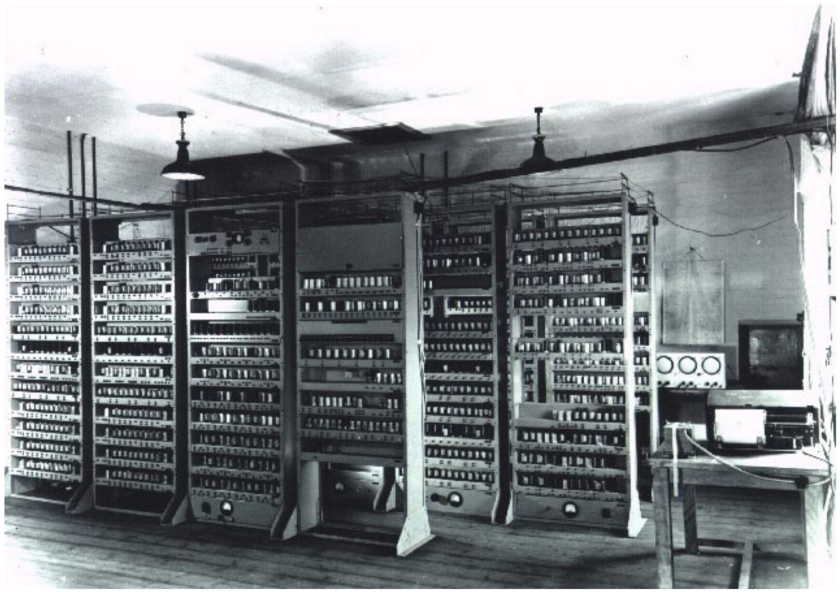
Hack assembly I: The basics
COMSM1302 Overview of Computer Architecture

John Lapinskas, University of Bristol

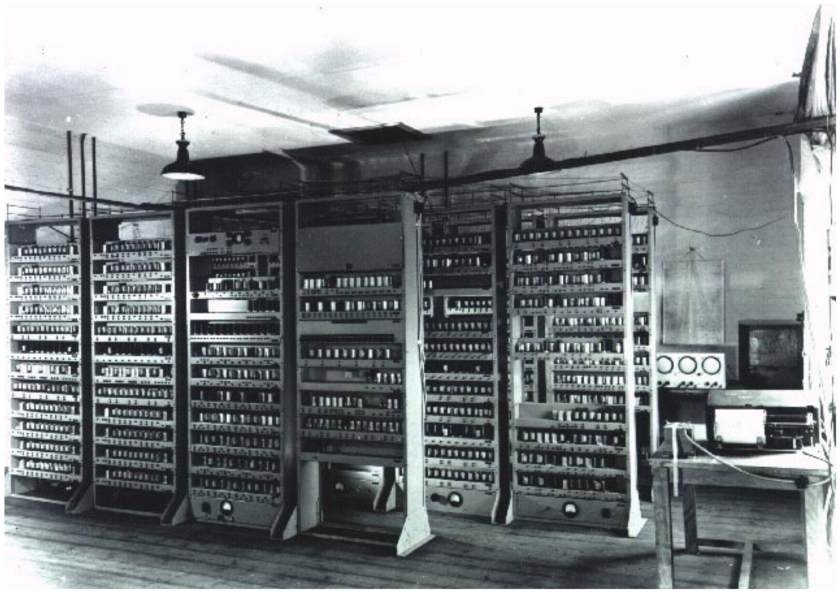
The early days of computing



Source: University of Cambridge (here)

In 1949, EDSAC was one of the first stored-program computers ever to go into regular use (contested with the Manchester Mark 1).

The early days of computing

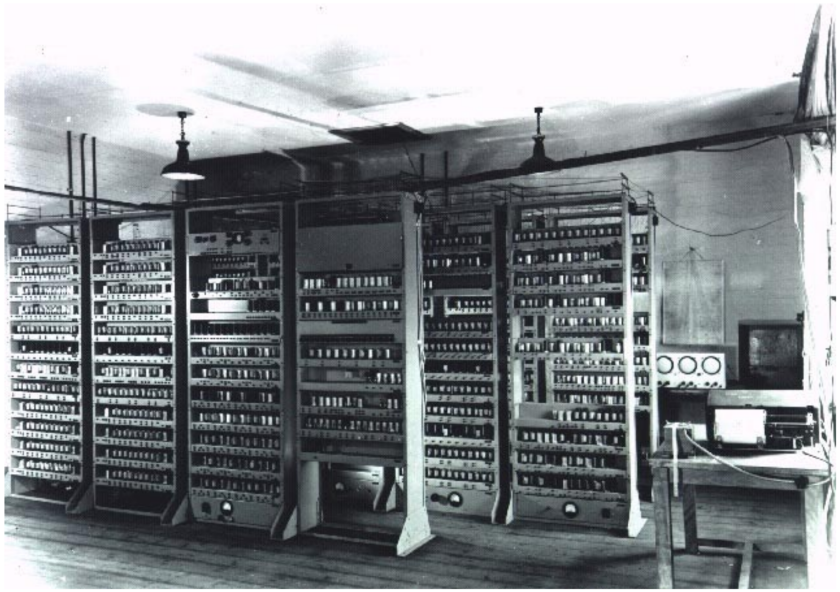


Source: University of Cambridge (here)

In 1949, EDSAC was one of the first stored-program computers ever to go into regular use (contested with the Manchester Mark 1).

And *even then* people didn't want to write in machine code.

The early days of computing



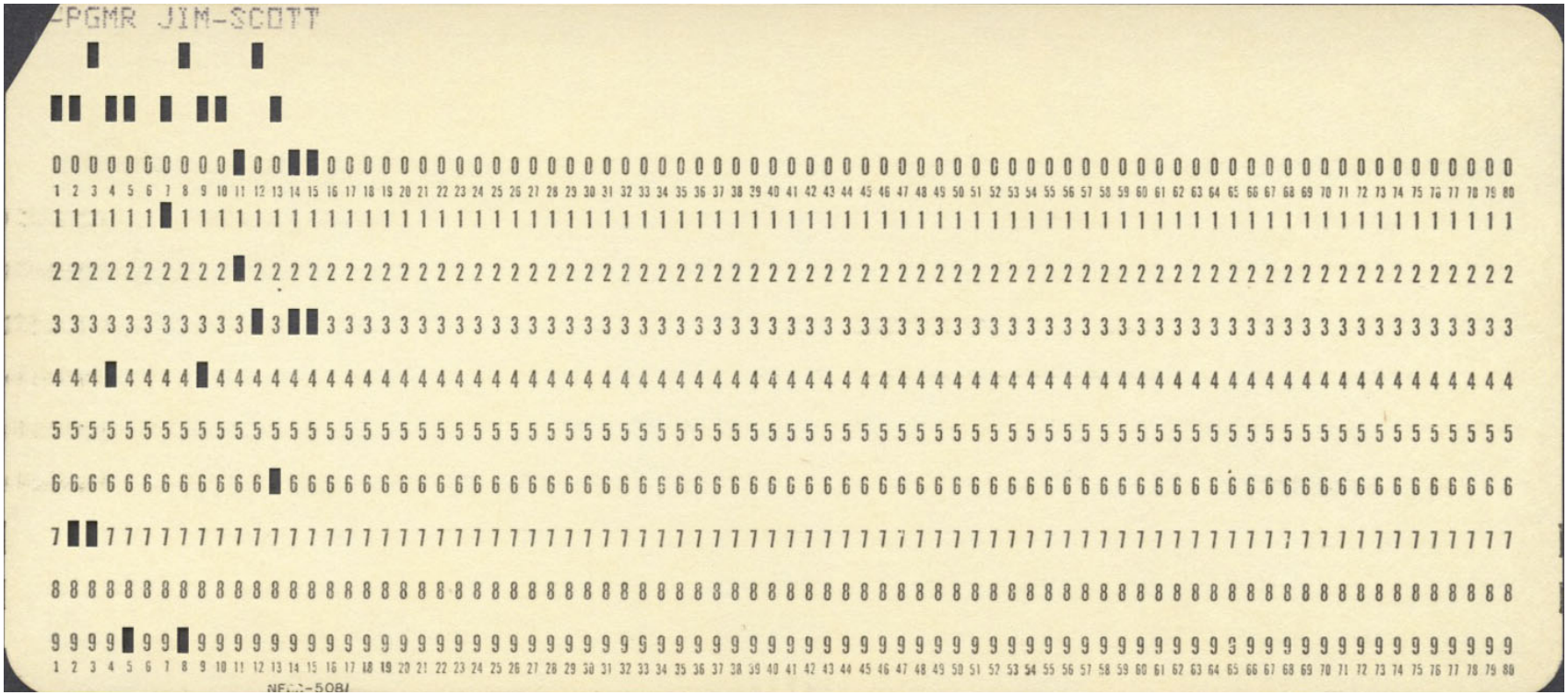
Source: University of Cambridge (here)

In 1949, EDSAC was one of the first stored-program computers ever to go into regular use (contested with the Manchester Mark 1).

And *even then* people didn't want to write in machine code.

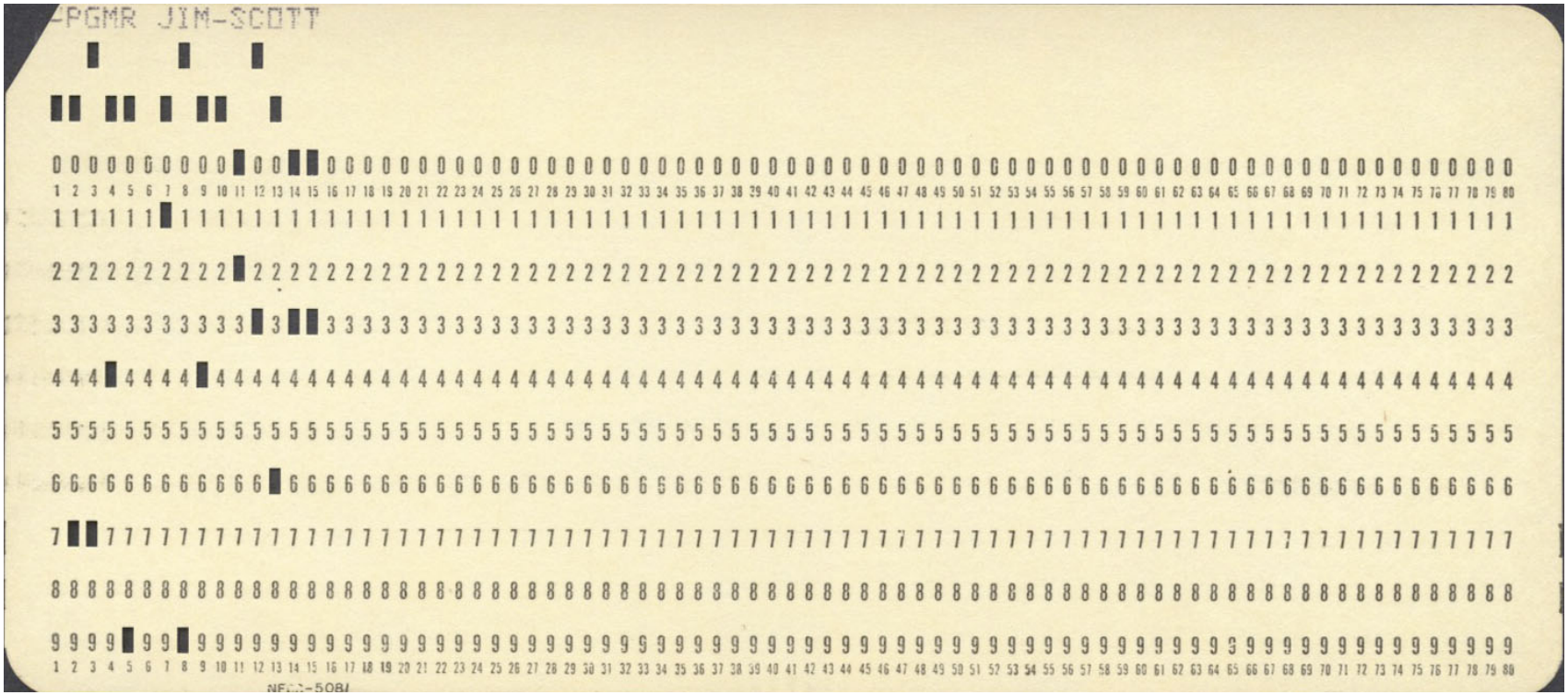
Instead, they used mnemonics: One line of text per instruction. This was the first **assembly language**, and they became ubiquitous in the 1950s. A program called an **assembler** translates assembly into machine code.

Wait, what about punchcards?



Source: Jim Scott's collection of punched cards (here)

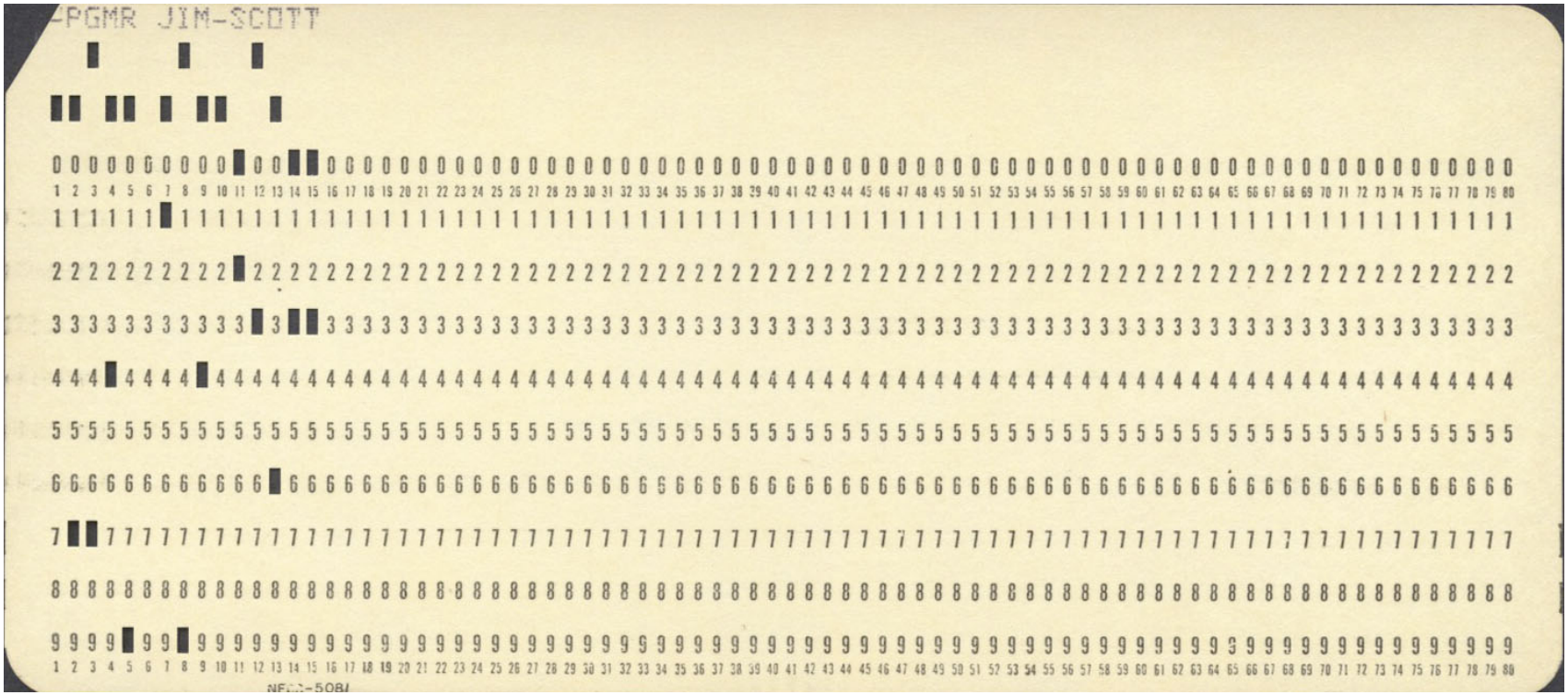
Wait, what about punchcards?



Source: Jim Scott's collection of punched cards (here)

People did program computers with machine code on punchcards even into the 1970s, but not because assembly hadn't been invented!

Wait, what about punchcards?



Source: Jim Scott's collection of punched cards (here)

People did program computers with machine code on punchcards even into the 1970s, but not because assembly hadn't been invented!

If you only have one giant mainframe in the company, it's much faster to load batches of code or data in from punchcards than it is to have people come into the computer room one by one and type it out...

The rise of the PC

Source: Telwink on Flickr (here)

Punchcards gave way to **dumb terminals** (with no computing power of their own but connected to a mainframe via a wired network) in the 1970s.

Dumb terminals gave way to **personal computers** in the 1980s–90s.

Hack assembly: Why?

We don't have to worry about this, so we will focus on Hack assembly. We'll discuss machine code later, when we assemble the Hack CPU.

Hack assembly: Why?

We don't have to worry about this, so we will focus on Hack assembly. We'll discuss machine code later, when we assemble the Hack CPU.

Since one line of assembly corresponds to one instruction, there is no one "assembly language". Different architectures (e.g. x86-64, i386, ARM, MIPS) have different assembly languages.

We teach Hack assembly not because it's directly useful, but because it primes you to learn assembly for any other architecture.

Optimised assembly code is often faster than compiled code. Most high-performance languages (e.g. C, C++, Rust) allow assembly to be embedded inline for optimisation (e.g. via `__asm__` for C with gcc).

Hack assembly: Why?

We don't have to worry about this, so we will focus on Hack assembly. We'll discuss machine code later, when we assemble the Hack CPU.

Since one line of assembly corresponds to one instruction, there is no one "assembly language". Different architectures (e.g. x86-64, i386, ARM, MIPS) have different assembly languages.

We teach Hack assembly not because it's directly useful, but because it primes you to learn assembly for any other architecture.

Optimised assembly code is often faster than compiled code. Most high-performance languages (e.g. C, C++, Rust) allow assembly to be embedded inline for optimisation (e.g. via `__asm__` for C with gcc).

Warning: Assembly is neither portable nor legible nor easily maintainable!

You should **only** "optimise" a piece of code by using assembly if you *already know* that code is a serious performance bottleneck (e.g. from profiling tools). In general, premature optimisation is a bad idea.

Registers in Hack

The Hack CPU has registers A, M, D and PC. Each holds one 16-bit word.

Registers in Hack

The Hack CPU has registers A, M, D and PC. Each holds one 16-bit word.

A is the **address register**. It's the only register we can load values into directly, rather than needing to read them from memory first.

Registers in Hack

The Hack CPU has registers A, M, D and PC. Each holds one 16-bit word.

A is the **address register**. It's the only register we can load values into directly, rather than needing to read them from memory first.

M is the **memory register**. When read from, it returns RAM[*A*]. When written to, it updates RAM[*A*]. It's interpreting *A*'s value as a pointer and dereferencing it, like in C.

Registers in Hack

The Hack CPU has registers A, M, D and PC. Each holds one 16-bit word.

A is the **address register**. It's the only register we can load values into directly, rather than needing to read them from memory first.

M is the **memory register**. When read from, it returns RAM[*A*]. When written to, it updates RAM[*A*]. It's interpreting *A*'s value as a pointer and dereferencing it, like in C.

D is the **data register** for general-purpose storage that (unlike *A*) doesn't alter the value of *M*. For longer-term or larger storage, we use RAM.

Registers in Hack

The Hack CPU has registers A, M, D and PC. Each holds one 16-bit word.

A is the **address register**. It's the only register we can load values into directly, rather than needing to read them from memory first.

M is the **memory register**. When read from, it returns RAM[*A*]. When written to, it updates RAM[*A*]. It's interpreting *A*'s value as a pointer and dereferencing it, like in C.

D is the **data register** for general-purpose storage that (unlike *A*) doesn't alter the value of *M*. For longer-term or larger storage, we use RAM.

PC is the **program counter**. The next instruction to be executed is always ROM[*PC*], so by writing to *PC* we can jump around in the code. This works differently to writing to the others (see next video).

Operations on (non-PC) registers in Hack

`@[number]` loads `number` into A, e.g. `@42` sets A to 42.

Operations on (non-PC) registers in Hack

`@[number]` loads `number` into A, e.g. `@42` sets A to 42.

All other operations are of the form:

Constants	Example
Assign 0	`M=0`
Assign 1	`D=1`
Assign -1	`A=-1`

Don't try to learn this table by heart — refer back to it as needed!

Operations on (non-PC) registers in Hack

`@[number]` loads `number` into A, e.g. `@42` sets A to 42.

All other operations are of the form:

Constants	Example	**Unary** operations	Example
Assign 0	`M=0`	Identity	`D=A`
Assign 1	`D=1`	Negation	`A=-D`
Assign -1	`A=-1`	Bitwise NOT	`D=!D`
		Increment	`A=A+1`
		Decrement	`M=M-1`

Don't try to learn this table by heart — refer back to it as needed!

Operations on (non-PC) registers in Hack

`@[number]` loads `number` into A, e.g. `@42` sets A to 42.

All other operations are of the form:

Constants	Example	**Unary** operations	Example	**Binary** operations	Example
Assign 0	`M=0`	Identity	`D=A`	Addition	`D=A+D`
Assign 1	`D=1`	Negation	`A=-D`	Subtraction	`M=M-D`
Assign -1	`A=-1`	Bitwise NOT	`D=!D`	Bitwise AND	`D=D&A`
		Increment	`A=A+1`	Bitwise OR	`A=D|M`
		Decrement	`M=M-1`		

Don't try to learn this table by heart — refer back to it as needed!

Important: All binary operations are between two different registers, at least one of which is D. So e.g. `D=A+M` or `M=D+D` are not allowed.

More complex expressions like `A=D+M+A` or `D=17` are also not allowed.

Operations on (non-PC) registers in Hack

`@[number]` loads `number` into A, e.g. `@42` sets A to 42.

All other operations are of the form:

Constants	Example	**Unary** operations	Example	**Binary** operations	Example
Assign 0	`M=0`	Identity	`D=A`	Addition	`D=A+D`
Assign 1	`D=1`	Negation	`A=-D`	Subtraction	`M=M-D`
Assign -1	`A=-1`	Bitwise NOT	`D=!D`	Bitwise AND	`D=D&A`
		Increment	`A=A+1`	Bitwise OR	`A=D|M`
		Decrement	`M=M-1`		

Don't try to learn this table by heart — refer back to it as needed!

Important: All binary operations are between two different registers, at least one of which is D. So e.g. `D=A+M` or `M=D+D` are not allowed.

More complex expressions like `A=D+M+A` or `D=17` are also not allowed.

You can do also **multiple assignment**, replacing the left-hand side by two or three registers. E.g. `MD=D-M` assigns $D - M$ to both M and D.

The syntax for this is that A, M and D must appear in order — so e.g. `AMD=D+M` is valid but `DAM=D+M` is not.

Example: add.asm

```
@0
D=M
@1
D=D+M
@17
D=D+A
@2
M=D
```

This is assembly code to store RAM[0] + RAM[1] + 17 in RAM[2].

(NB all Hack programs should end with an infinite loop — see next video.)

[See a demonstration of the Hack assembler and CPU simulator.]

A small mercy: Comments and keywords

Another benefit of assembly over machine code is **comments**. The assembler skips lines starting with //, plus empty lines and *leading* spaces. Mid-line comments (e.g. "`A=D // Comment`") are also fine.

The assembler will also replace any instance of `@R0` with `@0`, `@R1` with `@1`, and so on up to `@R15` with `@15`.

We sometimes call RAM addresses 0 through 15 **virtual registers**. In hardware they're the same as any other memory address, but we think of them as being used to hold e.g. inputs and outputs.

By writing e.g. `@R5` instead of `@5`, we signal to ourselves that we care about 5 as a memory address (e.g. to read from RAM[5] with M) rather than as a literal number (to e.g. add 5 to D).

There are other keywords like this, which we'll discuss (much) later:

$$\texttt{SP} \mapsto 0, \quad \texttt{LCL} \mapsto 1, \quad \texttt{ARG} \mapsto 2, \quad \texttt{THIS} \mapsto 3, \quad \texttt{THAT} \mapsto 4,$$
$$\texttt{SCREEN} \mapsto 16384, \quad \texttt{KBD} \mapsto 24576.$$

Example: add.asm redux

```
// D <- RAM[0]
@R0
D=M
// D <- D + RAM[1]
@R1
D=D+M
// D <- D + 17
@17
D=D+A

// RAM[2] <- D
@R2
M=D
```

Much better! Well, at least a little better.

A larger mercy: Variables

The assembler also handles alphabetical **variables** in `@` statements.

The first time you write e.g. `@var`, it associates the string `var` with a unique address in RAM (say 16). It then replaces that and every future instance of `@var` with `@16`. These are case sensitive.

Addresses are assigned to variables starting from 16. E.g. if you use variables `foo`, `Foo`, and `FOO` in that order, then `@foo` becomes `@16`, `@Foo` becomes `@17` and `@FOO` becomes `@18`.

Use variables rather than memory addresses where possible — it makes code more legible. But be careful. These are not C's variables, there are no types, and the **only** memory allocation is what's explained above!

A larger mercy: Variables

The assembler also handles alphabetical **variables** in `@` statements.

The first time you write e.g. `@var`, it associates the string `var` with a unique address in RAM (say 16). It then replaces that and every future instance of `@var` with `@16`. These are case sensitive.

Addresses are assigned to variables starting from 16. E.g. if you use variables `foo`, `Foo`, and `FOO` in that order, then `@foo` becomes `@16`, `@Foo` becomes `@17` and `@FOO` becomes `@18`.

Use variables rather than memory addresses where possible — it makes code more legible. But be careful. These are not C's variables, there are no types, and the **only** memory allocation is what's explained above!

Where might it *not* be possible or sensible to use variables?

A larger mercy: Variables

The assembler also handles alphabetical **variables** in `@` statements.

The first time you write e.g. `@var`, it associates the string `var` with a unique address in RAM (say 16). It then replaces that and every future instance of `@var` with `@16`. These are case sensitive.

Addresses are assigned to variables starting from 16. E.g. if you use variables `foo`, `Foo`, and `FOO` in that order, then `@foo` becomes `@16`, `@Foo` becomes `@17` and `@FOO` becomes `@18`.

Use variables rather than memory addresses where possible — it makes code more legible. But be careful. These are not C's variables, there are no types, and the **only** memory allocation is what's explained above!

Where might it *not* be possible or sensible to use variables?

E.g. if your desired output is two lists of a hundred numbers each!
Or worse, two lists of length determined at run-time.