CPU

Michael Dann
School of Computing Technologies

What's next...

CPU

- **Main takeaways:**
  - Instruction Set Architectures (ISAs).
  - What a compiler does.
  - Pipelining.
  - Other tricks to speed things up (superscalar execution, SIMD).
  - RISC vs CISC.

Instruction Set Architecture (ISA)

Recall from previously:

The CPU reads instructions, expressed as binary words, from memory.

Loosely speaking, there are 4 stages here: The CPU *fetches* the instruction, then *decodes* it, then *executes* it, then *writes* results somewhere.

The program counter increments, and everything repeats again...

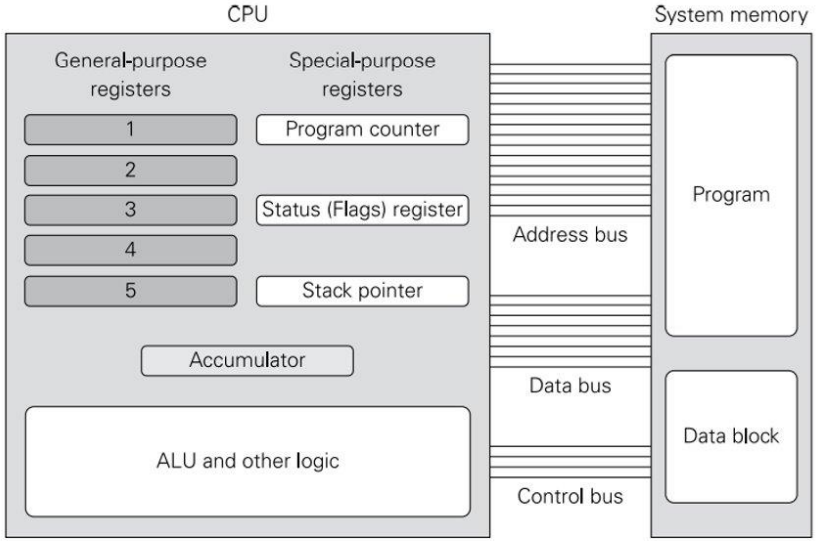


FIGURE 2-2: A simplified modern computer

Instruction Set Architecture (ISA)

LC-3 has a small, specific set of instructions including ADD, LD, AND, HALT…

However, different CPUs have different instruction set architectures (ISAs).

- For example: Intel Pentium vs Apple M1 vs Cortex A11 vs Motorola 68040, etc.
- Different word sizes.
- Different sets of instructions.

Historically, there have been two main paradigms in instruction set design:
RISC and CISC (more on this soon).

The expressivity and complexity of an instruction set have big implications, especially on execution speed and memory usage.

What does an instruction look like?

Each instruction is stored in memory as a binary word.

- One end of the word indicates what type of instruction to execute, e.g., "ADD".
- The other end of the word indicates what operands the instruction is to be applied to, e.g., "the values in registers R0 and R5".
- This information gets parsed during the *decode* step.

Programming in 1s and 0s isn't much fun. Assembly language offers a shorthand that's easier for humans to remember. It's converted to binary by an assembler.

There's only so much information you can encode about an instruction given a fixed word size. LC-3's 16-bit word size necessitates a small instruction set.

Coming up with a good ISA isn't easy, which is why standards exist.

ADDR	HEX	BINARY	LN	ASSEMBLY
			1	.ORIG x3000
x3000	x2006	0010000000000110	2	LD R0, N0
x3001	x2206	0010001000000110	3	LD R1, N1
x3002	x54A0	0101010010100000	4	AND R2, R2, #0
			5	mul_loop
x3003	x1481	0001010010000001	6	ADD R2, R2, R1
x3004	x103F	0001000000111111	7	ADD R0, R0, #-1
x3005	x03FD	0000001111111101	8	BRp mul_loop
x3006	xF025	1111000000100101	9	HALT
x3007	x0003	0000000000000011	10	N0 .FILL x0003
x3008	x0006	0000000000000110	11	N1 .FILL x0006
			12	.END

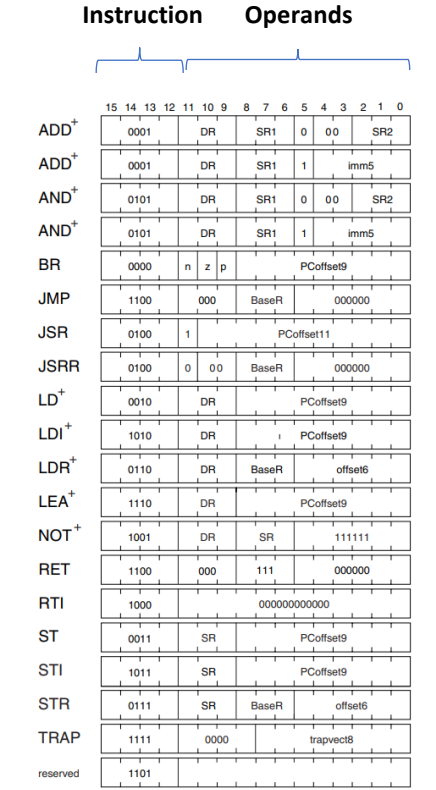


Figure A.2 Format of the entire LC-3 instruction set. **Note:** + indicates instructions that modify condition codes

Jumping and Branching

Some types of instruction are so important that they are common to virtually all ISAs, e.g.:

- An instruction for adding two values.
- An instruction for loading a value from memory into a register.

Another very important type of instruction involves being able *branch*, i.e., skipping forward or backward in the sequence of machine instructions.

- These are needed for "if" statements, subroutine calls, for loops, etc.
- Unconditional branch instructions – always jump to a given address.
- Conditional branch instructions – only perform the jump if some test is passed.

Conditional branches use a group of single-bit binary values called *flags*, which are stored on the CPU (in the flags register or status word).

Different machine instructions set different flags, giving rise to different branching strategies.

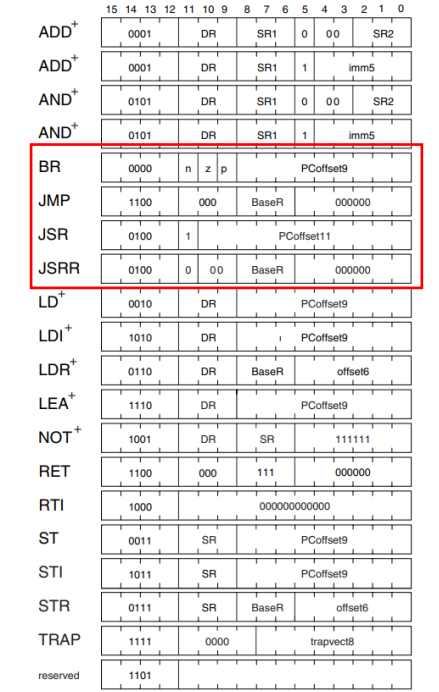


Figure A.2 Format of the entire LC-3 instruction set. **Note:** + indicates instructions that modify condition codes

CISC vs RISC

Modern architectures have elements of two main approaches to computer architectures and ISAs:

- CISC (Complex Instruction Set Computer): Large ISA containing some very complex instructions.
- RISC (Reduced Instruction Set Computer): Small ISA containing simple "building blocks".

Computer performance equation:

$$\frac{\text{time}}{\text{program}} = \frac{\text{time}}{\text{cycle}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{instructions}}{\text{program}}$$

RISC tries to minimise this.

CISC tries to minimise this.

CISC	RISC
Complex instructions in ISA, some of which will be slow.	Very simple and fast instructions, requiring a longer program.
Compiler is easier to write.	Compiler hard to write.
Large chip with lots of transistors, but not much memory required to store programs.	More memory required to store programs, which was expensive in the early days, but not so much anymore.
Used to be power inefficient but there are now power efficient implementations.	Used to be focused on power efficiency, but more general-purpose now.
Widespread in mainstream applications. Most well-known example is the Intel x86 instruction set.	More common in mobile and embedded computing (albeit this is changing e.g., Apple M1).

Microprogrammed vs Hardwired Control Unit

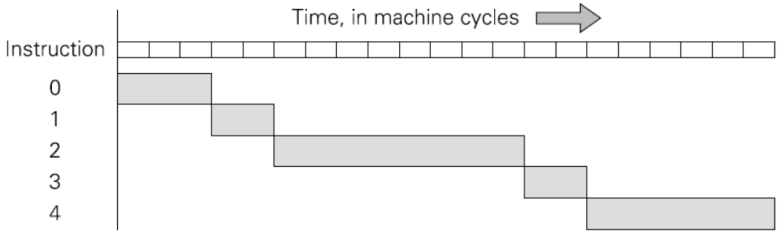


FIGURE 4-7: Machine instructions and clock cycles

Early days: Some instructions take way longer than others.

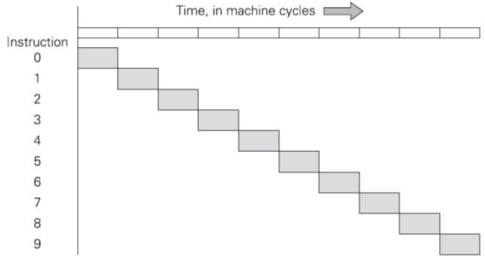


FIGURE 4-8: Single-cycle machine instructions

By about 2000, most instructions hardwired so 1 clock cycle on average. (Due to more and more silicon on the chip.)

In the early days, there wasn't much space on CPU chips for hardwiring instructions.

To address this, only *microinstructions* were hardwired in silicon.

Microcode (a list of microinstructions) was required to perform more complex instructions.

The downside of this was that it was slow: a single complex instruction could take between 4 – 40 clock cycles to execute. Some instructions took much longer to execute than others.

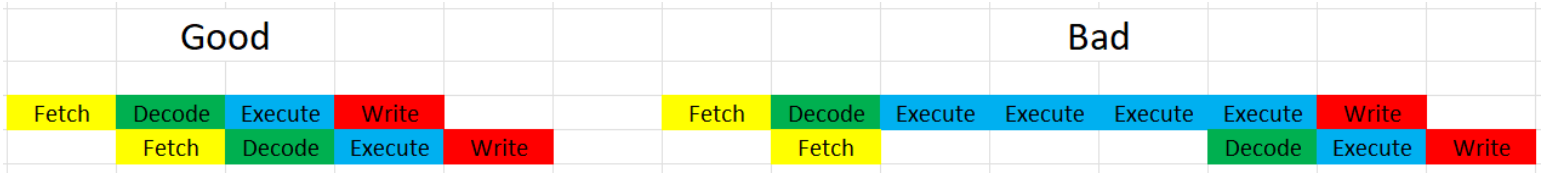
Nowadays, many more transistors can be fit on a single chip, allowing very complex instructions to be hardwired. --> Most instructions now execute in almost the same amount of time.

Pipelining

Recall that the processing of an instruction can be split into four stages: Fetch, decode, execute, write.
Once the circuitry involved in fetching an instruction has completed its task, we don't want it so sit idle!
Solution: Start fetching a new instruction before the first instruction has been fully executed.

This won't work if there's a dependency between instructions (e.g., the first instruction writes a value that the second instruction is meant to read), but smart compilers can optimise machine code to avoid such dependencies where possible.

Having consistent instruction execution times makes pipelining more efficient:



Superscalar Execution

A superscalar processor allows multiple unrelated instructions to start on the *same* clock cycle on separate hardware units or pipelines.

Note: This isn't the same thing as having multiple cores.

- When you have multiple cores, different programs run on different cores.
- In superscalar execution, the instructions come from the same program.

Instead of one pipeline running overlapping instructions, there are now two pipelines of overlapping instructions.

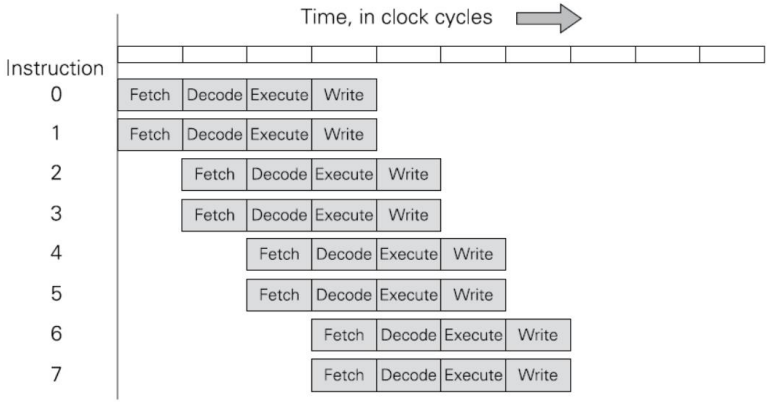


FIGURE 4-13: Superscalar execution

Theoretically, this can double the execution speed of programs, but instruction dependencies or other constraints will almost certainly prevent this.

Again, smart compilers can potentially reorder instructions to avoid issues like this.
Compilers started to become more important, and more complicated around the time superscalar execution was introduced!

Specialised circuits

From the textbook:

In modern CPUs, separate subsystems execute different groups of machine instructions:

- **Arithmetic logic unit (ALU):** Handles simple integer maths and logical operations
- **Floating point unit (FPU):** Handles floating point maths
- **Single-instruction, multiple data (SIMD) unit:** Handles vector maths that performs operations on multiple data values at once. This type of maths is essential in audio and video applications.

A modern high-performance CPU may have multiple copies of each unit to support parallel execution of instructions, as we explain a little later.

Single instruction, multiple data (SIMD)

The mathematics 3D graphics, video and machine learning, require repetitive math operations, on each component of tensors (vectors/matrices). A SIMD instruction can perform vector/matrix operations on all components at once.

Recent Raspberry Pi models have special registers of 128-bits that get split into 8 * 16 bits.

Intel first introduced MMX on Pentium CPUs. They then improved the formula via SSE instructions, providing dedicated 128-bit registers (XMM0-XMM7).

AMD provided a further extension, matched by Intel, named x86-64, adding XMM8-XMM15. The 128 bit can be configured in different ways (4 * 32, 2 * 64, etc.)

GPUs (Graphics Processing Units) rely heavily on SIMD, performing many tensor operations at once, very fast!

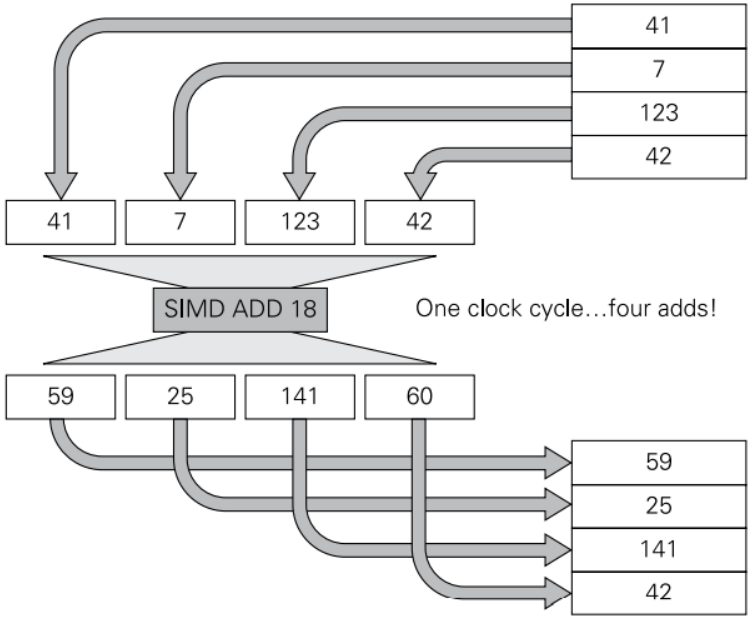


FIGURE 4-15: How SIMD instructions work

Programming in higher-level languages

Programming in assembly is problematic:

- It's very low-level, meaning it takes a lot of code to do anything of note.
- It's hard to read, not at all like a written sentence.
- Different ISAs have different instructions, so a program written for one ISA won't run under another.

Higher-level programming languages, such as C and Pascal, emerged in the 70s.

- Much more readable, and less code required.
- A *compiler* enables cross-platform compatibility.

An example of a compiler is GCC, a C compiler on Unix.

Technically, "compiling" just means translating code from one programming language into another.

Usually though it refers to translating high-level code in a text file to assembly language or binary.

In general, every new instruction set architecture needs a new port of the compiler.

Final LC-3 challenge

LAB 7

Compute Day of the Week

7.1 Problem Statement

Write an LC-3 program that given the day, month and year will return the day of the week.

7.1.1 Inputs

Before execution begins, it is assumed that locations **x31F0, 31F1,** and **x31F2** contain the following inputs:

x31F0	The usual number of the month
x31F1	The day of the month
x31F2	The year

For the example we have been using, **June 1, 2005**, we could use this code fragment in a different module: