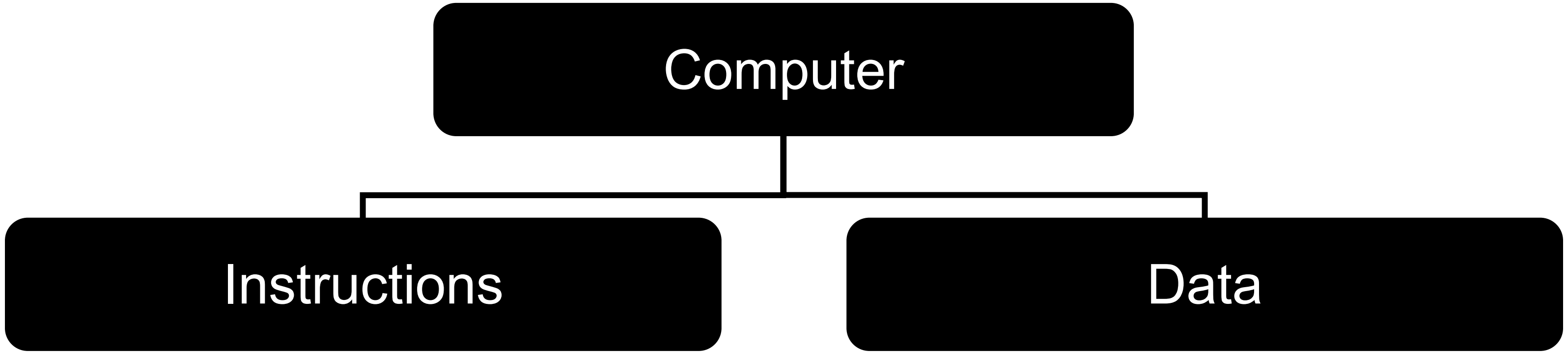
Lecture 2: Parallel Architectures

Tsung-Wei (TW) Huang
Department of Electrical and Computer Engineering
University of Utah, Salt Lake City, UT

Hardware Parallelism

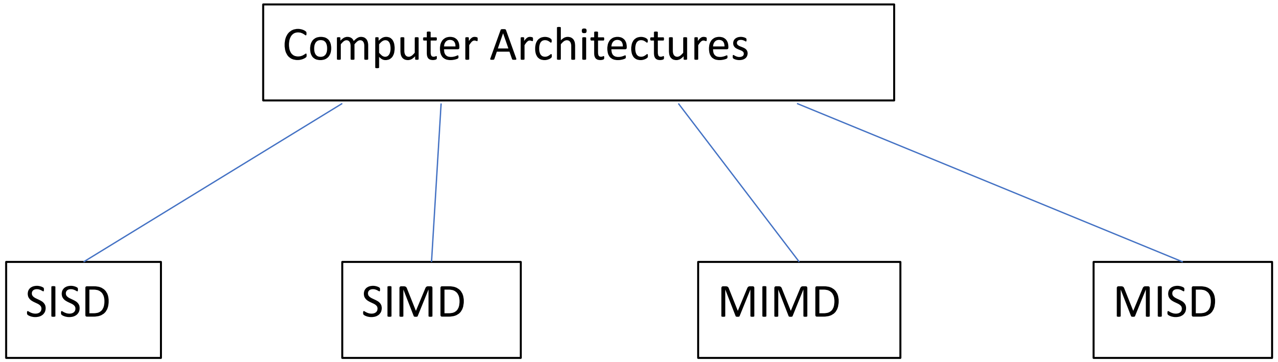
- Computing: execute instructions that operate on data.



- We can classify computer architectures based on the number of instructions that can be executed and how they operate on data.

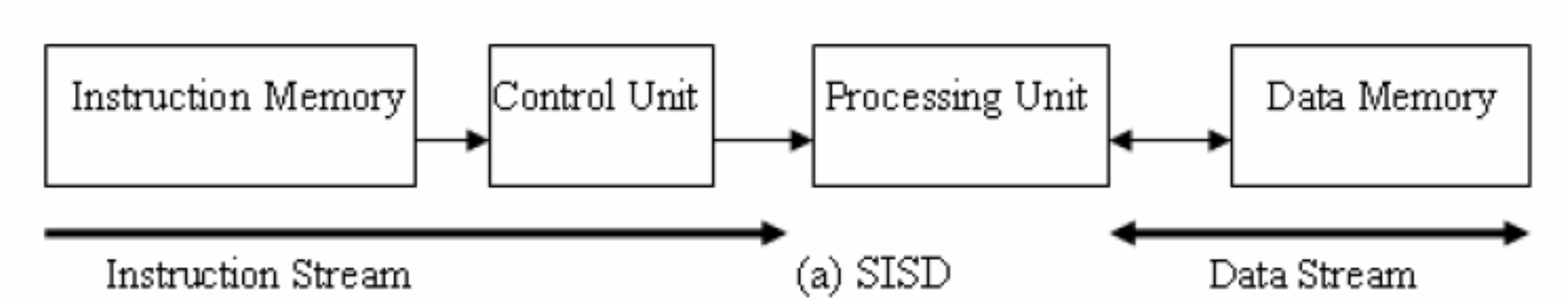
Flynn's Taxonomy

- Single Instruction Single Data (SISD)
  - Traditional sequential computing systems
- Single Instruction Multiple Data (SIMD)
- Multiple Instructions Multiple Data (MIMD)
- Multiple Instructions Single Data (MISD)



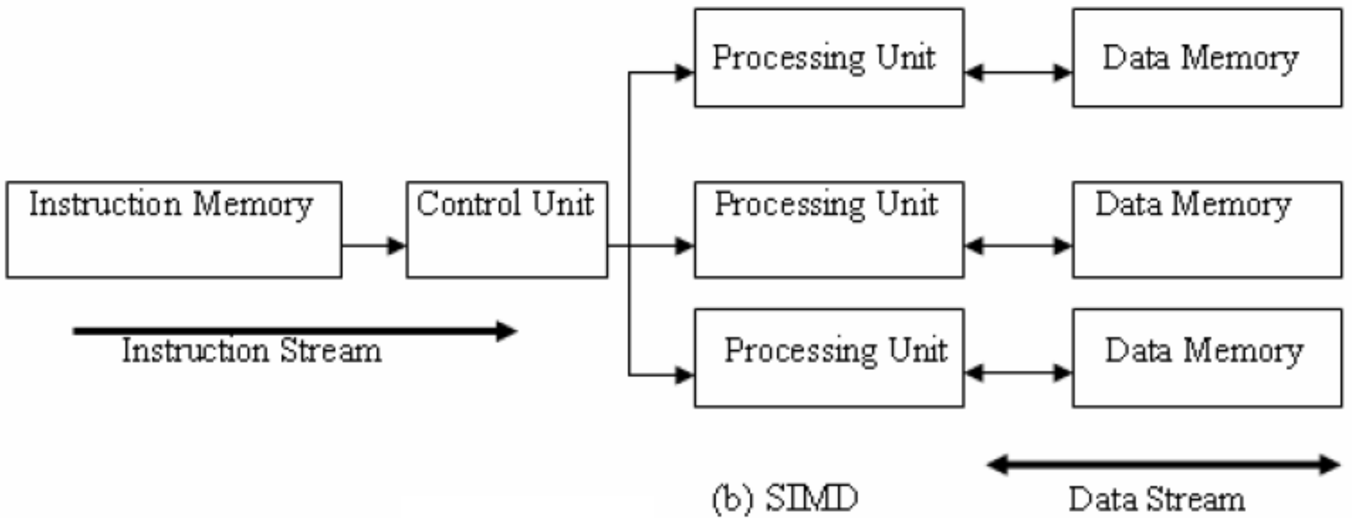
SISD

- At one time, one instruction operates on one data
- Traditional sequential architecture



SIMD

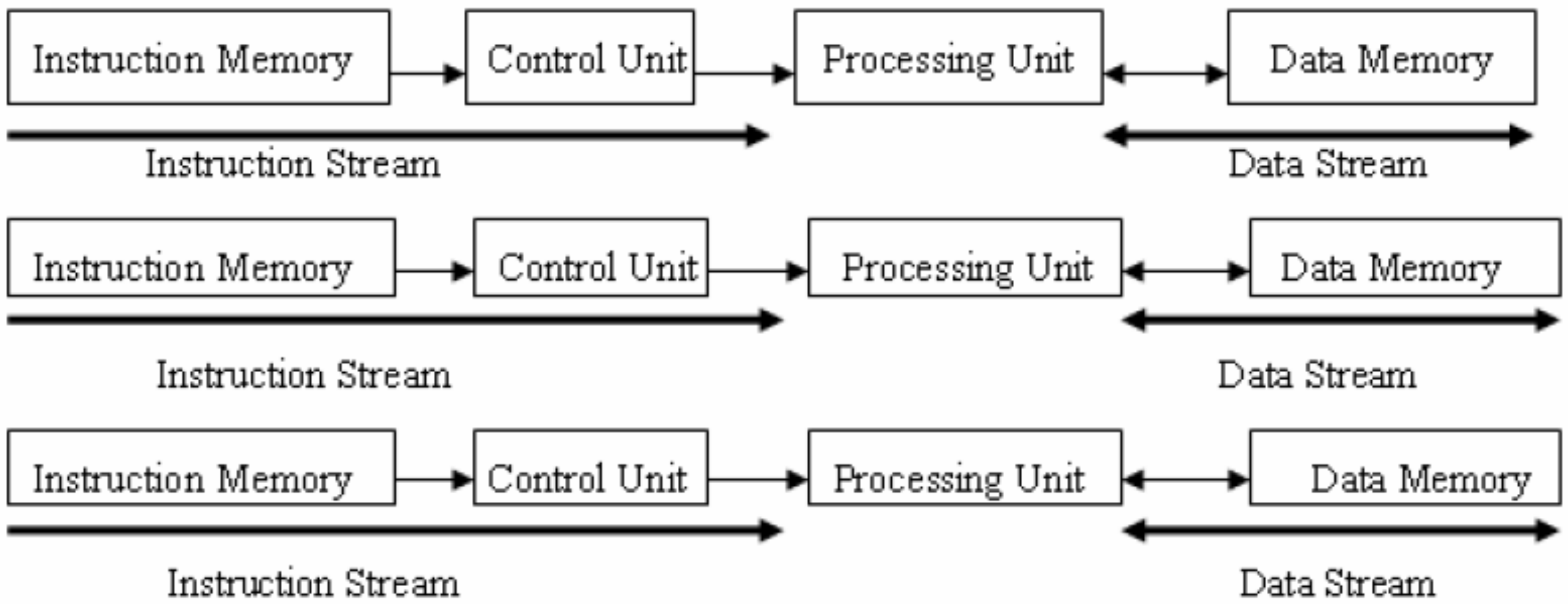
- **At one time, one instruction operates on many data**
  - Data parallel architecture
  - Vector architecture has similar characteristics, but achieve the parallelism with pipelining.
- **Array processors**



(b) SIMD

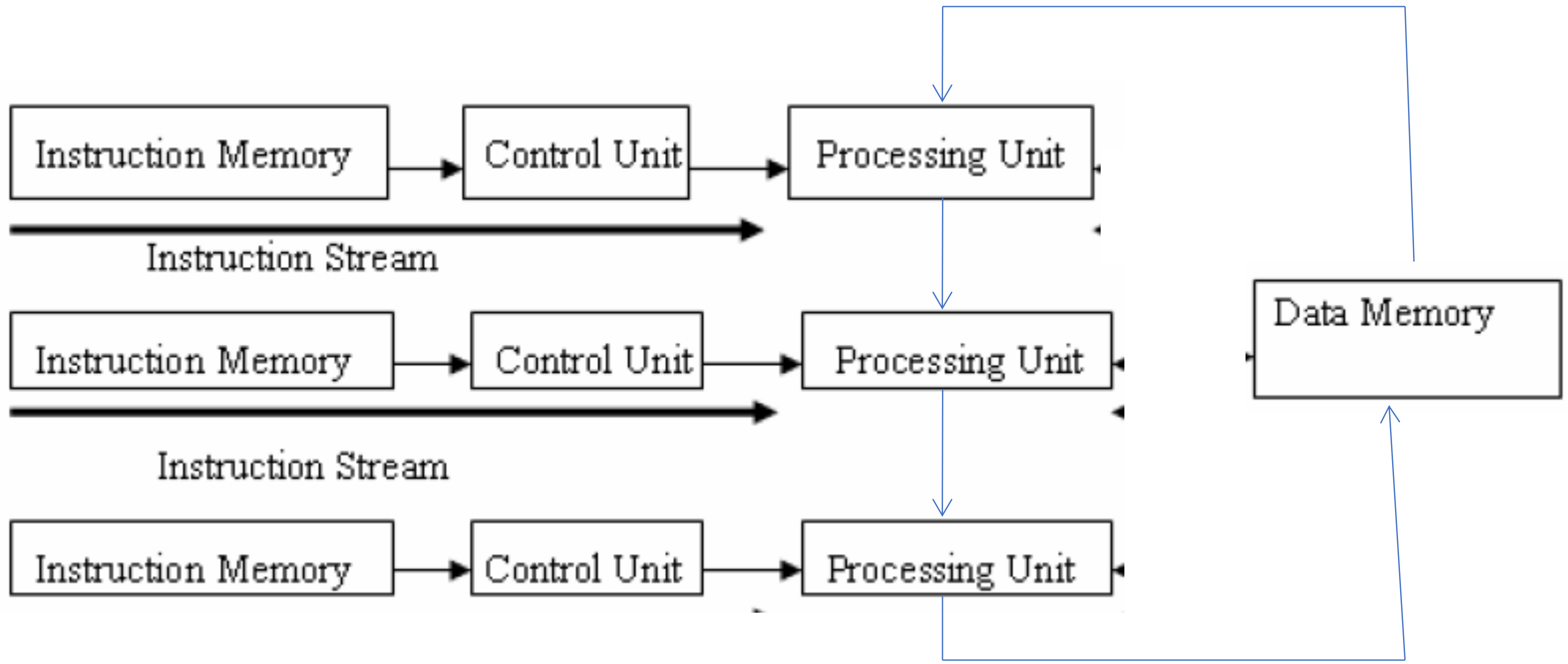
MIMD

- **Multiple instructions operating on multiple data**
  - Classical distributed memory or SMP architectures



MISD

- **Multiple instructions operating on single data**
  - Not commonly seen, unfortunately …



Flynn’s Taxonomy Summary

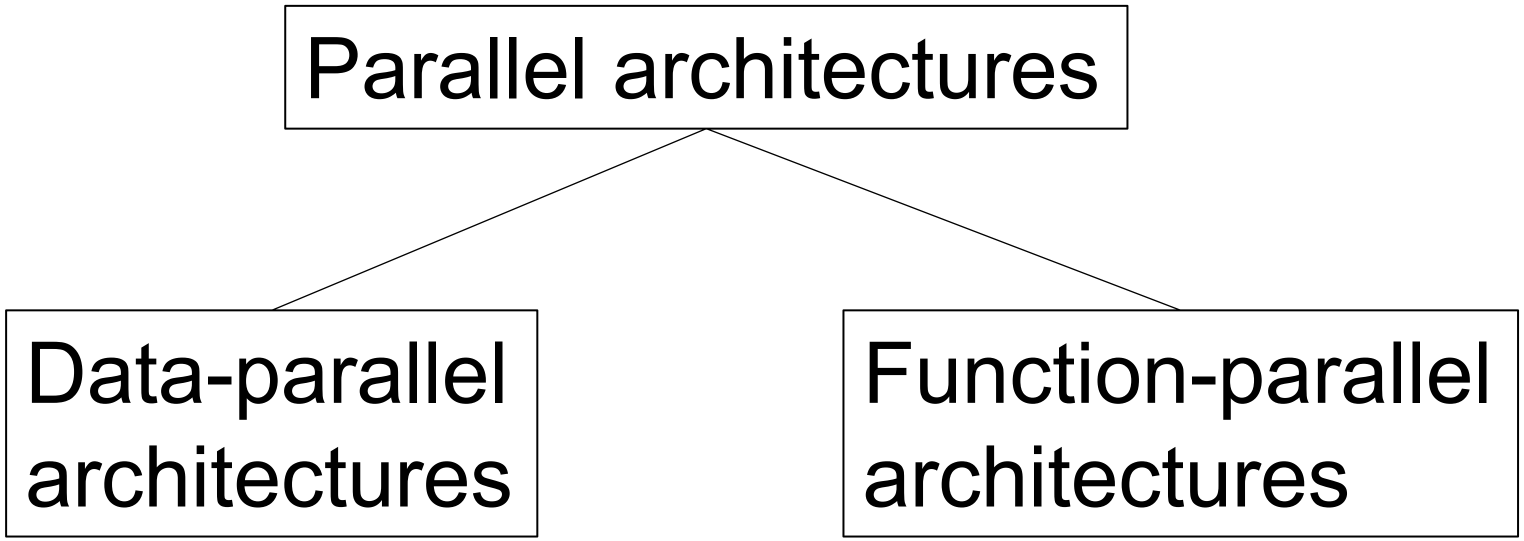
- **SISD: traditional sequential architecture**
- **SIMD: processor arrays, vector processor**
  - Parallel computing on a budget – reduced control unit cost
  - Many early supercomputers
- **MIMD: most general purpose parallel computer today**
  - Clusters, MPP, data centers
- **MISD: not a general purpose architecture**

Flynn’s Taxonomy on Architectures Today

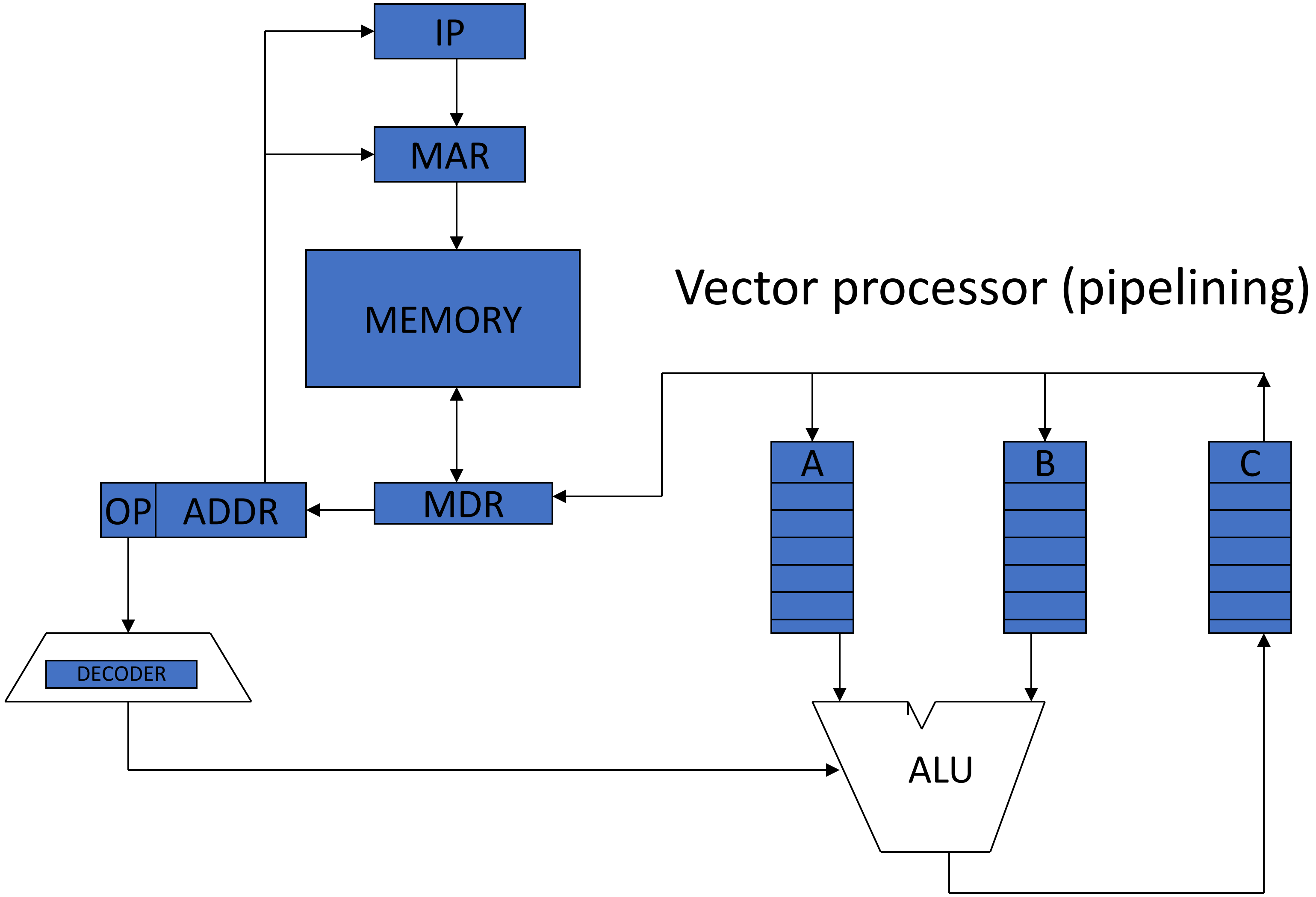
- **Manycore processors**
- **Superscalar:** Pipelined + multiple issues.
- **Streaming SIMD Extension (SSE)**
  - Intel and AMD’s support for performing operation on 2 doubles or 4 floats simultaneously
  - Now can handle 512-bits at a time
- **GPU: CUDA architecture**
- **IBM BlueGene**

Modern Classification

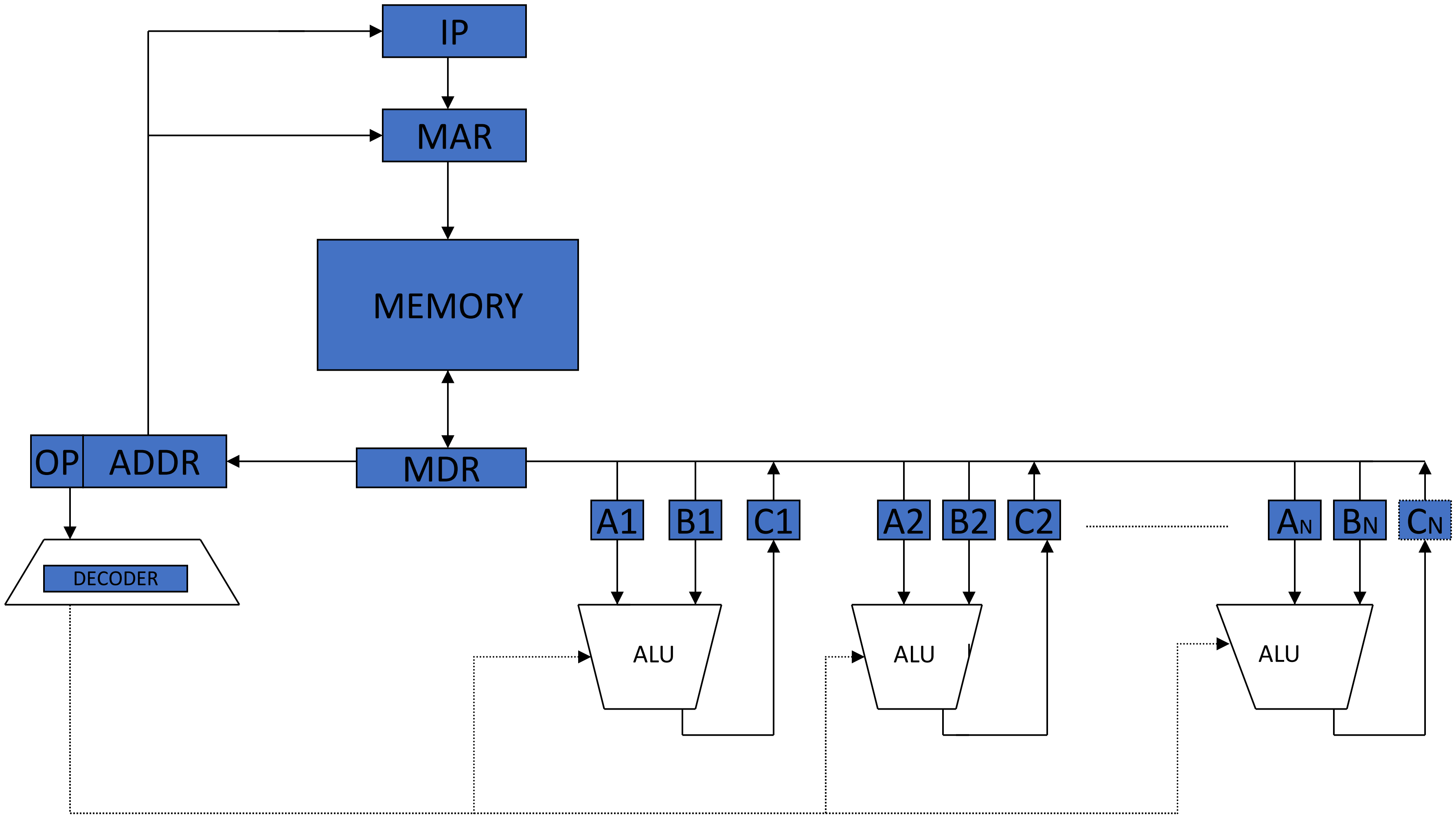
- **Classify based on how parallelism is achieved**
  - by operating on multiple data: data parallelism
  - by performing many functions in parallel: function parallelism
    - Control parallelism, task parallelism depending on the level of the functional parallelism.



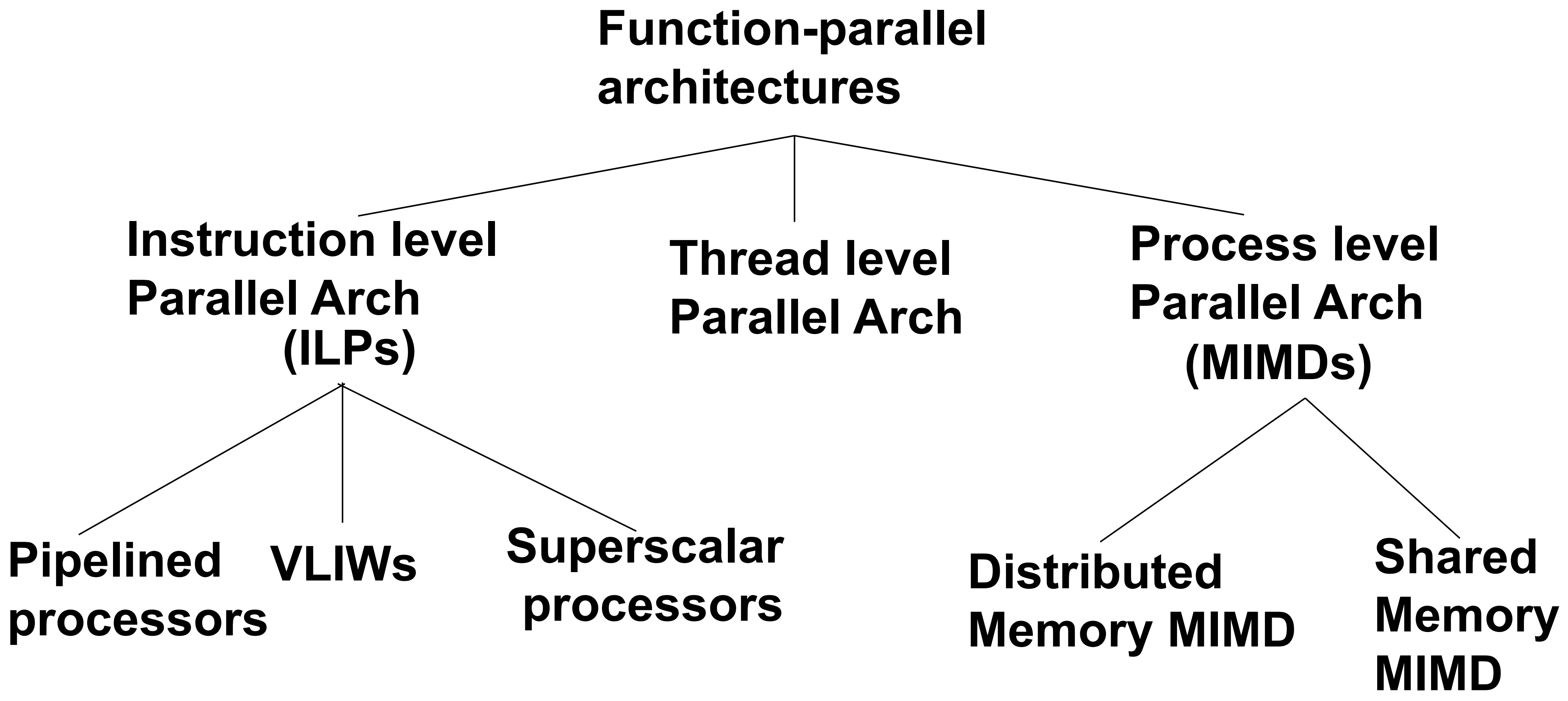
Data-parallel Architecture: Vector



Data-parallel Architecture: Array



Function-parallel Architecture



Performance of Parallel Architectures

- **Common metrics**
  - MIPS: million instructions per second
    - MIPS = instruction count/(execution time x 10^6)
  - MFLOPS: million floating point operations per second.
    - MFLOPS = FP ops in program/(execution time x 10^6)
- **Which is a better metric?**
    - FLOP is more related to the time of a task in numerical code
      - # of FLOP / program is determined by the matrix size

FLOPS Conventions

- FLOPS units
  - kiloFLOPS (KFLOPS) 10^3
  - megaFLOPS (MFLOPS) 10^6
  - gigaFLOPS (GFLOPS) 10^9 ← single CPU performance
  - teraFLOPS (TFLOPS) 10^12

  - petaFLOPS (PFLOPS) 10^15 ← we are here right now
    - 150 petaFLOPS supercomputers

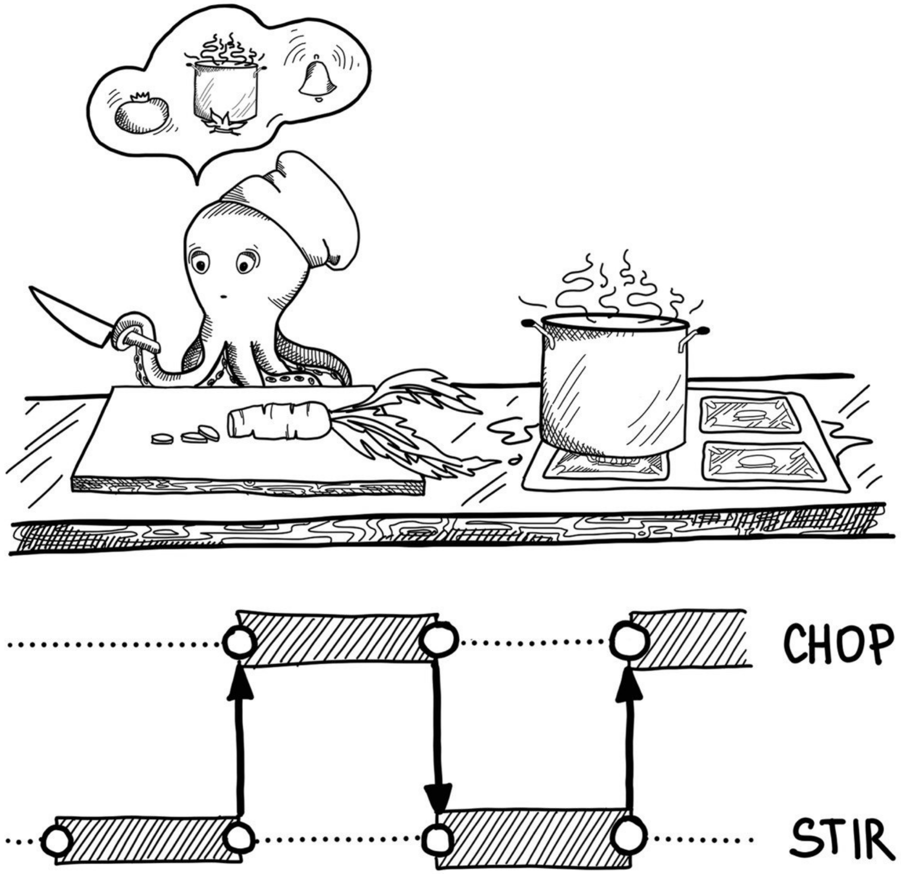
  - exaFLOPS (EFLOPS) 10^18 ← the next milestone

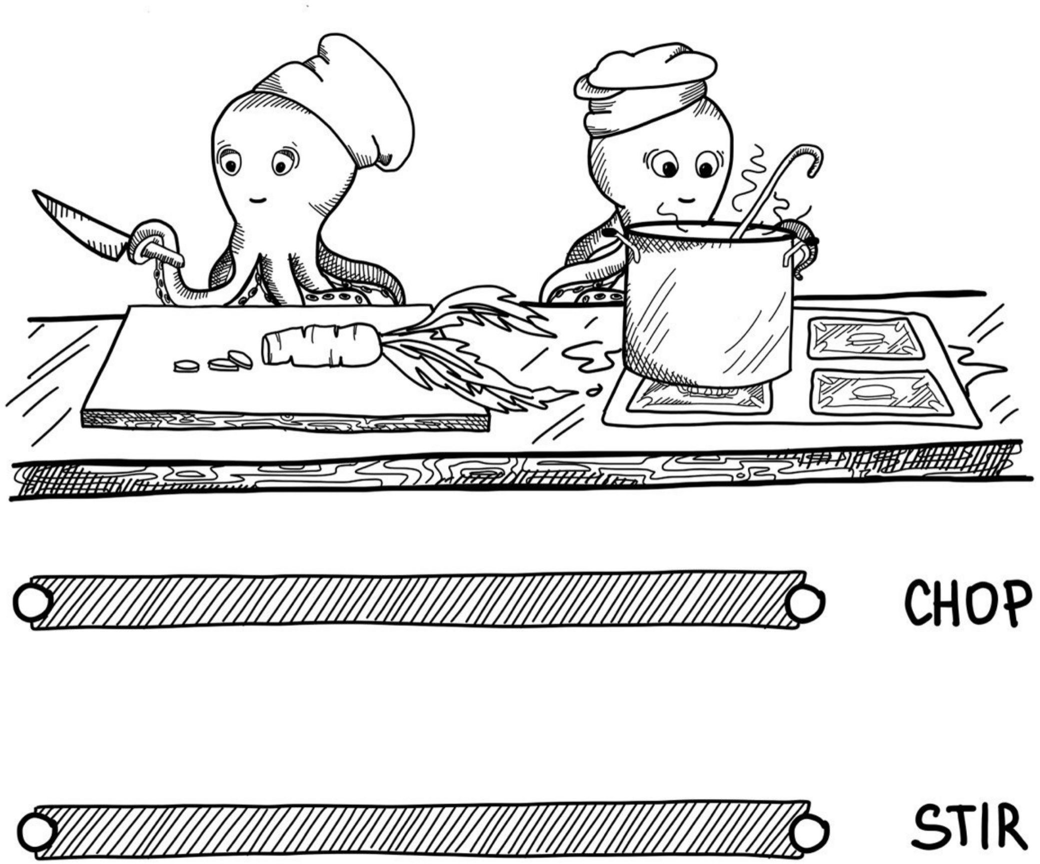
FLOP Widely Used in Benchmarks

- **Micro benchmarks suit**
  - Numerical computing
    - LAPACK
    - ScaLAPACK
  - Memory bandwidth
    - STREAM
- **Kernel benchmarks**
  - NPB (NAS parallel benchmark)
  - PARKBENCH
  - SPEC
  - Splash

Software Parallelism

- **Concurrency** (multi-tasking) vs **Parallelism** (collaboration)



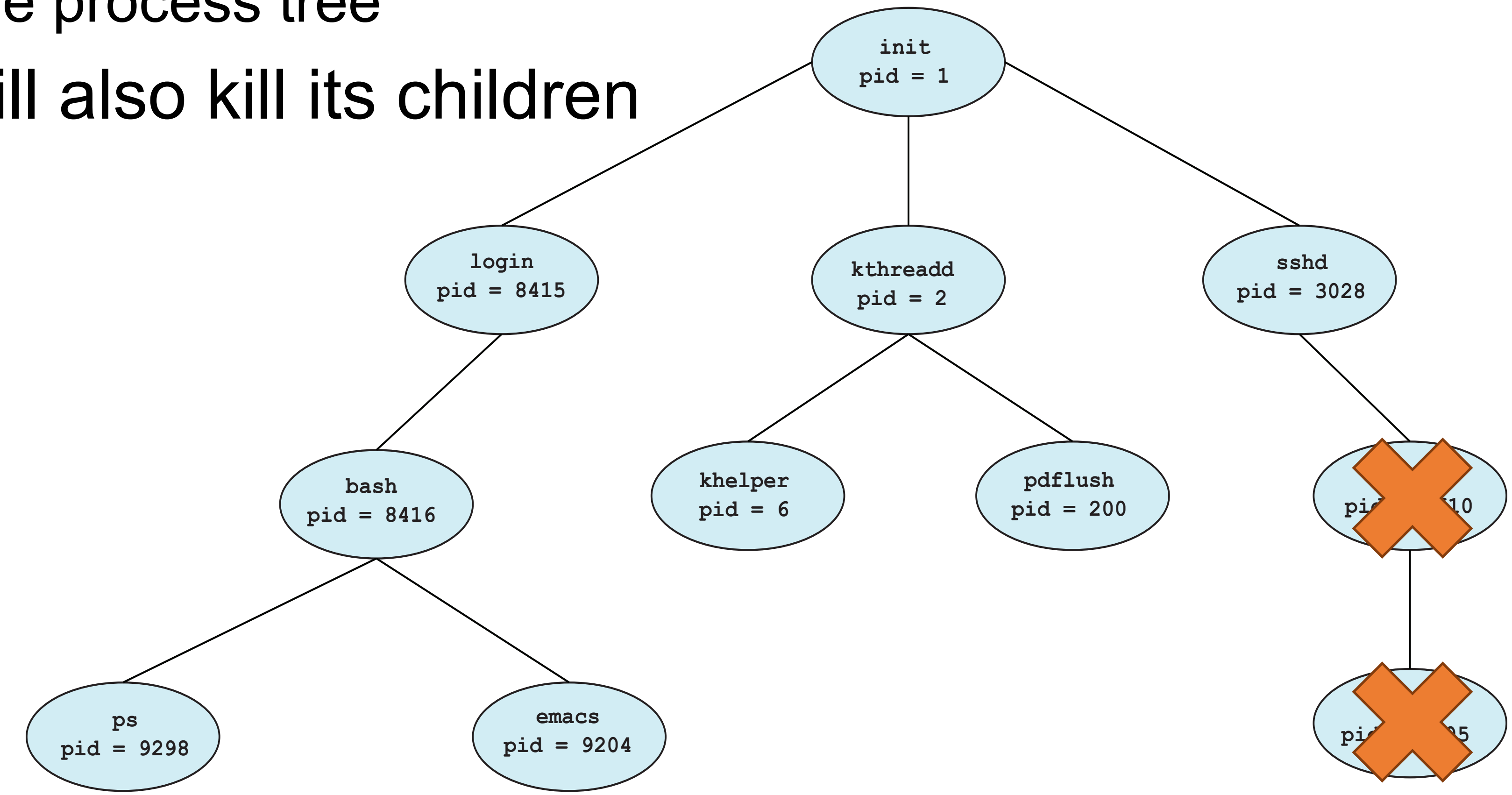


Process (Program)

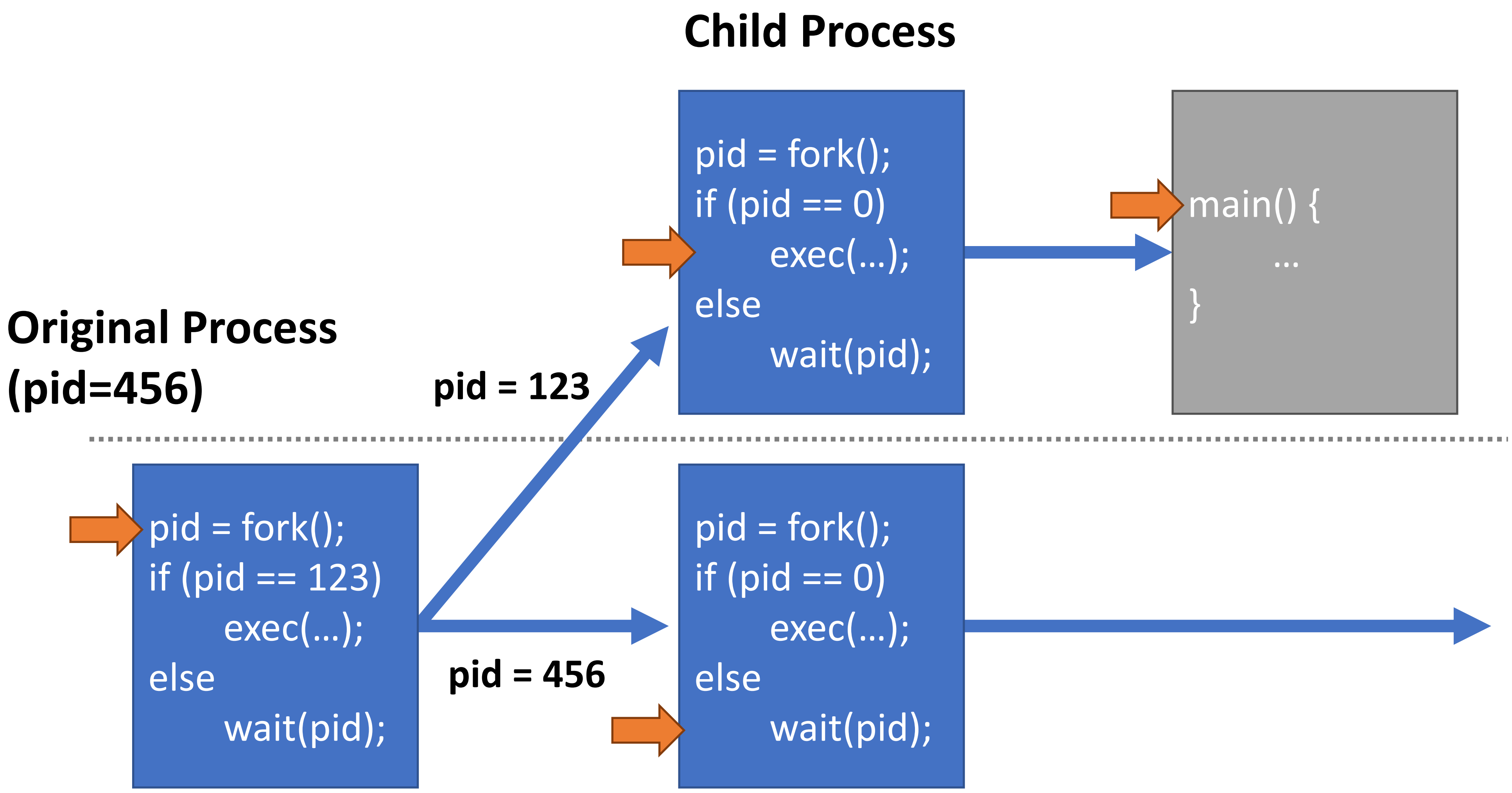
- A process is an instance of a computer program
- **On Unix/Linux platforms, all processes have parents**
  - i.e. which process executed this new process?
- If a process spawns other processes, they become it’s children
  - This creates a tree of processes
- If a parent exits before its children, the children become orphans
- If a child exits before the parent calls wait(), the child becomes a zombie

Process Tree

- init is a special process started by the kernel
  - Always roots the process tree
- Kill a process will also kill its children

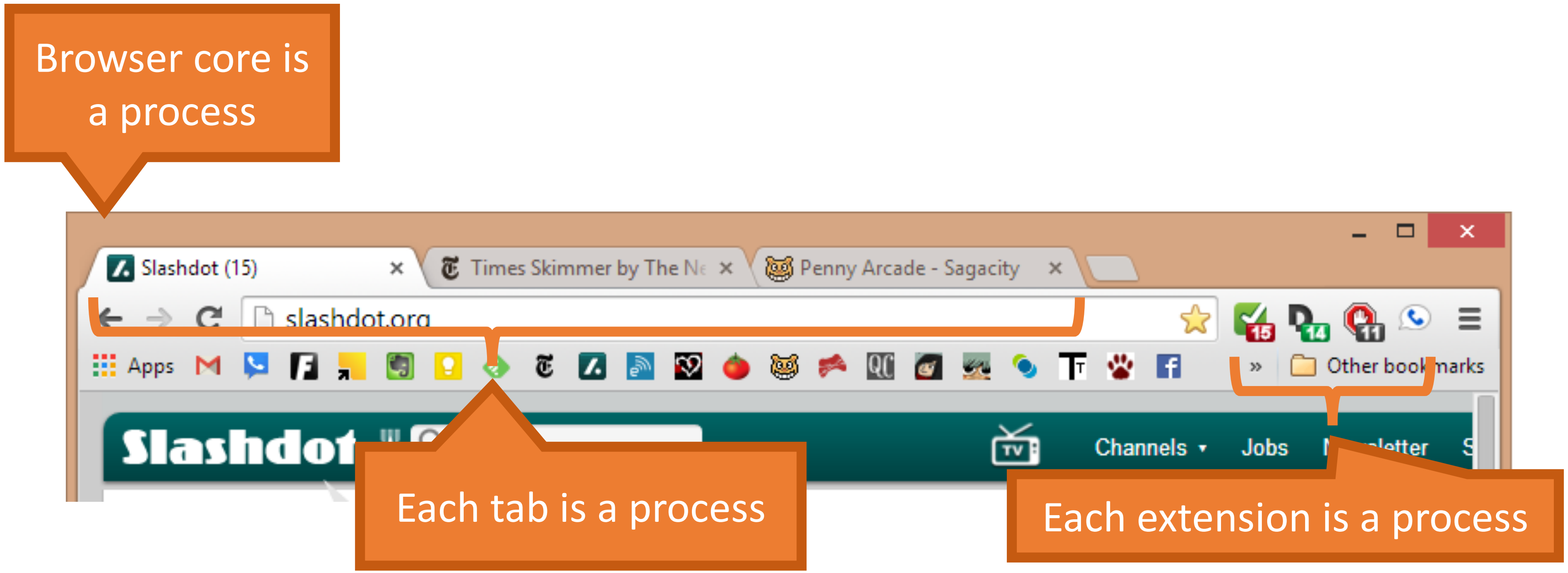


Linux Process Management



Process Properties

- We can load programs as processes
- We can context switch between processes
- Processes are protected from each other



Are Processes Enough?

- **At this point, we have the ability to run processes**
  - And processes can communicate with each other through inter-process communication (IPC)
- **Is this enough functionality?**
  - Can we just do many processes to perform parallelism?
    - Ex: Python multi-processing
- **Possible scenarios:**
  - A large server with many clients
  - A powerful computer with many CPU cores

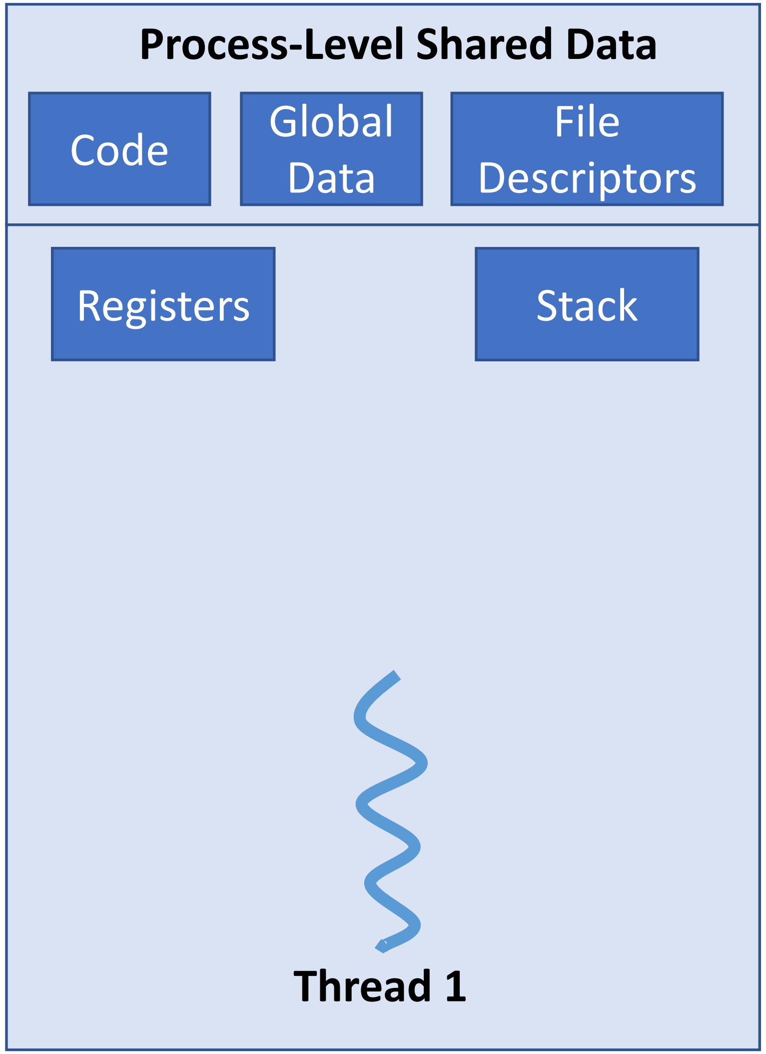
Problems with Processes

- **Process creation is heavyweight (i.e. slow)**
  - Space must be allocated for the new process
  - fork() copies all state of the parent to the child
- **IPC mechanisms are cumbersome**
  - Difficult to use fine-grained synchronization
  - Message passing is slow
    - Each message may have to go through the kernel

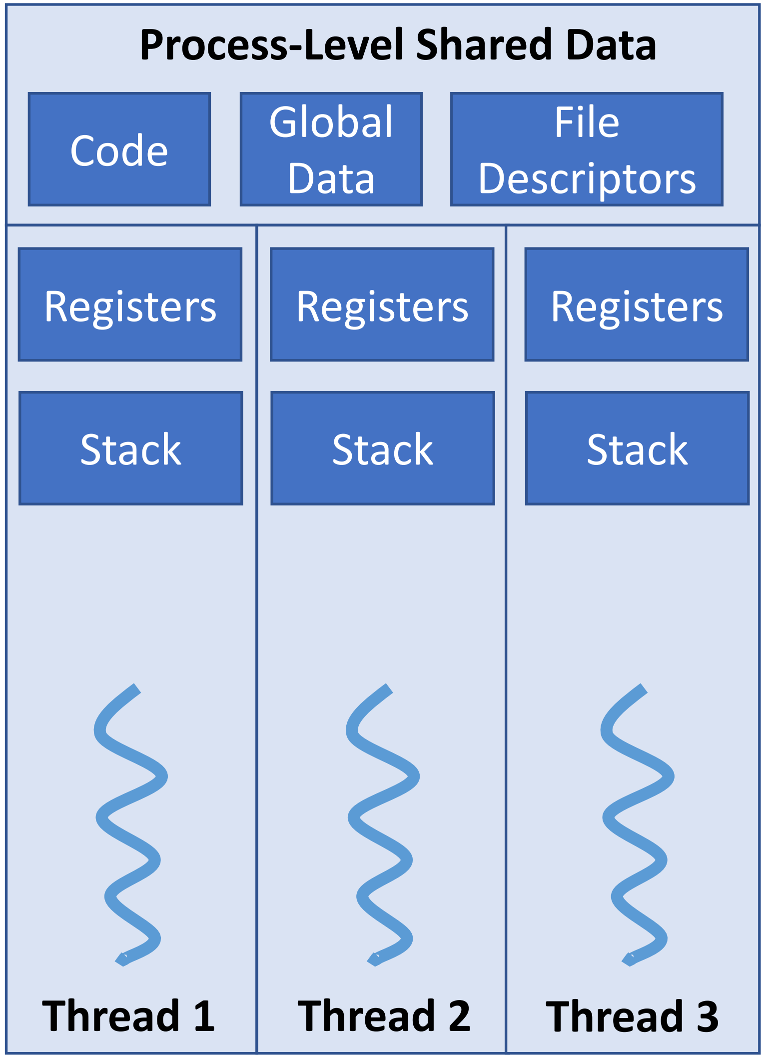
Threads

- Light-weight execution units that share the same memory and state space **within** a process
- Every process has at least one thread, i.e., a thread of itself
- Benefits:
  - Resource sharing, no need for IPC
  - Economy: faster to create, faster to context switch
  - Scalability: simple to take advantage of multi-core CPUs

Single-Threaded Process



Multi-Threaded Process



Thread Implementation

- **POSIX standard API for thread creation**
  - IEEE 1003.1c
  - *Specification*, not *implementation*
    - Defines the API and the expected behavior
    - … but not how it should be implemented
- **Implementation is system dependent**
  - On some platforms, user-level threads
  - On others, maps to kernel-level threads
- **Starting in C++11, library includes thread implementation**
  - We will use C++ thread heavily in this class

Summary

- **Flynn's classification on hardware parallelism**
  - SISD, SIMD, MIMD, MISD
- **Modern classification**
  - Data parallelism
  - function parallelism
- **Performance**
  - MIPS, MFLOPS
- **Software parallelism**
  - Concurrency vs Parallelism
  - Process and Thread