CS107: Virtual Machines

Guannan Wei
guannan.wei@tufts.edu

April 2, 2026

Spring 2026

Tufts University

Implementing a VM in C

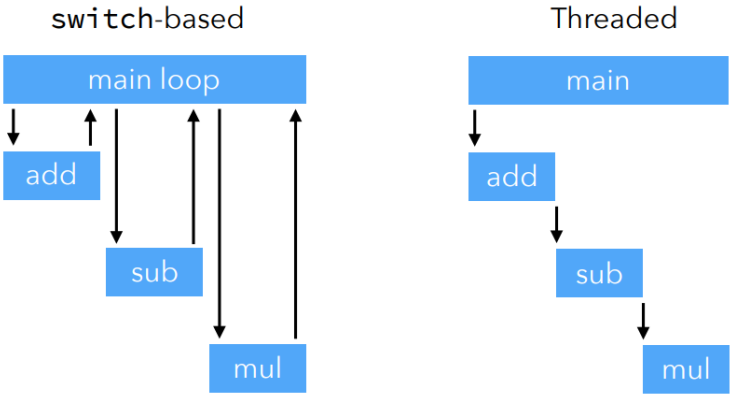
```c
typedef enum {
  add, /* ... */
} instruction_t;

void interpret() {
  static instruction_t program[] = { add /* ... */ };
  instruction_t* pc = program;
  int* sp = ...; /* stack pointer */
  for (;;) {
    switch (*pc++) {
      case add:
        sp[1] += sp[0];
        sp++;
        break;
      /* ... other instructions */
    }
  }
}
```

Today

- Improving performance of virtual machines:
  - Threaded code
  - Top-of-stack caching
  - Super-instructions
  - JIT compilation
- Introduce the MiniScala VM

Threaded Code

Program: `add sub mul`



Implementing Threaded Code

To implement threaded code, one technique is known as **direct threading** where *instructions are pointers to the code handling them*.

Depending on how we represent and invoke instructions, there are other variants, such as indirect threading, where instructions index an array containing pointers to the code handling them.

Threaded Code in C

To implement threaded code, it must be possible to manipulate code pointers. How can this be achieved in C?

- In ANSI C, the only way to do this is to use function pointers.
- GCC further allows the manipulation of labels as values, which is much more efficient!

Direct Threading in ANSI C

Idea: Define one function per VM instruction.

- The program can then simply be represented as an array of function/code pointers.
- Additional code is inserted at the end of every function to call the function handling the next VM instruction.

Direct Threading in ANSI C

```
typedef void (*instruction_t)();
static instruction_t* pc;
static int* sp = ...;
/* function implementing the add instruction */
static void add() {
  sp[1] += sp[0];
  ++sp;
  (*++pc)(); /* handle next instruction */
}
/* ... other instructions */
/* representing the program as an array of functions */
static instruction_t program[] = { add, /* ... */ };
void interpret() {
  sp = ...;
  pc = program;
  (*pc)(); /* handle first instruction */
}
```

Direct Threading in ANSI C

Implementing direct threading in ANSI C is easy, but there are two major problems with this approach:

Direct Threading in ANSI C

Implementing direct threading in ANSI C is easy, but there are two major problems with this approach:

- It is not very efficient, because function calls are expensive.
- It leads to stack overflow very quickly unless the compiler implements tail-call elimination. The function call at the end of **add** and other instruction handlers is a tail call, and should be optimized as a jump.

Direct Threading in ANSI C

Implementing direct threading in ANSI C is easy, but there are two major problems with this approach:

- It is not very efficient, because function calls are expensive.
- It leads to stack overflow very quickly unless the compiler implements tail-call elimination. The function call at the end of **add** and other instruction handlers is a tail call, and should be optimized as a jump.

Tail-call elimination (TCE) ensures for tail calls only constant stack space is used, and thus allows to run programs with deep call stacks without overflowing the stack.

While recent versions of GCC perform tail-call elimination in some cases, not all compilers do. Without TCE compilers, the only option is to eliminate tail calls by hand (e.g., using trampolines).

Direct Threading with GCC

The GNU C Compiler (GCC) – and others like Clang – offers an extension that is very useful to implement direct threading: labels can be treated as values and a `goto` instruction can jump to a computed label.

With this extension, the program can be represented as an array of labels, and jumping to the next instruction is achieved by a goto to the label currently referred to by the program counter.

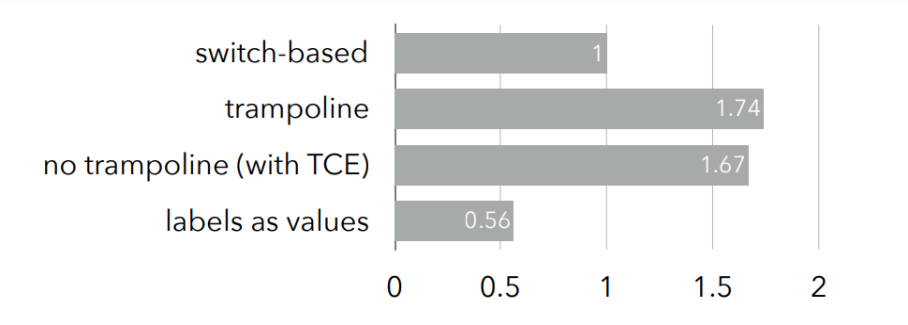
Direct Threading with GCC

```
void interpret() {
                    /* label as value, of type void * */
  void* program[] = { &&l_add, /* ... */ };
  int* sp = ...;
  void** pc = program;
  /* computed goto */
  goto **pc; /* jump to first instruction */

l_add:
  sp[1] += sp[0];
  ++sp;
  goto **(++pc); /* jump to next instruction */
  /* ... other instructions */
}
```

Threading Benchmark

The benchmark below compares several versions of a small interpreter measured while interpreting 500,000,000 iterations of a simple loop. The code was compiled using Clang v503.0.38 with full optimizations and run on an Intel Core i5.

The normalized times are presented below.



Optimizing VMs

The basic, `switch`-based implementation of the virtual machine just presented can be made faster using several techniques:

- Threaded code
- *Top-of-stack caching*
- Super-instructions
- JIT compilation

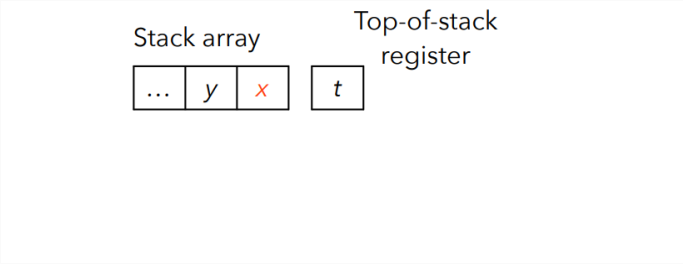
Top-of-stack Caching

In a stack-based VM, the stack is typically represented as an array in memory. Since almost all instructions access the stack, it can be beneficial to store some of its topmost elements in registers.

Top-of-stack Caching

In a stack-based VM, the stack is typically represented as an array in memory. Since almost all instructions access the stack, it can be beneficial to store some of its topmost elements in registers.

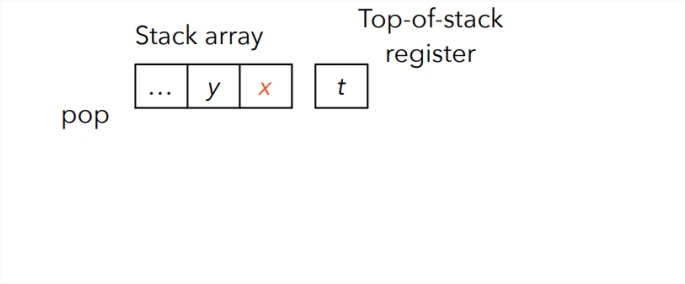
However, keeping a *fixed* number of stack elements in registers is usually a bad idea, as the following example illustrates:



Top-of-stack Caching

In a stack-based VM, the stack is typically represented as an array in memory. Since almost all instructions access the stack, it can be beneficial to store some of its topmost elements in registers.

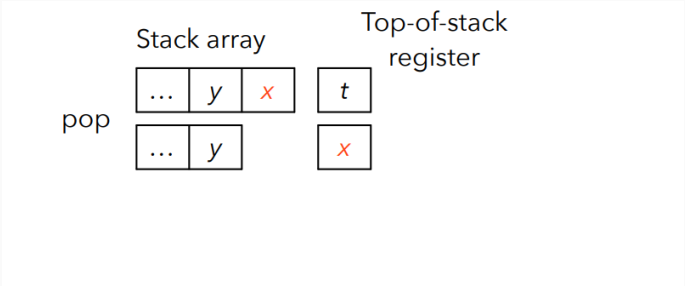
However, keeping a *fixed* number of stack elements in registers is usually a bad idea, as the following example illustrates:



Top-of-stack Caching

In a stack-based VM, the stack is typically represented as an array in memory. Since almost all instructions access the stack, it can be beneficial to store some of its topmost elements in registers.

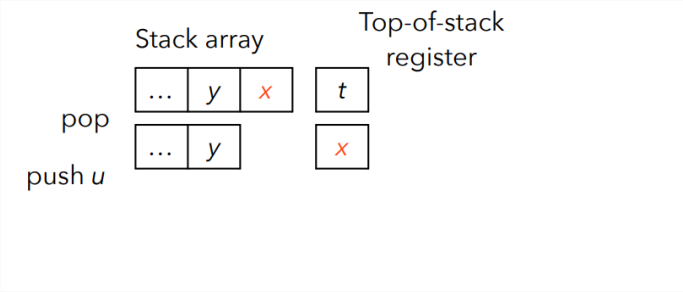
However, keeping a *fixed* number of stack elements in registers is usually a bad idea, as the following example illustrates:



Top-of-stack Caching

In a stack-based VM, the stack is typically represented as an array in memory. Since almost all instructions access the stack, it can be beneficial to store some of its topmost elements in registers.

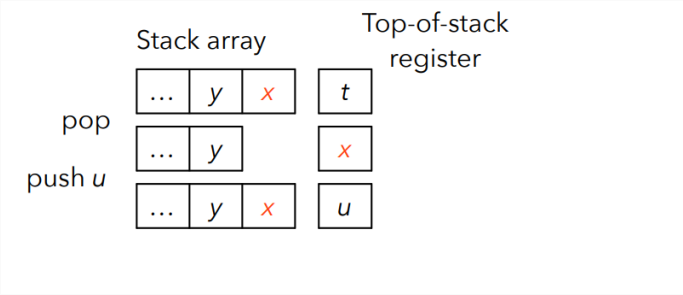
However, keeping a *fixed* number of stack elements in registers is usually a bad idea, as the following example illustrates:



Top-of-stack Caching

In a stack-based VM, the stack is typically represented as an array in memory. Since almost all instructions access the stack, it can be beneficial to store some of its topmost elements in registers.

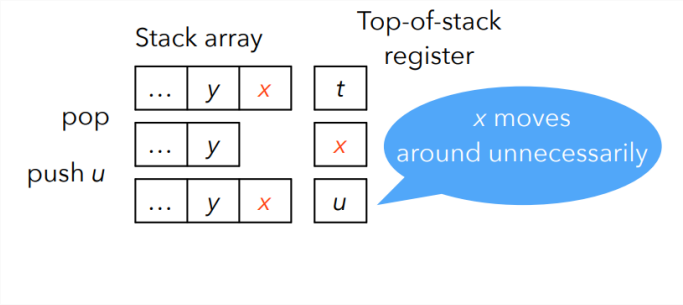
However, keeping a *fixed* number of stack elements in registers is usually a bad idea, as the following example illustrates:



Top-of-stack Caching

In a stack-based VM, the stack is typically represented as an array in memory. Since almost all instructions access the stack, it can be beneficial to store some of its topmost elements in registers.

However, keeping a *fixed* number of stack elements in registers is usually a bad idea, as the following example illustrates:



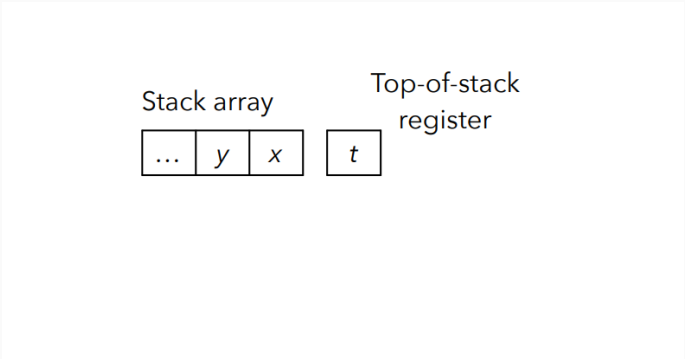
Top-of-stack Caching

Since caching a fixed number of stack elements in registers seems like a bad idea, it is natural to try to cache a *variable* number of them.

Top-of-stack Caching

Since caching a fixed number of stack elements in registers seems like a bad idea, it is natural to try to cache a *variable* number of them.

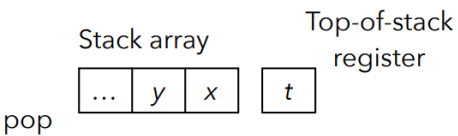
For example, here is what happens when caching *at most one* stack element in a register:



Top-of-stack Caching

Since caching a fixed number of stack elements in registers seems like a bad idea, it is natural to try to cache a *variable* number of them.

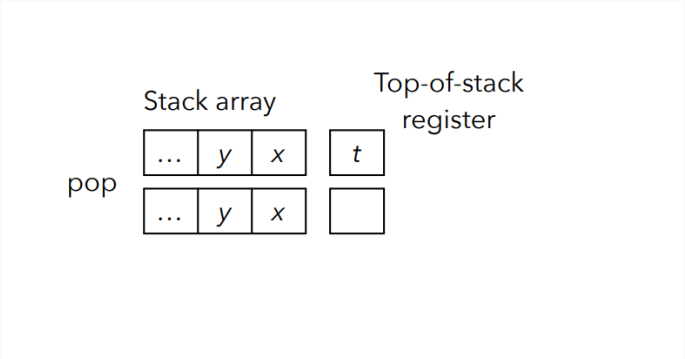
For example, here is what happens when caching *at most one* stack element in a register:



Top-of-stack Caching

Since caching a fixed number of stack elements in registers seems like a bad idea, it is natural to try to cache a *variable* number of them.

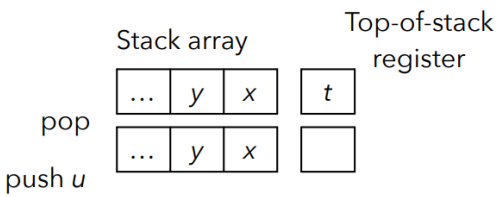
For example, here is what happens when caching *at most one* stack element in a register:



Top-of-stack Caching

Since caching a fixed number of stack elements in registers seems like a bad idea, it is natural to try to cache a *variable* number of them.

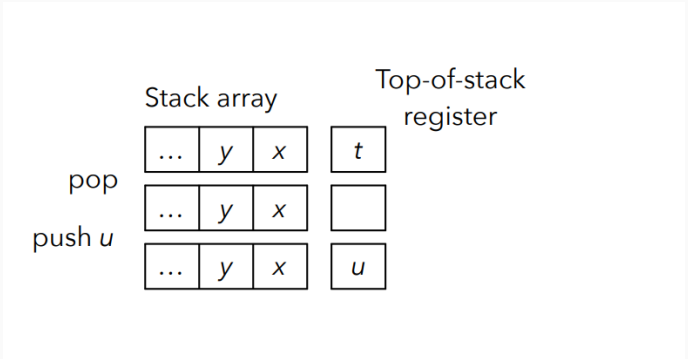
For example, here is what happens when caching *at most one* stack element in a register:



Top-of-stack Caching

Since caching a fixed number of stack elements in registers seems like a bad idea, it is natural to try to cache a *variable* number of them.

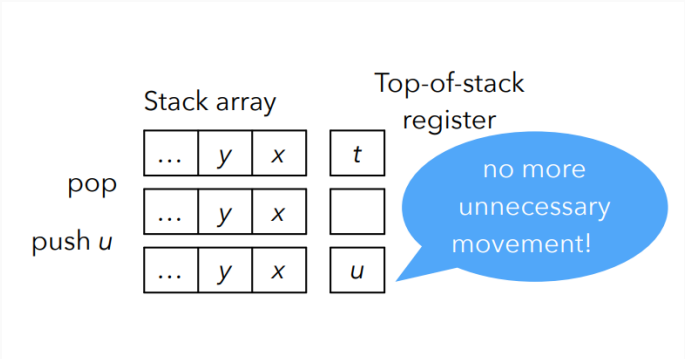
For example, here is what happens when caching *at most one* stack element in a register:



Top-of-stack Caching

Since caching a fixed number of stack elements in registers seems like a bad idea, it is natural to try to cache a *variable* number of them.

For example, here is what happens when caching *at most one* stack element in a register:



Top-of-stack Caching

Consequence: Caching a variable number of stack elements in registers complicates the implementation of instructions.

There must be one implementation of each VM instruction per **cache state** — defined as the number of stack elements currently cached in registers.

Top-of-stack Caching

Consequence: Caching a variable number of stack elements in registers complicates the implementation of instructions.

There must be one implementation of each VM instruction per **cache state** — defined as the number of stack elements currently cached in registers.

For example, when caching at most one stack element, the `add` instruction needs the following two implementations:

State 0: no elements in reg.

```
add_0:
  tos = sp[0]+sp[1];
  sp += 2;
  // go to state 1
```

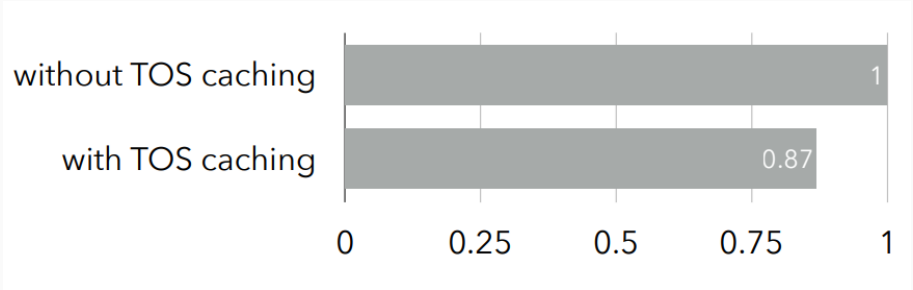
State 1: top-of-stack in reg.

```
add_1:
  tos += sp[0];
  sp += 1;
  // stay in state 1
```

Benchmark

The benchmark below compares two versions of a small interpreter measured while interpreting a program summing the first 200,000,000 integers. Both interpreters were compiled with Clang v503.0.38 with maximum optimizations, and run on an Intel Core i5.

The normalized times are presented below.



Optimizing VMs

The basic, `switch`-based implementation of the virtual machine just presented can be made faster using several techniques:

- Threaded code
- Top-of-stack caching
- *Super-instructions*
- JIT compilation

Static Super-instructions

Since instruction dispatch is expensive in a VM, one way to reduce its cost is simply to dispatch less.

This can be done by grouping several instructions that often appear in sequence into a **super-instruction**.

Static Super-instructions

Since instruction dispatch is expensive in a VM, one way to reduce its cost is simply to dispatch less.

This can be done by grouping several instructions that often appear in sequence into a **super-instruction**.

For example, if the `mul` instruction is often followed by the `add` instruction, the two can be combined in a single `madd` (multiply and add) super-instruction.

Profiling is typically used to determine which sequences should be transformed into super-instructions, and the instruction set of the VM is then modified accordingly.

Static and Dynamic Super-instructions

Super-instructions can be pushed to its limits by generating one super-instruction for every basic block of the program!

This effectively transform all basic blocks into single (super-)instructions.

It is also possible to generate super-instructions at run time, to adapt them to the program being run. This is the idea behind **dynamic super-instructions**.

Just-in-time Compilation

Idea: compiling code into native machine code at runtime, rather than ahead-of-time (AOT) or interpreting it.

Just-in-time Compilation

Idea: compiling code into native machine code at runtime, rather than ahead-of-time (AOT) or interpreting it.

Why JIT vs AOT? JIT compilation allows to optimize the generated code based on the actual program and dynamic data being run, and thus can sometimes achieve better performance than AOT compilation.

- If an interface only has one implementation, then we can optimize calls as static calls instead of virtual calls (with dispatching).
- We can make speculative assumptions based on observations, such as the objects in array are all of the same type, and optimize accordingly. If the assumption turns out to be wrong, we can deoptimize and fall back to a more generic code.
- ...

Just-in-time Compilation

The HotSpot JVM as an example of a JIT compiler:

- The Java source code is compiled into Java bytecode first, which is a platform-independent instruction set.
- JVM load the bytecode and interpret it initially, but it also *profiles* the running program to identify hot code paths.
- When a hot is identified, the JVM compiles it into native machine code and replaces the interpreted version with the compiled one for subsequent executions.
  - C1: the client compiler, fast compilation with moderate optimizations; could be profiled too.
  - C2: the server compiler, slower compilation with more aggressive optimizations.

MiniScalaVM

MiniScalaVM is the virtual machine developed for this course. Its main characteristics are:

- it is 32-bit, in the sense that the basic unit of storage is a 32-bit word – therefore, (untagged) integers and pointers are both 32-bit
- it is register-based, although its notion of registers is a bit non-standard
- it is relatively simple, with only 32 instructions

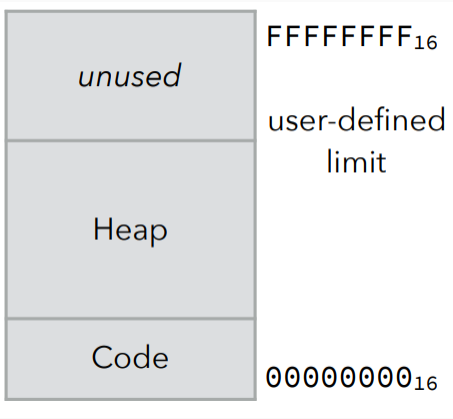
Very compact implementation (~700 lines of code), as you will see with project 7.

Memory

MiniScalaVM has a 32-bit address space — even when running on a 64-bit machine — which is used to store both code and data.

Code is stored starting at address 0 and the rest of the available memory is used for the heap.

The MiniScalaVM address space is virtual, in that it is not the same as the one of the host architecture.



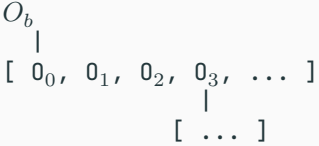
Registers

Strictly speaking, MiniScalaVM has only four registers:

- **PC** is the program counter, containing the address of the instruction being executed
- $\mathbf{I}_b$, $\mathbf{L}_b$, and $\mathbf{O}_b$ are the input, local, and output **base registers** (respectively), each of which contains either 0 or the address of a heap-allocated block

Register Slots

The slots of the blocks pointed by $\mathbf{I}_b$, $\mathbf{L}_b$, and $\mathbf{O}_b$ are reachable through pseudo-registers. For example, the pseudo-register $\mathbf{O}_3$ designates the slot at index 3 of the block pointed by $\mathbf{O}_b$.



Note: In the following, we use the term **register** to designate a pseudo-register, and **base register** to designate a base register.

Engine

```
value_t engine_run() {
  instr_t* pc = memory_start;
  // initialize memory/registers ...
  void** labels[OPCODE_COUNT];
  labels[opcode_ADD] = &&l_ADD;
  labels[opcode_SUB] = &&l_SUB;
  // ... other instructions
  goto *labels[instr_opcode(*pc)];
l_ADD: {
    Ra = Rb + Rc;
    pc += 1;
  } goto *labels[instr_opcode(*pc)];
l_SUB: {
    Ra = Rb - Rc;
    pc += 1;
  } goto *labels[instr_opcode(*pc)];
  // ... other instructions
}
```

Example

The factorial in (hand-coded) MiniScalaVM assembly:

```
fact:   RALO Lb,1
        RALO Ob,5
        LDLO L0,0
        JNE  L0,I4,else
        LDLO I4,1
        RET
else:   LDLO L0,1
        SUB  O4,I4,L0
        LDLO L0,fact
        CALL L0
        MUL  I4,I4,O0
        RET
```

Function Call and Return

- A function gets its arguments through its **input registers**, stores its local variables in its **local registers**, and uses its **output registers** to pass arguments to the functions it calls.

Function Call and Return

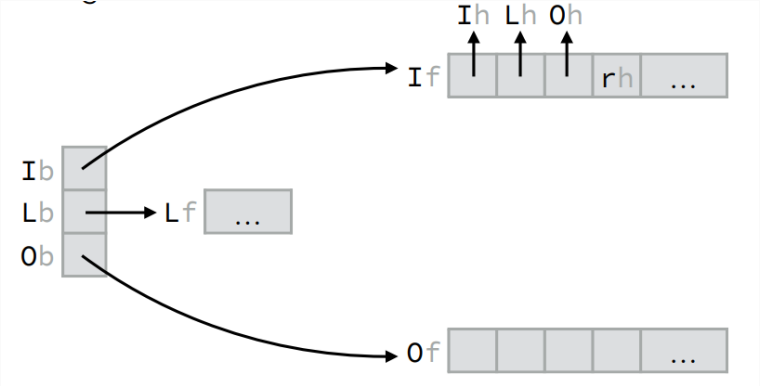
- A function gets its arguments through its **input registers**, stores its local variables in its **local registers**, and uses its **output registers** to pass arguments to the functions it calls.
- The `CALL` instruction takes care of saving the caller's context, composed of its base registers ($\mathbf{I}_b$, $\mathbf{L}_b$, and $\mathbf{O}_b$) as well as its return address.
- They are saved in the callee's first four input registers ($\mathbf{I}_0$ to $\mathbf{I}_3$), and can be seen as implicit arguments passed to the callee.

Function Call and Return

- A function gets its arguments through its **input registers**, stores its local variables in its **local registers**, and uses its **output registers** to pass arguments to the functions it calls.
- The `CALL` instruction takes care of saving the caller's context, composed of its base registers ($\mathbf{I}_b$, $\mathbf{L}_b$, and $\mathbf{O}_b$) as well as its return address.
- They are saved in the callee's first four input registers ($\mathbf{I}_0$ to $\mathbf{I}_3$), and can be seen as implicit arguments passed to the callee.
- Symmetrically, the `RET` instruction takes care of restoring the caller's context.

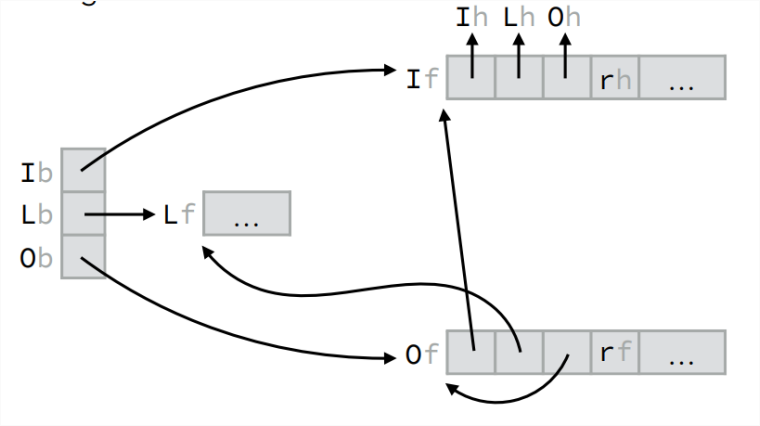
Non-tail Call Example

Example: saving of the caller's context as well as the installation of the callee's context during a non-tail call from a function **f** to a function **g**, with **h** being **f**'s caller:



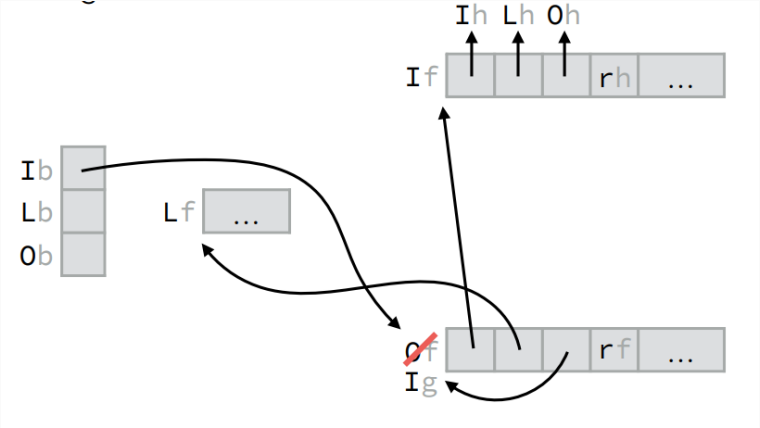
Non-tail Call Example

Example: saving of the caller's context as well as the installation of the callee's context during a non-tail call from a function **f** to a function **g**, with **h** being **f**'s caller:



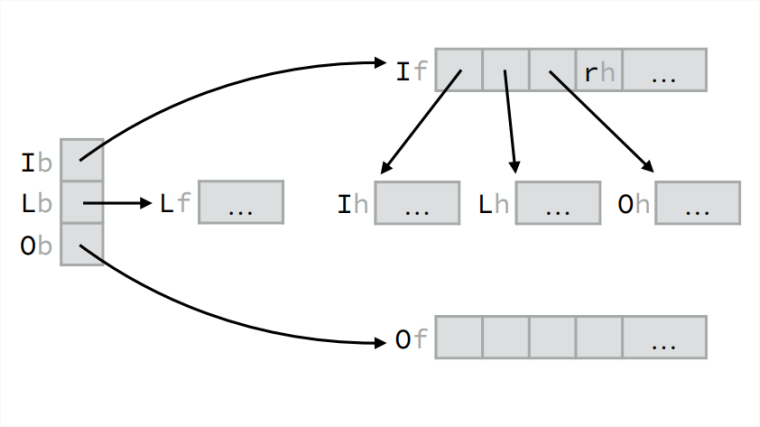
Non-tail Call Example

Example: saving of the caller's context as well as the installation of the callee's context during a non-tail call from a function **f** to a function **g**, with **h** being **f**'s caller:



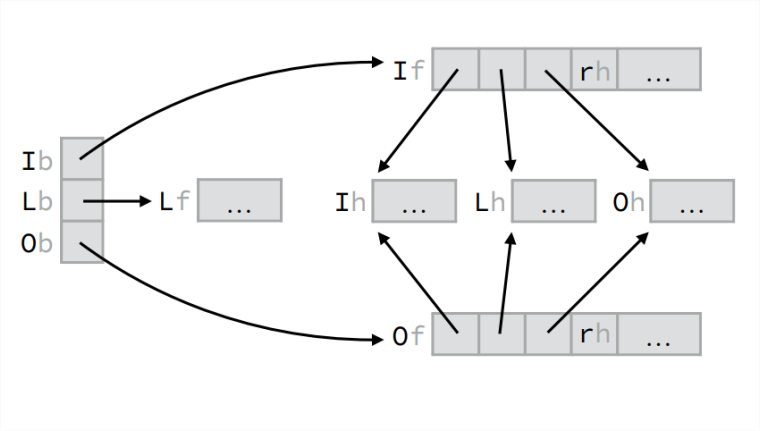
Tail Call Example

Example: saving of the caller's context as well as the installation of the callee's context during a **tail call** from a function **f** to a function **g**, with **h** being **f**'s caller:



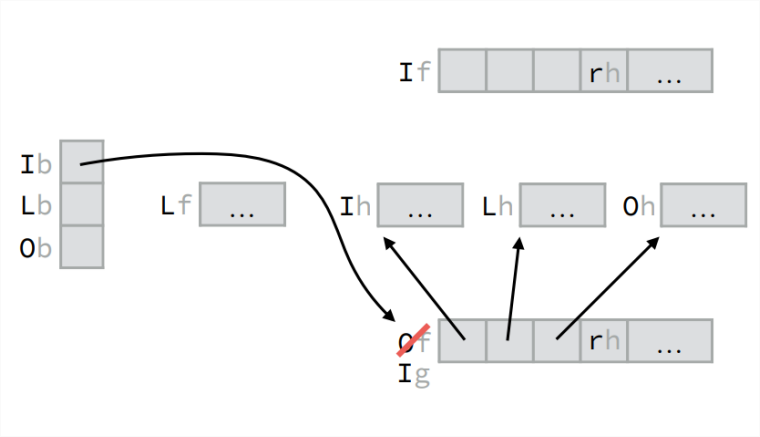
Tail Call Example

Example: saving of the caller's context as well as the installation of the callee's context during a **tail call** from a function **f** to a function **g**, with **h** being **f**'s caller:



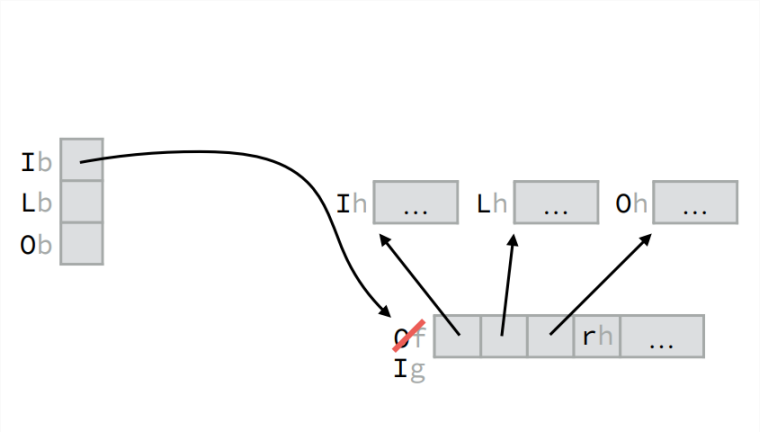
Tail Call Example

Example: saving of the caller's context as well as the installation of the callee's context during a **tail call** from a function **f** to a function **g**, with **h** being **f**'s caller:



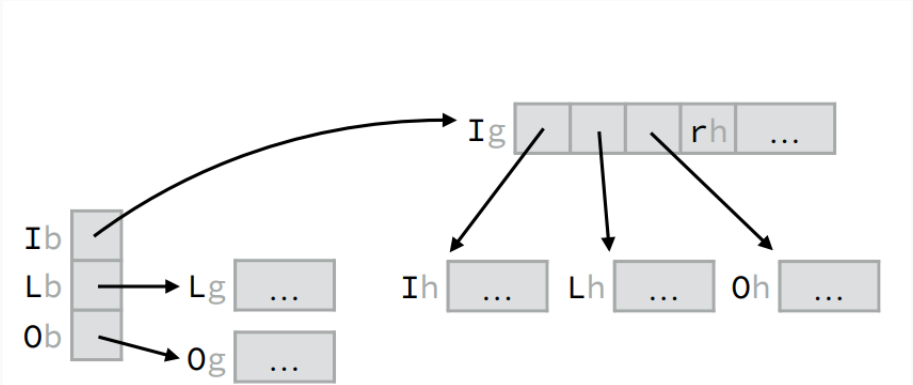
Tail Call Example

Example: saving of the caller's context as well as the installation of the callee's context during a **tail call** from a function **f** to a function **g**, with **h** being **f**'s caller:



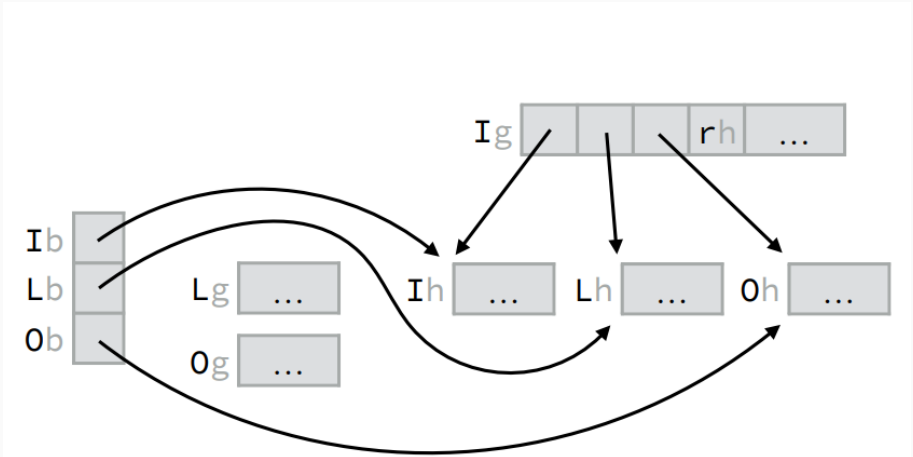
Return Example

Example: restoration of the caller's context during a function return from **g** to **h** (**g** was tail called from **f**):



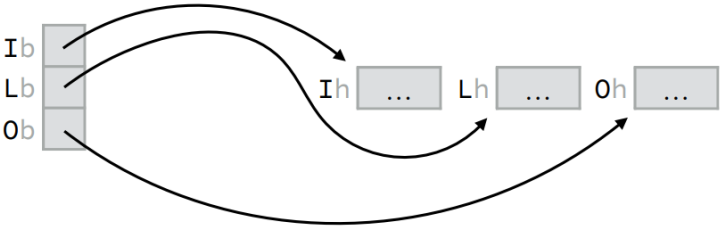
Return Example

Example: restoration of the caller's context during a function return from **g** to **h** (**g** was tail called from **f**):



Return Example

Example: restoration of the caller's context during a function return from **g** to **h** (**g** was tail called from **f**):



Arithmetic Instructions (1)

`ADD` *Ra Rb Rc*	*Ra* ← *Rb* + *Rc*
`SUB` *Ra Rb Rc*	*Ra* ← *Rb* - *Rc*
`MUL` *Ra Rb Rc*	*Ra* ← *Rb* * *Rc*
`DIV` *Ra Rb Rc*	*Ra* ← *Rb* / *Rc*
`MOD` *Ra Rb Rc*	*Ra* ← *Rb* % *Rc*

Ra, *Rb*, *Rc*: registers
`PC` implicitly augmented by 4 by each instruction

Arithmetic Instructions (2)

ASL *Ra Rb Rc*	*Ra* ← *Rb* << *Rc*
ASR *Ra Rb Rc*	*Ra* ← *Rb* >> *Rc*
AND *Ra Rb Rc*	*Ra* ← *Rb* & *Rc*
OR *Ra Rb Rc*	*Ra* ← *Rb* | *Rc*
XOR *Ra Rb Rc*	*Ra* ← *Rb* ^ *Rc*

Ra, *Rb*, *Rc*: registers
PC implicitly augmented by 4 by each instruction

Control Instructions (1)

`JLT` *Ra Rb* D^{10}	if *Ra* < *Rb* then PC ← PC + 4·D^{10}
`JLE` *Ra Rb* D^{10}	if *Ra* ≤ *Rb* then PC ← PC + 4·D^{10}
`JEQ` *Ra Rb* D^{10}	if *Ra* = *Rb* then PC ← PC + 4·D^{10}
`JNE` *Ra Rb* D^{10}	if *Ra* ≠ *Rb* then PC ← PC + 4·D^{10}
`JGE` *Ra Rb* D^{10}	if *Ra* ≥ *Rb* then PC ← PC + 4·D^{10}
`JGT` *Ra Rb* D^{10}	if *Ra* > *Rb* then PC ← PC + 4·D^{10}
`JI` D^{26}	PC ← PC + 4·D^{26}

Ra, *Rb*, *Rc*: registers, D^k: *k*-bit signed displacement

Control Instructions (2)

CALL *Ra*	$O_0 \leftarrow I_b$, $O_1 \leftarrow L_b$, $O_2 \leftarrow O_b$, $O_3 \leftarrow PC + 4$, $I_b \leftarrow O_b$, $PC \leftarrow Ra$
TCAL *Ra*	$O_0 \leftarrow I_0$, $O_1 \leftarrow I_1$, $O_2 \leftarrow I_2$, $O_3 \leftarrow I_3$, $I_b \leftarrow O_b$, $PC \leftarrow Ra$
RET	$r \leftarrow I_4$ $PC \leftarrow I_3$, $O_b \leftarrow I_2$, $L_b \leftarrow I_1$, $I_b \leftarrow I_0$ $O_0 \leftarrow r$
HALT	stop execution

Ra: register
r: temporary value

Register Instructions

`LDLO` Ra, S^{18}	$Ra \leftarrow S^{18}$
`LDHI` Ra, U^{16}	$Ra \leftarrow (U^{16} << 16) \mid (Ra \,\&\, \mathrm{FFFF}_{16})$
`MOVE` Ra, Rb	$Ra \leftarrow Rb$
`RALO` B, U^{8}	$B \leftarrow$ new block of size U^{8} and tag 201

Ra, Rb: registers, B: base register ($\mathtt{I}_b$, $\mathtt{L}_b$ or $\mathtt{O}_b$),
S^k: k-bit signed constant, U^k: k-bit unsigned constant
`PC` implicitly augmented by 4 by each instruction

Register Instructions

`BALO` *Ra Rb* T^8	*Ra* ← new block of size *Rb* and tag T^8
`BSIZ` *Ra Rb*	*Ra* ← size of block *Rb*
`BTAG` *Ra Rb*	*Ra* ← tag of block *Rb*
`BGET` *Ra Rb Rc*	*Ra* ← element at index *Rc* of block *Rb*
`BSET` *Ra Rb Rc*	element at index *Rc* of block *Rb* ← *Ra*

Ra, *Rb*, *Rc*: registers, T^8: 8-bit block tag
`PC` implicitly augmented by 4 by each instruction

Register Instructions

`BREA` *Ra*	*Ra* ← byte read from console
`BWRI` *Ra*	write byte *Ra* to console

Ra: register
`PC` implicitly augmented by 4 by each instruction

Summary

Today:

- Various techniques to optimize VMs
- Introduce the MiniScala VM

Next week:

- Garbage collection: how to automatically manage memory resources in a VM?