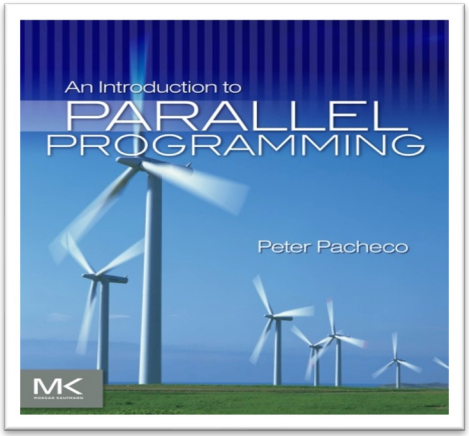
14013204-3 - PARALLEL COMPUTING



Chapter 2

Parallel Hardware

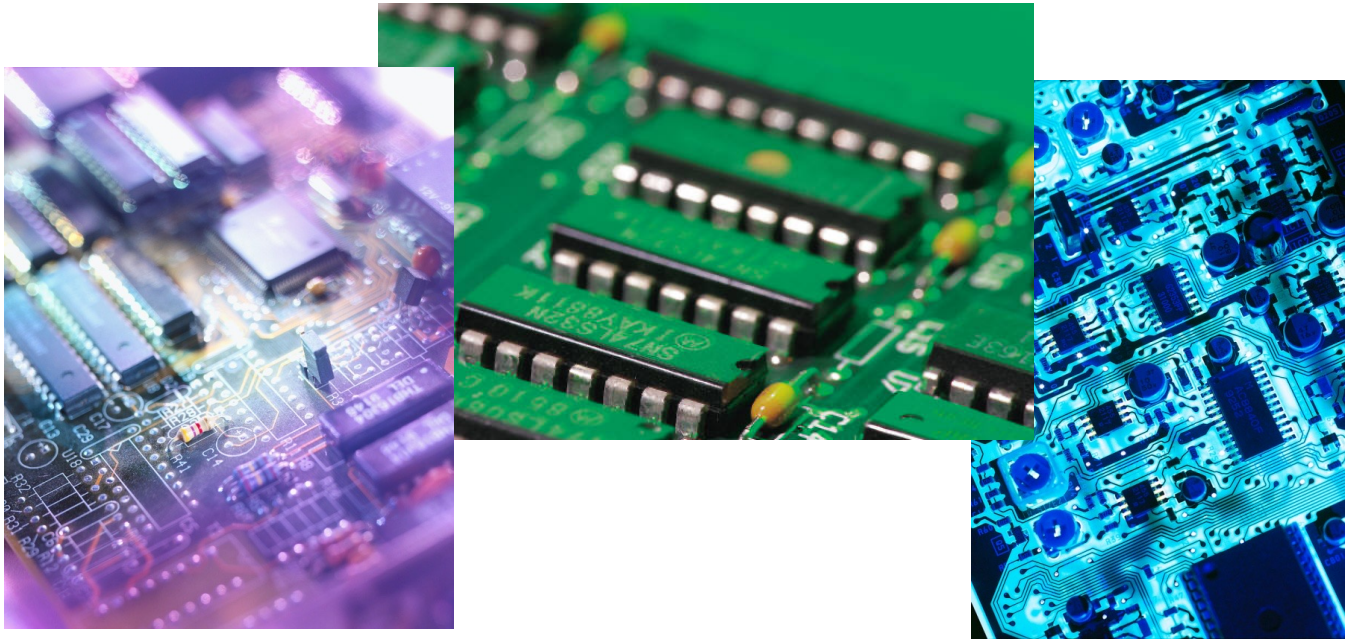
Roadmap

- 2.1 Some background
  - 2.1.1 The von Neumann architecture
  - 2.1.2 Processes, multitasking, and threads
- 2.2 Modifications to the von Neumann model
  - 2.2.1 The basics of caching
  - 2.2.2 Cache mappings
  - 2.2.3 Caches and programs: an example
  - 2.2.4 Virtual memory
  - 2.2.5 Instruction-level parallelism
  - 2.2.6 Hardware multithreading
- 2.3 Parallel hardware
  - 2.3.1 Classifications of parallel computers
  - 2.3.2 SIMD systems
  - 2.3.3 MIMD systems

2.1 Some Background

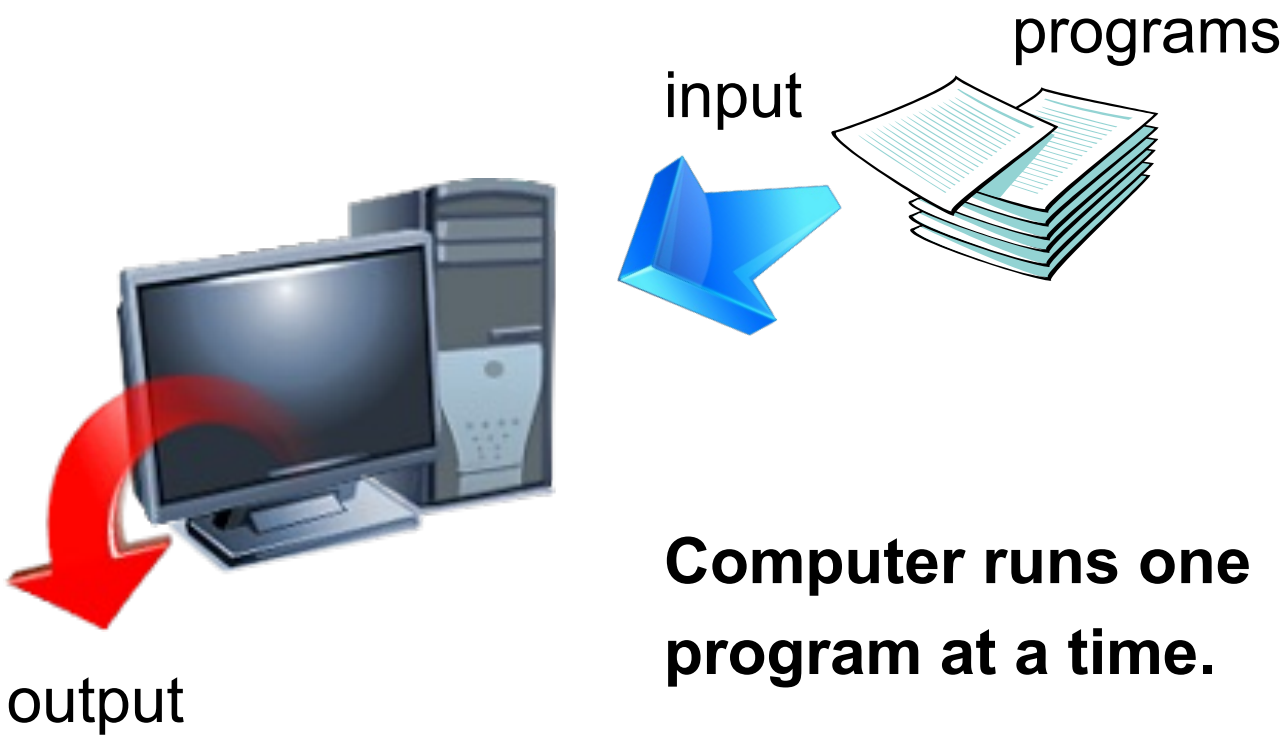
The von Neumann architecture

SOME BACKGROUND



- Parallel hardware and software have grown out of conventional serial hardware and software: hardware and software that runs (more or less) a single job at a time.
- So to better understand the current state of parallel systems, let's begin with a brief look at a few aspects of serial systems.

Serial hardware and software



Computer runs one program at a time.

The von Neumann Architecture

- For over 40 years, virtually all computers have followed a **common machine model** known as the **von Neumann computer**. Named after the Hungarian mathematician John von Neumann.
- The "classical" von Neumann architecture consists of **main memory**, a **central processing unit (CPU)** or processor or core, and an **interconnection between the memory and the CPU**.
- A von Neumann computer uses the **stored-program** concept. The CPU executes a stored program that specifies a sequence of **read and write** operations on the memory.
- A von Neumann **machine executes a single instruction at a time, and each instruction operates on only a few pieces of data.**

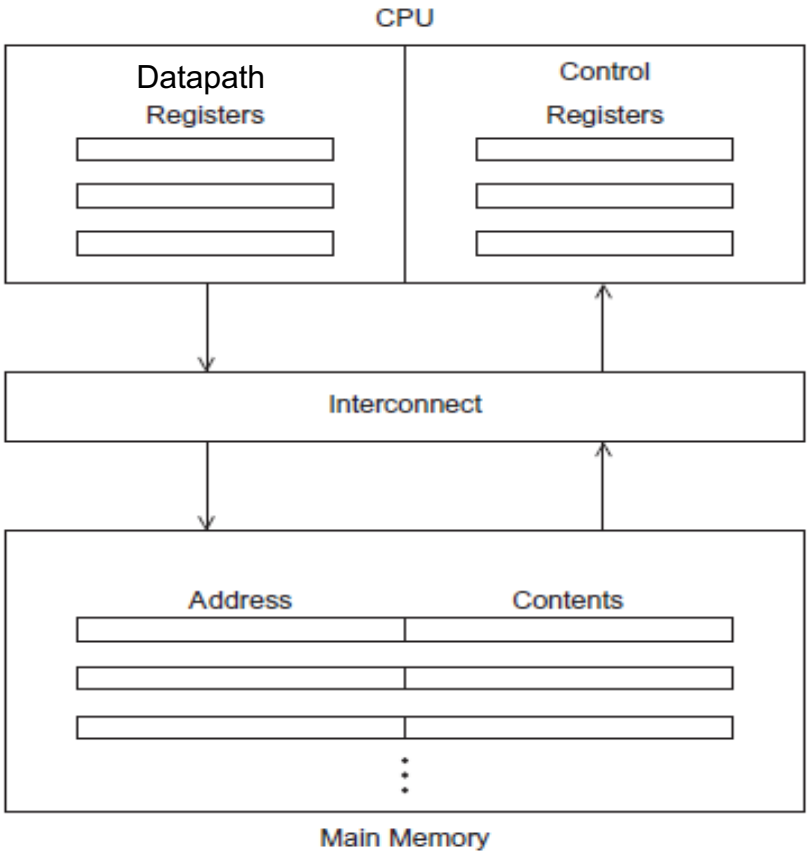
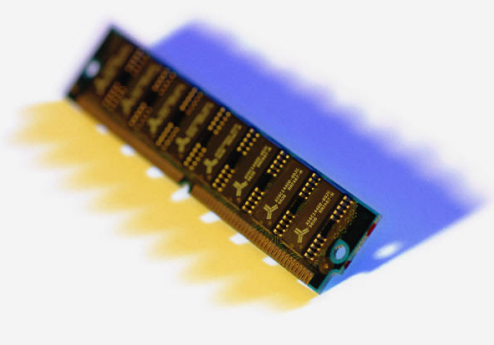


Figure 2.1

Main memory

- This is a collection of locations, each of which is capable of **storing both instructions and data**.

- Every location consists of an **address**, which is used to access the location, and the contents of the location.



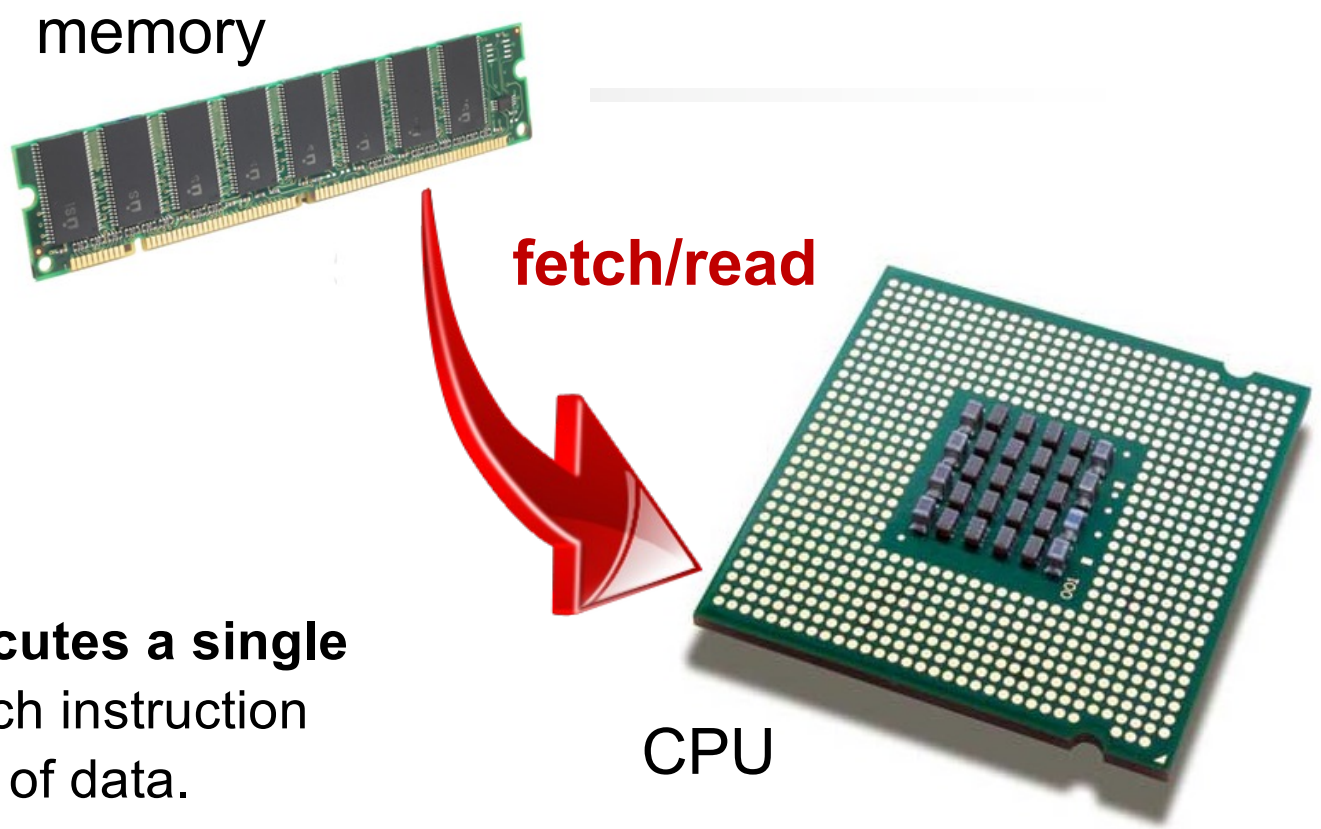
Central processing unit (CPU)

- Divided into two parts.

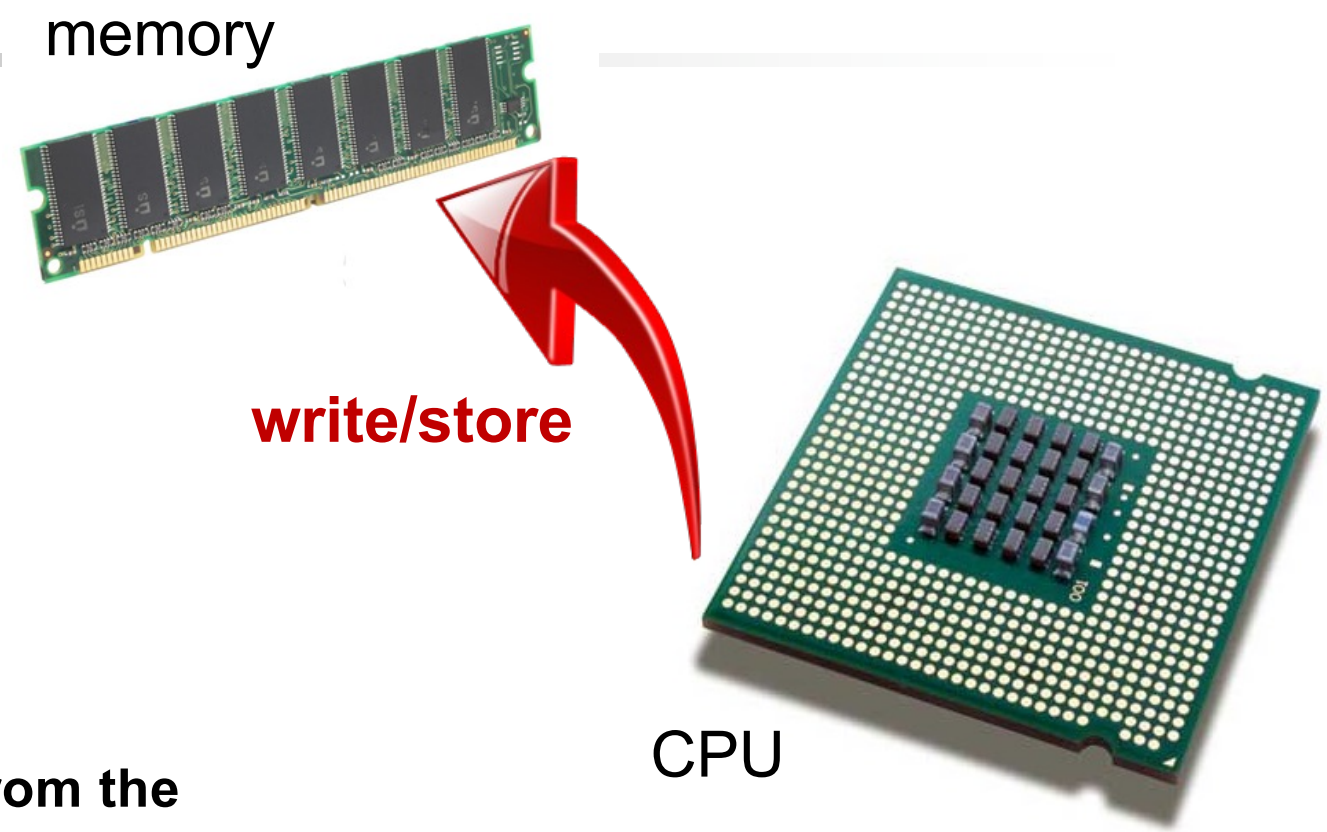
1. **Control unit** - responsible for deciding which instruction in a program should be executed. (*the boss*)
2. **Datapath** - responsible for executing the actual instructions. (*the worker*)
   - Combination of **different functional units** including ALU, registers, and internal buses that work together to process the instructions and manage the data.
   - The ALU is a critical component of the datapath responsible for performing **arithmetic and logical operations** on data. It executes calculations based on the instructions provided by the CPU's control unit.

Key terms

- **Register** – very fast storage, part of the CPU.
  - Store Data in the CPU and information about the state of an executing program.
- **Program counter** – special register in the control unit - stores the address of the next instruction to be executed.
- **Bus** – wires, and hardware that connect the CPU and memory.
  - **Instructions and data** are transferred between the CPU and memory via the interconnect.
  - This has traditionally been a bus, which consists of a collection of parallel wires and some hardware controlling access to the wires. More recent systems use more complex interconnects.



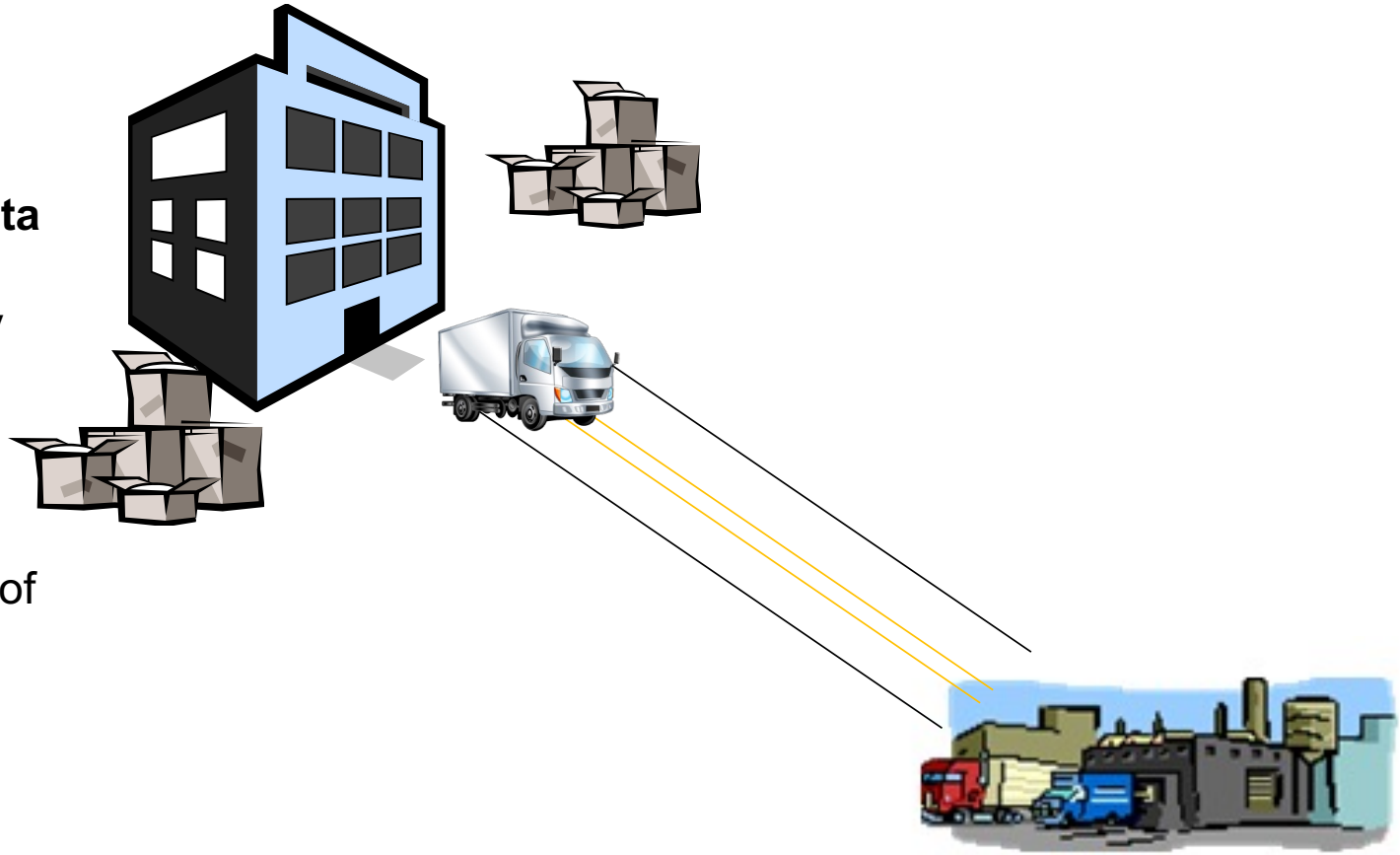
- A von Neumann machine **executes a single instruction at a time**, and each instruction operates on only a few pieces of data.
- **When data or instructions are transferred from memory to the CPU**, we sometimes say the data or instructions are **fetched** or **read** from memory.



- **When data are transferred from the CPU to memory**, we sometimes say the data are **written to memory** or **stored**.

von Neumann bottleneck

- The separation of memory and CPU is often called the **von Neumann bottleneck** since **the interconnect determines the rate at which instructions and data can be accessed.**
- The potentially vast quantity of data and instructions needed to run a program is effectively isolated from the CPU.
- In 2021, CPUs are capable of executing instructions **more than one hundred times faster** than they can fetch items from main memory.



2.1 Some Background

Processes, multitasking, and threads

An operating system “process”

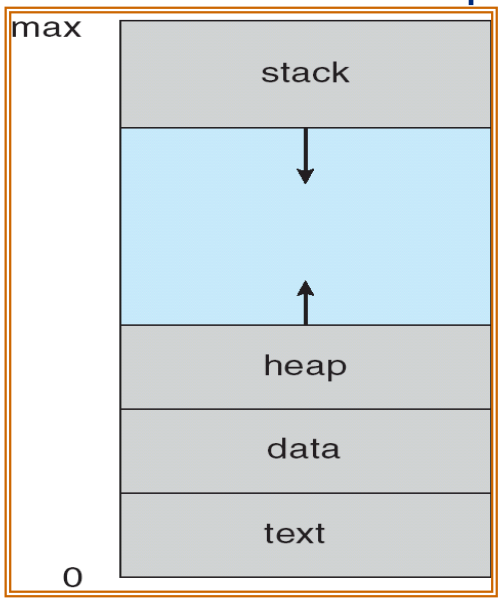
- Recall that the **operating system (OS)** is a major piece of software, whose purpose is to **manage hardware and software resources** on a computer.
- An OS determines which programs can run and when they can run.
- It also controls the **allocation of memory** to running programs and **access to peripheral devices** such as hard disks and network interface cards.
- When a user runs a program, the operating system creates a **process**.
- A **process** is an instance of a computer program that is being executed.

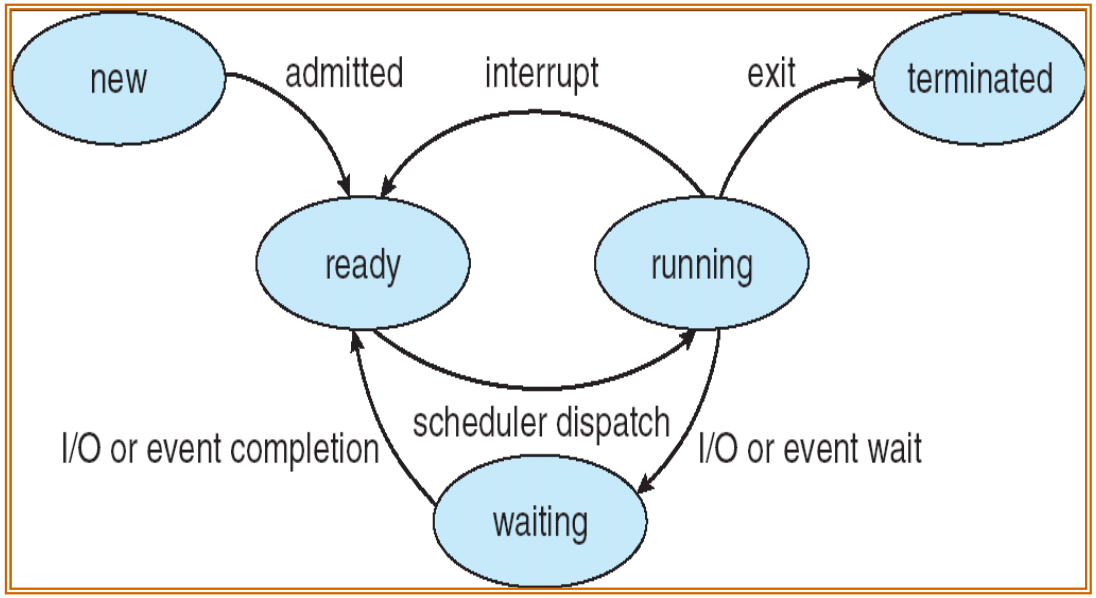
An operating system “process”

- Components of a process:
  - The executable machine language program.
  - A block of memory: which will include the **executable code**, a **call stack** that keeps track of active functions, a **heap** that can be used for memory explicitly allocated by the user program, and some other memory locations.
  - Descriptors of resources the OS has allocated to the process e.g. file descriptors.
  - Security information e.g. information specifying which hardware and software resources the process can access.
  - Information about the state of the process - such as whether the process is ready to run or is waiting on some resource, the content of the registers, and information about the process’s memory.

An operating system “process”

- As a process executes, it changes its ***state***
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a CPU
  - **terminated**: The process has finished execution





Multitasking

- Most modern operating systems are **multitasking**.
- Gives the illusion that a single processor system is running multiple programs simultaneously.
- Each process takes turns running. (**time slice**)
- After its time is up, it waits until it has a turn again (**blocks**) and the operating system can run a different program.
- A multitasking OS may **change the running process many times in a minute**, even though changing the running process can take a long time.
- In a multitasking OS, if a process needs to wait for a resource—for example, it needs to read data from external storage—it will **block**. This means that it will stop executing and the operating system can run another process.
  - However, many programs can continue to do useful work even though the part of the program that is currently executing must wait on a resource. For example, an airline reservation system that is blocked waiting for a seat map for one user could provide a list of available flights to another user.

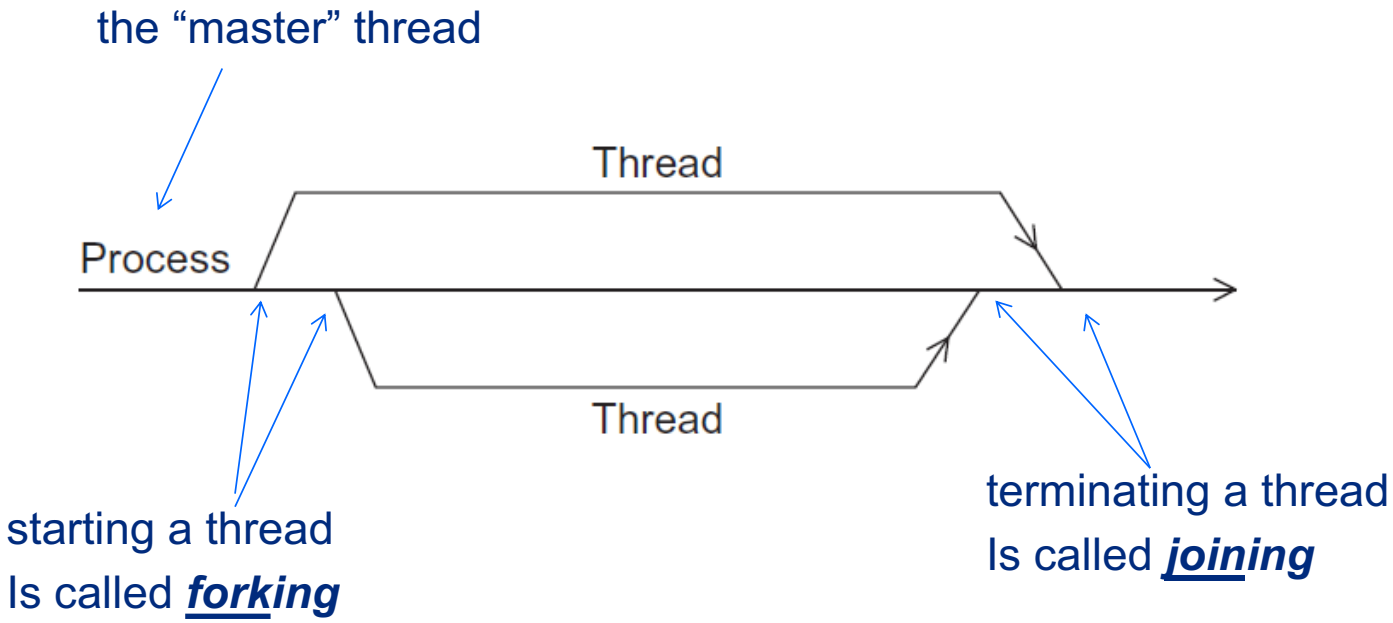
Threading

- **Threading** provides a mechanism for programmers to **divide their programs into more or less independent tasks** with the property that when one thread is blocked another thread can be run.
- Furthermore, in most systems it's possible to switch between threads much **faster** than it's possible to switch between processes.
- This is because threads are "**lighter weight**" than processes.
- **Threads** are contained within processes, so they can **use the same executable**, and they usually **share the same memory and the same I/O devices.**

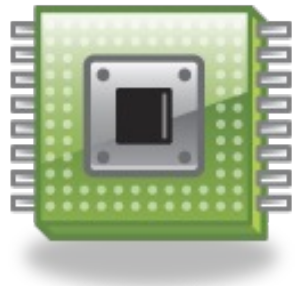
Threading

- In fact, two threads belonging to one process can share most of the process' resources.
- The two most important exceptions are:
  1. They'll need a record of **their own program counters**
  2. And they'll need **their own call stacks** so that they can execute independently of each other.

A process and two threads



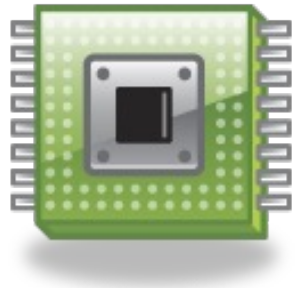
2.2 Modifications To The Von Neumann Model



- As we noted earlier since the first electronic digital computers were developed back in the 1940s, computer scientists and computer engineers have made **many improvements to the basic von Neumann architecture.**
- Many are targeted at **reducing the problem of the von Neumann bottleneck**, but many are also targeted at simply **making CPUs faster**.
- We will look at three of these improvements:
  1. **caching**
  2. **virtual memory**
  3. **low-level parallelism.**

Modifications To The Von Neumann Model

1. caching



Basics of caching

- A collection of memory locations that can be accessed in less time than some other memory locations.
- A **CPU cache** is a collection of memory locations that the CPU can access **more quickly** than it can access main memory.
- A CPU cache can either be located on the same chip as the CPU or it can be located on a separate chip that can be accessed much faster than an ordinary memory chip.

Basics of Caching – Related Issues

1. **Block Placement:** Where to Place the Main Memory Block in the Cache?
2. **Block Identification:** How to Find the Main Memory Block in the Cache?
3. **Block Replacement:** During the Cache Miss, how to choose which entry to replace from the cache?
4. **Write Strategy:** How are the updates propagated? – *will be covered in the next lecture with other issues related to cache coherency and false sharing*

- You have already studied the first three points before. Watch these videos to review:
  https://www.youtube.com/watch?v=1tvW8kzOpSA&list=PLBInK6fEyqRgLLlzdgiTUKULKJPYc0A4q&index=25
- https://www.youtube.com/watch?v=7IxAfszjy68&list=PLBInK6fEyqRgLLlzdgiTUKULKJPYc0A4q&index=26
- https://www.youtube.com/watch?v=_Hh-NcdbHCY&list=PLBInK6fEyqRgLLlzdgiTUKULKJPYc0A4q&index=27
- https://www.youtube.com/watch?v=QyQf9KvkQA4&list=PLBInK6fEyqRgLLlzdgiTUKULKJPYc0A4q&index=28

Caches and programs

```
double A[MAX][MAX], x[MAX], y[MAX];
. . .
/* Initialize A and x, assign y = 0 */
. . .
/* First pair of loops */
for (i = 0; i < MAX; i++)
    for (j = 0; j < MAX; j++)
        y[i] += A[i][j]*x[j];
. . .
/* Assign y = 0 */
. . .
/* Second pair of loops */
for (j = 0; j < MAX; j++)
    for (i = 0; i < MAX; i++)
        y[i] += A[i][j]*x[j];
```

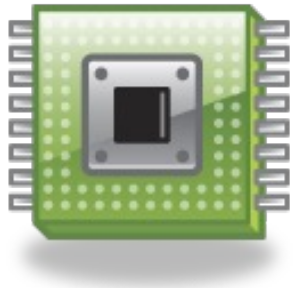
A total of 4 misses

Cache Line	Elements of A			
0	A[0][0]	A[0][1]	A[0][2]	A[0][3]
1	A[1][0]	A[1][1]	A[1][2]	A[1][3]
2	A[2][0]	A[2][1]	A[2][2]	A[2][3]
3	A[3][0]	A[3][1]	A[3][2]	A[3][3]

A total of 16 misses

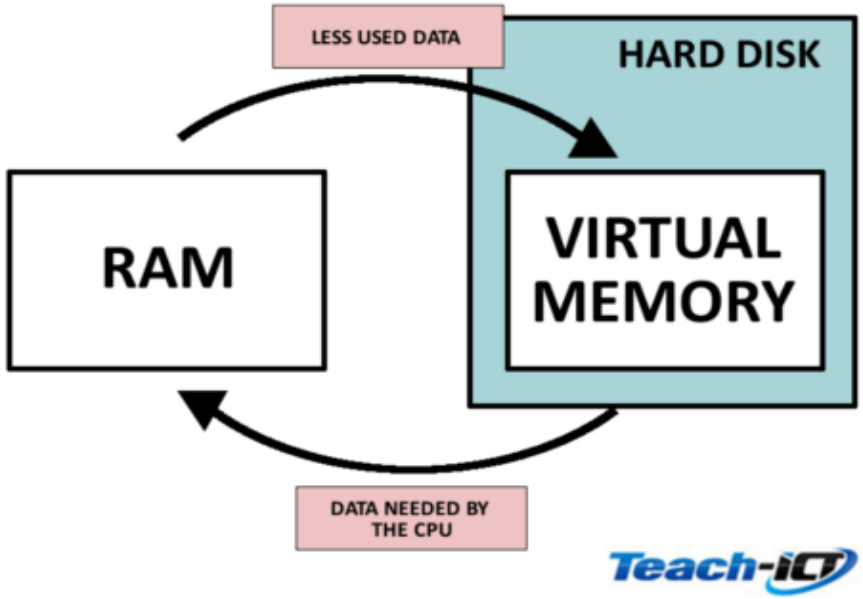
Modifications To The Von Neumann Model

2. virtual memory

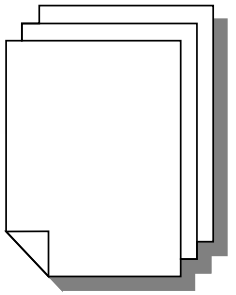


Virtual memory

- If we run a very large program or a program that accesses very large data sets, all of the instructions and data **may not fit into main memory**.
- **Virtual memory** functions as a cache for secondary storage.



Virtual memory

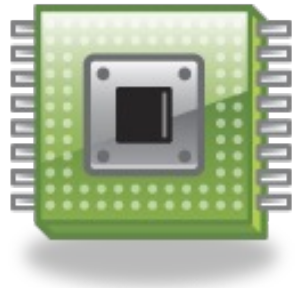


- It exploits the principle of **spatial and temporal locality**.
- Keeping in **main memory** only the **active parts** of the many running programs; those parts that are idle are kept in a block of secondary storage called **swap space**.
- Like **CPU caches**, virtual memory operates **on blocks of data and instructions**. These blocks are commonly called **pages**, and since secondary storage access can be hundreds of thousands of times slower than main memory access, pages are relatively large (page size ranges from 4 to 16 KB).

Virtual memory

- The relative slowness of disk accesses has a couple of additional consequences for virtual memory.
  - We as programmers don't directly control virtual memory.
  - Unlike CPU caches, which are handled by system hardware, virtual memory is usually controlled by a combination of system hardware and operating system software.
- You should have studied the virtual memory management details in OS course. To review: Read the book (pages: 25- 27)

Modifications To The Von Neumann Model



3. low-level parallelism.

Instruction Level Parallelism (ILP) (1)

- **Instruction-level parallelism (ILP)** attempts to improve processor performance by having multiple processor components or **functional units** simultaneously executing instructions in parallel within a single program or thread.
- It **exploits parallelism** at **the instruction level**.
- There are two approaches to instruction-level parallelism:
  1. **Hardware**: hardware level works upon dynamic parallelism.
     - Dynamic parallelism means the **processor** decides at run time which instructions to execute in parallel.
  2. **Software**: software level works on static parallelism.
     - Static parallelism means the **compiler** decides which instructions to execute in parallel.

Instruction Level Parallelism (ILP) (2)

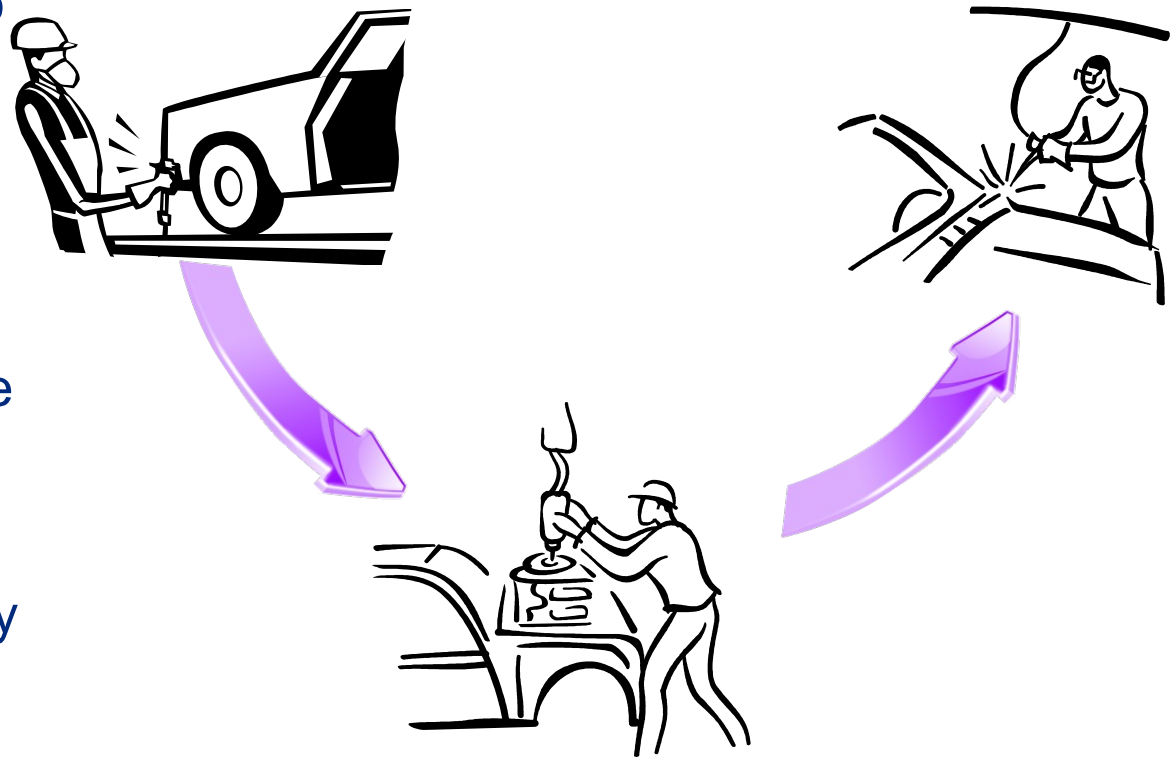
- **There are two main approaches to ILP** on hardware level :
  1. **Pipelining** - functional units are arranged in stages.
  2. **Multiple issue** - multiple instructions can be simultaneously initiated.

- Both approaches are used in virtually all modern CPUs.

Pipelining

The principle of pipelining is similar to a **factory assembly line**:

1. while one team is bolting a car's engine to the chassis.
2. another team can connect the transmission to the engine and the driveshaft of a car that's already been processed by the first team.
3. and a third team can bolt the body to the chassis in a car that's been processed by the first two teams.



Pipelining example (1)

- suppose we want to **add the floating-point numbers 9.87×10^4 and 6.54×10^3.** Then we can use the following steps:

Time	Operation	Operand 1	Operand 2	Result
1	Fetch operands	9.87×10^4	6.54×10^3	
2	Compare exponents	9.87×10^4	6.54×10^3	
3	Shift one operand	9.87×10^4	0.654×10^4	
4	Add	9.87×10^4	0.654×10^4	10.524×10^4
5	Normalize result	9.87×10^4	0.654×10^4	1.0524×10^5
6	Round result	9.87×10^4	0.654×10^4	1.05×10^5
7	Store result	9.87×10^4	0.654×10^4	1.05×10^5

- Here we're using base 10 and a three-digit mantissa or significand with one digit to the left of the decimal point.
- **Normalizing** shifts the decimal point one unit to the left, and **rounding** rounds to three digits.
- If each of the operations takes **one nanosecond** (10^{-9} seconds), the addition operation will take **seven nanoseconds.**

Pipelining example (2)

```
float x[1000], y[1000], z[1000];
. . .
for (i = 0; i < 1000; i++)
    z[i] = x[i] + y[i];
```

- This for loop takes about **7000 nanoseconds.**

Pipelining example (3)

- As an alternative, suppose we **Divide** the floating-point adder into 7 separate pieces of **hardware or functional units**.
  - First unit fetches two operands, the second unit compares exponents, etc.
  - Output of one functional unit is input to the next.
- Then a single floating-point addition will still take 7 nanoseconds.
- However, when we execute the for loop, we can **fetch x[1]** and **y[1]** while we're **comparing the exponents of x[0] and y[0].**

Pipelining example (4)

- More generally, it's possible for us to **simultaneously** execute **seven different stages in seven different additions**

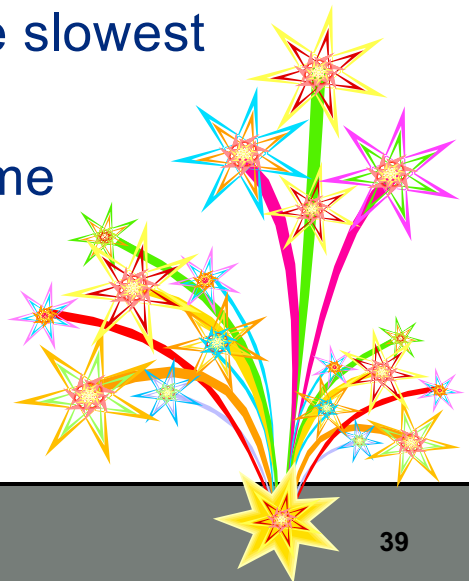
Time	Fetch	Compare	Shift	Add	Normalize	Round	Store
0	0						
1	1	0					
2	2	1	0				
3	3	2	1	0			
4	4	3	2	1	0		
5	5	4	3	2	1	0	
6	6	5	4	3	2	1	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
999	999	998	997	996	995	994	993
1000		999	998	997	996	995	994
1001			999	998	997	996	995
1002				999	998	997	996
1003					999	998	997
1004						999	998
1005							999

Table 2.3: Pipelined Addition.

Numbers in the table are subscripts of operands/results.

Pipelining example (5)

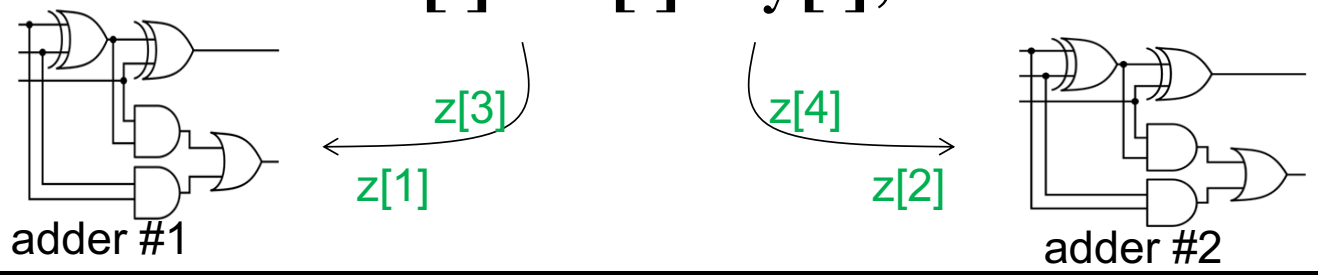
- One floating point addition still takes **7 nanoseconds,** But 1000 floating point additions now takes **1006 nanoseconds!**
- In general, **a pipeline with k stages won't get a k-fold** improvement in performance.
  - For example, if the times required by the various functional units are different, then the stages will effectively run at the speed of the slowest functional unit.
  - Furthermore, delays—such as waiting for an operand to become available—can cause the pipeline to stall.



Multiple Issue (1)

- Pipelines improve performance by **taking individual pieces of hardware or functional units and connecting them in sequence**.
- Multiple issue processors **replicate functional units** and try to **simultaneously execute different instructions** in a program.
  - if we have **two complete floating point adders**, we can approximately **halve the time** it takes to execute the loop.
  - While the first adder is computing z[0], the second can compute z[1]; while the first is computing z[2], the second can compute z[3]; and so on.

```
for (i = 0; i < 1000; i++)
    z[i] = x[i] + y[i];
```



Multiple Issue (2)

- If the functional units are scheduled at compile time, the multiple issue system is said to use **static multiple issue**.
- If they're scheduled at run-time, the system is said to use **dynamic multiple issue**.
- A processor that supports dynamic multiple issue is sometimes said to be **superscalar**.
- Superscalar architecture is a method of designing processors to **allow multiple instructions to be issued and executed by multiple execution units in a single clock cycle**.
- It's based on the concept of instruction-level parallelism (ILP).
- In a superscalar processor, there are **multiple execution units** such as **arithmetic logic units (ALUs), floating-point units (FPUs),** and more. These units can work concurrently, executing different instructions from a single instruction stream.

Speculation (1)

- In order to make use of multiple issue, the system **must find instructions that can be executed simultaneously.**
- One of the most important techniques is **speculation**.

  - In speculation, the compiler or the processor makes a guess about an instruction and then executes the instruction on the basis of the guess.

Speculation (2)

```
z = x + y;
if (z > 0)
    w = x;
else
    w = y;
```

If the system speculates incorrectly, it must go back and recalculate w = y.

```
z = x + y;
w = *a_p;
```

If the system speculates incorrectly, it must go back and re-execute w = *a_p.

Speculation (3)

- If the **compiler** does the speculation, it will usually **insert code that tests whether the speculation was correct**, and, if not**, takes corrective action.**
- If the **hardware** does the speculation, the processor usually **stores the result(s) of the speculative execution in a buffer**.
  - When it's known that the **speculation was correct**, the **contents of the buffer are transferred to registers or memory**.
  - If the speculation was **incorrect**, the **contents of the buffer are discarded**, and the **instruction is re-executed**.

Thread Level Parallelism (TLP)

- ILP can be very difficult to exploit: a program with a **long sequence of dependent statements** offers few opportunities.
- For example, in a direct calculation of the Fibonacci numbers

```
f [ 0 ] = f [ 1 ] = 1;
for ( i = 2; i <= n ; i ++)
    f [ i] = f [ i−1] + f [ i−2];
```

- There's essentially **no opportunity** for simultaneous execution of instructions.

Thread Level Parallelism (TLP)

- **Thread-level parallelism (TLP) :** attempts to provide parallelism through the simultaneous execution of **different threads**, so it provides **coarser-grained parallelism** than **ILP**,
- That is, the program units that are being simultaneously executed—threads—are **larger or coarser** than the **finer-grained** units—individual instructions.

Hardware multithreading (1)

- **Hardware multithreading** provides a means for systems to continue doing useful work when the task being currently executed has stalled.
  - Ex., the current task has to wait for data to be loaded from memory
- Instead of looking for parallelism in the currently executing thread, it may make sense to simply run another thread.
- Of course, for this to be useful, the system must support very rapid switching between threads.

Hardware multithreading (2)

- **Fine-grained multithreading** - the processor switches between threads after each instruction, skipping threads that are stalled (*experienced a delay or pause in execution*).
  - Pros: potential to avoid wasted machine time due to stalls.
  - Cons: a thread that's ready to execute a long sequence of instructions may have to wait to execute every instruction.

Hardware multithreading (3)

- **Coarse-grained multithreading** - only switches threads that are stalled waiting for a time-consuming operation to complete (e.g., a load from main memory).
  - Pros: Switching threads doesn't need to be nearly instantaneous.
  - Cons: The processor can be idled on shorter stalls, and thread switching will also cause delays.

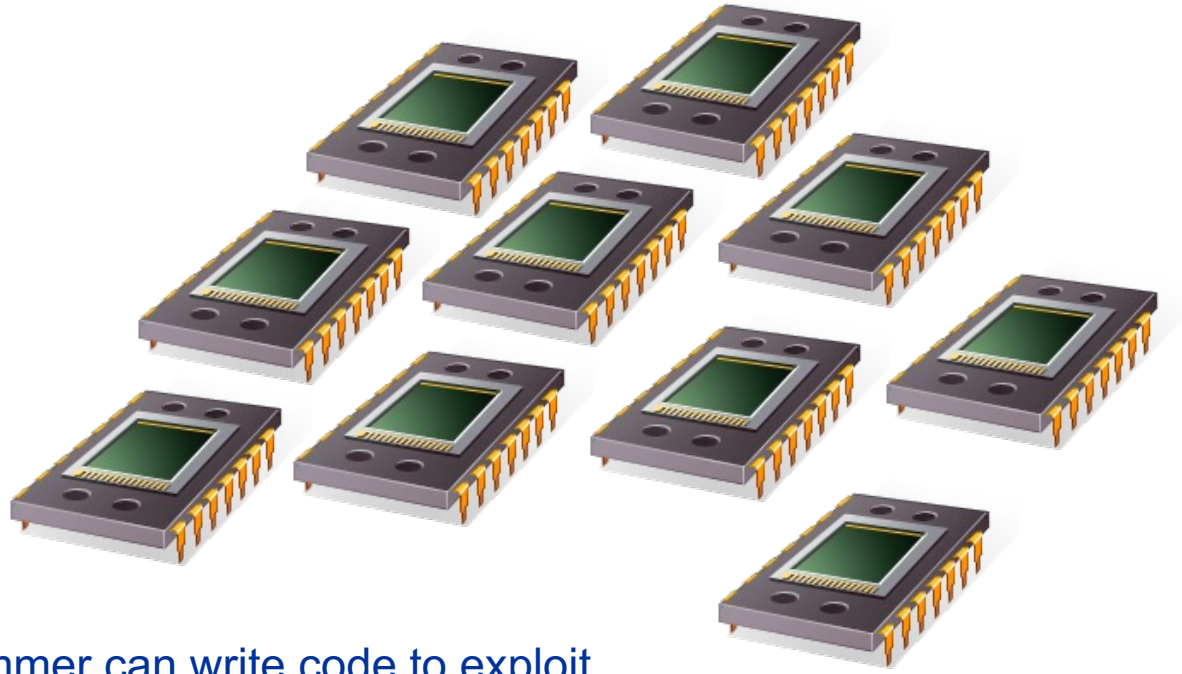
Hardware multithreading (4)

- **Simultaneous multithreading** (SMT) - a variation on fine-grained multithreading.

- It attempts to exploit superscalar processors by allowing multiple threads to make use of the multiple functional units.

Hardware visible to programmer

- **Multiple issue, Pipelining** can clearly be considered to be **parallel hardware**, since functional units can be executed simultaneously
- This form of **parallelism** is **not** usually **visible** to the programmer.
  - We're treating them as extensions to the basic **von Neumann model**
- we will talk about **parallel hardware** that is **visible** to the programmer (must modify the source code to exploit it) ➔ **Flynn's taxonomy** .



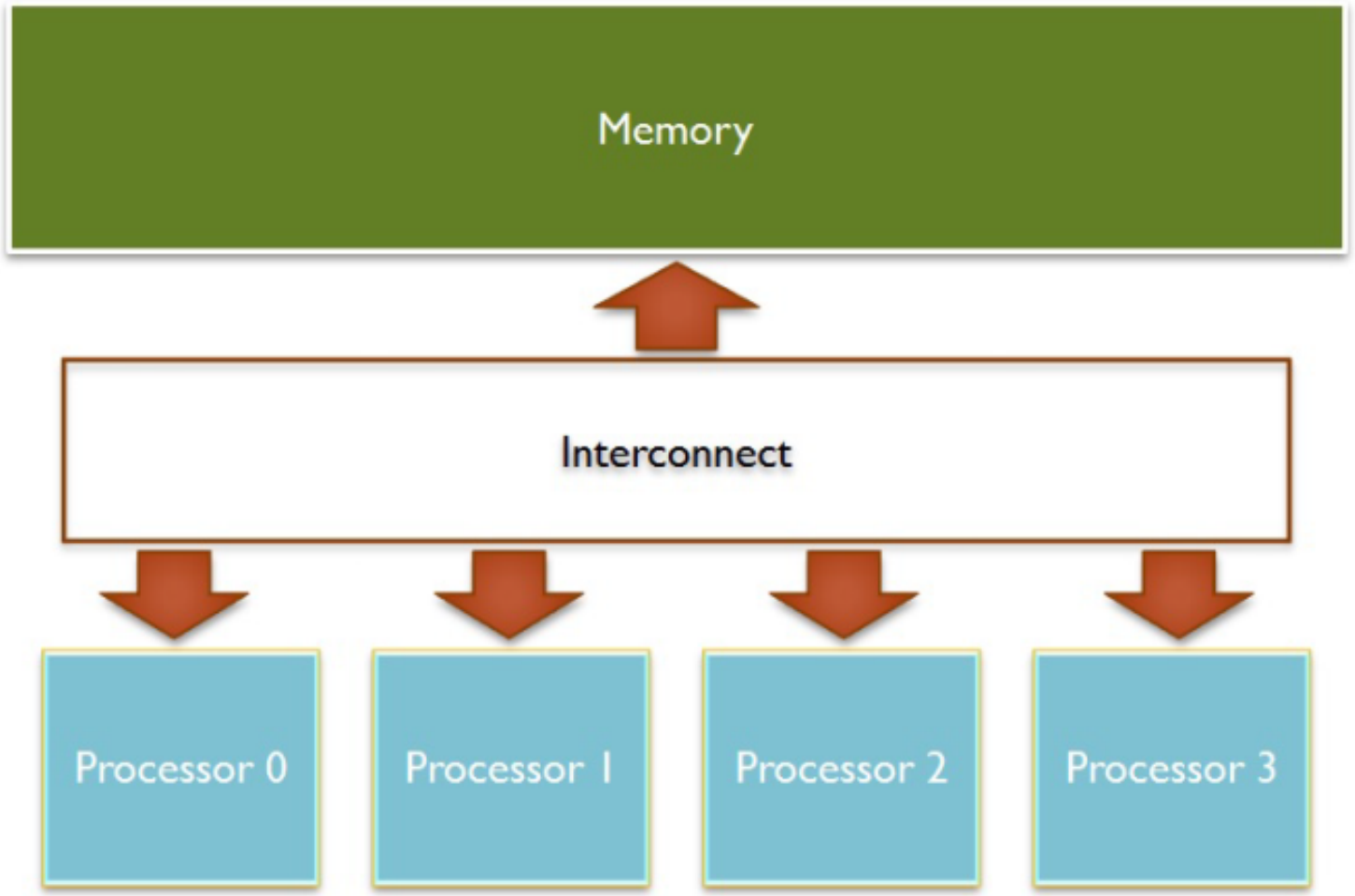
A programmer can write code to exploit.

2.3 Parallel hardware

Basic Parallel Architecture

- Still consist mainly of the same **three** main building blocks:
  - **The memory modules:** to store programs and their data.
  - **The interconnection network:** to communicate with other processors, and to communicate with memory.
    - **Fetch/Read**
    - **Write/Store**
  - **The processors (the cores):** **more than one core** to execute the parallel program.

Basic Parallel Architecture



Basic Parallel Architecture

- Over recent decades, there has been a gradual development in the degree of sophistication of each of these building blocks.
- What makes one parallel computer different from another is **the manner in which they are arranged.**
- Parallel computing cores themselves are essentially similar to those used in single-core systems with new technologies increasingly embedded with the cores for parallel architectures
  - The aim is to add more improvements or new functionalities e.g., GPUs, and FPGAs.

Classifications of parallel computers

- **Parallel computers** have been classified in different ways.
- The earliest and the most widely used classification is Flynn's Taxonomy, 1967 (**Flynn, Michael. (1967). Very High-Speed Computing Systems. Proceedings of the IEEE. 54. 1901 - 1909. 10.1109/PROC.1966.5273.**)

- Flynn's taxonomy distinguishes multi-processor computer architectures according to how they can be classified along the two independent dimensions of ***Instruction*** and ***Data***.
- Each of these dimensions can have only one of two possible states: ***Single*** or ***Multiple***.

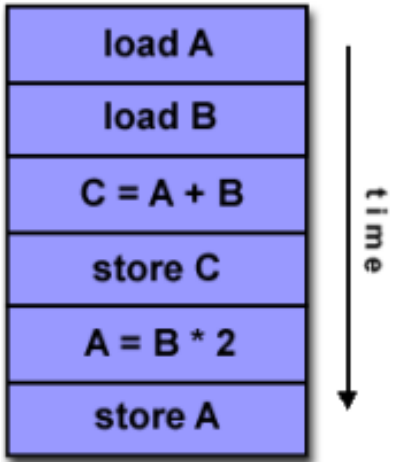
Flynn's Taxonomy

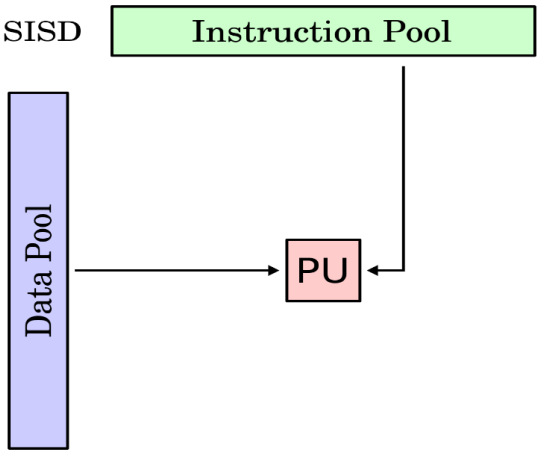
- Flynn classified computing architectures into four categories, based on the number of **instruction streams** and **data streams** they can simultaneously manage:

classic von Neumann **(SISD)** **Single instruction stream** **Single data stream**	**(SIMD)** **Single instruction stream** **Multiple data stream**
(MISD) **Multiple instruction stream** **Single data stream** not covered	**(MIMD)** **Multiple instruction stream** **Multiple data stream**

SISD

- It is a **sequential** computer (Not parallel) with a single processor ➔ **classical von Neumann system**.
- **Single instruction**: Only one instruction stream is being acted on by the CPU during any **one clock cycle.**
- **Single data**: Only one data stream is being used as **input** during any one clock cycle.
- Deterministic execution.
- This is the oldest and until recently, the most **prevalent** form of computer.

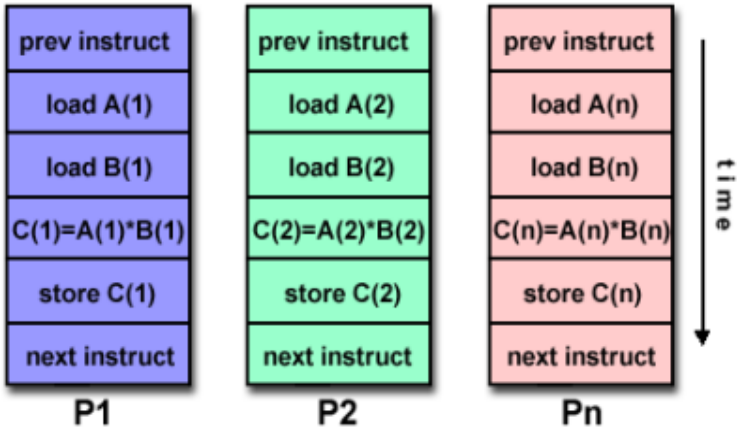


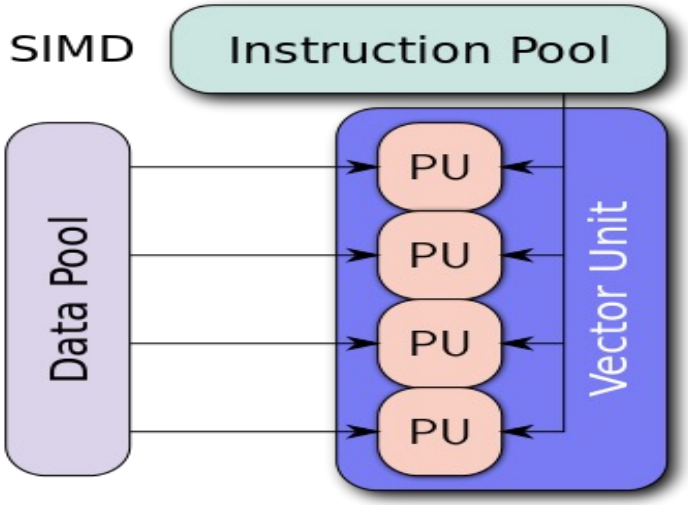


Source: https://en.wikipedia.org/wiki/SISD

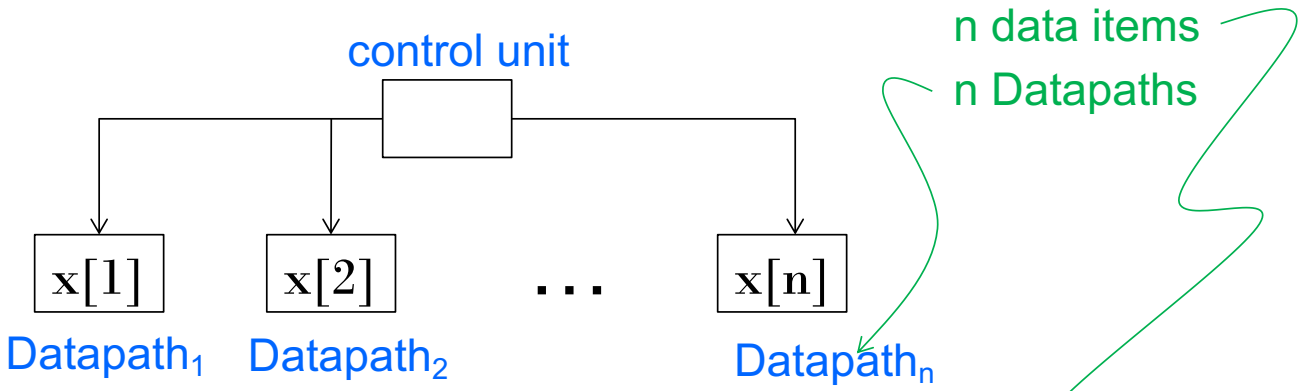
SIMD

- Parallelism is achieved by dividing data among the processors.
- Applies the same **single instruction** to **multiple data** items.
- An abstract SIMD system can be thought of as having **a single control unit** and **multiple datapaths**.
  - An instruction is broadcast **from the control unit** to **the datapaths**, and each datapath either applies the instruction to the current data item, or it is idle.
- Support data parallelism.





SIMD example




```
for (i = 0; i < n; i++)
    x[i] += y[i];
```

SIMD

- What if we don't have as many Datapaths as data items?
- Divide the work and process iteratively.
- Ex. m = 4 Datapaths and n = 15 data items.

Round	$Datapath_1$	$Datapath_2$	$Datapath_3$	$Datapath_4$
1	X[0]	X[1]	X[2]	X[3]
2	X[4]	X[5]	X[6]	X[7]
3	X[8]	X[9]	X[10]	X[11]
4	X[12]	X[13]	X[14]	

SIMD drawbacks

- All Datapaths are required to execute the same instruction or remain idle → which can seriously **degrade the overall performance.**
  - For example, suppose we only want to carry out the addition if y[i] is positive:

```
for ( i = 0; i < n ; i ++)
        i f ( y [ i ] > 0 . 0 ) x [ i] += y [ i ] ;
```

  - Some Datapaths will be idle if the condition is not true while others can proceed with the computation.
- In classic design, they must also operate **synchronously**, that is, each datapath **must wait** for the next instruction to be broadcast before proceeding.
- The Datapaths **have no instruction storage**, so a datapath can't delay the execution of an instruction by storing it for later execution.
- Efficient for large **data parallel** problems, but not other types of more complex parallel problems.

SIMD Systems

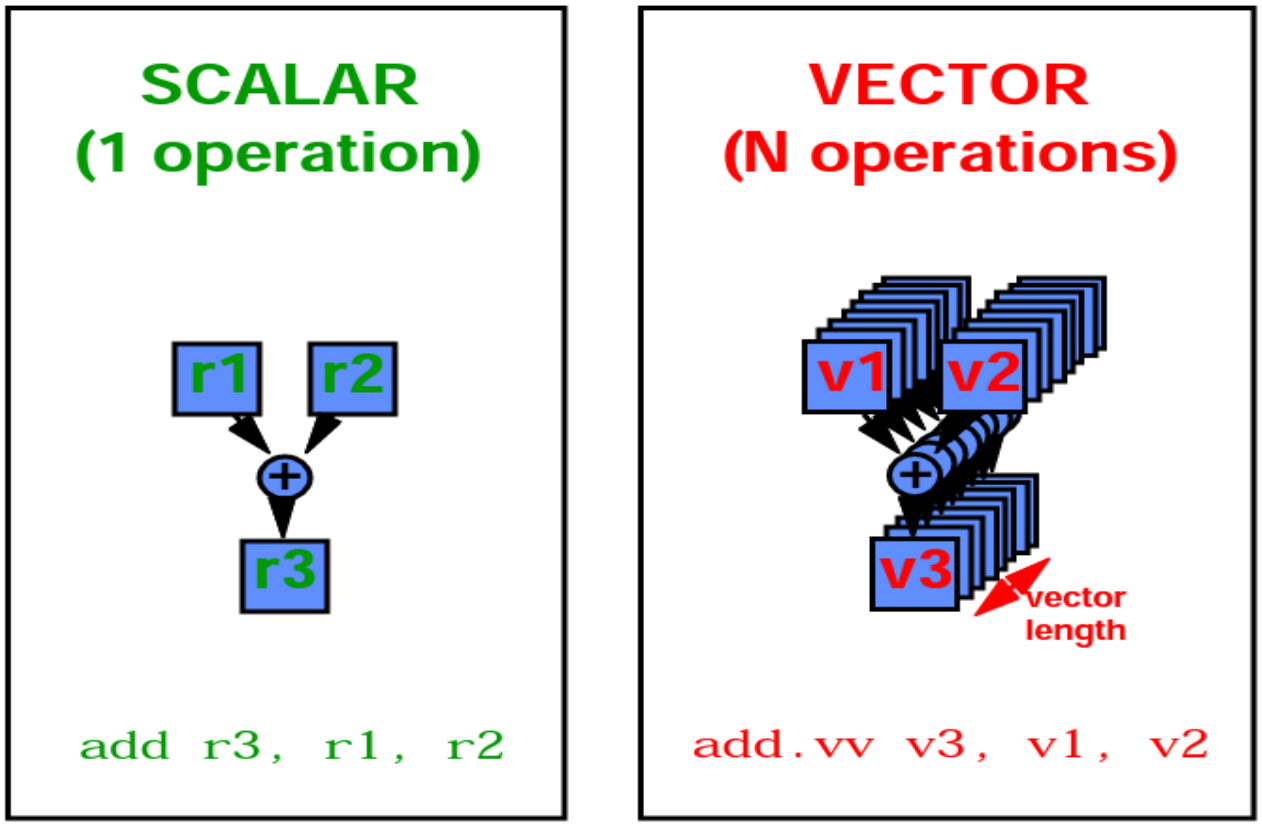
- SIMD systems have had a somewhat checkered history.
  - In the early 1990s, a maker of SIMD systems (Thinking Machines) was one of the largest manufacturers of parallel supercomputers.
  - However, by the late 1990s, the only widely produced SIMD systems were **vector processors**.
  - More recently, **graphics processing units (GPUs)**, and desktop CPUs are making use of aspects of SIMD computing.

Vector processors (1)

- Operate on **arrays or vectors** of data while conventional CPUs operate on **individual data elements** or **scalars**.
  - The operand to the instructions are complete vectors instead of one element
- Typical recent systems have the following characteristics:

1. **Vector registers:** Capable of storing a **vector of operands** and operating simultaneously on their contents.
   - The vector length is fixed by the system and can range from 4 to 256 64-bit elements.
2. **Vectorized and pipelined functional units:** The **same operation** is applied to each element in the vector, or, in the case of operations like addition, the same operation is applied to each pair of corresponding elements in the two vectors. → **vector operations are SIMD**.

Vector processors (2)



Vector processors (3)

3. **Vector instructions.**
   - Operate on vectors rather than scalars.
   - If the vector length is vector_length, these instructions have the great virtue that a simple loop such as

$$\text{for}\,(\,i = 0; i < n\,; i{+}{+})$$

$$x\,[\,i\,] \mathrel{+}= y\,[\,i\,]\,;$$

   requires only a single load, add, and store for each block of vector_length elements, while a conventional system requires a load, add, and store for each element.

Vector processors (3)

4. **optimized memory access**
   - Interleaved memory.
     - Multiple “banks” of memory, which can be accessed more or less independently.
     - Distribute elements of a vector across multiple banks, so reduce or eliminate delay in loading/storing successive elements.
   - Strided memory access
     - The program accesses elements of a vector located at fixed intervals.

Vector processors - Pros

- Fast.
- Easy to use.
- Vectorizing compilers are good at identifying code to exploit.
- Compilers also can provide information about code that cannot be vectorized.
  - Helps the programmer re-evaluate code.
- High memory bandwidth.
- Uses every item in a cache line.

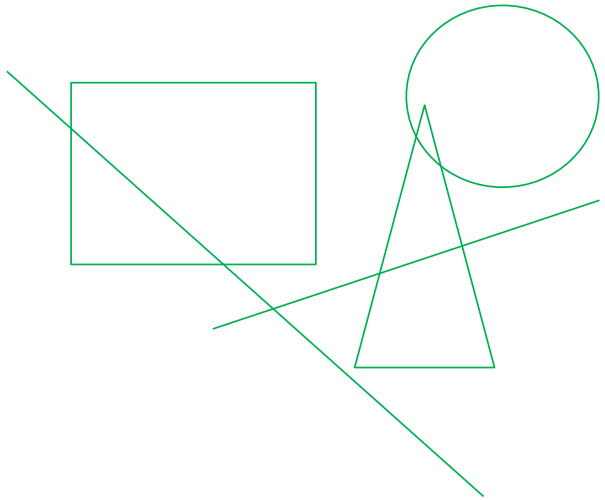
Vector processors - Cons

- They don’t handle irregular data structures as well as other parallel architectures.

- A very finite limit to their ability to handle ever larger problems. (scalability)

Graphics Processing Units (GPU)

- Real time graphics application programming interfaces or API's use points, lines, and triangles to internally represent the surface of an object.



GPUs

- A **graphics processing pipeline** converts the internal representation into an array of pixels that can be sent to a computer screen.

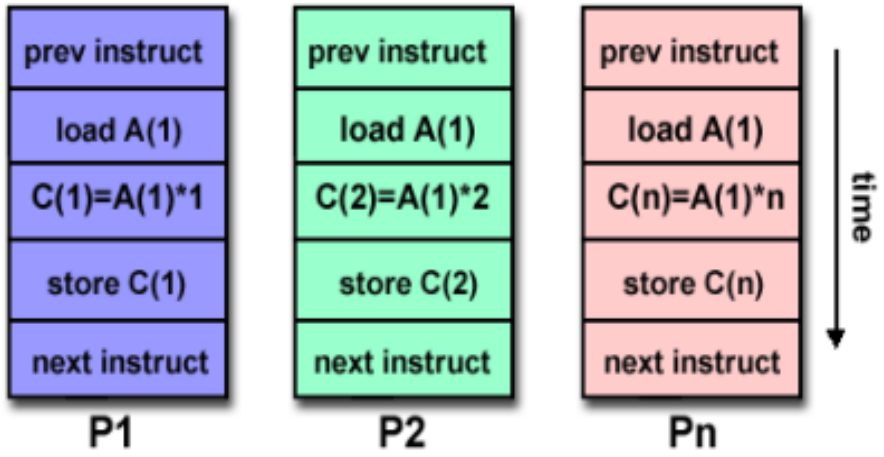
- Several stages of this pipeline (called shader functions) are programmable.
  - Typically just a few lines of C code.

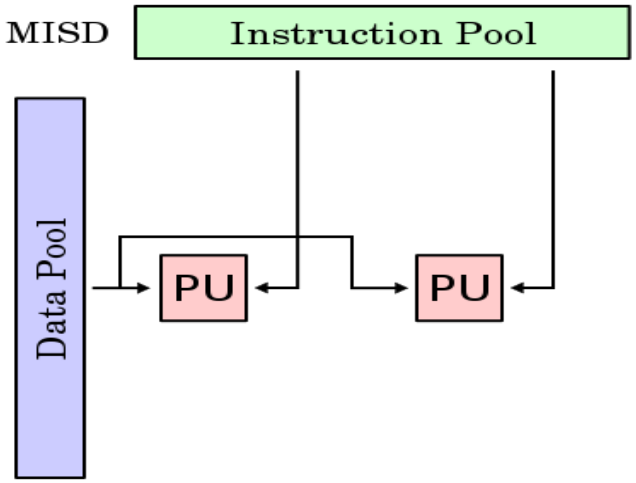
GPUs

- Shader functions are also implicitly parallel, since they can be applied to multiple elements in the graphics stream.
- GPU's can often optimize performance by using SIMD parallelism.
- The current generation of GPUs use SIMD parallelism.
  - Although they are not pure SIMD systems.
- GPUs are becoming increasingly popular for general, high-performance computing, and several languages have been developed that allow users to exploit their power.

MISD

- It is not very common. Mainly mentioned for completeness.
- Multiple functional units or processors execute **multiple instructions** concurrently on the same **single data** stream.
  - A **single data** stream is **fed into multiple processing units**.
  - Each processing unit operates on the data independently via independent instruction streams.

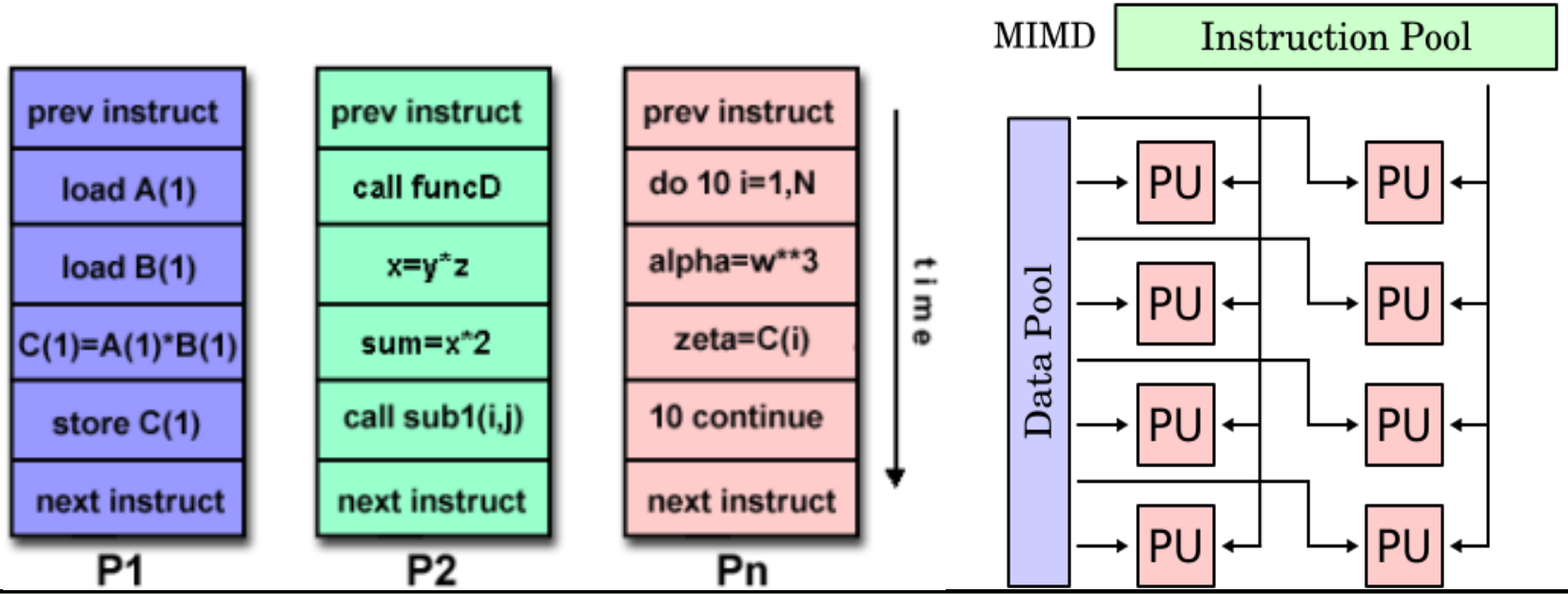




Source: https://en.wikipedia.org/wiki/MISD

MIMD

- Currently, the **most common type of parallel computer**. Most modern computers fall into this category.
- **Multiple Instruction**: Every processor may be executing a different instruction stream
- **Multiple Data**: Every processor may be working with a different data stream



Source: https://en.wikipedia.org/wiki/MIMD

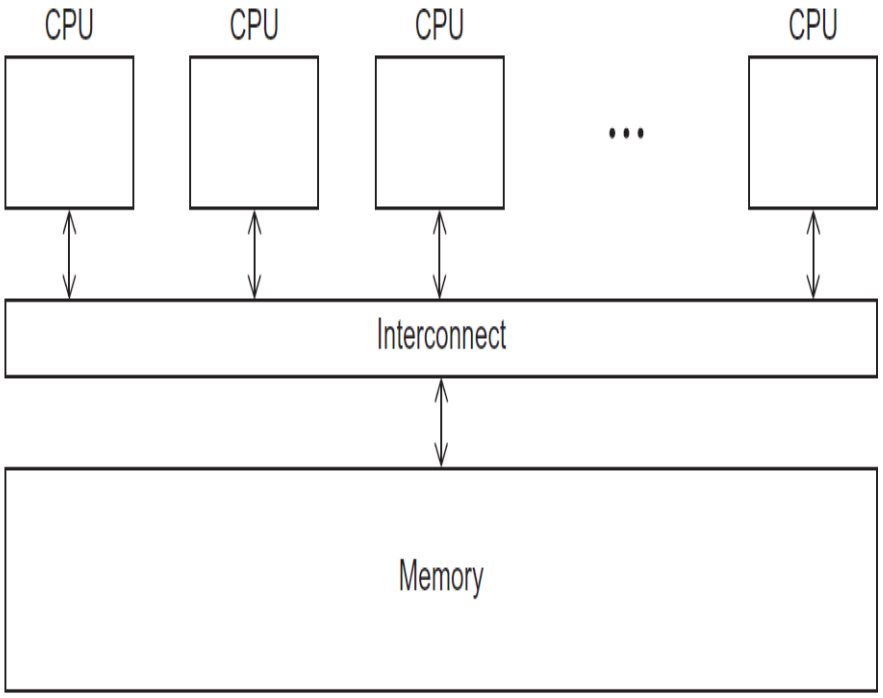
MIMD

- **MIMD systems** typically consist of a collection of fully independent processing units or cores, each of which has its own control unit and its own Datapath.
- Furthermore, unlike SIMD systems, MIMD systems are usually **asynchronous**, that is, the processors can operate at their own pace.
- Factors that promote the development of MIMD architectures:
  - Computers in this category offer support for a wider range of parallel algorithms → with wider applicability.
  - MIMD is extremely cost-effective
- We will focus our study on MIMD architectures ..

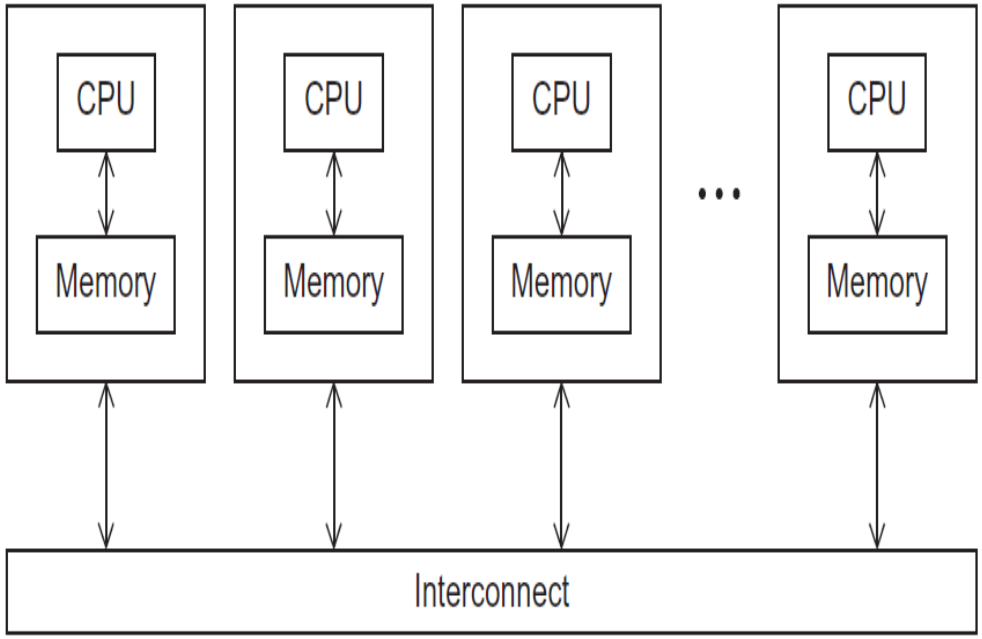
Classification of MIMD memory Architectures

- Two main categories of MIMD architectures, based on the organization of memory:
  1. **Shared memory architectures** → all cores **share access** to the **same** memory.
  2. **Distributed memory architectures** → each core has its **own** local memory, and accesses values in other memories using the network.
     - the processors usually communicate explicitly by **sending messages** or by using special functions that provide access to the memory of another processor.

Classification of MIMD memory Architectures



Shared memory architectures



Distributed memory architectures