Computer Hardware Architecture
Lecture 4

Manfred Liebmann
Technische Universität München
Chair of Optimal Control
Center for Mathematical Sciences, M17
manfred.liebmann@tum.de

November 10, 2015

Reading List

- Pacheco - *An Introduction to Parallel Programming* (Chapter 1 - 2)
  - Introduction to computer hardware architecture from the parallel programming angle
- Hennessy-Patterson - *Computer Architecture - A Quantitative Approach*
  - Reference book for computer hardware architecture

All books are available on the Moodle platform!

UMA Architecture

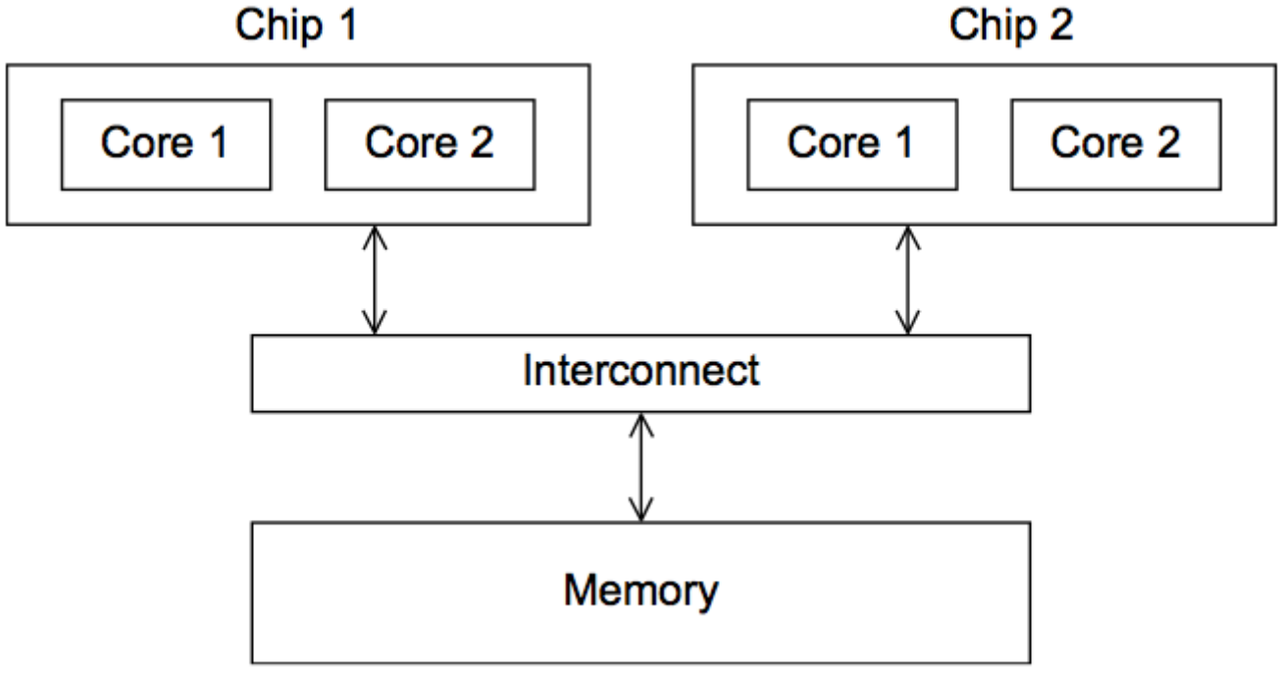

Figure 1: A uniform memory access (UMA) multicore system

Access times to main memory is the same for all cores in the system!

NUMA Architecture

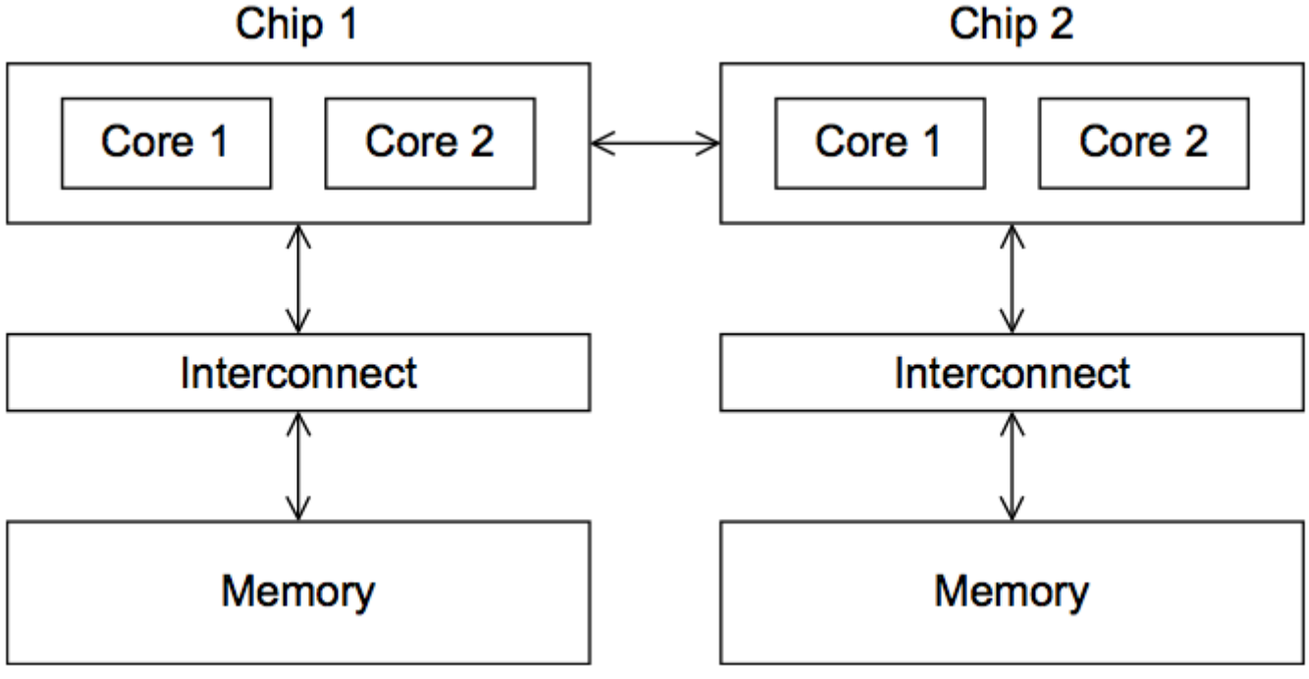

Figure 2: A nonuniform memory access (UMA) multicore system

Access times to main memory differs form core to core depending on the proximity of the main memory. This architecture is often used in dual and quad socket servers, due to improved memory bandwidth.

Cache Coherence

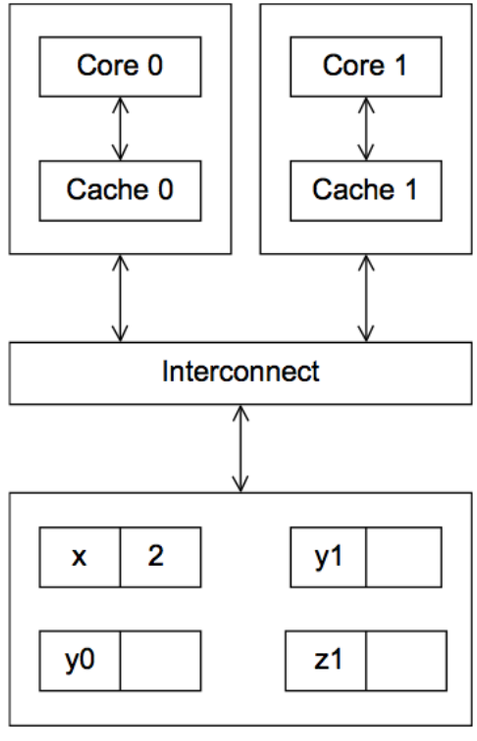

Figure 3: A shared memory system with two cores and two caches

What happens if the same data element z1 is manipulated in two different caches?

The hardware enforces *cache coherence*, i.e. consistency between the caches. **Expensive!**

False Sharing

The cache coherence protocol works on the granularity of a cache line.

If two threads manipulate *different* element within a single cache line, the cache coherency protocol is activated to ensure consistency, even if every thread is only manipulating its own data.

Expensive Operation!

Do not manipulate data with multiple threads within a single cache line! This can easily happen when using OpenMP!

Flynn's Taxonomy of Computer Architectures

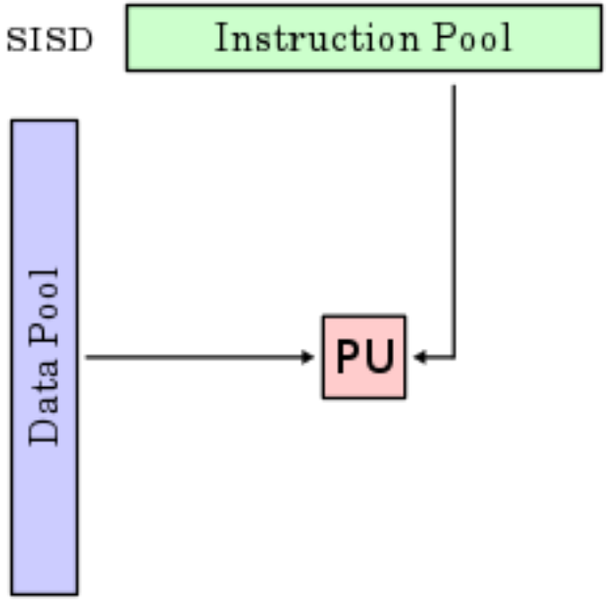

Figure 4: Flynn's taxonomy: SISD (Single instruction stream, single data stream)

A sequential computer which exploits no parallelism in either the instruction or data streams. A single control unit fetches a single instruction stream from memory. The control unit then directs a single processing unit to operate on single data stream one operation at a time.

An example of SISD architecture is a single core processor.

Flynn's Taxonomy of Computer Architectures

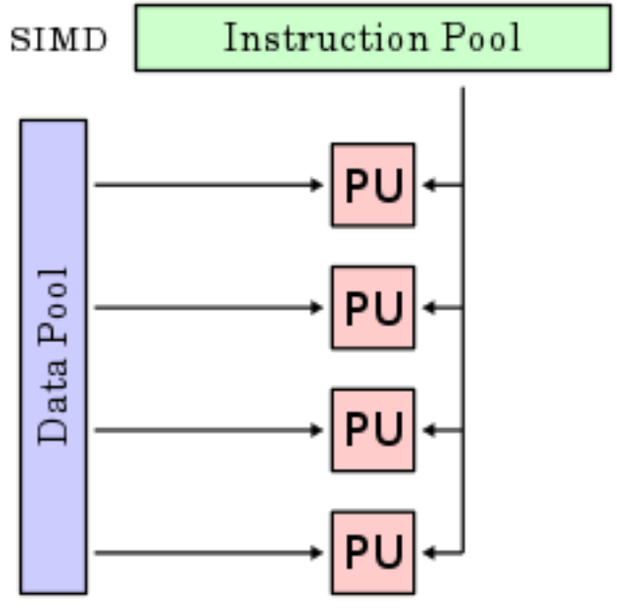

Figure 5: Flynn's taxonomy: SIMD (Single instruction stream, multiple data streams))

A computer which exploits multiple data streams against a single instruction stream to perform operations which may be naturally parallelized.

Examples are the vector units in modern CPU cores, i.e. SSE/SSE2, AVX, AVX512.

Flynn's Taxonomy of Computer Architectures

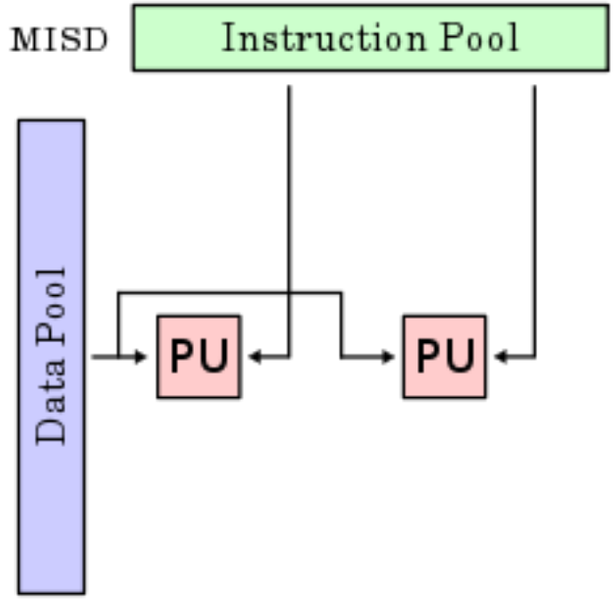

Figure 6: Flynn's taxonomy: MISD (Multiple instruction streams, single data stream)

Multiple instructions operate on a single data stream. Uncommon architecture which is generally used for fault tolerance. Heterogeneous systems operate on the same data stream and must agree on the result.

Examples are critical fault tolerant systems.

Flynn's Taxonomy of Computer Architectures

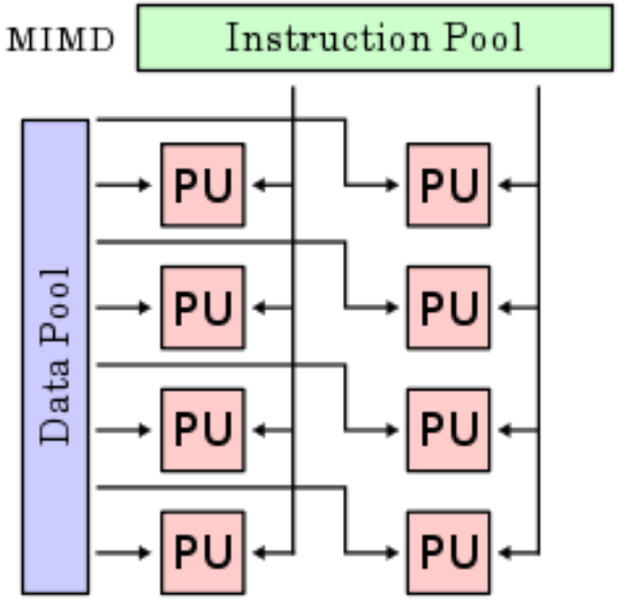

Figure 7: Flynn's taxonomy: MIMD (Multiple instruction streams, multiple data streams)

Multiple autonomous processors simultaneously executing different instructions on different data. Distributed systems are generally recognized to be MIMD architectures; either exploiting a single shared memory space or a distributed memory space. A multi-core superscalar processor is a MIMD processor.

Pipelining and Instruction-Level Parallelism

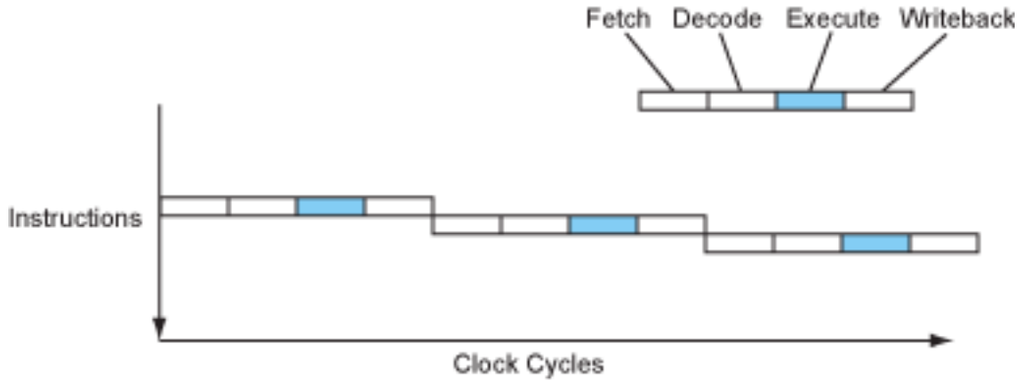

Figure 8: The instruction flow of a sequential processor.

Consider how an instruction is executed first it is fetched, then decoded, then executed by the appropriate functional unit, and finally the result is written back. With this scheme, a simple processor takes 4 cycles per instruction.

Pipelining and Instruction-Level Parallelism

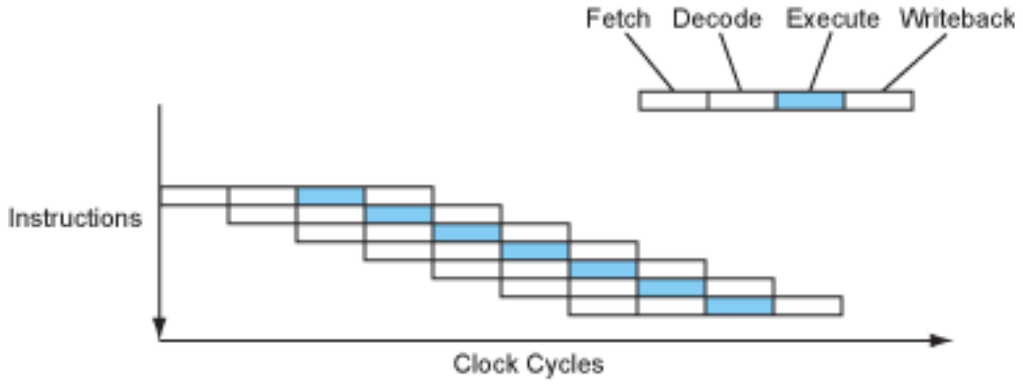

Figure 9: The instruction flow of a pipelined processor.

Modern processors overlap the stages in a pipeline, like an assembly line. While one instruction is executing, the next instruction is being decoded, and the one after that is being fetched.

Now the processor is completing one instruction every cycle. This is a four-fold speedup without changing the clock speed.

Pipelining and Instruction-Level Parallelism

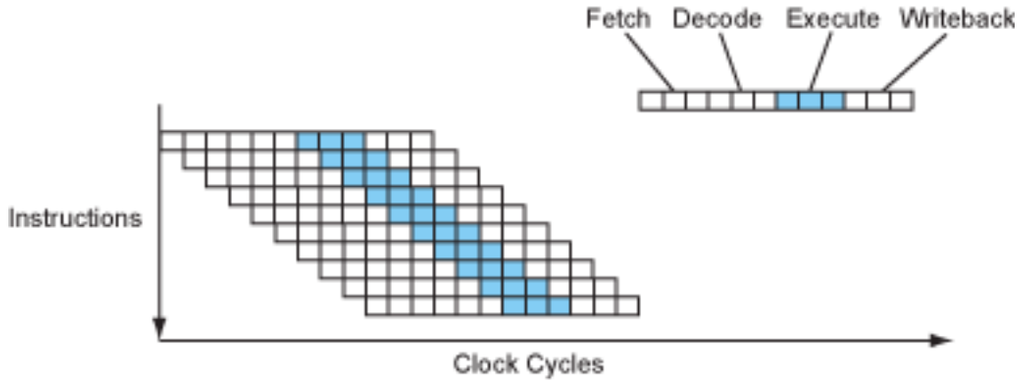

Figure 10: The instruction flow of a superpipelined processor

Since the clock speed is limited by the length of the longest, slowest stage in the pipeline, the logic gates that make up each stage can be subdivided, especially the longer ones, converting the pipeline into a deeper super-pipeline with a larger number of shorter stages. Then the whole processor can be run at a higher clock speed! Of course, each instruction will now take more cycles to complete (latency), but the processor will still be completing 1 instruction per cycle (throughput), and there will be more cycles per second, so the processor will complete more instructions per second

Pipelining and Instruction-Level Parallelism

Pipeline Depth	Processors
6	UltraSPARC T1
7	PowerPC G4e
8	UltraSPARC T2/T3, Cortex-A9
10	Athlon, Scorpion
11	Krait
12	Pentium Pro/II/III, Athlon 64/Phenom, Apple A6
13	Denver
14	UltraSPARC III/IV, Core 2, Apple A7/A8
14/19	Core i*2/i*3 Sandy/Ivy Bridge, Core i*4/i*5 Haswell/Broadwell
15	Cortex-A15/A57
16	PowerPC G5, Core i*1 Nehalem
18	Bulldozer/Piledriver, Steamroller
20	Pentium 4
31	Pentium 4E Prescott

Table 1: Pipeline depths of common processors.

Pipelining and Instruction-Level Parallelism

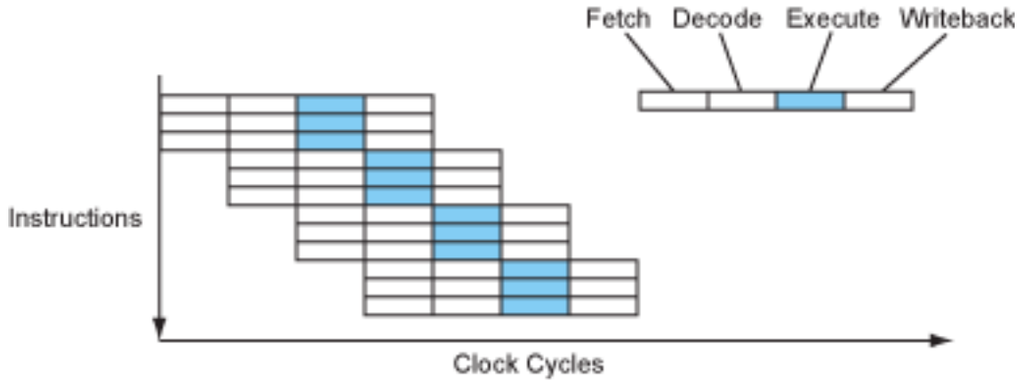

Figure 11: The instruction flow of a superscalar processor.

Since the execute stage of the pipeline consists of different functional units, each doing its own task, it seems tempting to try to execute multiple instructions in parallel, each in its own functional unit. To do this, the fetch and decode/dispatch stages must be enhanced so they can decode multiple instructions in parallel and send them out to the execution resources.

In the above example, the processor could potentially issue 3 different instructions per cycle for example 1 integer, 1 floating-point and 1 memory instruction.

Pipelining and Instruction-Level Parallelism

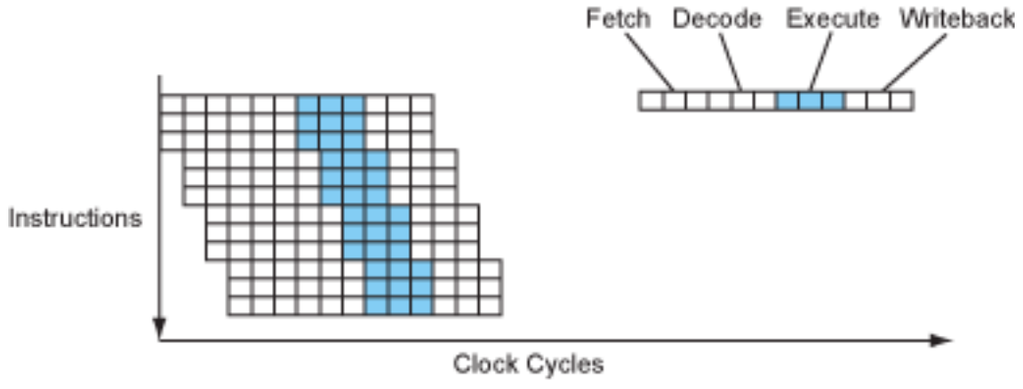

Figure 12: The instruction flow of a superpipelined-superscalar processor.

Of course, there's nothing stopping a processor from having both a deep pipeline and multiple instruction issue, so it can be both superpipelined and superscalar at the same time. Today, virtually every processor is a superpipelined-superscalar, so they're just called superscalar for short. Strictly speaking, superpipelining is just pipelining with a deeper pipe anyway.

Pipelining and Instruction-Level Parallelism

Issue Width	Processors
1	UltraSPARC T1
2	UltraSPARC T2/T3, Scorpion, Cortex-A9
3	Pentium Pro/II/III/M, Pentium 4, Krait, Apple A6, Cortex-A15/A57
4	UltraSPARC III/IV, PowerPC G4e
4/8	Bulldozer/Piledriver
5	PowerPC G5
6	Athlon, Athlon 64/Phenom, Apple A7/A8
6	Core 2, Core i*1 Nehalem, Core i*2/i*3 Sandy/Ivy Bridge
7	Denver
8	Core i*4/i*5 Haswell/Broadwell, Steamroller

Table 2: Issue widths of common processors.

Explicit Parallelism - VLIW

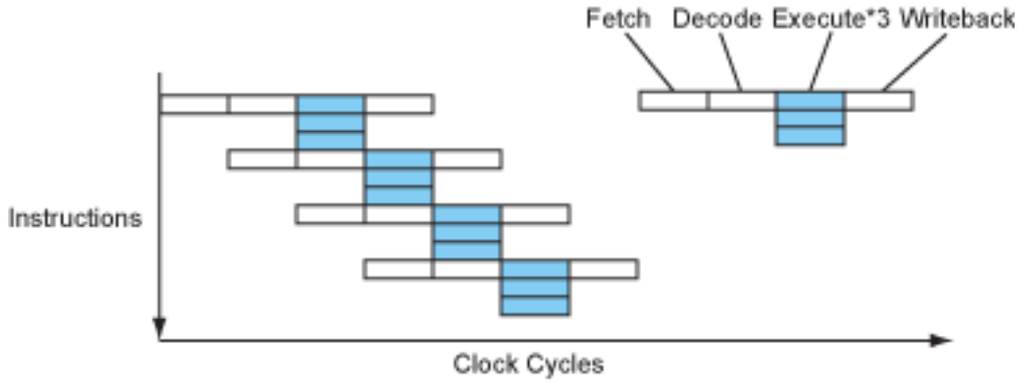

Figure 13: The instruction flow of a VLIW processor.

In this style of processor, the instructions are really groups of little sub-instructions, and thus the instructions themselves are very long, often 128 bits or more, hence the name VLIW – very long instruction word. Each instruction contains information for multiple parallel operations.

Some Graphics processors (GPUs) implement VLIW designs, as are many digital signal processors (DSPs).

Simultaneous Multithreading (SMT)

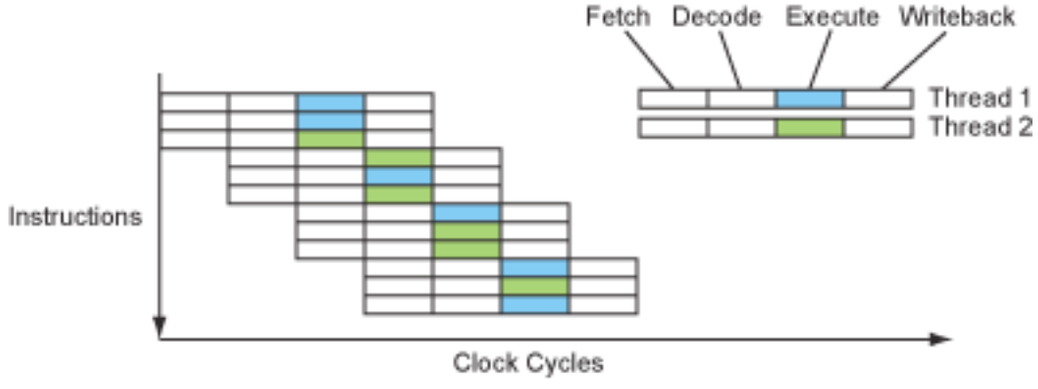

Figure 14: The instruction flow of an SMT processor.

If additional independent instructions aren't available within the program being executed, there is another potential source of independent instructions – other running programs (or other threads within the same program). Simultaneous multithreading (SMT) is a processor design technique which exploits exactly this type of thread-level parallelism.

Simultaneous multithreading is named hyper-threading for Intel CPUs.

Instruction Dependencies and Latencies

Consider the following two instructions:

```
a = b * c;
d = a + 1;
```

The second instruction depends on the first the processor can't execute the second instruction until after the first has completed calculating its result. This is a serious problem, because instructions that depend on each other cannot be executed in parallel. Thus, multiple issue is impossible in this case.

So, the processor will need to stall the execution of the second instruction until its data is available, inserting a *bubble* into the pipeline where no work gets done.