Lecture 3 — Computer Organization

Jeff Zarnett & Mike Cooper-Stachowsky
jzarnett@uwaterloo.ca, mstachowsky@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

September 7, 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

To execute a program we need:

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

Of course, this is the minimal set.

Main Memory

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 μs	1000 GB
Backup Hard Disk Drive	10 ms	2 TB

Memory Access Analogy

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.

Memory Access Analogy

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

What might I do in the meantime…?

System Bus

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.

System Buses

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.

Central Processing Unit (CPU)

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 $\rightarrow$ 32-bit word. 64-bit computer $\rightarrow$ 64-bit word.

Central Processing Unit (CPU)

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.

Central Processing Unit (CPU)

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.

CPU Registers

A few of the registers in a typical CPU:

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

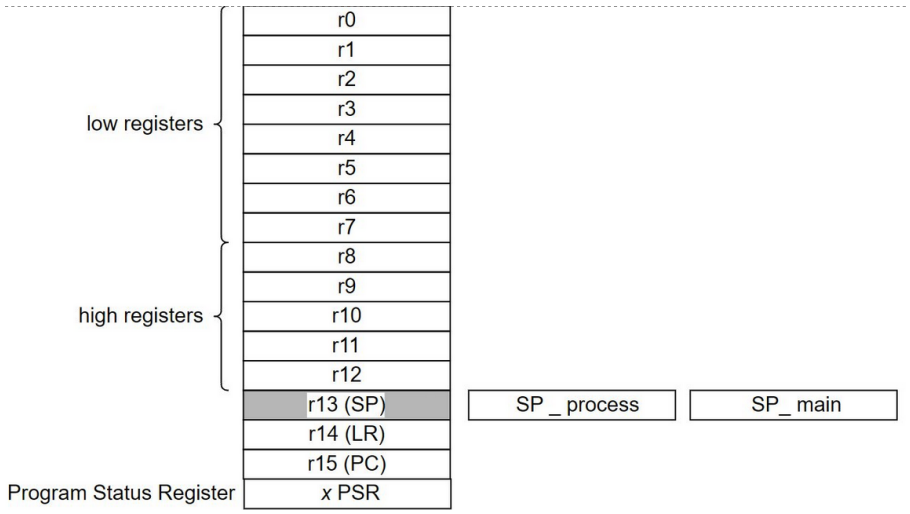
Who Manages the Registers?

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

If we write an assembly program, then our responsibility!

ARM Cortex M4 Registers

- 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

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.

Caching

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.

Variable Size?

Caches can operate on data of any size, the CPU's operate on blocks of fixed size.

An entry in a cache is often called a line.

Assume that a cache line always corresponds to a fixed-size block of memory.

Bring Me Pictures of Spiderman!

Let's imagine that our program wants to read some data:

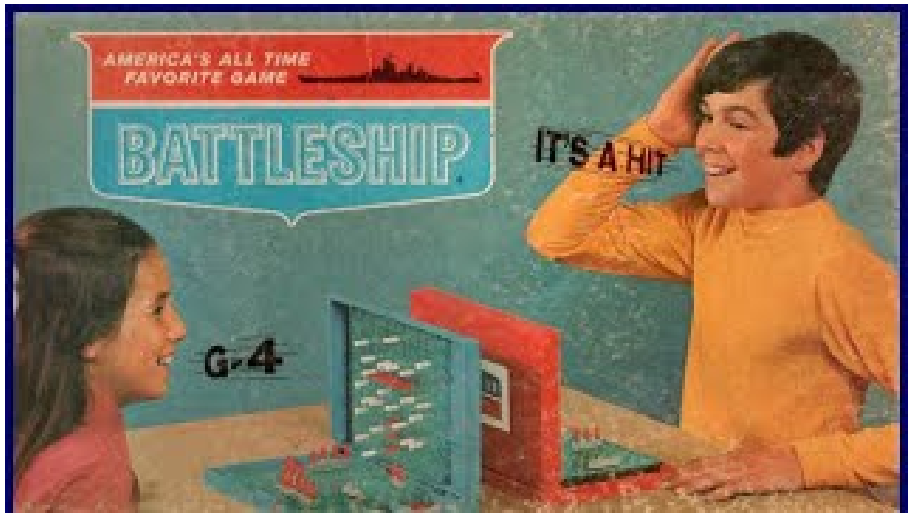
e.g., `*rate = 1.13;`

The variable `rate` is a pointer and it contains an address…

The CPU instruction looks like "store 1.13 in address `0x1234ABCD`".

Battleship Sunk!

When that write is issued, is the block containing that data already in the cache?



If the requested memory block is, in fact, in the cache, we call that a cache hit.

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

Please Play Again

In case of a miss, we must go to memory, a slow operation.

The percentage of the time a block 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.

Effective Access Time

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 block from cache.
t_m is the time to load a block 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...

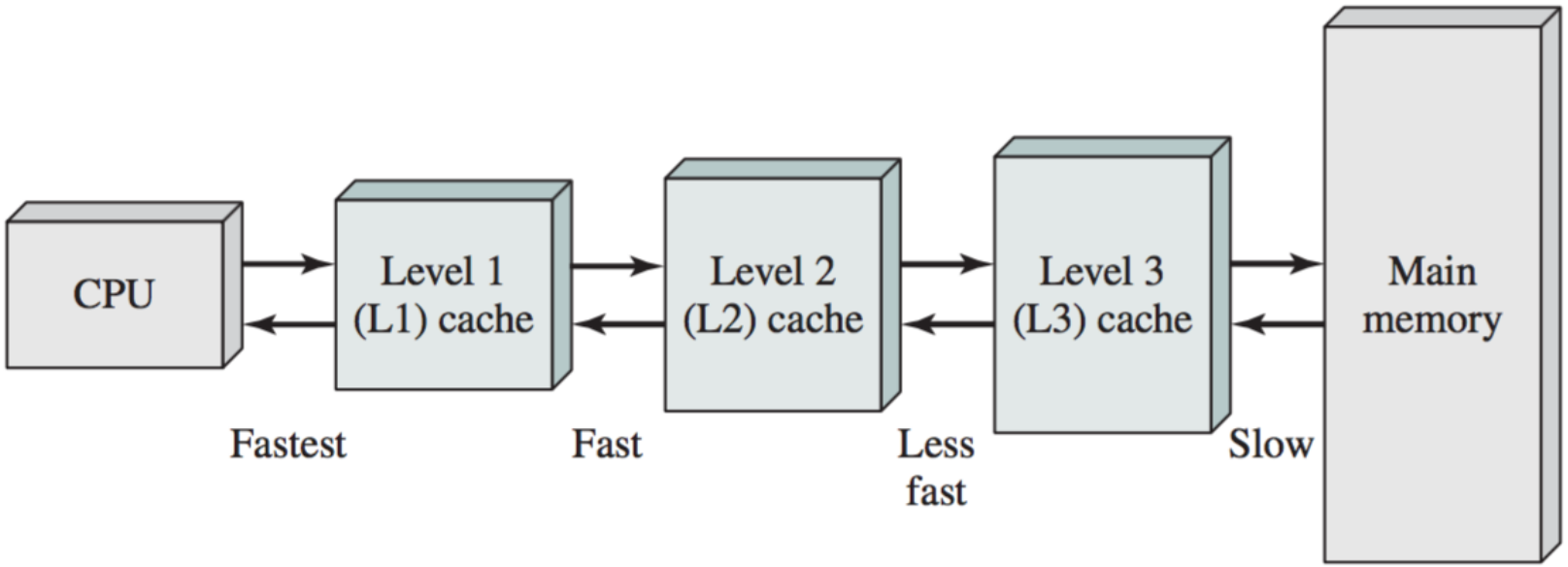
Caches have limited size, because faster caches are more expensive.

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



Note: blocks are not at all to scale.

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

Darn, Missed Again

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

If the L2 cache contains the desired block, 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.

Replacement Algorithms

Whenever a miss occurs, we have to choose which block 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 one…

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

Block Replacement Algorithms

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.

Here's some patterns about usage:

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

Temporal & 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.

Managing Cache

Managing the cache efficiently is a complicated subject and we'll want to talk about it more when we get to the memory unit of the course.

Some systems have > 1 cache – e.g., separate ones for instructions and data.

Cortex M4 has a data cache of 8×128 bits. What about instructions?

Cortex M4 Flash Frames

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.

Hurry Up and Wait

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 < HCLK≤ 30	0 < HCLK ≤24	0 < HCLK ≤ 18	0 < HCLK ≤ 16
1 WS (2 CPU cycles)	30 < HCLK ≤60	24 < HCLK ≤48	18 < HCLK ≤ 36	16 <HCLK ≤ 32
2 WS (3 CPU cycles)	60 < HCLK ≤84	48 < HCLK ≤72	36 < HCLK ≤ 54	32 < HCLK ≤48
3 WS (4 CPU cycles)		72 < HCLK ≤84	54 < HCLK ≤72	48 < HCLK ≤64
4 WS (5 CPU cycles)	-	-	72 < HCLK ≤84	64 < HCLK ≤80
5 WS (6 CPU cycles)	-	-	-	80 < HCLK ≤84

Source: STM32 Reference Manual

Sometimes We Guess Wrong

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)

Goal: Reduce Waiting

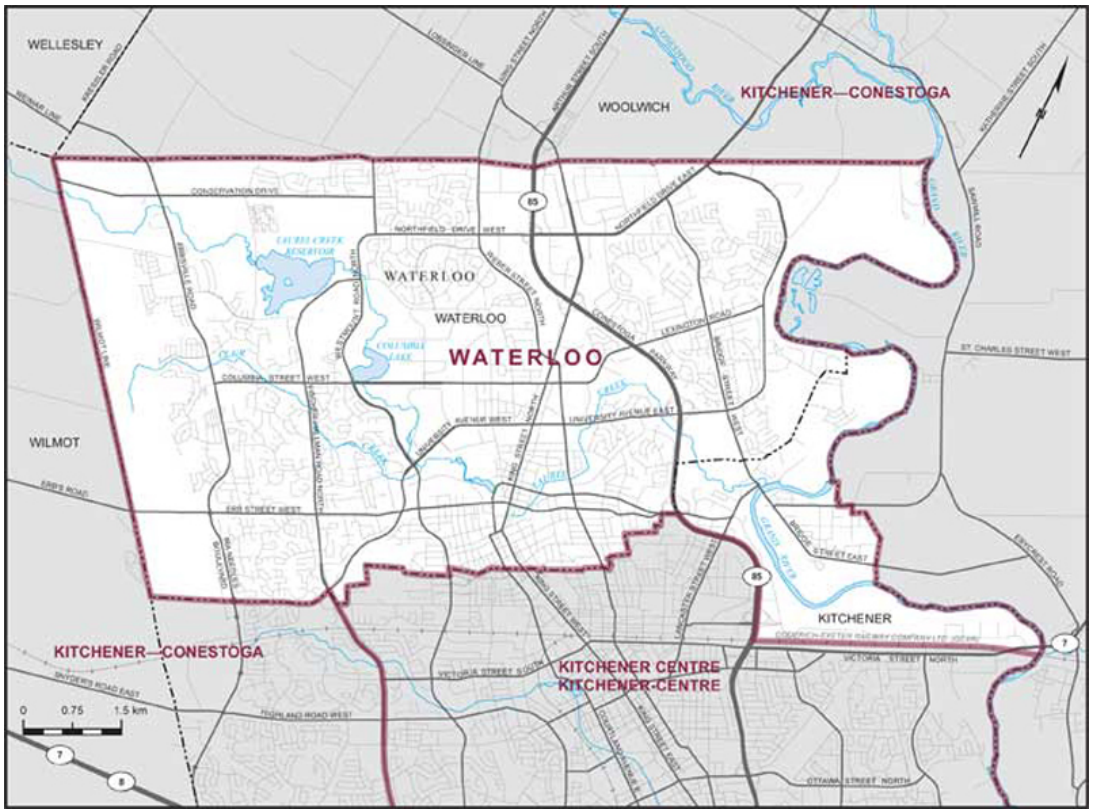
Without pre-fetch: get instructions, do them, get next instructions.

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

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

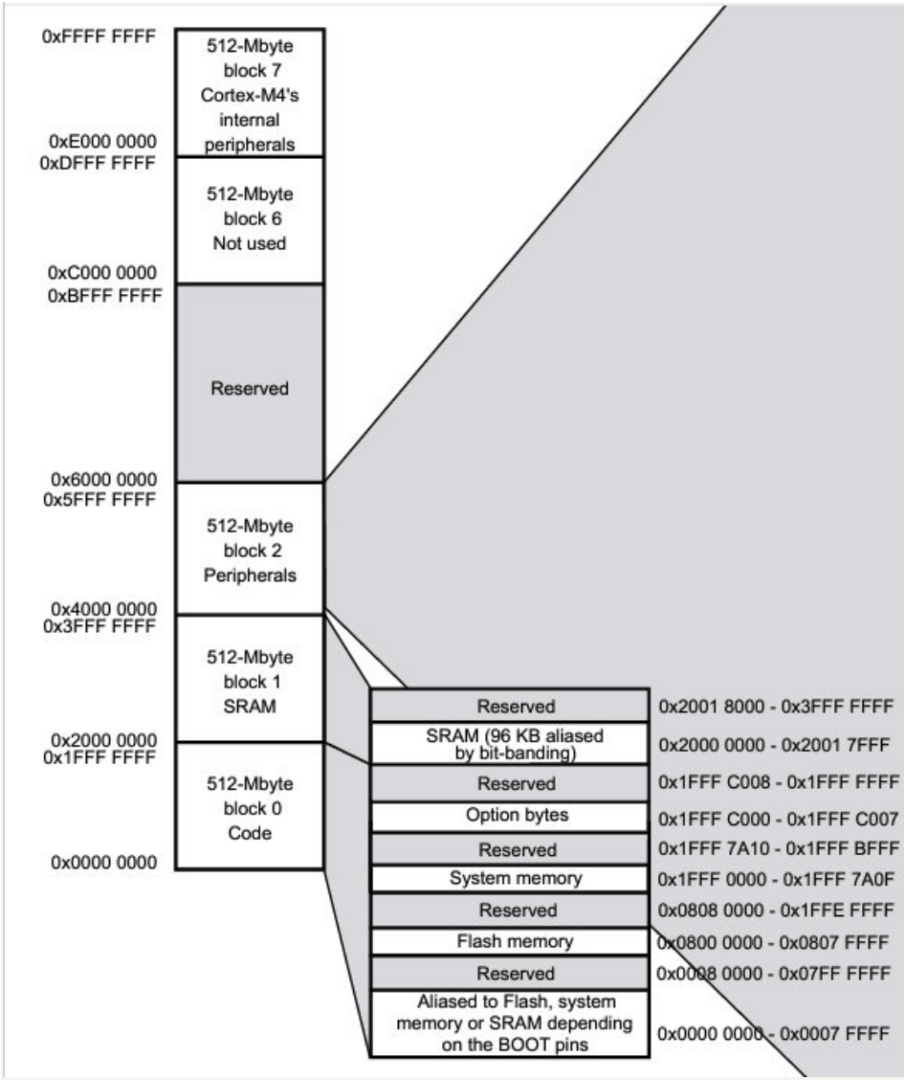
Memory Maps

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



That's Nice?

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 that it?

"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.