

Lecture 3 — Computer Organization

Jeff Zarnett & Mike Cooper-Stachowsky

jzarnett@uwaterloo.ca, mstachowsky@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

March 31, 2024

A regular program like a word processor need not be concerned with the underlying hardware of the computer.

What is a program? Instructions and data.



To execute a program we need:

- 1 Main Memory**
- 2 System Bus**
- 3 Processor**

Of course, this is the minimal set.

Ideally, memory would be:

- Fast enough that the processor never has to wait;
- Large enough to hold all the data;
- Inexpensive.

The **Iron Triangle**: “fast, good, cheap; pick two.”

Good news: we can have different levels of memory.

Let us compare the various levels I might have in my laptop from 2019:

Memory Level	Access Time	Total Capacity
Register	1 ns	< 1 KB
Cache	2 ns	16 MB
Main Memory (RAM)	10 ns	64 GB
Solid State Hard Disk	250 μ s	1000 GB
Backup Hard Disk Drive	10 ms	2 TB

I am the CPU and a particular book is the piece of data needed.

If the data is in the cache: the book is on a bookshelf in my office.

If the data for the CPU on a magnetic hard disk, I have to get the book from Library and Archives Canada in Ottawa (550 km away).

And I would have to walk.

The CPU doesn't go get the data; instead it must wait for it to arrive.

What might I do in the meantime...?

This is the data transfer mechanism between most components.

It is the physical wires that connect things.

Every sort of communication using the same bus.

Contention for this resource is a limiting factor.

The original IBM PC did work like that.

A modern system has numerous buses.

- Data Bus (memory)
- Address Bus (select for r/w)
- Instruction Bus
- System Bus (peripherals)

Central Processing Unit (CPU)



The processor (CPU) is the brain of the computer.

Fetch instructions, decode them, execute them.

Okay, there's a bit more to it than this.

Fetch-decode-execute cycle repeated until the program finishes.

Different steps may be completed in parallel (**pipeline**).

Processors' largest unit is the **word**.

32-bit computer → 32-bit word. 64-bit computer → 64-bit word.

CPU instructions are specific to the processor.

Written assembly? You might have to do some in your OS.

Some operations only available in supervisor mode.
Attempting to run it in user mode is an error.

CPUs have storage locations: **registers**.

They may store data or instructions.

Management of registers is partly the role of the OS.

Let us examine a few of the critical registers.

A few of the registers in a typical CPU:

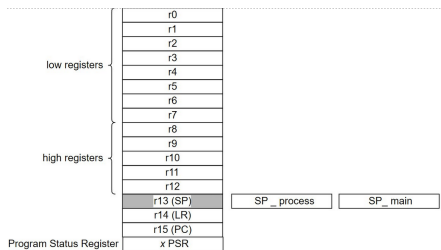
- **Program Counter**
- **Status Register**
- **Instruction Register**
- **Stack Pointer**
- **General Purpose Registers**

Who Manages the Registers?

In many programs, the compiler is responsible for register assignment.

If we write an assembly program, then our responsibility!

- 13 general purpose r0 - r12
- Stack Pointer (okay, 2)
- Link Register
- Program Counter
- xPSR (Operating Mode)



Caching is very... hit and miss.

- Nicholas Armstrong



Caching is very important in computing, and not just memory.

We examine the idea of caching in the context of memory.

It is applicable any time there is a large resource that is divided into pieces, some of which are used more often than others.

Caching provides a benefit in some circumstances; not useful in others.

The goal of caching is to speed up operations.

It is desirable to read information from cache, when possible, because it takes less time to get data from cache to the CPU than from main memory to the CPU.

CPUs are a lot faster than memory and it is best if we do not keep them waiting.

Caches do not have to operate on pages.

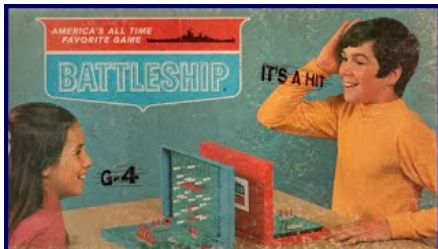
They can operate on anything, but they are typically blocks of a given size.

An entry in a cache is often called a **line**.

We will assume that a cache line maps nicely to a page.

As discussed, the CPU generates a memory address for a read/write operation.

The address will be mapped to a page; ideally that page is in the cache.



If the requested page is, in fact, in the cache, we call that a cache **hit**.

If the page is not found in the cache, it is considered a cache **miss**.

In case of a miss, we must load the page from memory, a slow operation.

A page miss is also called a **page fault**.

The percentage of the time a page is found in the cache is called the **hit ratio**.



We can calculate the effective access time if we have an estimate of the hit ratio.

The effective access time is therefore computed as:

$$\text{Effective Access Time} = h \times t_c + (1 - h) \times t_m$$

Where:

h is the hit ratio.

t_c is the time required to load a page from cache.

t_m is the time to load a page from memory.

Of course, we would like the hit ratio to be as high as possible.

And the effective access time to be as small as possible.

We Put a Cache in Your Cache...

Caches have limited size, because faster caches are more expensive.

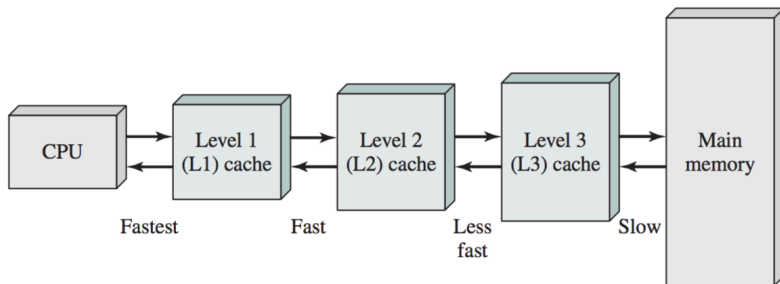
With infinite money we might put everything in registers.

Caches for memory are very often multileveled.

Intel 64-bit CPUs tend to have L1, L2, and L3 caches.

L1 is the smallest and fastest; L3 is the largest and slowest.

Multiple Levels of Cache



The effective access time formula needs to be updated and expanded.

If we have a miss in the L1 cache, the L2 cache is checked.

If the L2 cache contains the desired page, it will be copied to the L1 cache and sent to the CPU.

If it is not in L2, then L3 is checked.

If it is not there either, it is in main memory and will be retrieved from there and copied to the in-between levels on its way to the CPU.

Because caches have limited size, we have to manage this storage carefully.

Whenever a page fault occurs, the operating system needs to choose which page to **evict** from the cache to make space for the new one.

This assumes that the cache is full, which it likely is except at system startup.

We could, of course, just select a random page...

We should do this task more intelligently, if we can.

To make an intelligent decision about what sort of strategy to choose, we need to know a few things about how data is accessed in a system.

- 1 The 90/10 rule.
- 2 The principle of **temporal locality**.
- 3 The principle of **spatial locality**.

Example: a function that sums up all the values of an array.

The sum variable is accessed repeatedly, and the fact that it was recently accessed means it is likely to be accessed again soon.

The array being accessed at index i now means it is likely that the array at index $i + 1$ is likely to be accessed soon.

The Cortex M4 is a 32-bit system: instructions are (up to) 32 bits.

We can fetch more than one instruction at a time (4 to 8).
The instruction bus is 128 bit.

Think of it like reading ahead a bit in IKEA instructions.

The Cortex M4 requires CPU cycles be wasted to access flash.

At 84 MHz, 6 CPU cycles wasted.

Faster CPU clock equals more waiting.

Table 6. Number of wait states according to CPU clock (HCLK) frequency

Wait states (WS) (LATENCY)	HCLK (MHz)			
	Voltage range 2.7 V - 3.6 V	Voltage range 2.4 V - 2.7 V	Voltage range 2.1 V - 2.4 V	Voltage range 1.71 V - 2.1 V
0 WS (1 CPU cycle)	$0 < \text{HCLK} \leq 30$	$0 < \text{HCLK} \leq 24$	$0 < \text{HCLK} \leq 18$	$0 < \text{HCLK} \leq 16$
1 WS (2 CPU cycles)	$30 < \text{HCLK} \leq 60$	$24 < \text{HCLK} \leq 48$	$18 < \text{HCLK} \leq 36$	$16 < \text{HCLK} \leq 32$
2 WS (3 CPU cycles)	$60 < \text{HCLK} \leq 84$	$48 < \text{HCLK} \leq 72$	$36 < \text{HCLK} \leq 54$	$32 < \text{HCLK} \leq 48$
3 WS (4 CPU cycles)		$72 < \text{HCLK} \leq 84$	$54 < \text{HCLK} \leq 72$	$48 < \text{HCLK} \leq 64$
4 WS (5 CPU cycles)	-	-	$72 < \text{HCLK} \leq 84$	$64 < \text{HCLK} \leq 80$
5 WS (6 CPU cycles)	-	-	-	$80 < \text{HCLK} \leq 84$

Source: STM32 Reference Manual

Fetch as many instructions as we can in one go.

If we pre-fetch instructions, what if we get it wrong?

Solution in STM32F401RE: branch cache! (Be prepared)

Without pre-fetch: get instructions, do them, get next instructions.

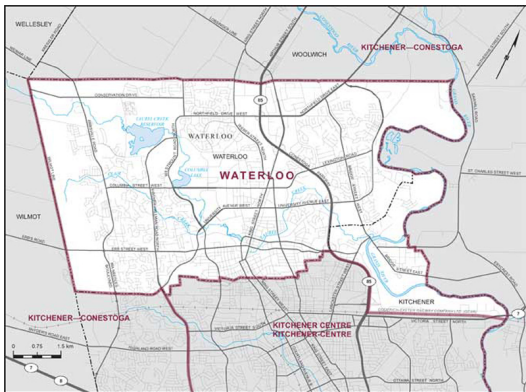
With: get instructions, do them + get next instructions in parallel.

Benefit: avoid waiting for next instructions after we're done executing.

Often instructions and data have their own caches.

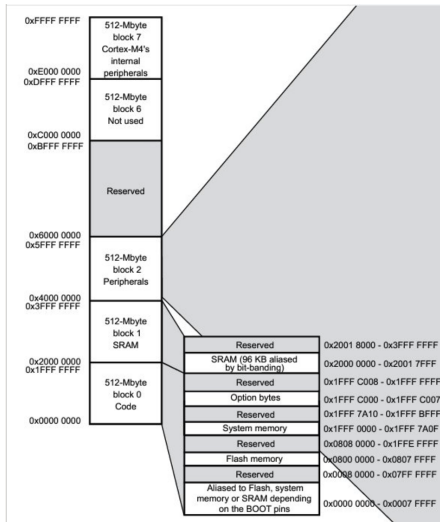
Cortex M4 has a data cache of 8×128 bits.

We can generate any memory address in the CPU; not everything goes to RAM.



Some addresses are reserved for I/O, peripherals, chip settings...

Cortex M4 Memory Map Overview



Every kind of data (RAM, Flash, etc) are accessed via pointers.

This reduces the maximum potential size of RAM.
Address ranges are limited.

Microcontrollers commonly do this; desktops usually do not.



"Is this all that I am? Is there nothing more?" – Spock, sharing V'Ger's thoughts.

No! Virtual memory is *much* more complicated than this.
We'll have to revisit that subject later.