Interpreters and virtual machines

Michel Schinz

Advanced compiler construction, 2008-02-29

Interpreters

Interpreters

An **interpreter** is a program that executes another program, represented as some kind of data-structure.

Common program representations include:

- raw text (source code),
- trees (AST of the program),
- linear sequences of instructions.

Why interpreters?

Interpreters enable the execution of a program without requiring its compilation to native code.

They simplify the implementation of programming languages and – on modern hardware – are efficient enough for many tasks.

Text-based interpreters

Text-based interpreters directly interpret the textual source of the program.

They are very seldom used, except for trivial languages where every expression is evaluated at most once – i.e. languages without loops or functions.

Plausible example: a calculator program, which evaluates arithmetic expressions while parsing them.

Tree-based interpreters

Tree-based interpreters walk over the abstract syntax tree of the program to interpret it.

Their advantage compared to string-based interpreters is that parsing – and name/type analysis, if applicable – is done only once.

Plausible example: a graphing program, which has to repeatedly evaluate a function supplied by the user to plot it. The minischeme interpreter is also tree-based.

Virtual Machines

Virtual machines

Virtual machines behave in a similar fashion as real machines (*i.e.* CPUs), but are implemented in software. They accept as input a program composed of a sequence of instructions.

Virtual machines often provide more than the simple interpretation of programs: they also abstract the underlying system by managing memory, threads, and sometimes I/O.

Perhaps surprisingly, virtual machines are a very old concept, dating back to ~1950.

They have been – and still are – used in the implementation of many important languages, like SmallTalk, Lisp, Forth, Pascal, and more recently Java and C#.

Why *virtual* machines?

Since the compiler has to generate code for some machine, why prefer a virtual over a real one?

- for portability: compiled VM code can be run on many actual machines,
- for simplicity: a VM is usually more high-level than a real machine, which simplifies the task of the compiler,
- for simplicity (2): a VM is easier to monitor and profile, which eases debugging.

Virtual machines drawbacks

The only drawback of virtual machines compared to real machines is that the former are slower than the latter.

This is due to the overhead associated with interpretation: fetching and decoding instructions, executing them, etc.

Moreover, the high number of indirect jumps in interpreters causes pipeline stalls in modern processors.

Kinds of virtual machines

There are two kinds of virtual machines:

1. **stack-based VMs**, which use a stack to store intermediate results, variables, etc.
2. **register-based VMs**, which use a limited set of registers for that purpose, like a real CPU.

There is some controversy as to which kind is better, but most VMs today are stack-based.

For a compiler writer, it is usually easier to target a stack-based VM than a register-based VM, as the complex task of register allocation can be avoided.

Virtual machines input

Virtual machines take as input a program expressed as a sequence of instructions.

Each instruction is identified by its **opcode** (**op**eration **code**), a simple number. Often, opcodes occupy one byte, hence the name **byte code**.

Some instructions have additional arguments, which appear after the opcode in the instruction stream.

VM implementation

Virtual machines are implemented in much the same way as a real processor:

- the next instruction to execute is fetched from memory and decoded,
- the operands are fetched, the result computed, and the state updated,
- the process is repeated.

VM implementation

Virtual machines are implemented in much the same way as a real processor:

- the next instruction to execute is fetched from memory and decoded,
- the operands are fetched, the result computed, and the state updated,
- the process is repeated.

VM implementation

Many VMs today are written in C or C++, because these languages are at the right abstraction level for the task, fast and relatively portable.

As we will see later, the Gnu C compiler (`gcc`) has an extension that makes it possible to use labels as normal values. This extension can be used to write very efficient VMs, and for that reason, several of them are written for `gcc`.

Implementing a VM in 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 */
    }
  }
}
```

Optimising VMs

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

- threaded code,
- top of stack caching,
- super-instructions,
- JIT compilation.

Threaded code

Threaded code

In a `switch`-based interpreter, each instruction requires two jumps:

1. one indirect jump to the branch handling the current instruction,
2. one direct jump from there to the main loop.

It would be better to avoid the second one, by jumping directly to the code handling the next instruction. This is called **threaded code**.

Threaded code *vs.* switch

Program: `add sub mul`

`switch`-based

main loop

add

sub

mul

Threaded

main

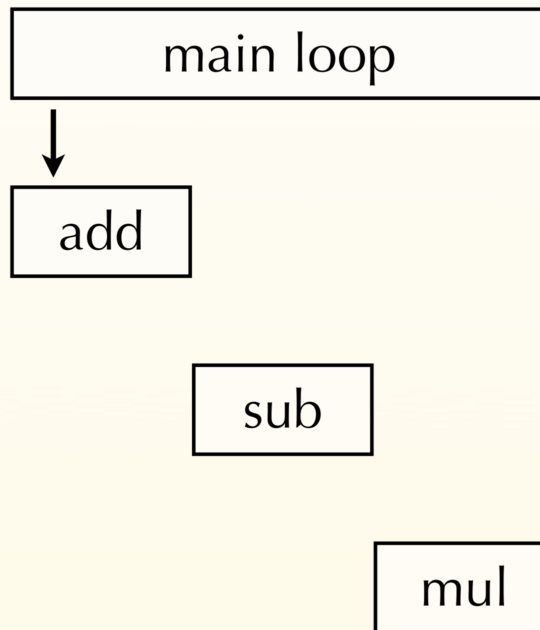
add

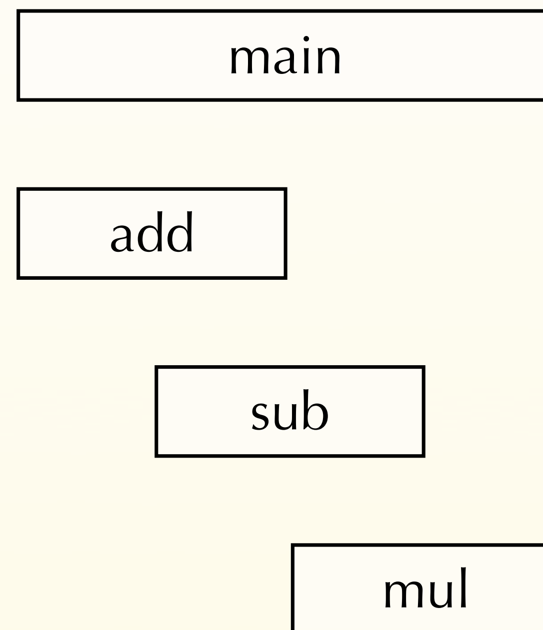
sub

mul

Threaded code *vs.* switch

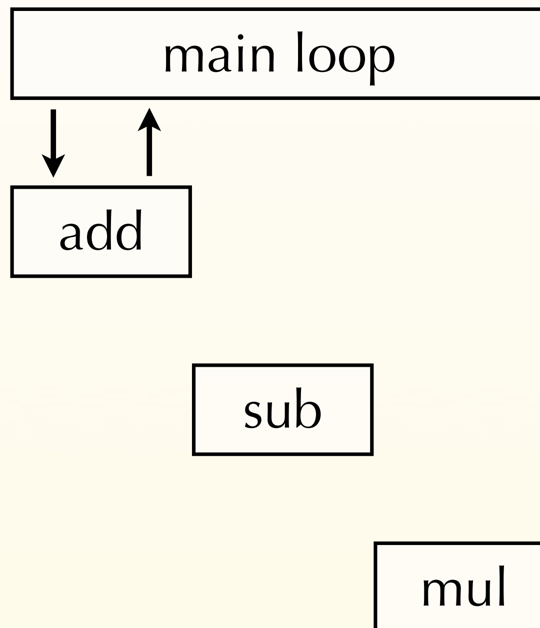
Program: `add sub mul`

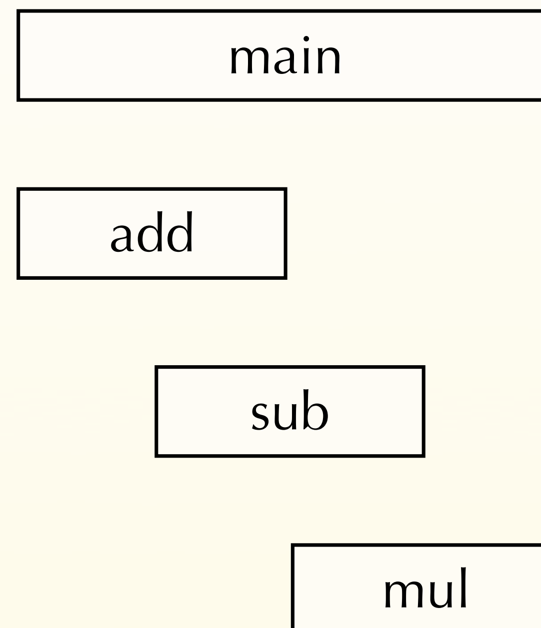




Threaded code *vs.* switch

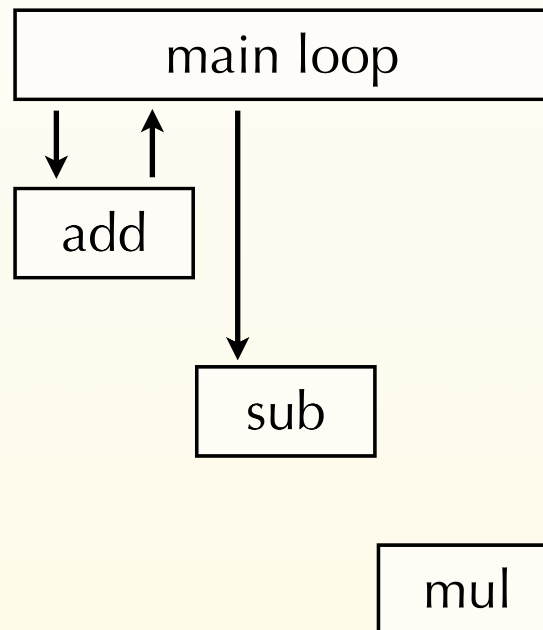
Program: `add sub mul`

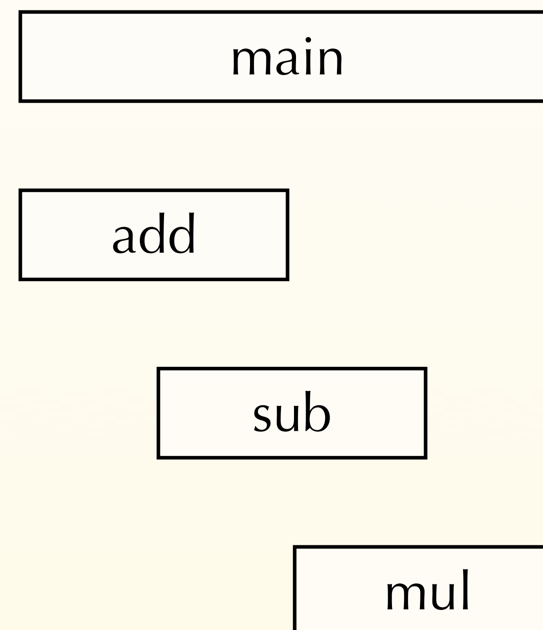




Threaded code *vs.* switch

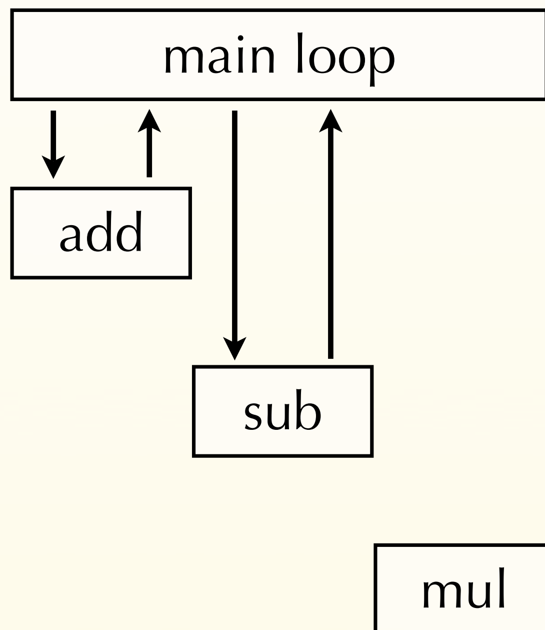
Program: `add sub mul`

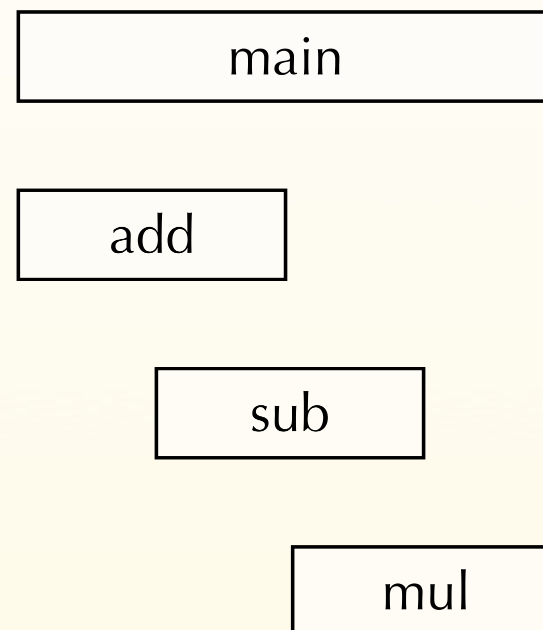




Threaded code *vs.* switch

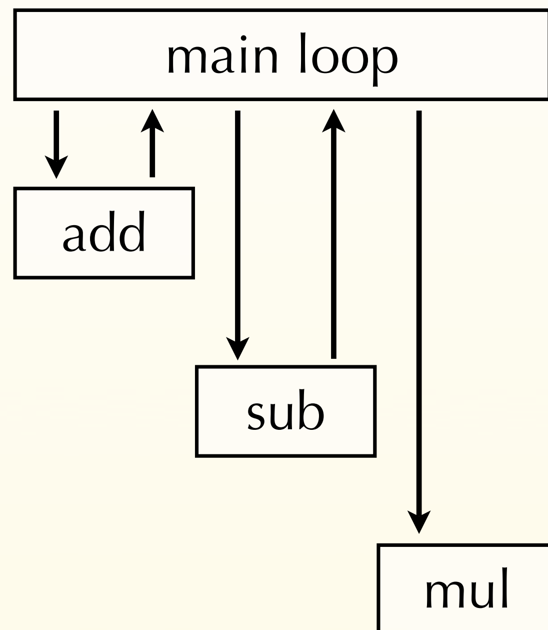
Program: `add sub mul`

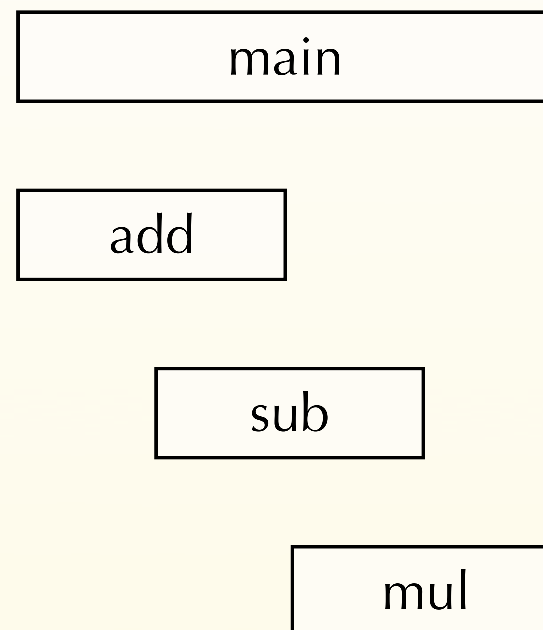




Threaded code *vs.* switch

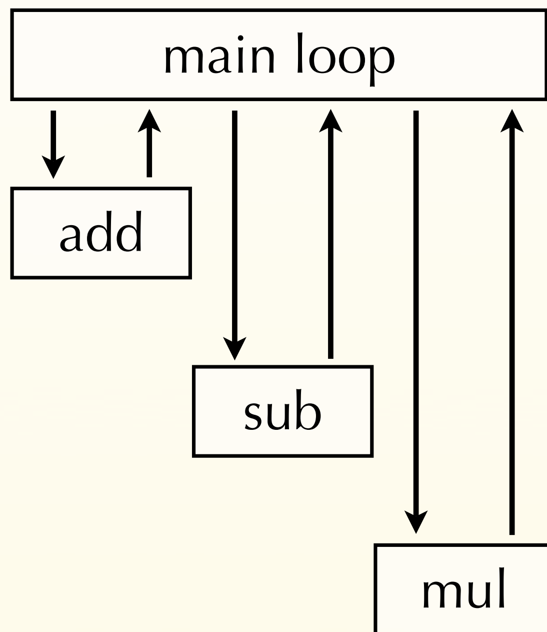
Program: `add sub mul`

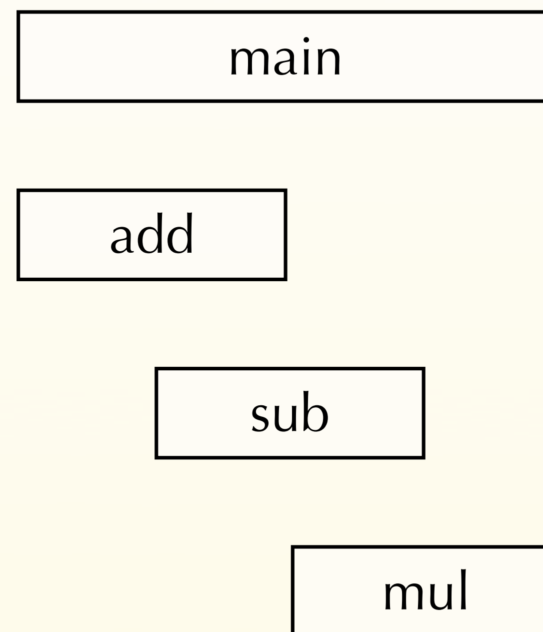




Threaded code *vs.* switch

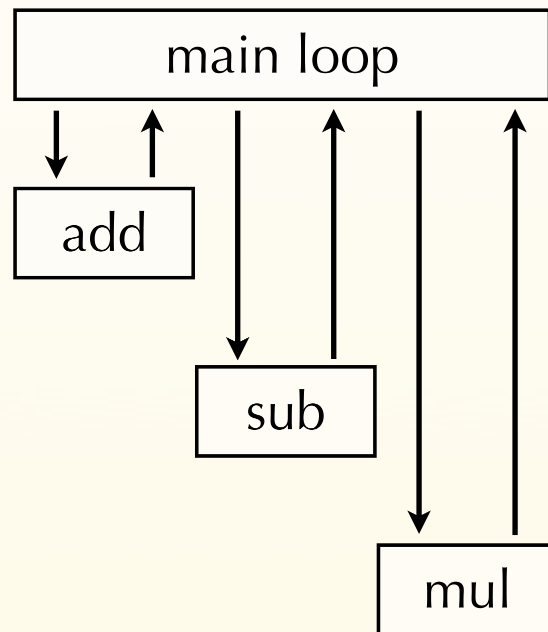
Program: `add sub mul`

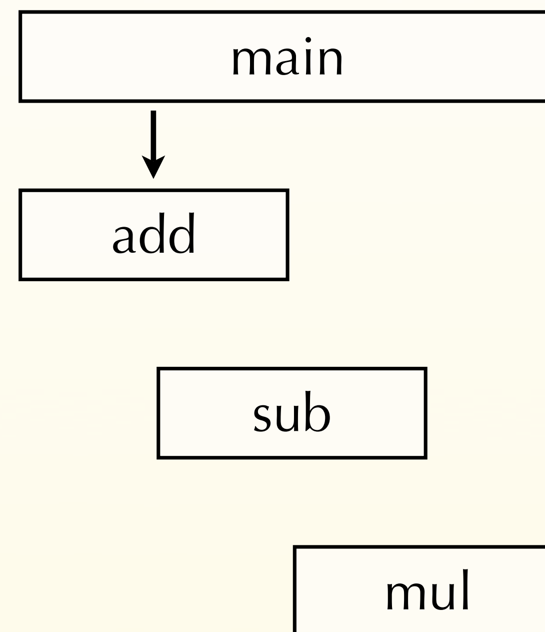




Threaded code *vs.* switch

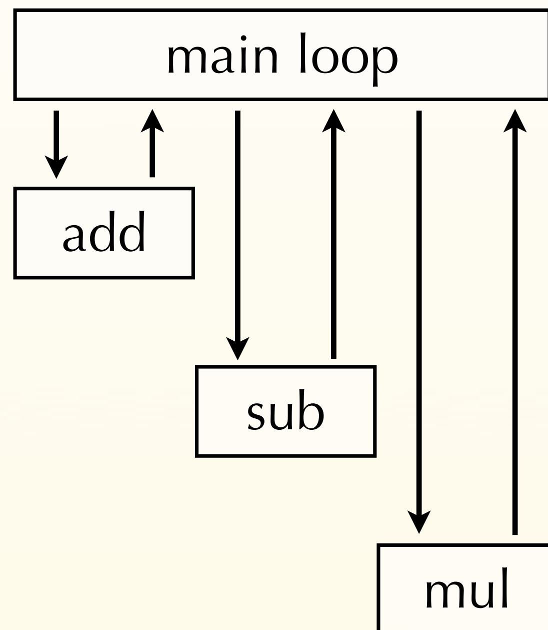
Program: `add sub mul`

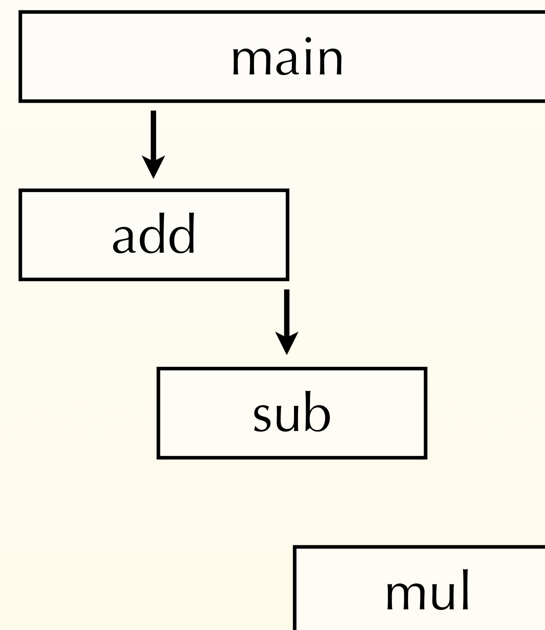




Threaded code *vs.* switch

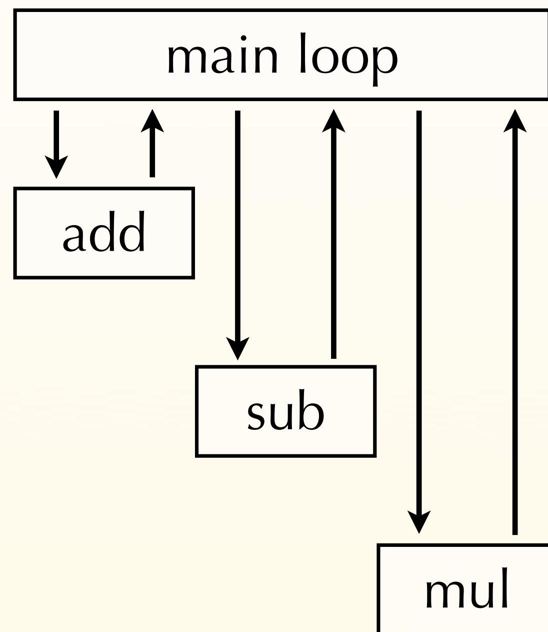
Program: `add sub mul`

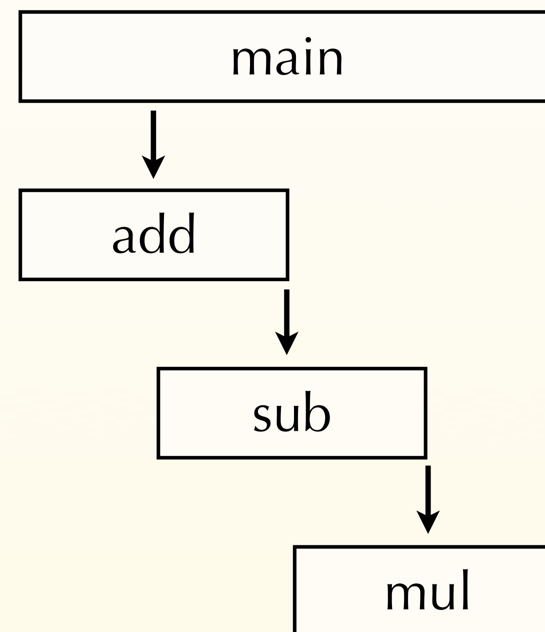




Threaded code *vs.* switch

Program: `add sub mul`





Implementing threaded code

To implement threaded code, there are two main techniques:

- with **indirect threading**, instructions index an array containing pointers to the code handling them,
- with **direct threading**, instructions are pointers to the code handling them.

Direct threading is the most efficient of the two, and the most often used in practice. For these reasons, we will not look at indirect threading.

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.

But `gcc` allows the manipulation of labels as values, which is much more efficient!

Direct threading in ANSI C

Implementing direct threading in ANSI C is easy, but unfortunately very inefficient!

The idea is to define one function per VM instruction. The program can then simply be represented as an array of function pointers. Some 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 = ...;

static void add() {
  sp[1] += sp[0];
  ++sp;
  (*++pc)(); /* handle next instruction */
}

/* ... other instructions */

static instruction_t program[] = { add, /* ... */ };

void interpret() {
  sp = ...;
  pc = program;
  (*pc)(); /* handle first instruction */
}
```

Direct threading in ANSI C

This implementation of direct threading in ANSI C has a major problem: it leads to stack overflow very quickly, unless the compiler implements an optimisation called **tail call elimination** (**TCE**).

Briefly, the idea of tail call elimination is to replace a function call that appears as the last statement of a function by a simple jump to the called function.

In our interpreter, the function call appearing at the end of `add` – and all other functions implementing VM instructions – can be optimised that way.

Unfortunately, few C compilers implement tail call elimination in all cases. However, `gcc` 4.01 is able to avoid stack overflows for the interpreter just presented.

Trampolines

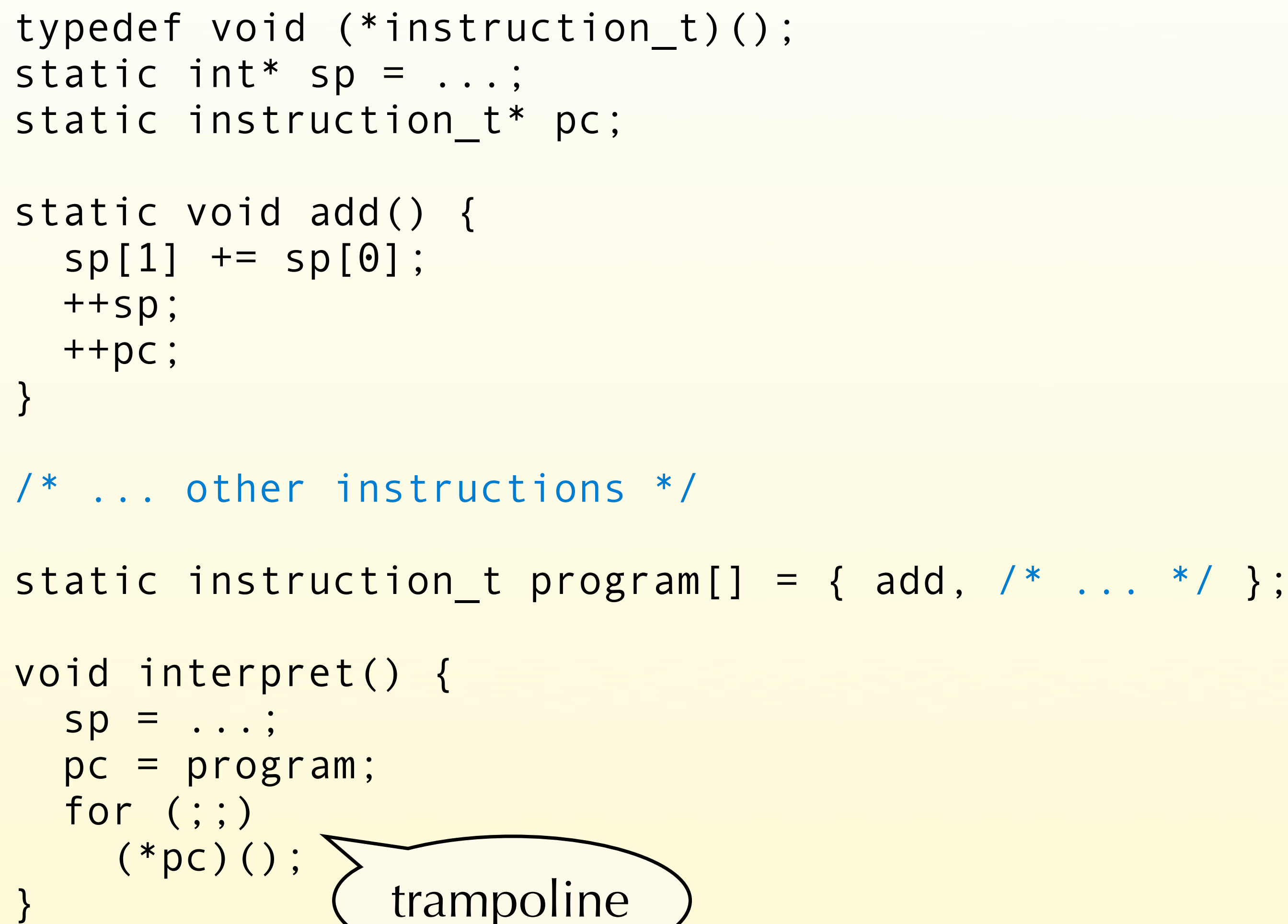
It is possible to avoid stack overflows in a direct threaded interpreter written in ANSI C, even if the compiler does not perform tail call elimination.

The idea is that functions implementing VM instructions simply return to the main function, which takes care of calling the function handling the next VM instruction.

While this technique – known as a **trampoline** – avoids stack overflows, it leads to interpreters that are extremely slow. Its interest is mostly academic.

Direct threading in ANSI C

```
typedef void (*instruction_t)();
static int* sp = ...;
static instruction_t* pc;

static void add() {
  sp[1] += sp[0];
  ++sp;
  ++pc;
}

/* ... other instructions */

static instruction_t program[] = { add, /* ... */ };

void interpret() {
  sp = ...;
  pc = program;
  for (;;)
    (*pc)();
}
```



Direct threading with `gcc`

The Gnu C compiler (`gcc`) offers an extension that is very useful to implement direct threading: labels can be treated as values, and a `goto` 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.

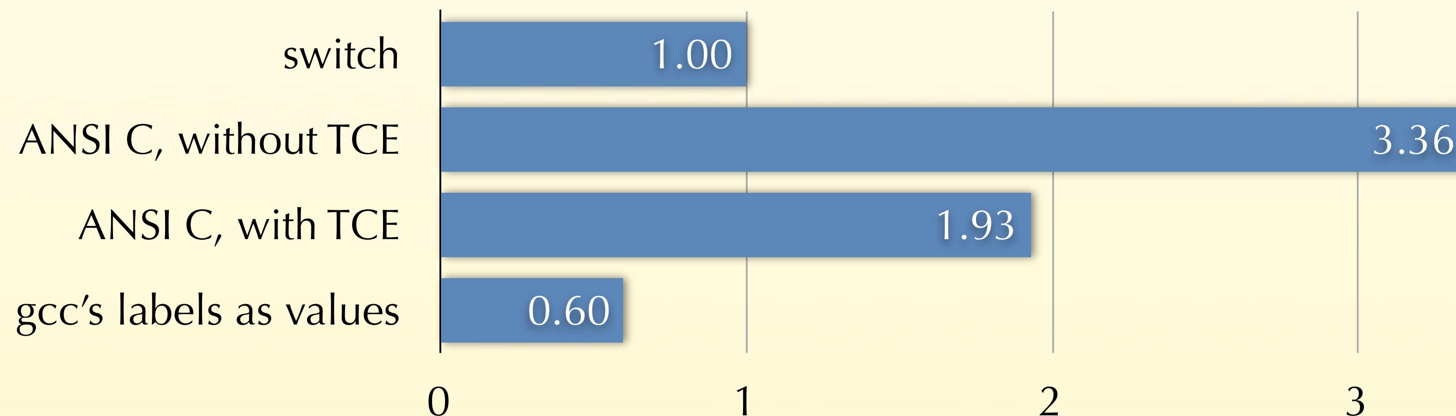
Direct threading with gcc


```
void interpret() {
  void* program[] = { &&l_add, /* ... */ };

  int* sp = ...;
  void** pc = program;
  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 `gcc` v4.01 with full optimisations, and run on an Intel Core 2 Duo.

The normalised times are presented below, and show that only direct threading using `gcc`'s labels-as-values performs better than a switch-based interpreter.

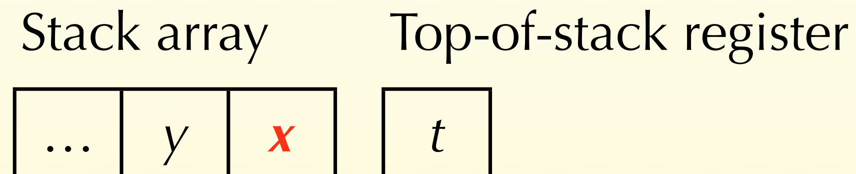


Top-of-stack caching

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 interesting to store some of its topmost elements in registers.

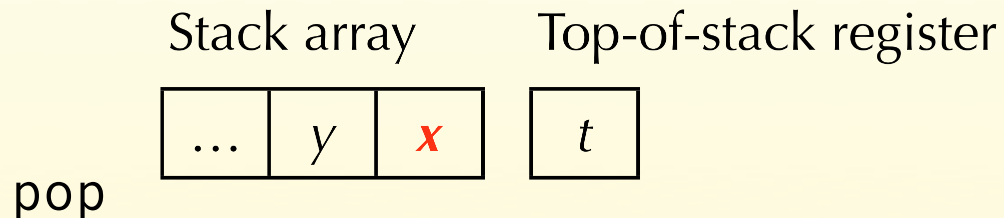
However, keeping a *fixed* number of stack elements in registers is usually a bad idea, as the following example illustrates:



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 interesting to store some of its topmost elements in registers.

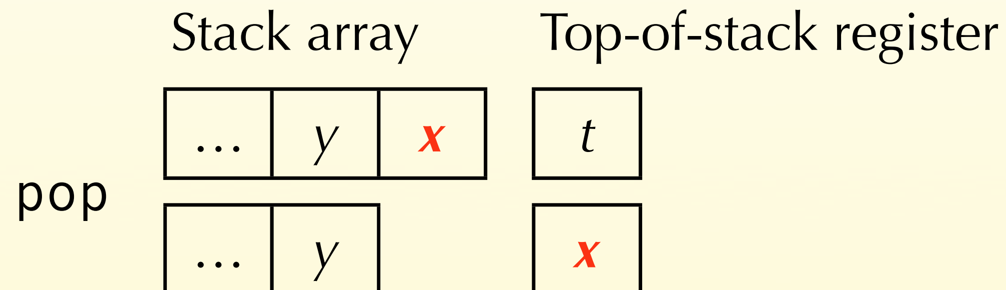
However, keeping a *fixed* number of stack elements in registers is usually a bad idea, as the following example illustrates:



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 interesting to store some of its topmost elements in registers.

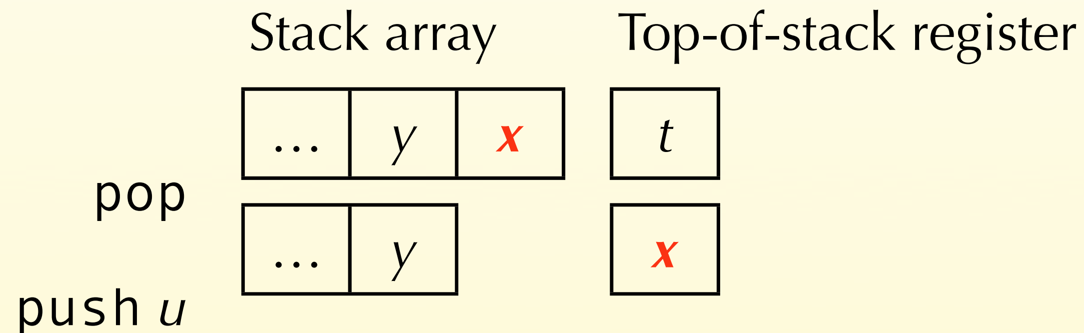
However, keeping a *fixed* number of stack elements in registers is usually a bad idea, as the following example illustrates:



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 interesting to store some of its topmost elements in registers.

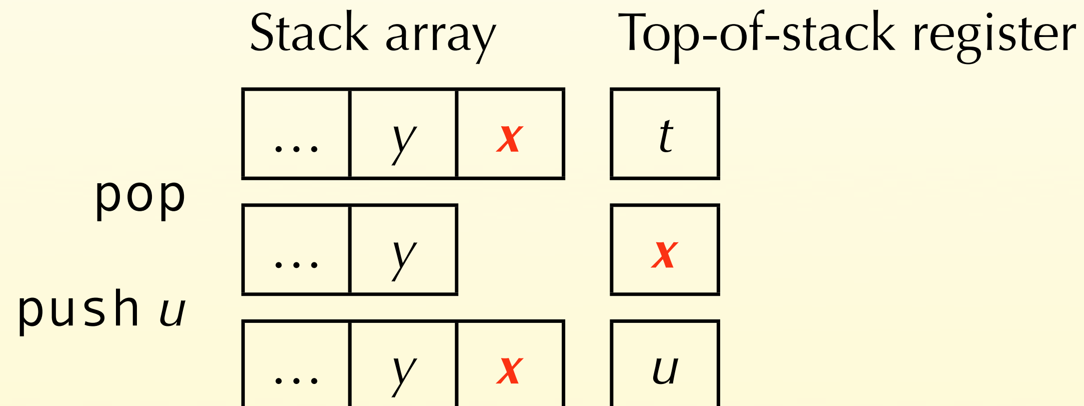
However, keeping a *fixed* number of stack elements in registers is usually a bad idea, as the following example illustrates:



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 interesting to store some of its topmost elements in registers.

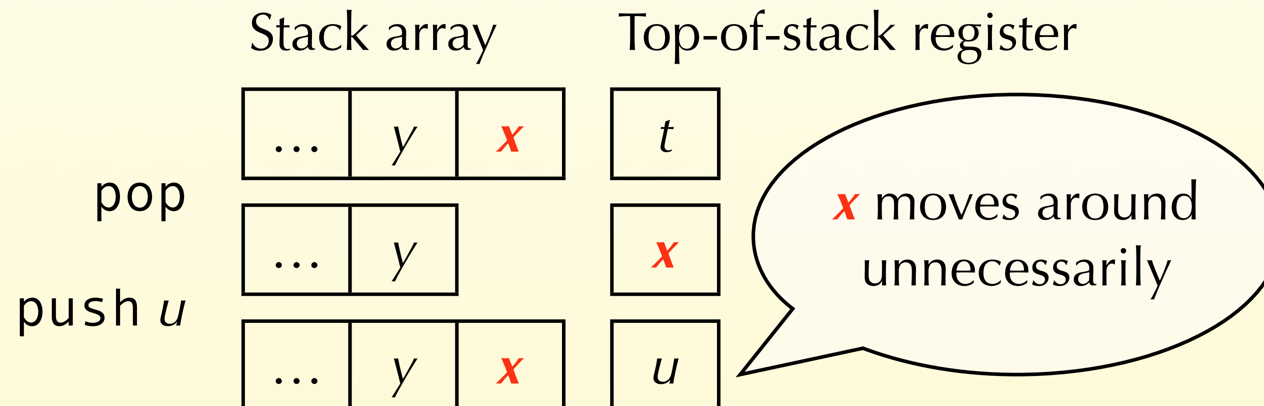
However, keeping a *fixed* number of stack elements in registers is usually a bad idea, as the following example illustrates:



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 interesting to store some of its topmost elements in registers.

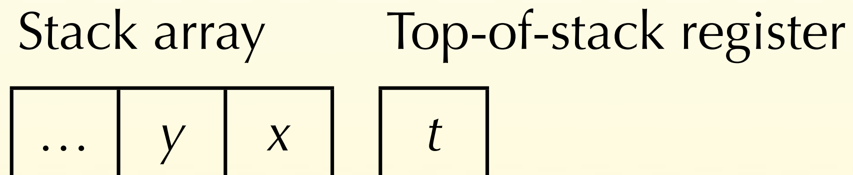
However, keeping a *fixed* number of stack elements in registers is usually a bad idea, as the following example illustrates:



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.

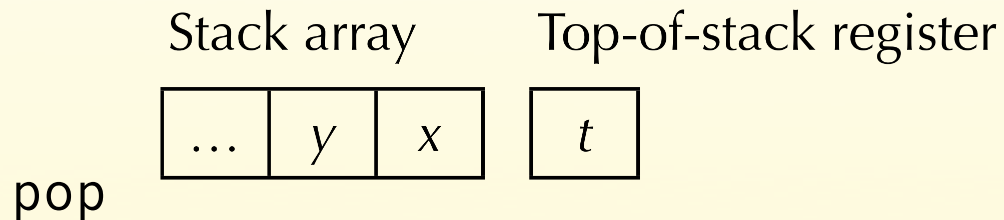
For example, here is what happens when caching *at most* one stack element in a register:



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.

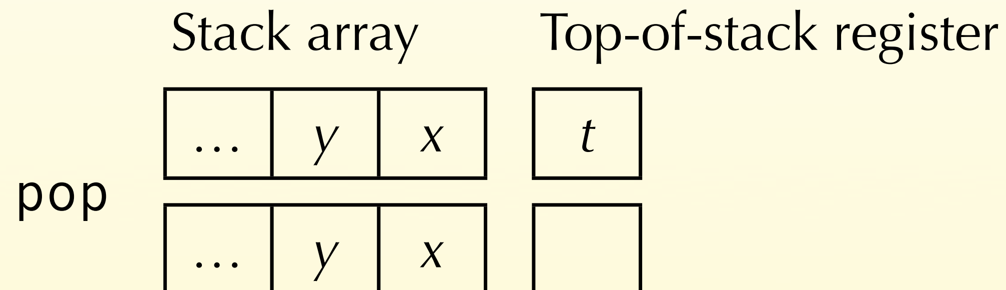
For example, here is what happens when caching *at most* one stack element in a register:



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.

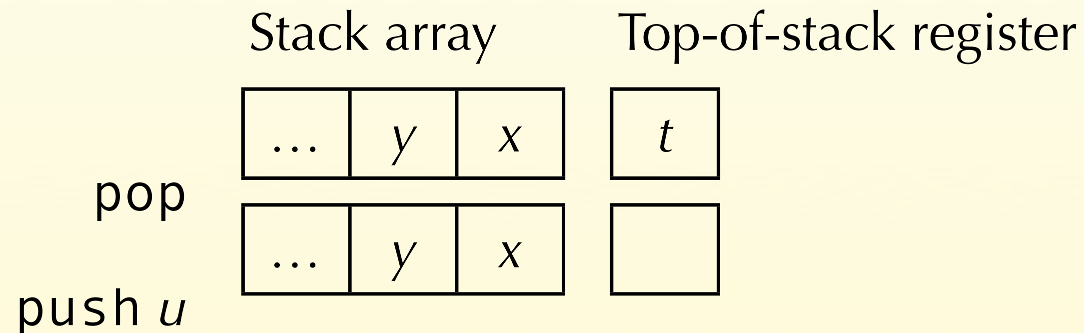
For example, here is what happens when caching *at most* one stack element in a register:



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.

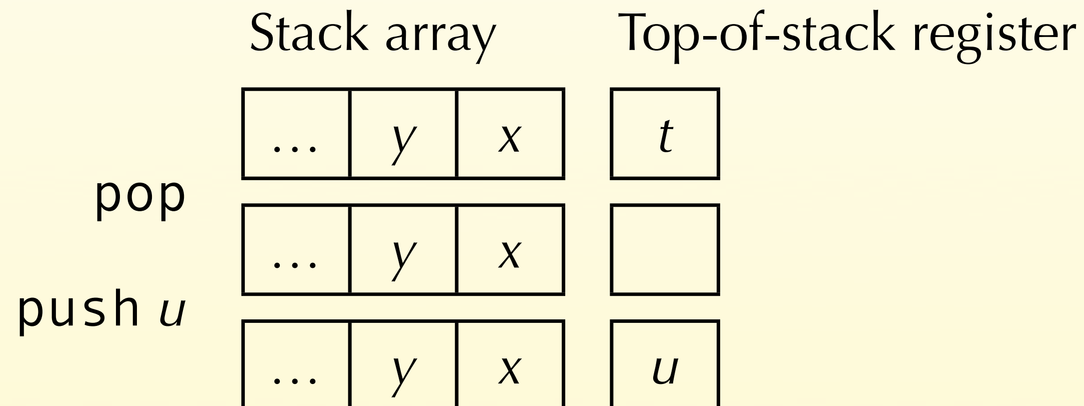
For example, here is what happens when caching *at most* one stack element in a register:



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.

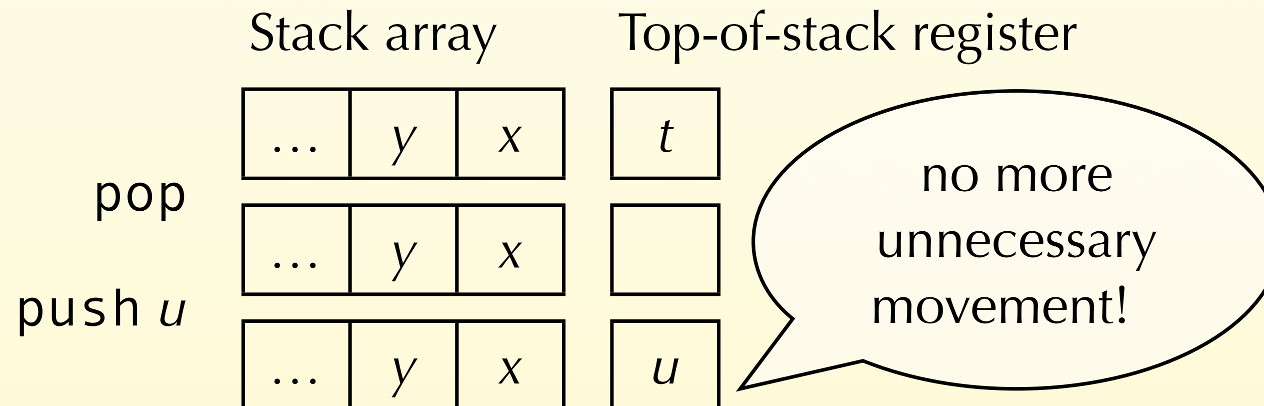
For example, here is what happens when caching *at most* one stack element in a register:



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

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
```

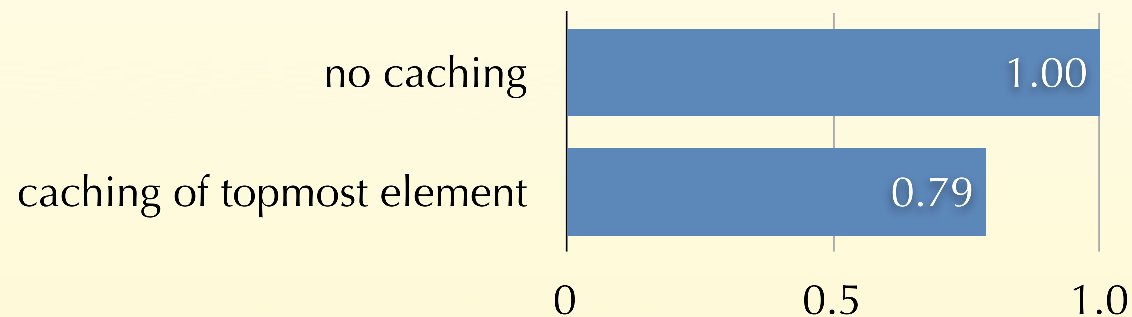
State 1: top-of-stack in reg.

```
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 `gcc` 4.0.1 with maximum optimisations, and run on an Intel Core 2 Duo.

The normalised times are presented below, and show that top-of-stack caching brought a 21% improvement to the interpreter.



Super-instructions

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.

Dynamic 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**.

This technique 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.

Just-in-time compilation

Just-in-time compilation

Virtual machines can be sped up through the use of **just-in-time** (**JIT**) – or **dynamic** – **compilation**.

The basic idea is relatively simple: instead of interpreting a piece of code, first compile it to native code – at run time – and then execute the compiled code.

In practice, care must be taken to ensure that the cost of compilation followed by execution of compiled code is not greater than the cost of interpretation!

JIT: how to compile?

JIT compilers have one constraint that “off-line” compilers do not have: they must be fast – fast enough to make sure the time lost compiling the code is regained during its execution.

For that reason, JIT compilers usually do not use costly optimisation techniques, at least not for the whole program.

JIT: what to compile?

Some code is executed only once over the whole run of a program. It is usually faster to interpret that code than go through JIT compilation.

Therefore, it is better to start by interpreting all code, and monitor execution to see which parts of the code are executed often – the so-called **hot spots**.

Once the hot spots are identified, they can compiled to native code, while the rest of the code continues to be interpreted.

Virtual machine generators

Virtual machine generators

Several tools have been written to automate the creation of virtual machines based on a high-level description.

`vmgen` is such a tool, which we will briefly examine.

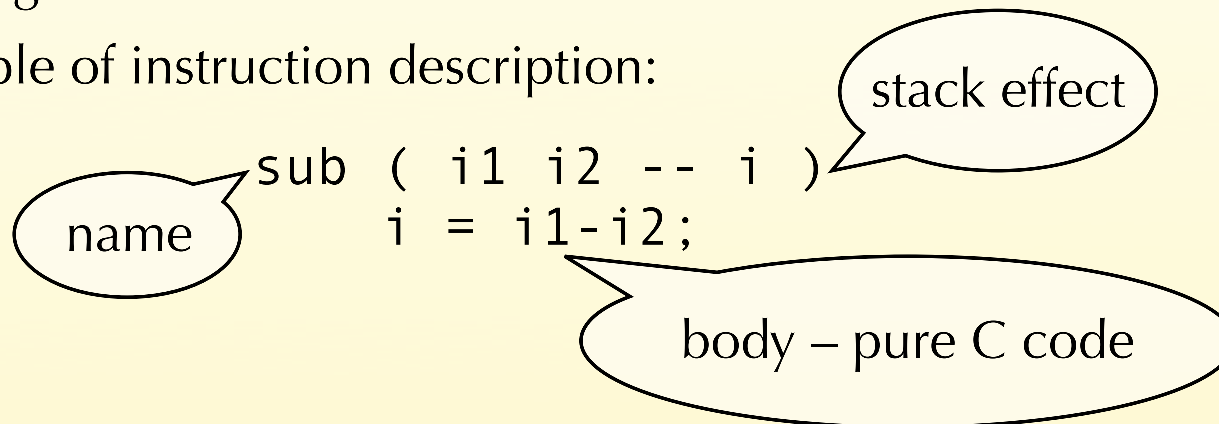
vmgen

Based on a single description of the VM, `vmgen` can produce:

- an efficient interpreter, with optional tracing,
- a disassembler,
- a profiler.

The generated interpreters include all the optimisations we have seen – threaded code, super-instructions, top-of-stack caching – and more.

Example of instruction description:



Real-world example:
the Java Virtual Machine

The Java Virtual Machine

With Microsoft's Common Language Runtime (CLR), the **Java Virtual Machine** (**JVM**) is certainly the best known and most used virtual machine.

Its main characteristics are:

- it is stack based,
- it includes most of the high-level concepts of Java 1.0: classes, interfaces, methods, exceptions, monitors, etc.
- it was designed to enable verification of the code before execution.

Notice that the JVM has remained the same since Java 1.0. All recent improvements to the Java language were implemented by changing the compiler.

The JVM model

The JVM is composed of:

- a stack, used to store intermediate values,
- a set of local variables private to the method being executed, which include the method's arguments,
- a heap, from which objects are allocated – deallocation is performed automatically by the garbage collector.

It accepts **class files** as input, each of which contains the definition of a single class or interface. These class files are loaded on-demand as execution proceeds, starting with the class file containing the main method of the program.

The language of the JVM

The JVM has 201 instructions to perform various tasks like loading values on the stack, computing arithmetic expressions, jumping to different locations, etc.

One interesting feature of the JVM is that all instructions are typed. This feature is used to support verification.

Example instructions:

- `iadd` – add the two integers on top of stack, and push back result,
- `invokevirtual` – invoke a method, using the values on top of stack as arguments, and push back result,
- etc.

The factorial on the JVM

```
static int fact(int x) {
  return x == 0 ? 1 : x * fact(x - 1);
}
```

byte code

```
 0: iload_0
 1: ifne 8
 4: iconst_1
 5: goto 16
 8: iload_0
 9: iload_0
10: iconst_1
11: isub
12: invokestatic fact
15: imul
16: ireturn
```

The factorial on the JVM

```
static int fact(int x) {
  return x == 0 ? 1 : x * fact(x - 1);
}
```

byte code	stack contents
`0: iload_0`	`[int]`
`1: ifne 8`	`[]`
`4: iconst_1`	`[int]`
`5: goto 16`	`[int]`
`8: iload_0`	`[int]`
`9: iload_0`	`[int,int]`
`10: iconst_1`	`[int,int,int]`
`11: isub`	`[int,int]`
`12: invokestatic fact`	`[int,int]`
`15: imul`	`[int]`
`16: ireturn`	`[]`

Byte code verification

A novel feature of the JVM is that it verifies programs before executing them, to make sure that they satisfy some safety requirements.

To enable this, all instructions are typed and several restrictions are put on programs, for example:

- it must be possible to compute statically the type of all data on the stack at any point in a method,
- jumps must target statically known locations – indirect jumps are forbidden.

Sun’s HotSpot JVM

HotSpot is Sun’s implementation of the JVM. It is a quite sophisticated VM, featuring:

- an interpreter including all optimisations we have seen,
- the automatic detection of hot spots in the code, which are then JIT compiled,
- two separate JIT compilers:
  1. a **client** compiler, fast but non-optimising,
  2. a **server** compiler, slower but optimising (based on SSA).

Summary

Interpreters enable the execution of a program without having to compile it to native code, thereby simplifying the implementation of programming languages.

Virtual machines are the most common kind of interpreters, and are a good compromise between ease of implementation and speed.

Several techniques exist to make VMs fast: threaded code, top-of-stack caching, super-instructions, JIT compilation, etc.