

**CENG443**

# **Heterogeneous Parallel Programming**

## **Parallelism**

**Işıl ÖZ, IZTECH, Fall 2023**

**13 October 2023**

The summary of the concepts in the book:

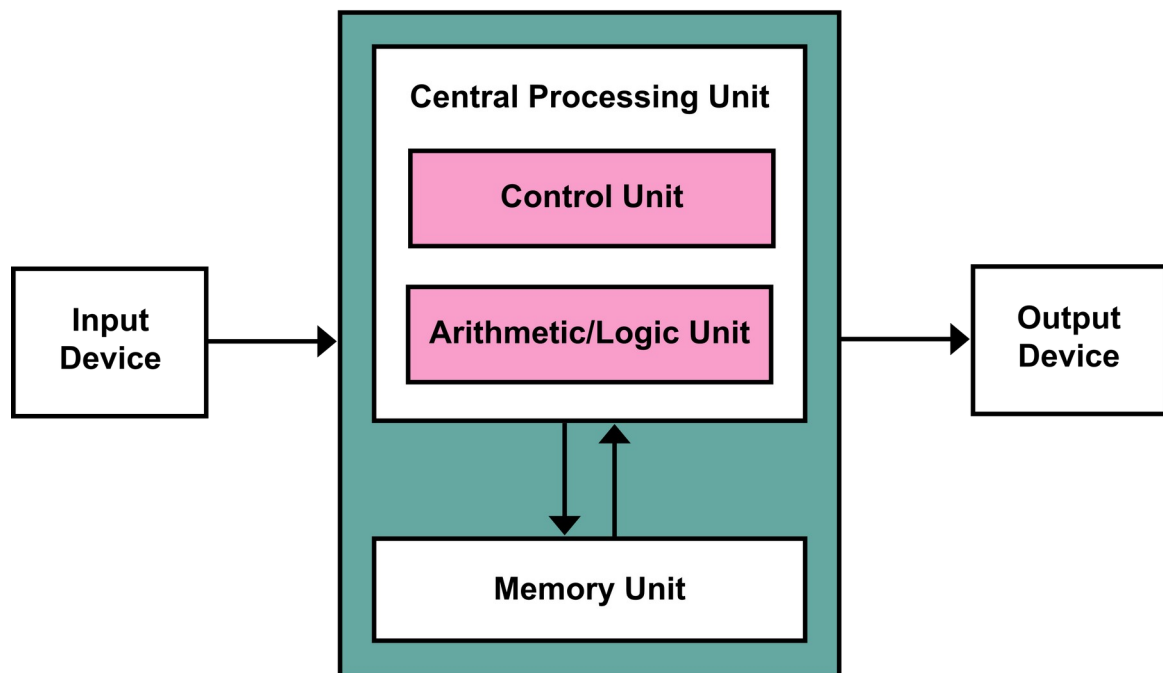
Computer Architecture, A Quantitative Approach. John L. Hennessy, David A. Patterson

# The von Neumann Architecture

**Main memory**

**Central processing unit  
(CPU)**

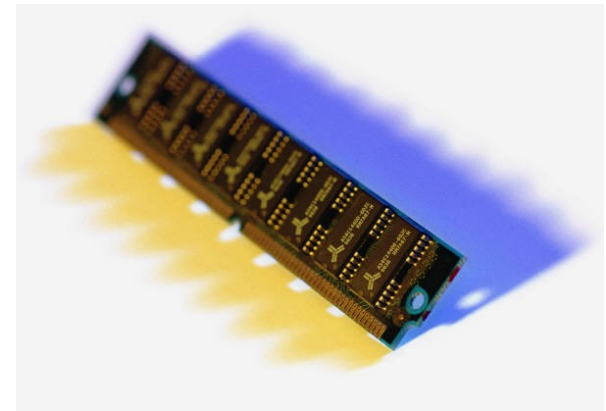
**Interconnection**



# Main Memory

**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**



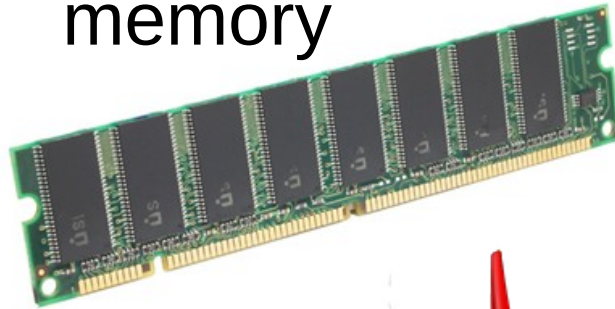
# Central Processing Unit (CPU)

**Control unit : responsible for deciding which instruction in a program should be executed**

**Arithmetic and logic unit (ALU) : responsible for executing the actual instructions**

**Register : quickly accessible location available to CPU**

memory



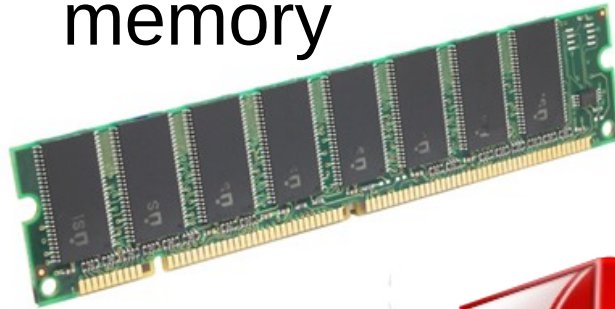
**fetch/read**



CPU



memory



**write/store**



CPU



# Example Program

Compute  $\sin(x)$  using taylor expansion:  $\sin(x) = x - x^3/3! + x^5/5! + x^7/7! + \dots$

```
void sinx(int N, int terms, float* x, float* result)
{
    for (int i=0; i<N; i++)
    {
        float value = x[i];
        float numer = x[i] * x[i] * x[i];
        int denom = 6; // 3!
        int sign = -1;

        for (int j=1; j<=terms; j++)
        {
            value += sign * numer / denom
            numer *= x[i] * x[i];
            denom *= (j+3) * (j+4);
            sign *= -1;
        }

        result[i] = value;
    }
}
```

# Compile Program

```
void sinx(int N, int terms, float* x, float* result)
{
    for (int i=0; i<N; i++)
    {
        float value = x[i];
        float number = x[i] * x[i] * x[i];
        int denom = 6; // 3!
        int sign = -1;

        for (int j=1; j<=terms; j++)
        {
            value += sign * number / denom;
            number *= x[i] * x[i];
            denom *= (j+3) * (j+4);
            sign *= -1;
        }

        result[i] = value;
    }
}
```

x[i]



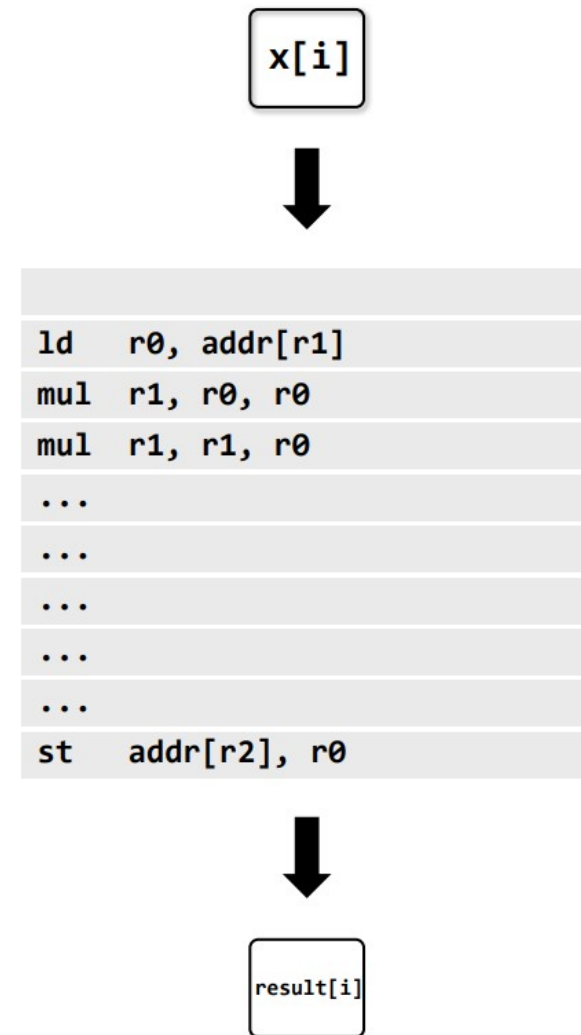
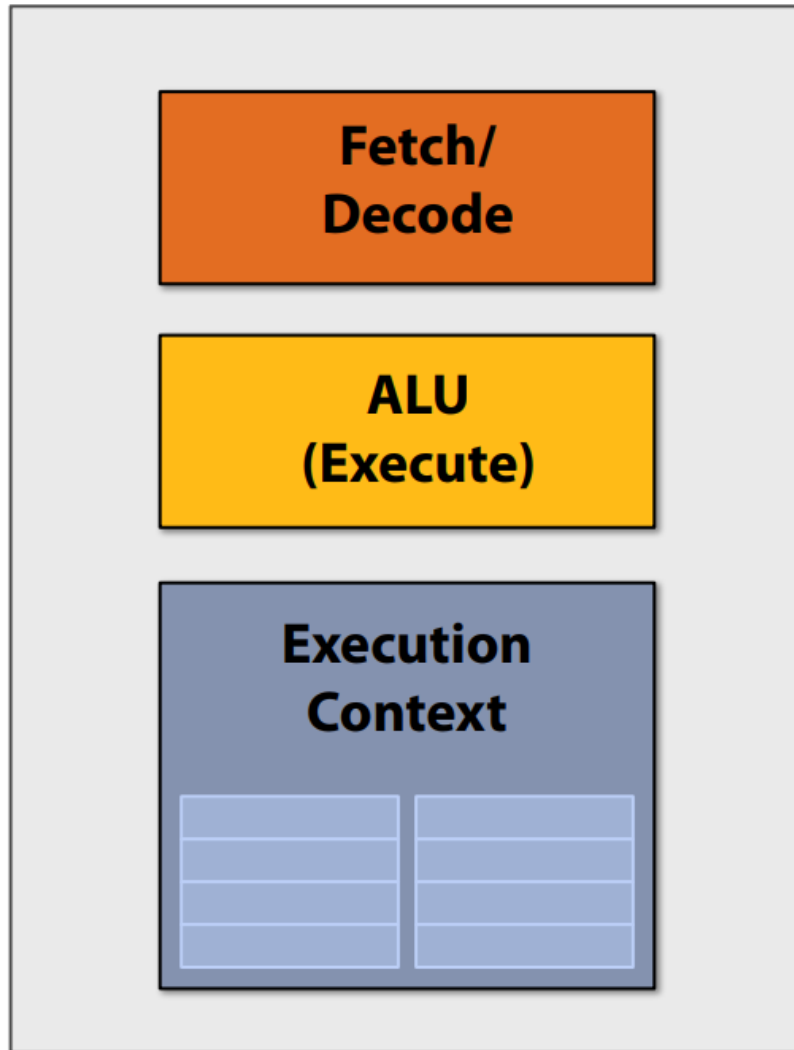
```
ld    r0, addr[r1]
mul   r1, r0, r0
mul   r1, r1, r0
...
...
...
...
...
st    addr[r2], r0
```



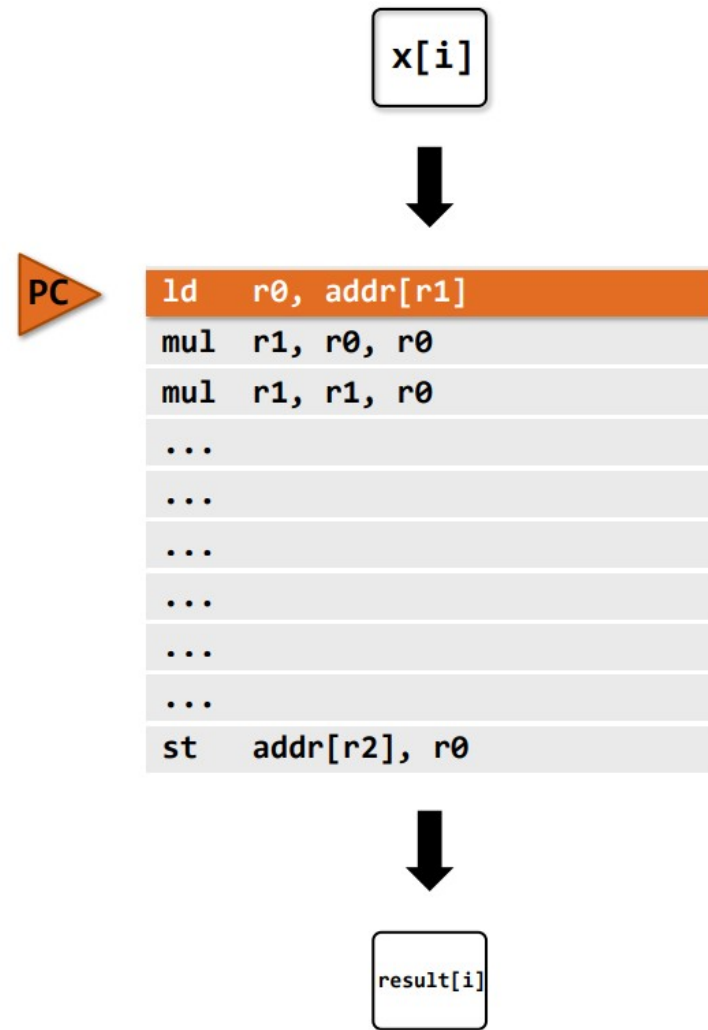
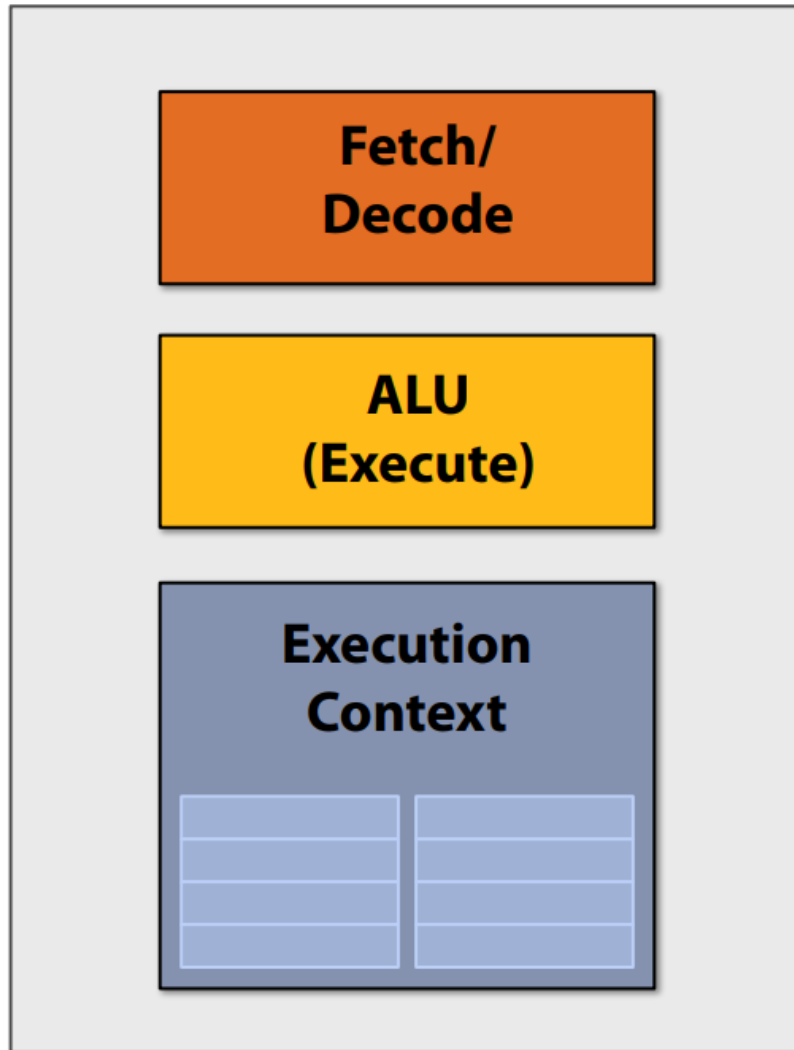
result[i]



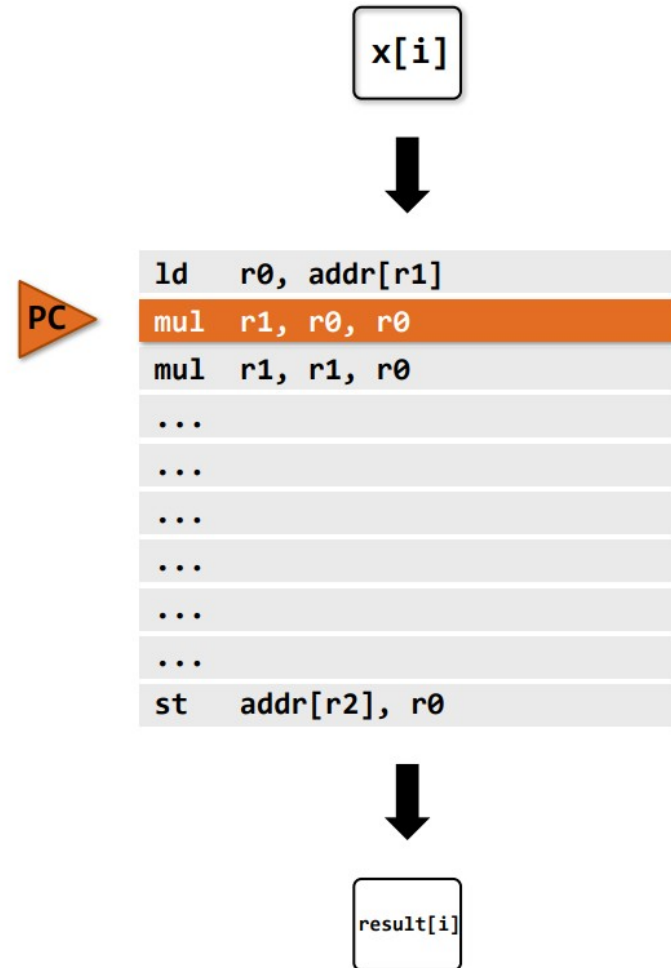
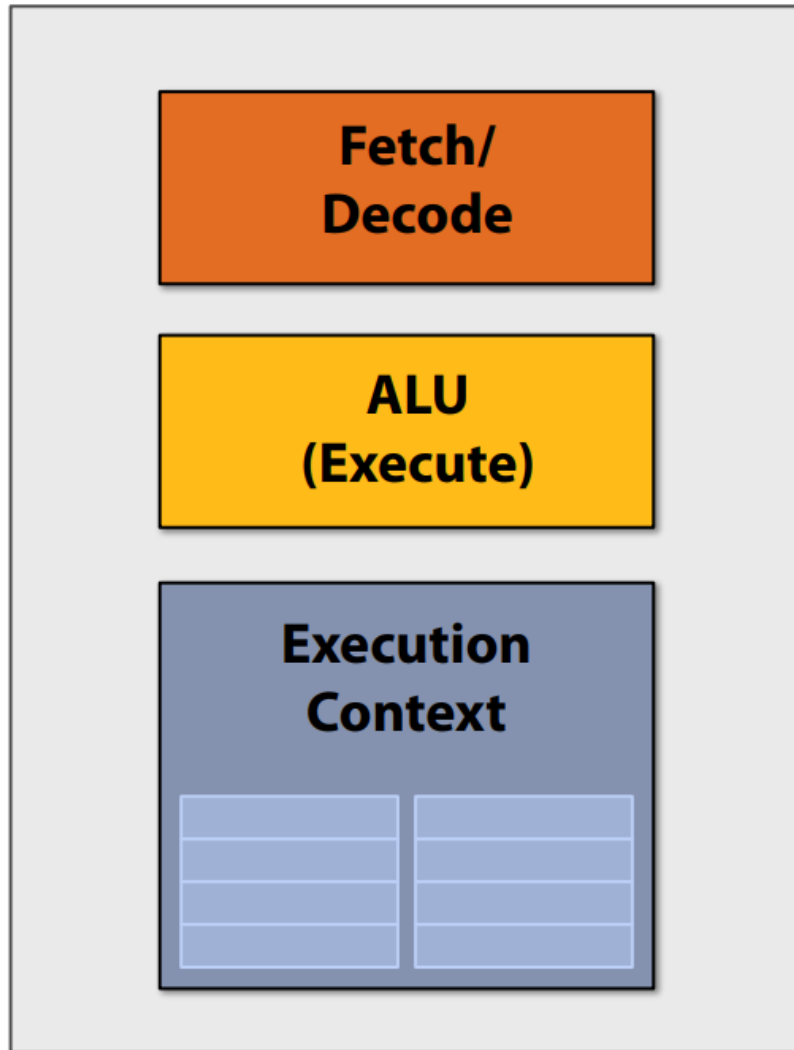
# Execute Program



# Execute One Instruction per clock



# Execute One Instruction per clock



# Von Neumann Bottleneck

**Limited throughput due to data transfer between CPU and memory**

**Improvements to von Neumann to solve the bottleneck or to make CPUs faster**

Caching

Virtual memory

Instruction-level parallelism

Thread-level parallelism

Data-level parallelism

# Improvements to von Neumann

**Caching**

**Virtual memory**

**Instruction-level parallelism**

**Thread-level parallelism**

**Data-level parallelism**

# Limitations of Memory System Performance

## Memory system performance is largely captured by two parameters, latency and bandwidth

Latency: the time from the issue of a memory request to the time the data is available at the processor

Bandwidth: the rate at which data can be pumped to the processor by the memory system

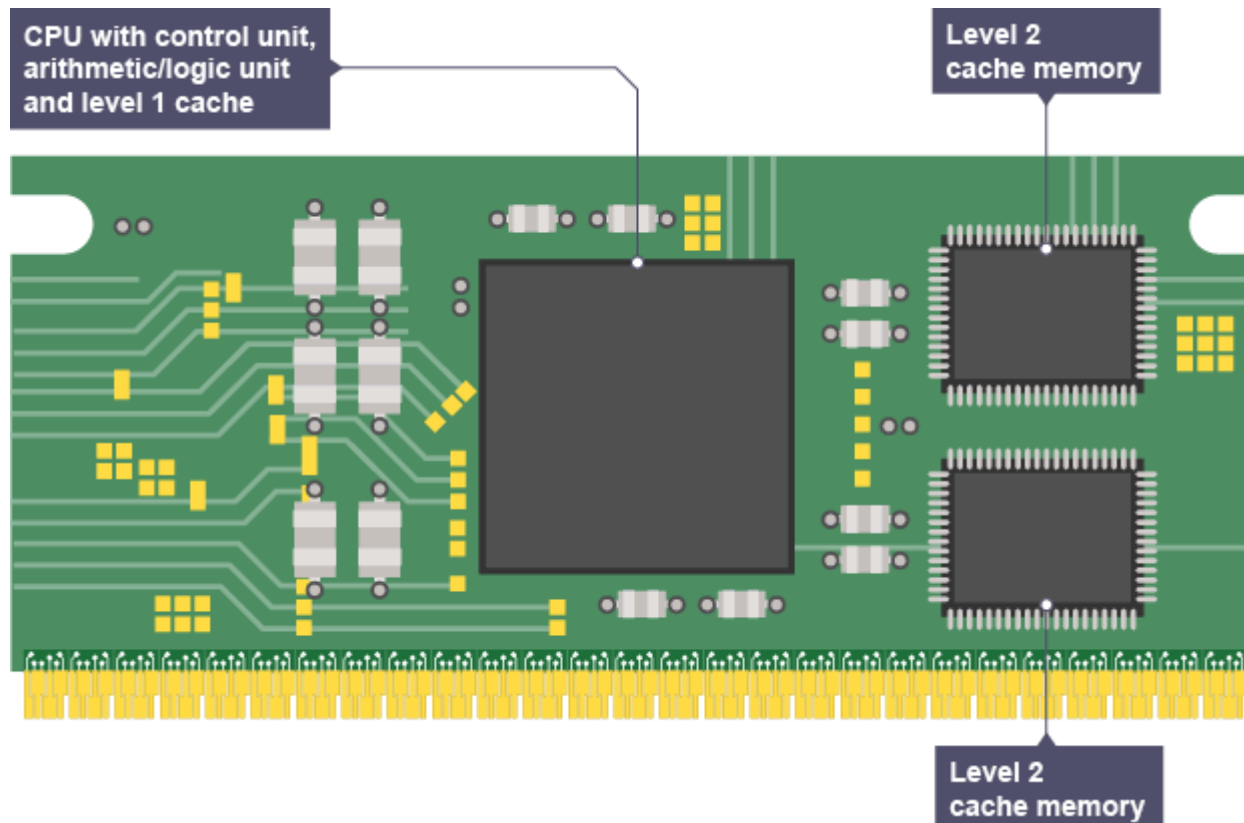
Latency vs bandwidth: Fire-hose analogy

If the water comes out of the hose 2 seconds after the hydrant is turned on, the **latency** of the system is 2 seconds (put out a fire immediately)

If the hydrant delivers water at the rate of 1 gallon/second, the **bandwidth** of the system is 1 gallon/second (large fires)

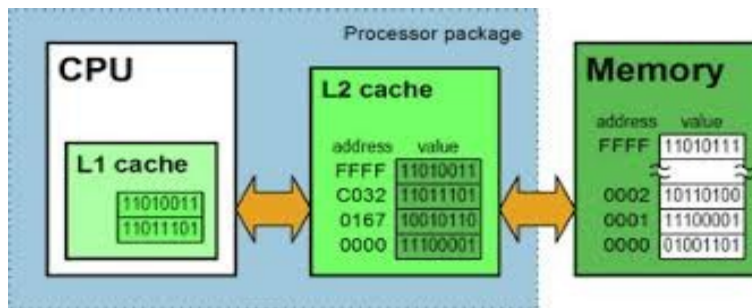
# Caching

**Caches: small and fast memory elements between the processor and main memory**



# Caching

**Collection of memory locations acting as a low-latency high-bandwidth storage**



**CPU can access data in caches more quickly than it can access main memory**

**Which data stored in caches?**



# Locality

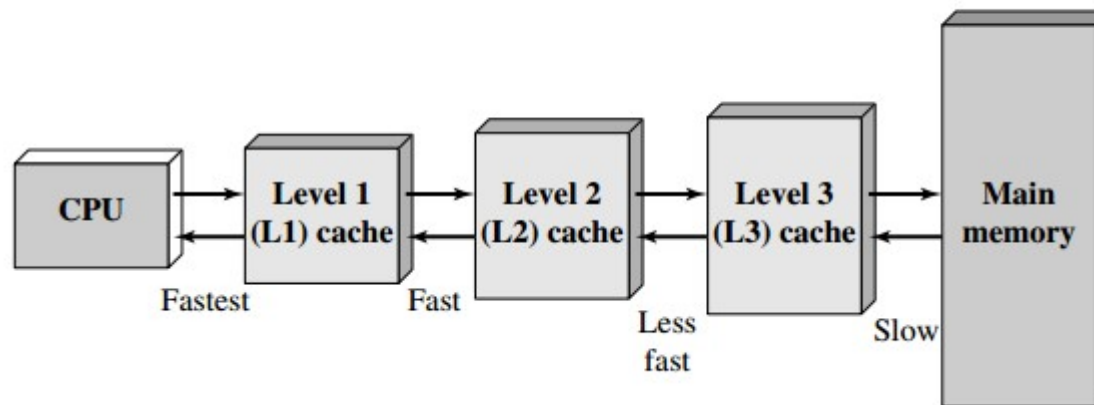
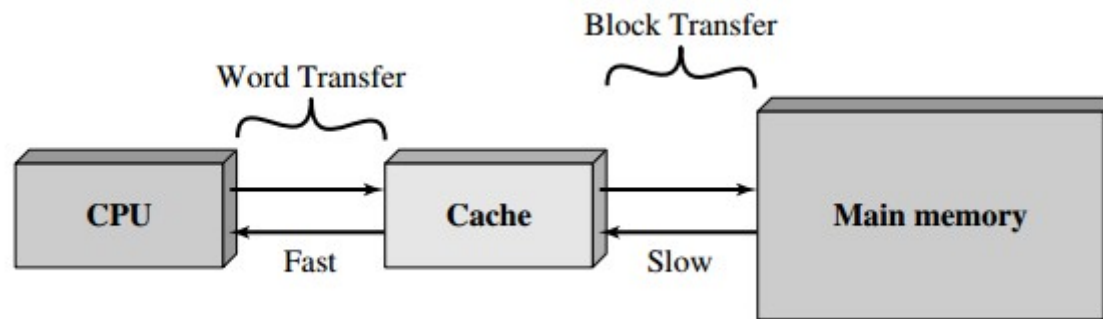
## Temporal locality

Recently accessed items will be accessed in the near future (e.g., code in loops)

## Spatial locality

Items at addresses close to the addresses of recently accessed items will be accessed in the near future (e.g., elements of an array)

```
float z[1000];  
...  
sum = 0.0;  
for (i = 0; i < 1000; i++)  
    sum += z[i];
```



# Improvements to von Neumann

**Caching**

**Virtual memory**

**Instruction-level parallelism**

**Thread-level parallelism**

**Data-level parallelism**

# Virtual Memory

**If we run a program that accesses very large datasets, all data may not fit into main memory**

## **Virtual memory**

Main memory as a cache for secondary storage

Keeping only the active parts of a program in main memory, keeping idle parts in a block of secondary storage called **swap space**

Blocks of data: **pages**, virtual page numbers for programs

Mapping virtual page numbers to physical addresses: **page table**

# Improvements to von Neumann

**Caching**

**Virtual memory**

**Instruction-level parallelism**

**Thread-level parallelism**

**Data-level parallelism**

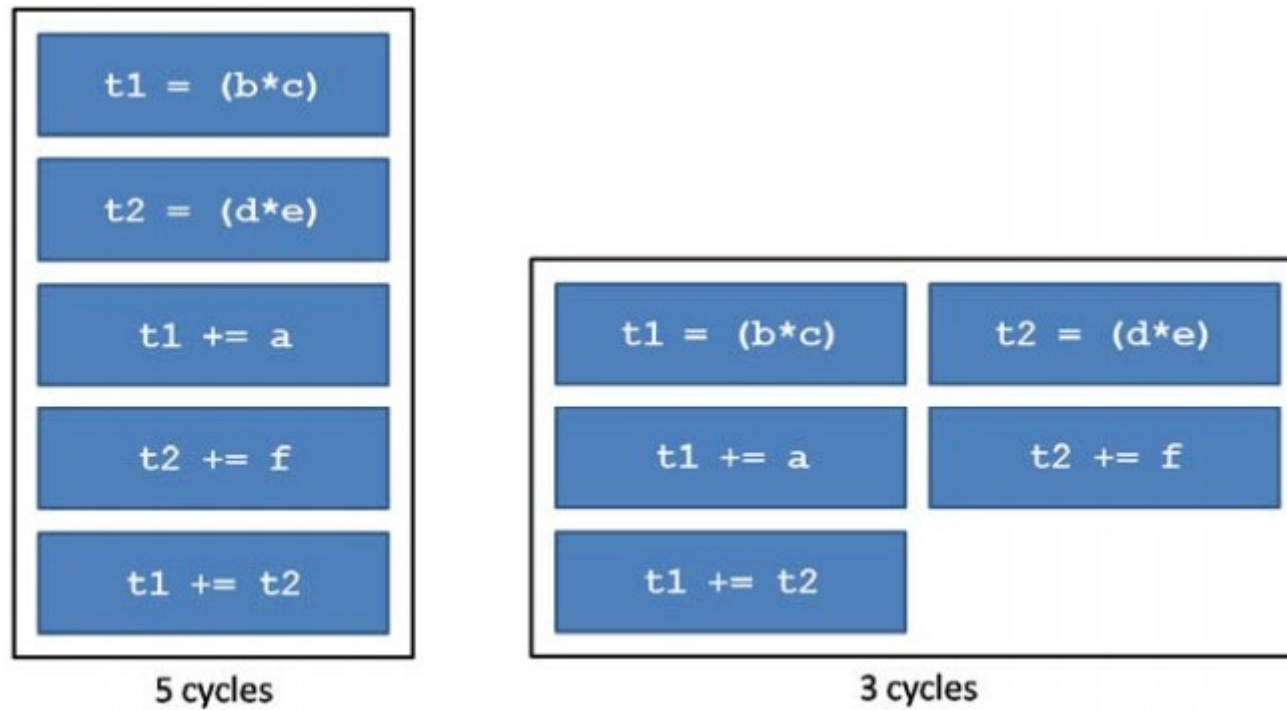
# Instruction-Level Parallelism (ILP)

**Having multiple processor components or functional units simultaneously executing instructions**

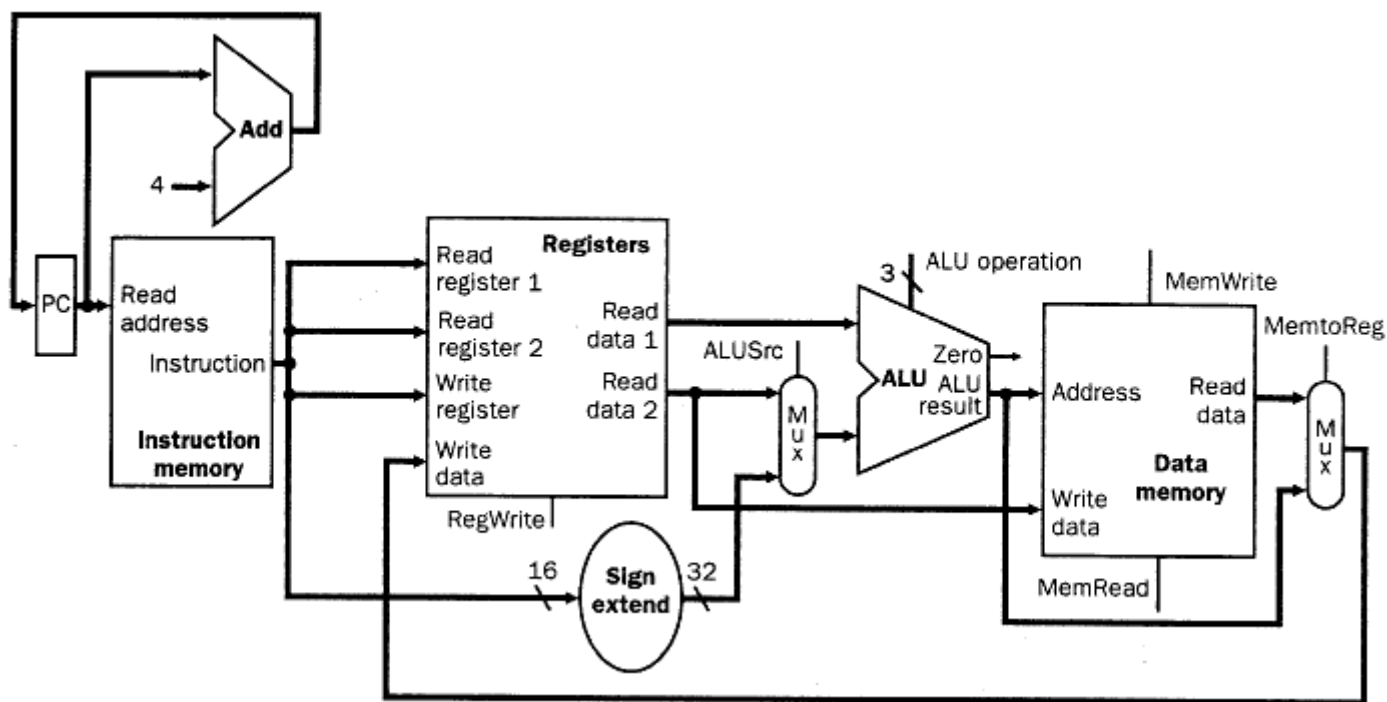
Pipelining

Multiple issue

# Instruction-Level Parallelism



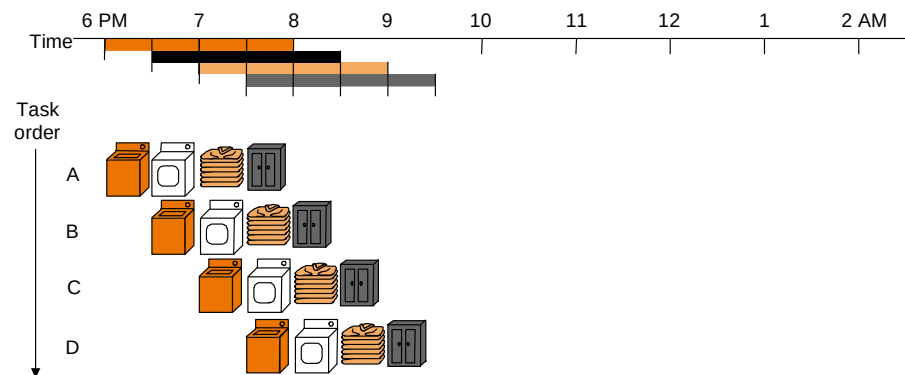
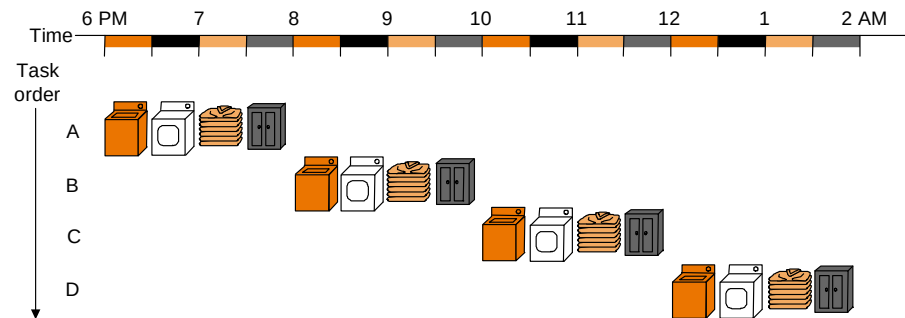
# Single-cycle CPU Datapath



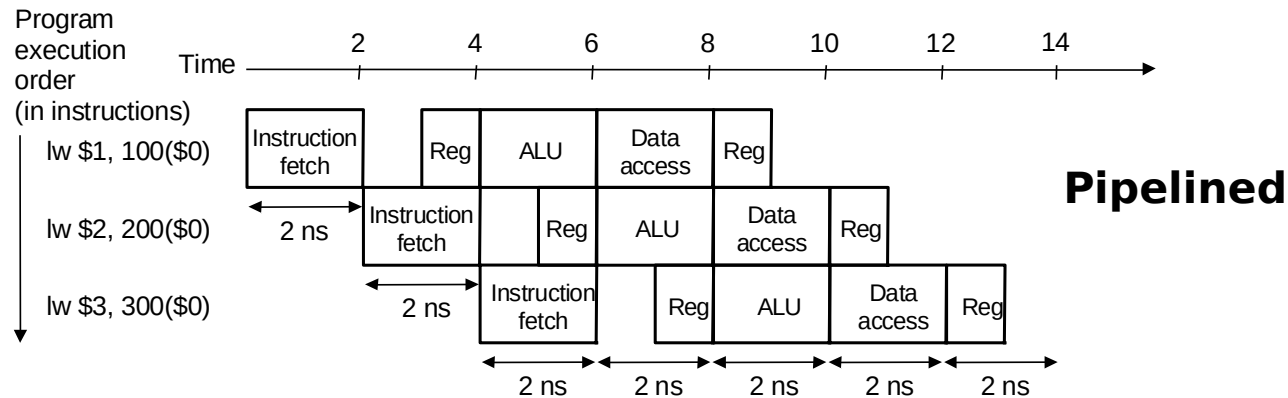
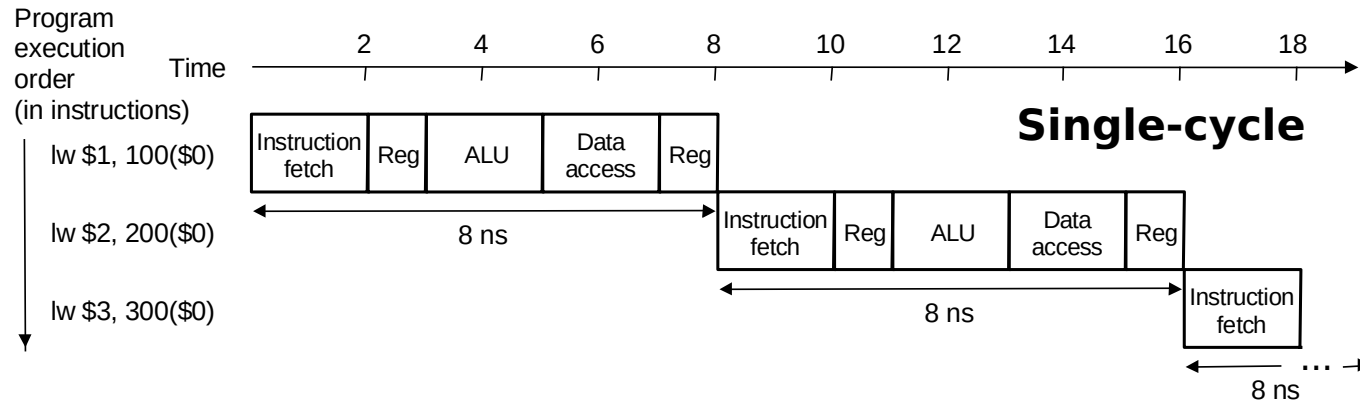


# Pipelining

**Takes individual pieces of hardware or functional units and connects them in sequence**



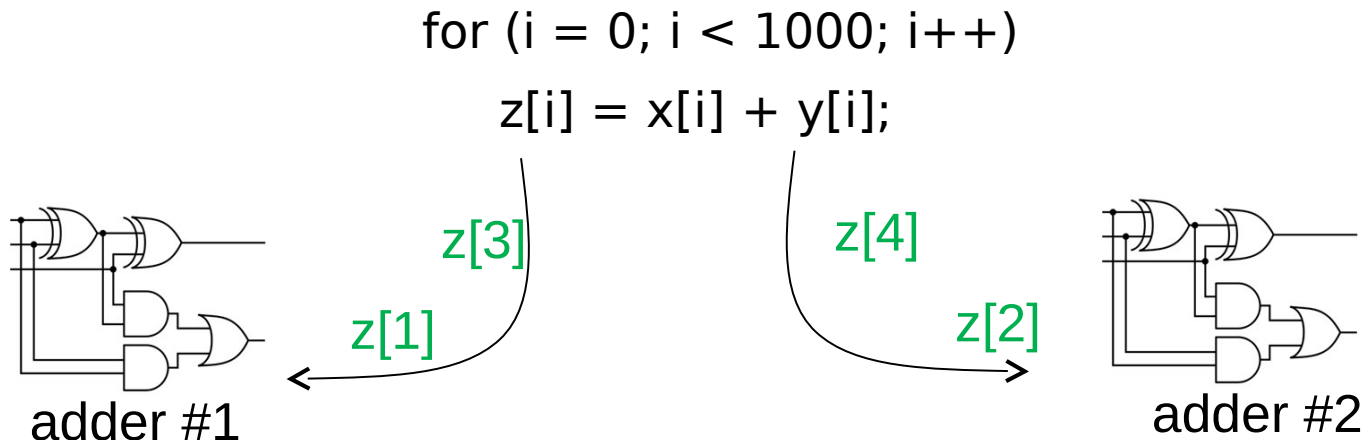
# Pipelined vs Single-Cycle Execution



# Multiple Issue Processors

**Replicate functional units and try to simultaneously execute multiple different instructions in a clock cycle**

**If we have two adders, we can halve the time it takes to execute the loop (first computes  $z[0]$ , the second computes  $z[1]$ , and so on)**



# Multiple Issue Processors

## **Superscalar**

Static

Dynamic

Speculative

## **VLIW**

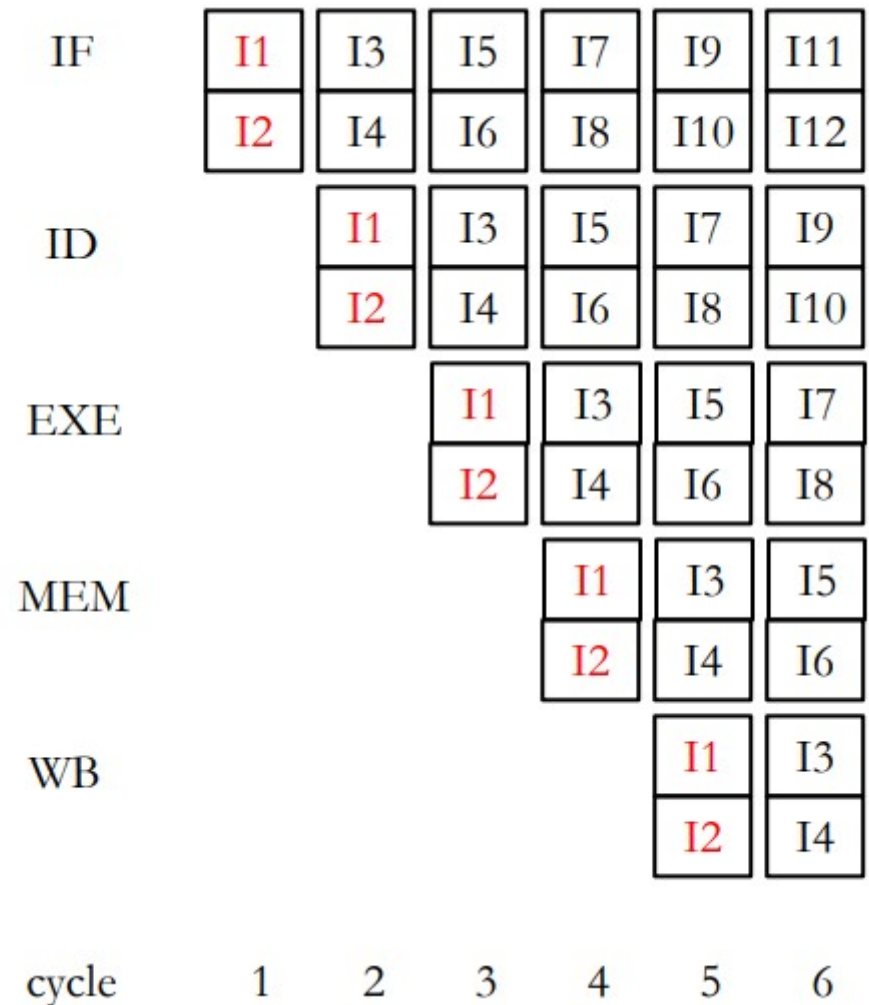
# Superscalar Processors

## Start two instructions per clock cycle by scheduling functional units

Static: If functional units are scheduled at compile-time

Dynamic: If they are scheduled at run-time

Speculative: Scheduling by making some predictions about the outcome of the instructions



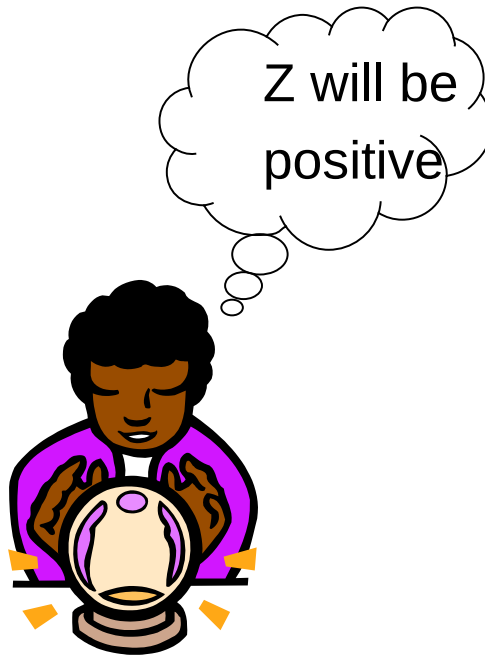
# Speculation

**In order to make use of multiple issue, the system must find instructions that can be executed simultaneously**

**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

```
z = x + y ;  
if ( z > 0 )  
    w = x + 2 ;  
else  
    w = y * 3;
```



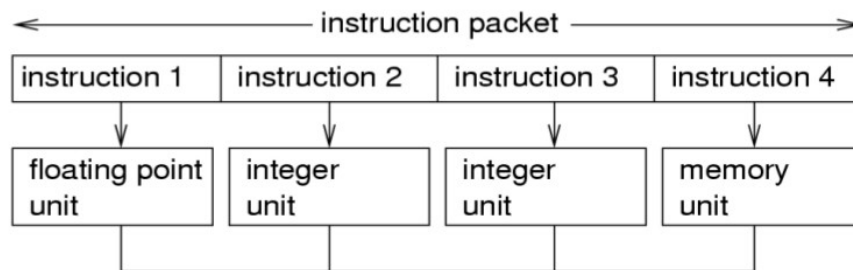
If the system speculates incorrectly,  
it must go back and recalculate  $w = y * 3$

# VLIW (Very large instruction window)

**Uses multiple, independent functional units**

**Does not issue multiple independent instructions to the units**

**Packages the multiple operations into one very long instruction**





# Improvements to von Neumann

**Caching**

**Virtual memory**

**Instruction-level parallelism**

**Thread-level parallelism**

**Data-level parallelism**

# Thread-Level Parallelism

**Parallelism through the simultaneous execution of different threads**

**Multiple data from multiple instructions (MIMD)**

**Multiprocessors**

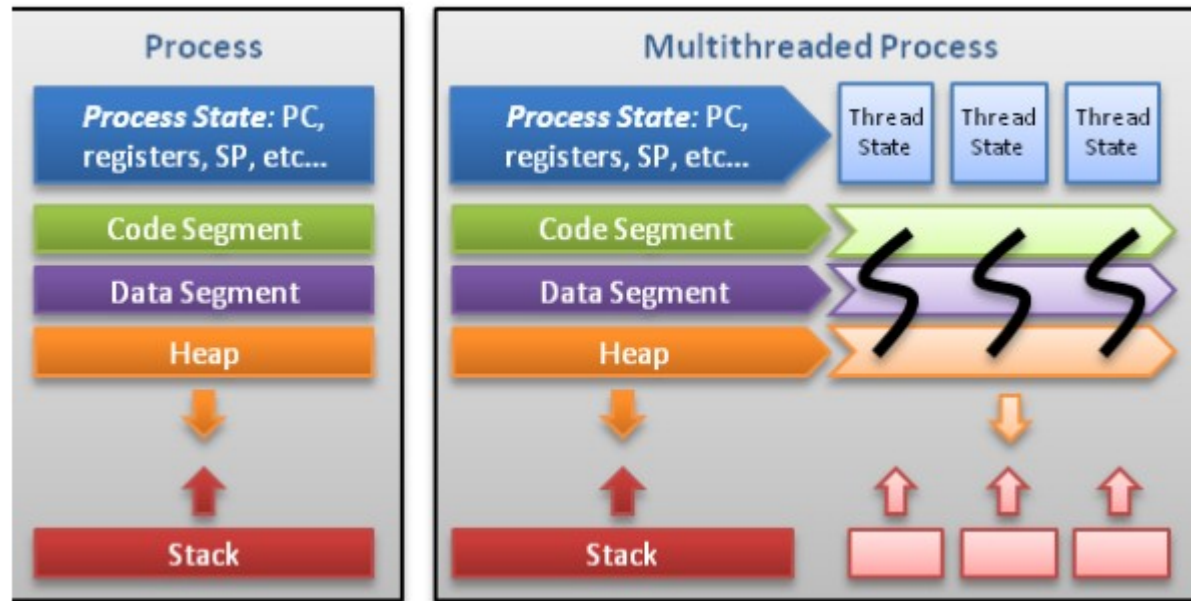
Shared-memory multiprocessors

Distributed-memory multiprocessors

# Threads

**Threads use, and exist within, the process resources**

**Scheduled and run as independent entities**

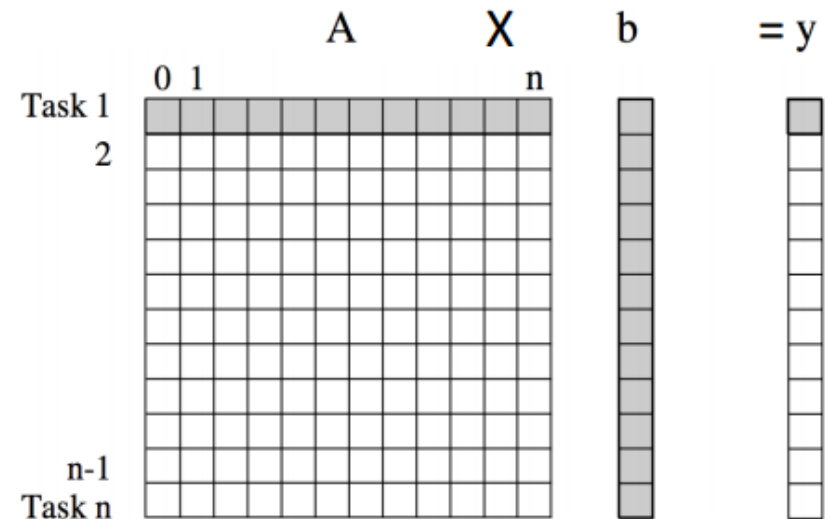


# Thread as Function Instance

```
for (i = 0; i < n; i++)  
    y[i] = dot_product(row(A, i), b);
```

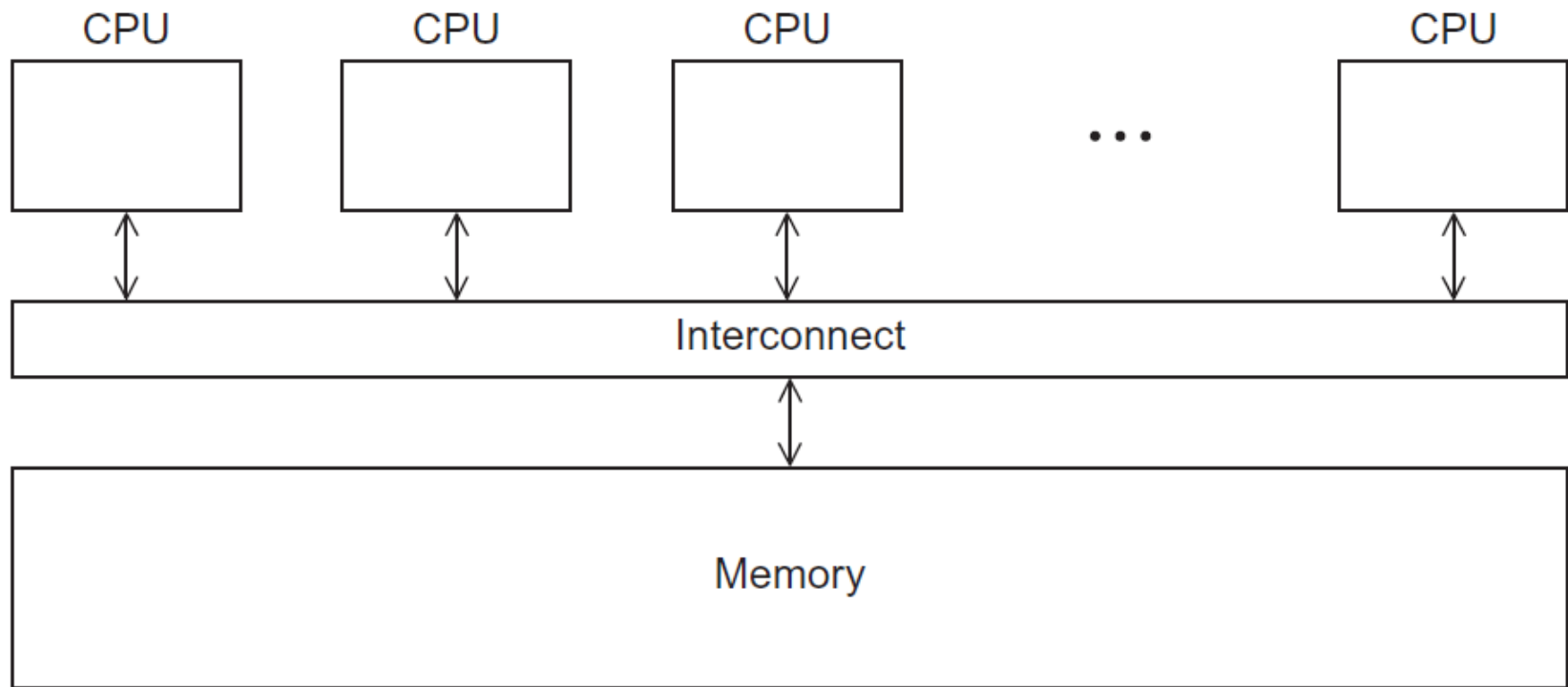


```
for (i = 0; i < n; i++)  
    y[i] = create_thread(dot_product(row(A, i), b));
```



# Shared-Memory System

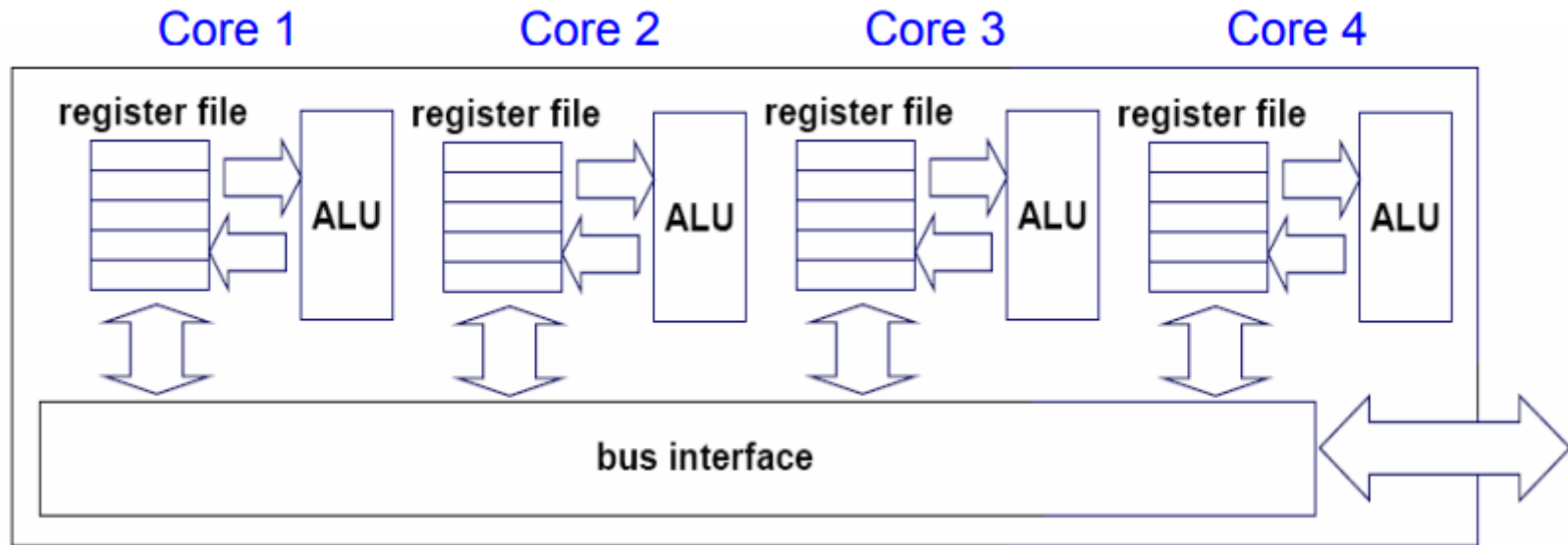
**A collection of processors connected to a memory system via an interconnection network, each processor can access each memory location**



# Multicore Processors

## Multiple CPUs on a single chip

**The most widely available shared-memory systems use one or more multicore processors**



# Shared-Memory MPs

## **Load/store instructions can directly access all part of the shared address space**

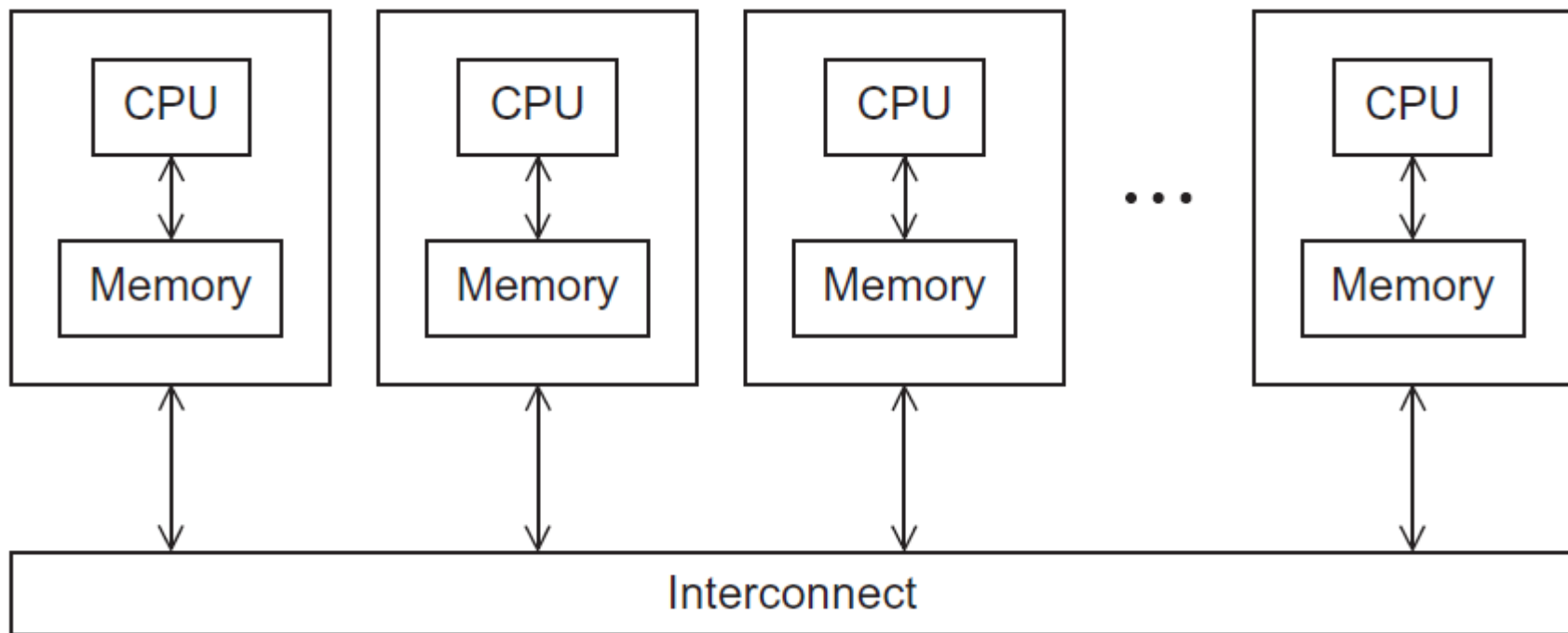
The same instruction supported by uniprocessors

Simple and fast mechanism to communicate and share information

## **Specific instructions to build mechanisms of synchronization**

# Distributed-Memory System

**A collection of computers connected via an interconnection network with their own memory, processors need to communicate each other by *message passing***





# Improvements to von Neumann

**Caching**

**Virtual memory**

**Instruction-level parallelism**

**Thread-level parallelism**

**Data-level parallelism**

# Data-Level Parallelism

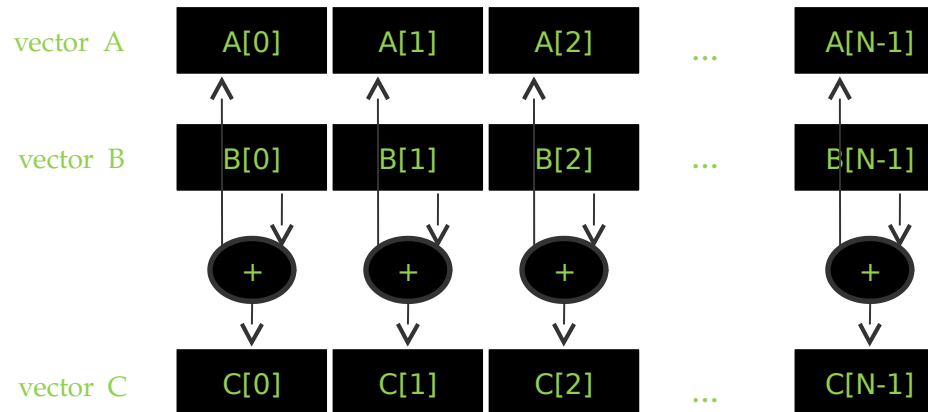
## Single instruction processing multiple data (SIMD)

Vector processors

SIMD Multimedia instructions

GPUs

# Data Parallelism - Vector Addition Example



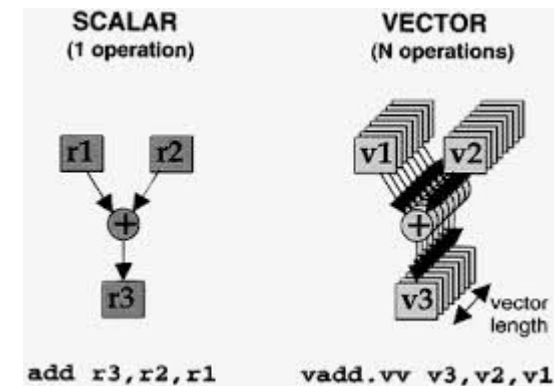
# Vector Processors

**Operate on arrays or vectors of data while conventional CPUs operate on individual data elements or scalars**

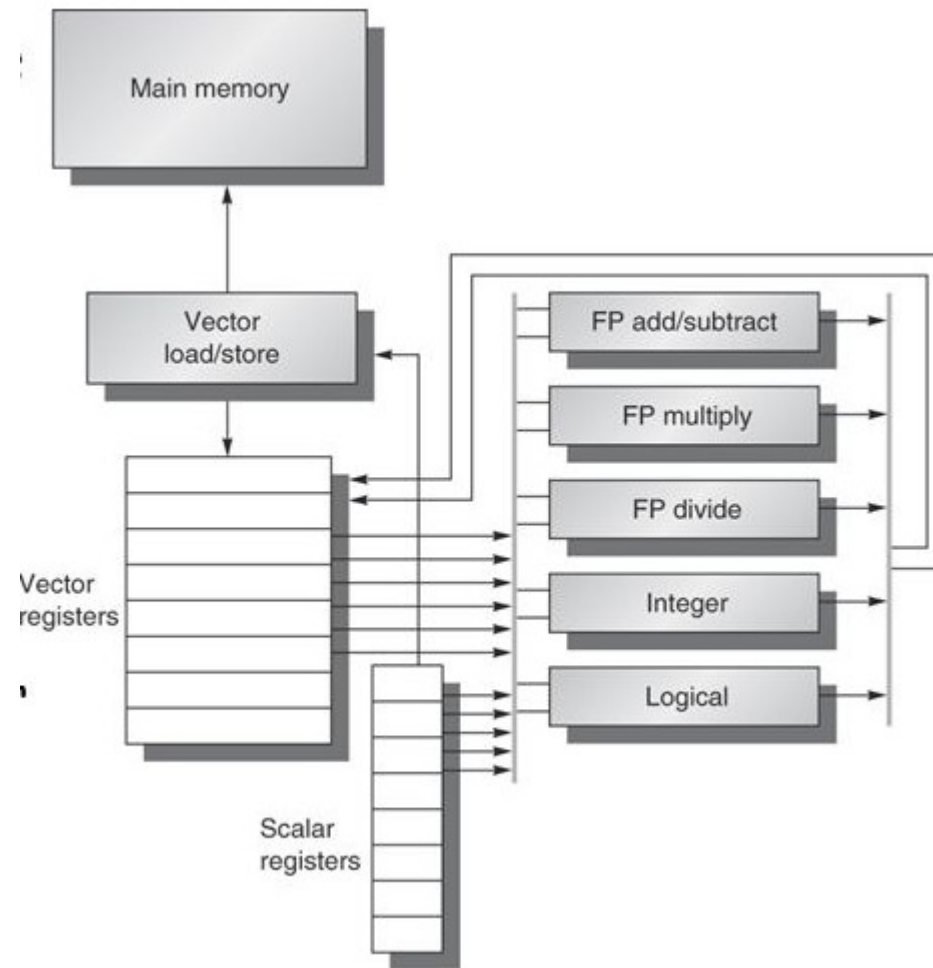
Vector registers

Vectorized and pipelined functional units

Vector instructions



# Vector Architecture



# AVX-512

## Advanced vector extensions by Intel 512-bit extensions

# SIMD Multimedia Instructions

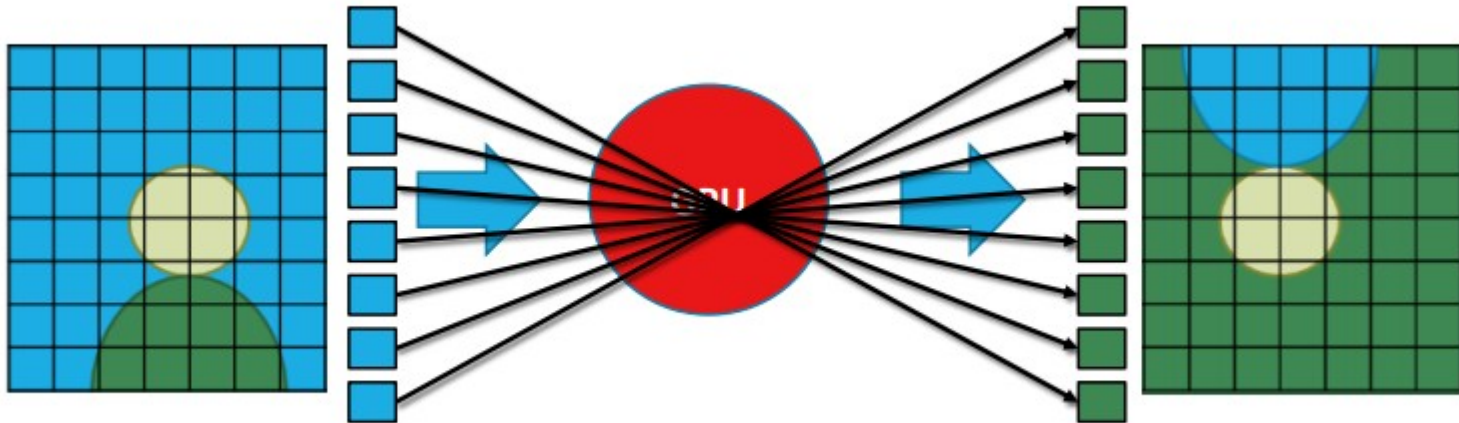
**Since multimedia applications require narrower data types than 64-bit processors, partitioning registers and datapath**

**Treat a 64-bit register as a vector of two 32-bit or four 16-bit or eight 8-bit values (like short vectors)**

**Partition 64-bit datapaths to handle multiple narrow operations in parallel**

# Graphics Processing Units (GPUs)

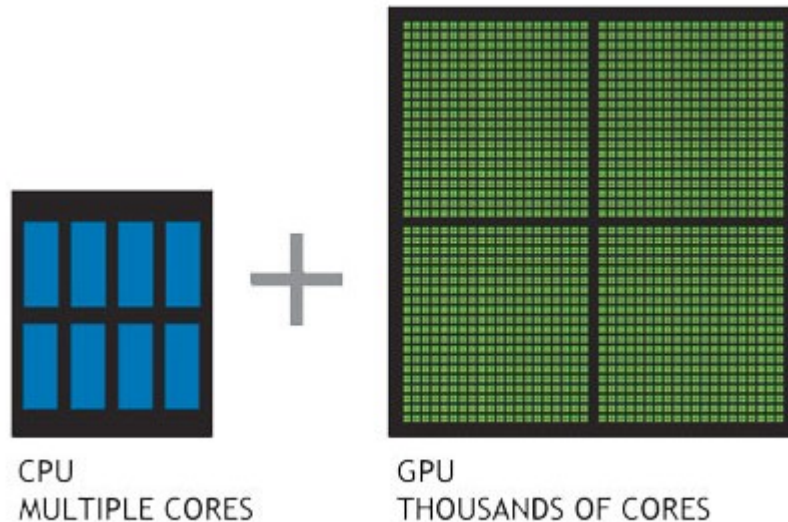
**Graphics workloads: Identical, independent, streaming computation on pixels**





# Graphics Processing Units (GPUs)

**A massively parallel architecture consisting of thousands of smaller, more efficient cores designed for handling multiple tasks simultaneously**



# Graphics Pipeline

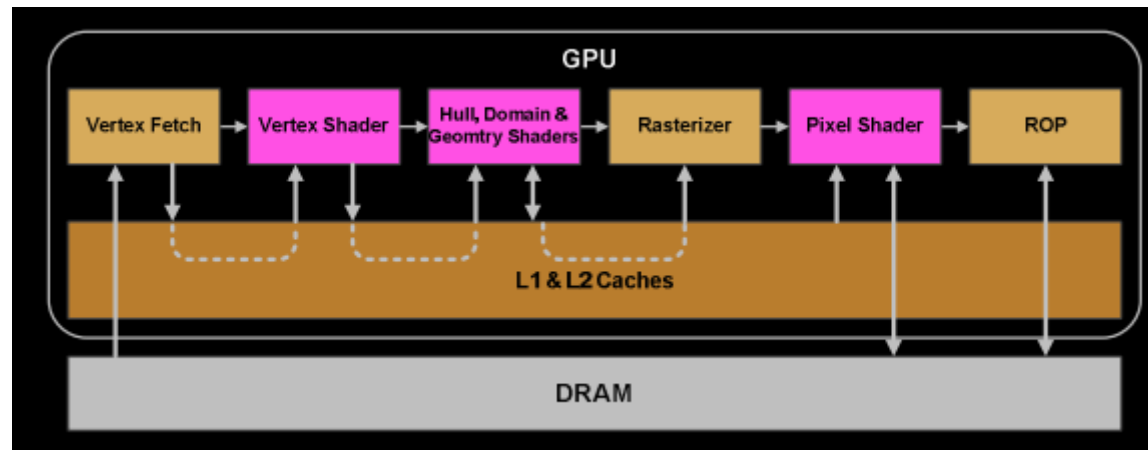
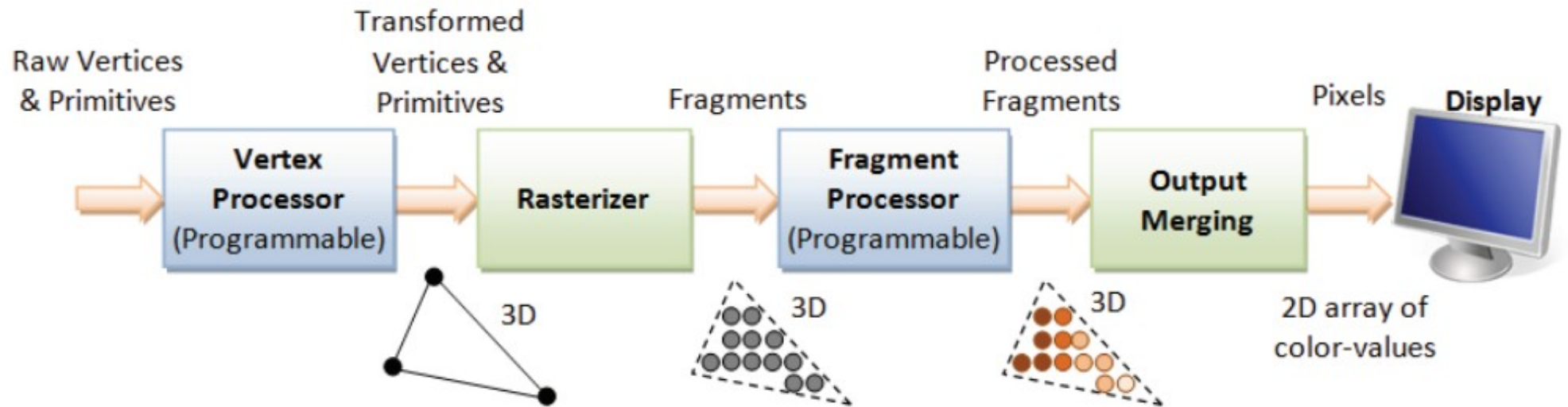
**Due to the nature of the computer graphics, most graphic pipelines have been structured as a computing states with data flowing as streams between them**

**Converts the internal representation into an array of pixels that can be sent to a screen**

**Several stages of this pipeline (shader functions) are programmable**

**Shader functions are processed independently, but no explicit parallel programming, they are implicitly parallel**

# Graphics Pipeline



# Evolution of Graphics Pipelines

## Fixed-function pipelines

configurable but not programmable

DirectX, OpenGL, early NVIDIA GeForce GPUs

## First step toward achieving true general shader programmability

GeForce 3, 2001

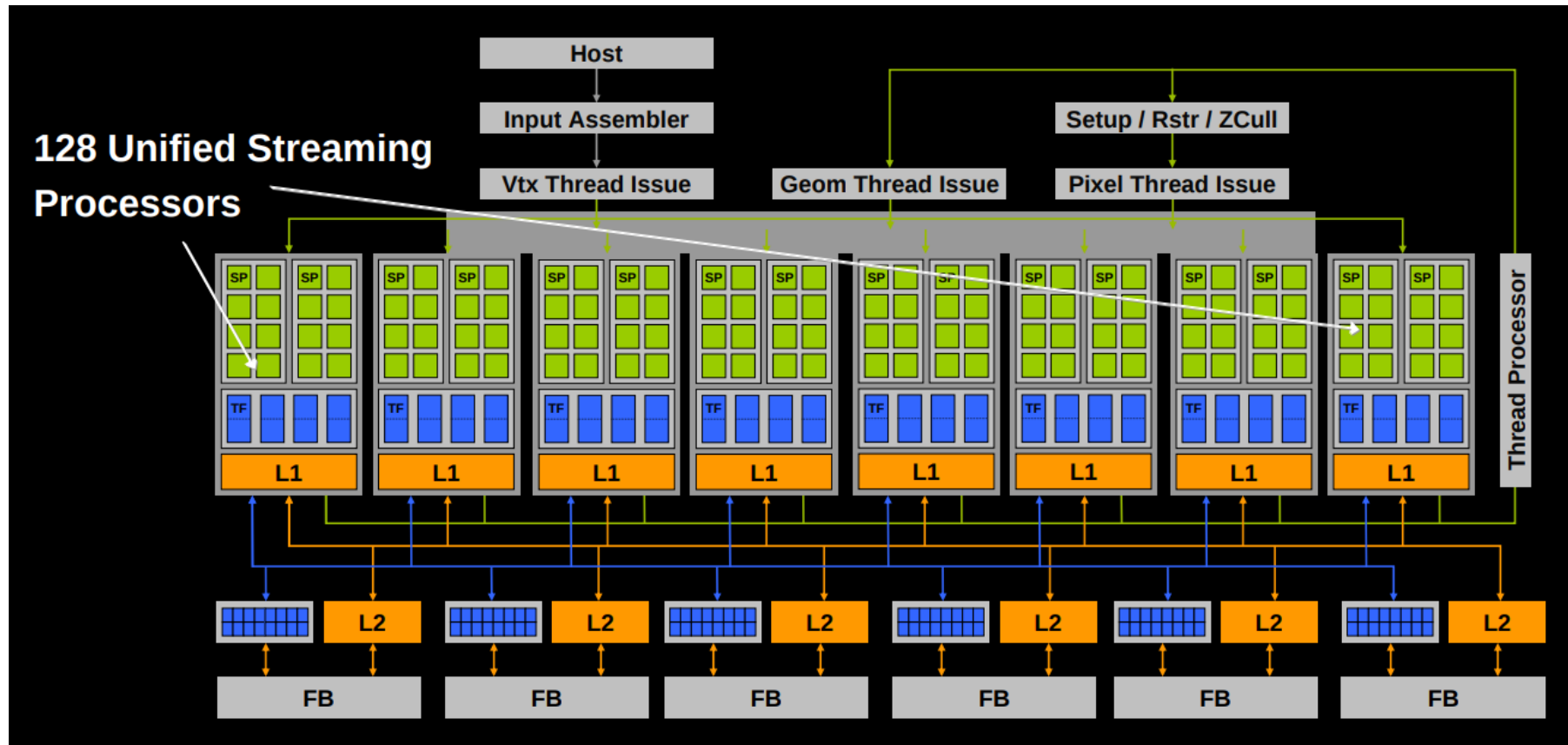
## Mapping the separate **programmable** graphics stages to an array of unified processors

GeForce 8800 GPU (G80), 2006

## GPGPU

GPU like a processor, resembled high-performance parallel computers

# G80 Replaces Pipeline Model



# Inside a Streaming Multiprocessor (SM)

## Multithreaded Instruction Unit

Hardware thread/warp scheduler

In-order issue

## Register File

32-bit registers

## Multiple Cores

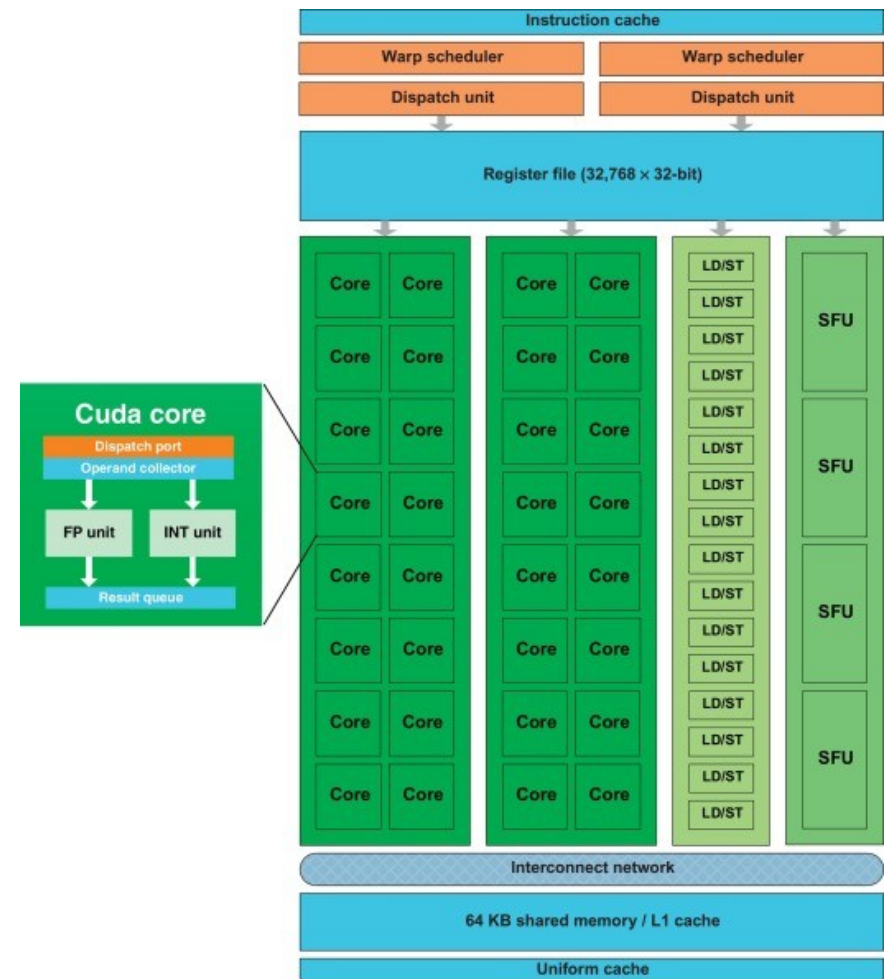
IEEE 754 32-bit floating point

32-bit and 64-bit integer

## Special Function Units (SFU)

sin, cos, log, exp

## Shared Memory



# GPU Computing Today

## **GPUs and CUDA bring parallel computing to the masses**

Over 100M CUDA-capable GPUs sold to date

60K CUDA developers

A “developer kit” (i.e. GPU) costs ~\$200 (for 500 GFLOPS)

## **Data-parallel supercomputers are everywhere!**

CUDA makes this power accessible

We're already seeing innovations in data-parallel computing

## **Massively parallel computing has become a commodity technology!**