # vload 0       # A
15 01         # vload 1       # i
63            # aadds         # &A[i]
15 01         # vload 1       # i
4E            # imstore       # A[i] = i;
15 01         # vload 1       # i
10 01         # bipush 1      # 1
60            # iadd          #
36 01         # vstore 1      # i += 1;
A7 FF E7      # goto -25      # goto <00:loop>
# <02:exit>
15 00         # vload 0       # A
10 63         # bipush 99     # 99
63            # aadds         # &A[99]
2E            # imload        # A[99]
B0            # return        #
  …
```

```
  …
for (int i = 0; i < 100; i++)
  A[i] = i;
  return A[99];
  …
```

Strings

Another Example

- Next, let's compile

```
#use <string>
#use <conio>

int main() {
  string h = "Hello ";
  string hw = string_join(h, "World!\n");
  print(hw);
  return string_length(hw);
}
```

- Novelty:
  - strings
  - system libraries

```
C0 C0 FF EE       # magic number
00 17             # version 11, arch = 1 (64 bits)

00 00             # int pool count
# int pool

00 0F             # string pool total size
# string pool
48 65 6C 6C 6F 20 00          # "Hello "
57 6F 72 6C 64 21 0A 00       # "World!\n"

00 01             # function count
# function_pool

#<main>
00                # number of arguments = 0
02                # number of local variables = 2
00 1B             # code length = 27 bytes
14 00 00    # aldc 0            # s[0] = "Hello "
36 00       # vstore 0          # h = "Hello ";
15 00       # vload 0           # h
14 00 07    # aldc 7            # s[7] = "World!\n"
B7 00 00    # invokenative 0    # string_join(h, "World!\n")
36 01       # vstore 1          # hw = string_join(h, "World!\n");
15 01       # vload 1           # hw
B7 00 01    # invokenative 1    # print(hw)
57          # pop               # (ignore result)
15 01       # vload 1           # hw
B7 00 02    # invokenative 2    # string_length(hw)
B0          # return            #

00 03             # native count
# native pool
00 02 00 64    # string_join
00 01 00 06    # print
00 01 00 65    # string_length
```

Strings

- String ***literals*** are stored in the **string pool**
  - one after the other
  - each is NUL-terminated

- Computed strings
  - e.g., using string_join

live on the heap

  - the details are abstracted away

```
C0 C0 FF EE       # magic number
00 17             # version 11, arch = 1 (64 bits)

00 00             # int pool count
# int pool

00 0F             # string pool total size
# string pool
48 65 6C 6C 6F 20 00          # "Hello "
57 6F 72 6C 64 21 0A 00       # "World!\n"

00 01             # function count
# function_pool

#<m
00
02
00 1
14 0
36 0
15 00       # vload 0           # h
14 00 07    # aldc 7            # s[7] = "World!\n"
B7 00 00    # invokenative 0    # string_join(h, "World!\n")
36 01       # vstore 1          # hw = string_join(h, "World!\n");
15 01       # vload 1           # hw
B7 00 01    # invokenative 1    # print(hw)
57          # pop               # (ignore result)
15 01       # vload 1           # hw
B7 00 02    # invokenative 2    # string_length(hw)
B0          # return            #

00 03             # native count
# native pool
00 02 00 64    # string_join
00 01 00 06    # print
00 01 00 65    # string_length
```

```
…
int main() {
  string h = "Hello ";
  string hw = string_join(h, "World!\n");
  …
```

Strings

- String ***literals*** are accessed with aldc

0x14 aldc <c1,c2> S -> S, a:* (a = &string_pool[c1<<8|c2])

  - the operand is the byte offset in the string pool
  - it pushes on the stack the address of the 1st character of the string

```
C0 C0 FF EE       # magic number
00 17             # version 11, arch = 1 (64 bits)

00 00             # int pool count
# int pool

00 0F             # string pool total size
# string pool
48 65 6C 6C 6F 20 00          # "Hello "
57 6F 72 6C 64 21 0A 00       # "World!\n"

00 01             # function count
#<main>
00                # number of arguments = 0
02                # number of local variables = 2
00 1B             # code length = 27 bytes
14 00 00    # aldc 0            # s[0] = "Hello "
36 00       # vstore 0          # h = "Hello ";
15 00       # vload 0           # h
14 00 07    # aldc 7            # s[7] = "World!\n"
B7 00 00    # invokenative 0    # string_join(h, "World!\n")
36 01       # vstore 1          # hw = string_join(h, "World!\n");
15 0
B7 0
57
15 0
B7 0
B0

00 03             # native count
# native pool
00 02 00 64     # string_join
00 01 00 06     # print
00 01 00 65     # string_length
```

```
…
int main() {
  string h = "Hello ";
  string hw = string_join(h, "World!\n");
  …
```

Native Functions

- System library functions
  - ➢ e.g., print, provided by <conio>

  are noted in the **native pool**
  - ○ one entry for each native function called in the program
  - ○ each entry is 4 bytes long

    nn nn aa aa
    - ➢ nn nn is the number of arguments
    - ➢ aa aa is where to find the function

```
C0 C0 FF EE       # magic number
00 17             # version 11, arch = 1 (64 bits)

00 00             # int pool count
# int pool

00 0F             # string pool total size
# string pool
48 65 6C 6C 6F 20 00          # "Hello "
57 6F 72 6C 64 21 0A 00       # "World!\n"

00 01             # function count
# function pool

#<m
00
02
00 1
14 0
36 0
15 00       # vload 0          # h
14 00 07    # aldc 7           # s[7] = "World!\n"
B7 00 00    # invokenative 0   # string_join(h, "World!\n")
36 01       # vstore 1         # hw = string_join(h, "World!\n");
15 01       # vload 1          # hw
B7 00 01    # invokenative 1   # print(hw)
57          # pop              # (ignore result)
15 01       # vload 1          # hw
B7 00 02    # invokenative 2   # string_length(hw)
B0          # return           #

00 03             # native count
# native pool
00 02 00 64     # string_join
00 01 00 06     # print
00 01 00 65     # string_length
```

```
…
  string hw = string_join(h, "World!\n");
  print(hw);
  return string_length(hw);
  …
```

Calling Native Functions

0xB7 invokenative <c1,c2> S, v1, v2, …, vn -> S, v

(native_pool[c1<<8|c2] => g, g(v1,...,vn) = v)

g(v1, … vn) returns v

Read the two bytes c1,c2 as a 16-bit unsigned integer and use that to index the native_pool

- System functions are called with invokenative
  - similar to invokestatic

```
C0 C0 FF EE       # magic number
00 17             # version 11, arch = 1 (64 bits)

00 00             # int pool count
# int pool

00 01             # function count
# function_pool

#<m
00
02
00 1
14 0
36 0
15 00       # vload 0          # h
14 00 07    # aldc 7           # s[7] = "World!\n"
B7 00 00    # invokenative 0   # string_join(h, "World!\n")
36 01       # vstore 1         # hw = string_join(h, "World!\n");
15 01       # vload 1          # hw
B7 00 01    # invokenative 1   # print(hw)
57          # pop              # (ignore result)
15 01       # vload 1          # hw
B7 00 02    # invokenative 2   # string_length(hw)
B0          # return           #

00 03             # native count
# native pool
00 02 00 64       # string_join
00 01 00 06       # print
00 01 00 65       # string_length
```

```
…
  string hw = string_join(h, "World!\n");
  print(hw);
  return string_length(hw);
  …
```

Contracts

One Last Example

- Next, let's compile

```
int main() {
  int[] A = alloc_array(int, 7);
  //@assert(\length(A) == 7);
  return 0;
}
```

  - with cc0 -d

- Novelty: contracts

```
…
00 2B                # string pool total size
# string pool
65 78 34 2E 63 30 3A 33 2E 36 2D 33 2E 33 30 3A 20 40 61 73 73 65 72 74 20
61 6E 6E 6F 74 61 74 69 6F 6E 20 66 61 69 6C 65 64 00  # "ex4.c0:3.6-3.30:
@assert annotation failed"

…
00 01                # function count
# function_pool

#<main>
00                   # number of arguments = 0
01                   # number of local variables = 1
00 1F                # code length = 31 bytes
10 07          # bipush 7        # 7
BC 04          # newarray 4      # alloc_array(int, 7)
36 02          # vstore 0        # A = alloc_array(int, 7);
15 02          # vload 0         # A
BE             # arraylength     # \length(A)
10 07          # bipush 7        # 7
9F 00 06       # if_cmpeq +6     # if (\length(A) == 7) goto <00:cond_true>
A7 00 08       # goto +8         # goto <01:cond_false>
# <00:cond_true>
10 01          # bipush 1        # true
A7 00 05       # goto +5         # goto <02:cond_end>
# <01:cond_false>
10 00          # bipush 0        # false
# <02:cond_end>
14 00 07       # aldc 0          # s[0] = "ex4.c0:3.6-3.30: @assert annotation failed"
CF             # assert          # assert(\length(A) == 7) [failure message on stack]
10 00          # bipush 0        # 0
B0             # return          #
…
```

Slightly simplified

Contracts

- When compiled with -d, contracts are turned into conditionals
  - jumps in C0VM

- True for all contracts
  - //@requires
  - //@ensures
  - //@loop_invariant
  - //@assert
  - and even assert

```
…
00 2B                # string pool total size
# string pool
65 78 34 2E 63 30 3A 33 2E 36 2D 33 2E 33 30 3A 20 40 61 73 73 65 72 74 20
61 6E 6E 6F 74 61 74 69 6F 6E 20 66 61 69 6C 65 64 00  # "ex4.c0:3.6-3.30:
@assert annotation failed"

…
00 01                # function count
# function_pool

#<main>
00                   # number of arguments
01                   # number of local variables = 1
00 1F                # code length = 31 bytes
10 07        # bipush 7        # 7
BC 04        # newarray 4      # alloc_array(int, 7)
36 02        # vstore 0        # A = alloc_array(int, 7);
15 02        # vload 0         # A
BE           # arraylength     # \length(A)
10 07        # bipush 7        # 7
9F 00 06     # if_cmpeq +6     # if (\length(A) == 7) goto <00:cond_true>
A7 00 08     # goto +8         # goto <01:cond_false>
# <00:cond_true>
10 01        # bipush 1        # true
A7 00 05     # goto +5         # goto <02:cond_end>
# <01:cond_false>
10 00        # bipush 0        # false
# <02:cond_end>
14 00 07     # aldc 0          # s[0] = "ex4.c0:3.6-3.30: @assert annotation failed"
CF           # assert          # assert(\length(A) == 7) [failure message on stack]
10 00        # bipush 0        # 0
B0           # return          #
…
```

```
int main() {
  int[] A = alloc_array(int, 7);
  //@assert(\length(A) == 7);
  return 0;
}
```

Contracts

- Contracts are handled by assert
- The stack contains
  - a boolean x
    - aborts execution if x is false
  - a pointer a to a string
    - the error message to display when aborting

```
…
00 2B                # string pool total size
# string pool
65 78 34 2E 63 30 3A 33 2E 36 2D 33 2E 33 30 3A 20 40 61 73 73 65 72 74 20
61 6E 6E 6F 74 61 74 69 6F 6E 20 66 61 69 6C 65 64 00  # "ex4.c0:3.6-3.30:
@assert annotation failed"

…
00 01                # function count
# function_pool

#<main>
00                   # number of arguments
01                   # number of local variables = 1
00 1F                # code length = 31 bytes
10 07       # bipush 7       # 7
BC 04       # newarray 4     # alloc_array(int, 7)
36 02       # vstore 0       # A = alloc_array(int, 7);
15 02       # vload 0        # A
BE          # arraylength    # \length(A)
10 07       # bipush 7       # 7
9F 00 06    # if_cmpeq +6    # if (\length(A) == 7) goto <00:cond_true>
A7 00 08    # goto +8        # goto <01:cond_false>
# <00:cond_true>
10 01       # bipush 1       # true
A7 00 05    # goto +5        # goto <02:cond_end>
# <01:cond_false>
10 00       # bipush 0       # false
# <02:cond_end>
14 00 07    # aldc 0         # s[0] = "ex4.c0:3.6-3.30: @assert annotation failed"
CF          # assert         # assert(\length(A) == 7) [failure message on stack]
10 00       # bipush 0       # 0
```

```
int main() {
  int[] A = alloc_array(int, 7);
  //@assert(\length(A) == 7);
  return 0;
}
```

0xCF assert S, x:w32, a:* -> S (c0_assertion_failure(a) if x == 0)

Contracts

- The stack contains
  - a boolean x
    - ➢ aborts execution if x is false
- x is determined by the value of the contract expression
  - \length is handled by the C0VM instruction arraylength

```
…
00 2B                # string pool total size
# string pool
65 78 34 2E 63 30 3A 33 2E 36 2D 33 2E 33 30 3A 20 40 61 73 73 65 72 74 20
61 6E 6E 6F 74 61 74 69 6F 6E 20 66 61 69 6C 65 64 00  # "ex4.c0:3.6-3.30:
@assert annotation failed"

…
00 01                # function count
# function_pool

#<main>
00                   # number of arguments
01                   # number of local variables = 1
00 1F                # code length = 31 bytes
10 07       # bipush 7       # 7
BC 04       # newarray 4     # alloc_array(int, 7)
36 02       # vstore 0       # A = alloc_array(int, 7);
15 02       # vload 0        # A
BE          # arraylength    # \length(A)
10 07       # bipush 7       # 7
9F 00 06    # if_cmpeq +6    # if (\length(A) == 7) goto <00:cond_true>
A7 00 08    # goto +8        # goto <01:cond_false>
# <00:cond_true>
10 01       # bipush 1       # true
A7 00 05    # goto +5        # goto <02:cond_end>
# <01:cond_false>
10 00       # bipush 0       # false
# <02:cond_end>
                                                  failed"
                                                  stack]
B0          # return         #
…
```

```
int main() {
  int[] A = alloc_array(int, 7);
  //@assert(\length(A) == 7);
  return 0;
}
```

0xBE arraylength S, a:* -> S, n:w32 (n = \length(a))