# Parallel Programming

## Introduction to Parallel Hardware and Software

Professor Yi-Ping You (游逸平)
Department of Computer Science
http://www.cs.nctu.edu.tw/~ypyou/

# Outline

- Uniprocessor Parallelism
  - Pipelining, Out-of-order execution, Superscalar, VLIW
  - Vector Processing/SIMD
  - Multithreading
  - Uniprocessor Memory Systems
- Explicit Parallel Computer Architectures
  - Shared-memory Architectures
  - Distributed-memory Architectures
- Parallel Software
  - Shared-memory model
  - Distributed-memory model

# Parallelism is Everywhere

- Modern Processor Chips have ≈ 1 billion transistors
  - Clearly must get them working in parallel
  - Question: how much of this parallelism must programmer understand?
- How do uniprocessor computer architectures extract parallelism?
  - By finding parallelism within instruction stream
  - Called “Instruction Level Parallelism” (ILP)
- Goal of Computer Architects until about 2002:
  - Hide underlying parallelism from everyone (sort of)
    - OS, Compiler, Programmer

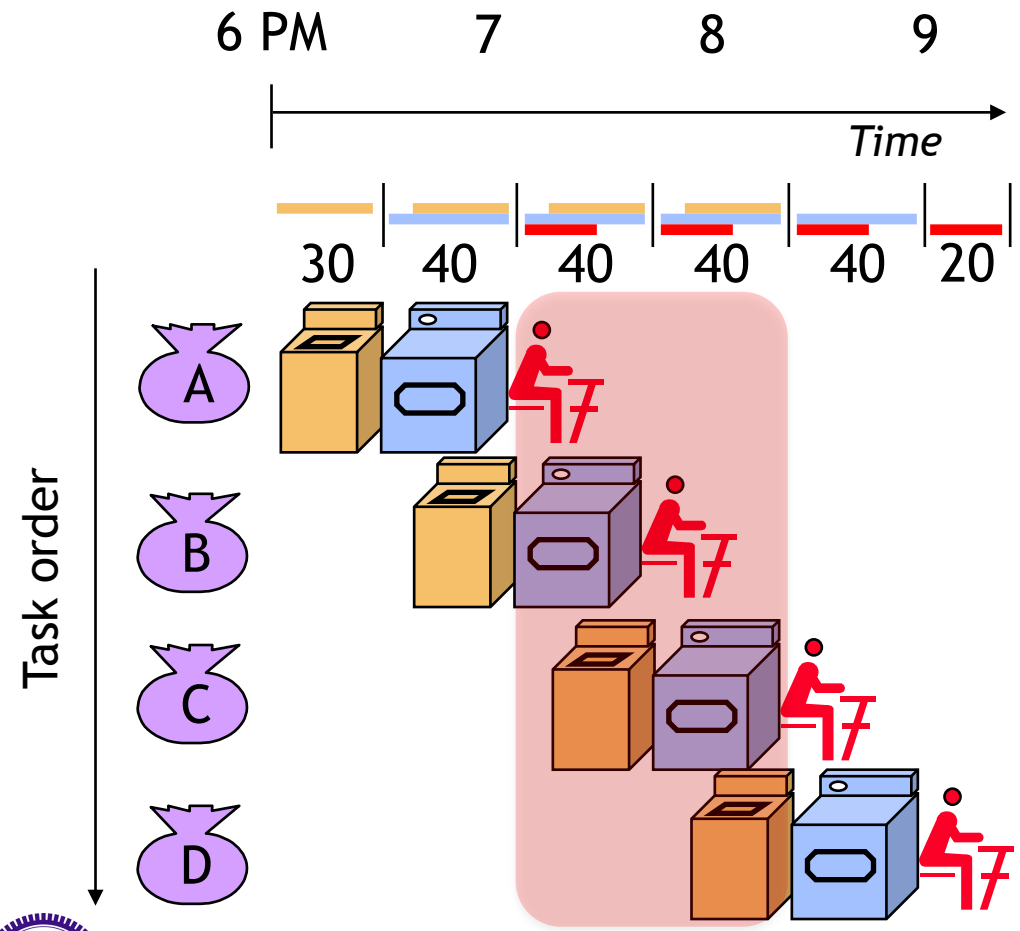
# Examples of ILP Techniques

- Pipelining: overlapping individual parts of instructions
- Superscalar execution: do multiple things at same time
- VLIW: Let compiler specify which operations can run in parallel
- Vector Processing: Specify groups of similar (independent) operations
- Out of Order Execution (OOO): Allow long operations to happen

# What is Pipelining?

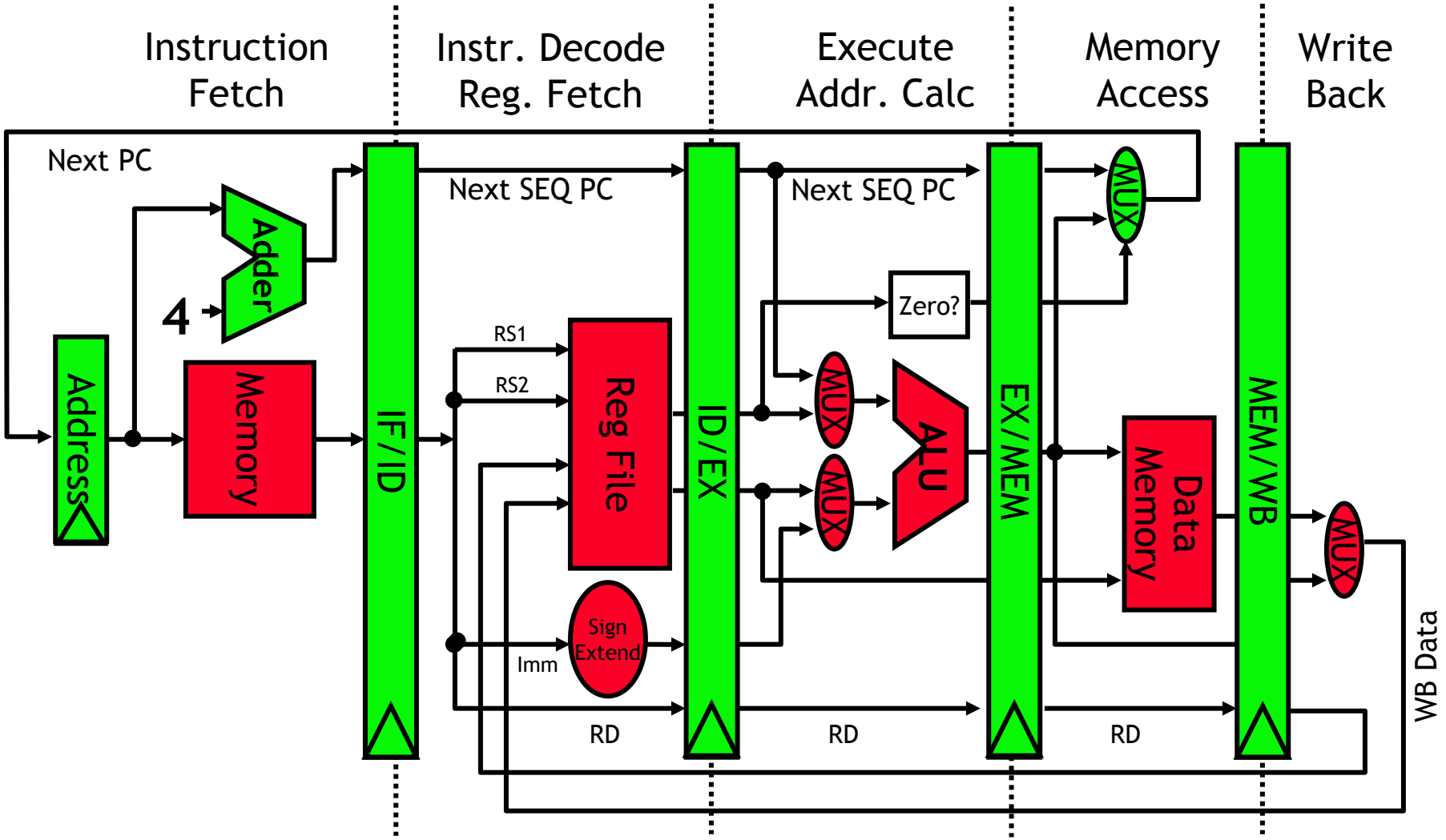
David Patterson's Laundry example: 4 people doing laundry

wash (30 min) + dry (40 min) + fold (20 min) = 90 min Latency

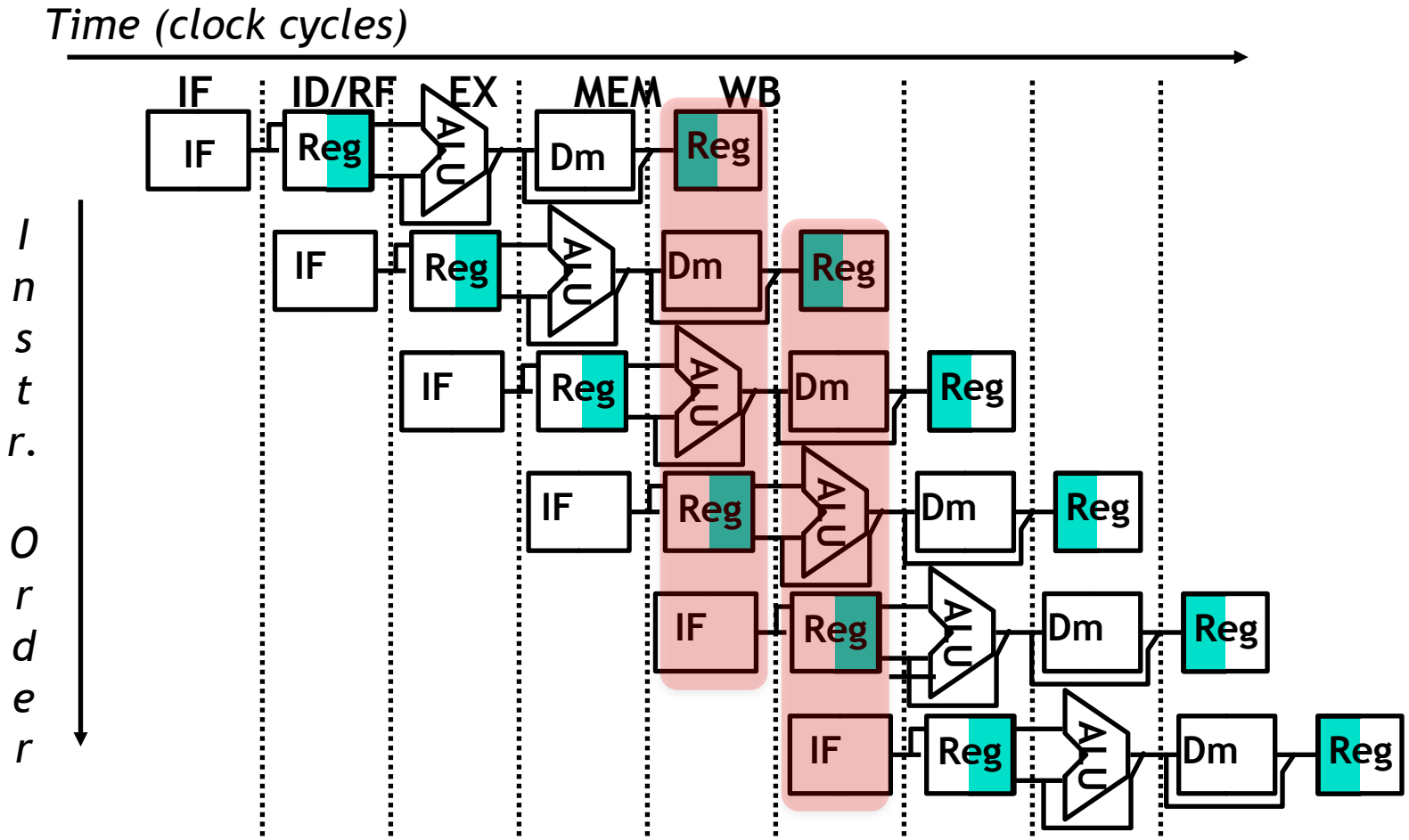



- In this example:
  - Sequential execution takes 4 * 90min = 6 hours
  - Pipelined execution takes 30+4*40+20min = 3.5 hours
- Throughput = loads/hour
  - TP = 4/6 l/h w/o pipelining
  - TP = 4/3.5 l/h w pipelining
- Pipelining helps throughput but not latency (90 min)
- Throughput limited by slowest pipeline stage
- Potential speedup = Number of pipe stages

# Example: 5 Steps of MIPS Pipeline



# Visualizing The Pipeline

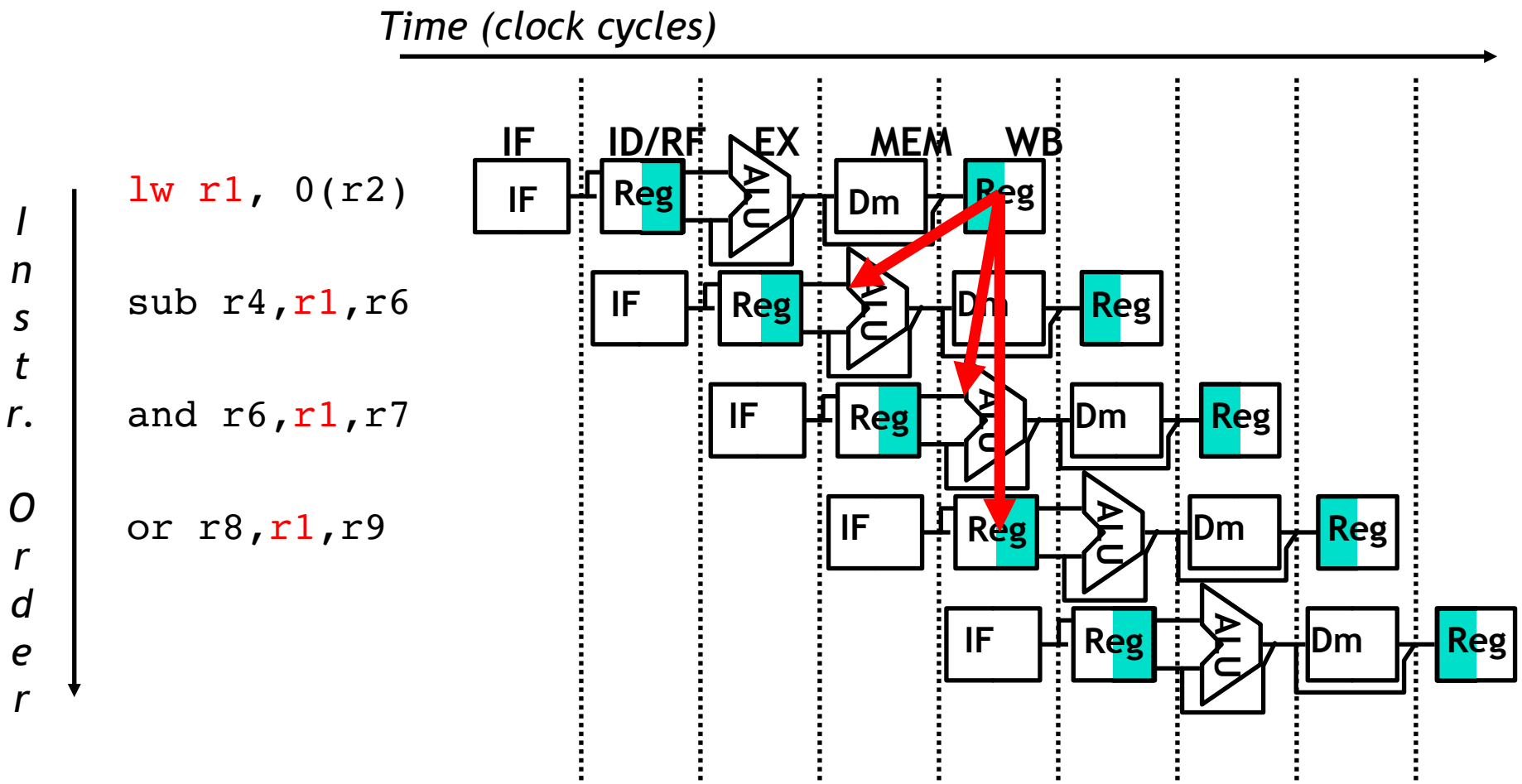



- In ideal case: CPI (cycles/instruction) = 1!
  - On average, put one instruction into pipeline, get one out
- Superscalar: launch more than one instruction/cycle (more functional units)
  - In ideal case, CPI < 1

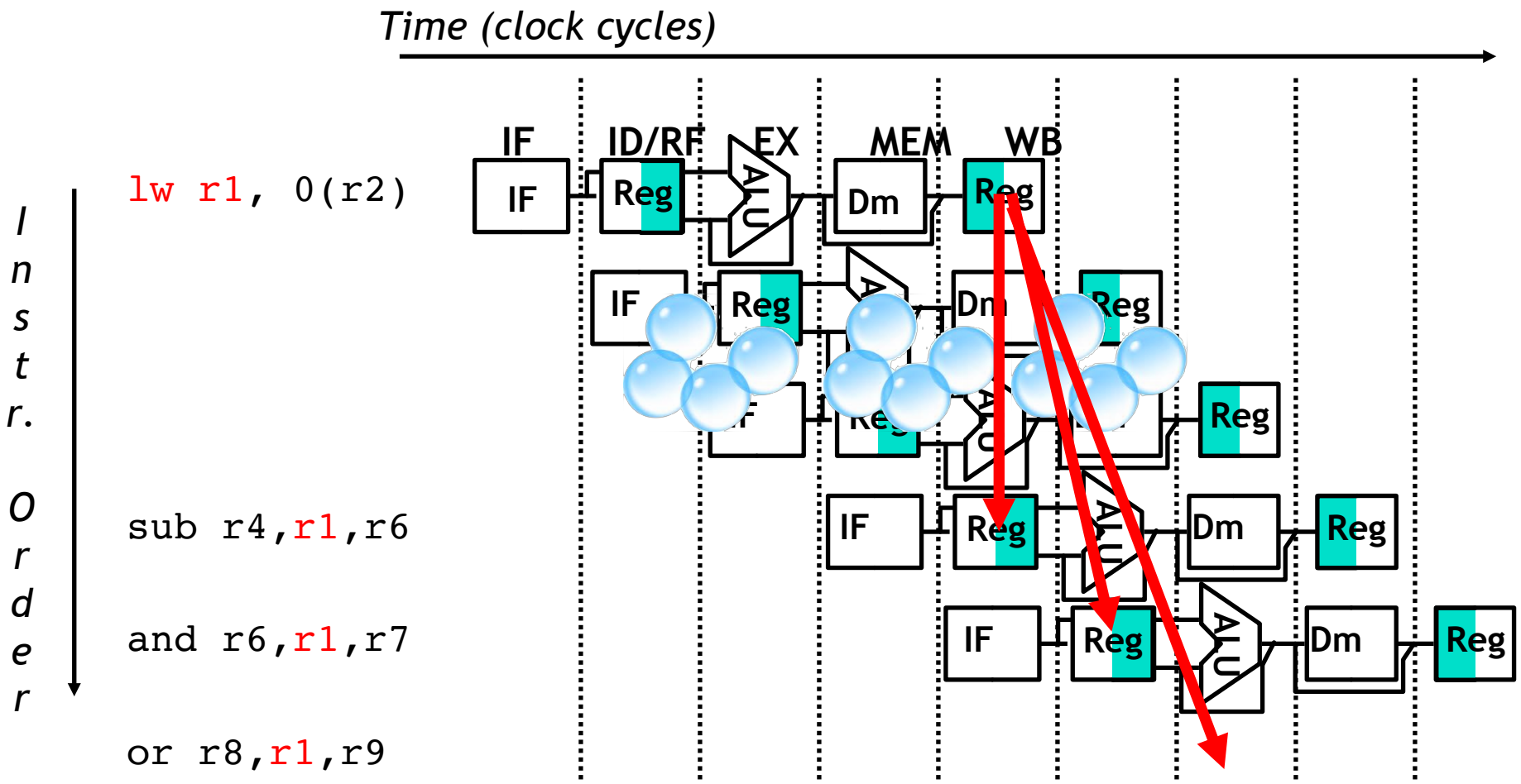
# Limits to Pipelining

- Hazards prevent next instruction from executing during its designated clock cycle
  - Structural hazards
    - Attempt to use the same hardware to do two different things at once
  - Data hazards
    - Instruction depends on result of prior instruction still in the pipeline
  - Control hazards
    - Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps)

# Example of Data Hazard



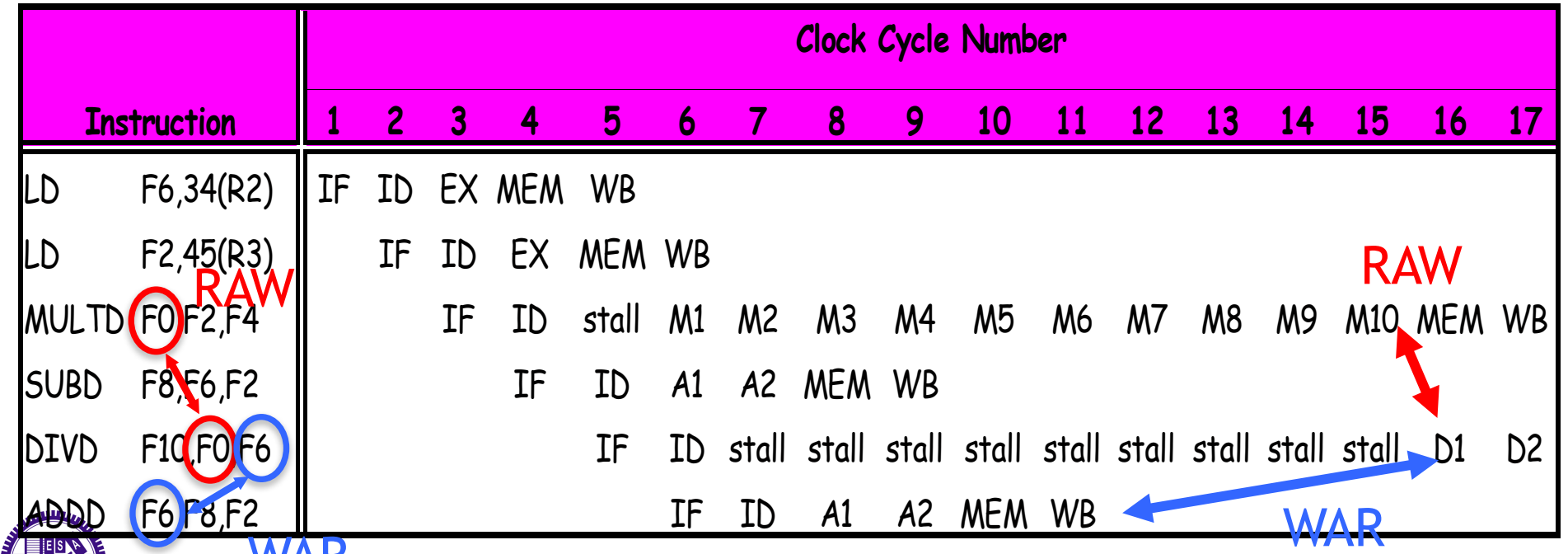
# Example of Data Hazard




- Data dependencies between adjacent instructions
  - Must wait ("stall") for result to be done (No "back in time" exists!)
  - Net result is that CPI > 1
- Superscalar increases frequency of hazards

# Types of Data Hazards

- Read-after-write (RAW) dependencies
  - Flow dependencies (true dependencies)
  - Cannot be abandoned
- Write-after-read (WAR) dependencies
  - Anti-dependencies (false dependencies)
  - Can be eliminated through register renaming

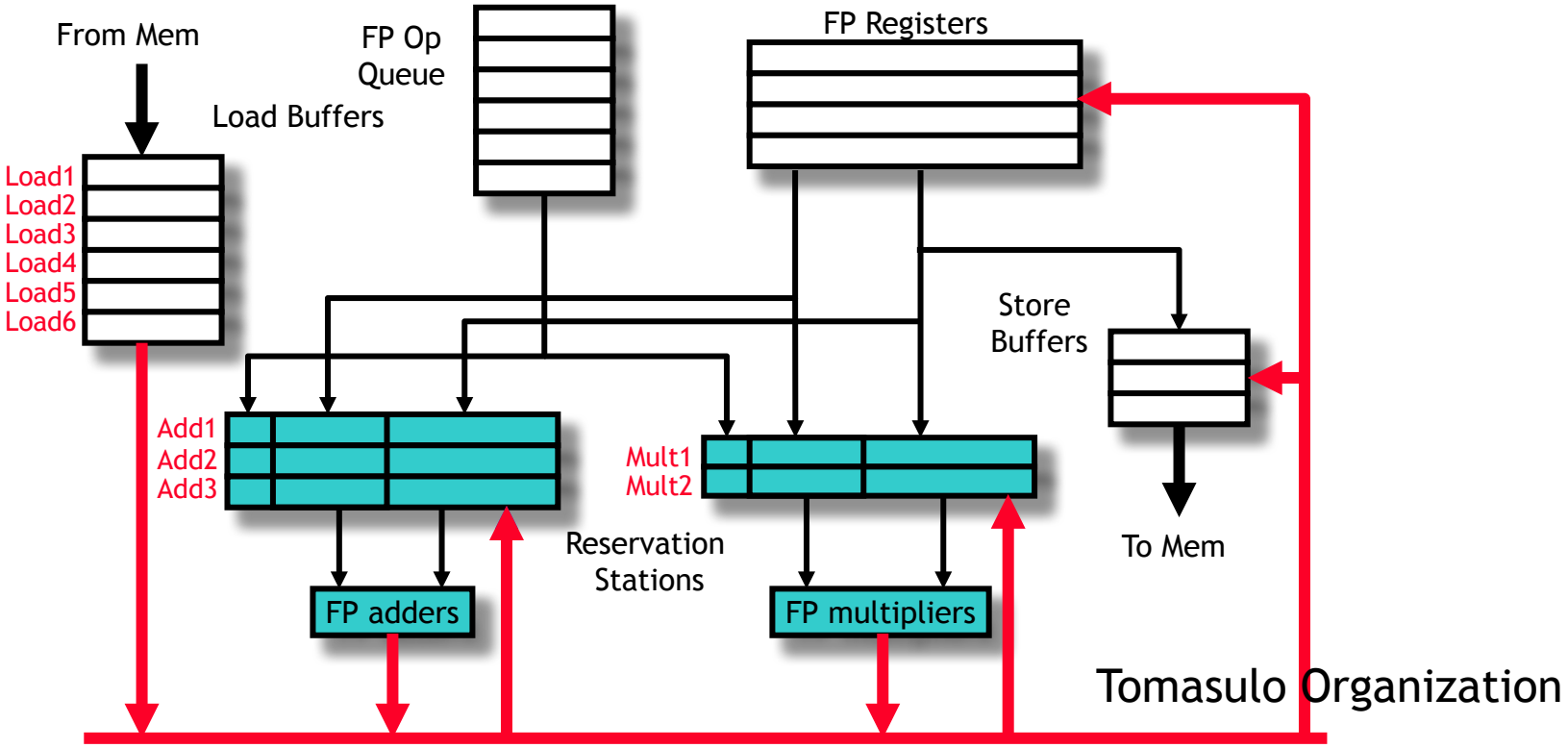


# Out-of-Order (OOO) Execution

- Key idea: Allow instructions behind stall to proceed

```
DIVD   F0,F2,F4
ADDD   F10,F0,F8
SUBD   F12,F8,F14
```

- Out-of-order execution => out-of-order completion

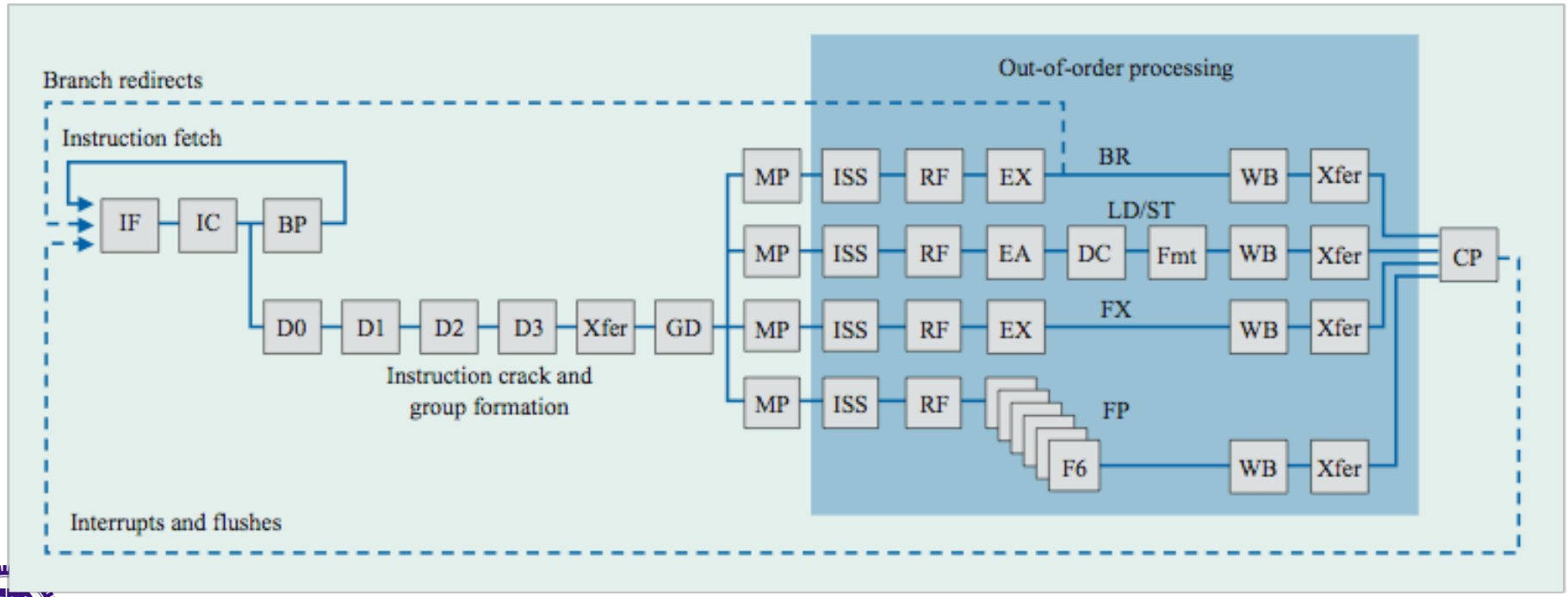


# Modern ILP

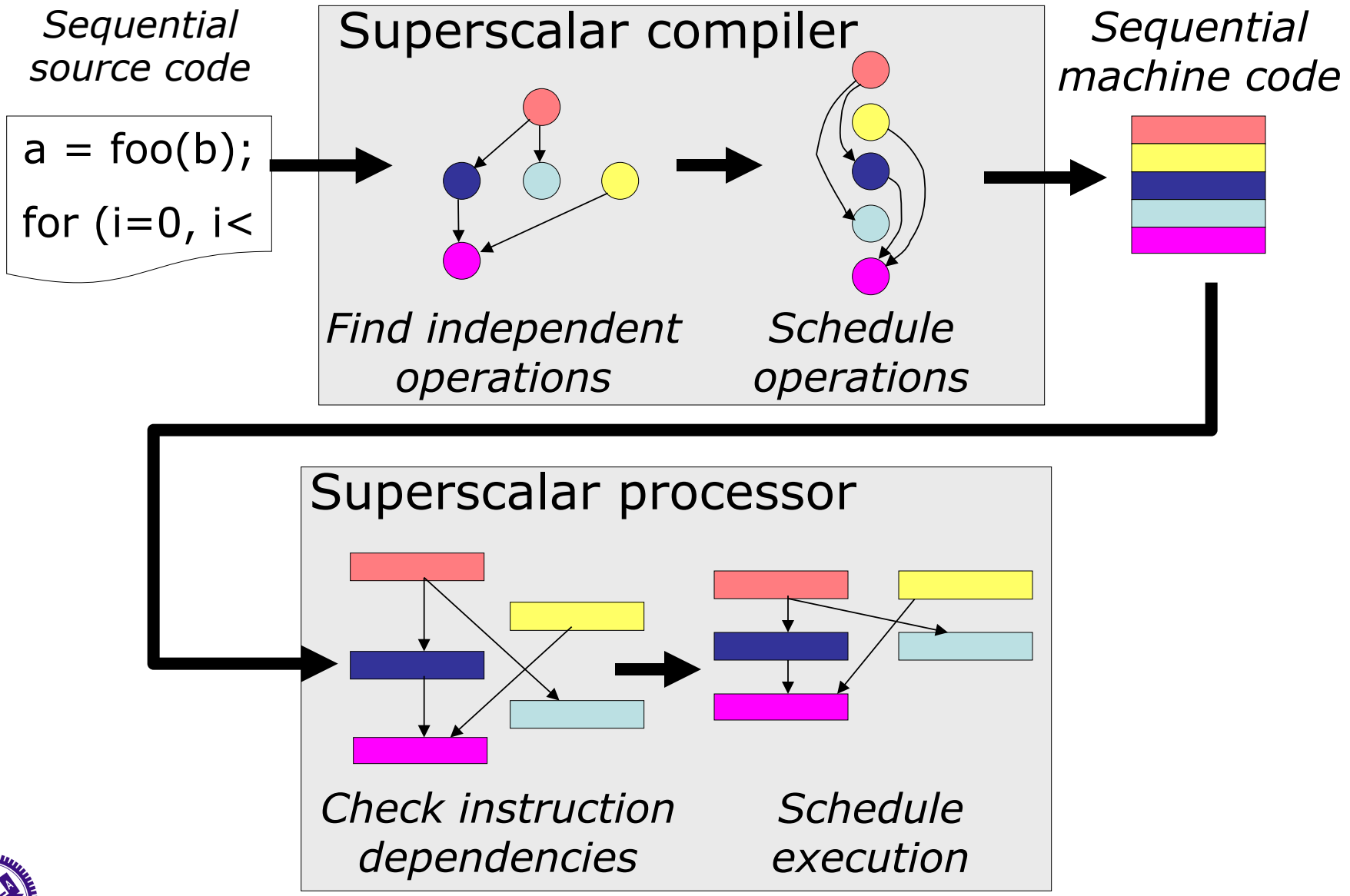
- Dynamically scheduled, out-of-order execution
  - Current microprocessors fetch 6-8 instructions per cycle
  - Pipelines are 10s of cycles deep many overlapped instructions in execution at once, although work often discarded
- What happens:
  - Grab a bunch of instructions, determine all their dependences, eliminate dep's wherever possible, throw them all into the execution unit, let each one move forward as its dependences are resolved
  - Appears as if executed sequentially
- Dealing with Hazards: May need to guess!
  - Called "Speculative Execution"
  - Speculate on Branch results, Dependencies, even Values!
  - If correct, don't need to stall for result => yields performance
  - If not correct, waste time and power
  - Must be able to UNDO a result if guess is wrong
  - Problem: accuracy of guesses decreases with number of simultaneous instructions in pipeline

# IBM Power 4

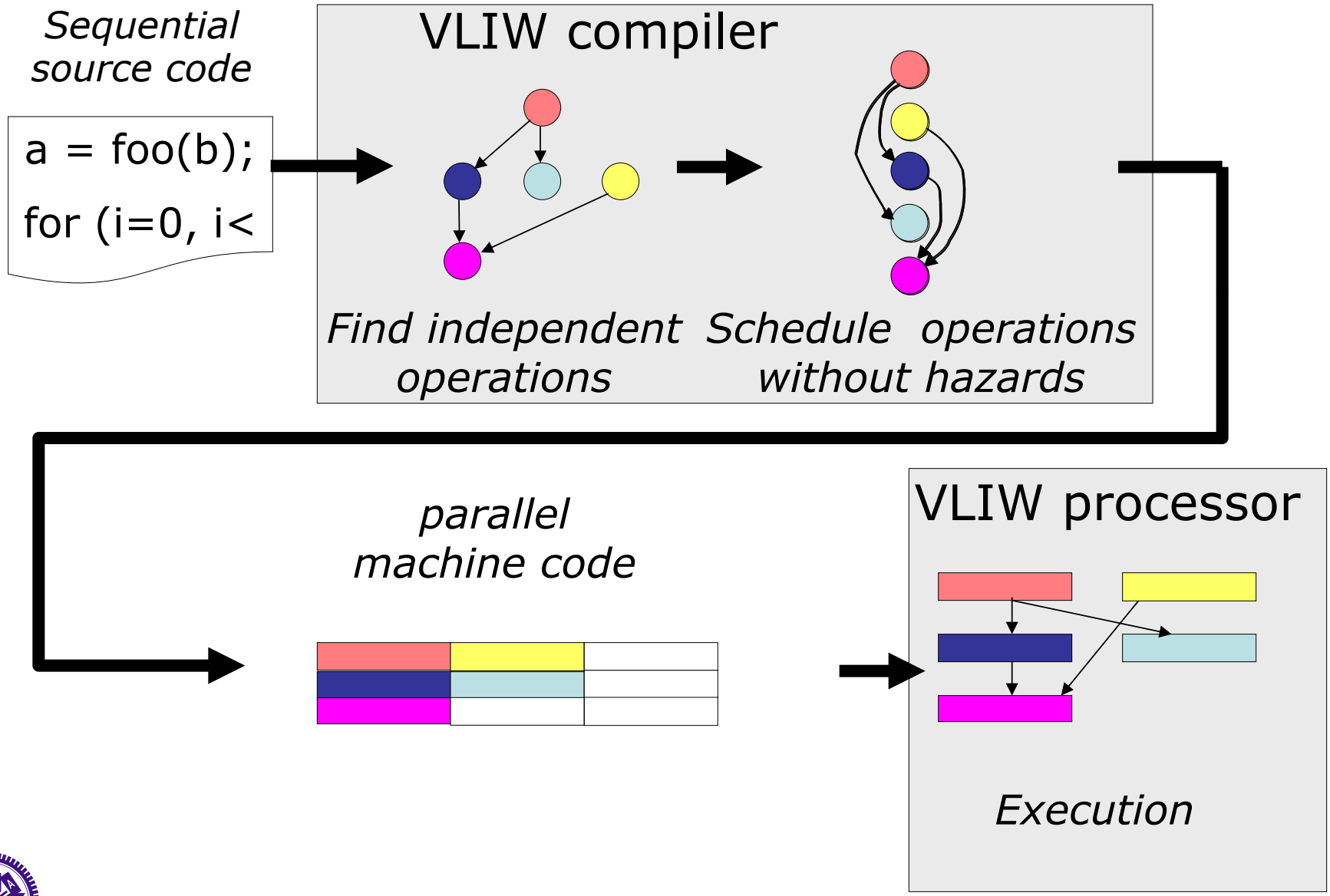
- Combines: Superscalar and OOO
- Properties:
  - 8 execution units in out-of-order engine, each may issue an instruction each cycle
  - In-order Instruction Fetch, Decode (compute dependencies)

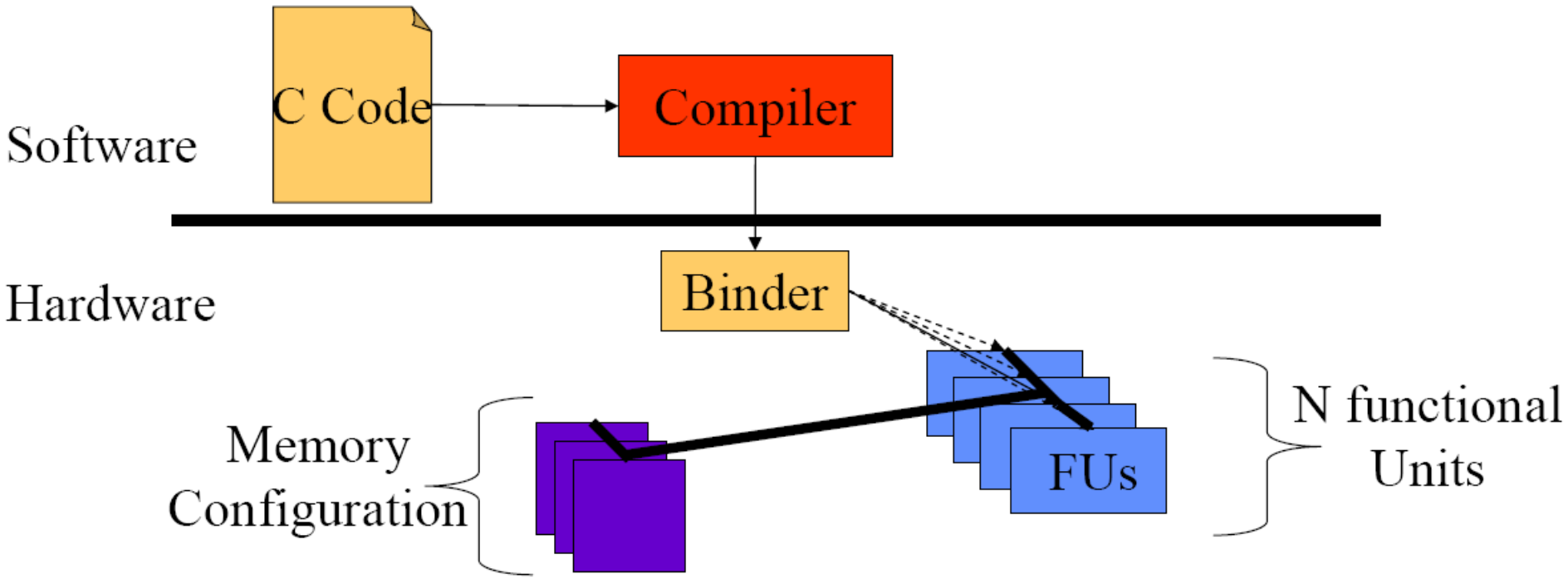
# Superscalar Architecture



# VLIW Architecture

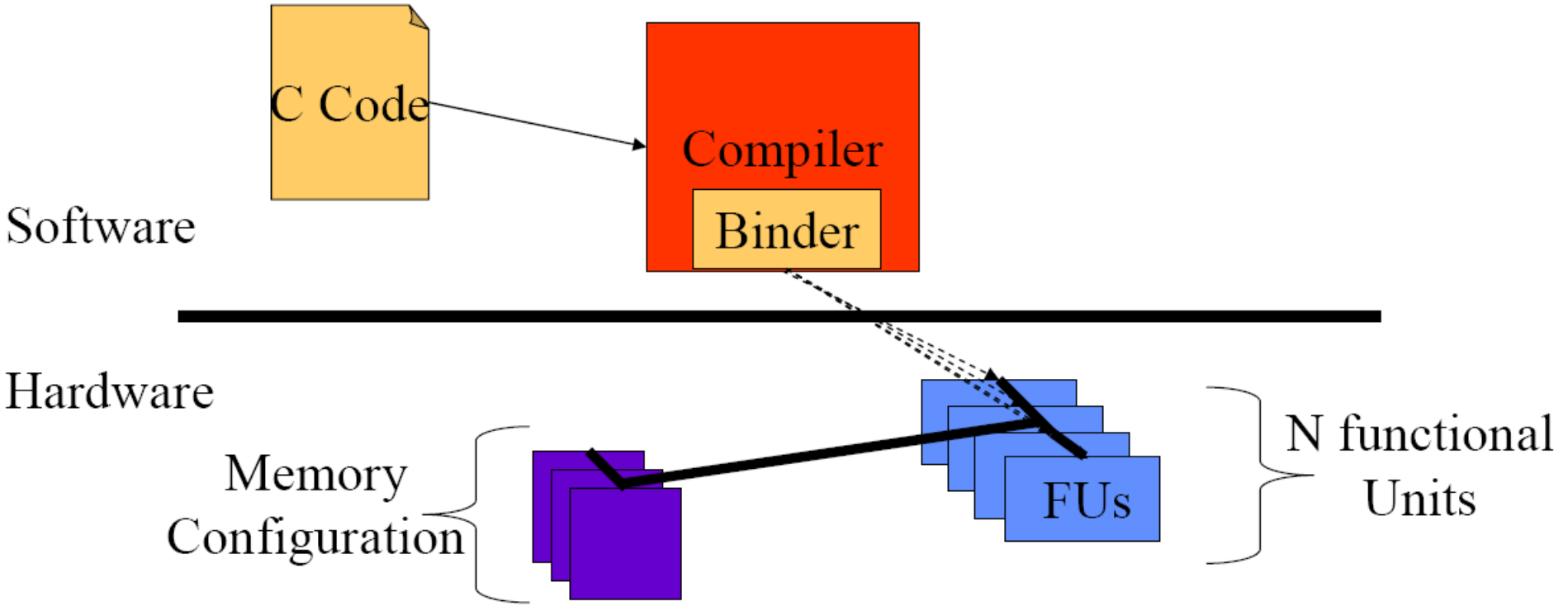


# Superscalar ISA View




- This ISA only allows a HW binder that maps instructions to FUs after dependence and conflict analysis (dynamic)
- Compiler designers only see one FU
- The HW binder is a very complex piece of HW

# VLIW ISA View




- This ISA enables a SW binder that maps instructions to FUs (Static)
- Compiler designer sees details of FUs
- The SW binder removes HW complexity

# Outline

- Uniprocessor Parallelism
  - Pipelining, Out-of-order execution, Superscalar, VLIW
  - Vector Processing/SIMD
  - Multithreading
  - Uniprocessor Memory Systems
- Explicit Parallel Computer Architectures
  - Shared-memory Architectures
  - Distributed-memory Architectures
- Parallel Software
  - Shared-memory model
  - Distributed-memory model

# Vector Code Example

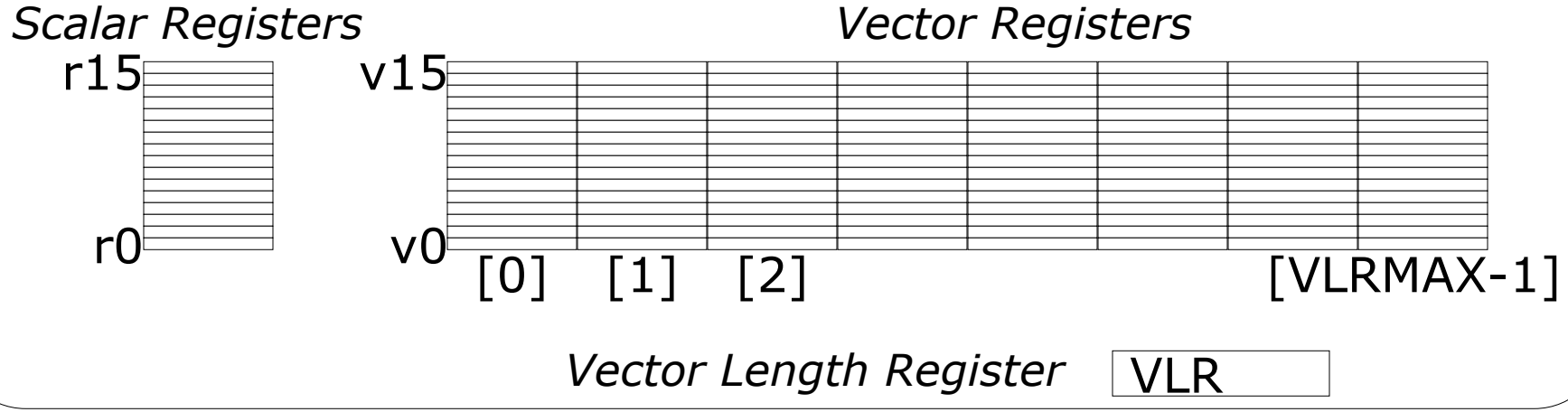
```
# C code
for (i=0; i<64; i++)
  C[i] = A[i] + B[i];
```

```
# Scalar Code
  LI R4, 64
loop:
  L.D F0, 0(R1)
  L.D F2, 0(R2)
  ADD.D F4, F2, F0
  S.D F4, 0(R3)
  DADDIU R1, 8
  DADDIU R2, 8
  DADDIU R3, 8
  DSUBIU R4, 1
  BNEZ R4, loop
```

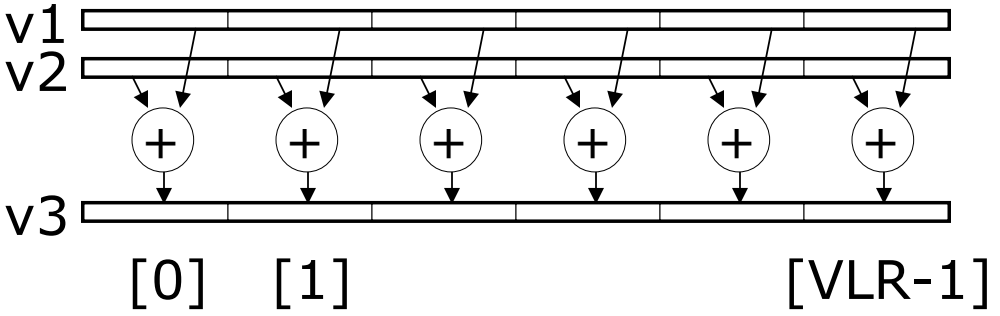
```
# Vector Code
  LI VLR, 64
  LV V1, R1
  LV V2, R2
  ADDV.D V3, V1, V2
  SV V3, R3
```

- Require programmer (or compiler) to identify parallelism
  - Hardware does not need to re-extract parallelism
- Many multimedia/HPC applications are natural consumers of vector processing

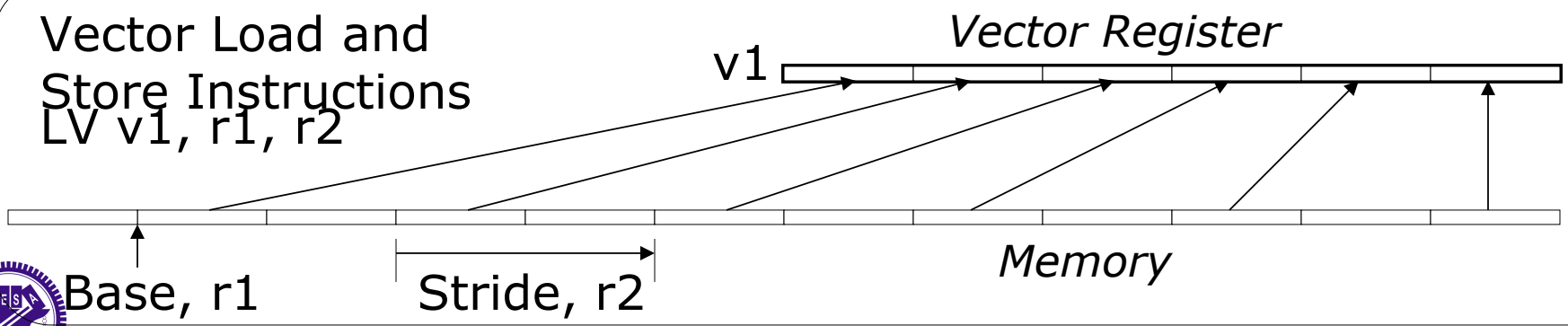
# Vector Programming Model



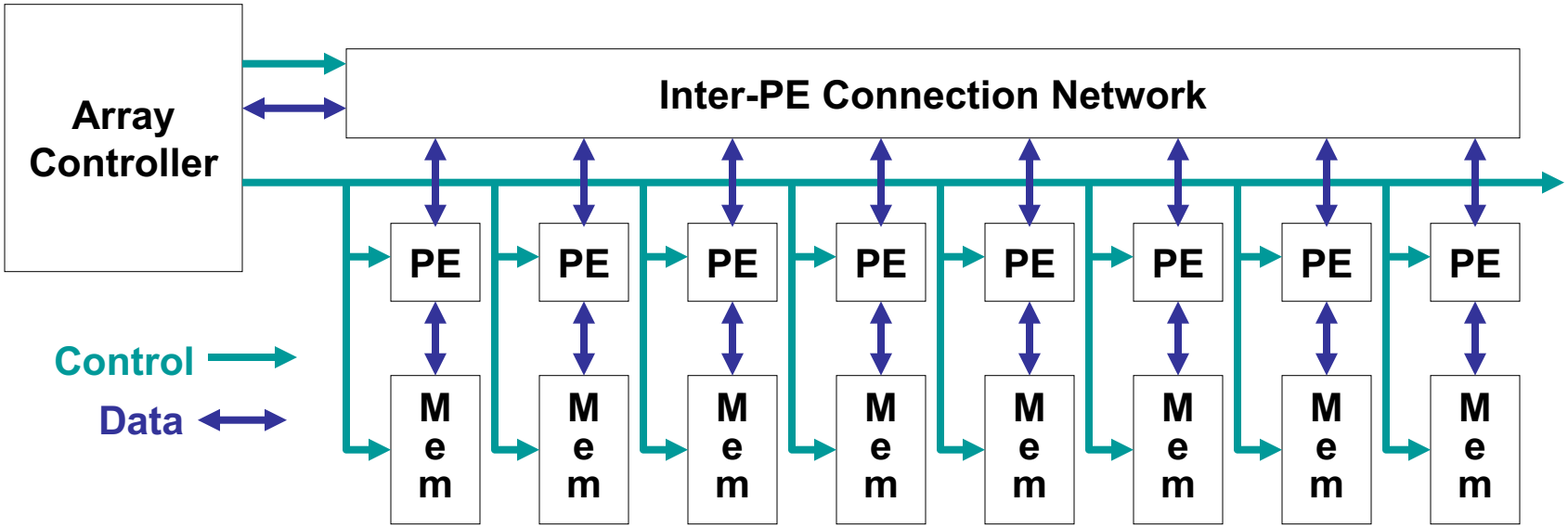

Vector Arithmetic Instructions
ADDV v3, v1, v2




Vector Load and Store Instructions
LV v1, r1, r2



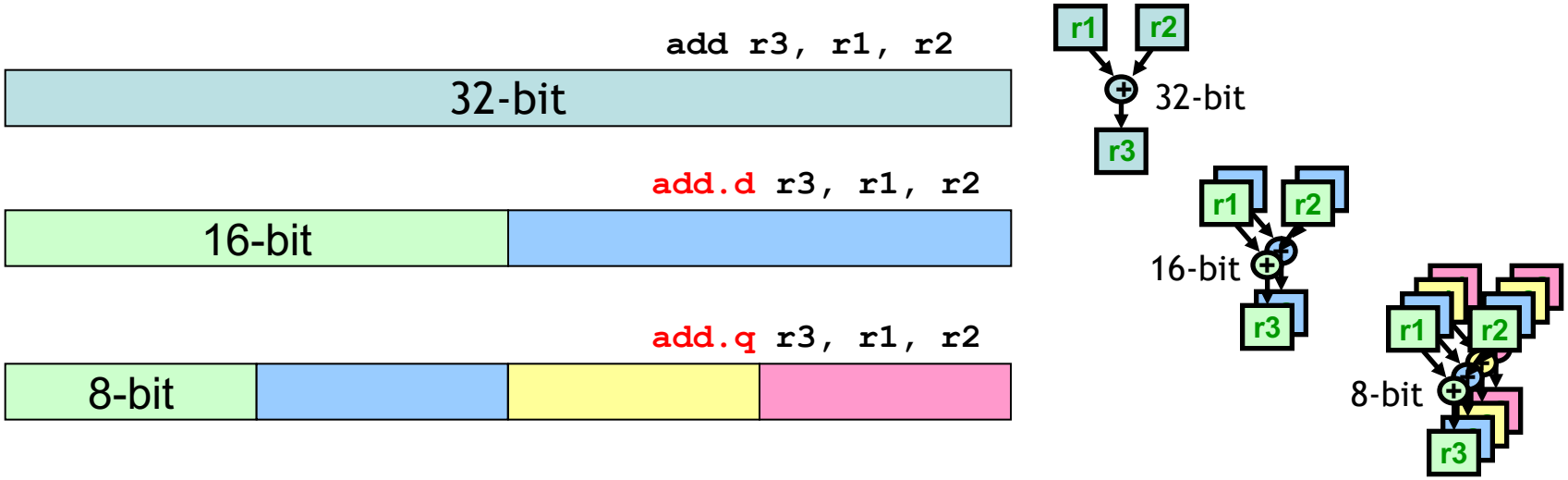
# SIMD Architecture




- Single Instruction, Multiple Data (SIMD)
- Central controller broadcasts instructions to multiple processing elements (PEs)
  - Only requires one controller for whole array
  - Only requires storage for one copy of program
  - All computations fully synchronized
- Recent return to popularity
  - GPU (Graphics Processing Units) have SIMD properties
  - However, also multicore behavior, so mix of SIMD and MIMD (more later)

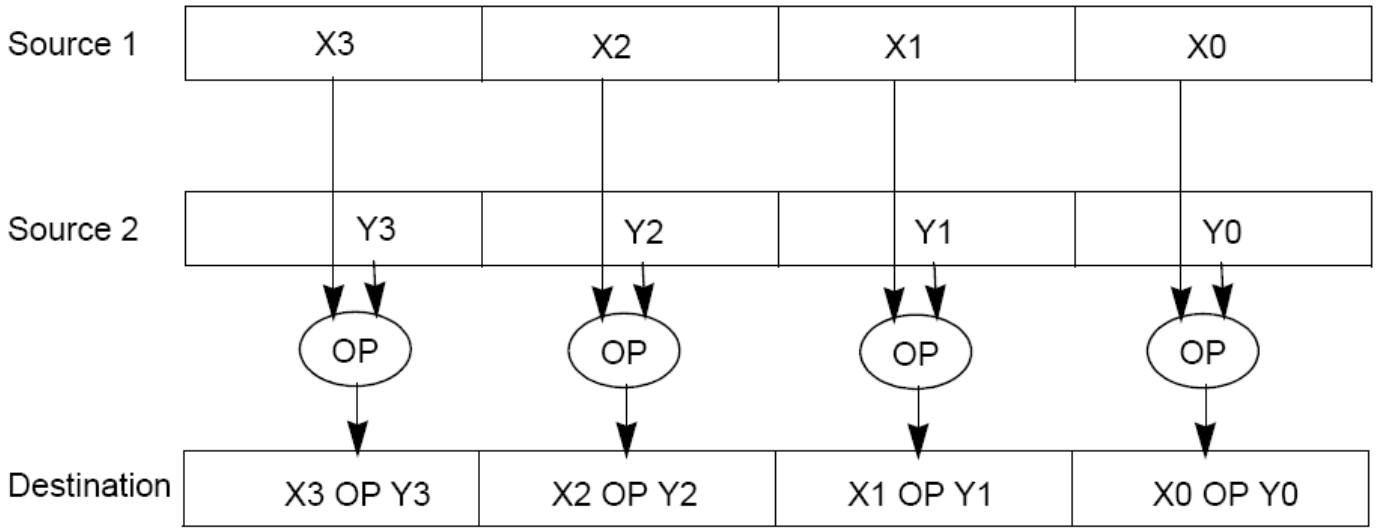
# Multimedia Extensions (SIMD Extensions)

- Motivation
  - Low media-processing performance of GPPs
  - Cost and lack of flexibility of specialized ASICs for graphics/video
  - Underutilized datapaths and registers
- Basic idea: sub-word parallelism
  - Treat a 32-bit register as a vector of 2 16-bit or 4 8-bit (short vectors)



# SIMD Instructions

- Exploits low precision and high data parallelism of multimedia applications
- E.g.




# Example of SIMD Operation (1)

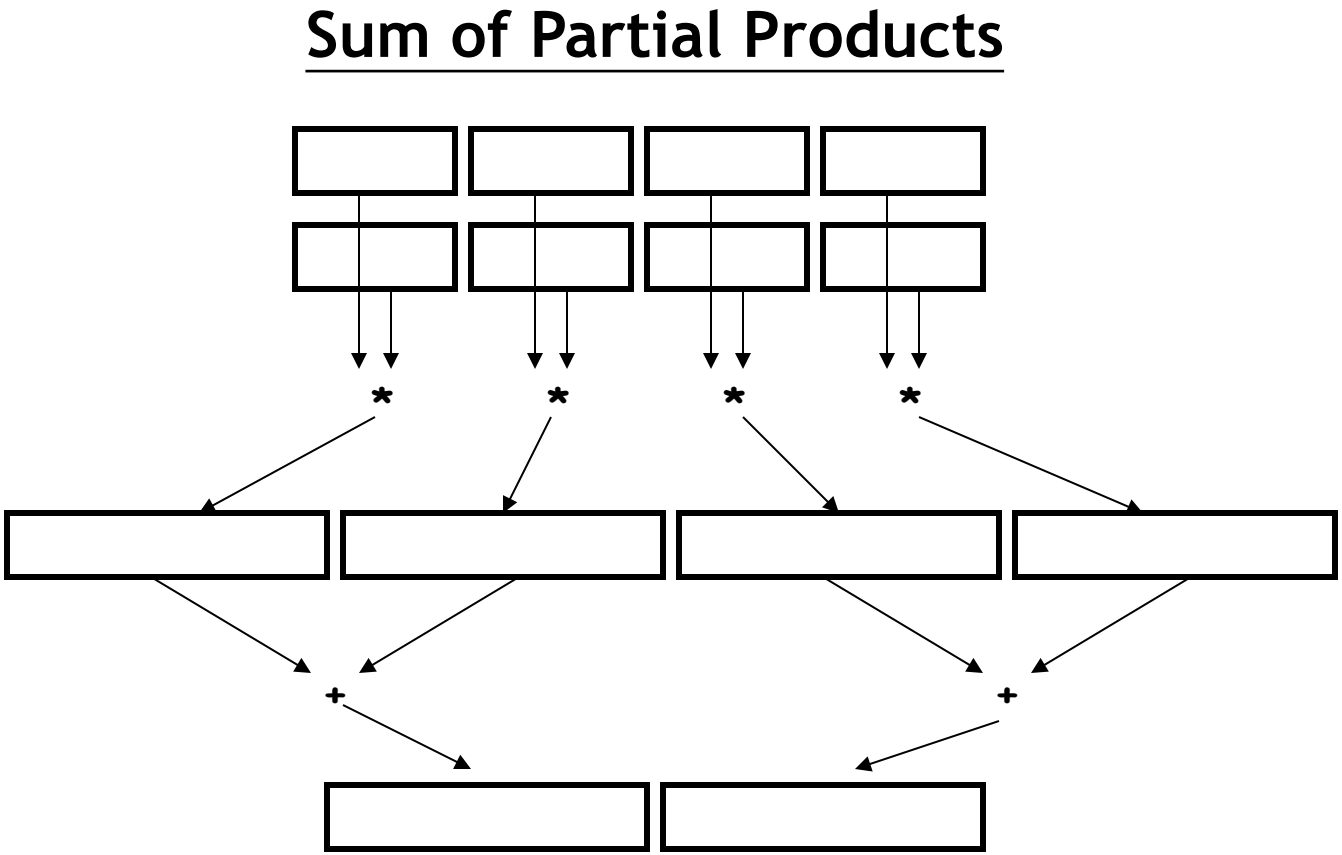
- Perform loop vectorization to exploit subword-level parallelism

```
char a[100], b[100], c[100];
for (int i = 0; i < 100; i++)
    c[i] = a[i] + b[i];
```

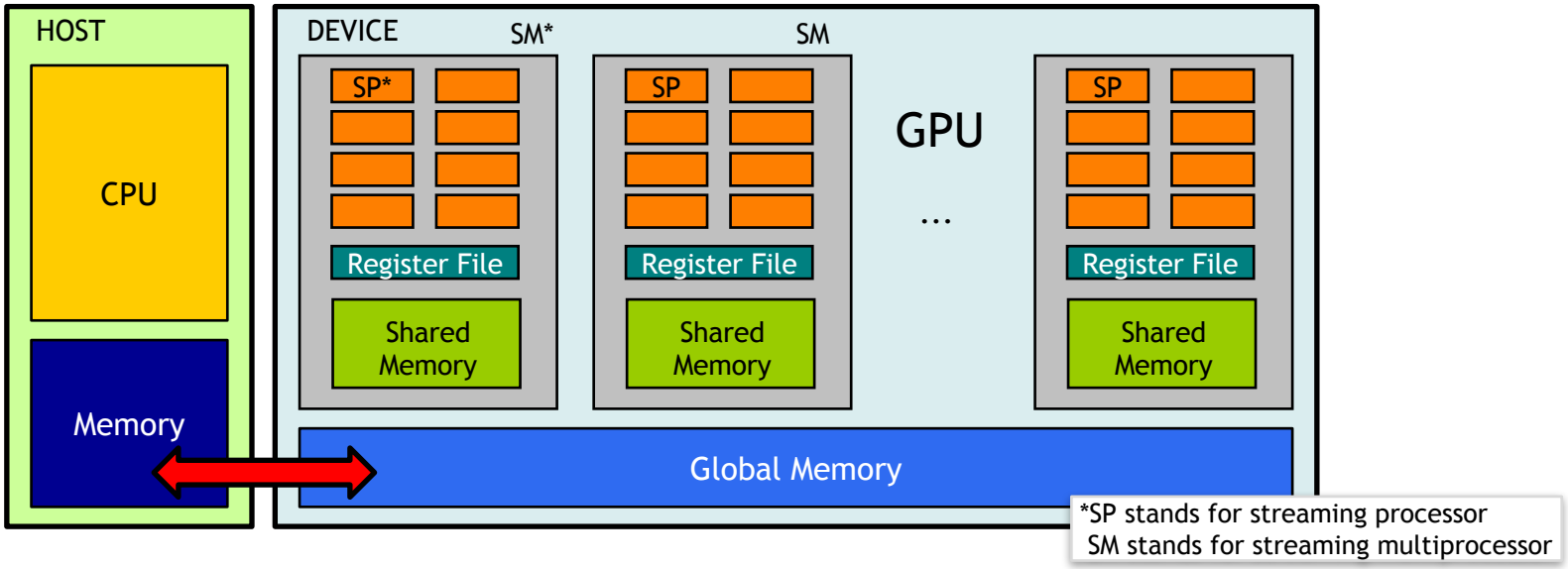
```
char a[100], b[100], c[100];
for (int i = 0; i < 100; i+=4)
  c[i:i+3] = a[i:i+3] + b[i:i+3];
```

ADD.Q c, a, b

# Example of SIMD Operation (2)



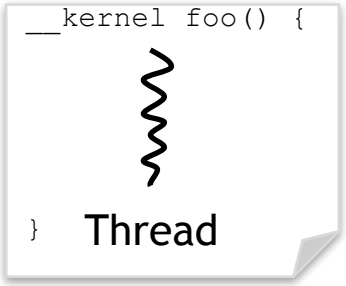
# General-Purpose GPUs (GP-GPUs)

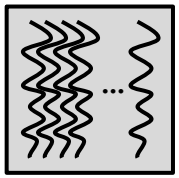



- In 2006, Nvidia introduced GeForce 8800 GPU supporting a new programming language: CUDA
  - "Compute Unified Device Architecture"
  - OpenCL is a vendor-neutral version of same ideas
- Idea: Take advantage of GPU computational performance and memory bandwidth to accelerate some kernels for general-purpose computing
- Attached processor model: Host CPU issues data-parallel kernels to GP-GPU for execution

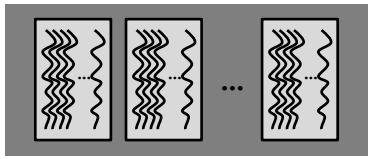
# CUDA Execution Model

Software



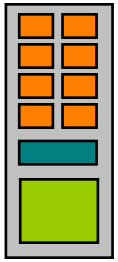
Thread Block



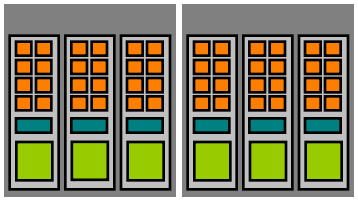
Grid

Hardware

Streaming Processor



Streaming Multiprocessor



Device

- Threads are executed by SP
- Thread blocks are executed on SM
  - Several concurrent thread blocks can reside on one SM---***limited by multiprocessor resources (shared memory and register file)***
- A kernel is launched as a grid of thread blocks
- SPMD/SIMT

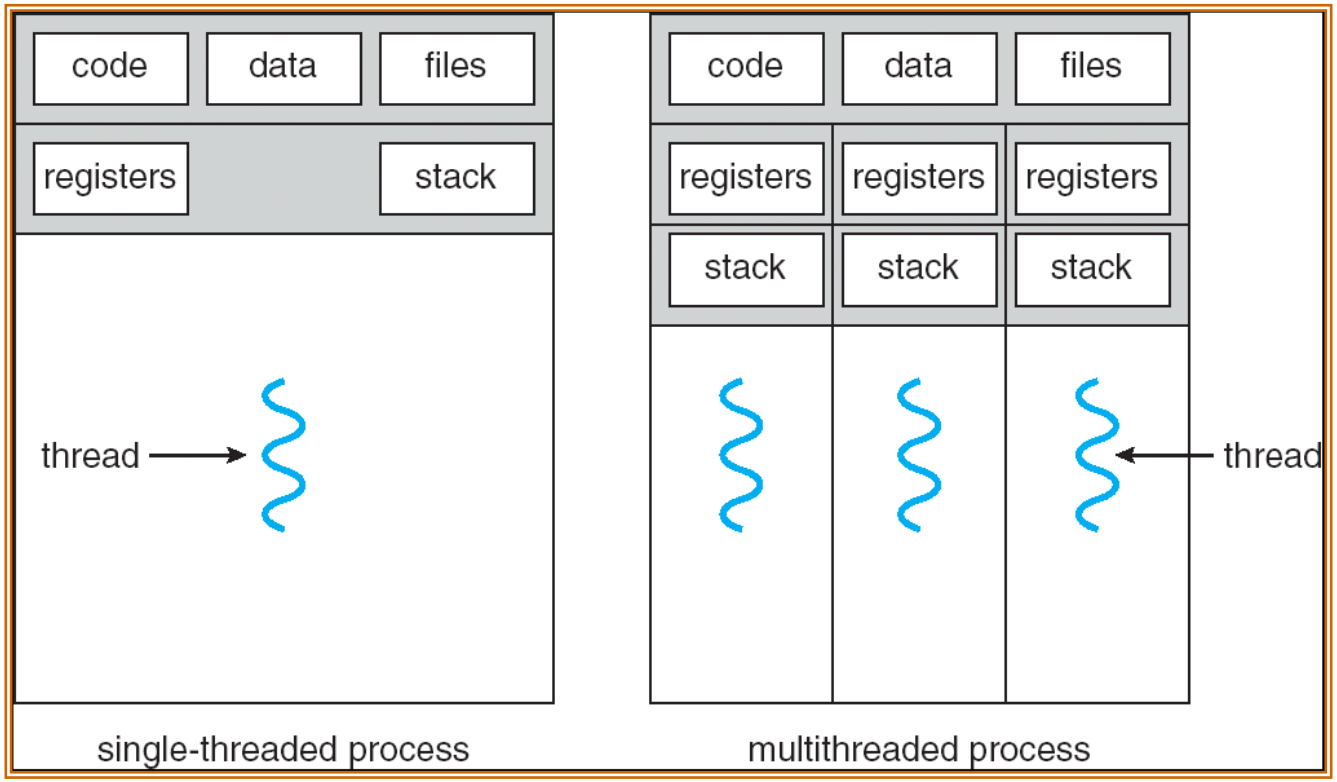
# Outline

- Uniprocessor Parallelism
  - Pipelining, Out-of-order execution, Superscalar, VLIW
  - Vector Processing/SIMD
  - Multithreading
  - Uniprocessor Memory Systems
- Explicit Parallel Computer Architectures
  - Shared-memory Architectures
  - Distributed-memory Architectures
- Parallel Software
  - Shared-memory model
  - Distributed-memory model

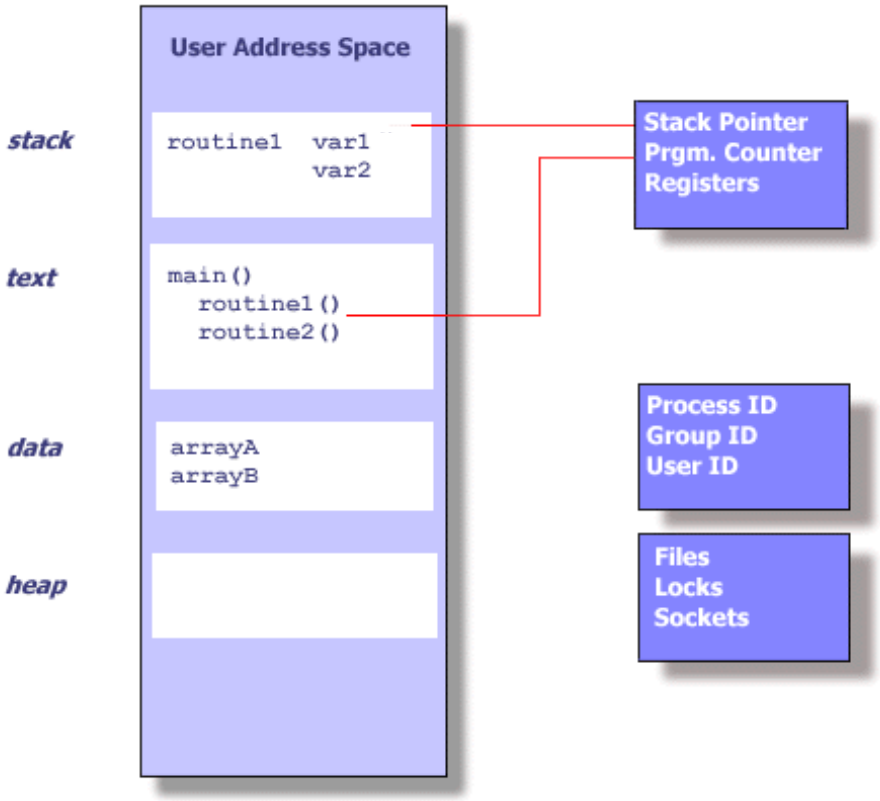
# Thread-Level Parallelism (TLP)

- ILP exploits implicit parallel operations within a loop or straight-line code segment
- TLP explicitly represented by the use of multiple threads of execution that are inherently parallel
  - Threads can be on a single processor
  - Or, on multiple processors
- Goal: Use multiple instruction streams to improve
  - Throughput of computers that run many programs
  - Execution time of multi-threaded programs

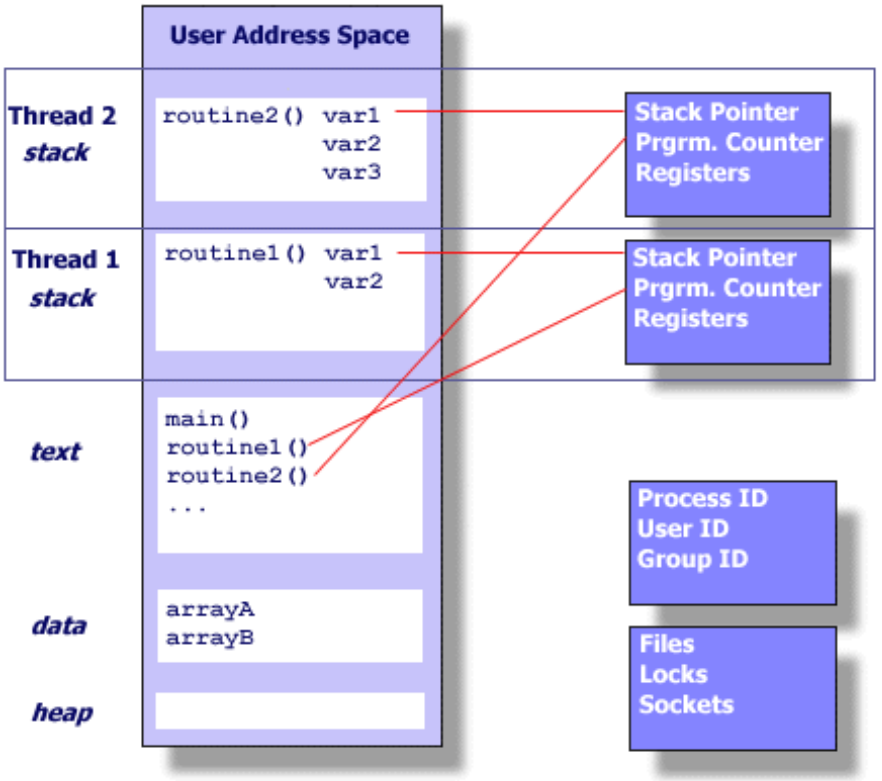
# Single and Multithreaded Processes



# Processes v.s. Threads: Memory Maps




A process




Threads within a process

Source: https://computing.llnl.gov/tutorials/pthreads

# Common Notions of Thread Creation

- cobegin/coend

```
cobegin
        job1(a1);
        job2(a2);
coend
```

  - Statements in block may run in parallel
  - cobegins may be nested
  - Scoped, so you cannot have a missing coend

- fork/join

```
tid1 = fork(job1, a1);
job2(a2);
join tid1;
```

  - Forked procedure runs in parallel
  - Wait at join point if it's not finished

- future

```
v = future(job1(a1));
… = …v…;
```

  - Future expression evaluated in parallel
  - Attempt to use return value will wait

- Cobegin cleaner than fork, but fork is more general
- Futures require some compiler (and likely hardware) support

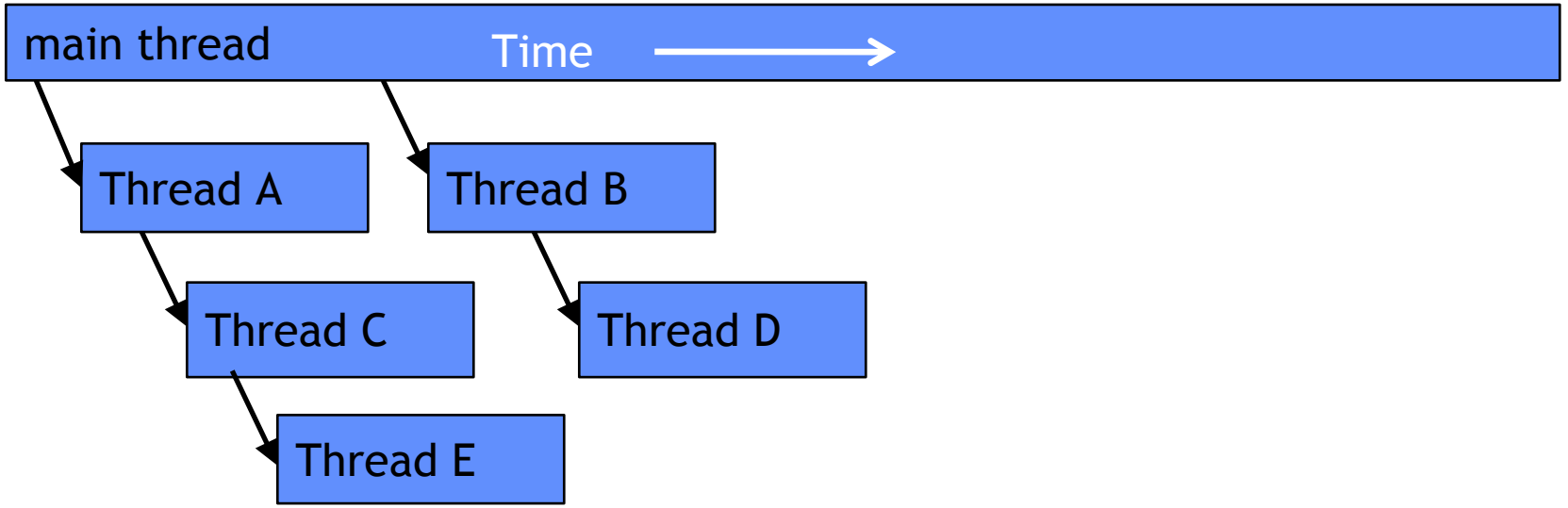
# Loop-Level Parallelism

- Many scientific applications have parallelism in loops

```
double stuff [n][n];
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    … pthread_create (…, update_stuff, …,
                      &stuff[i][j]);
```

- But overhead of thread creation is nontrivial
  - `update_stuff` should have a significant amount of work
- Common Performance Pitfall: Too many threads
  - The cost of creating a thread is 10s of thousands of cycles on modern architectures
  - Solution: Thread blocking: use a small # of threads, often equal to the number of cores/processors or hardware threads

# Thread Scheduling




- Once created, when will a given thread run?
  - It is up to the Operating System or hardware (e.g., on GPUs), but it will run eventually, even if you have more threads than cores
  - But – scheduling may be non-ideal for your application
- Programmer can provide hints or affinity in some cases
  - E.g., create exactly P threads and assign to P cores
- Can provide user-level scheduling for some systems

# Multithreaded Execution

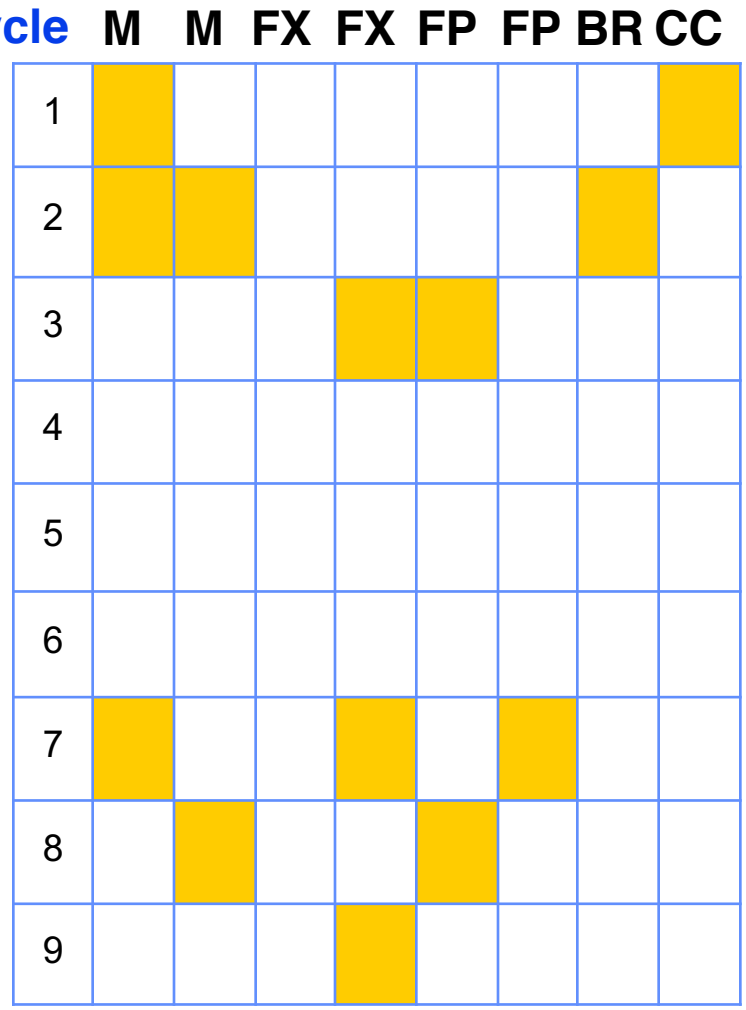
- Multitasking operating system:
  - Gives “illusion” that multiple things happening at same time
  - Switches at a course-grained time quanta (for instance: 10ms)
- Hardware Multithreading: multiple threads share processor simultaneously (with little OS help)
  - Hardware does switching
    - HW for fast thread switch in small number of cycles
    - Much faster than OS switch which is 100s to 1000s of clocks
  - Processor duplicates independent state of each thread
    - E.g., a separate copy of register file, a separate PC, and for running independent programs, a separate page table
- When to switch between threads?
  - Alternate instruction per thread (fine grain)
  - When a thread is stalled, perhaps for a cache miss, another thread can be executed (coarse grain)

# What about combining ILP and TLP?

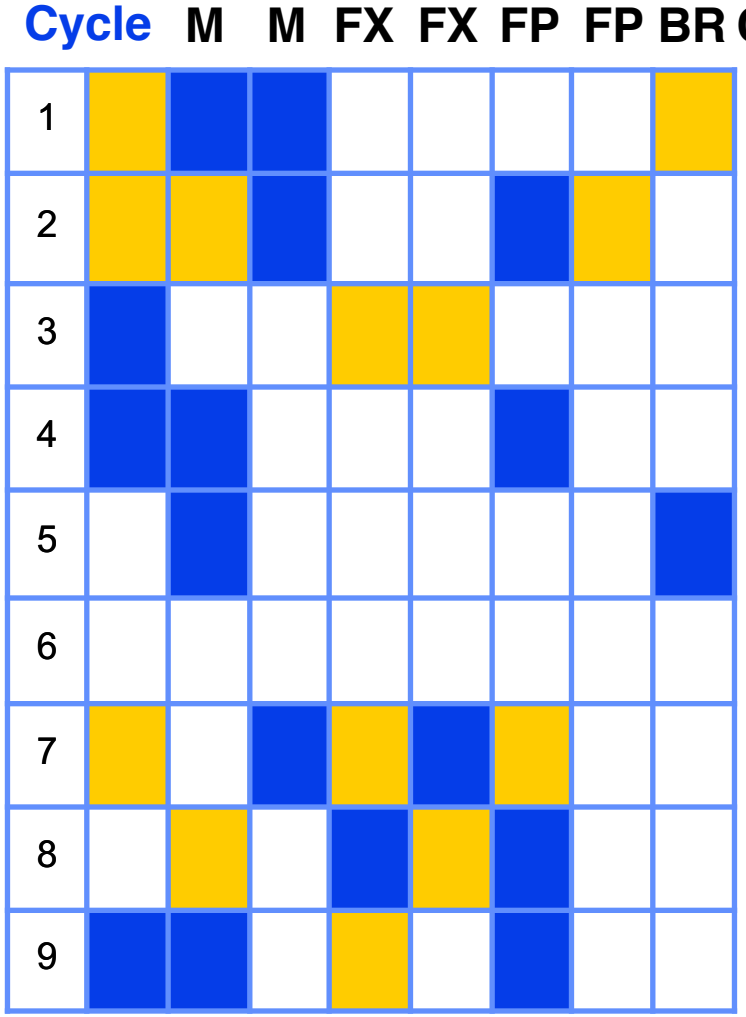
- TLP and ILP exploit two different kinds of parallel structure in a program
- Could a processor oriented at ILP benefit from exploiting TLP?
  - Functional units are often idle in data path designed for ILP because of either stalls or dependences in the code
  - TLP used as a source of independent instructions that might keep the processor busy during stalls
  - TLP be used to occupy functional units that would otherwise lie idle when insufficient ILP exists
- Called “Simultaneous Multithreading” (SMT)
  - Intel renamed this as “Hyperthreading”

# Simultaneous Multithreading

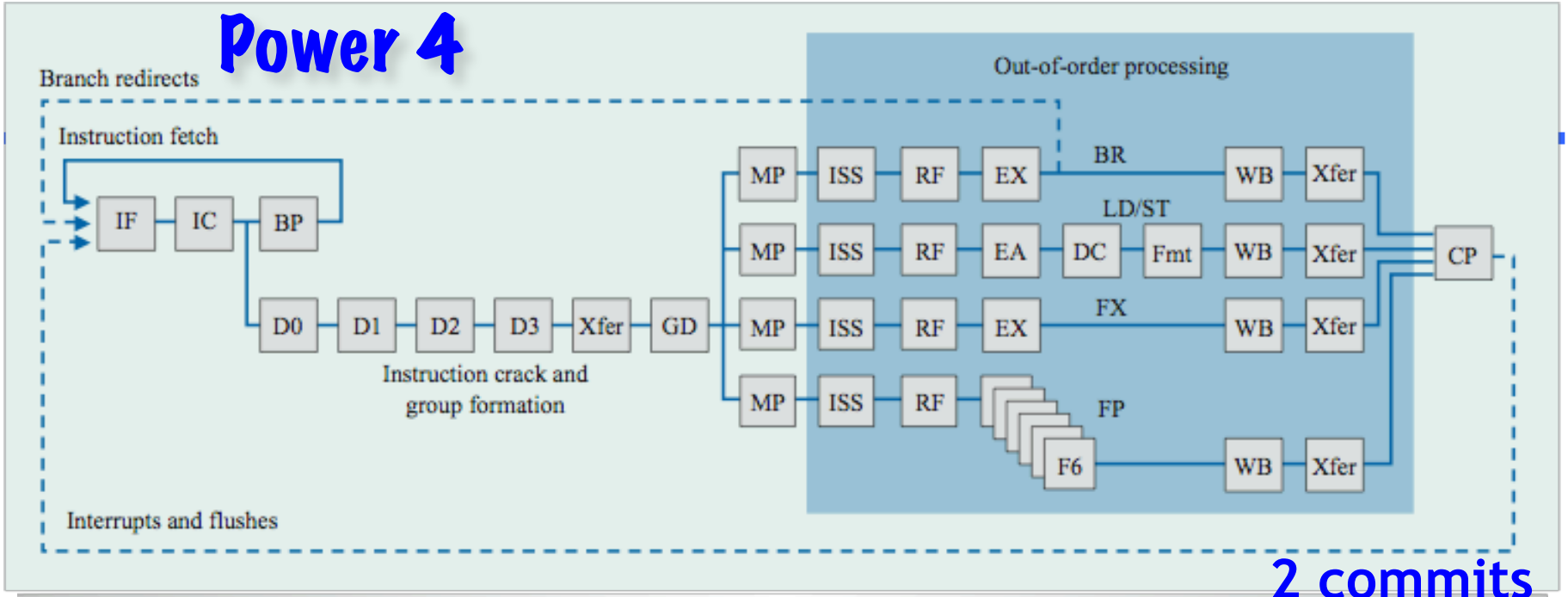
## One thread, 8 units

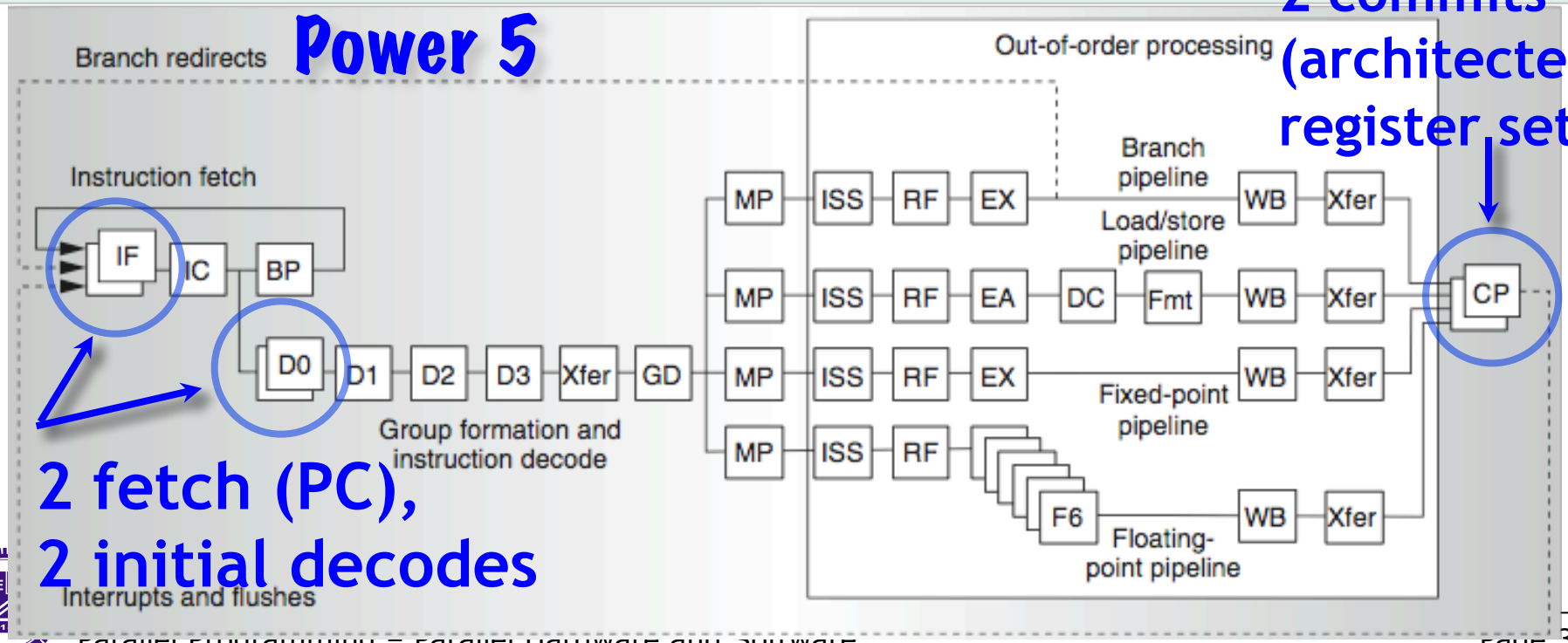



## Two threads, 8 units




M = Load/Store, FX = Fixed Point, FP = Floating Point, BR = Branch, CC = Condition Codes

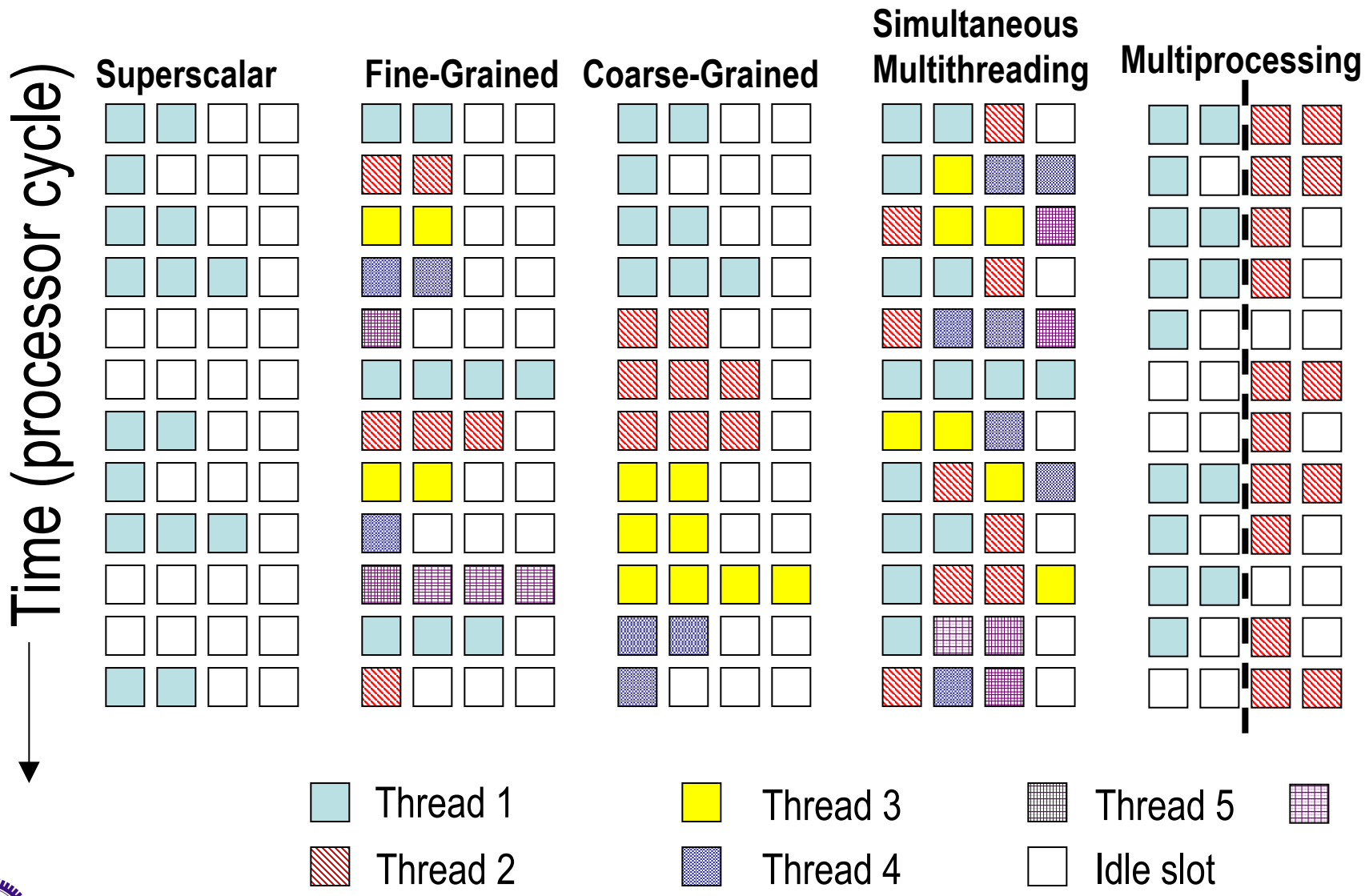



2 commits (architected register sets)

2 fetch (PC), 2 initial decodes

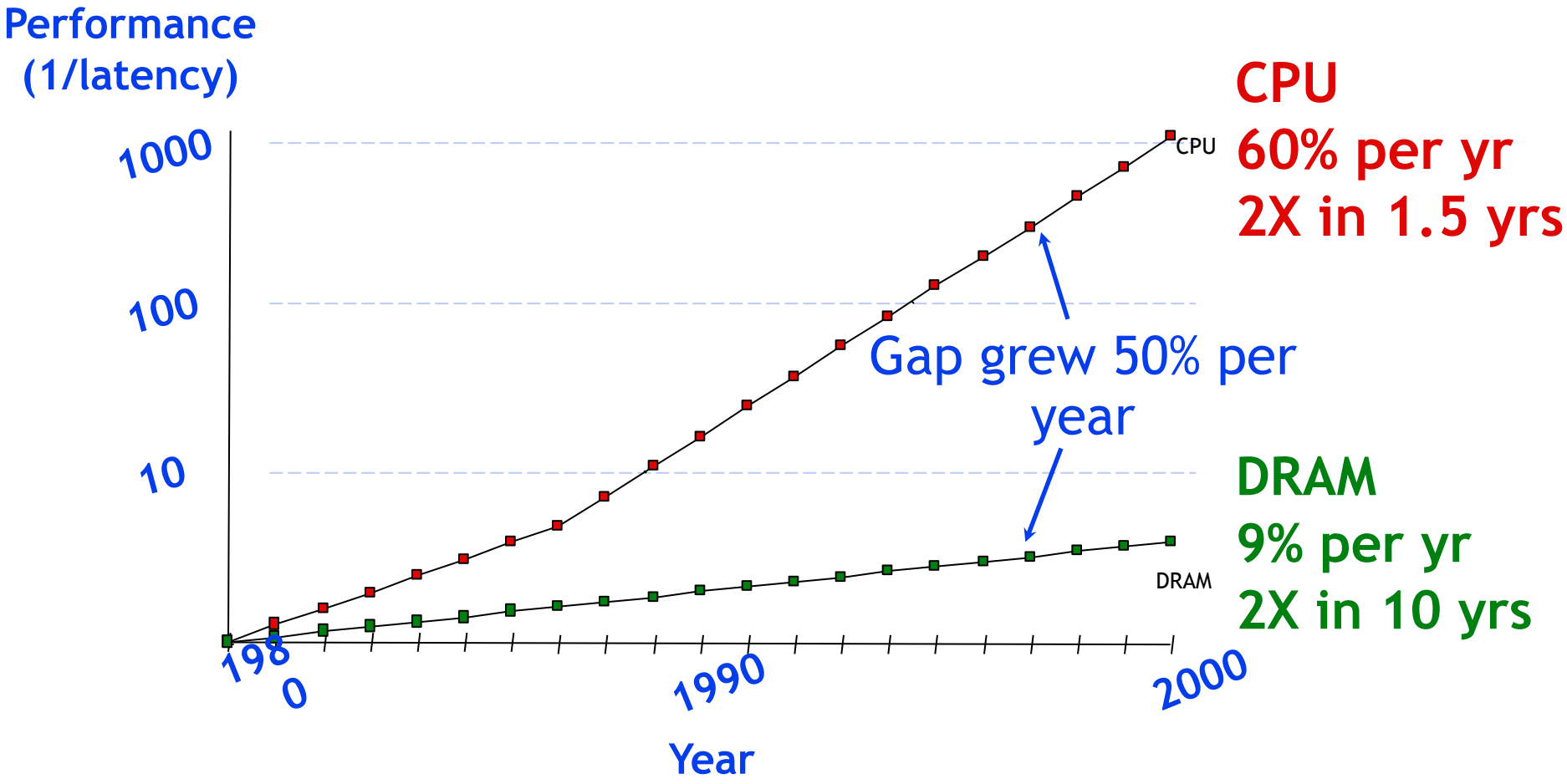
# Summary: Multithreaded Categories



# Outline

- Uniprocessor Parallelism
  - Pipelining, Out-of-order execution, Superscalar, VLIW
  - Vector Processing/SIMD
  - Multithreading
  - Uniprocessor Memory Systems
- Explicit Parallel Computer Architectures
  - Shared-memory Architectures
  - Distributed-memory Architectures
- Parallel Software
  - Shared-memory model
  - Distributed-memory model

# Limiting Force: Memory System

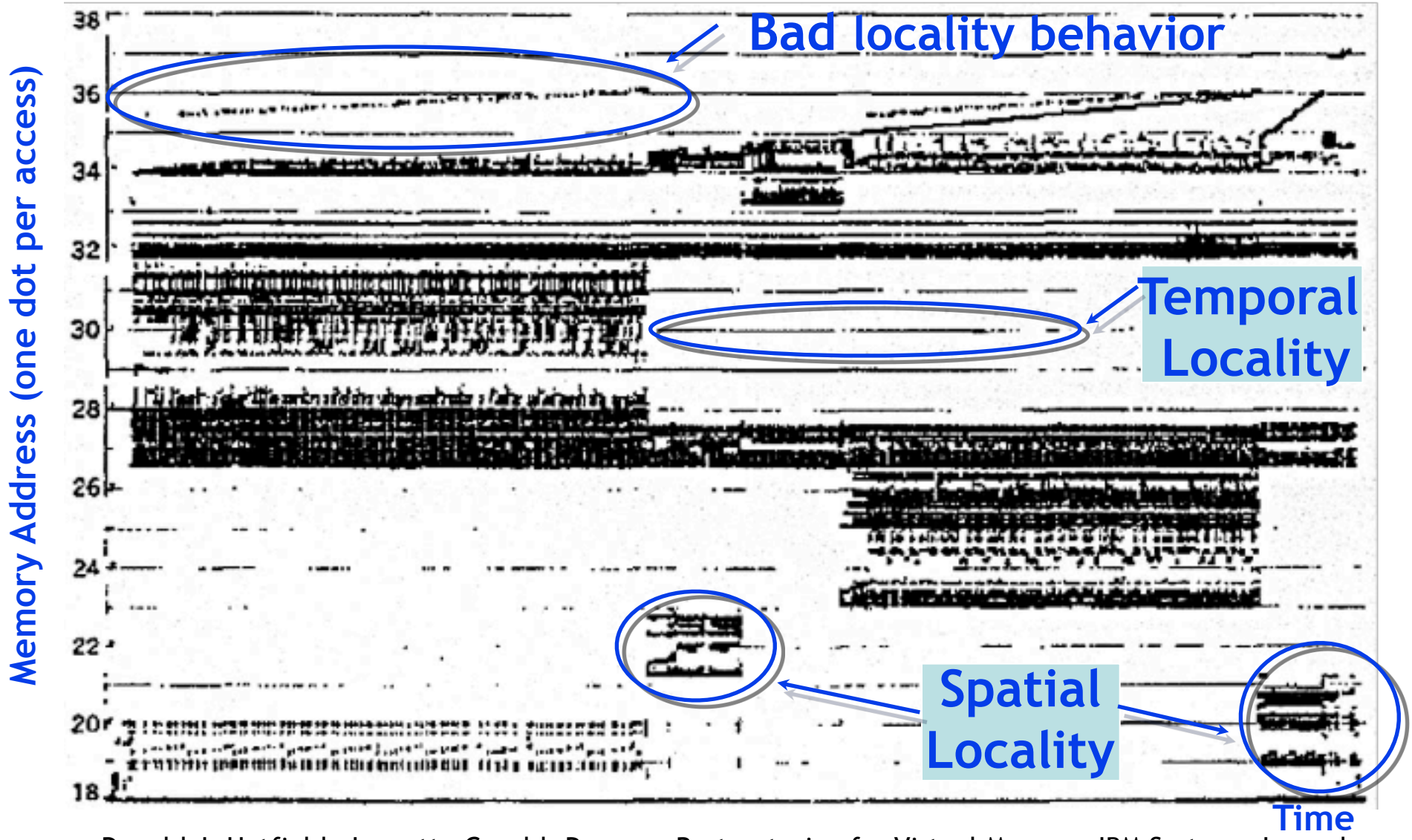



- How do architects address this gap?
  - Put small, fast "cache" memories between CPU and DRAM
  - Create a "memory hierarchy"

# Principle of Locality

- The Principle of Locality:
  - Program access a relatively small portion of the address space at any instant of time
- Two Different Types of Locality:
  - Temporal Locality (Locality in Time): If an item is referenced, it will tend to be referenced again soon (e.g., loops, reuse)
  - Spatial Locality (Locality in Space): If an item is referenced, items whose addresses are close by tend to be referenced soon (e.g., straightline code, array access)
- Last 30 years, HW relied on locality for speed

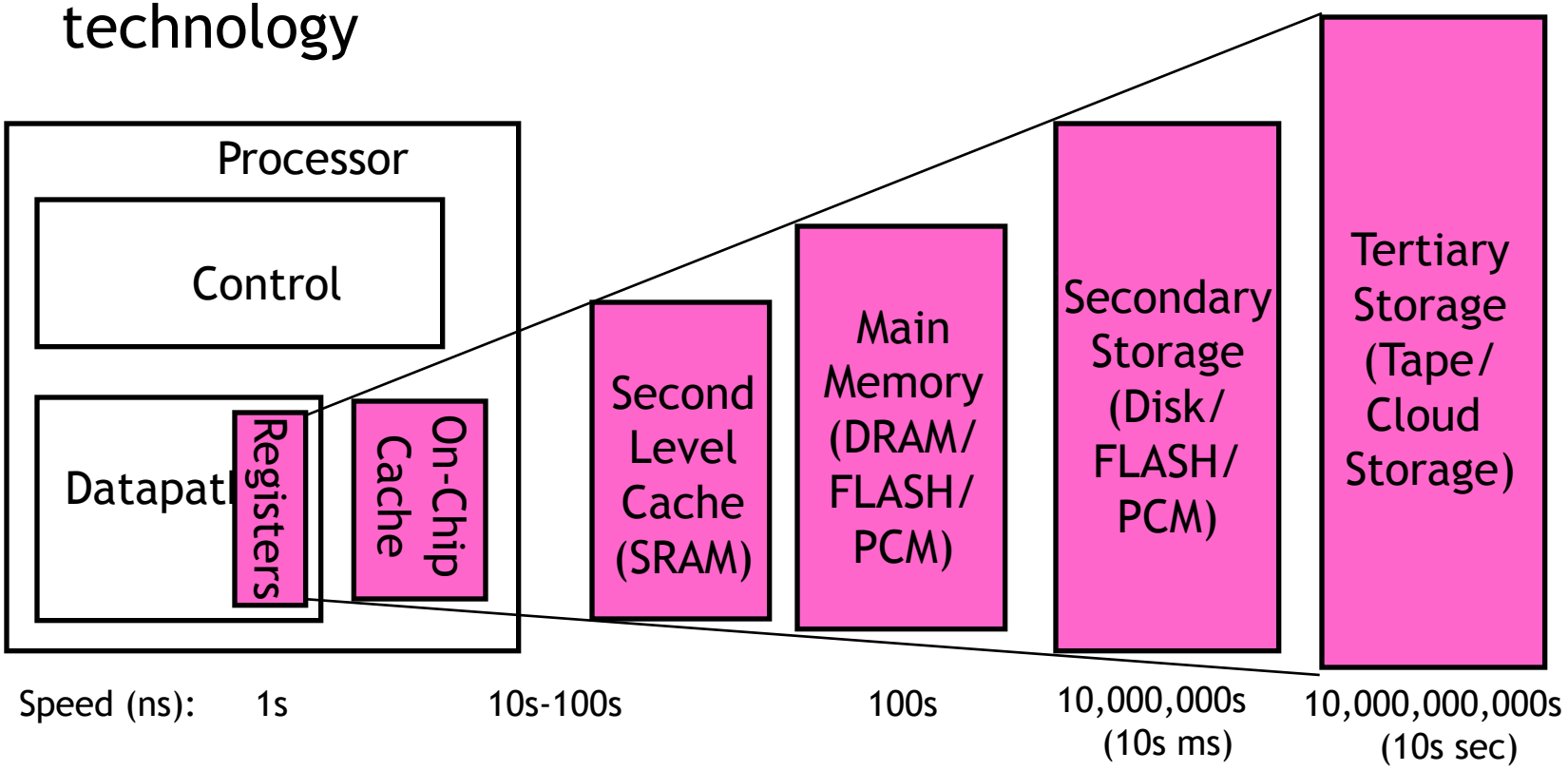
# Programs with locality cache well …

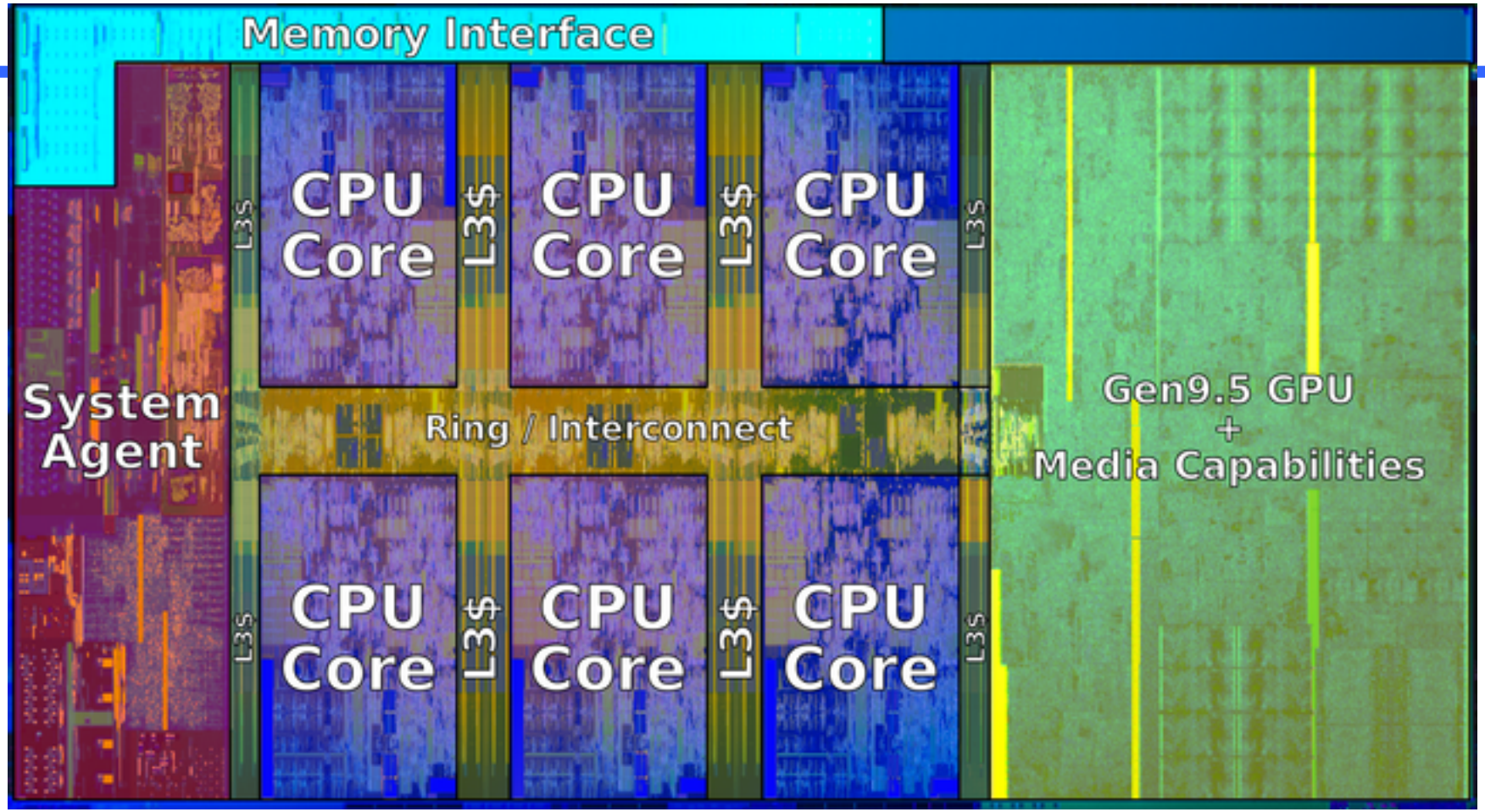



Donald J. Hatfield, Jeanette Gerald: Program Restructuring for Virtual Memory. IBM Systems Journal 10(3): 168-192 (1971)

# Memory Hierarchy

- Take advantage of the principle of locality to:
  - Present as much memory as in the cheapest technology
  - Provide access at speed offered by the fastest technology






(Coffee Lake)

- 64 KB (32 KB Instruction + 32 KB Data) L1 cache
- 256 KB L2 cache and 2 MB L3 cache per core
- 128 MB shared L4 cache

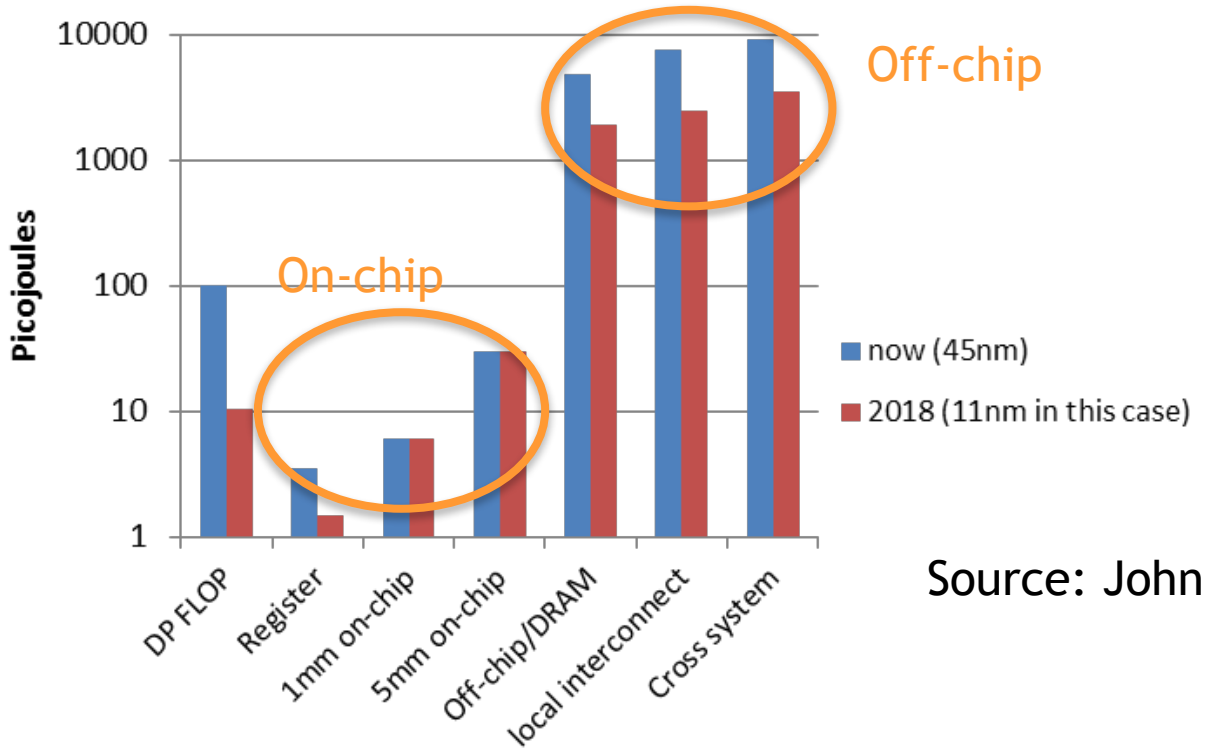
# Outline

- Uniprocessor Parallelism
  - Pipelining, Out-of-order execution, Superscalar, VLIW
  - Vector Processing/SIMD
  - Multithreading
  - Uniprocessor Memory Systems
- Explicit Parallel Computer Architectures
  - Shared-memory Architectures
  - Distributed-memory Architectures
- Parallel Software
  - Shared-memory model
  - Distributed-memory model

# What is Parallel Architecture?

- A parallel computer is a collection of processing elements that cooperate to solve large problems
  - Most important new element: It is all about communication!
- What does the programmer (or OS or Compiler writer) think about?
  - Resource Allocation:
    - How powerful are the elements?
    - How much memory?
- What does a single processor look like?
  - High performance general purpose processor
  - SIMD processor/Vector Processor
- Data access, Communication and Synchronization
  - How do the elements cooperate and communicate?
  - How are data transmitted between processors?
  - What are the abstractions and primitives for cooperation?

# Communication Dominant Factor!

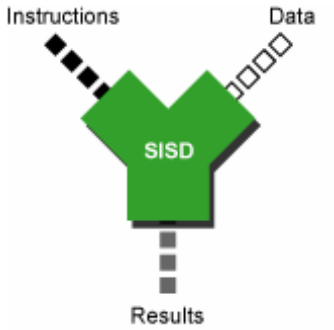


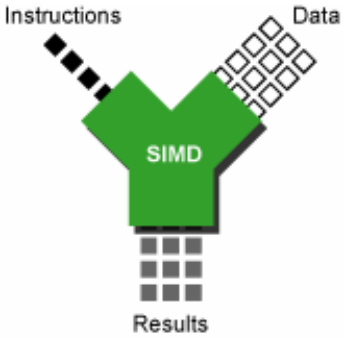

Source: John Shalf, LBNL

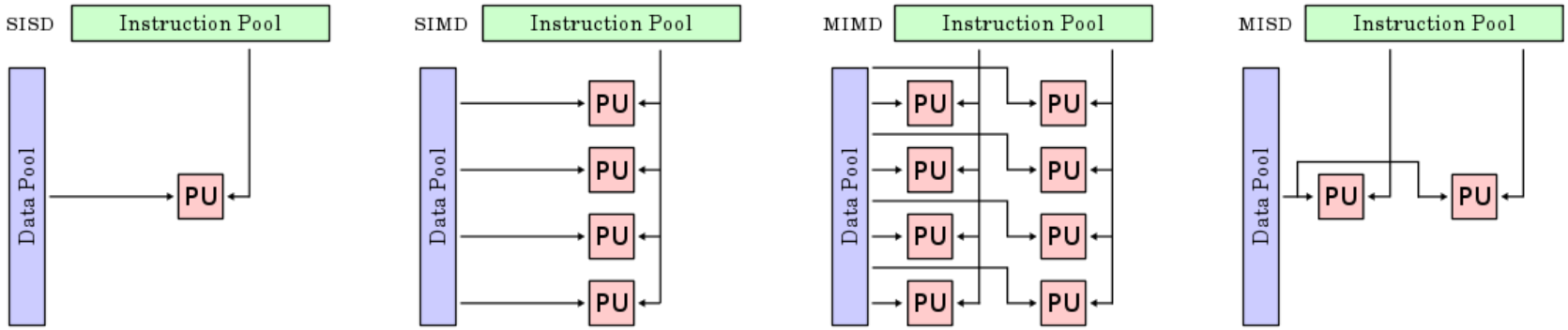
- Must schedule data movement to optimize performance
  - Time per flop << latency
- Energy of Communication >> Energy of Computation!
  - Flops are free, Mops (movement ops) are not!

# Flynn's Taxonomy of Processors

- Single-instruction single-data (SISD)
  - Conventional uniprocessor
- Single-instruction multiple-data (SIMD)
  - All processors perform the same operations
- Multiple-instruction multiple-data (MIMD)
  - Homogeneous or heterogeneous multiprocessor
- Multiple-instruction single-data (MISD)
  - Systolic arrays

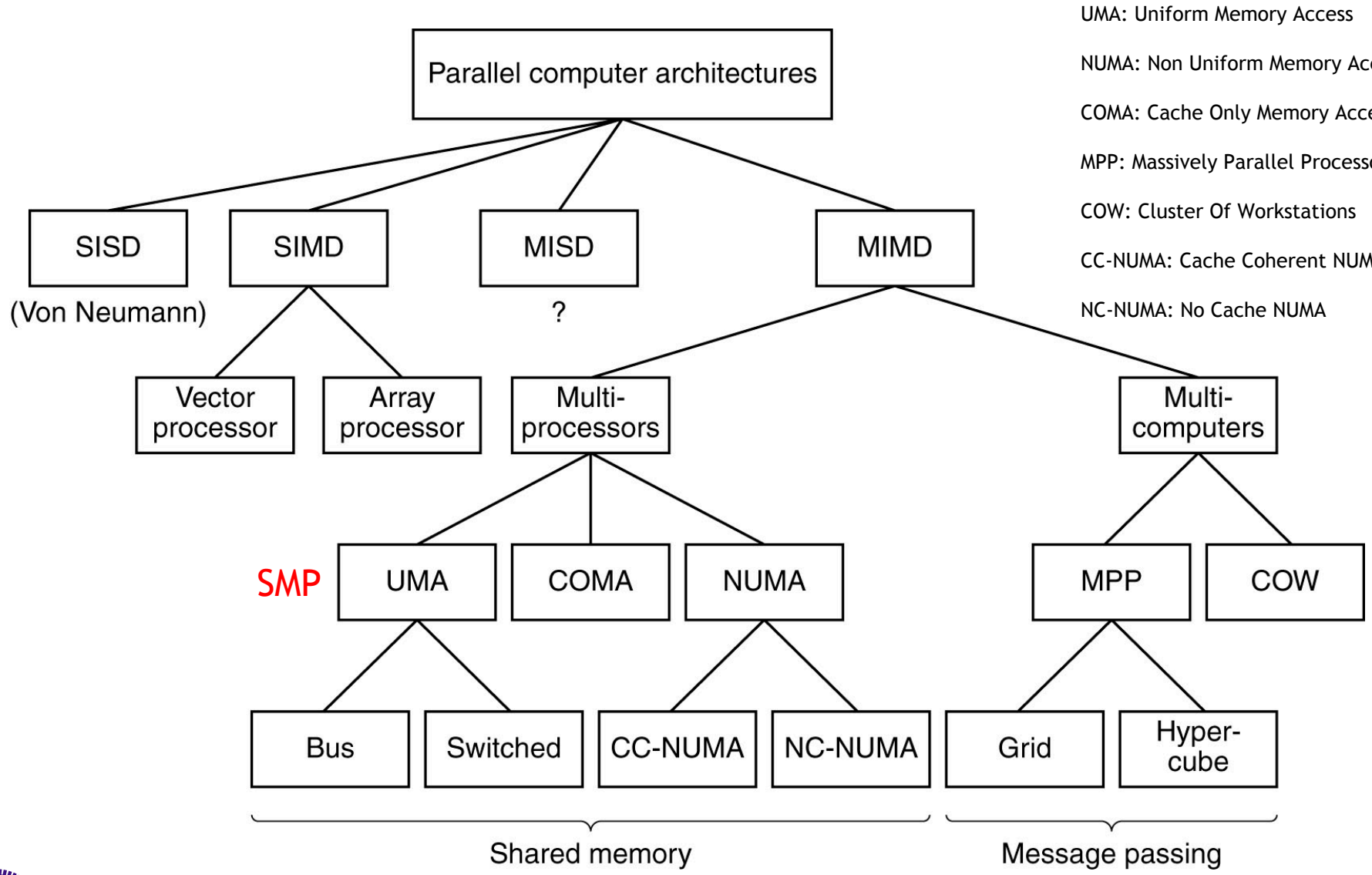



Proposed by Michael Flynn, 1966

# Taxonomy of Parallel Architectures




UMA: Uniform Memory Access

NUMA: Non Uniform Memory Access

COMA: Cache Only Memory Access

MPP: Massively Parallel Processor

COW: Cluster Of Workstations

CC-NUMA: Cache Coherent NUMA

NC-NUMA: No Cache NUMA

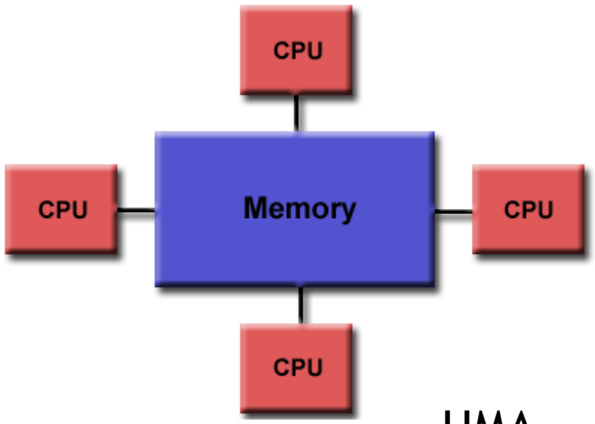
Source: Tanenbaum, Structured Computer Organization

# Outline

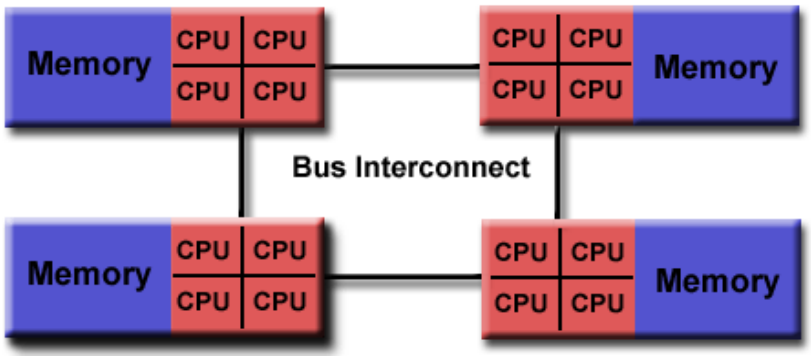
- Uniprocessor Parallelism
  - Pipelining, Out-of-order execution, Superscalar, VLIW
  - Vector Processing/SIMD
  - Multithreading
  - Uniprocessor Memory Systems
- Explicit Parallel Computer Architectures
  - Shared-memory Architectures
  - Distributed-memory Architectures
- Parallel Software
  - Shared-memory model
  - Distributed-memory model

# Shared-Memory Architecture

- Memory (centralized or distributed) can be directly accessed by different CPUs
- Communication between programs/threads occurs implicitly via memory instructions (e.g., loads and stores)
- A natural extension of uniprocessor model
  - Shared data are location transparent

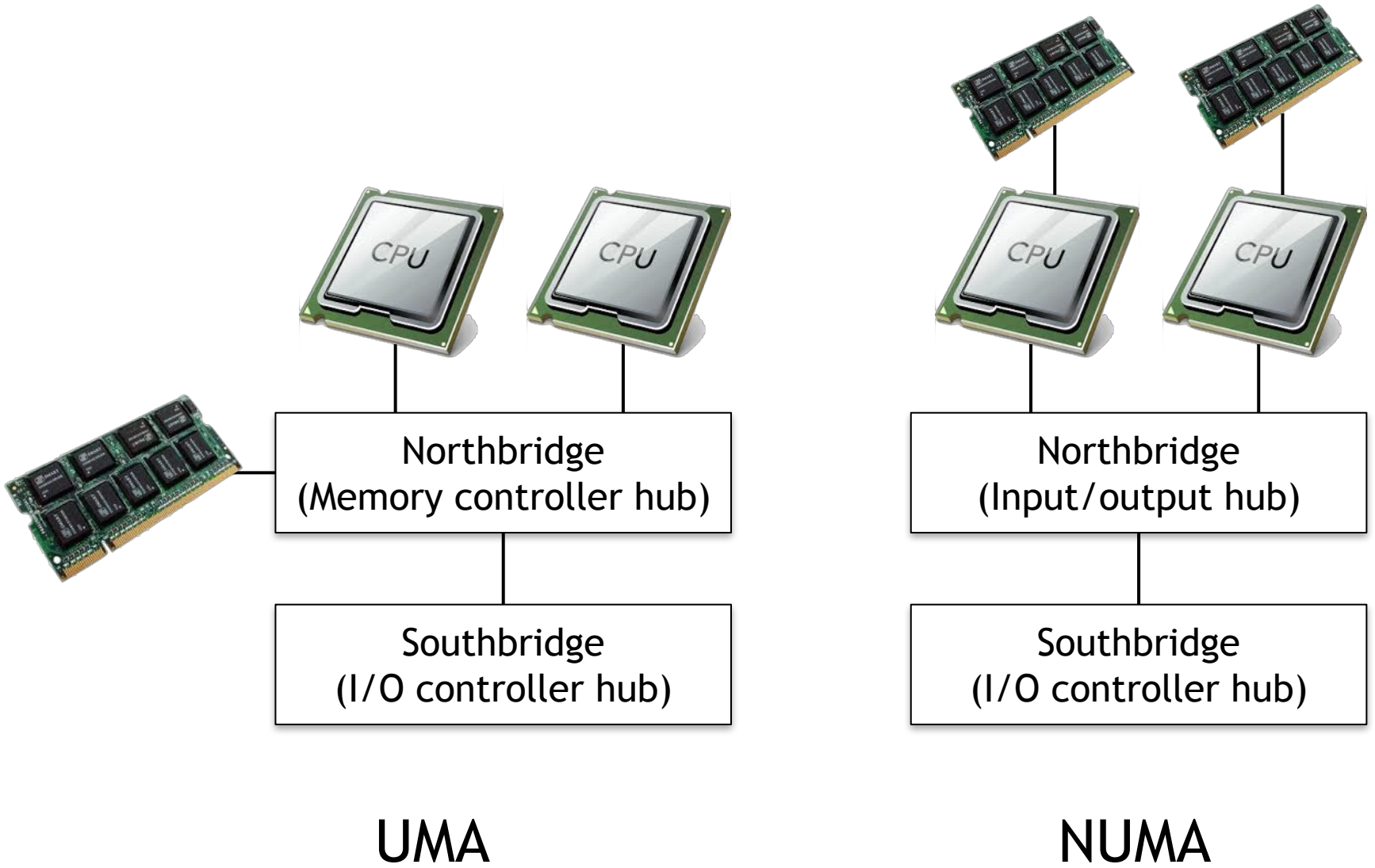



UMA




NUMA

# UMA vs NUMA

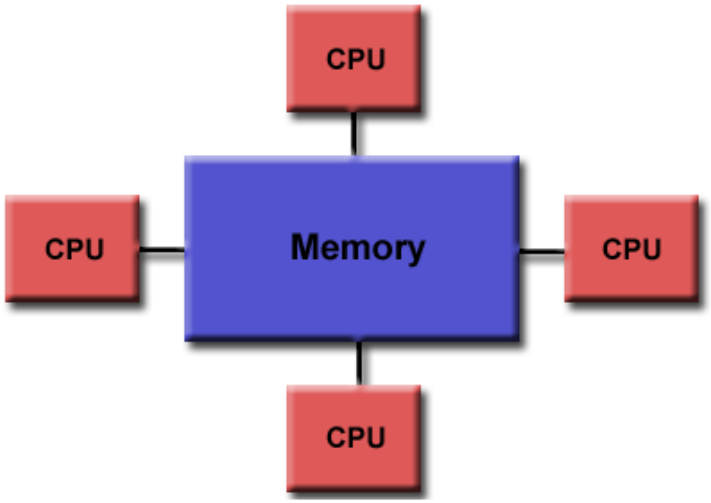



UMA

NUMA

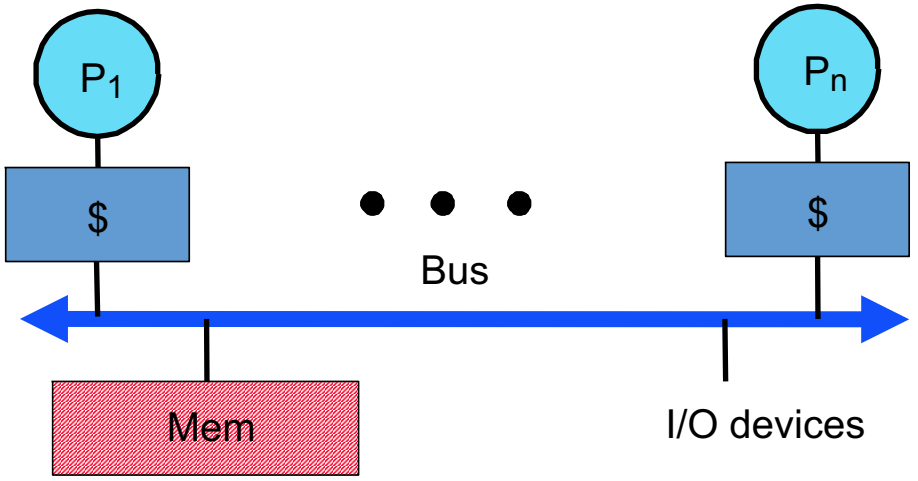
# Uniform Memory Access (UMA)

- Most commonly represented by Symmetric Multiprocessor (SMP) machines
  - Multiple cores on a die
  - Equal access and access times to memory

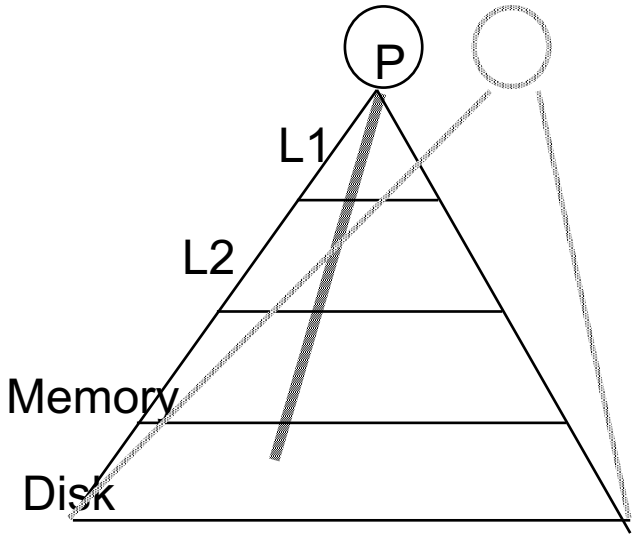



# What About Caching???

- Want high performance for shared memory: Use Caches!
  - Each processor has its own cache (or multiple caches)
  - Place data from memory into cache
  - Caches reduce average latency
    - Automatic replication closer to processor
    - *More* important to multiprocessor than uniprocessor: latencies longer
- Writeback cache: doesn't send all writes over bus to memory immediately
- Problem: Cache Coherence!
  - Write-through cache also has the problem

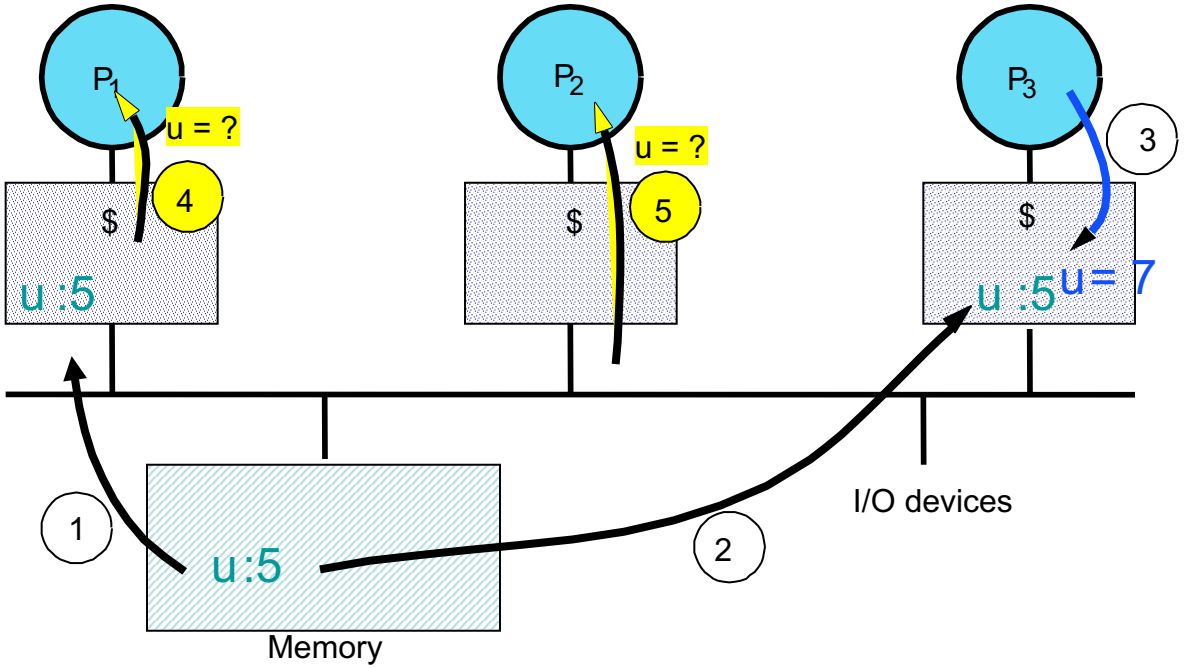


# Intuitive Memory Model



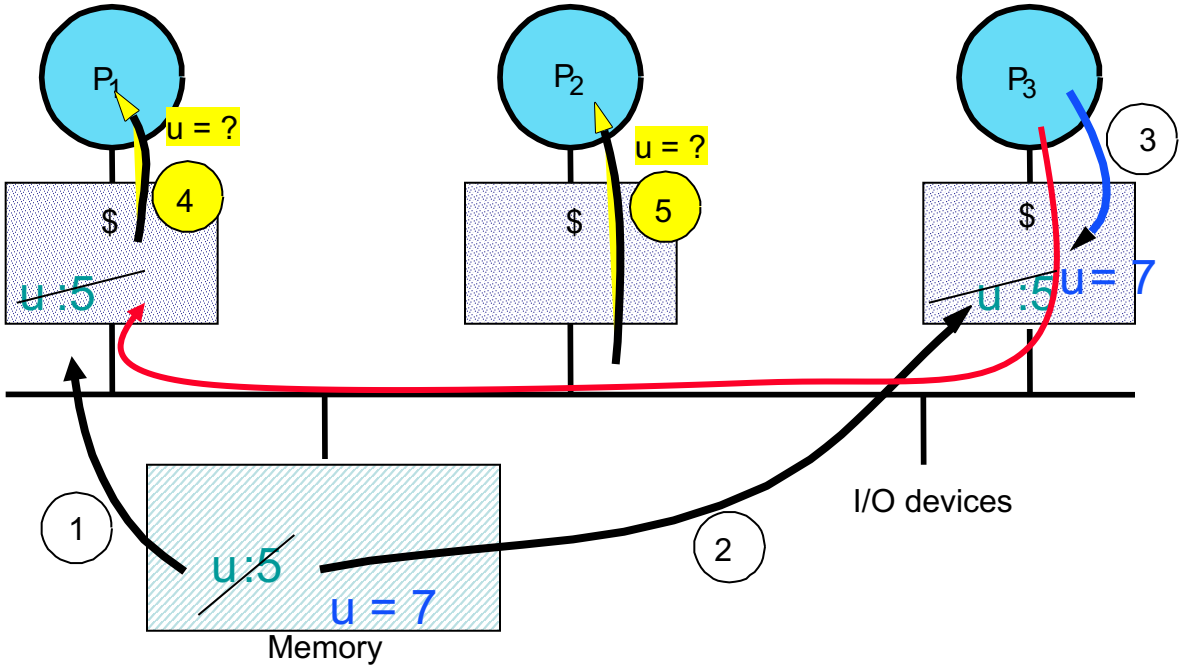

- Reading an address should return the last value written to that address
  - Easy in uniprocessors, except for I/O
- Too vague and simplistic

# Example: Cache Coherence Problem



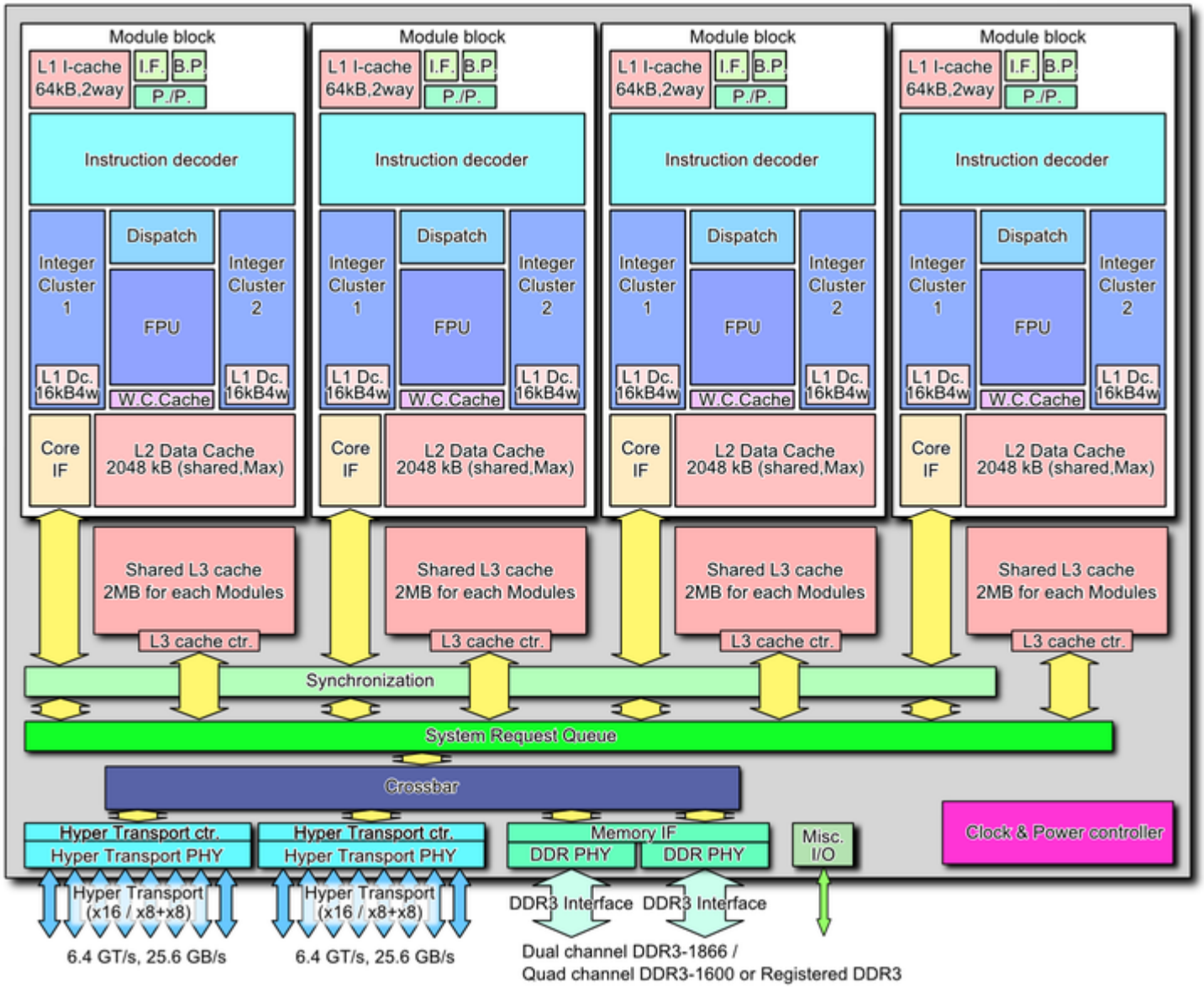

- Things to note:
  - Processors could see different values for u after event 3
  - With write back caches, value written back to memory depends on happenstance of which cache flushes or writes back value when
- How to fix with a bus: Coherence Protocol
  - Use bus to broadcast writes or invalidations
  - Simple protocols rely on presence of broadcast medium

# Example: Write-thru Invalidate



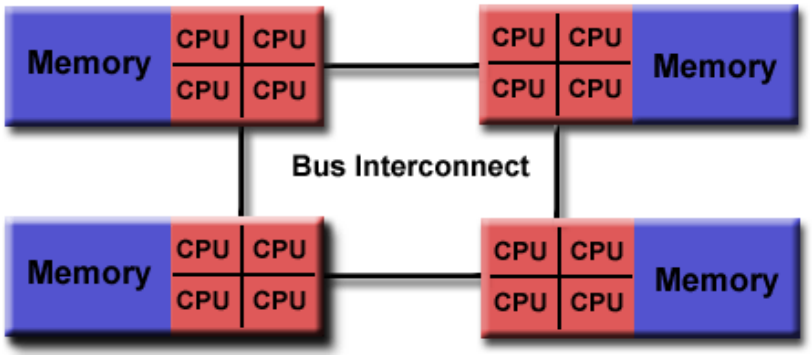

- Must invalidate before step 3
- Write update uses more broadcast medium BW
  - ☞ all recent MPUs use write invalidate
- Bus not scalable beyond about 64 processors (max)
  - Capacity, bandwidth limitations

# 8-core AMD Opteron (CC-NUMA)

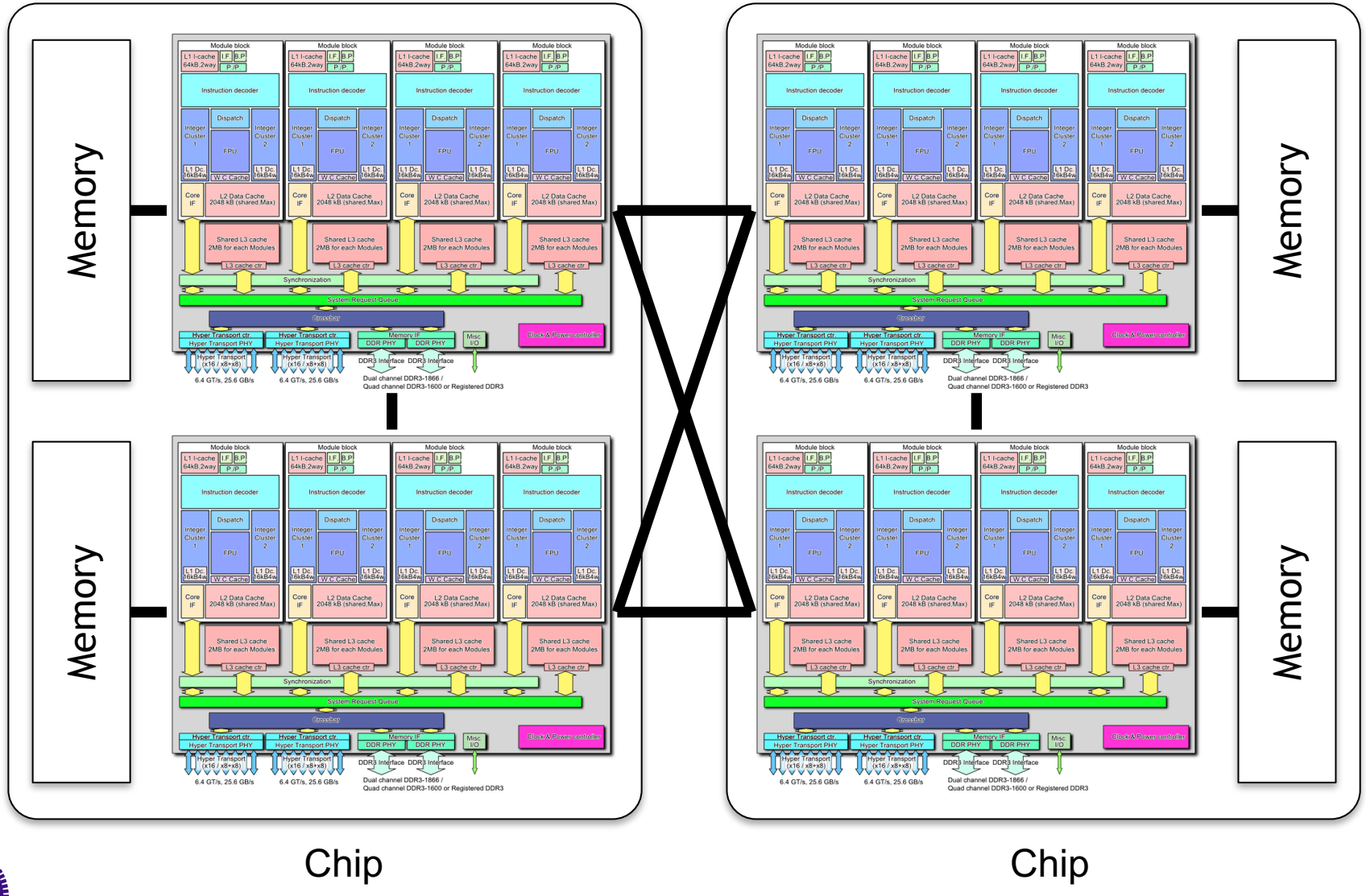


# Non-Uniform Memory Access (NUMA)

- Multiple "dies" on a single chip (i.e., single socket)
- Distributed shared memory
- Not all processors have equal access time to all memories
- Memory access across link is slower
- Same programming semantics as SMP



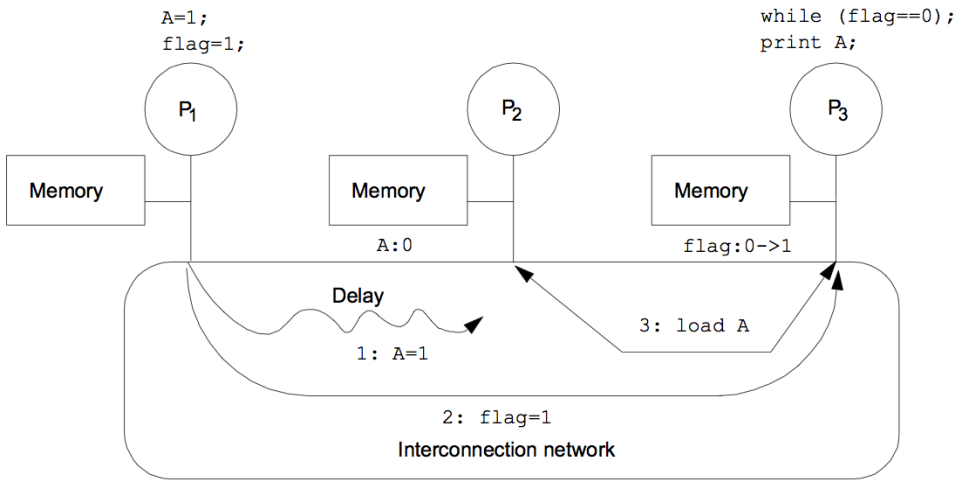
# 32-core Cray XE6 Supercomputer

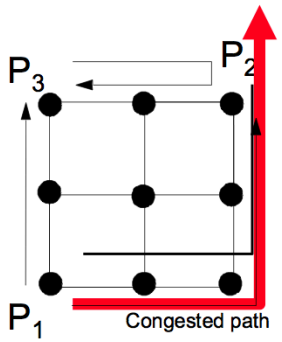


# Example: Coherence not Enough

| $P_1$ | $P_3$ |
|---|---|
| /*Assume initial value of `A` and `flag` is 0*/ | |
| `A = 1;` | `while (flag == 0);` /*spin idly*/ |
| `flag = 1;` | `print A;` |

- Expect memory to respect order between accesses to different locations issued by a given process
  - to preserve orders among accesses to same location by different processes
- Intuition not guaranteed by coherence

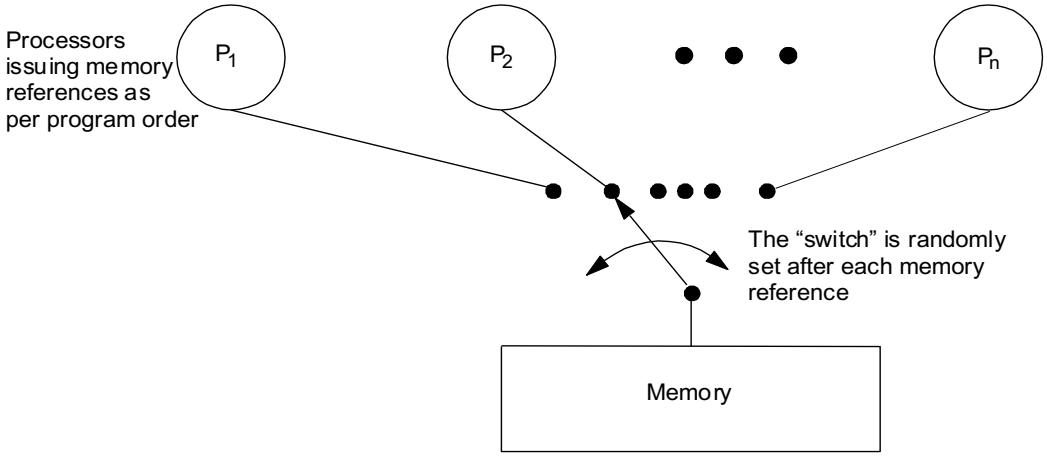


# Memory Consistency Model

- Specifies constraints on the order in which memory operations (from any process) can appear to execute with respect to one another
  - What orders are preserved?
  - Given a load, constrains the possible values returned by it
  - The memory consistency model defines when a written value must be seen by a following read instruction made by the other processors
- Without it, can't tell much about a single address space program's execution
- Implications for both programmer and system designer
  - Programmer uses to reason about correctness and possible results
  - System designer can use to constrain how much accesses can be reordered by compiler or hardware
- Contract between programmer and system

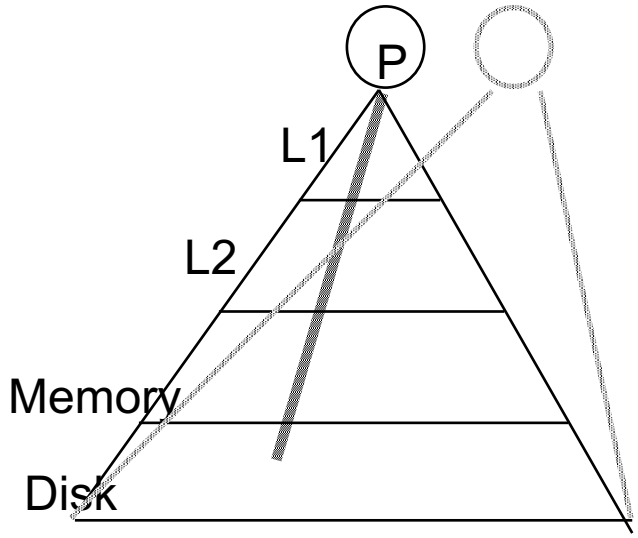
# Sequential Consistency




- Total order achieved by *interleaving* accesses from different processes
  - Maintains *program order*, and memory operations, from all processes, appear to [issue, execute, complete] atomically w.r.t. others
  - as if there were no caches, and a single memory

- "A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." [Lamport, 1979]

# Intuitive Memory Model




- Reading an address should return the last value written to that address
  - Easy in uniprocessors, except for I/O
- Too vague and simplistic

- Two issues
  - Coherence defines what values can be returned by a read
  - Consistency determines when a written value is returned by a read
- Coherence defines behavior to same location; Consistency defines behavior to other locations

# Notes on NUMA

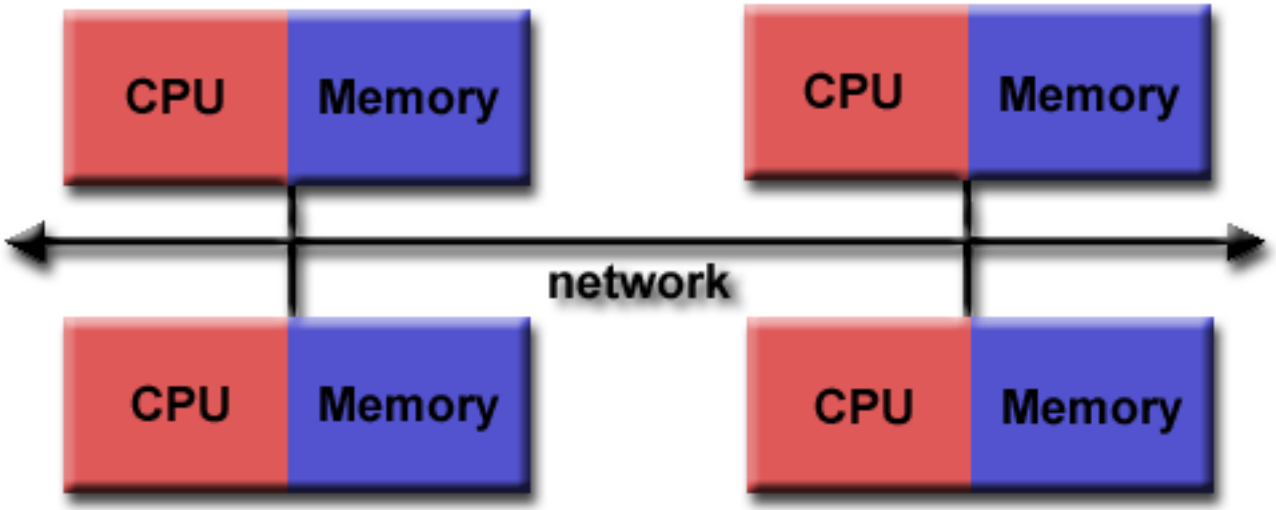
- Very difficult to program!
  - The tools don't help programmer account for NU!
  - Easy to write programs that work correctly!
  - More difficult to write programs that run fast!
- But, all multicore is NUMA!
  - Even SMPs today have NUMA properties!
  - Because of cache hierarchy!

# Outline

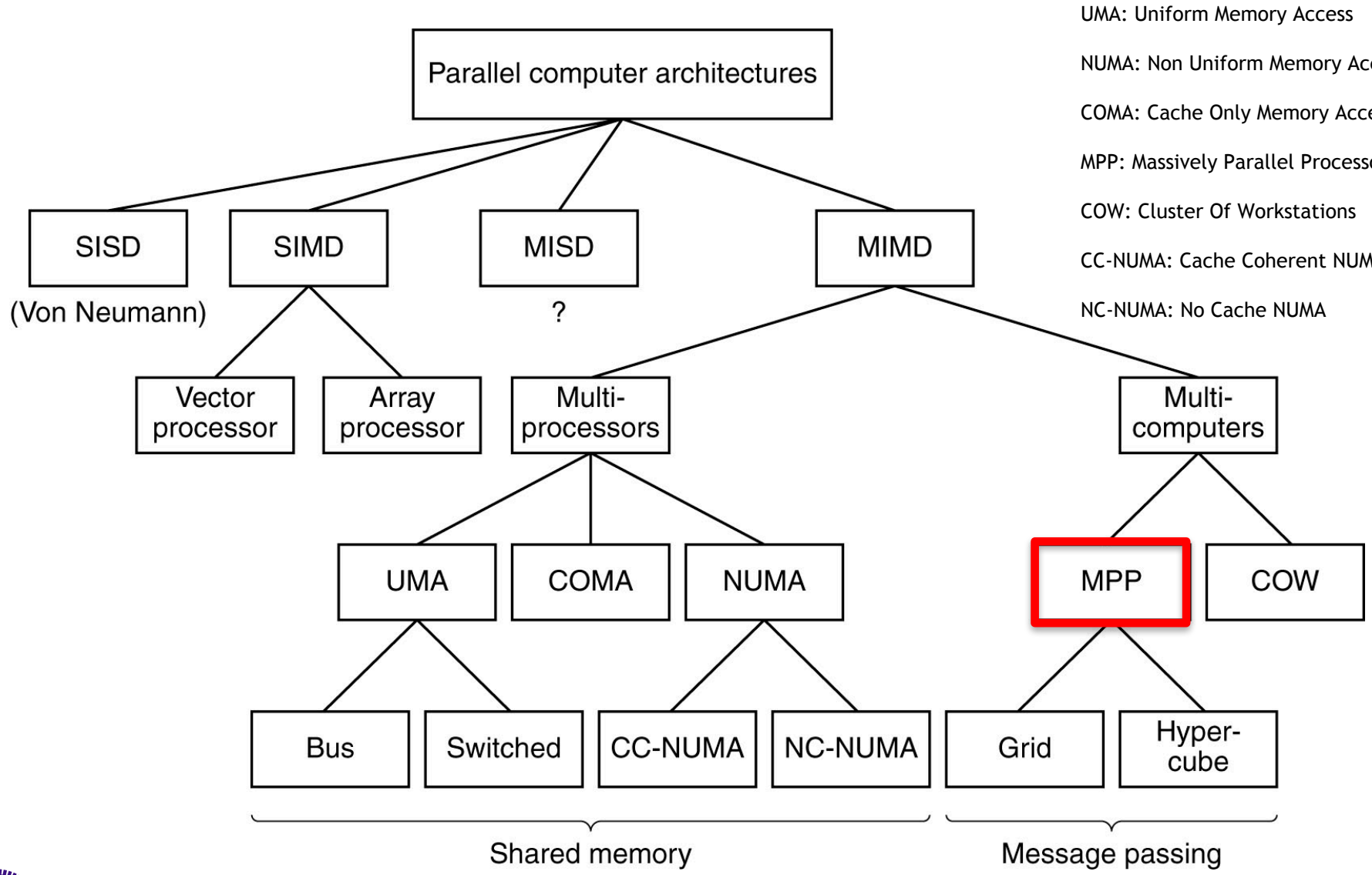
- Uniprocessor Parallelism
  - Pipelining, Out-of-order execution, Superscalar, VLIW
  - Vector Processing/SIMD
  - Multithreading
  - Uniprocessor Memory Systems
- Explicit Parallel Computer Architectures
  - Shared-memory Architectures
  - Distributed-memory Architectures
- Parallel Software
  - Shared-memory model
  - Distributed-memory model

# Distributed-Memory Systems

- Processors have their own local memory
- Memory addresses in one processor do not map to another processor
  - So there is no concept of global address space across all processors



# Taxonomy of Parallel Architectures




UMA: Uniform Memory Access

NUMA: Non Uniform Memory Access

COMA: Cache Only Memory Access

MPP: Massively Parallel Processor

COW: Cluster Of Workstations

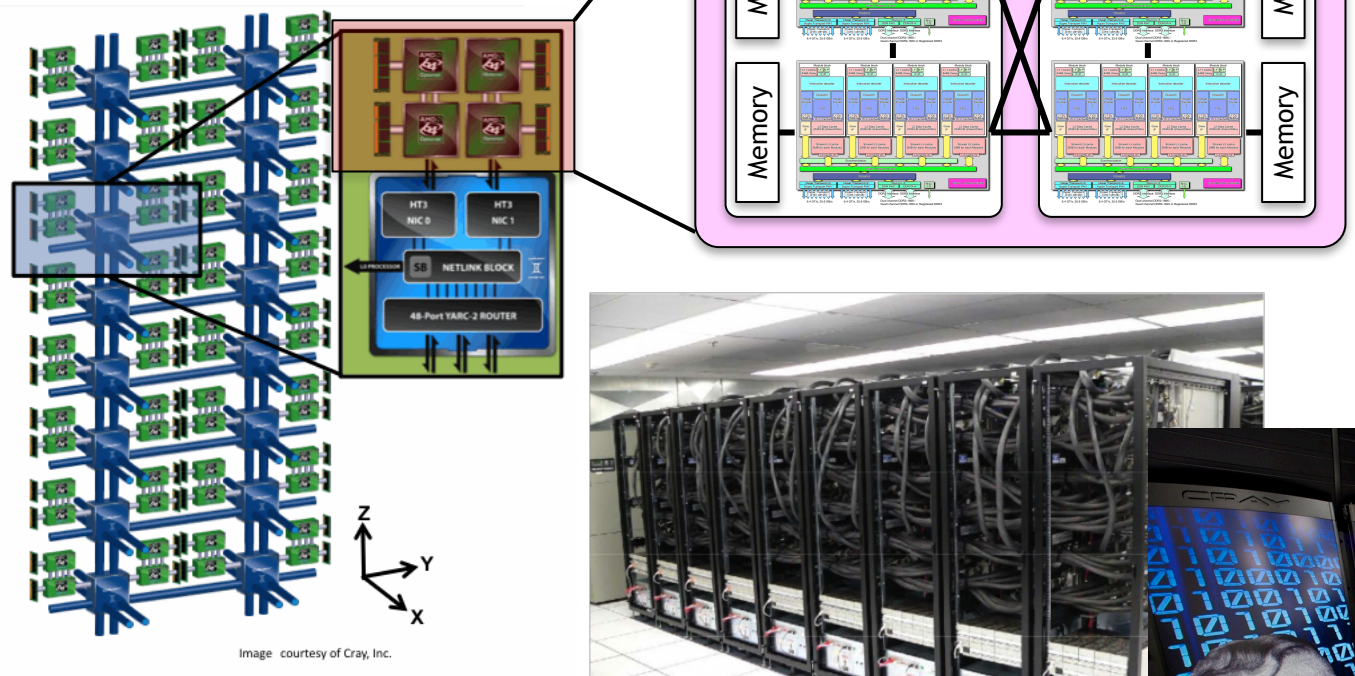
CC-NUMA: Cache Coherent NUMA

NC-NUMA: No Cache NUMA

Source: Tanenbaum, Structured Computer Organization

# Hopper (#62, 06/2015)

- Cray XE6, Opteron 6172 12C 2.10GHz
- 153,408 cores

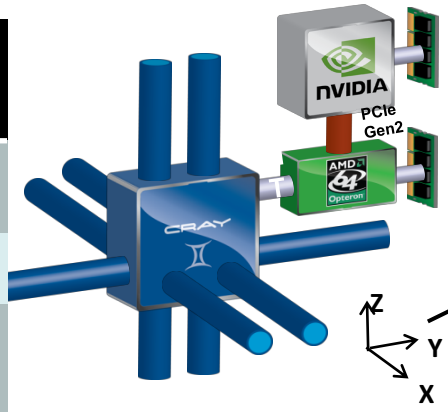



Cray XE6

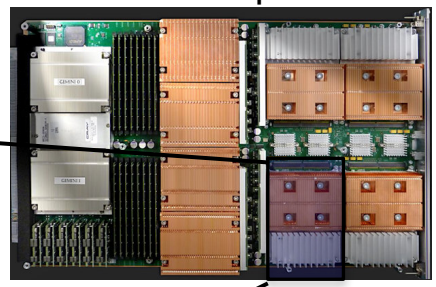
# Titan (#2, 06/2015)

- Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x
- 560,640 cores

| XK7 Compute Node Characteristics |
|---|
| AMD Series 6200 (Interlagos) |
| NVIDIA Kepler |
| Host Memory<br>32GB<br>1600 MT/s DDR3 |
| NVIDIA Tesla X2090 Memory<br>6GB GDDR5 capacity |
| Gemini High Speed Interconnect |
| Keplers in final installation |




Board: 4 Compute Nodes

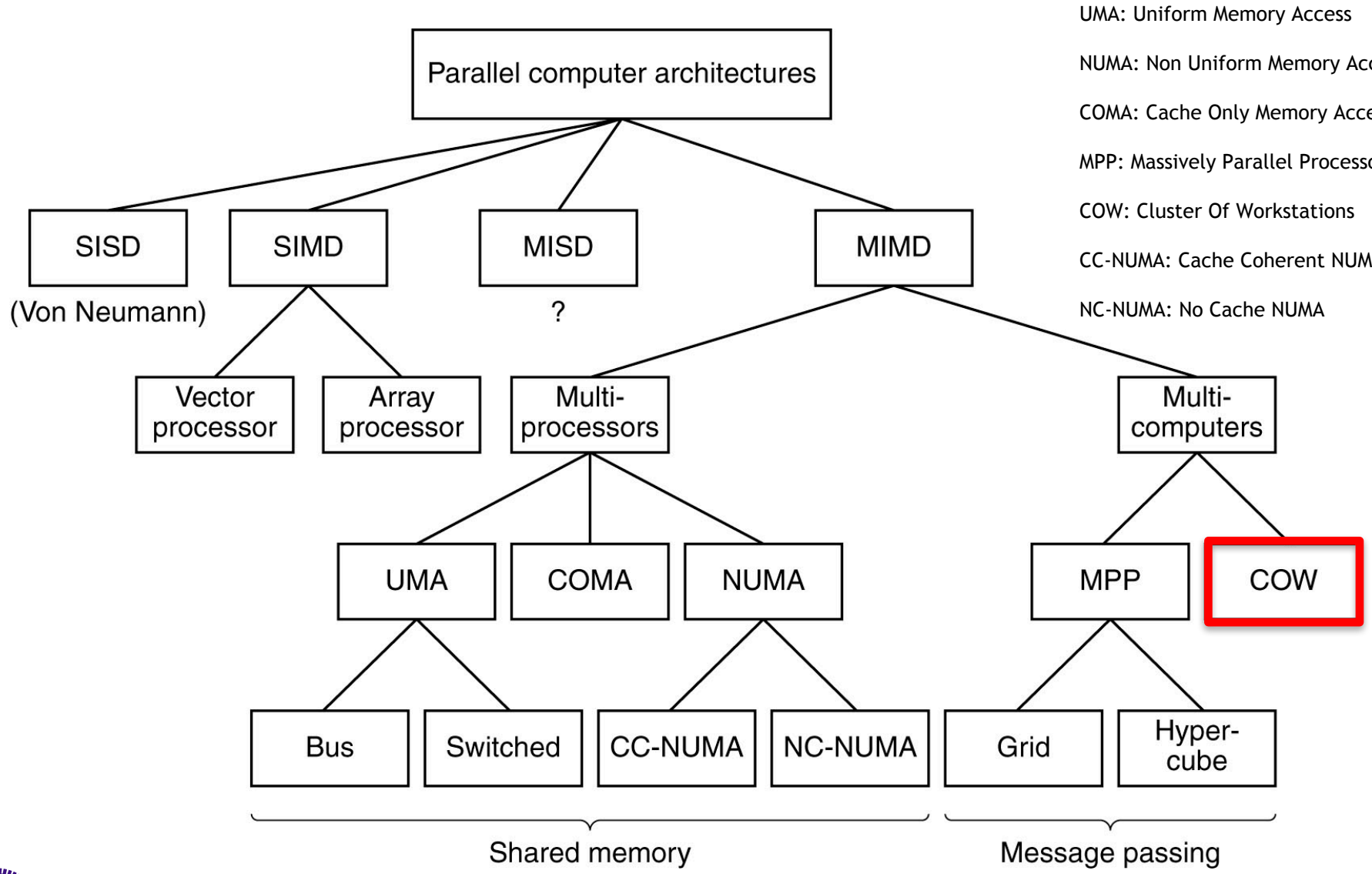


Cabinet: 24 Boards (96 Nodes)



System: 200 Cabinets (18,688 Nodes)

# Taxonomy of Parallel Architectures




UMA: Uniform Memory Access

NUMA: Non Uniform Memory Access

COMA: Cache Only Memory Access

MPP: Massively Parallel Processor

COW: Cluster Of Workstations

CC-NUMA: Cache Coherent NUMA

NC-NUMA: No Cache NUMA

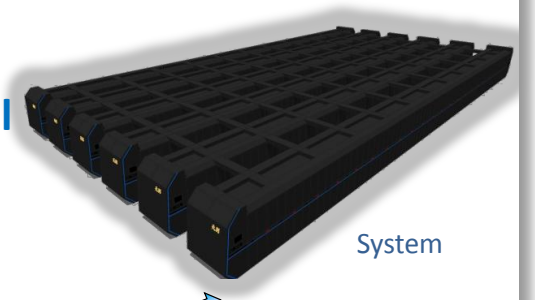
Source: Tanenbaum, Structured Computer Organization

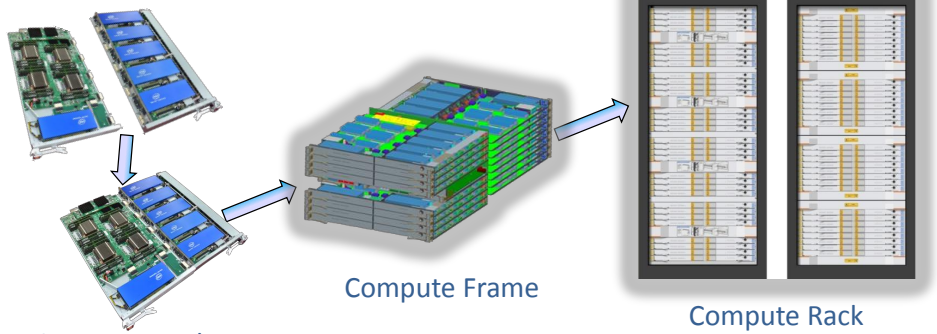
# Tianhe-2 (#1, 06/2015)

- 天河二號
- TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P

## From Chips to Entire System

- ◆16000 compute nodes in total
- ◆Frame: 32 compute Nodes
- ◆Rack: 4 Compute Frames
- ◆Whole System: 125 Racks




国防科学技术大学
National University of Defense Technology

Each node contains:
- 2 12-core Xeon
- 3 57-core Xeon Phi

| Items | Configuration |
|---|---|
| Processors | 32000 Intel Xeon CPUs + 48000 Xeon Phis + 4096 FT CPUs<br>Peak performance is 54.9PFlops, HPL |
| Interconnect | Proprietary high-speed interconnection network<br>TH Express-2 |

Rpeak: 54,902.4 TFlop/s

# Sunway TaihuLight (MPP) (#1, 06/2016)


Rpeak: 125,435.9 TFlop/s

- 40,960 Chinese-designed SW26010 manycore 64-bit RISC processors based on the Sunway architecture
- Total of 10,649,600 CPU cores across the entire system

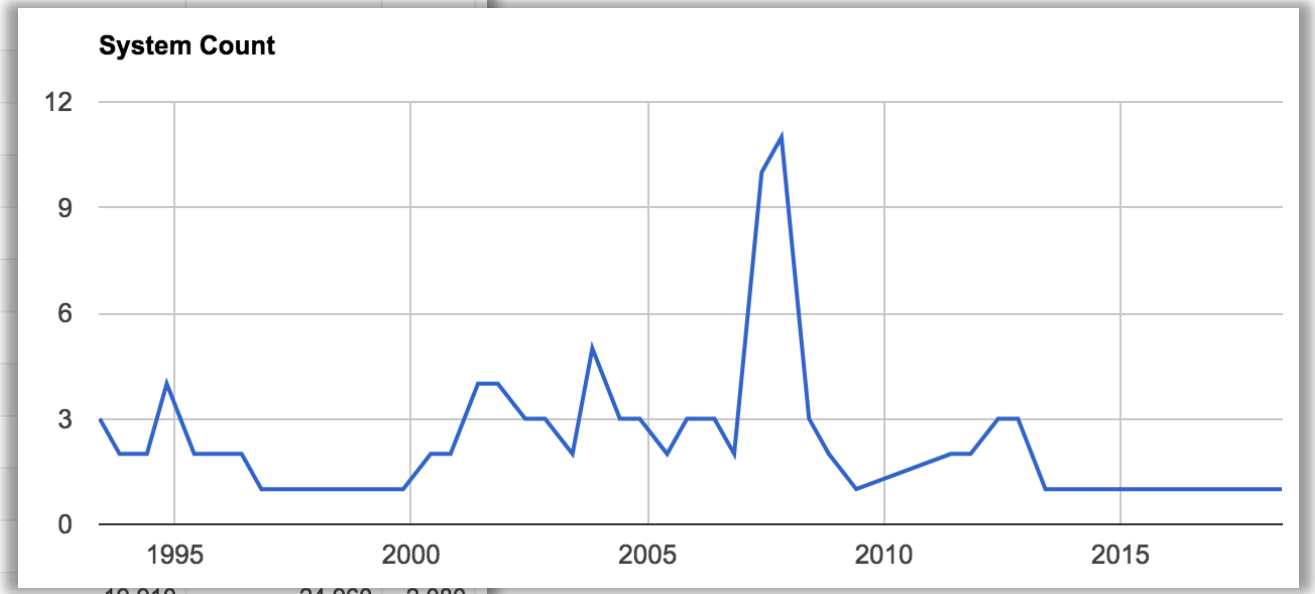
# Summit (#1, 06/2018)

- 2,282,544 CPU cores
  - 103,752 IBM POWER9 22-core CPUs
- 27,648 Nvidia Tesla V100 GPUs
- Site: DOE(Department of Energy)/SC(Office of Science)/Oak Ridge National Laboratory


Rpeak: 187,659 TFlop/s

# Supercomputers in Taiwan (within TOP500)

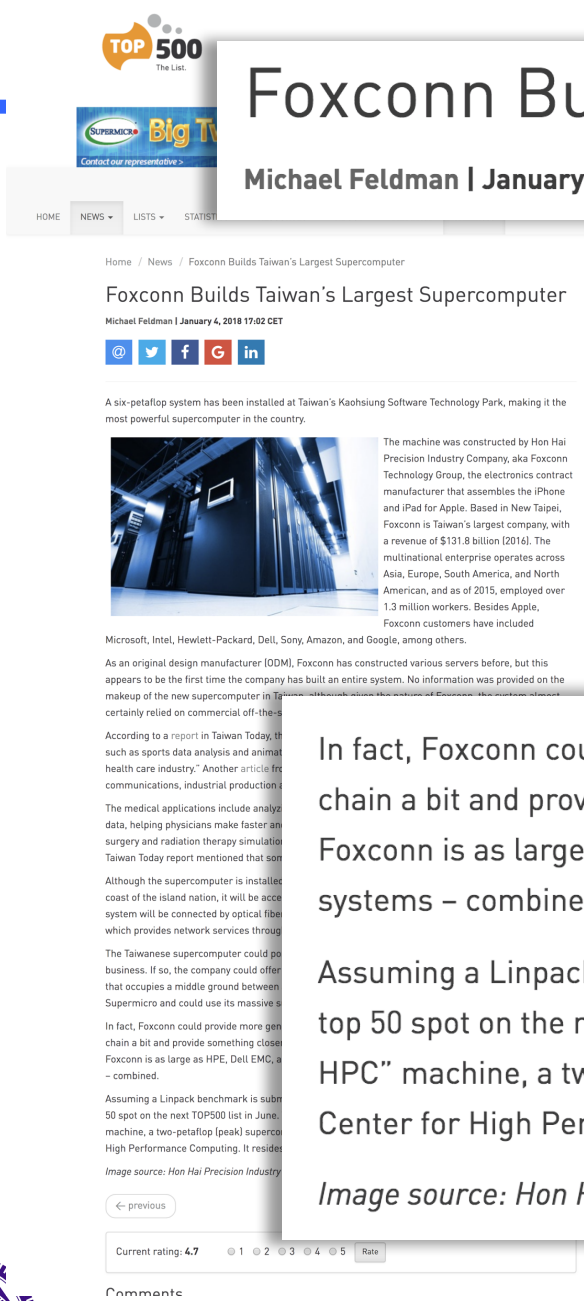
| List | Count | System Share (%) | Rmax (GFlops) | Rpeak (GFlops) | Cores |
|---|---|---|---|---|---|
| 6月 2018 | 1 | 0.2 | | | |
| 11月 2017 | | 0.2 | | | |
| 6月 2015 | 1 | 0.2 | | | |
| 11月 2014 | 1 | 0.2 | | | |
| 6月 2014 | 1 | 0.2 | | | |
| 11月 2013 | 1 | 0.2 | | | |
| 6月 2013 | 1 | 0.2 | | | |
| 11月 2012 | 3 | 0.6 | | | |
| 6月 2012 | 3 | 0.6 | | | |
| 11月 2011 | 2 | 0.4 | | | |
| 6月 2011 | 2 | 0.4 | | | |
| 6月 2009 | 1 | 0.2 | 19,910 | 24,960 | 2,080 |
| 11月 2008 | 2 | 0.4 | 33,592 | 48,960 | 4,080 |
| 6月 2008 | 3 | 0.6 | 45,677 | 69,792 | 5,816 |
| 11月 2007 | 11 | | | | |
| 6月 2007 | 10 | | | | |
| 11月 2006 | 2 | | | | |
| 6月 2006 | 3 | | | | |
| 11月 2005 | 3 | | | | |




| | | | | | | |
|---|---|---|---|---|---|---|
| 148 | National Center for High Performance Computing<br>Taiwan | **Taiwania** - PRIMERGY CX2550 M4/CX2560 M4/CX2570 M4, Xeon Gold 6148 20C 2.4GHz, Intel Omni-Path<br>Fujitsu | 27,200 | 1,325.2 | 2,089.0 | 332 |

# Foxconn Builds Taiwan's Largest Supercomputer

**Michael Feldman | January 4, 2018 17:02 CET**

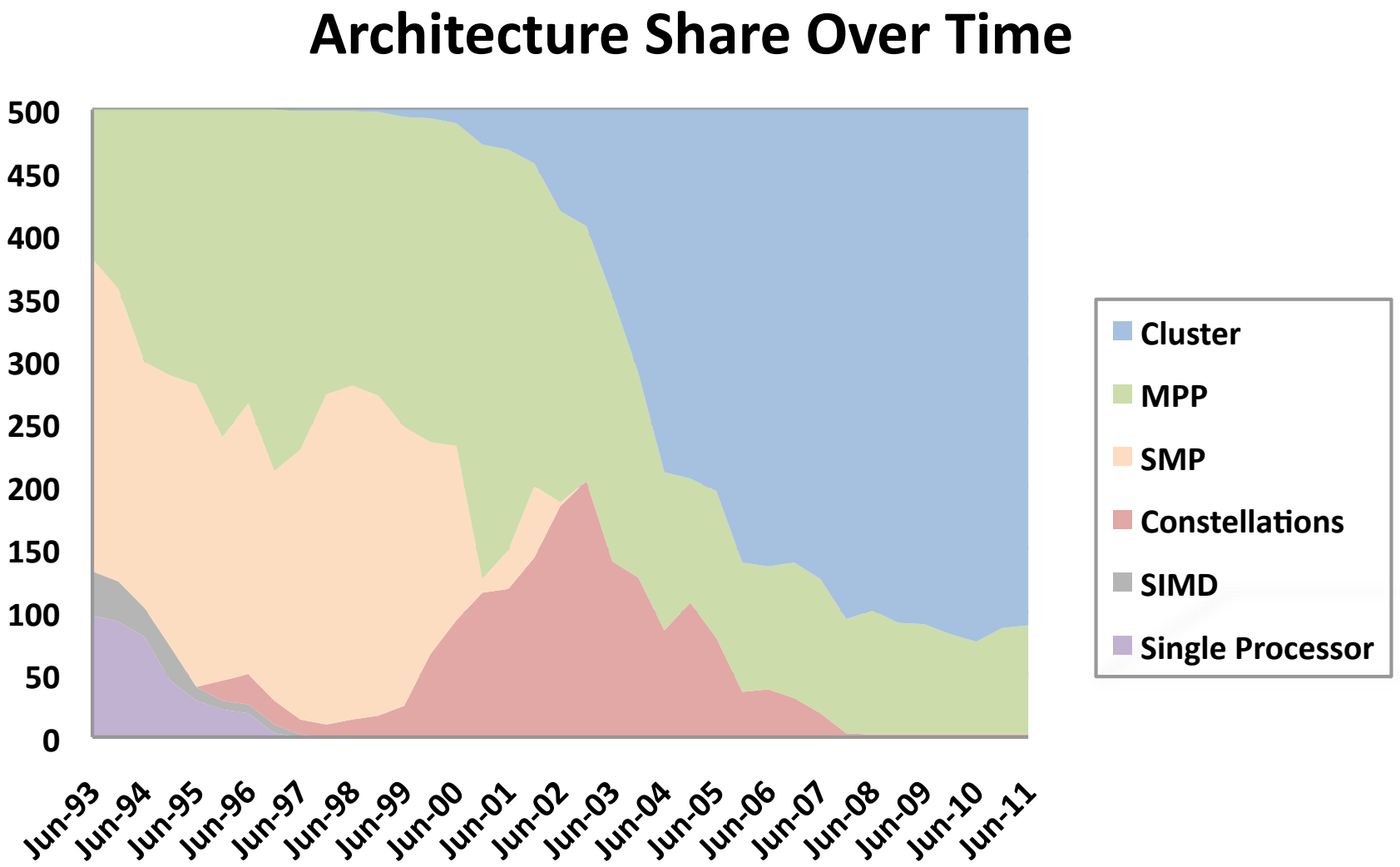



In fact, Foxconn could provide more general competition to HPC OEMs if it decided to move up the food chain a bit and provide something closer to a turnkey hardware-software solution. On a revenue basis, Foxconn is as large as HPE, Dell EMC, and Lenovo – currently, the three biggest suppliers of HPC systems – combined.

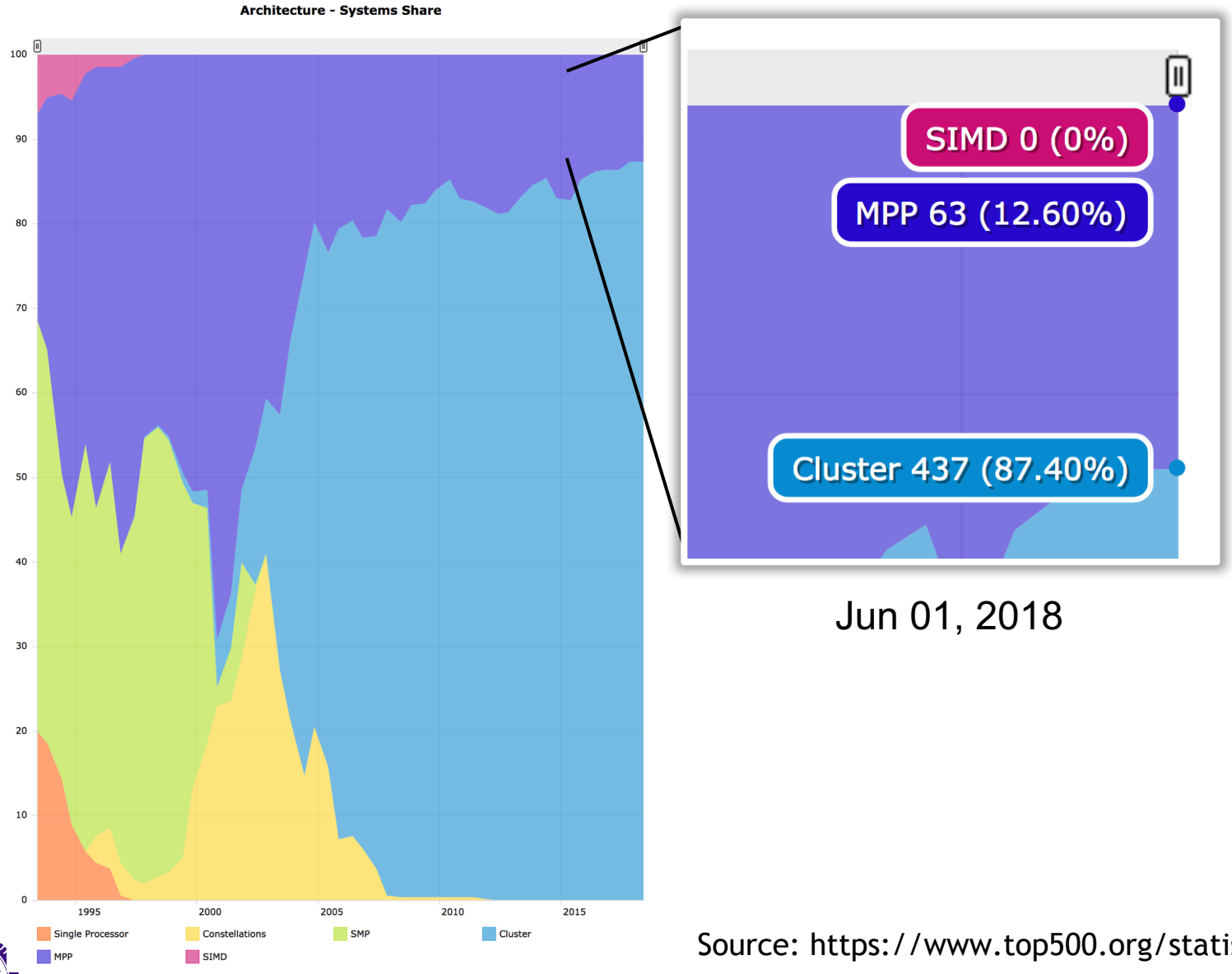
Assuming a Linpack benchmark is submitted, the Foxconn supercomputer will almost certainly land a top 50 spot on the next TOP500 list in June. Taiwan's current, and only entry in the list is the "Peta HPC" machine, a two-petaflop (peak) supercomputer built by Fujitsu and installed at the National Center for High Performance Computing. It resides at the number 95 position in the latest rankings.

*Image source: Hon Hai Precision Industry Company*

# Architectural Categories in the TOP500




# Architectural Categories in the TOP500

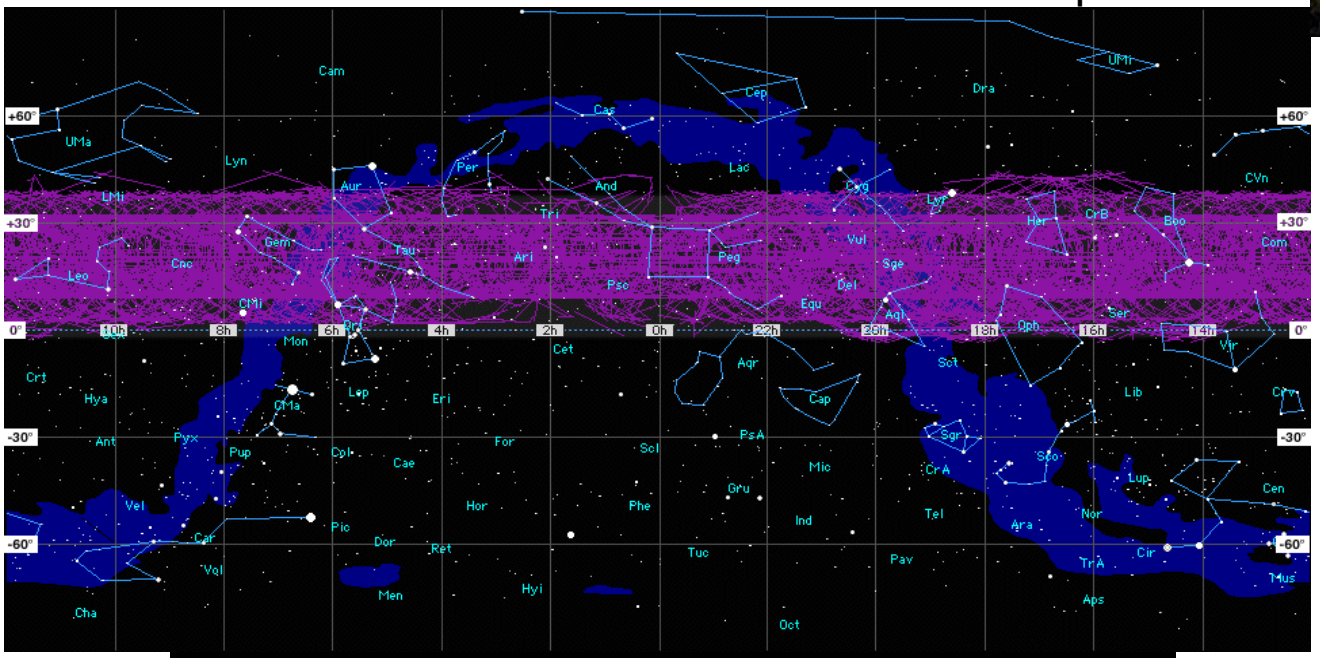



Jun 01, 2018

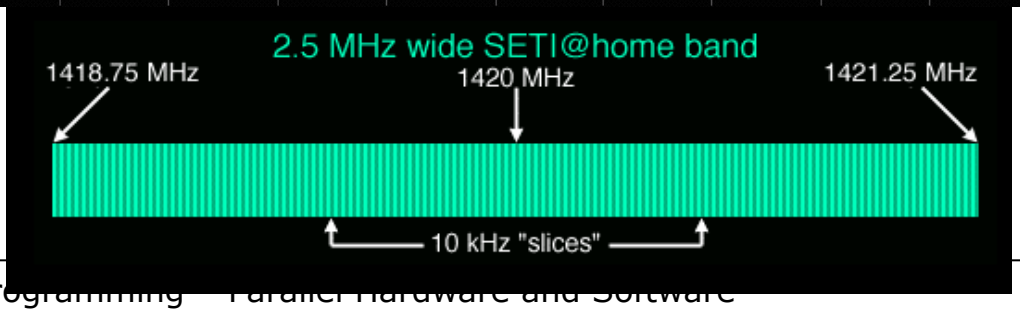
Source: https://www.top500.org/statistics/overtime/

# Internet/Grid Computing

- SETI@Home: Running on 3.3M hosts, 1.3M users (1/2013)
  - ~1000 CPU Years per Day (older data)
  - 485,821 CPU Years so far
- Sophisticated Data & Signal Processing Analysis
- Distributes Datasets from Arecibo Radio Telescope




**Next Step-**
**Allen Telescope Array**




**Google**
**"volunteer computing"**
**or "BOINC"**

# Overview of amazon web services

- Elastic Compute Cloud (EC2) Features:
  - Amazon EC2 allows you to dynamically allocate and terminate Linux "virtual" machines with a variety of hardware configurations
  - Pay only for what you use (i.e. machine hours and data transfer)
  - Ability to capture software configurations into Amazon Machine Images (AMI) for later use
  - AMI's can be used to launch multiple machines with identical software configurations

# Elatic Compute Cloud Hardware

## Standard Instances

One EC2 Compute Unit provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor.

| Instance | Arch | CPU | RAM | Storage | I/O Performance | Cost/hr |
|---|---|---|---|---|---|---|
| Small | 32bit | 1.0-1.2GHz | 1.7GB | 160GB | Moderate | $0.10/hr |
| Large | 64bit | 2.0-2.4GHz dual-core | 7.5GB | 860GB | High | $0.40/hr |
| Extra Large | 64bit | 2.0-2.4GHz quad-core | 15GB | 1.690TB | High | $0.80/hr |

## High CPU Instances

| Instance | Arch | CPU | RAM | Storage | I/O Performance | Cost/hr |
|---|---|---|---|---|---|---|
| Medium | 32bit | 2.5-3.0GHz dual-core | 1.7GB | 350GB | Moderate | $0.20/hr |
| Extra Large | 64bit | 2.5-3.0GHz quad-core(ht) | 7GB | 1.690TB | High | $0.80/hr |

# Driving EC2 using Python

- MapReduce (Hadoop + Python)
- Python MPI options (mpi4py, pyMPI, ...)
- Wrap existing C++/Fortran libs (ScaLAPACK,PETSc, ...) Pyro
- Twisted
- IPython1

# Outline

- Uniprocessor Parallelism
  - Pipelining, Out-of-order execution, Superscalar, VLIW
  - Vector Processing/SIMD
  - Multithreading
  - Uniprocessor Memory Systems
- Explicit Parallel Computer Architectures
  - Shared-memory Architectures
  - Distributed-memory Architectures
- Parallel Software
  - Shared-memory model
  - Distributed-memory model

# Parallel Software

- We can no longer rely on hardware and compilers to provide a steady increase in application performance
- If we're to continue to have routine increases in application performance, software developers must learn to write applications that exploit shared- and distributed-memory architectures

# Shared-Memory Model

- In shared-memory programs, variables can be **shared** or **private**
  - Shared variables can be read or written by any thread
  - Private variables can ordinarily only be accessed by one thread

# Explicit v.s. Implicit Threads Programming

- Two threads programming styles:
  - E: Explicit
    - User creates threads using threads API
    - E.g., “Windows threads” and “POSIX Threads”
  - I: Implicit
    - User uses high-level directives to create threads with the help of tool chain
    - E.g., OpenMP, Intel Thread Building Block (TBB)
- Examples:
  - Thread creation
    - E: Programmers create threads and manage threads
    - I: Thread pools created and maintained by library
  - Assigning computation
    - E: Programmer inserts work division logic to assign tasks to threads
    - I: Work divided by library or additional pragma options
  - Wait for threads to complete
    - E: API call to pause waiting thread and detect thread termination
    - I: Implicit barrier at end of threading constructs

# Nondeterminism

- In any MIMD system in which the processors execute asynchronously it is likely that there will be nondeterminism
- A computation is nondeterministic if a given input can result in different outputs
- We often say the program has a **race condition**
- The most commonly used mechanism for ensuring mutual exclusion
  - Mutex
  - Busy-waiting
  - Semaphores
  - Monitor
  - Transactional memory

# Thread Safety

- Thread Safe:
  - Many serial functions can be used safely in multithreaded programs
- The most important exception for C programmers occurs in functions that make use of *static* local variables
  - Since each thread has its own stack, ordinary C local variables are private
  - However, recall that a static variable that's declared in a function persists from one call to the next
  - Thus, static variables are effectively shared among any threads that call the function, and this can have unexpected and unwanted consequences

# Outline

- Uniprocessor Parallelism
  - Pipelining, Out-of-order execution, Superscalar, VLIW
  - Vector Processing/SIMD
  - Multithreading
  - Uniprocessor Memory Systems
- Explicit Parallel Computer Architectures
  - Shared-memory Architectures
  - Distributed-memory Architectures
- Parallel Software
  - Shared-memory model
  - Distributed-memory model

# Distributed-Memory Model

- In distributed-memory programs, the cores can directly access only their own, private memories
- By far the most widely used is message-passing
- A message-passing API provides (at a minimum) a send and a receive function
- Processes typically identify each other by ranks in the range 0, 1, … , p-1, where p is the number of processes

# Example: Message Passing

```
char message[100];
. . .
my_rank = Get_rank();
if (my_rank == 1) {
   sprintf(message, "Greetings from process 1");
   Send(message, MSG_CHAR, 100, 0);
} else if (my_rank == 0) {
   Receive(message, MSG_CHAR, 100, 1);
   printf("Process 0 > Received: %s\n", message);
}
```

- The two processes are using the same executable, but carrying out different actions
- The variable message refers to different blocks of memory on the different processes

# Message-Passing – Additional Functions

- Typical message-passing APIs also provide a wide variety of additional functions
  - Broadcast:
    - In which a single process transmits the same data to all the processes
  - Reduction:
    - In which results computed by the individual processes are combined into a single result

# One-Sided Communication

- One-sided communication (remote memory access)
  - A single process calls a function, which updates either local memory with a value from another process or remote memory with a value from the calling process
- This can simplify communication

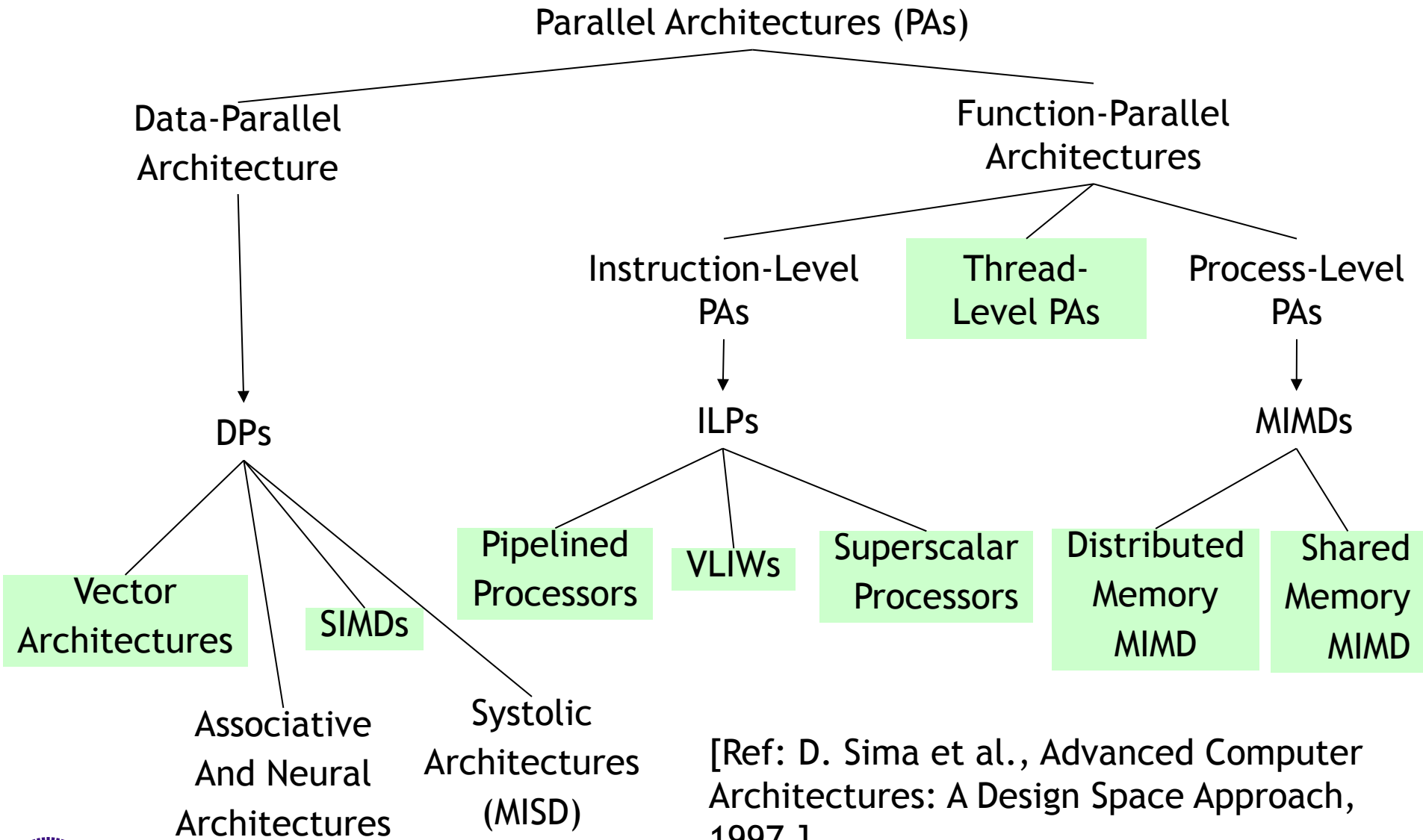
# Partitioned Global Address Space Languages

- A number of groups are developing parallel programming languages that allow the user to use some shared-memory techniques for programming distributed-memory hardware
- Issue: Accessing remote memory can take hundreds or even thousands of times longer than accessing local memory

# Programming Hybrid Systems

- It is possible to program systems such as clusters of multicore processors using a combination of a shared-memory API on the nodes and a distributed-memory API for internode communication
- Since the complexity of this "hybrid" API makes program development extremely difficult

# Summary of Parallel Architectures




[Ref: D. Sima et al., Advanced Computer Architectures: A Design Space Approach, 1997.]

# References

- Short Course on Parallel Programming (UCB)
  - http://parlab.eecs.berkeley.edu/2012bootcamp
- Carnegie Mellon University's public course, Parallel Computer Architecture and Programming, (CS 418)
  - http://www.cs.cmu.edu/afs/cs/academic/class/15418-s11/public/lectures/
- Blaise Barney, Lawrence Livermore National Laboratory, "Introduction to Parallel Computing"
  - https://computing.llnl.gov/tutorials/parallel_comp/