**Concurrent Architectures**

**Out of Order Superscalar:**

* Exploits very fine grain instruction level parallelism
* Hardware identifies and executes **independent instructions** in parallel
* Has few issues exploiting abundant parallelism (has to be obvious)
* Gets caught up with **data dependencies** stalls and waits
* Hardware can’t re-order code at run time
* Deals well with predictable branches
* Disambiguates memory access’ by examining addresses at execution time
* Code with no dependencies run beforehand

Advantage:

* No change in programmer behaviour
* Good for predictable branching code with **non-dependent data**

Disadvantage:

* Complexity, core is more power hungry with extra hardware
* Limited by code implementation

**Very Long Instruction Word (VLIW):**

* Also executes fine-grain instruction level parallelism
* Uses a **complex compiler** vs actual hardware
* Compiler restructures code at compile time to remove dependency
* The compiler identifies parallelism in the code and re-orders code
* Instructions are always fetched, decoded and executed in pairs (VLIW)

Advantage:

* Simpler hardware (potentially less power hungry)
* Potentially more scalable

Disadvantage:

* Compiler complexity (longer compile time).
* Needs lots of **program memory bandwidth** due to large instruction size
* Code bloat

**Vector Processing:**

* Uses SSE parallelism techniques.
* Operates on an array/vector of data rather than individual data.
* Handles large arrays of data well.
* Cannot work with all code, has to be suited to it.
* Loop size must be divisible by 4.
* Array access must not display data hazards.
* Requires programmer to implement a suitable code structure.

Advantage:

* Single core execution, speeds up array handling a lot.
* Excellent for image processing, sound processing etc.

Disadvantage:

* Requires program to implement suitable code structure.
* Still usually affected by data dependencies and hazards.

**Multi-Threaded Architecture:**

* Runs multiple threads and can switch between these rapidly.
* When a thread stalls (cache miss) another thread takes over.
* **Fine-Grain:** Switches threads over after every clock cycle.
* **Coarse-Grain:** Switches threads only after particularly long stalls.
* **Simultaneous Multi-Threading:** Mixes multiple threads together during execution.
* Relies on the programmer/compiler to divide code into threads (e.g OpenMP)
* Single core with larger register set can best use ALU.
* Loads/stalls are highly utilized as another thread takes over execution.
* If the array is large, pipeline stalls due to cache misses can be covered by having more threads running,

Advantage:

* Handles stalls and loads excellently as other threads take over execution.

Disadvantage:

* Programmer must divide the code into threads.

**Shared-Memory Multi-Core Processor:**

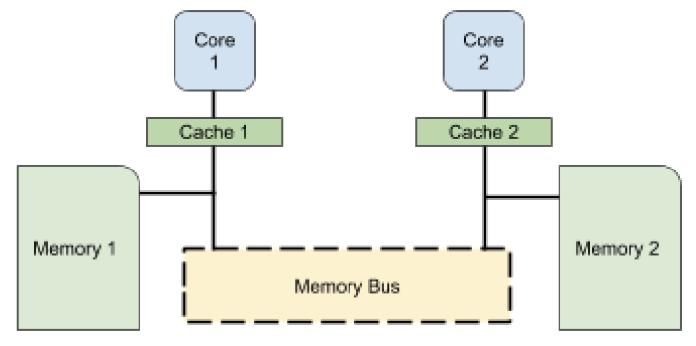
* Also operates using threads.
* Instructions carried out across multiple cores, some of which will be specialized.
* Has a shared-memory between all of the cores.
* Trade-off between communicating thread cost and parallel efficiency gains.
* Introduces race conditions and system must implement a cache coherency protocol.
* Dark Silicon is associated with multi-core architectures, big caches.

**Multi-Chip Symmetric Multi-Processor:**

* Similar to shared-memory multi-core.
* **Key difference: Individual memory buses.**
* Same important trade-off however particularly dependent threads.
* Due to more cache, **more memory bandwidth** to fetch array elements.
* Perfect for extremely large arrays.
* The hardware to manage caches and memory are consistent.
* Communication across threads is much more expensive but more memory bandwidth.
* **If threads don’t need to communicate this is the ideal preferred over above.**

**Non-Uniform Memory Access:**

* Single shared memory address space, **physical memory split into regions**.
* Regions range from far away to close (logically).
* Introduces non-uniform (not constant) memory access times.
* Closer regions incredibly cheap, farther ones more expensive (non-uniform).
* Computation much more expensive vs data movement.

****

**Distributed Memory Multi-Computer:**

* Every core has access to its own local, private version of memory.
* Introduces overheads such as a cache coherency protocol.
* Onus is on the programmer to think of data distribution.
* Secure way of implementing parallel programs.
* Master-slave relationship