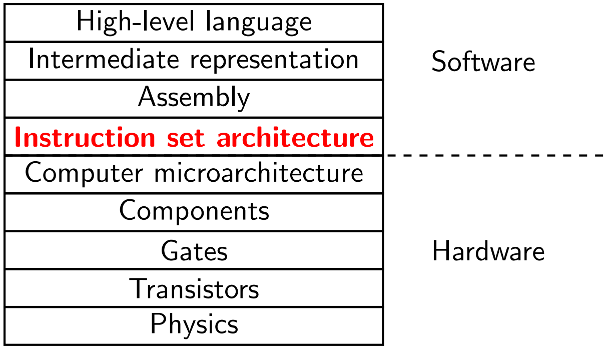
The Hack instruction set architecture
COMSM1302 Overview of Computer Architecture

John Lapinskas, University of Bristol

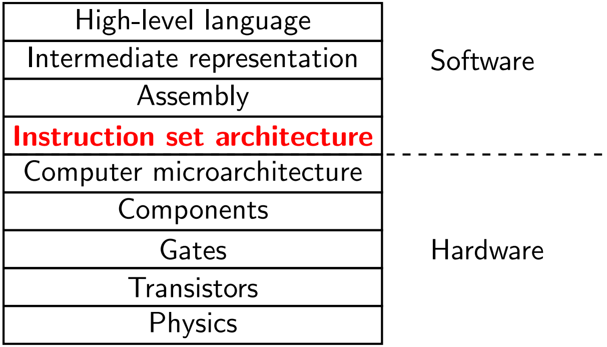
What is an ISA?


High-level language
Intermediate representation
Assembly
Instruction set architecture
Computer microarchitecture
Components
Gates
Transistors
Physics
Software
Hardware

The **microarchitecture** is the physical design of the computer in hardware — circuit diagrams and PCB layouts.

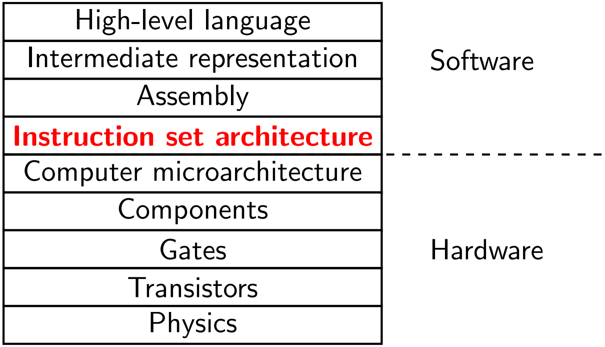
The **instruction set architecture** (**ISA**) is the way the computer acts in response to machine code instructions.

What is an ISA?


High-level language
Intermediate representation
Assembly
Instruction set architecture
Computer microarchitecture
Components
Gates
Transistors
Physics
Software
Hardware

Once you've implemented a C function to a specification (e.g. "`mult(x,y)` should return $x \times y$"), you can use that function without needing to remember how you coded it. You can change the implementation, and as long as it still returns $x \times y$, you won't introduce bugs.

What is an ISA?


High-level language
Intermediate representation
Assembly
Instruction set architecture
Computer microarchitecture
Components
Gates
Transistors
Physics
Software
Hardware

In the same way, the microarchitecture *implements* the ISA. When you're writing assembly, you don't need to know how the ISA is implemented, and your code will work on any hardware implementation without bugs.

For example, modern CPUs from both AMD and Intel generally implement the x86-64 ISA despite having very different microarchitectures.

ISAs versus microarchitecture — a comparison

ISA properties	Microarchitecture properties
Word length	Clock speed
Machine code instructions	Energy efficiency
Registers and memory	ALU circuit design
I/O memory-mapping	Connections between I/O and CPU
Execution model	Response to unspecified behaviour
(e.g. fetch-execute cycle)	(e.g. `A=D;JMP`)
⋮	⋮

We covered almost all of the Hack ISA already while covering Hack assembly. All that's missing is the machine code instructions (this video).

We've also covered most of the Hack microarchitecture in labs. Next video, we'll talk about what's left.

Your assignment this week is to build a Hack CPU in Logisim!

A-instructions

An **address instruction** or **A-instruction** is as follows:

$$\texttt{@8} \quad \leftrightarrow \quad \overset{\text{Specifies } A\text{-instruction}}{\swarrow}\!\!\!\!\!\!0\,\underbrace{000000000001000}_{\text{8 in binary}}$$

The **opcode** of an instruction says what sort of instruction it is.
The **operand(s)** are arguments to it.

An *A*-instruction has an opcode of `0`, followed by a single 15-bit operand. It simply copies its operand into A. (Note this also sets M to $RAM[A]$.)

This is why the `@` command will only work with a 15-bit operand. There's no space in the instruction for more while still fitting in an opcode.

C-instructions

A **compute** instruction or **C-instruction** is as follows:

$$\texttt{111}\,\underbrace{\texttt{1000010}}_{\texttt{comp}}\,\overbrace{\texttt{110}}^{\texttt{dest}}\,\underbrace{\texttt{111}}_{\texttt{jump}}$$

It has an opcode of 1, followed by two unused bits (which are set to `1` by convention) and three operands:

- `comp` specifies which computation to do.
- `dest` specifies where the result should be stored.
- `jump` specifies whether or not to update the program counter to A.

In the assembly instruction `MD=A+D;JMP`, `comp` corresponds to `A+D`, `dest` corresponds to `MD=`, and `jump` corresponds to `;JMP`.

C-instructions: `comp`

With input bits `a`$\texttt{c}_1\texttt{c}_2\texttt{c}_3\texttt{c}_4\texttt{c}_5\texttt{c}_6$, the computation specified by `comp` is:

$a = 0$	$a = 1$	c_1	c_2	c_3	c_4	c_5	c_6
`0`		1	0	1	0	1	0
`1`		1	1	1	1	1	1
`-1`		1	1	1	0	1	0
`D`		0	0	1	1	0	0
`A`	`M`	1	1	0	0	0	0
`!D`		0	0	1	1	0	1
`!A`	`!M`	1	1	0	0	0	1
`-D`		0	0	1	1	1	1
`-A`	`-M`	1	1	0	0	1	1
`D+1`		0	1	1	1	1	1
`A+1`	`M+1`	1	1	0	1	1	1
`D-1`		0	0	1	1	1	0
`A-1`	`M-1`	1	1	0	0	1	0
`D+A`	`D+M`	0	0	0	0	1	0
`D-A`	`D-M`	0	1	0	0	1	1
`A-D`	`M-D`	0	0	0	1	1	1
`D&A`	`D&M`	0	0	0	0	0	0
`D|A`	`D|M`	0	1	0	1	0	1

Notice that a chooses between A or M as an input.

C-instructions: `dest`

With input bits $\mathtt{d}_1\mathtt{d}_2\mathtt{d}_3$, the destination specified by `dest` is:

Destination	$\mathtt{d}_1$	$\mathtt{d}_2$	$\mathtt{d}_3$
[None]	0	0	0
`M=`	0	0	1
`D=`	0	1	0
`DM=`	0	1	1
`A=`	1	0	0
`AM=`	1	0	1
`AD=`	1	1	0
`ADM=`	1	1	1

If $\mathtt{d}_1$ is high, the computation result is stored in A. Likewise for $\mathtt{d}_2$ and D, and for $\mathtt{d}_3$ and M. If all three bits are low, the result is not stored.

C-instructions: `jump`

With input bits $\mathtt{j}_1\mathtt{j}_2\mathtt{j}_3$, the jump condition specified by `jump` is:

Condition	$\mathtt{j}_1$	$\mathtt{j}_2$	$\mathtt{j}_3$
[None]	0	0	0
;JGT	0	0	1
;JEQ	0	1	0
;JGE	0	1	1
;JLT	1	0	0
;JNE	1	0	1
;JLE	1	1	0
;JMP	1	1	1

The CPU jumps (i.e. stores A in the program counter) if:

- $\mathtt{j}_1$ is high and the computation is negative; or
- $\mathtt{j}_2$ is high and the computation is zero; or
- $\mathtt{j}_3$ is high and the computation is positive.

C-instructions: Two examples

Machine code: `1110101010000111`

C-instructions: Two examples

Machine code: `1110101010000111`

`comp:` `0101010`, i.e. `0`

C-instructions: Two examples

Machine code: `1110101010000111`

`comp:` 0101010, i.e. 0

`dest:` 000, i.e. none

C-instructions: Two examples

Machine code: `1110101010000111`

`comp:`	`0101010`, i.e. `0`
`dest:`	`000`, i.e. none
`jump:`	`111`, i.e. `;JMP`

C-instructions: Two examples

Machine code: `1110101010000111`

`comp:` `0101010`, i.e. `0`
`dest:` `000`, i.e. none
`jump:` `111`, i.e. `;JMP`

Assembly: `0;JMP`

Machine code: `1111000010101000`

C-instructions: Two examples

Machine code: `1110101010000111`

`comp:` 0101010, i.e. `0`
`dest:` 000, i.e. none
`jump:` 111, i.e. `;JMP`

Assembly: `0;JMP`

Machine code: `1111000010101000`

`comp:` 1000010, i.e. `D+M`

C-instructions: Two examples

Machine code: `1110101010000111`

`comp:`	`0101010`, i.e. `0`
`dest:`	`000`, i.e. none
`jump:`	`111`, i.e. `;JMP`

Assembly: `0;JMP`

Machine code: `1111000010101000`

`comp:`	`1000010`, i.e. `D+M`
`dest:`	`101`, i.e. `AM=`

C-instructions: Two examples

Machine code: `1110101010000111`

`comp:`	0101010, i.e. `0`
`dest:`	000, i.e. none
`jump:`	111, i.e. `;JMP`

Assembly: `0;JMP`

Machine code: `1111000010101000`

`comp:`	1000010, i.e. `D+M`
`dest:`	101, i.e. `AM=`
`jump:`	000, i.e. none

C-instructions: Two examples

Machine code: `1110101010000111`

`comp:`	0101010, i.e. `0`
`dest:`	000, i.e. none
`jump:`	111, i.e. `;JMP`

Assembly: `0;JMP`

Machine code: `1111000010101000`

`comp:`	1000010, i.e. `D+M`
`dest:`	101, i.e. `AM=`
`jump:`	000, i.e. none

Assembly: `AM=D+M`

C-instructions: Two examples

Machine code: `1110101010000111`

`comp:`	0101010, i.e. `0`
`dest:`	000, i.e. none
`jump:`	111, i.e. `;JMP`

Assembly: `0;JMP`

Machine code: `1111000010101000`

`comp:`	1000010, i.e. `D+M`
`dest:`	101, i.e. `AM=`
`jump:`	000, i.e. none

Assembly: `AM=D+M`

Go back and look at the design of the ALU from labs. Compare the behaviour of `comp` to that of the ALU output `out`.

Do you see how the ALU can be used to implement a C-instruction?

(Don't memorise any of these tables — just use them as a reference. You'll have access to a copy of them in the exam!)