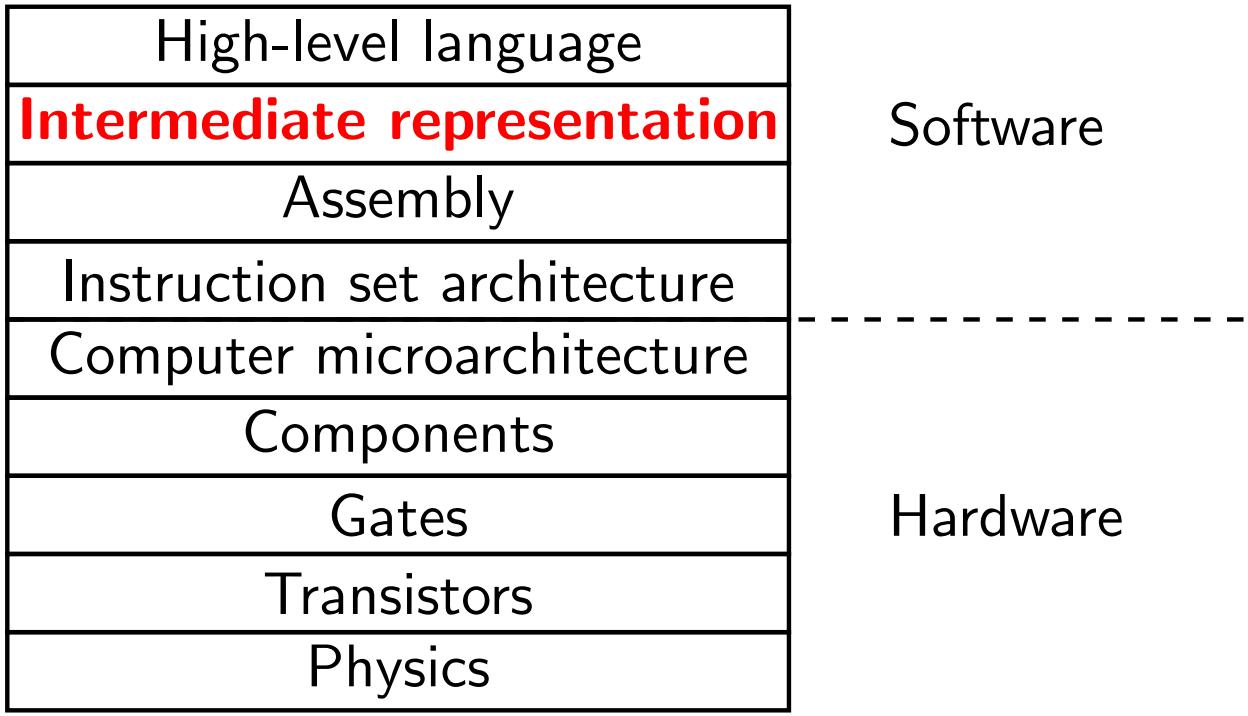
Virtual machines and intermediate representations

John Lapinskas, University of Bristol

Our next goal



Recall that most high-level languages are compiled to an **intermediate representation** (**IR**) before further compilation to assembly.

We'll spend the next two weeks learning an IR for Hack called **Hack VM**, and writing a **VM translator** to compile Hack VM code into assembly.

Why use an IR?

An IR allows a clean separation of compilation into three phases:

Front end	High-level language to IR (including optimisations). Architecture-independent, language-dependent.
Middle end	Optimisations within IR. Architecture-independent, language-independent.
Back end	IR to assembly (including optimisations). Architecture-dependent, language-independent.

Compilers for many languages can and do use the same IR. When creating a new compiler based on an existing IR, you only have to make a front end.

When ISAs/microarchitectures update, only the back end needs changing... so this work can (mostly!) be done once, rather than once per compiler.

IRs for real languages

Some prominent real-world IRs include:

- **LLVM** was designed from the ground up to be a general-purpose open-source IR suited to as many languages as possible. Most new compilers are built as front-ends for LLVM.
- **GIMPLE** is the (most important) IR used by GCC. GCC compares well to LLVM-based compilers on C and C++ performance, with significantly stronger support for older architectures.
- **JVM** is the IR used by Java (next TB). Java avoids using a compiler back end entirely and instead runs a VM to execute the IR directly — this guarantees portability, but at the cost of performance.

Virtual machines

Recall that in Hack, the designs of the assembly, the ISA, and the microarchitecture were bound together very tightly. In deciding which instructions to support, we must compromise between:

- What would be useful in coding Hack assembly.
- What can fit into a single 16-bit word in the ISA.
- What can be implemented easily in the microarchitecture.

In an IR, we sever these connections. We think of IR code as running on a **virtual machine** (**VM**) — a computer that doesn't exist as physical hardware, but that is being simulated by a computer that does exist as physical hardware.

Annoyingly, the term VM is often used both for concrete instances of this "computer" running on physical hardware and for the design of the "computer". So you might talk about both "running two Hack VMs at once on Windows" and "the instruction set of the Hack VM".

The other kind of virtual machine

In this unit, we will be concerned with VMs for IRs, which will likely never be implemented in real hardware.

VMs of real-world architectures are also commonly used for reasons like:

The other kind of virtual machine

In this unit, we will be concerned with VMs for IRs, which will likely never be implemented in real hardware.

VMs of real-world architectures are also commonly used for reasons like:

- Running untrusted software without allowing it to access the "real" underlying operating system and hardware.

The other kind of virtual machine

In this unit, we will be concerned with VMs for IRs, which will likely never be implemented in real hardware.

VMs of real-world architectures are also commonly used for reasons like:

- Running untrusted software without allowing it to access the "real" underlying operating system and hardware.
- Multiple physical computers running synchronised copies of a single virtual server to allow for failover — if one computer crashes, the others take over.

The other kind of virtual machine

In this unit, we will be concerned with VMs for IRs, which will likely never be implemented in real hardware.

VMs of real-world architectures are also commonly used for reasons like:

- Running untrusted software without allowing it to access the "real" underlying operating system and hardware.
- Multiple physical computers running synchronised copies of a single virtual server to allow for failover — if one computer crashes, the others take over.
- One physical supercomputer running multiple virtual machines for distributed high-performance computing (e.g. AWS, BC4).

The other kind of virtual machine

In this unit, we will be concerned with VMs for IRs, which will likely never be implemented in real hardware.

VMs of real-world architectures are also commonly used for reasons like:

- Running untrusted software without allowing it to access the "real" underlying operating system and hardware.
- Multiple physical computers running synchronised copies of a single virtual server to allow for failover — if one computer crashes, the others take over.
- One physical supercomputer running multiple virtual machines for distributed high-performance computing (e.g. AWS, BC4).
- Analysing a running operating system from the outside (e.g. to find security flaws).

The other kind of virtual machine

In this unit, we will be concerned with VMs for IRs, which will likely never be implemented in real hardware.

VMs of real-world architectures are also commonly used for reasons like:

- Running untrusted software without allowing it to access the "real" underlying operating system and hardware.
- Multiple physical computers running synchronised copies of a single virtual server to allow for failover — if one computer crashes, the others take over.
- One physical supercomputer running multiple virtual machines for distributed high-performance computing (e.g. AWS, BC4).
- Analysing a running operating system from the outside (e.g. to find security flaws).
- Running software native to another architecture (e.g. to test while developing for mobile).

The other kind of virtual machine

In this unit, we will be concerned with VMs for IRs, which will likely never be implemented in real hardware.

VMs of real-world architectures are also commonly used for reasons like:

- Running untrusted software without allowing it to access the "real" underlying operating system and hardware.
- Multiple physical computers running synchronised copies of a single virtual server to allow for failover — if one computer crashes, the others take over.
- One physical supercomputer running multiple virtual machines for distributed high-performance computing (e.g. AWS, BC4).
- Analysing a running operating system from the outside (e.g. to find security flaws).
- Running software native to another architecture (e.g. to test while developing for mobile).
- If a game engine was taken over by a dangerously insane clownlord who retroactively made it pay-per-install for developers, anyone could use VMs to automatically install hundreds of copies on distinct machines and rack up huge bills for any developer they didn't like.

The other kind of virtual machine

In this unit, we will be concerned with VMs for IRs, which will likely never be implemented in real hardware.

VMs of real-world architectures are also commonly used for reasons like:

- Running untrusted software without allowing it to access the "real" underlying operating system and hardware.
- Multiple physical computers running synchronised copies of a single virtual server to allow for failover — if one computer crashes, the others take over.
- One physical supercomputer running multiple virtual machines for distributed high-performance computing (e.g. AWS, BC4).
- Analysing a running operating system from the outside (e.g. to find security flaws).
- Running software native to another architecture (e.g. to test while developing for mobile).
- If a game engine was taken over by a dangerously insane clownlord who retroactively made it pay-per-install for developers, anyone could use VMs to automatically install hundreds of copies on distinct machines and rack up huge bills for any developer they didn't like.

These uses are beyond the scope of the unit, but if someone says "virtual machine" outside the context of compilers, this is probably what they mean!

Goals of Hack VM

- Implement function calls! (Next week...)

Goals of Hack VM

- Implement function calls! (Next week...)
- Proper compile-time memory allocation! (Next week...)

Goals of Hack VM

- Implement function calls! (Next week...)
- Proper compile-time memory allocation! (Next week...)
- Multi-file compilation support for libraries. (Next week...)

Goals of Hack VM

- Implement function calls! (Next week...)
- Proper compile-time memory allocation! (Next week...)
- Multi-file compilation support for libraries. (Next week...)
- Cleaner code for arithmetic/logical expressions like

$$(x \ \& \ y) \mid (x \ \& \ z) + 5 > y + y - 42.$$

Goals of Hack VM

- Implement function calls! (Next week...)
- Proper compile-time memory allocation! (Next week...)
- Multi-file compilation support for libraries. (Next week...)
- Cleaner code for arithmetic/logical expressions like

$$(x \ \& \ y) \mid (x \ \& \ z) + 5 > y + y - 42.$$

- More working storage available than just the D register.

Goals of Hack VM

- Implement function calls! (Next week...)
- Proper compile-time memory allocation! (Next week...)
- Multi-file compilation support for libraries. (Next week...)
- Cleaner code for arithmetic/logical expressions like

$$(x \ \& \ y) \mid (x \ \& \ z) + 5 > y + y - 42.$$

- More working storage available than just the D register.
- More compact syntax than assembly.

Goals of Hack VM

- Implement function calls! (Next week...)
- Proper compile-time memory allocation! (Next week...)
- Multi-file compilation support for libraries. (Next week...)
- Cleaner code for arithmetic/logical expressions like

$$(x \ \& \ y) \mid (x \ \& \ z) + 5 > y + y - 42.$$

- More working storage available than just the D register.
- More compact syntax than assembly.
- Memory addresses separated from physical memory.

Goals of Hack VM

- Implement function calls! (Next week...)
- Proper compile-time memory allocation! (Next week...)
- Multi-file compilation support for libraries. (Next week...)
- Cleaner code for arithmetic/logical expressions like

$$(x \ \& \ y) \mid (x \ \& \ z) + 5 > y + y - 42.$$

- More working storage available than just the D register.
- More compact syntax than assembly.
- Memory addresses separated from physical memory.

Many of these will come from a single source: basing the Hack VM around a stack, which we'll see next video.

Non-goals of Hack VM

We're laying the foundations for these goals with Hack VM, but we'll only achieve them at the end of the unit with a high-level language (**Jack**).

- Human-friendly code like variable names or for/while loops. (Hack VM code should be written by a compiler...)

Non-goals of Hack VM

We're laying the foundations for these goals with Hack VM, but we'll only achieve them at the end of the unit with a high-level language (**Jack**).

- Human-friendly code like variable names or for/while loops. (Hack VM code should be written by a compiler...)
- Direct implementation of expressions like

$$(x \ \& \ y) \mid (x \ \& \ z) + 5 > y + y - 42 \text{ or } x = y^2 + z^2.$$

Non-goals of Hack VM

We're laying the foundations for these goals with Hack VM, but we'll only achieve them at the end of the unit with a high-level language (**Jack**).

- Human-friendly code like variable names or for/while loops. (Hack VM code should be written by a compiler...)
- Direct implementation of expressions like

$$(x \ \& \ y) \mid (x \ \& \ z) + 5 > y + y - 42 \text{ or } x = y^2 + z^2.$$

- Variables with types including strings and arrays.

Non-goals of Hack VM

We're laying the foundations for these goals with Hack VM, but we'll only achieve them at the end of the unit with a high-level language (**Jack**).

- Human-friendly code like variable names or for/while loops. (Hack VM code should be written by a compiler...)
- Direct implementation of expressions like

$$(x \ \& \ y) \mid (x \ \& \ z) + 5 > y + y - 42 \text{ or } x = y^2 + z^2.$$

- Variables with types including strings and arrays.
- Abstract data types (e.g. structs in C).

Non-goals of Hack VM

We're laying the foundations for these goals with Hack VM, but we'll only achieve them at the end of the unit with a high-level language (**Jack**).

- Human-friendly code like variable names or for/while loops. (Hack VM code should be written by a compiler...)
- Direct implementation of expressions like

$$(x \ \& \ y) \mid (x \ \& \ z) + 5 > y + y - 42 \text{ or } x = y^2 + z^2.$$

- Variables with types including strings and arrays.
- Abstract data types (e.g. structs in C).
- Run-time memory allocation (e.g. malloc in C).

Non-goals of Hack VM

We're laying the foundations for these goals with Hack VM, but we'll only achieve them at the end of the unit with a high-level language (**Jack**).

- Human-friendly code like variable names or for/while loops. (Hack VM code should be written by a compiler...)
- Direct implementation of expressions like

$$(x \ \& \ y) \mid (x \ \& \ z) + 5 > y + y - 42 \text{ or } x = y^2 + z^2.$$

- Variables with types including strings and arrays.
- Abstract data types (e.g. structs in C).
- Run-time memory allocation (e.g. malloc in C).

Full disclosure: You likely won't have time to write code for all this in week 11! But you will at least understand *how* to write that code.