

## Computer Organization (Architecture)

### Lecture 3

Dr Jeff Drobman

website



[drjeffsoftware.com/classroom.html](http://drjeffsoftware.com/classroom.html)

email



[jeffrey.drobman@csun.edu](mailto:jeffrey.drobman@csun.edu)

# Index (Vol. 3)

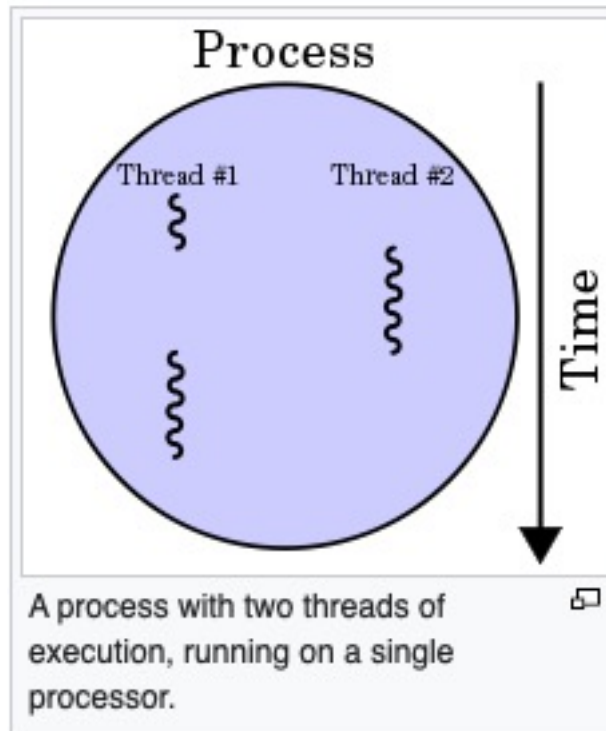
- ❖ Multi-Threading (ILP) → slide 3
  - ❑ Superscalar → slide 25
  - ❑ Hyperthreading → slide 37
  - ❑ Perf/Benchmarks → slide 46
  - ❑ x86: AMD vs Intel → slide 69
  - ❑ MT in Java → slide 78
- ❖ Data Parallelism (DLP) → slide 99
  - ❑ Flynn, SIMD, MMX/AVX Vectors → slide 104
  - ❑ AMX → slide 134
  - ❑ VLIW/IMC → slide 148 (see separate slide set)
- ❖ Micro arch. → slide 156 (see separate slide set)
- ❖ GPU (Graphics) → slide 157 (see separate slide set)
- ❖ DSP → slide 158

# Section

# Multi-Threading

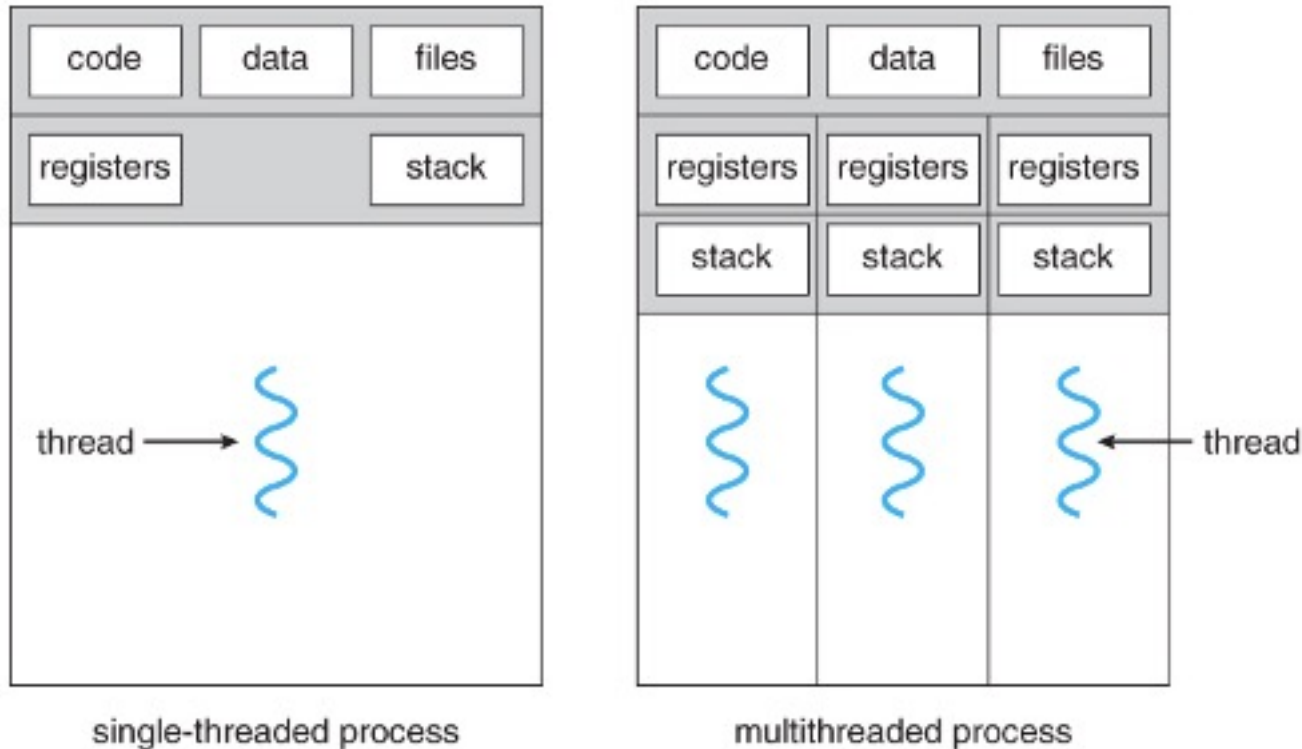
In **computer architecture**, **multithreading** is the ability of a **central processing unit** (CPU) (or a single core in a **multi-core processor**) to provide multiple **threads of execution** concurrently, supported by the **operating system**. This approach differs from **multiprocessing**. In a multithreaded application, the threads share the resources of a single or multiple cores, which include the computing units, the **CPU caches**, and the **translation lookaside buffer** (TLB).

Where multiprocessing systems include multiple complete processing units in one or more cores, multithreading aims to increase utilization of a single core by using **thread-level parallelism**, as well as **instruction-level parallelism**. As the two techniques are complementary, they are sometimes combined in systems with multiple multithreading CPUs and with CPUs with multiple multithreading cores.



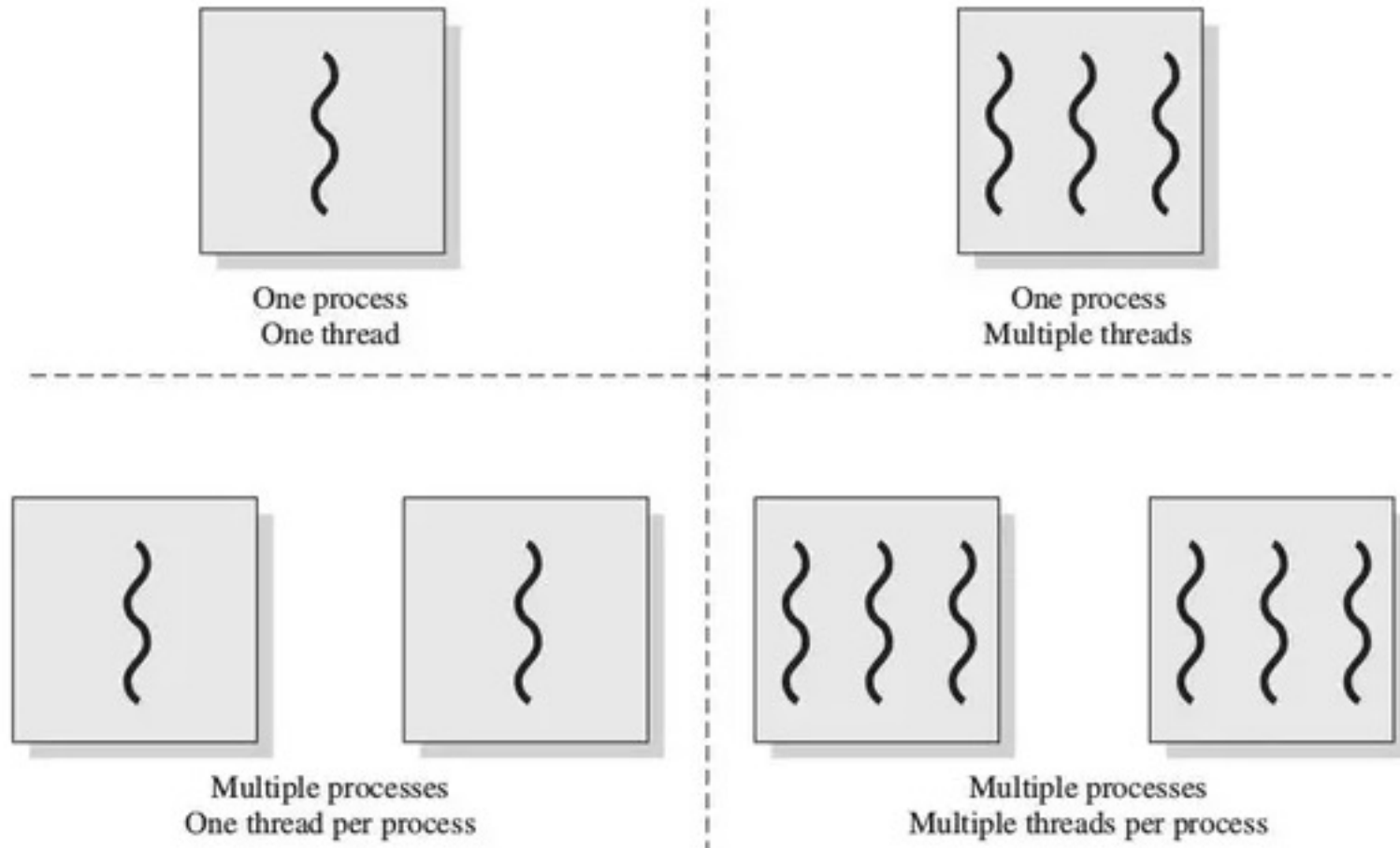


# MT Diagram

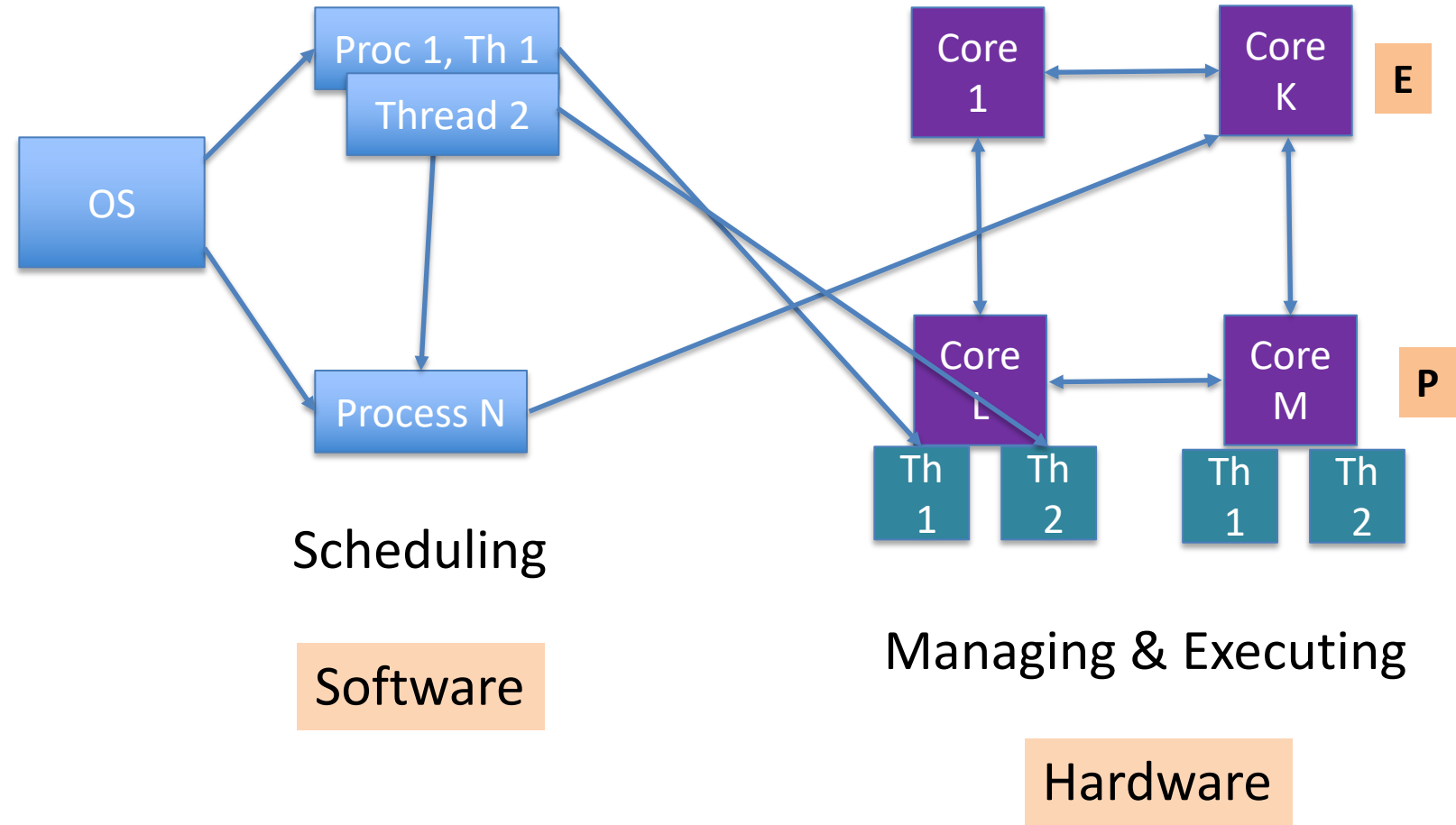


*Multithreading Diagram 1*

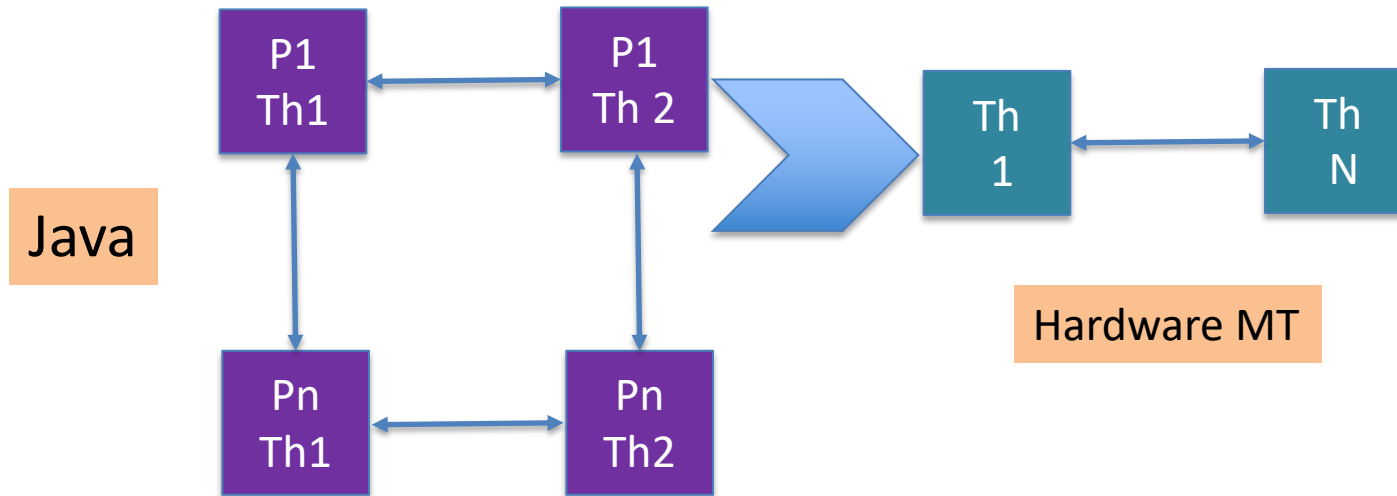
# Threads



# Cores + Threads



# Java Threads



Assignment

# Threads

**Which one is better, a processor with 4 cores, 8 threads and 1.8 GHz or a processor with 4 cores, 4 threads and 3.6 GHz?**



**Jeff Drobman** · just now

Lecturer at California State University, Northridge (2016–present)

the latter if MT is "temporal": faster clock frequency gives a 2x boost to all threads running. 4 cores can only run 4 threads at a time for "temporal" MT; but can run all 8 threads simultaneously with "SMT" — so 1st case is equal.

# MT Types

## 1. Temporal (TMT)

*a. Coarse-grained*

*b. Fine-grained (Interleaved)*

➤ 1 at a time

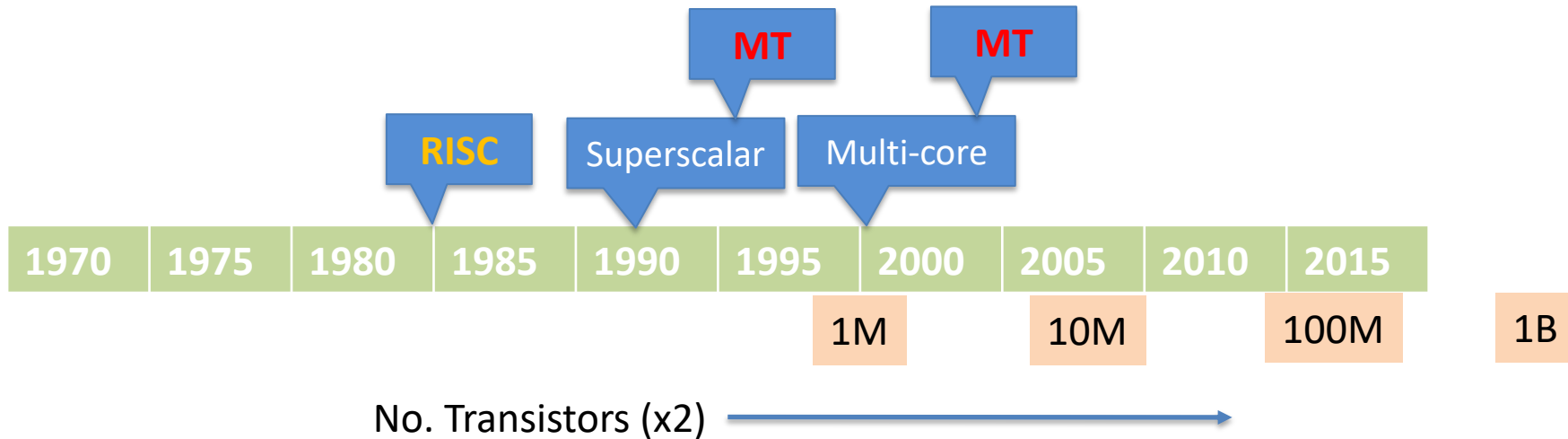
## 2. Simultaneous (SMT)

*a. SMT*

*b. HTT*

➤ >1 at a time

# MT Timeline



# MT Summary

## ❖ Temporal

### ☐ Sequential

#### ☐ Coarse grained

- Run until stall, then switch
- Need to use algorithms (fairness)

#### ☐ Fine grained

- **Interleaved** (per cycle)
- Round-robin
- Barrel processor

## ❖ SMT (AMD)

### ☐ Parallel

#### ☐ Superscalar

## ❖ Hyper (Intel)

### ☐ SMT

#### ☐ Virtual pipelines with **scheduler**

❖ ARM?  
❖ MIPS?



# Shared Resources

---

## ❖ Registers

- ☐ Shared (*w/renaming*)
- ☐ Partitioned
- ☐ Replicated

## ❖ Caches (L1 I+D, TLB)

- ☐ Shared (*w/thrashing*)
- ☐ *Virtual* Partitioned: Set Associative
- ☐ *Physical* Partitioned (Replicated)

## ❖ EU's

- ☐ Int ALU/FPU
- ☐ Ld/St
- ☐ Br

## Overview [\[ edit \]](#)

The multithreading [paradigm](#) has become more popular as efforts to further exploit [instruction-level parallelism](#) have stalled since the late 1990s. This allowed the concept of [throughput computing](#) to re-emerge from the more specialized field of [transaction processing](#). Even though it is very difficult to further speed up a single thread or single program, most computer systems are actually multitasking among multiple threads or programs. Thus, techniques that improve the throughput of all tasks result in overall performance gains.

Two major techniques for throughput computing are *multithreading* and *multiprocessing*.

## Advantages [\[ edit \]](#)

If a thread gets a lot of [cache misses](#), the other threads can continue taking advantage of the unused computing resources, which may lead to faster overall execution, as these resources would have been idle if only a single thread were executed. Also, if a thread cannot use all the computing resources of the CPU (because instructions depend on each other's result), running another thread may prevent those resources from becoming idle.

## Disadvantages [\[ edit \]](#)

Multiple threads can interfere with each other when sharing hardware resources such as caches or [translation lookaside buffers](#) (TLBs). As a result, execution times of a single thread are not improved and can be degraded, even when only one thread is executing, due to lower frequencies or additional pipeline stages that are necessary to accommodate thread-switching hardware.

Overall efficiency varies; Intel claims up to 30% improvement with its [Hyper-Threading Technology](#),<sup>[1]</sup> while a synthetic program just performing a loop of non-optimized dependent floating-point operations actually gains a 100% speed improvement when run in parallel. On the other hand, hand-tuned [assembly language](#) programs using [MMX](#) or [AltiVec](#) extensions and performing data prefetches (as a good video encoder might) do not suffer from cache misses or idle computing resources. Such programs therefore do not benefit from hardware multithreading and can indeed see degraded performance due to contention for shared resources.

From the software standpoint, hardware support for multithreading is more visible to software, requiring more changes to both application programs and operating systems than multiprocessing. Hardware techniques used to support [multithreading](#) often parallel the software techniques used for [computer multitasking](#). Thread scheduling is also a major problem in multithreading.

# Temporal MT

COMP222



1 at a time

Wiki

**Coarse-grained multithreading** [ [edit](#) ]

The simplest type of multithreading occurs when one thread runs until it is blocked by an event that normally would create a long-latency stall. Such a stall might be a cache miss that has to access off-chip memory, which might take hundreds of CPU cycles for the data to return. Instead of waiting for the stall to resolve, a threaded processor would switch execution to another thread that was ready to run. Only when the data for the previous thread had arrived, would the previous thread be placed back on the list of [ready-to-run](#) threads.

For example:

1. Cycle  $i$ : instruction  $j$  from thread  $A$  is issued.
2. Cycle  $i + 1$ : instruction  $j + 1$  from thread  $A$  is issued.
3. Cycle  $i + 2$ : instruction  $j + 2$  from thread  $A$  is issued, which is a load instruction that misses in all caches.
4. Cycle  $i + 3$ : thread scheduler invoked, switches to thread  $B$ .
5. Cycle  $i + 4$ : instruction  $k$  from thread  $B$  is issued.
6. Cycle  $i + 5$ : instruction  $k + 1$  from thread  $B$  is issued.



Run until blocked

## 1. Temporal

- a. Coarse-grained
- b. Interleaved

**Temporal multithreading** is one of the two main forms of multithreading that can be implemented on computer processor hardware, the other being simultaneous multithreading. The distinguishing difference between the two forms is the maximum number of concurrent threads that can

**Interleaved multithreading** [ [edit](#) ]

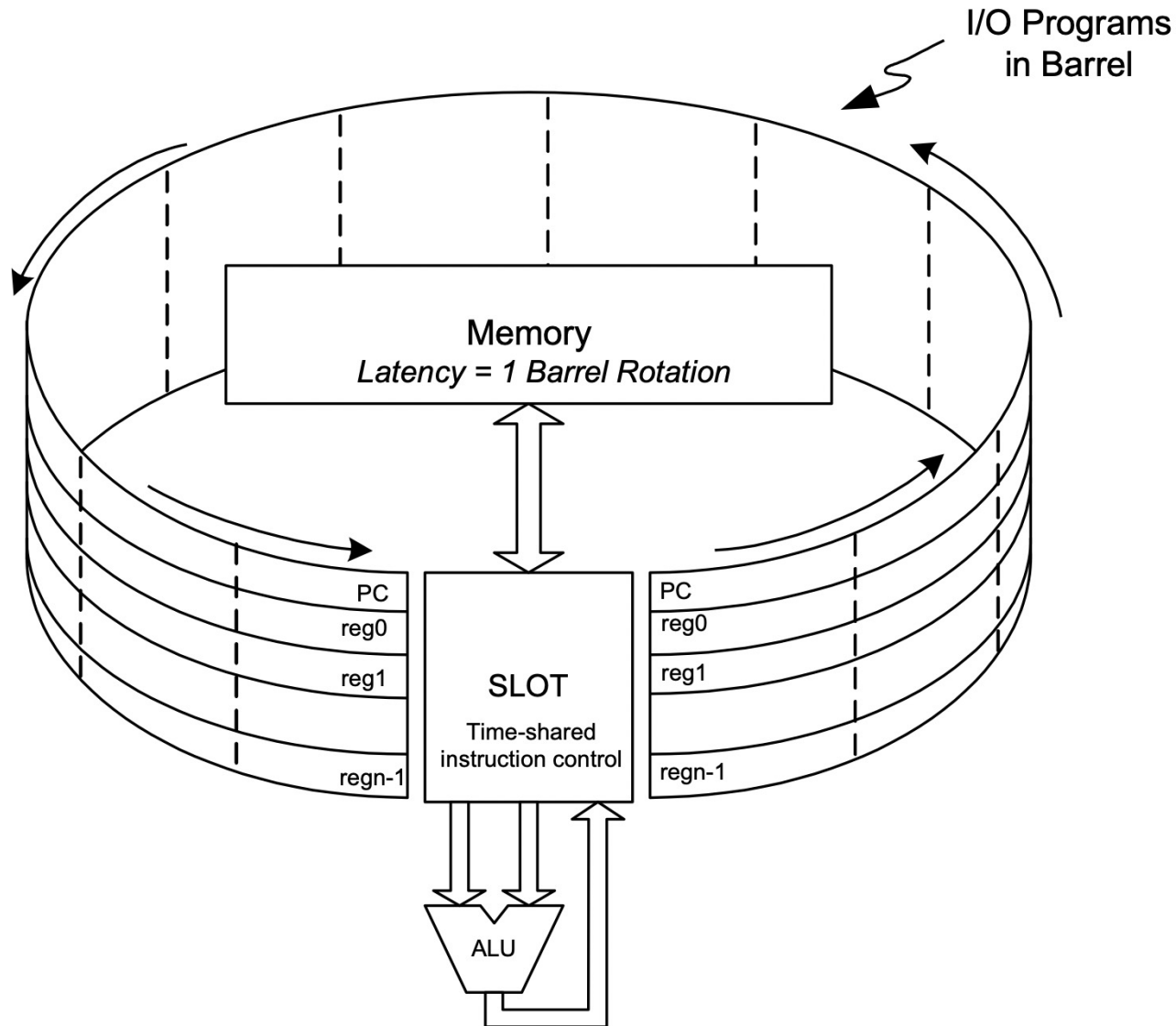
Main article: [Barrel processor](#)

The purpose of interleaved multithreading is to remove all [data dependency](#) stalls from the execution [pipeline](#). Since one thread is relatively independent from other threads, there is less chance of one instruction in one pipelining stage needing an output from an older instruction in the pipeline. Conceptually, it is similar to [preemptive](#) multitasking used in operating systems; an analogy would be that the time slice given to each active thread is one CPU cycle.



Interleaved = Time sliced (time-sharing)

# Interleaved Barrel Proc



## Simultaneous multithreading [\[ edit \]](#)

Main article: [Simultaneous multithreading](#)

The most advanced type of multithreading applies to [superscalar processors](#). Whereas a normal superscalar processor issues multiple instructions from a single thread every CPU cycle, in simultaneous multithreading (SMT) a superscalar processor can issue instructions from multiple threads every CPU cycle. Recognizing that any single thread has a limited amount of [instruction-level parallelism](#), this type of multithreading tries to exploit parallelism available across multiple threads to decrease the waste associated with unused issue slots.

For example:

1. Cycle  $i$ : instructions  $j$  and  $j + 1$  from thread  $A$  and instruction  $k$  from thread  $B$  are simultaneously issued.
2. Cycle  $i + 1$ : instruction  $j + 2$  from thread  $A$ , instruction  $k + 1$  from thread  $B$ , and instruction  $m$  from thread  $C$  are all simultaneously issued.
3. Cycle  $i + 2$ : instruction  $j + 3$  from thread  $A$  and instructions  $m + 1$  and  $m + 2$  from thread  $C$  are all simultaneously issued.

To distinguish the other types of multithreading from SMT, the term "[temporal multithreading](#)" is used to denote when instructions from only one thread can be issued at a time.

In addition to the hardware costs discussed for interleaved multithreading, SMT has the additional cost of each pipeline stage tracking the thread ID of each instruction being processed. Again, shared resources such as caches and TLBs have to be sized for the large number of active threads being processed.

Implementations include [DEC](#) (later [Compaq](#)) [EV8](#) (not completed), [Intel Hyper-Threading Technology](#), [IBM POWER5](#), [Sun Microsystems UltraSPARC T2](#), [Cray XMT](#), and [AMD Bulldozer](#) and [Zen](#) microarchitectures.

***Superscalar*** = multi-issue → multi-EU (pipeline + ALU)



MT



“**Hyper-threading** (officially called **Hyper-Threading Technology** or **HT Technology** and abbreviated as **HTT** or **HT**) is [Intel's proprietary simultaneous multithreading](#) (SMT) implementation used to improve [parallelization of computations](#) (doing multiple tasks at once) performed on [x86](#) microprocessors. It was introduced on [Xeon server processors](#) in February 2002 and on [Pentium 4](#) desktop processors in November 2002.

Since then, Intel has included this technology in [Itanium](#), [Atom](#), and [Core 'i' Series](#) CPUs, among others.

For each [processor core](#) that is physically present, the [operating system](#) addresses two virtual (logical) cores and shares the workload between them when possible. The main function of hyper-threading is to increase the number of independent instructions in the pipeline; it takes advantage of [superscalar](#) architecture, in which multiple instructions operate on separate data [in parallel](#).”

yes, AMD has their own version of “SMT” which is essentially “superscalar” with hardware MT. but Wikipedia says Intel’s version *shares* the Execution Unit, while true superscalar has duplicate EU’s.

# MT Research

Wiki

**Thread Level Speculation (TLS)** is a technique to speculatively execute a section of computer code that is anticipated to be executed later in parallel with the normal execution on a separate independent thread. Such a speculative thread may need to make assumptions about the values of input

TLS

## Implementation specifics [\[ edit \]](#)

A major area of research is the thread scheduler that must quickly choose from among the list of ready-to-run threads to execute next, as well as maintain the ready-to-run and stalled thread lists. An important subtopic is the different thread priority schemes that can be used by the scheduler. The thread scheduler might be implemented totally in software, totally in hardware, or as a hardware/software combination.

Another area of research is what type of events should cause a thread switch: cache misses, inter-thread communication, [DMA](#) completion, etc.

If the multithreading scheme replicates all of the software-visible state, including privileged control registers and TLBs, then it enables [virtual machines](#) to be created for each thread. This allows each thread to run its own operating system on the same processor. On the other hand, if only user-mode state is saved, then less hardware is required, which would allow more threads to be active at one time for the same die area or cost.

Thread Scheduler

# ILP/TLP: MT/SMT

## Instructions vs. Threads

## Taxonomy [\[ edit \]](#)

In processor design, there are two ways to increase on-chip parallelism with fewer resource requirements: one is superscalar technique which tries to exploit instruction level parallelism (ILP); the other is multithreading approach exploiting thread level parallelism (TLP).

Superscalar means executing multiple instructions at the same time while thread-level parallelism (TLP) executes instructions from multiple threads within one processor chip at the same time. There are many ways to support more than one thread within a chip, namely:

- Interleaved multithreading: Interleaved issue of multiple instructions from different threads, also referred to as [temporal multithreading](#). It can be further divided into fine-grained multithreading or coarse-grained multithreading depending on the frequency of interleaved issues. **Fine-grained** multithreading—such as in a [barrel processor](#)—issues instructions for different threads after every cycle, while **coarse-grained** multithreading only switches to issue instructions from another thread when the current executing thread causes some long latency events (like page fault etc.). Coarse-grain multithreading is more common for less context switch between threads. For example, Intel's [Montecito](#) processor uses coarse-grained multithreading, while Sun's [UltraSPARC T1](#) uses fine-grained multithreading. For those processors that have only one pipeline per core, interleaved multithreading is the only possible way, because it can issue at most one instruction per cycle.
- Simultaneous multithreading (SMT): Issue multiple instructions from multiple threads in one cycle. The processor must be superscalar to do so.
- Chip-level multiprocessing (CMP or [multicore](#)): integrates two or more processors into one chip, each executing threads independently.
- Any combination of multithreaded/SMT/CMP.

The key factor to distinguish them is to look at how many instructions the processor can issue in one cycle and how many threads from which the instructions come. For example, Sun Microsystems' UltraSPARC T1 is a multicore processor combined with fine-grain multithreading technique instead of simultaneous multithreading because each core can only issue one instruction at a time.

***Superscalar*** = multi-issue → multi-EU (upper pipeline)



# MT vs. Multi-core



**Jeff Drobman** · Just now

1st we have to rule out "temporal" MT, since there is little benefit going beyond 2 threads there. then for SMT, since we use multiple EU's, we have to closely examine the benefit of shared L1 caches for SMT vs separate dedicated L1/L2 caches (with shared L3) for multi-core. I conjecture that 2 single-thread cores would outperform an SMT2 single core; i.e., cores would outperform threads due to the cache and register dedication vs shared.

# MT in Programs



**Avinash K S**, works at Macquarie Group

Updated

## What are common mistakes made in multithreaded programming?

Creating unnecessary dependencies (also read shared resources) among threads - Sharing data unnecessarily between threads and using locks to access the data is detrimental to the performance of a multi-threaded program. Hence using the MPICH approach for multi-threaded programming is beneficial, wherever possible. The idea is to divide your input data into non overlapping sets, and give each set to a thread. At the end, your main thread will be responsible to collect the sub-results from the worker threads, and combine them. This approach is also used in the implementation of parallel versions of STL algorithms.

Excessive locking - Lock least amount of shared data and instructions as possible, and immediately unlock after the operation is completed on the shared\_data.

# MT in Programs



**Avinash K S**, works at Macquarie Group

Updated Aug 30, 2018 · Invited by Alan Mello

I/O operations after acquiring lock - I/O operations are terribly slow compared to CPU operations. Same goes with networking operations. Hence you should avoid performing I/O or network operations once you acquire lock.

Creating too many threads - `std::thread::hardware_concurrency()` in C++ gives you a good idea about the maximum threads you would want to create in your C++ program. If you create more threads than this, the system will become more busy in context switches than executing your program.

Not using `std::async` - `async` helps to think about your program more like a set of tasks, which is much easier to reason than a bunch of threads. `Async` is non blocking, also frees you from the worry of creating excessive number of threads.

Not using atomics - atomics are lock free (only for few primitive data types, basic user defined data structures) and hence faster than using locks.

# MT Comparison

## ❖ Temporal MT

### ❑ Coarse

- +Can assign **priorities**
- +*Less* cache thrashing
- -*Slower* context switch

### ❑ Fine

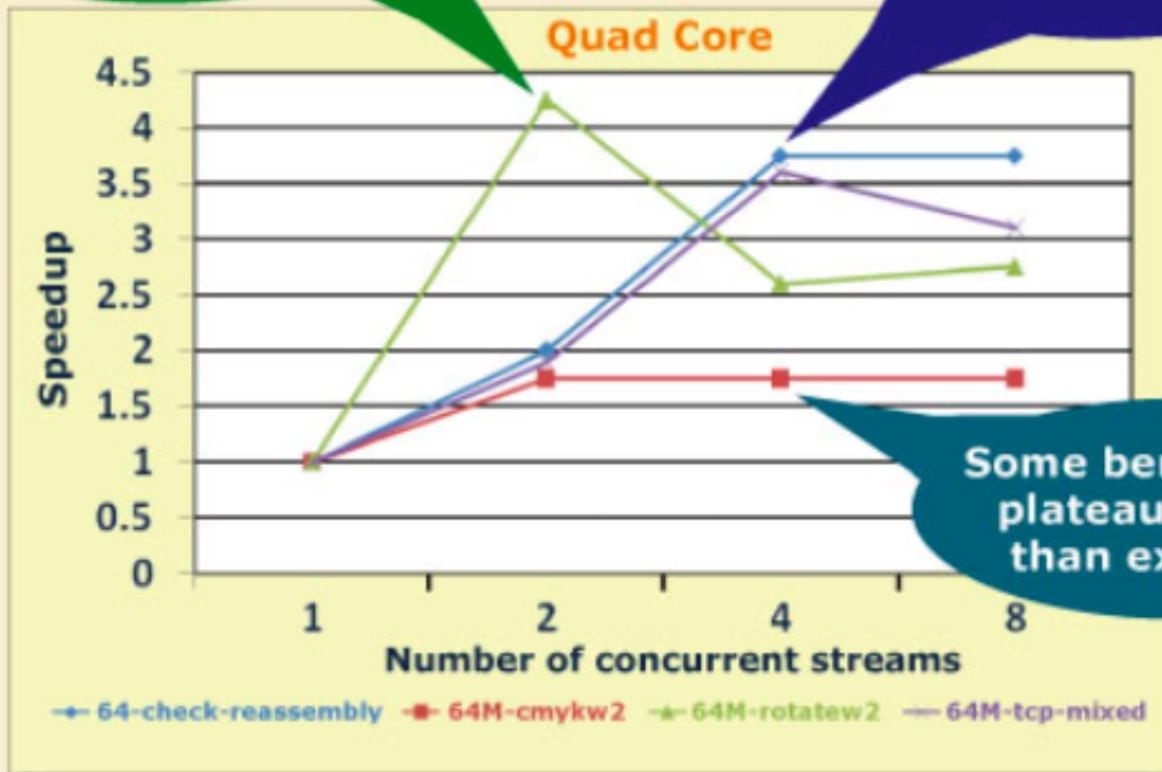
- +*Deterministic* **timing** (for real-time)
- -*More* cache thrashing (need Set Assoc)
- +*Faster* context switch
- -Wants **GR** partition

## ❖ SMT/HT

- +*Deterministic* **timing** (for real-time)
- -*Same as Fine* cache thrashing (need Set Assoc)
- +*NO* context switch
- +Highest **performance**
- -Needs *Superscalar*

Huge drop in performance when oversubscribed

Nice scaling on networking-only workloads



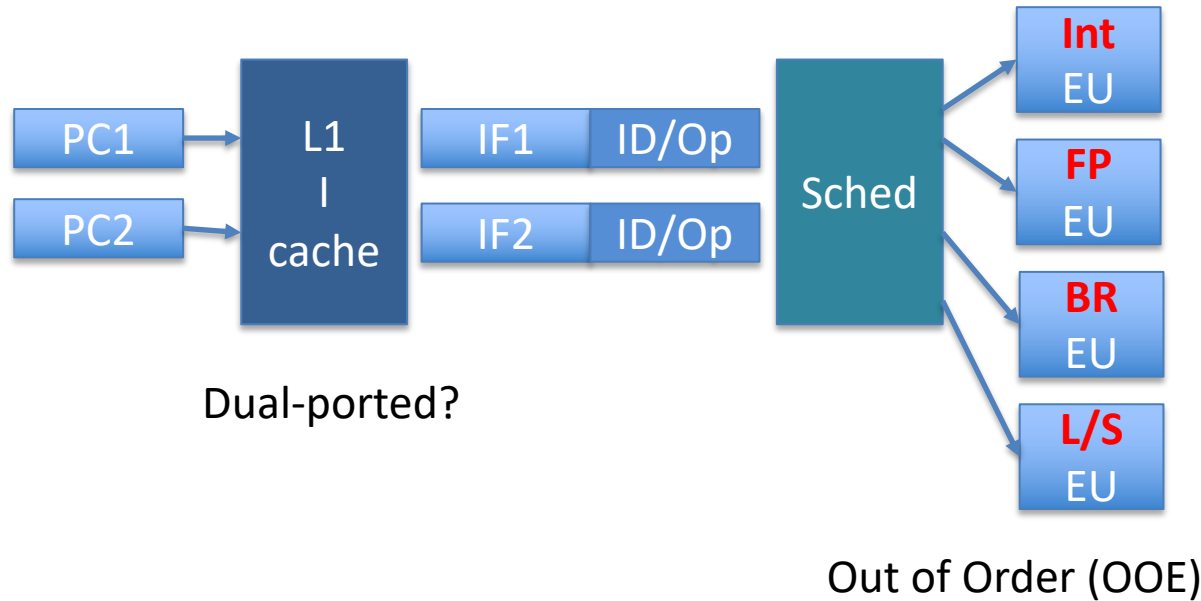
Some benchmarks plateau earlier than expected

*MultiBench demonstrates a wide variety of performance characteristics.*

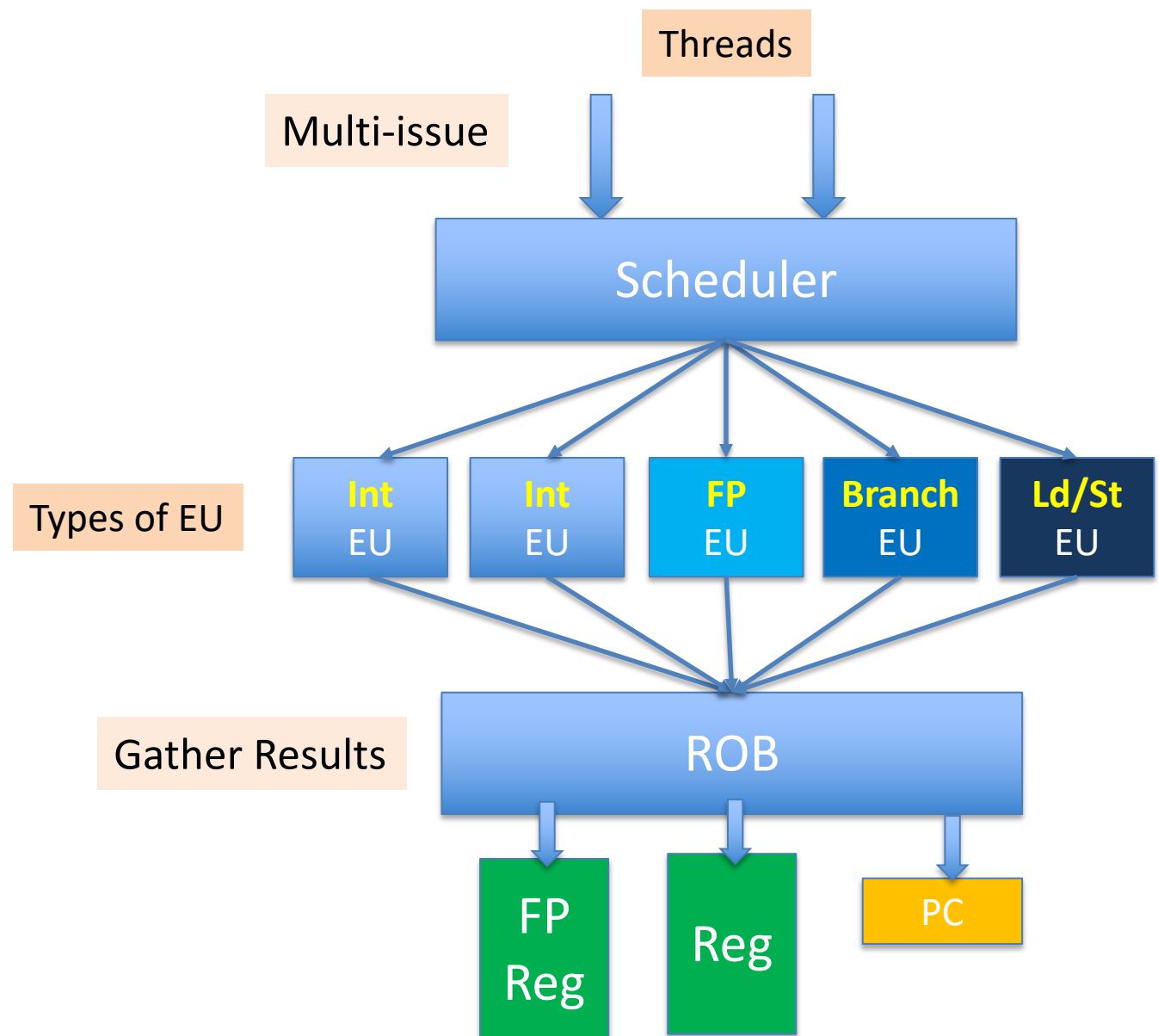
# Section

# ILP Superscalar

# SMT Upper Pipeline



# SMT/Superscalar

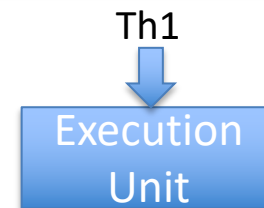




# Instruction Level *Parallelism*

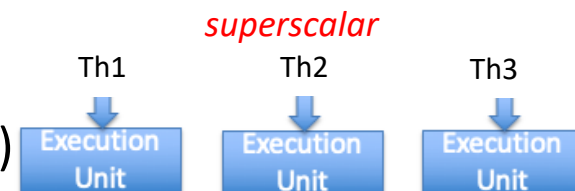
## ❖ Super- *Pipelining* SISD

- ❑ Split some pipeline stages (4-5 → 8-11)
- ❑ Faster clock cycle → higher *throughput* (mips)
- ❑ Affect CPI?



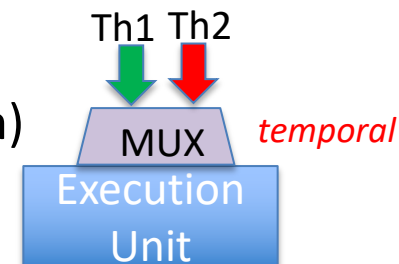
## ❖ Super- *Scalar* SISD

- ❑ Multiple **Execution Units** (multi-issue pipelines)
  - each EU = ICU+ALU, with shared GR's
- ❑ Hardware + compiler *schedules* instruction streams



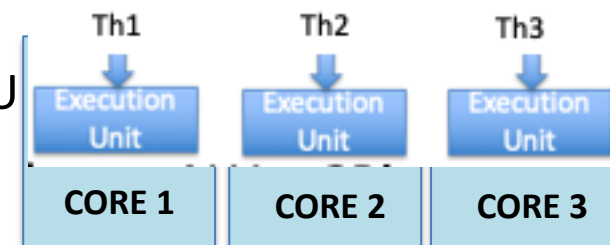
## ❖ Multi- *Threading* SISD

- ❑ Multiple control threads (usually 2, same/dif program)
- ❑ Programs can *allocate* code to threads
- ❑ Automatic *scheduling* of control threads
- ❑ 2 types: *SMT/superscalar* or *temporal* (interleaved: coarse/fine)



## ❖ Multi- *Core* MISD

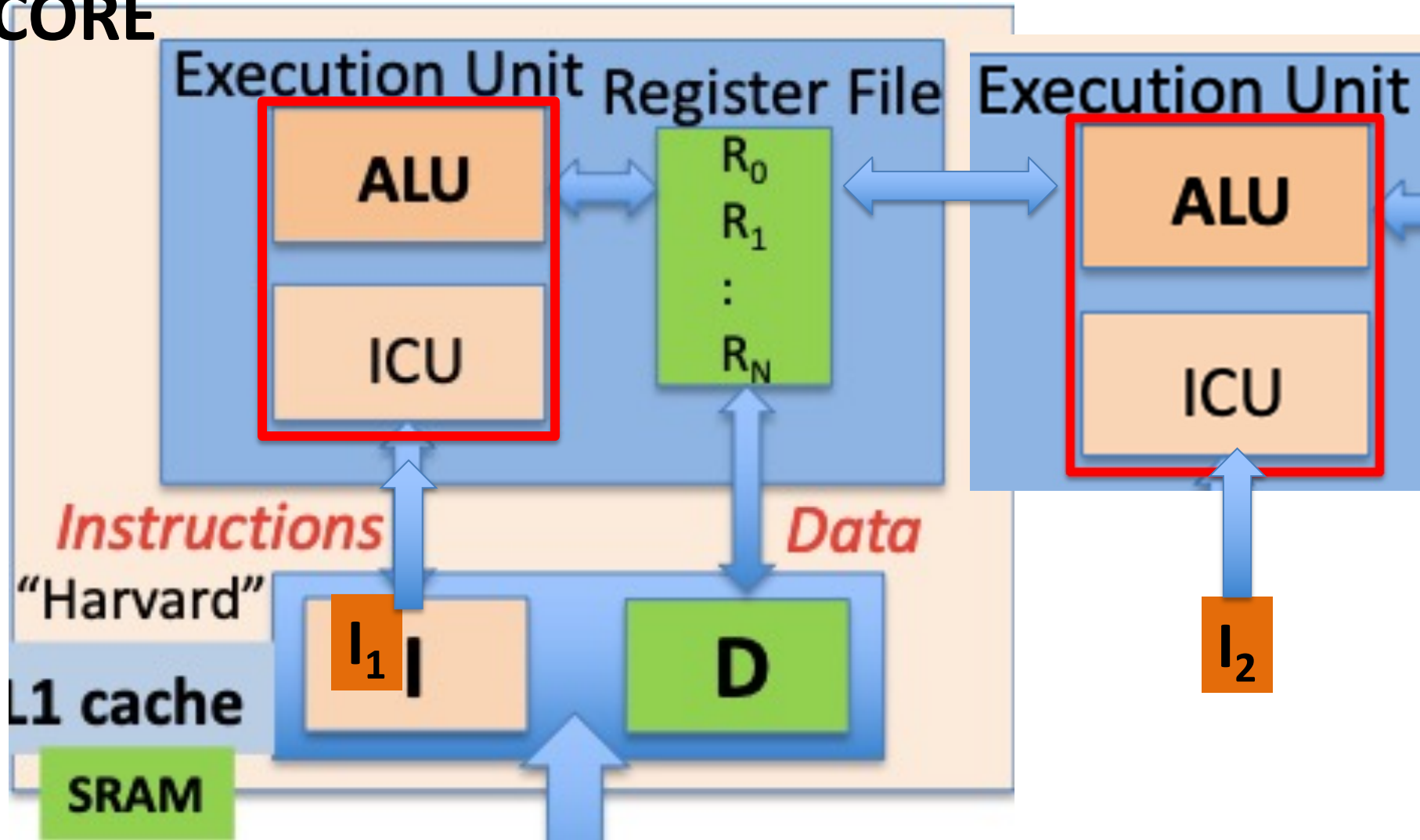
- ❑ Classic Parallelism: multiple copies of the CPU
- ❑ **Multiple L1/L2 caches** (one set per core)



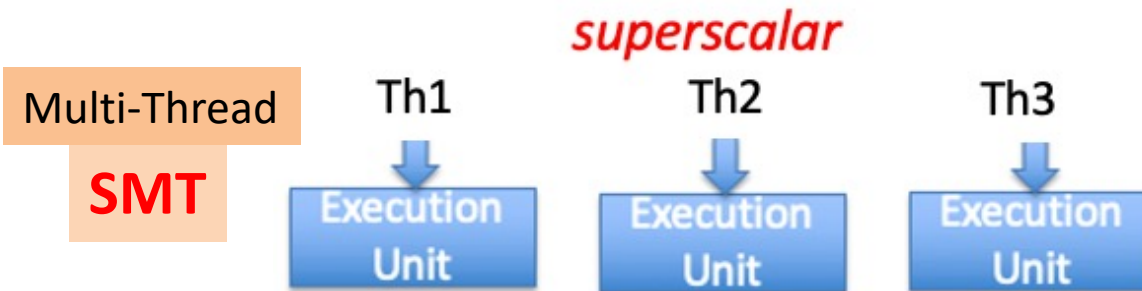
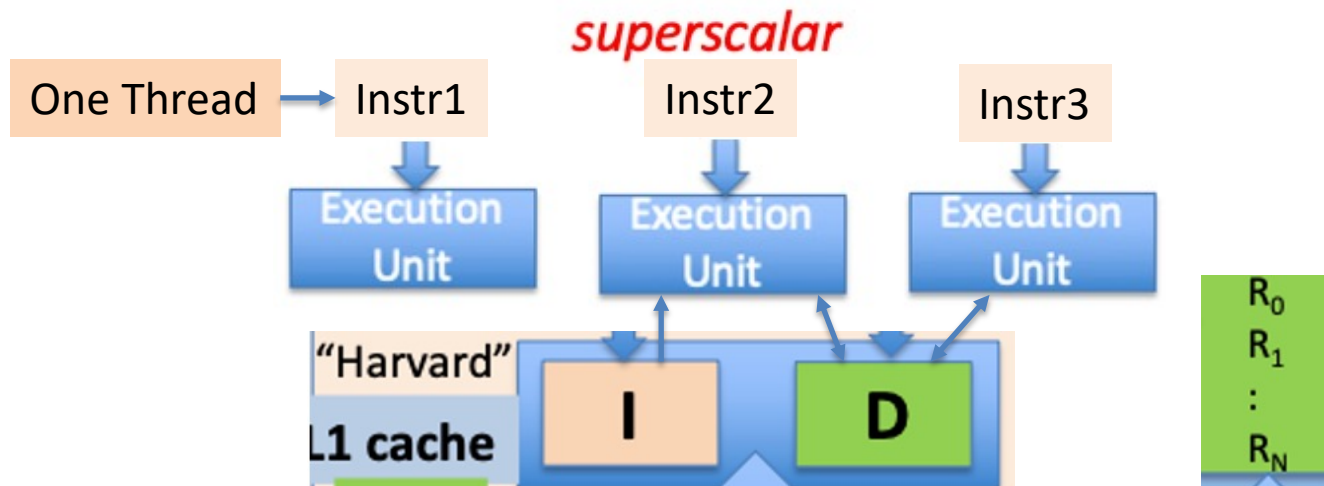
# Superscalar

2 Issue

**CORE**



# Superscalar



# Superscalar

## What is the difference between a superscalar CPU design and a super pipelined CPU design?



**Jeff Drobman**, Lecturer at California State University, Northridge (2016-present)

Answered just now

superscalar has >1 EU with multi-issue/ multiple pipelines. super-pipeline is a deep pipeline with >5 stages, usually 8 or more. MIPS R4000 was the first to use super-pipelining in 1992 with 8 stages.

a simple answer: superscalar splits EU's with their pipelines for ILP, while super-pipelining splits its stages for faster clock cycles.

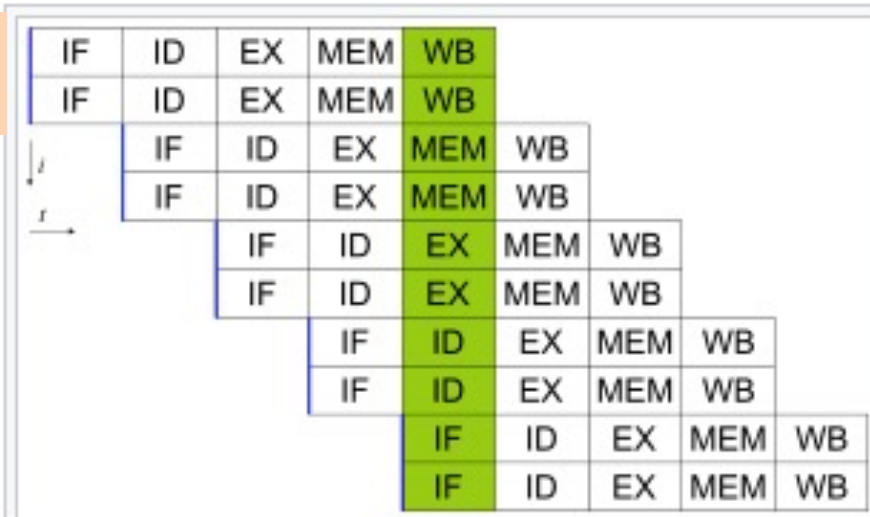
# Superscalar

## ❖ Super-Scalar SISD

- ❑ Multiple **Execution Units** (multi-issue pipelines)
  - each EU = ICU+ALU, with shared GR's
- ❑ Hardware + compiler schedules instruction streams



I1/Th1 EU1 pipeline1  
I2/Th2 EU2 pipeline2



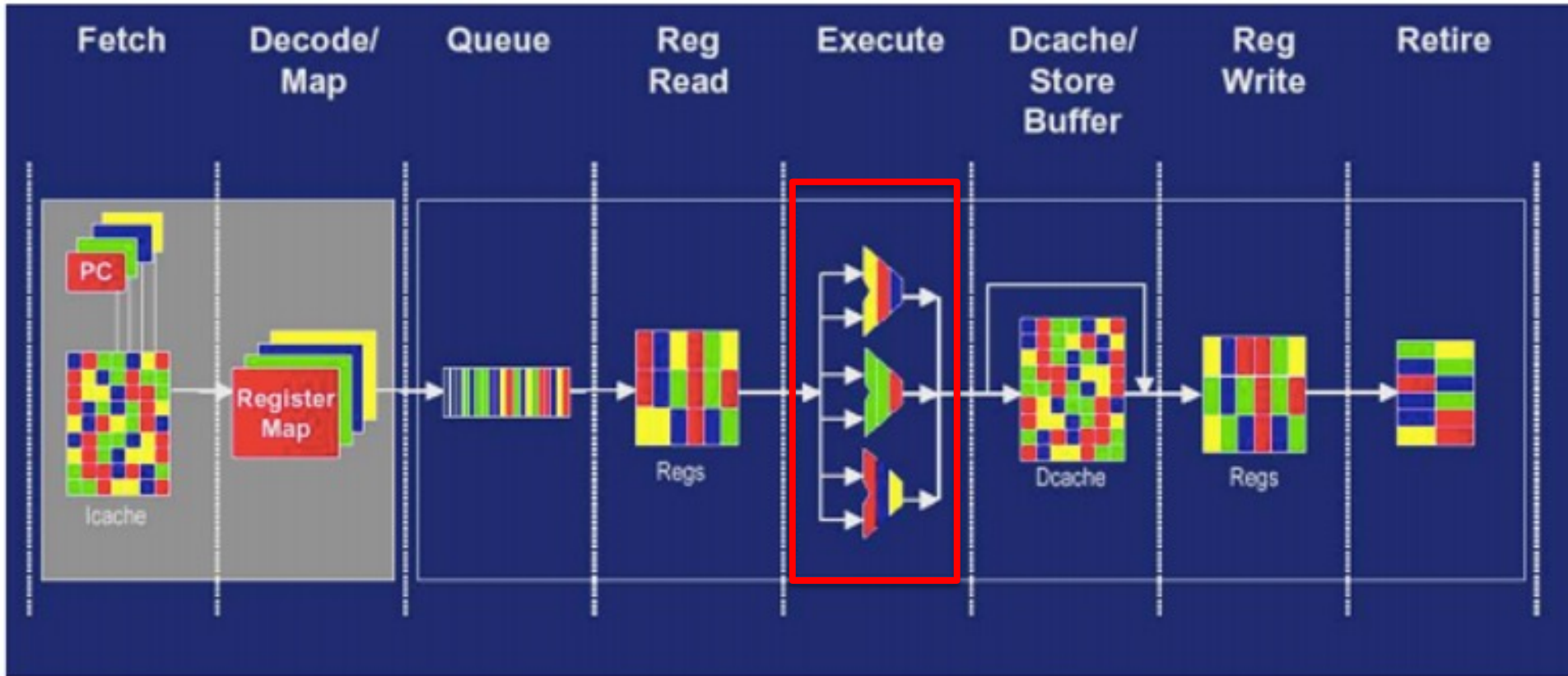
Simple superscalar pipeline. By fetching and dispatching two instructions at a time, a maximum of two instructions per cycle can be completed. (IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back,  $i$  = Instruction number,  $t$  = Clock cycle [i.e., time])



# Threads in a Pipeline

4 Threads 8 Stages

Mikko Lipasti-University of Wisconsin



# Superscalar on a Stack M

**Quora**

## Is it possible to build a superscalar stack-based CPU? E.g. a superscalar FORTH processor?



**Jeff Drobman**, Lecturer at California State University, Northridge (2016-present)

Answered just now

"superscalar" means having multi-issue, multi-EU hardware in the CPU. there would be some challenges to managing the stack, e.g. if 2 or more instructions wanted to access the stack at the same time (same stage), there would have to be priority-based interlocks (either hardware or software).

➤ Or separate Stack Frames

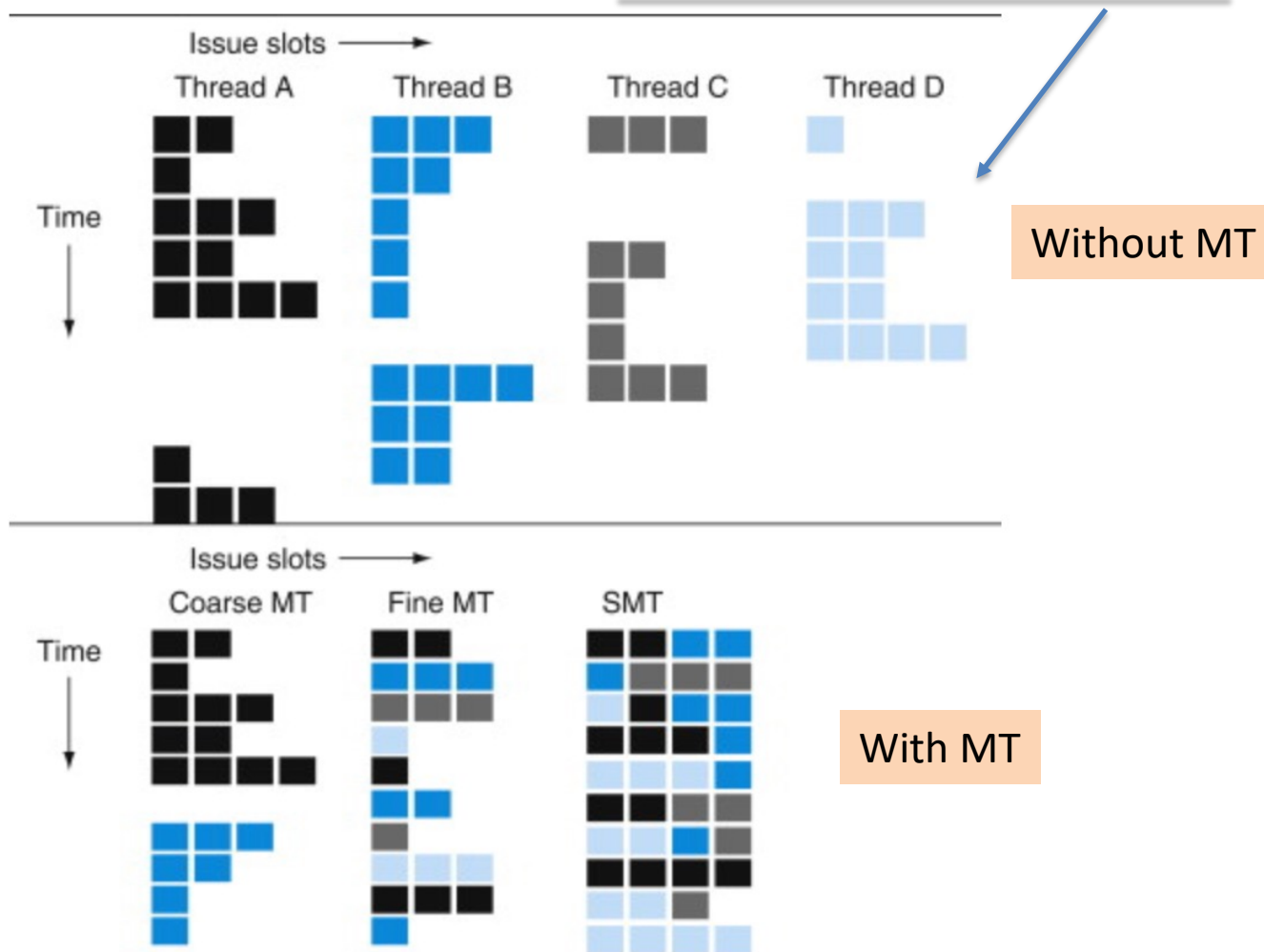
Stack  
Frame 1

Stack  
Frame 2

# SMT + Superscalar

P&H zyBook

CON how four threads would execute independently on a superscalar with no multithreading support.



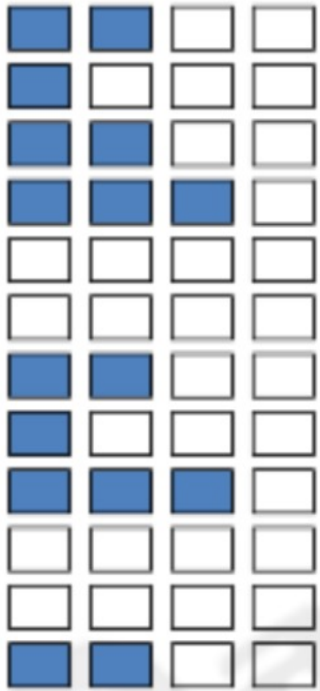
**Superscalar** = multi-issue → multi-pipeline + multiple ALU



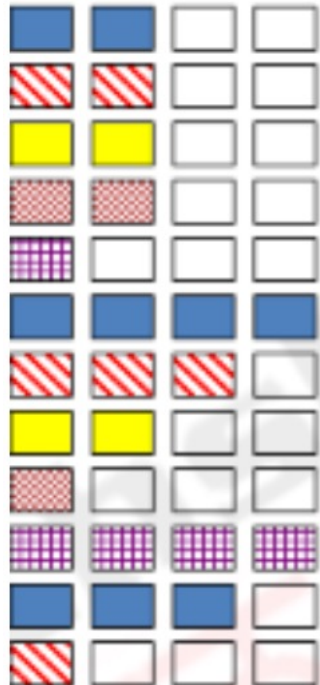
# MT Graphics

realworldtech.com

Superscalar



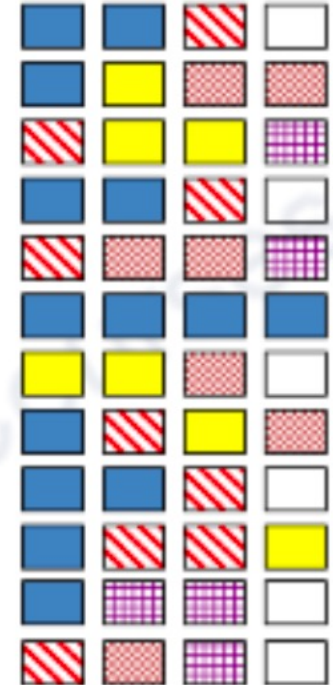
Fine-Grained



Coarse-Grained



Simultaneous  
Multithreading

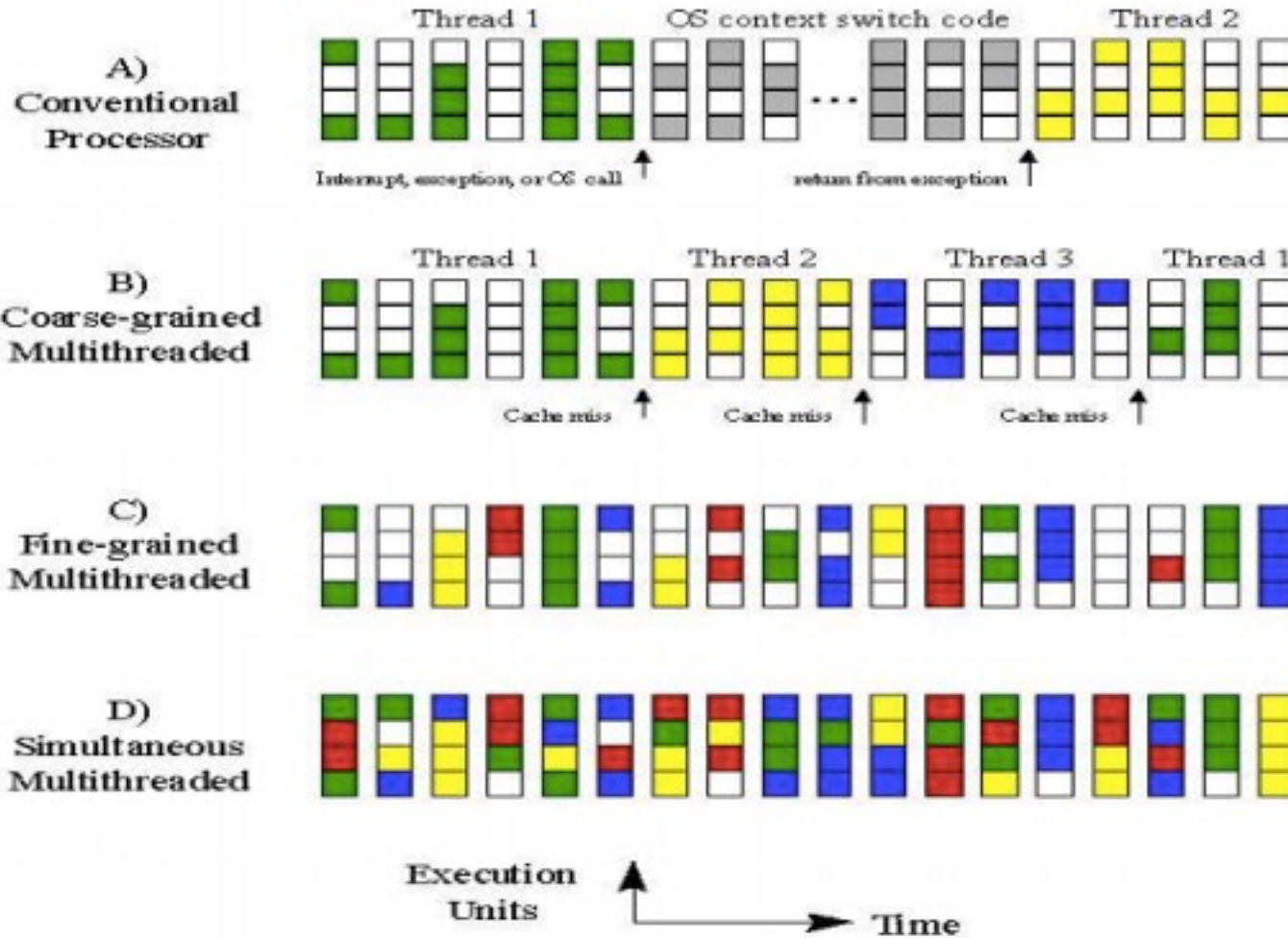


Thread 1  
Thread 2

Thread 3  
Thread 4

Thread 5  
Idle slot

# MT Graphics



# Section

---

# Intel Hyper-Threading

# Hyperthreading

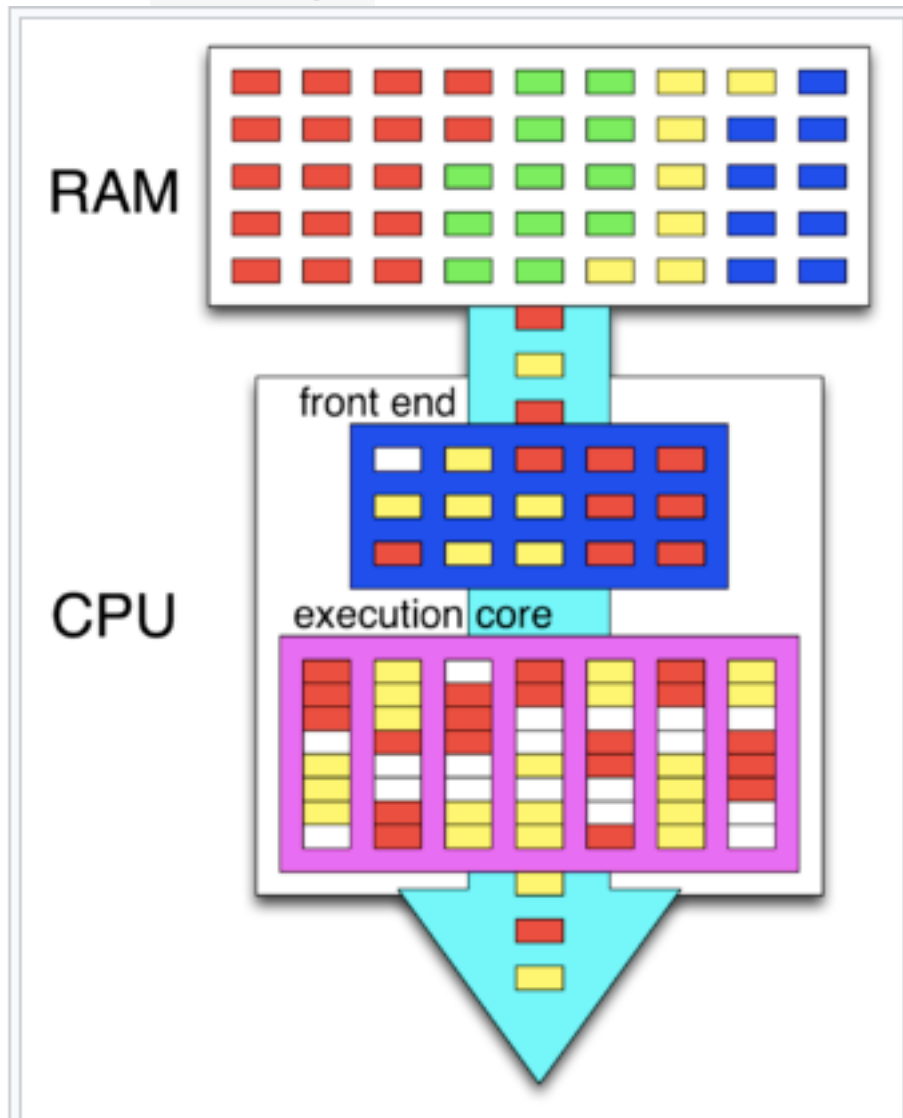
**What is hyperthreading?** Hyperthreading is a feature that allows each CPU core to emulate two cores at once, or threads. On some Xeon Phi processors, Intel supports four-way hyperthreading, effectively quadrupling the number of threads. Prior to the Coffee Lake architecture, most Xeon and all desktop and mobile Core i3 and i7 supported hyperthreading while only dual core mobile i5's supported it. Post Coffee Lake, increased core counts meant hyperthreading is not needed for Core i3, as it now replaced the i5 with four physical cores on the desktop platform. Core i7, on the desktop platform, no longer supports hyperthreading; instead now higher performing core i9s will support hyperthreading on both mobile and desktop platforms. Prior to 2007 and post Kaby Lake some Intel Pentiums support hyperthreading. Celerons and Atom processors have never supported it.

**Hyper-threading** (officially called **Hyper-Threading Technology** or **HT Technology** and abbreviated as **HTT** or **HT**) is [Intel's proprietary simultaneous multithreading](#) (SMT) implementation used to improve [parallelization](#) of computations (doing multiple tasks at once) performed on [x86](#) microprocessors. It first appeared in February 2002 on [Xeon](#) server [processors](#) and in November 2002 on [Pentium 4](#) desktop CPUs.<sup>[4]</sup> Later, Intel included this technology in [Itanium](#), [Atom](#), and [Core 'i' Series](#) CPUs, among others.

For each [processor core](#) that is physically present, the [operating system](#) addresses two virtual (logical) cores and shares the workload between them when possible. The main function of hyper-threading is to increase the number of independent instructions in the pipeline; it takes advantage of [superscalar](#) architecture, in which multiple instructions operate on separate data [in parallel](#). With HTT, one physical core appears as two processors to the operating system, allowing [concurrent](#) scheduling of two processes per core. In addition, two or more processes can use the same resources: If resources for one process are not available, then another process can continue if its resources are available.

In addition to requiring simultaneous multithreading (SMT) support in the operating system, hyper-threading can be properly utilized only with an operating system specifically optimized for it.<sup>[5]</sup> Furthermore, Intel recommends HTT to be disabled when using operating systems unaware of this hardware feature.<sup>[citation needed]</sup>

# Hyperthreading



Virtual **Thread** Machine

Sits on top of Superscalar Multi-cores

In this high-level depiction of HTT, instructions are fetched from RAM (differently colored boxes represent the instructions of four different **processes**), decoded and reordered by the front end (white boxes represent **pipeline bubbles**), and passed to the execution core capable of executing instructions from two different programs during the same **clock cycle**.<sup>[1][2][3]</sup>

Intel's proprietary **HTT**



# Intel Hyperthreading

Mikko Lipasti-University of Wisconsin

## Intel Hyperthreading

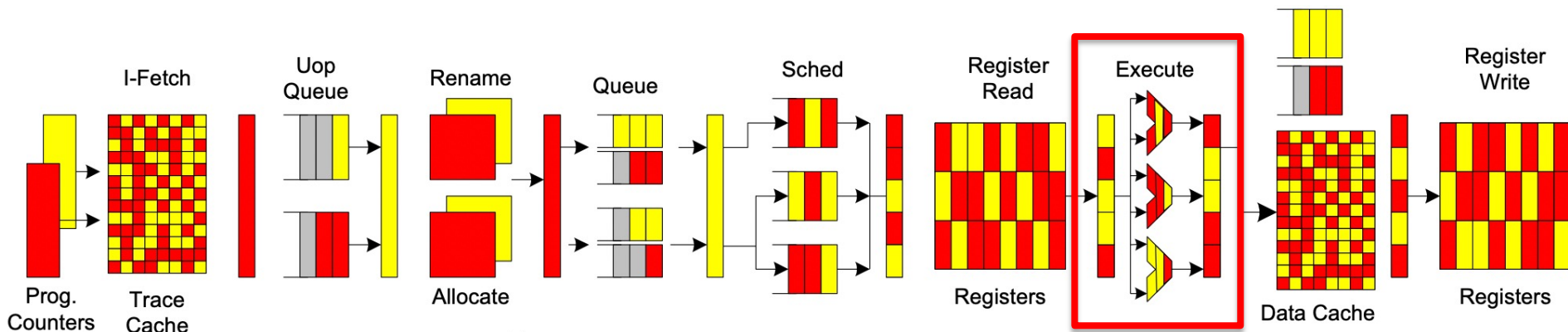
- Part of Pentium 4 design (Xeon)
- Two threads per processor
- Goals
  - Low cost – less than 5% overhead for replicated state
  - Assure forward progress of both threads
    - Make sure both threads get some buffer resources
    - through partitioning or budgeting
  - Single thread running alone does not suffer slowdown

# Intel Hyperthreading

Mikko Lipasti-University of Wisconsin

## Intel Hyperthreading

- Main pipeline
  - Pipeline prior to trace cache not shown
- Round-Robin instruction fetching
  - Alternates between threads
  - Avoids dual-ported trace cache
  - BUT trace cache is a shared resource

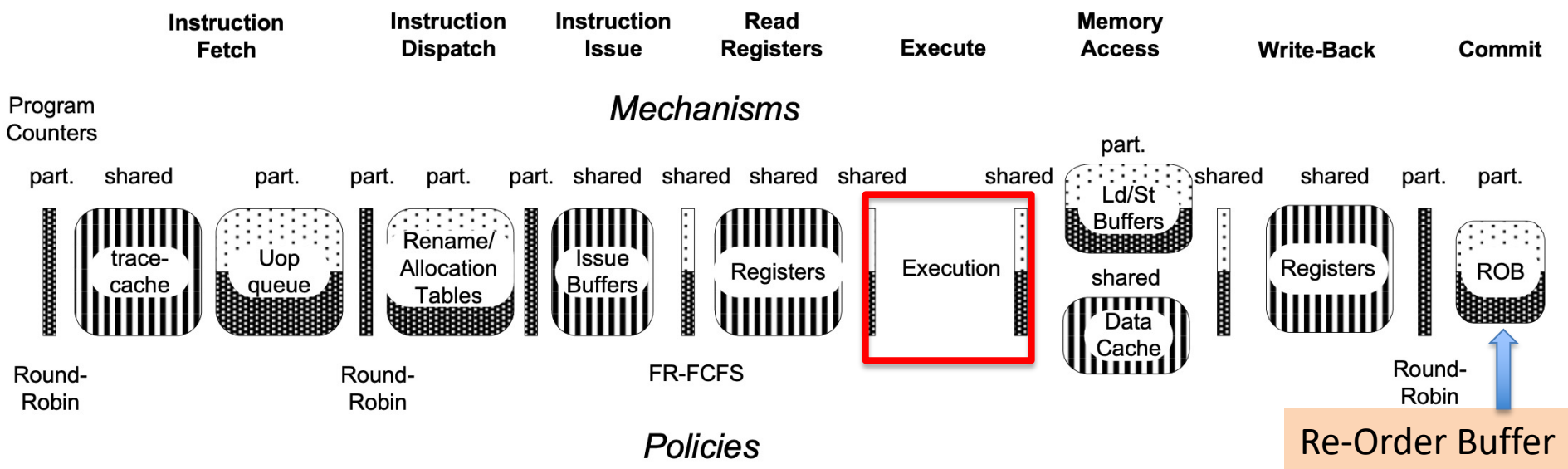


# HT Resources & Policies

Mikko Lipasti-University of Wisconsin

## Example: Hyperthreading

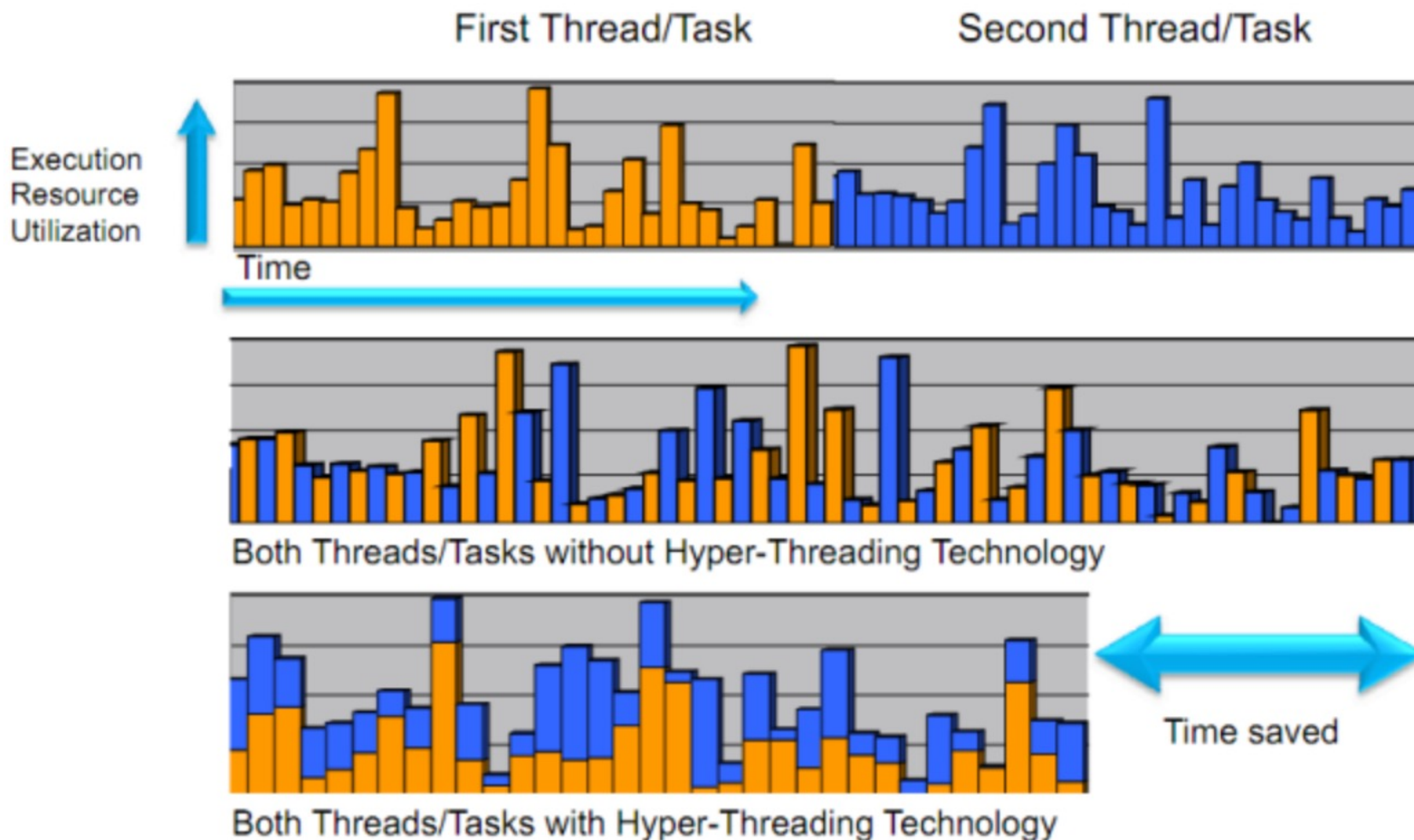
- Mechanisms (and Policies) in Pentium 4





# Intel HT Slide

## Benefits of HT Technology



An old slide from Intel, which has its own marketing term for SMT: Hyper-Threading

# MT on Phones

## Would smartphones ever have multithreaded cores? Why? (please see comment for more info)



**Jeff Drobman**, Lecturer at California State University, Northridge (2016-present)

Answered just now

Coarse grained Temporal MT has low silicon overhead but can produce good results. It doesn't require a superscalar architecture, so is less expensive than the SMT that Intel and AMD use. I would use a 2-thread version on all cores, or at least on several cores. Apple now has 10 CPU cores on its M1, and I bet at least some of them use MT.

# MT on Phones



**David Kra**, Received patents in ecommerce and microprocessor design. Published author.



Answered 25m ago

Yes, they would, if the usage pattern changes. Multithreaded cores are appropriate when the quantity of actively runnable application processes and threads gets large and their performance (instructions per second) needs are low.

Remember:

- CPU Multithreading only starts to help once every core is already busy on one thread.
- A 2-thread multithreaded core typically provides **both a 20% boost** in overall **throughput**, and a **40% decrease** in *per thread performance*.
- The core best achieves this boost for multithreaded applications where the threads are executing the same small set of code, but applying them to different pieces of data. Otherwise, the benefit is less. Good usage: Rendering a web page with multiple images, with each thread rendering a different image.

# Section

---

## Performance CPI vs IPC

# Peak Perf of SMT

$$IPC = N * (1/CPI)$$

TYP  $CPI = 1.3 \rightarrow 1/1.3 = 0.77$

Examples for N at 1 GHz

$$N = 1 \rightarrow IPC = 0.77 \rightarrow MIPS = 770$$

$$N = 2 \rightarrow IPC = 1.54 \rightarrow MIPS = 1540$$

$$N = 4 \rightarrow IPC = 3.08 \rightarrow MIPS = 3080$$

# U Wisc Slides: Perf

$$\text{Processor Performance} = \frac{\text{Time}}{\text{Program}}$$

$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}$$

(code size)                      (CPI)                      (cycle time)

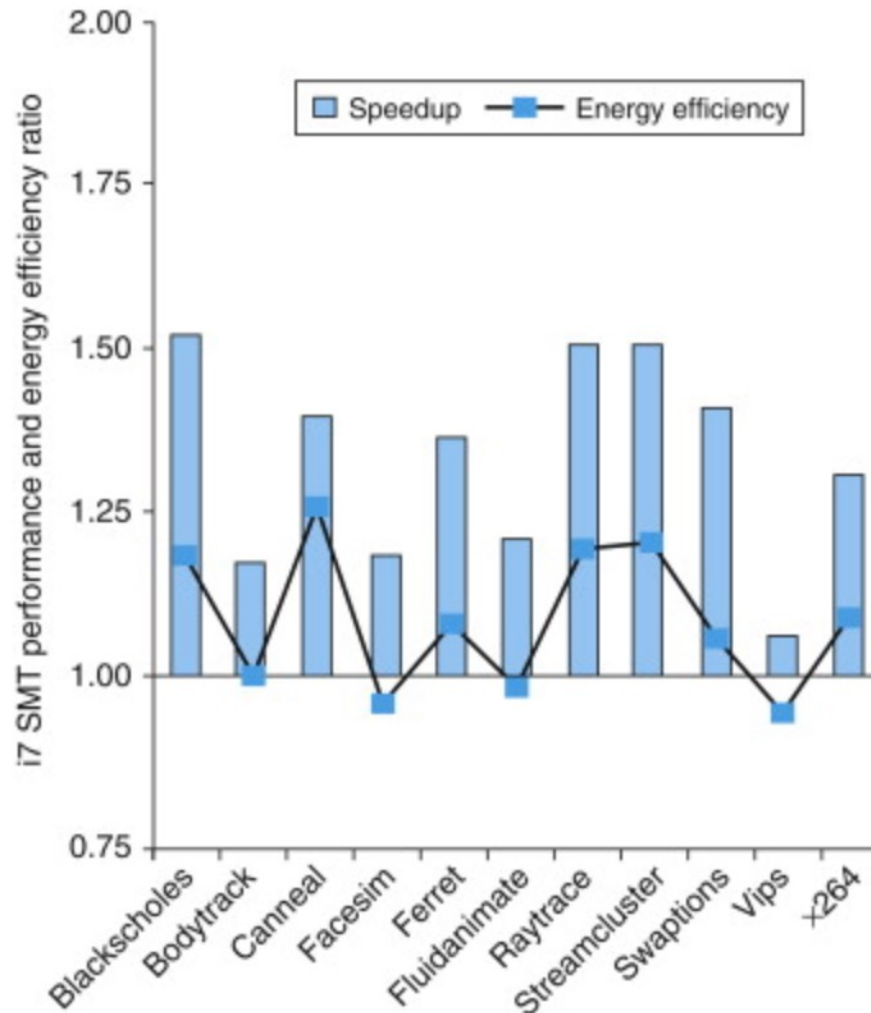
- In the 1980's (decade of pipelining):
  - CPI: 5.0 => 1.15 MIPS
- In the 1990's (decade of superscalar):
  - CPI: 1.15 => 0.5 (best case)
- In the 2000's (decade of multicore):
  - Core CPI unchanged; chip CPI scales with #cores

# SMT Performance

P&H zyBook

Figure 6.4.2: The speed-up from using multithreading on one core on an **i7** processor (COD Figure 6.6).

Processor averages 1.31 for the PARSEC benchmarks (see COD Section 6.9 (Communicating to the outside world: Cluster networking)) and the energy efficiency improvement is 1.0





# Section

**SMT ON vs OFF**

## MT Benchmarks

Investigating Performance of Multi-Threading on  
Zen 3 and AMD Ryzen 5000

by **Dr. Ian Cutress** on December 3, 2020 10:00 AM EST

<https://www.anandtech.com/show/16261/investigating-performance-of-multithreading-on-zen-3-and-amd-ryzen-5000/2>

“AMD Ryzen 9 3900X, SMT on vs SMT off, vs Intel 9900K”, TechPowerUp,  
<https://www.techpowerup.com/review/amd-ryzen-9-3900x-smt-off-vs-intel-9900k/>

<https://ece757.ece.wisc.edu/lect03-cores-multithread.pdf>



# SMT Product Examples

## Investigating Performance of Multi-Threading on Zen 3 and AMD Ryzen 5000

by [Dr. Ian Cutress](#) on December 3, 2020 10:00 AM EST

We can split up the systems that use SMT:

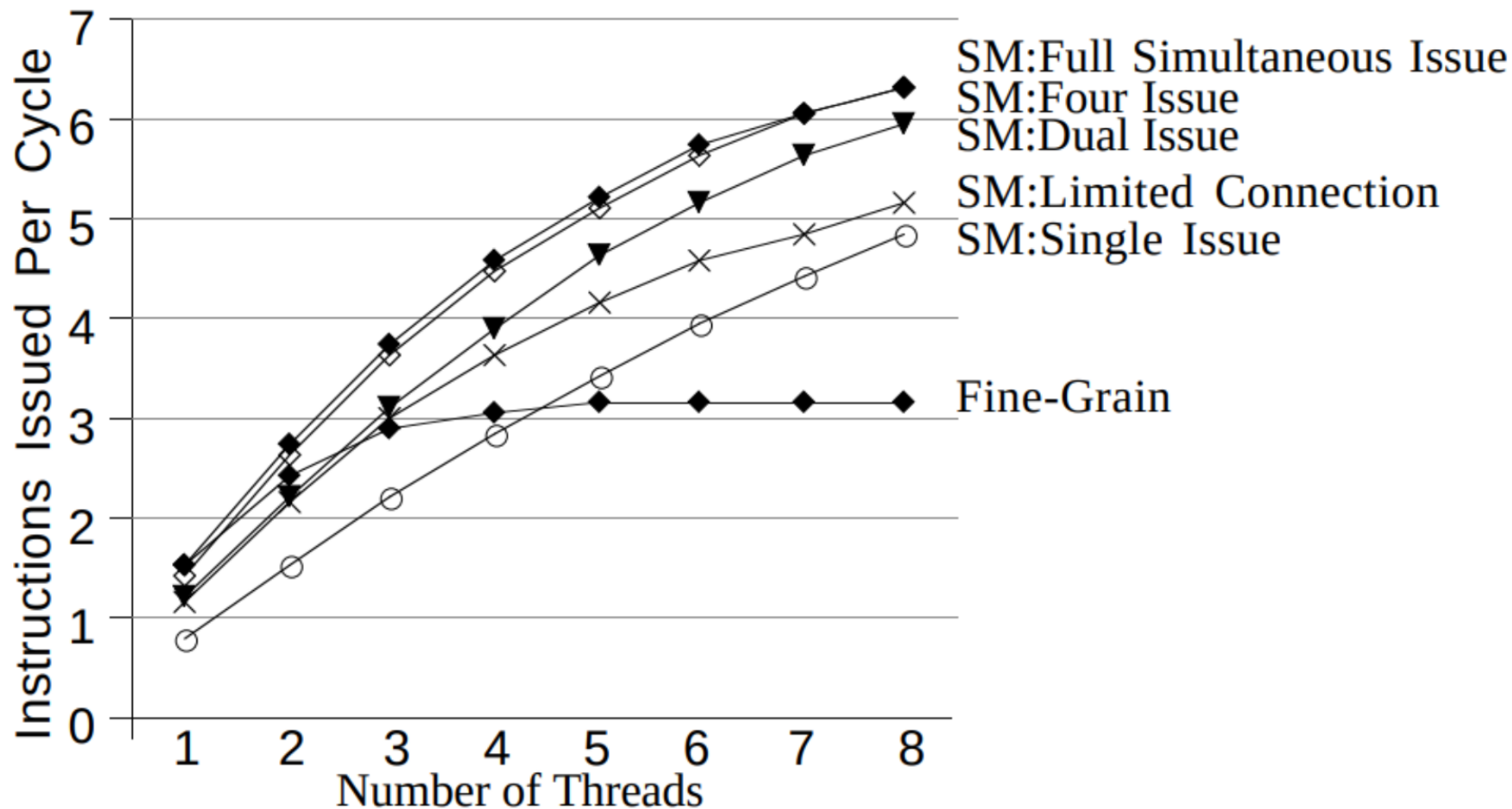
- *High-performance x86 from Intel*
- *High-performance x86 from AMD*
- *High-performance POWER/z from IBM*
- *Some High-Performance Arm-based designs*
- *High-Performance Compute-In-Memory Designs*
- *High-Performance AI Hardware*

Comparing to those that do not:

- *High-efficiency x86 from Intel*
- *All smartphone-class Arm processors*
- *Successful High-Performance Arm-based designs*
- *Highly focused HPC workloads on x86 with compute bottlenecks*

# SMT → IPC

COMP222



As the number of threads increases the number of instructions per cycle also increases improving overall throughput except when using Fine-grain because Fine-grain is limited to how much instruction can be executed at once.

# SMT in AMD Ryzen (Zen 3)

Simultaneous Multithreading

OFF ON

Multi-Threaded Tests  
AMD Ryzen 9 5950X

**SMT ON vs OFF**

AnandTech	SMT Off Baseline	SMT On
Agisoft Photoscan	100%	98.2%
3D Particle Movement	100%	165.7%
3DPM with AVX2	100%	177.5%
y-Cruncher	100%	94.5%
NAMD AVX2	100%	106.6%
AlBench	100%	88.2%
Blender	100%	125.1%
Corona	100%	145.5%
POV-Ray	100%	115.4%
V-Ray	100%	126.0%
CineBench R20	100%	118.6%
HandBrake 4K HEVC	100%	107.9%
7-Zip Combined	100%	133.9%
AES Crypto	100%	104.9%

# SMT Benchmarks

## Application Benchmarks

Simultaneous Multithreading

OFF ON

### Relative Performance CPU Tests

TECHPOWERUP  
Higher is Better

TECHPOWERUP

ENERMAX

Ready  
Intel

HOME REVIEWS FORUMS DOWNLOADS CASE MOD GALLERY DATABASES OUR SOFTWARE



### AMD Ryzen 9 3900X, SMT on vs SMT off, vs Intel 9900K

by W1zzard, on Jul 22nd, 2019, in Process

SMT ON vs OFF

Ryzen 3 1200 3.1/3.4 GHz: 38.8 %

Ryzen 3 2200G 3.5/3.7 GHz: 41.9 %

Ryzen 3 1300X 3.4/3.7 GHz: 43.5 %

Ryzen 5 1400 3.2/3.4 GHz: 47.2 %

Ryzen 5 2400G 3.6/3.9 GHz: 52.0 %

Core i3-8300 3.7 GHz: 52.1 %

Ryzen 5 1500X 3.5/3.7 GHz: 53.2 %

Core i3-9100F 3.6/4.2 GHz: 55.6 %

Core i3-8350K 4.0 GHz: 56.0 %

Ryzen 5 1600 3.2/3.6 GHz: 62.7 %

Core i5-8400 2.8/4.0 GHz: 66.3 %

Ryzen 5 1600X 3.6/4.0 GHz: 67.1 %

Core i5-9400F 2.9/4.1 GHz: 67.9 %

Ryzen 5 3600 3.6/4.2 GHz: 84.1 %

Core i7-8700K 3.7/4.7 GHz: 85.5 %

Ryzen 5 3600X 3.8/4.4 GHz: 85.6 %

Core i7-9700K 3.6/4.9 GHz: 85.7 %

Ryzen 7 3700X 3.6/4.4 GHz: 94.6 %

Core i9-9900K 3.6/5.0 GHz: 97.4 %

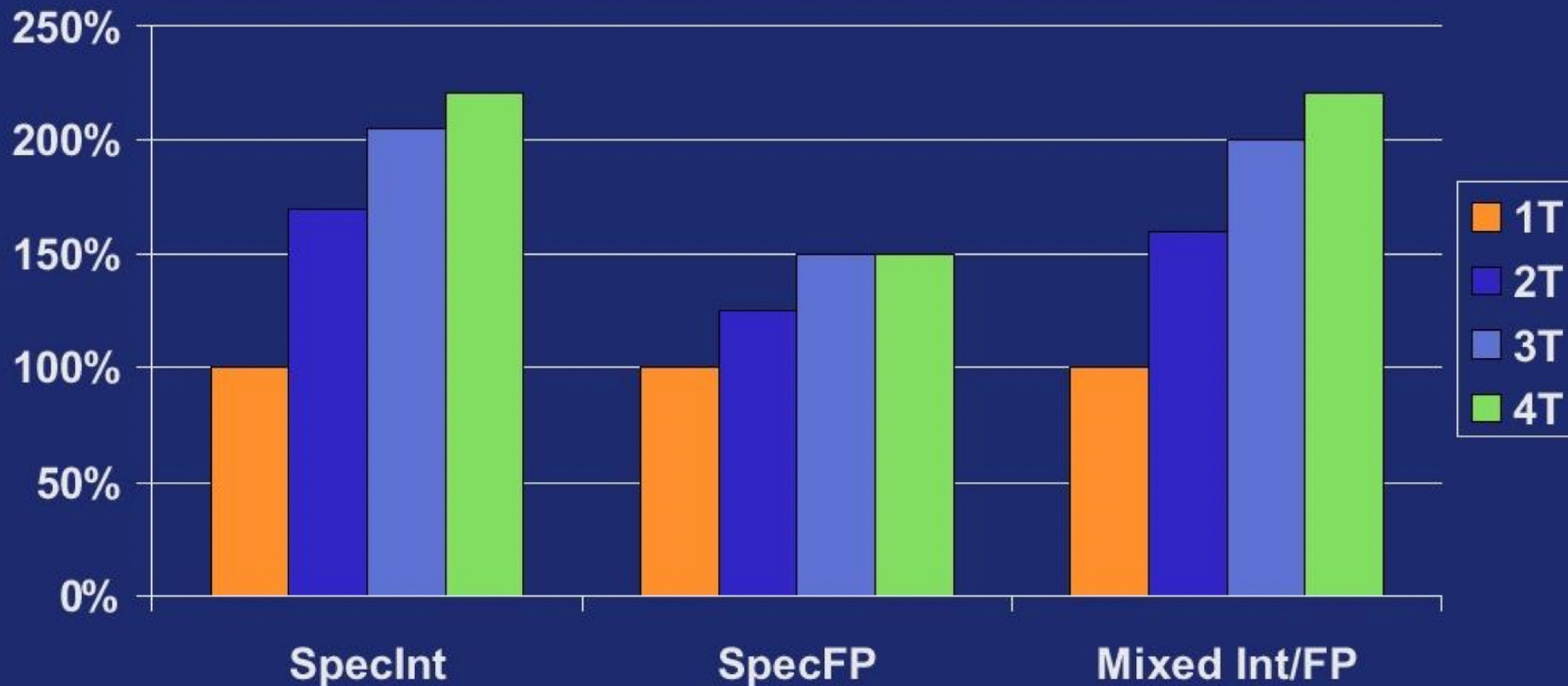
Ryzen 9 3900X SMT off: 100.0 %

Ryzen 9 3900X 3.8/4.6 GHz: 110.5 %

# U Wisc Slides: Benchmarks

Mikko Lipasti-University of Wisconsin

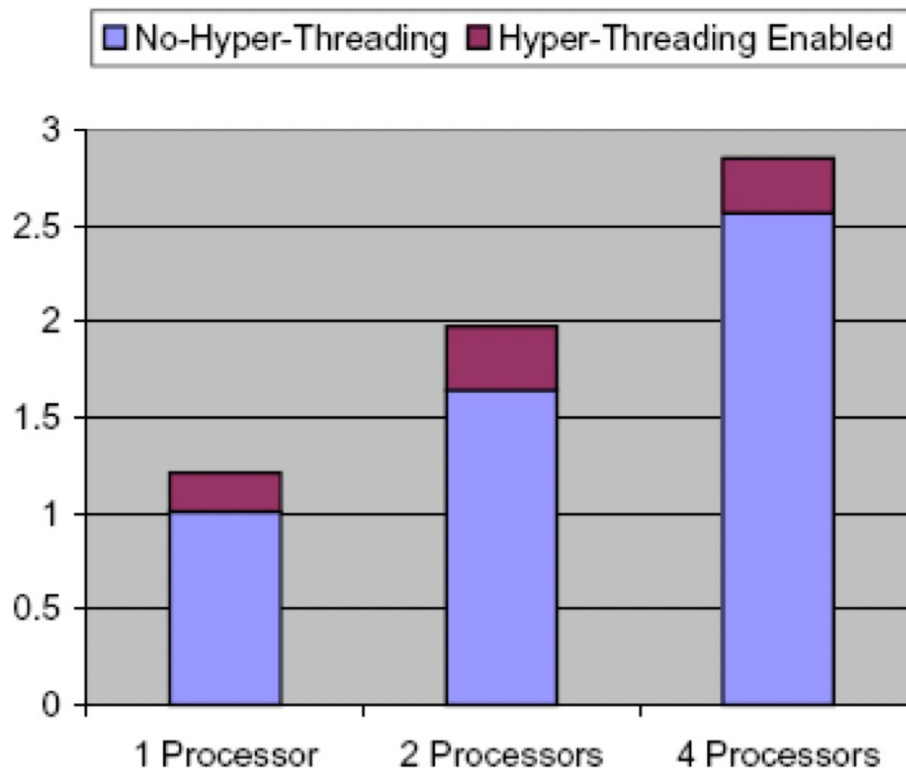
## Multiprogrammed workload



# U Wisc Slides: Benchmarks

© J.E. Smith

- OLTP workload
  - 21% gain in single and dual systems
  - Likely external bottleneck in 4 processor systems
    - Most likely front-side bus (FSB), i.e. memory bandwidth

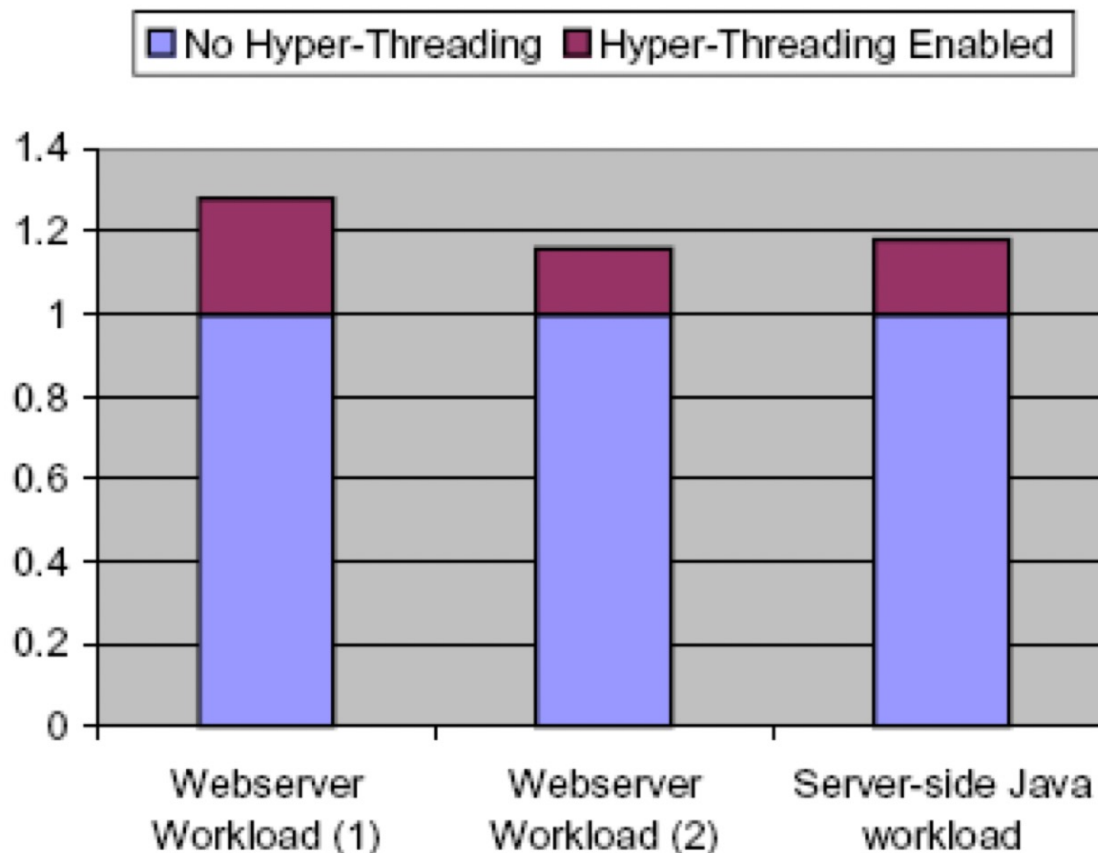




# U Wisc Slides: Benchmarks

© J.E. Smith

- Web server apps



# Section

---

## x86 CPU AMD vs Intel

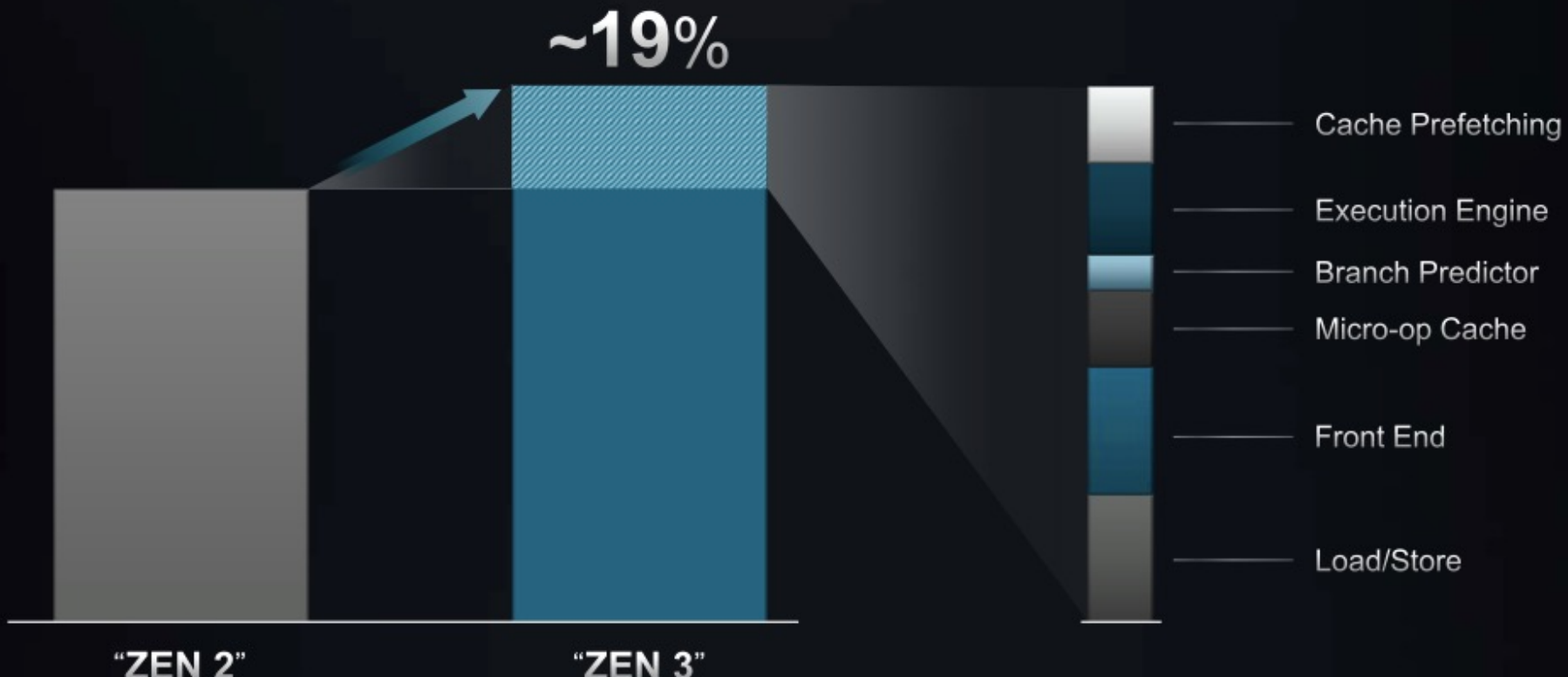


## INDUSTRY LEADERSHIP

“ZEN 3” ~19% IPC UPLIFT

GEOMEAN OF 28 WORKLOADS  
(FIXED 3.7GHZ FREQUENCY, 8 CORES)

“ZEN 3” PERFORMANCE  
CONTRIBUTORS



# Cache Set Assoc (Ways)

## AMD GPU

>14nm

2016-20

<b>L1 data cache per core (KiB)</b>	64	16		
<b>L1 data cache associativity (ways)</b>	2	4		8
<b>L1 instruction caches per core</b>	1	0.5		1
<b>Max APU total L1 instruction cache (KiB)</b>	256	128	192	256
<b>L1 instruction cache associativity (ways)</b>		2	3	4
<b>L2 caches per core</b>	1	0.5		1
<b>Max APU total L2 cache (MiB)</b>		4	2	4
<b>L2 cache associativity (ways)</b>		16		8
<b>APU total L3 cache (MiB)</b>		N/A		4
<b>APU L3 cache associativity (ways)</b>				16
<b>L3 cache scheme</b>	Victim	N/A		Victim

# Intel Itanium: IA-64

## IA-64

From Wikipedia, the free encyclopedia

*Not to be confused with [x86-64](#).*

**IA-64 (Intel Itanium architecture)** is the [instruction set architecture](#) (ISA) of the [Itanium](#) family of 64-bit [Intel microprocessors](#). The basic ISA specification originated at [Hewlett-Packard](#) (HP), and was evolved and then implemented in a new processor microarchitecture by Intel with HP's continued partnership and expertise on the underlying EPIC design concepts. In order to establish what was their first new ISA in 20 years and bring an entirely new product line to market, Intel made a massive investment in product definition, design, software development tools, OS, software industry partnerships, and marketing. To support this effort Intel created the largest design team in their history and a new marketing and industry enabling team completely separate from x86. The first Itanium processor, [codenamed Merced](#), was released in 2001.

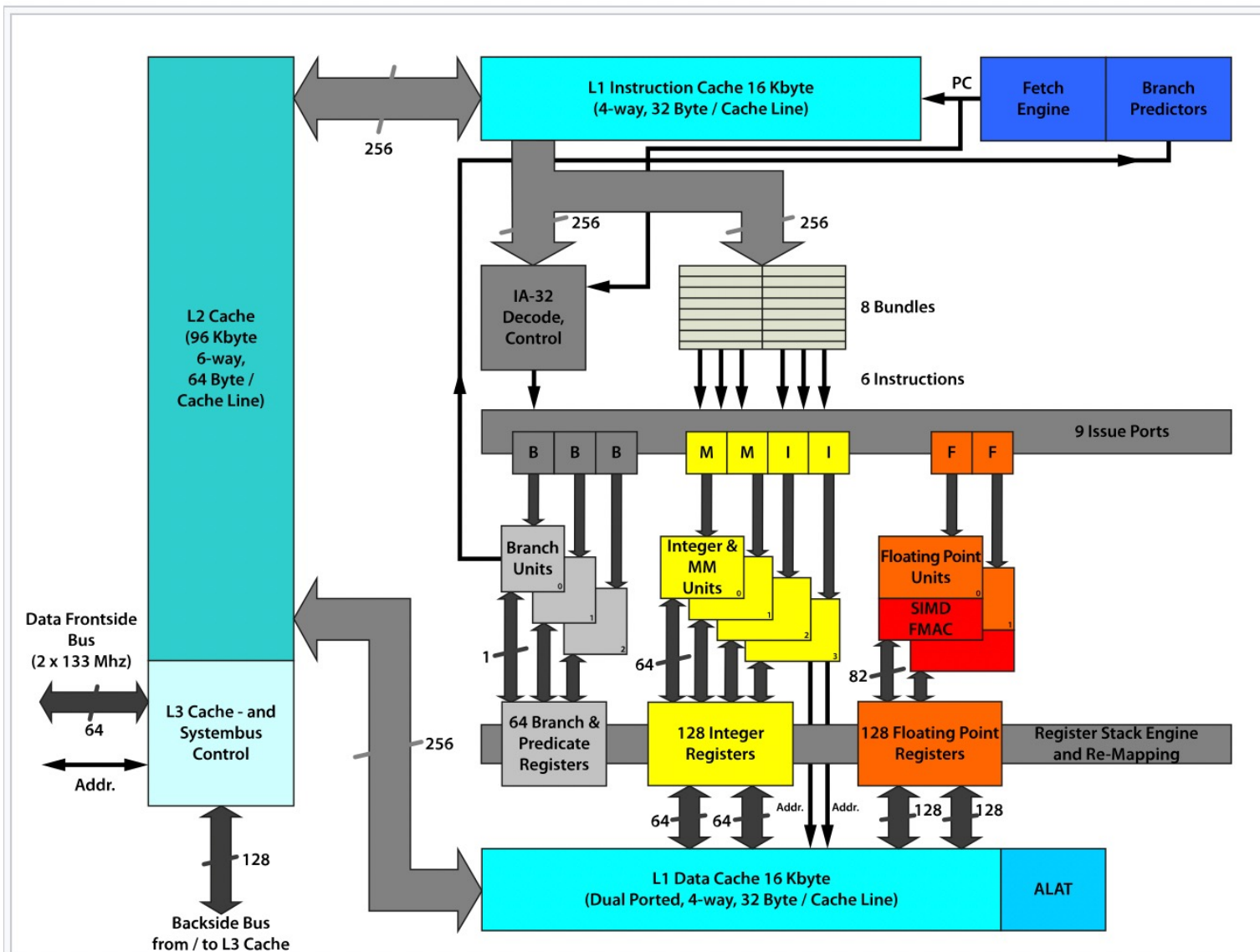
The Itanium architecture is based on explicit [instruction-level parallelism](#), in which the [compiler](#) decides which instructions to execute in parallel. This contrasts with [superscalar](#) architectures, which depend on the processor to manage instruction dependencies at runtime. In all Itanium models, up to and including [Tukwila](#), cores execute up to six [instructions per clock cycle](#). In 2008, Itanium was the fourth-most deployed microprocessor architecture for [enterprise-class systems](#), behind [x86-64](#), [Power ISA](#), and [SPARC](#).<sup>[1]</sup>

# Intel x86-64 from AMD

✧ X86-64 a 64-bit version of the x86 instruction set, first released in 1999. It introduced two new modes of operation, 64-bit mode and compatibility mode, along with a new 4-level paging mode.



# Intel Itanium



The Intel Itanium architecture







# Intel 's New 13<sup>th</sup> Gen



Oct 2022

## 13th Gen Intel® Core™ Unlocked

Processor Number	Processor Cores (P+E)	Processor Threads	Intel® Smart Cache (L3)	Total L2 Cache	P-core Max Turbo Frequency (GHz)	E-core Max Turbo Frequency (GHz)	P-core Base Frequency (GHz)	E-core Base Frequency (GHz)
i9-13900K	24 (8+16)	32	36MB	32MB	Up to 5.8	Up to 4.3	3.0	2.2
i9-13900KF	24 (8+16)	32	36MB	32MB	Up to 5.8	Up to 4.3	3.0	2.2
i7-13700K	16 (8+8)	24	30MB	24MB	Up to 5.4	Up to 4.2	3.4	2.5
i7-13700KF	16 (8+8)	24	30MB	24MB	Up to 5.4	Up to 4.2	3.4	2.5
i5-13600K	14 (6+8)	20	24MB	20MB	Up to 5.1	Up to 3.9	3.5	2.6
i5-13600KF	14 (6+8)	20	24MB	20MB	Up to 5.1	Up to 3.9	3.5	2.6



# Intel 's New 13<sup>th</sup> Gen



Oct 2022

## Desktop Processors

Processor Graphics	Total CPU PCIe Lanes	Max Memory Speed (MT/S)	Memory Capacity	Processor Base Power (W)	Max Turbo Power (W)	RCP (USD)
Intel® UHD Graphics 770	20	DDR5 5600 DDR4 3200	128GB	125	253	\$589
n/a	20	DDR5 5600 DDR4 3200	128GB	125	253	\$564
Intel® UHD Graphics 770	20	DDR5 5600 DDR4 3200	128GB	125	253	\$409
n/a	20	DDR5 5600 DDR4 3200	128GB	125	253	\$384
Intel® UHD Graphics 770	20	DDR5 5600 DDR4 3200	128GB	125	181	\$319
n/a	20	DDR5 5600 DDR4 3200	128GB	125	181	\$294



# Intel CPU's

Power Mgt

## LLC - Dynamic Cache Shrink Feature

- **LLC organized in 16 ways.**
- **When PCU detects low activity workload**
  - Flushes 14 ways of the cache and puts ways to sleep
  - Shrinks active ways from 16 to 2 to improve VccMin
- **When PCU detects high activity**
  - Expands active ways back to 16 to improve cache hit rate.

Power Down LLC

Active

Sleep



# Intel Dev Con 2021



In the figure above, note the units associated with the x86 instructions using vector-based operands, to improve performance of the “dot-product plus accumulate” calculations inherent to deep learning software applications:

- Vector Neural Network Instructions (VNNI, providing int8 calculations)
- Advanced Vector Extensions (AVX-512, for fp16/fp32 calculations)

# Intel Dev Con 2021

## Intel Thread Director

Intelligence built directly into the core

### Monitors the runtime instruction mix

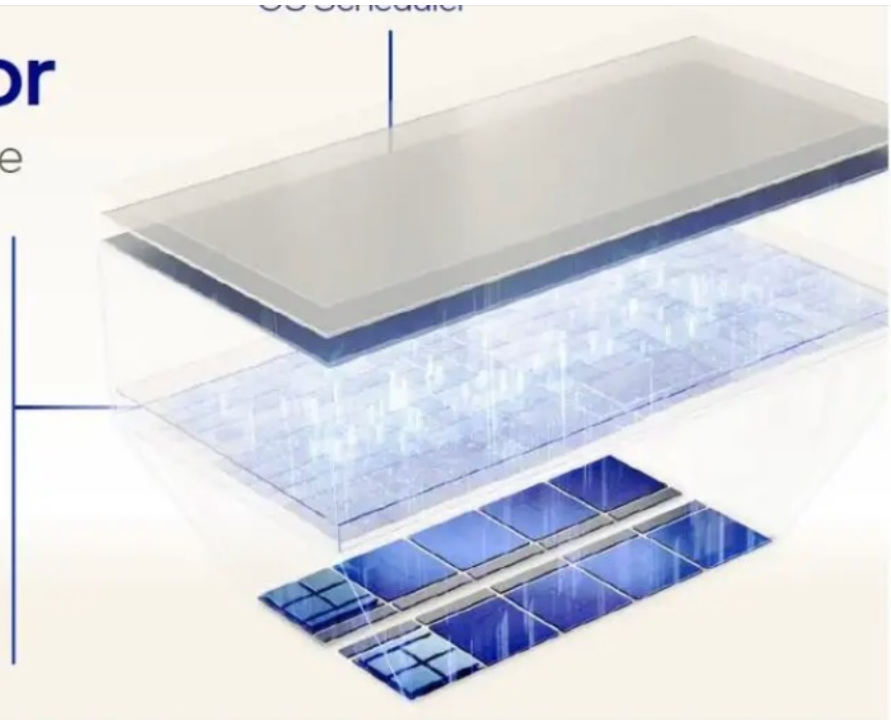
of each thread and as well as the state of each core – with nanosecond precision

### Provides runtime feedback to the OS

to make the optimal scheduling decision for any workload or workflow

### Dynamically adapts guidance

based on the thermal design point, operating conditions, and power settings – without any user input



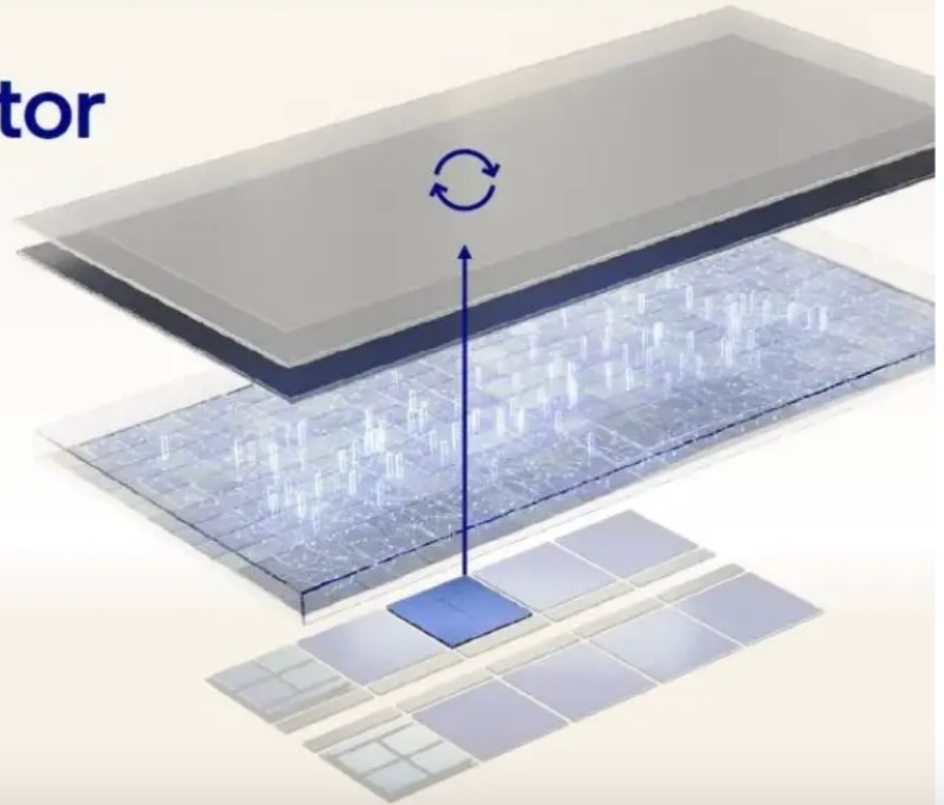
Additionally, based on thread priority, an executing thread could transition between a p-core and e-core. Also, threads may be “parked” or “unparked”.



## Intel Thread Director

### Scheduling Examples

- 1 Priority tasks scheduled on P-cores
- 2 Background tasks scheduled on E-cores
- 3 AI thread prioritized on P-core
- 4 Spin loop wait moved from P to E-core



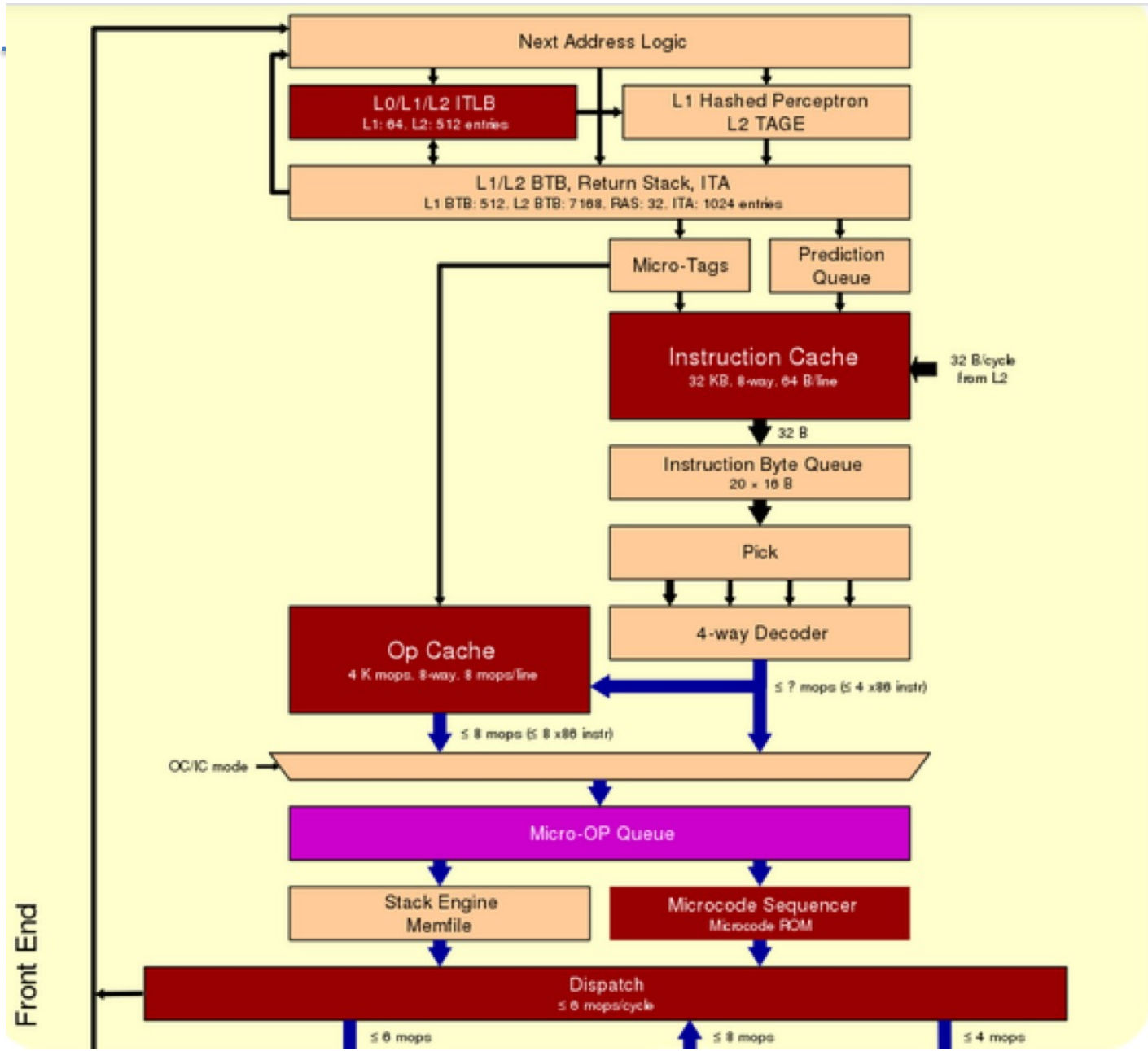
# Intel Dev Con 2021

## Thread Director

Another option is to distribute thread execution across separate (symmetric) cores on the CPU until all cores are busy, before invoking hyperthreading.

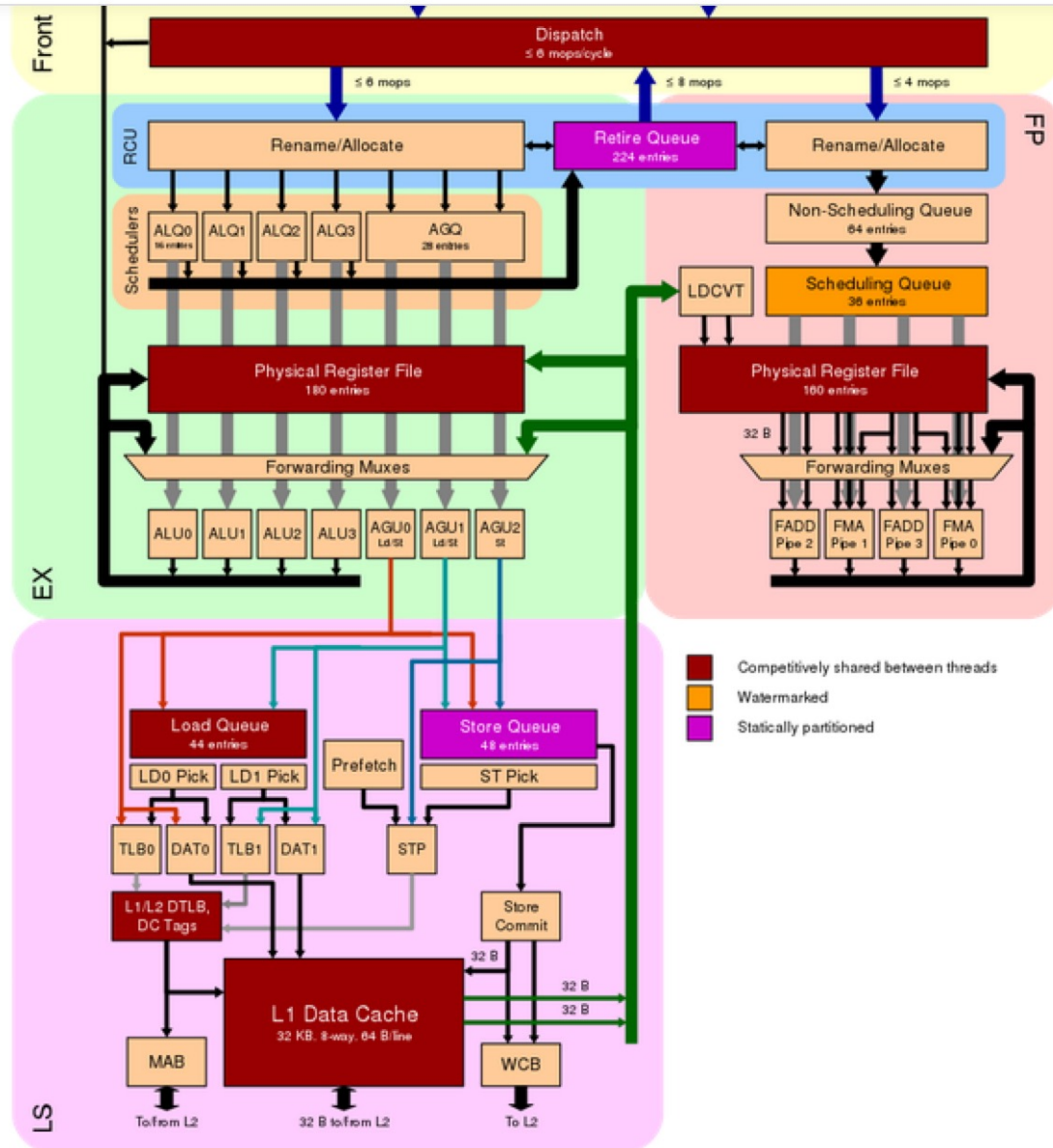
A combination of p-cores and e-cores in the same CPU (otherwise known as a “big/little” architecture) introduces asymmetry into the O/S scheduler algorithm. The simplest approach would be to distinguish threads based on foreground (performance) and background (efficiency) processes – e.g., using “static” rules for scheduling. For the upcoming CPUs with both p- and e-cores, Intel has integrated additional power/performance monitoring circuitry to provide the O/S scheduler with “hints” on the optimum core

# AMD Zen uArch



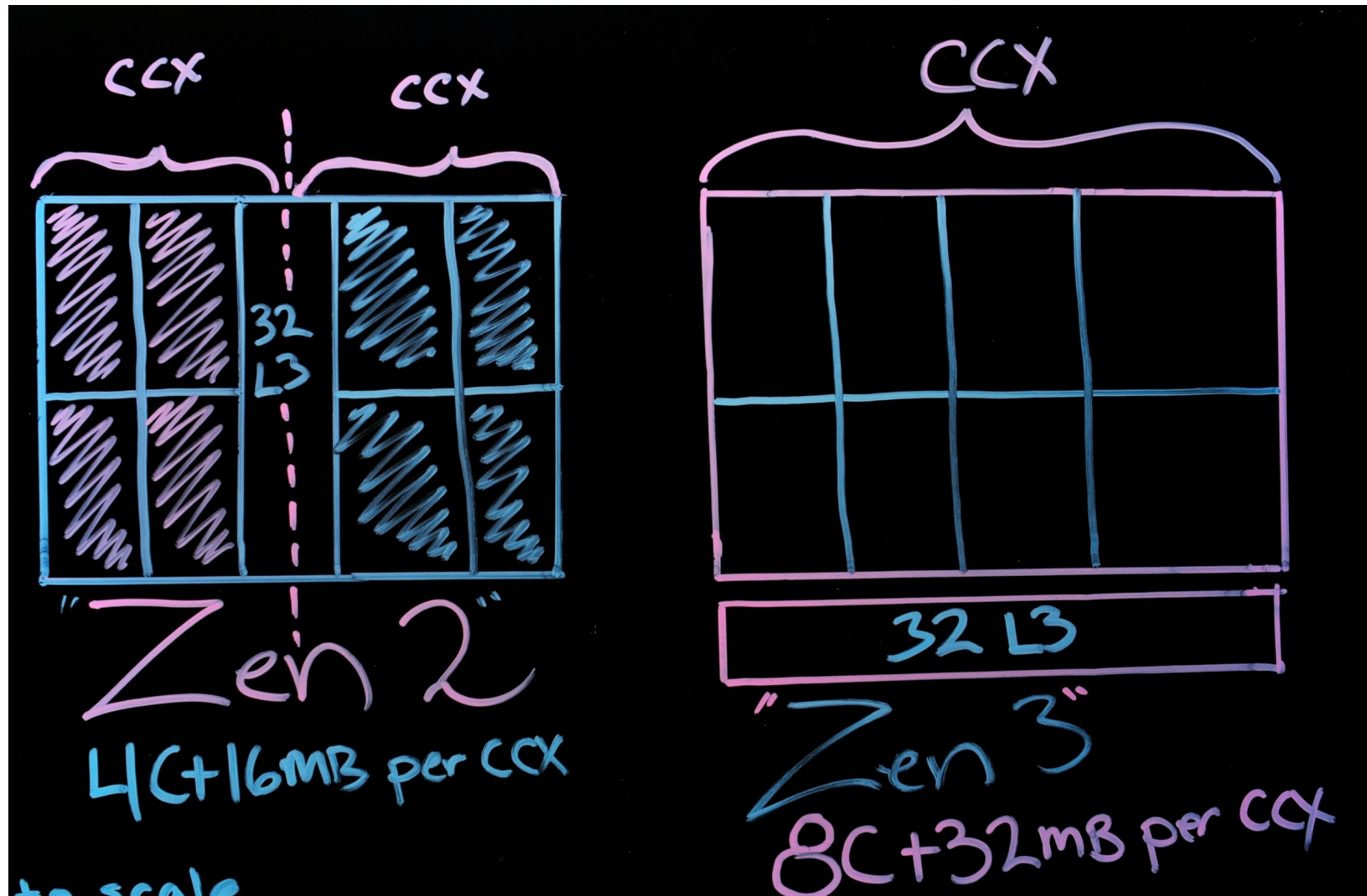


# AMD Zen uArch





# AMD Zen 3 Macro

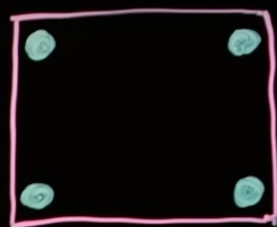


CCX = Compute Chip Complex

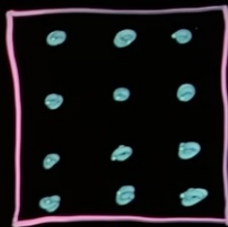
# AMD 3D V-cache



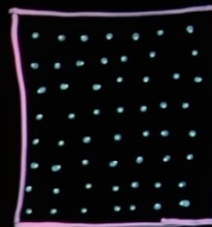
How It's Built: AMD 3D V-Cache Technology



C4

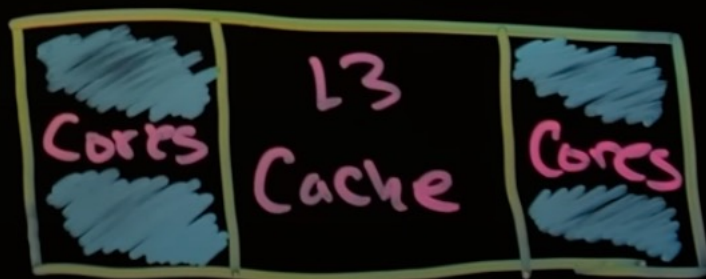
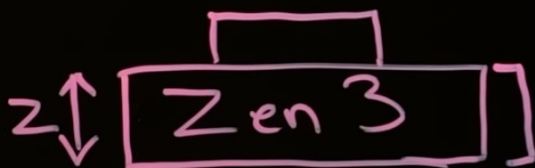


micro  
Bump



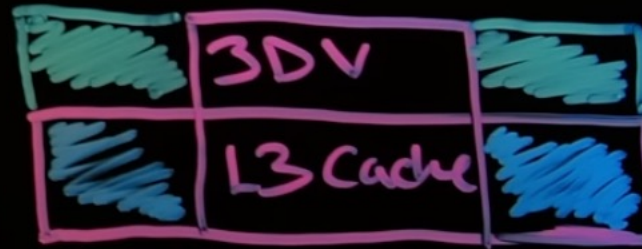
Hybrid  
Bond  
3D

AMD  
3D V-Cache



8 Cores

Zen 3



8 Cores

3D V-Cache

# AMD Radeon Benchmark

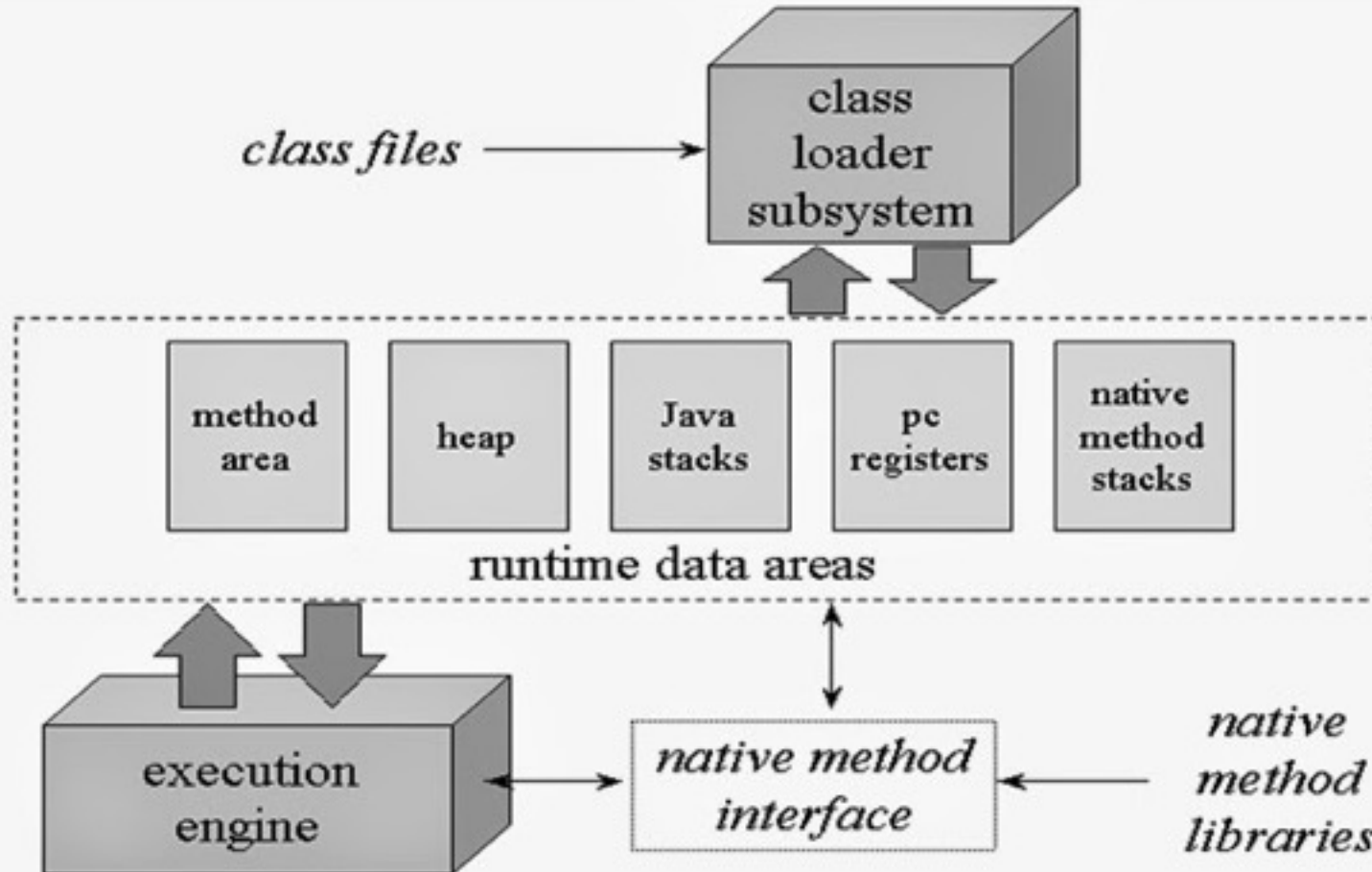


# Section

---

# Multi-Threading in Java

# Java Runtime





# Multi-Threading in Java

Java.lang.Thread

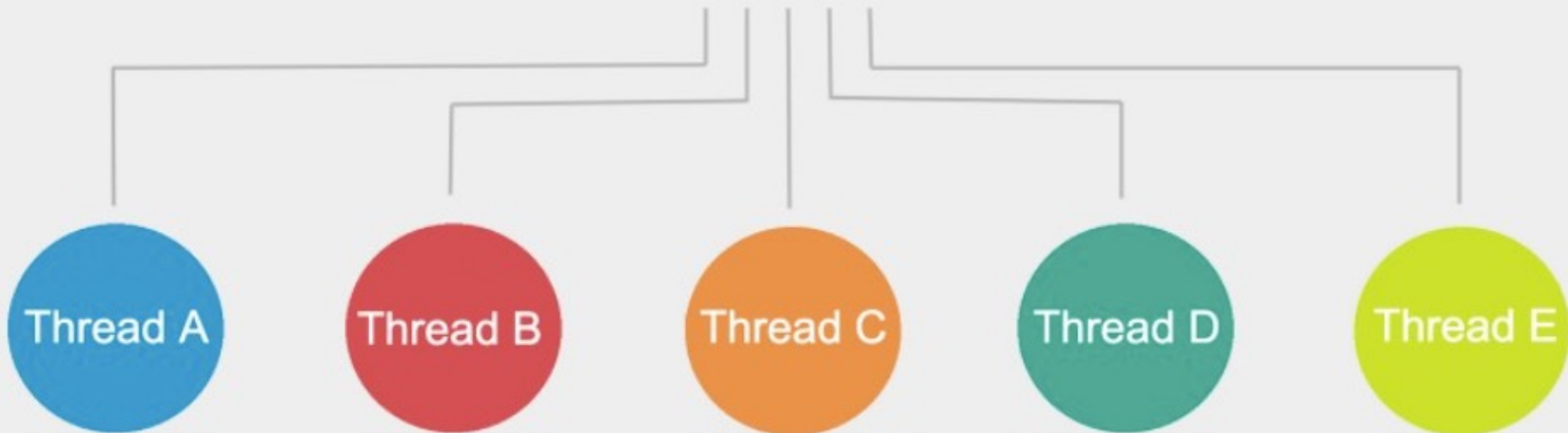


Dmytro Timchenko

Aug 29 · 6 min read

## JAVA CONCURRENCY

### Thread Creation





Medium

Java.lang.Thread

# Multi-Threading in Java

## How to Create a Java Thread

Java lets you create a thread one of three ways:

- By implementing the **Runnable** interface. **.start**
- By implementing the **Callable** interface. **implements**
- By extending the **Thread**. **extends**

## Runnable Interface

The easiest way to create a thread is to create a class that implements the **Runnable** interface. A Java object that implements the **Runnable** interface can be executed by a Java **Thread**.

The **Runnable** interface is a standard **Java Interface** that comes with the Java platform. The **Runnable** interface only has a single method **run()**. Here is an example of implementing the **Runnable** interface:





Medium

# Multi-Threading in Java

Java.lang.Thread

## Run via start

There are two ways of running a thread. One of them is by calling method `start()`:

```
1  @Test
2  public void myThreadTest()
3      throws Exception {
4
5      Thread thread = new MyThread("Test task");
6      thread.start();
7      thread.join();
8  }
```

MyThreadTest.java hosted with ♥ by GitHub

[view raw](#)

The **start()** call will return as soon as the thread is started. It will not wait until the **run()** method is done. The **run()** method will execute as if executed by a different CPU. When the **run()** method executes it will print out provided text.



# Multi-Threading in Java

## Thread Subclass

The other way is to create a subclass of Thread and override the **run()** method. The **run()** method is what is executed by the thread after you call **start()**. Here is an example of creating a Java Thread subclass:

```
1  public class MyThread extends Thread {  
2  
3      private String text;  
4  
5      @Override  
6      public void run() {  
7          System.out.println(text);  
8      }  
9  }
```



Medium

Java.lang.Thread

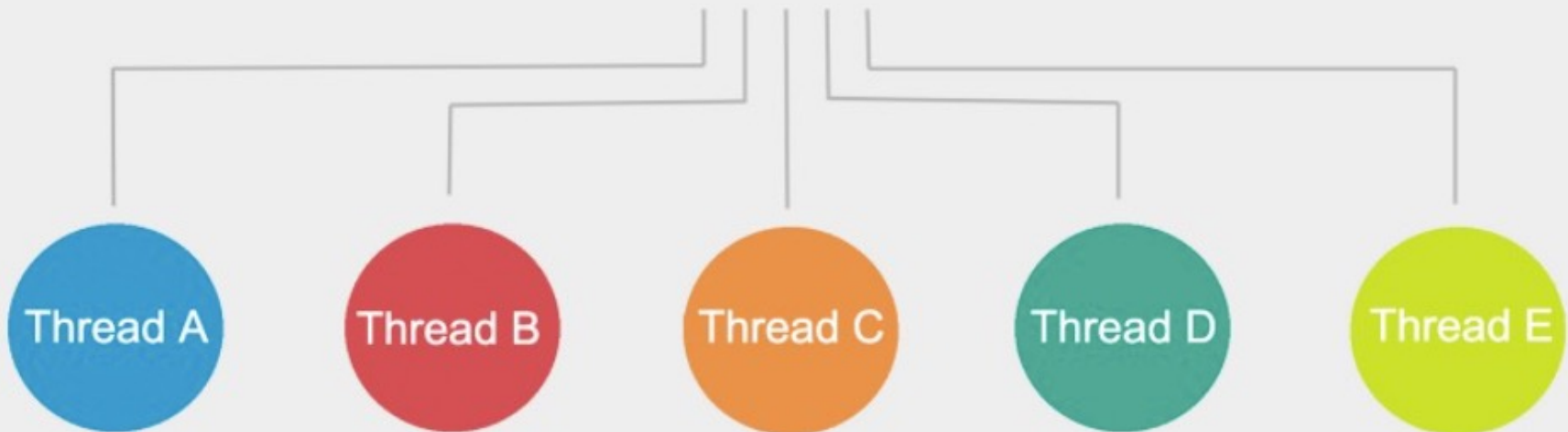


Dmytro Timchenko

Aug 29 · 6 min read

# JAVA CONCURRENCY

## Thread Life Cycle





Medium

# Multi-Threading in Java

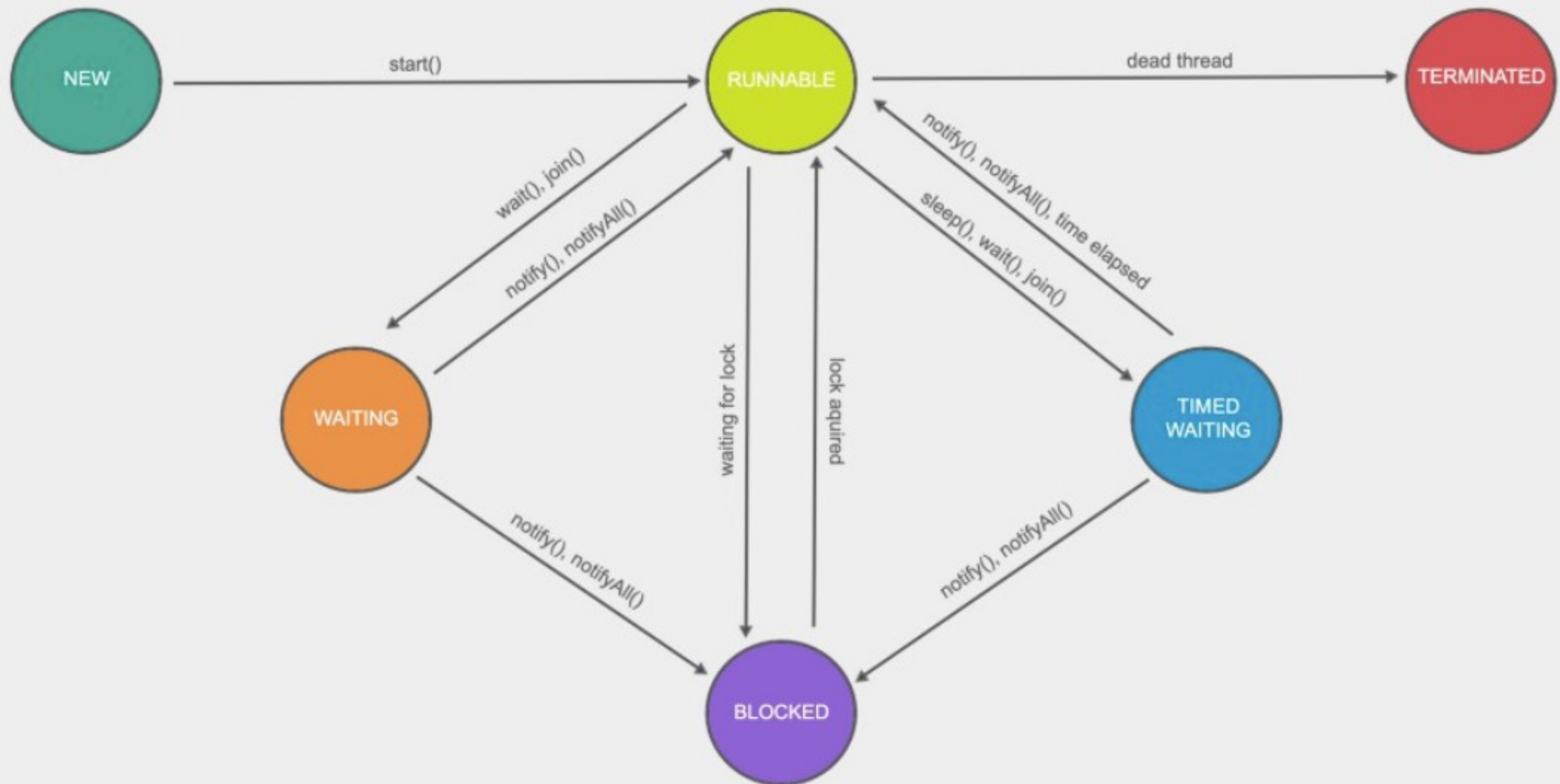
Java.lang.Thread



Dmytro Timchenko

Aug 29 · 6 min read

## THREAD LIFE CYCLE





# Multi-Threading in Java

## Thread states

The `java.lang.Thread` class contains a *static **State enum*** — which defines its potential states. The thread can only be in one of these states at a time:

1. **NEW** — a newly created thread that has not yet started the execution
2. **RUNNABLE** — either running or ready for execution but it's waiting for resource allocation
3. **WAITING** — waiting for some other thread to perform a particular action without any time limit
4. **TIMED\_WAITING** — waiting for some other thread to perform a specific action for a specified period
5. **BLOCKED** — waiting to acquire a lock to enter or re-enter a synchronized block/method
6. **TERMINATED** — has completed its execution



Medium

NEW

# Multi-Threading in Java

Java.lang.Thread



A *NEW Thread* is a thread that's been created but not yet started. It remains in this state until we start it using the *start()* method.

The following code snippet shows a newly created thread that's in the *NEW* state:

```
Thread t = new MyThread();
```

## RUNNABLE

When we've created a new thread and called the *start()* method on that, it's moved from *NEW* to *RUNNABLE* state. **Threads in this state are either running or ready to run, but they're waiting for resource allocation from the system.**

In a multi-threaded environment, the Thread-Scheduler (which is part of JVM) allocates a fixed amount of time to each thread. So it runs for a particular amount of time, then passes the control to other *RUNNABLE*





Medium

# Multi-Threading in Java



Java.lang.Thread

```
1  public class Waiting implements Runnable {
2      public static Thread threadA;
3
4      public static void main(String[] args) {
5          threadA = new Thread(new Waiting());
6          threadA.start();
7      }
8
9      public void run() {
10         Thread threadB = new Thread(new Sleeping());
11         threadB.start();
12         try {
13             threadB.join();
14         } catch (InterruptedException e) {
15             Log.error("Interrupt exception: ", e);
16         }
17     }
18
19
20     public static class Sleeping implements Runnable {
21         public void run() {
22             try {
23                 Thread.sleep(5000);
24             } catch (InterruptedException e) {
25                 Log.error("Interrupt exception: ", e);
26             }
27         }
28     }
29 }
```



Medium

WAITING

# Multi-Threading in Java

Java.lang.Thread



A thread is in *WAITING* state when it's waiting for some other thread to **perform a particular action**. According to JavaDocs, any thread can enter this state by calling any one of the following three methods:

1. *object.wait()*
2. *thread.join()* or
3. *LockSupport.park()*

Here is an example:

```
1  public class Waiting implements Runnable {
2      public static Thread threadA;
3
4      public static void main(String[] args) {
5          threadA = new Thread(new Waiting());
6          threadA.start();
7      }
8
9      public void run() {
10         Thread threadB = new Thread(new Sleeping());
11         threadB.start();
12         try {
13             threadB.join();
```



# Multi-Threading in Java

Java.lang.Thread

## TIMED WAITING

A thread is in *TIMED\_WAITING* state when it's waiting for another thread to perform a particular action within a specified amount of time.

According to JavaDocs, there are five ways to put a thread on *TIMED\_WAITING* state:

1. `thread.sleep(long millis)`
2. `wait(int timeout)` or `wait(int timeout, int nanos)`
3. `thread.join(long millis)`
4. `LockSupport.parkNanos`
5. `LockSupport.parkUntil`

```

1  public class TimedWaiting {
2      public static void main(String[] args) throws InterruptedException {
3          Thread threadA = new Thread(new Sleeping());
4          threadA.start();
5
6          // This will start processing threadA
7          Thread.sleep(1000);
8      }
9
10     class Sleeping implements Runnable {
11         @Override
12         public void run() {
13             try {
14                 Thread.sleep(5000);
15             } catch (InterruptedException e) {
16                 Log.error("Interrupted exception: ", e);
17             }
18         }
19     }
20 }
    
```



Medium  
BLOCKED

Java.lang.Thread



# Multi-Threading in Java

A thread is in the *BLOCKED* state when it's currently not eligible to run. It enters this state when it is waiting for a lock and is trying to access a section of code that is locked by some other thread.

Here is an example:

```

1  public class Blocking {
2      public static void main(String[] args) throws InterruptedException {
3          Thread threadA = new Thread(new InfiniteClass());
4          Thread threadB = new Thread(new InfiniteClass());
5
6          threadA.start();
7          threadB.start();
8
9          Thread.sleep(1000);
10     }
11 }
12
13 class InfiniteClass implements Runnable {
14
15     @Override
16     public void run() {
17         infinityMethod();
18     }
19
20     public static synchronized void infinityMethod() {
21         while(true) {

```



Medium

TERMINATED

# Multi-Threading in Java



Java.lang.Thread

This is the state of a dead thread. It's in the *TERMINATED* state when it has either finished execution or was terminated abnormally.

Here is an example:

```

1  public class Terminating implements Runnable {
2      public static void main(String[] args) throws InterruptedException {
3          Thread threadA = new Thread(new Terminating());
4          threadA.start();
5
6          // The following sleep time will be enough for
7          // threadA is completed
8          Thread.sleep(1000);
9      }
10
11     @Override
12     public void run() {
13         // empty method
14     }
15 }
    
```



Medium

Java.lang.Thread



Dmytro Timchenko in Javarevisited

Sep 4 · 3 min read ★



## How To Kill A Thread

# Java Concurrency: How To Kill A Thread

In this article, we'll cover stopping a Thread in Java, what is...





Medium

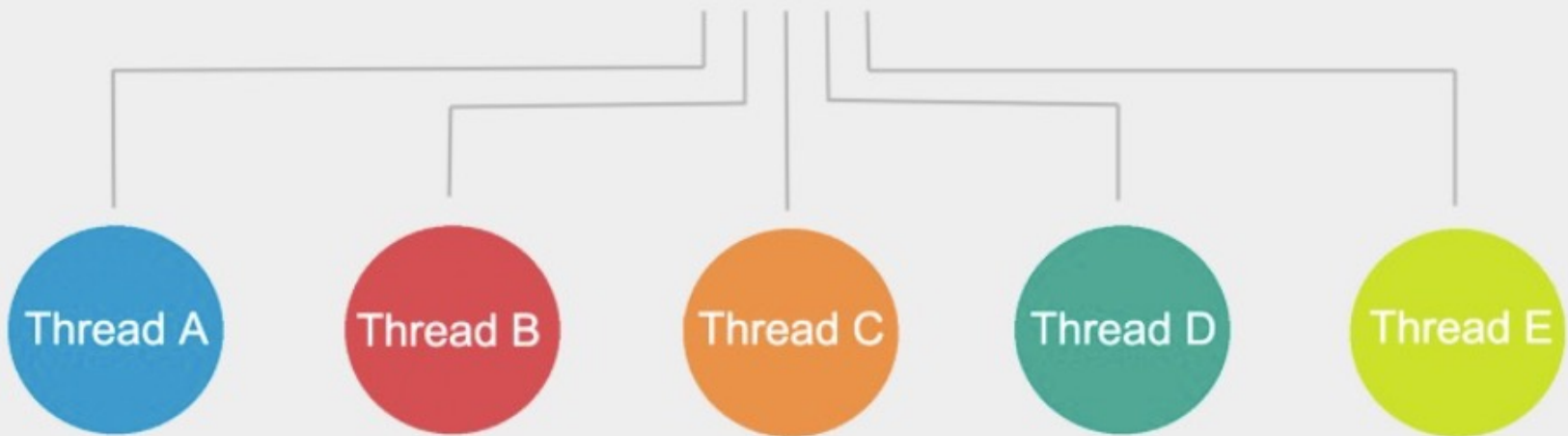
Java.lang.Thread



Dmytro Timchenko  
Aug 29 · 6 min read

# JAVA CONCURRENCY

## Thread Methods



# Multi-Threading in Java

M

Medium

Java.lang.Thread

Methods

## Java Concurrency: Thread Methods

• `Thread.currentThread()`

• `long getId()`

• `int getPriority()`

• `Thread.State getState()`

• `void interrupt()`

• `void join()`

• `void run()`

• `void setDaemon(boolean on)`

• `void setPriority(int newPriority)`

• `static void sleep(long millis)`

• `void start()`

• `static void yield()`

dae·mon<sup>2</sup> | 'dēmən | (also demon)

noun *Computing*

a background process that handles requests for services such as print spooling and file transfers, and is dormant when not required.

join()

# Multi-Threading in Java





# Multi-Threading in Java

## `interrupt()`

In the multithreading programming, there are cases when you need to stop some thread from execution — interrupt a thread. An *interrupt* is an indication to a thread that it should stop what it is doing and do something else. It's up to the programmer to decide exactly how a thread responds to an interrupt, but it is very common for the thread to terminate.



Medium

# Multi-Threading in Java



Java.lang.Thread

`Thread.currentThread()`

This method is available in package **java.lang.Thread**. It is used to return a reference to the currently executing thread object. This is a static method so it is accessible with a class name too. The return type of this method is **Thread**, it returns a reference of currently executing thread object. It does not raise any exception.

```
1  class ThreadA extends Thread {
2      public void run() {
3          System.out.println("The name of this thread is " + " " + Thread.currentThread().
4      }
5  }
6
7  class ThreadB extends Thread {
8      public void run() {
9          System.out.println("The name of this thread is " + " " + Thread.currentThread().
10     }
11 }
12
13 public class MainThread {
14
15     public static void main(String[] args) {
16
17         System.out.println("The name of this thread is " + " " + Thread.currentThread().
18
19         ThreadA threadA = new ThreadA();
20         threadA.start();
21     }
```

# Section

## Parallel Processing

- ☐ MP/SMP
- ☐ SIMD
- ☐ MMX/AVX/AMX
- ☐ ILP



# Parallel Processing

P&H Ch 6

**COMP 122: Computer  
 Architecture and  
 Assembly Language**  
 Spring 2020

Figure 6.1.2: Hardware/software categorization and examples of application perspective on concurrency versus hardware perspective on parallelism (COD Figure 6.1).

		Software	
		Sequential	Concurrent
Hardware	Serial	Matrix Multiply written in MatLab running on an Intel Pentium 4	Windows Vista Operating System running on an Intel Pentium 4
	Parallel	Matrix Multiply written in MATLAB running on an Intel Core i7	Windows Vista Operating System running on an Intel Core i7

# Parallel Processing

## P&H Ch 6

**Multiprocessor:** A computer system with at least two processors. This computer is in contrast to a uniprocessor, which has one, and is increasingly hard to find today.

Since multiprocessor software should scale, some designs support operation in the presence of broken hardware; that is, if a single processor fails in a multiprocessor with  $n$  processors, these systems would continue to provide service with  $n - 1$  processors. Hence, multiprocessors can also improve availability (see COD Chapter 5 (Large and Fast: Exploiting Memory Hierarchy)).



High performance can mean high throughput for independent tasks, called *task-level parallelism* or *process-level parallelism*. These tasks are independent single-threaded applications, and they are an important and popular use of multiple processors. This approach is in contrast to running a single job on multiple processors. We use the term *parallel processing program* to refer to a single program that runs on multiple processors simultaneously.

**Task-level parallelism** or **process-level parallelism:** Utilizing multiple processors by running independent programs simultaneously.

**Parallel processing program:** A single program that runs on multiple processors simultaneously.

There have long been scientific problems that have needed much faster computers, and this class of problems has been used to justify many novel parallel computers over the decades. Some of these problems can be handled simply today, using a cluster composed of microprocessors housed in many independent servers (see COD Section 6.7 (Clusters, warehouse scale computers, and other message-passing multiprocessors)). In addition, clusters can serve equally demanding applications outside the sciences, such as search engines, Web servers, email servers, and databases.

**Cluster:** A set of computers connected over a local area network that function as a single large multiprocessor.

# Parallel Processing



## P&H Ch 6

The difficulty with parallelism is not the hardware; it is that too few important application programs have been rewritten to complete tasks sooner on multiprocessors. It is difficult to write software that uses multiple processors to complete one task faster, and the problem gets worse as the number of processors increases.

Why has this been so? Why have parallel processing programs been so much harder to develop than sequential programs?

The first reason is that you *must* get better performance or better energy efficiency from a parallel processing program on a multiprocessor; otherwise, you would just use a sequential program on a uniprocessor, as sequential programming is simpler. In fact, uniprocessor design techniques such as superscalar and out-of-order execution take advantage of instruction-level parallelism (see COD Chapter 4 (The Processor)), normally without the involvement of the programmer. Such innovations reduced the demand for rewriting programs for multiprocessors, since programmers could do nothing and yet their sequential programs would run faster on new computers.

Why is it difficult to write parallel processing programs that are fast, especially as the number of processors increases? In COD Chapter 1 (Computer Abstractions and Technology), we used the analogy of eight reporters trying to write a single story in hopes of doing the work eight times faster. To succeed, the task must be broken into eight equal-sized pieces, because otherwise some reporters would be idle while waiting for the ones with larger pieces to finish. Another speed-up obstacle could be that the reporters would spend too much time communicating with each other instead of writing their pieces of the story. For both this analogy and parallel programming, the challenges include scheduling, partitioning the work into parallel pieces, balancing the load evenly between the workers, time to synchronize, and overhead for communication between the parties. The challenge is stiffer with the more reporters for a newspaper story and with the more processors for parallel programming.

# Parallel Processing

## P&H Ch 6

**Multicore microprocessor:** A microprocessor containing multiple processors ("cores") in a single integrated circuit. Virtually all microprocessors today in desktops and servers are multicore.

**Shared memory multiprocessor (SMP):** A parallel processor with a single physical address space.

The state of technology today means that programmers who care about performance must become parallel programmers, for sequential code now means slow code.

The tall challenge facing the industry is to create hardware and software that will make it easy to write correct parallel processing programs that will execute efficiently in performance and energy as the number of cores per chip scales.

**zyBooks** My library > COMP 222: Computer Organization h... >  
6.2: Difficulty of parallel programs

 Adopt a zyBook

 Instructor forum

 Help/FAQ

 Jeffre

**Strong scaling:** Speed-up achieved on a multiprocessor without increasing the size of the problem.

**Weak scaling:** Speed-up achieved on a multiprocessor while increasing the size of the problem proportionally to the increase in the number of processors.

Note that the **memory hierarchy** can interfere with the conventional wisdom about weak scaling being easier than strong scaling. For example, if the weakly scaled dataset no longer fits in the last level cache of a multicore microprocessor, the resulting performance could be much worse than by using strong scaling.

Depending on the application, you can argue for either scaling approach. For example, the TPC-C debit-credit database benchmark requires that you scale up the number of customer accounts in proportion to the higher transactions per

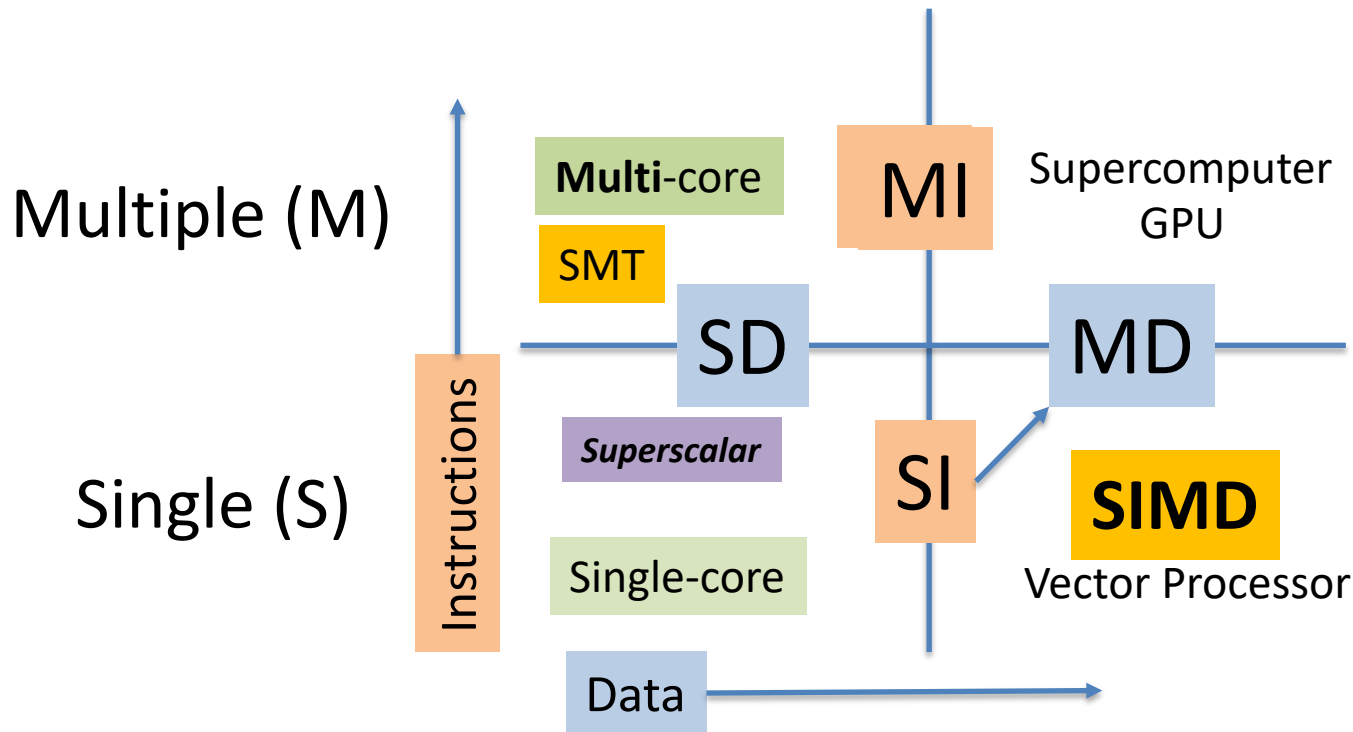




# I-D Parallelism: SIMD

## Flynn Partition

Michael J. Flynn paper (U Illinois (UIUC), Ca 1969)



## 6.3 SISD, MIMD, SIMD, SPMD, and vector

(Original section<sup>1</sup>)

One categorization of parallel hardware proposed in the 1960s is still used today. It was based on the number of instruction streams and the number of data streams. The figure below shows the categories. Thus, a conventional uniprocessor has a single instruction stream and single data stream, and a conventional multiprocessor has multiple instruction streams and multiple data streams. These two categories are abbreviated *SISD* and *MIMD*, respectively.

***SISD*** or ***single instruction stream, single data stream***: A uniprocessor.

***MIMD*** or ***multiple instruction streams, multiple data streams***: A multiprocessor.

Figure 6.3.1: Hardware categorization and examples based on number of instruction streams and data streams: SISD, SIMD, MISD, and MIMD (COD Figure 6.2).

		Data Streams	
		Single	Multiple
Instruction Streams	Single	SISD: Intel Pentium 4	SIMD: SSE instructions of x86
	Multiple	MISD: No examples today	MIMD: Intel Core i7



**Data-level parallelism:** Parallelism achieved by performing the same operation on independent data.

The so-called array processors that inspired the SIMD category have faded into history (see COD Section 6.15 (Historical perspective and further reading)), but two current interpretations of SIMD remain active today.

## SIMD in x86: Multimedia extensions

As described in COD Chapter 3 (Arithmetic for Computers), subword parallelism for narrow integer data was the original inspiration of the Multimedia Extension (MMX) instructions of the x86 in 1996. As **Moore's Law** continued, more instructions were added, leading first to *Streaming SIMD Extensions* (SSE) and now *Advanced Vector Extensions* (AVX). AVX supports the simultaneous execution of four 64-bit floating-point numbers. The width of the operation and the registers is encoded in the opcode of these multimedia instructions. As the data width of the registers and operations grew, the number of opcodes for multimedia instructions exploded, and now there are hundreds of SSE and AVX instructions (see COD Chapter 3 (Arithmetic for Computers)).



## Vector

**SSE → AVX**

An older and, as we shall see, more elegant interpretation of SIMD is called a *vector architecture*, which has been closely identified with computers designed by Seymour Cray starting in the 1970s. It is also a great match to problems with lots of data-level parallelism. Rather than having 64 ALUs perform 64 additions simultaneously, like the old array processors, the vector architectures pipelined the ALU to get good performance at lower cost. The basic philosophy of vector architecture is to collect data elements from memory, put them in order into a large set of registers, operate on them sequentially in registers using **pipelined execution units**, and then write the results back to memory. A key feature of vector architectures is then a set of vector registers. Thus, a vector architecture might have 32 vector registers, each with 64 64-bit elements.



## Example 6.3.1: Comparing vector to conventional code.

Suppose we extend the MIPS instruction set architecture with vector instructions and vector registers. Vector operations use the same names as MIPS operations, but with the letter "V" appended. For example, **addv.d** adds two double-precision vectors. The vector instructions take as their input either a pair of vector registers (**addv.d**) or a vector register and a scalar register (**addvs.d**). In the latter case, the value in the scalar register is used as the input for all operations—the operation **addvs.d** will add the contents of a scalar register to each element in a vector register. The names **lv** and **sv** denote vector load and vector store, and they load or store an entire vector of double-precision data. One operand is the vector register to be loaded or stored; the other operand, which is a MIPS general-purpose register, is the starting address of the vector in memory. Given this short description, show the conventional MIPS code versus the vector MIPS code for

$$Y = a \times X + Y$$

MADD

where  $X$  and  $Y$  are vectors of 64 double precision floating-point numbers, initially resident in memory, and  $a$  is a scalar double precision variable. (This example is the so-called DAXPY loop that forms the inner loop of the Linpack benchmark; DAXPY stands for double ax X plus Y). Assume that the starting addresses of  $X$  and  $Y$  are in **\$s0** and **\$s1**, respectively.

➤ **addv.d**

➤ **addvs.d**

# SIMD ISA Extensions

## ❖ SIMD

- MIPS
- ARM

❖ MMX = Multi-Media Extensions

❖ SSE = Streaming SIMD Extensions



❖ AVX = Advanced Vector Extensions

- Intel
- AMD

# SIMD/AVX Extensions

Integer or FP

128/256/512 Bits *Wide*

all ***subword*** sizes

B/H/W/D

➤ AVX-128/256/512



**N** x 8/16/32/64

# SIMD

## MIPS

## P&H Ch 6


### Answer

Here is the conventional MIPS code for DAXPY:

```

        l.d      $f0, a($sp)      : load scalar a
        addiu    $t0, $s0, #512    : upper bound of what to load
loop:   l.d      $f2, 0($s0)       : load x(i)
        mul.d    $f2, $f2, $f0     : a x x(i)
        l.d      $f4, 0($s1)       : load y(i)
        add.d    $f4, $f4, $f2     : a x x(i) + y(i)
        s.d      $f4, 0($s1)       : store into y(i)
        addiu    $s0, $s0, #8      : increment index to x
        addiu    $s1, $s1, #8      : increment index to y
        subu     $t1, $t0, $s0     : compute bound
        bne      $t1, $zero, loop  : check if done
    
```

Here is the vector MIPS code for DAXPY:



```

        l.d      $f0, a($sp)      : load scalar a
        lv       $v1, 0($s0)       : load vector x
        mulvs.d  $v2, $v1, $f0     : vector-scalar multiply
        lv       $v3, 0($s1)       : load vector y
        addv.d   $v4, $v2, $v3     : add y to product
        sv       $v4, 0($s1)       : store the result
    
```



There are some interesting comparisons between the two code segments in this example. The most dramatic is that the vector processor greatly reduces the dynamic instruction bandwidth, executing only 6 instructions versus almost 600 for the traditional MIPS architecture. This reduction occurs both because the vector operations work on 64 elements at a time and because the overhead instructions that constitute nearly half the loop on MIPS are not present in the vector code. As you might expect, this reduction in instructions fetched and executed saves energy.

Another important difference is the frequency of **pipeline** hazards (COD Chapter 4 (The Processor)). In the straightforward MIPS code, every **add.d** must wait for a **mul.d**, every **s.d** must wait for the **add.d** and every **add.d** and **mul.d** must wait on **l.d**. On the vector processor, each vector instruction will only stall for the first element in each vector, and then subsequent elements will flow smoothly down the pipeline. Thus, pipeline stalls are required only once per vector *operation*, rather than once per vector *element*. In this example, the pipeline stall frequency on MIPS will be about 64 times higher than it is on the vector version of MIPS. The pipeline stalls can be reduced on MIPS by using loop unrolling (see COD Chapter 4 (The Processor)). However, the large difference in instruction bandwidth cannot be reduced.



Since the vector elements are independent, they can be operated on in parallel, much like subword parallelism for AVX instructions. All modern vector computers have vector functional units with multiple parallel pipelines (called *vector lanes*; see COD Figures 6.2 (Hardware categorization and examples ...) and 6.3 (Using multiple functional units to improve the performance ...)) that can produce two or more results per clock cycle.

### Elaboration

*The loop in the example above exactly matched the vector length. When loops are shorter, vector architectures use a register that reduces the length of vector operations. When loops are larger, we add bookkeeping code to iterate full-length vector operations and to handle the leftovers. This latter process is called strip mining.*



## Vector versus scalar

Vector instructions have several important properties compared to conventional instruction set architectures, which are called *scalar architectures* in this context:

- A single vector instruction specifies a great deal of work—it is equivalent to executing an entire loop. The instruction fetch and decode bandwidth needed is dramatically reduced.
- By using a vector instruction, the compiler or programmer indicates that the computation of each result in the vector is independent of the computation of other results in the same vector, so hardware does not have to check for data hazards within a vector instruction.
- Vector architectures and compilers have a reputation of making it much easier than when using MIMD multiprocessors to write efficient applications when they contain data-level parallelism.
- Hardware need only check for data hazards between two vector instructions once per vector operand, not once for every element within the vectors. Reduced checking can save energy as well as time.
- Vector instructions that access memory have a known access pattern. If the vector's elements are all adjacent, then fetching the vector from a set of heavily interleaved memory banks works very well. Thus, the cost of the latency to main memory is seen only once for the entire vector, rather than once for each word of the vector.
- Because an entire loop is replaced by a vector instruction whose behavior is predetermined, control hazards that would normally arise from the loop branch are nonexistent.
- The savings in instruction bandwidth and hazard checking plus the efficient use of memory bandwidth give vector architectures advantages in power and energy versus scalar architectures.

For these reasons, vector operations can be made faster than a sequence of scalar operations on the same number of data items, and designers are motivated to include vector units if the application domain can often use them.

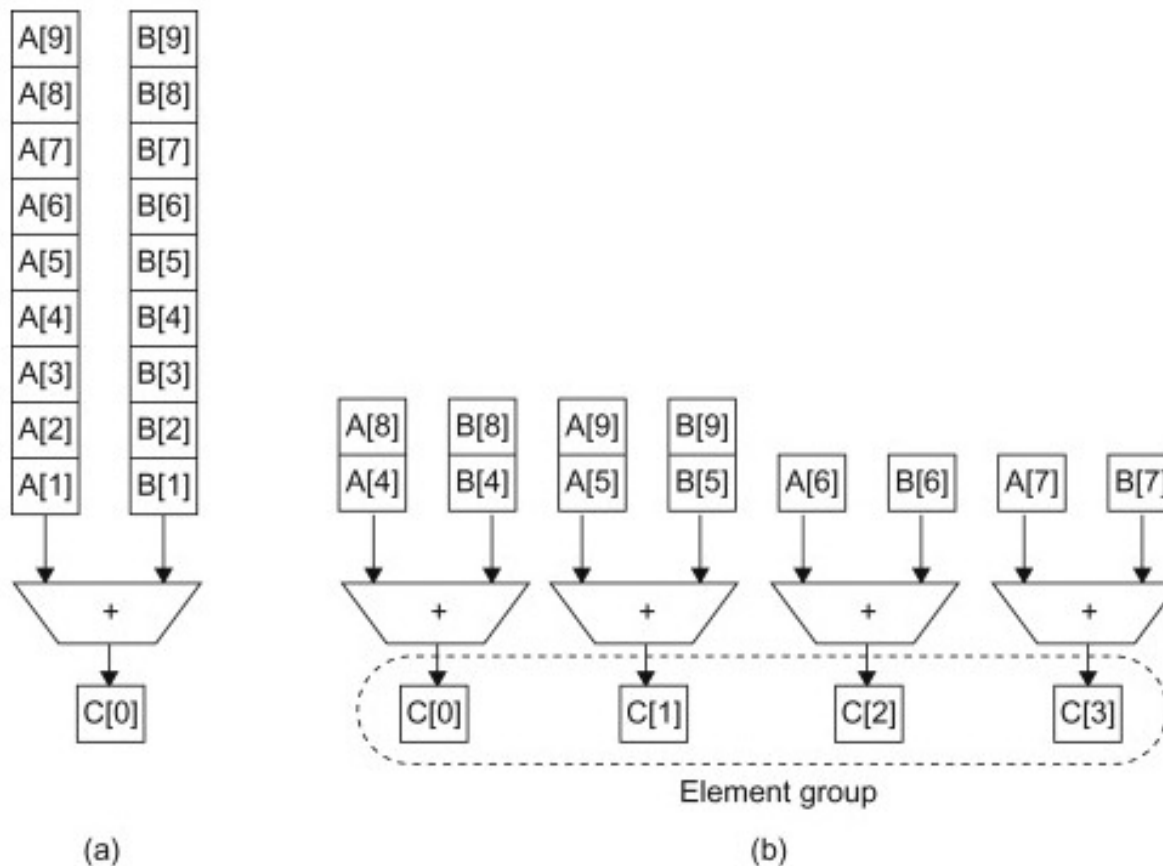
# SIMD

## P&H Ch 6

COMP222

Figure 6.3.2: Using multiple functional units to improve the performance of a single vector add instruction,  $C = A + B$  (COD Figure 6.3).

The vector processor (a) on the left has a single add pipeline and can complete one addition per cycle. The vector processor (b) on the right has four add pipelines or lanes and can complete four additions per cycle. The elements within a single vector add instruction are interleaved across the four lanes.



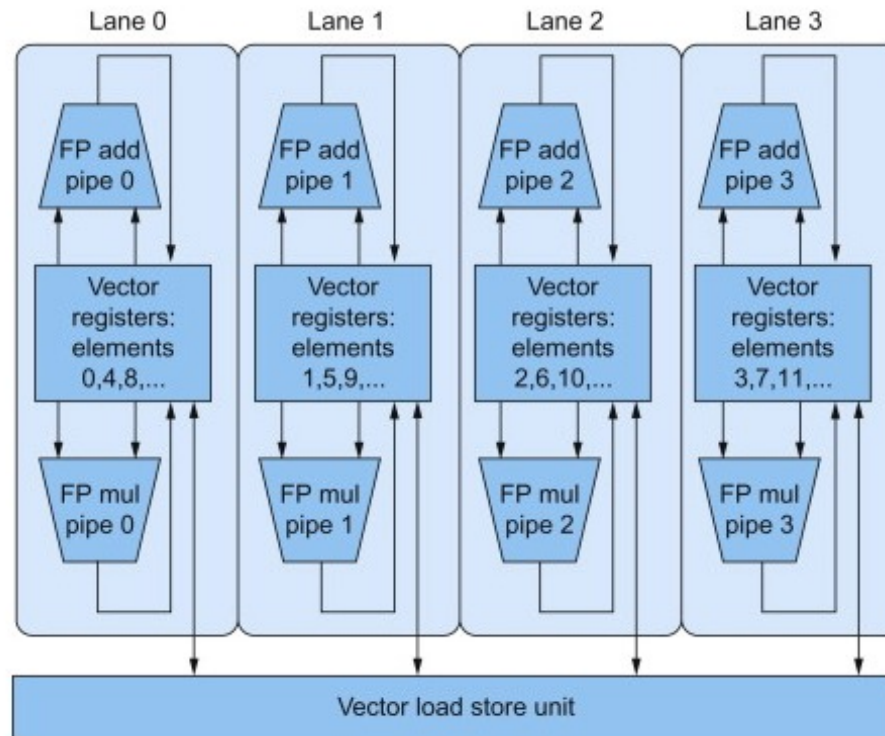
**Vector lane:** One or more vector functional units & traffic speed, multiple lanes execute vector operations

## P&H Ch 6

The vector-register storage is divided across the lanes, with each lane holding every fourth element of each vector register. The figure shows three vector functional units: an FP add, an FP multiply, and a load-store unit. Each of the vector arithmetic units contains four execution pipelines, one per lane, which acts in concert to complete a single vector instruction. Note how each section of the vector-register file only needs to provide enough read and write ports (see COD Chapter 4 (The Processor)) for functional units local to its lane.

$$Y = aX + b$$

$$Y = aX$$



Generally, vector architectures are a very efficient way to execute data parallel processing programs; they are better matches to compiler technology than multimedia extensions; and they are easier to evolve over time than the multimedia extensions to the x86 architecture.

Given these classic categories, we next see how to exploit parallel streams of instructions to improve the performance of a *single* processor, which we will reuse with multiple processors.

## Elaboration

*Given the advantages of vector, why aren't they more popular outside high-performance computing? There were concerns about the larger state for vector registers increasing context switch time and the difficulty of handling page faults in vector loads and stores, and SIMD instructions achieved some of the benefits of vector instructions. In addition, as long as advances in instruction level parallelism could deliver on the performance promise of Moore's Law, there was little reason to take the chance of changing architecture styles.*

## Elaboration

*Another advantage of vector and multimedia extensions is that it is relatively easy to extend a scalar instruction set architecture with these instructions to improve performance of data parallel operations.*

## Elaboration

*The Haswell-generation x86 processors from Intel support AVX2, which has a gather operation but not a scatter operation.*



## Advanced Vector Extensions

From Wikipedia, the free encyclopedia

**Advanced Vector Extensions (AVX)**, also known as **Sandy Bridge New Extensions** are extensions to the [x86 instruction set architecture](#) for [microprocessors](#) from [Intel](#) and [AMD](#) proposed by Intel in March 2008 and first supported by Intel with the [Sandy Bridge](#)<sup>[1]</sup> processor shipping in Q1 2011 and later on by AMD with the [Bulldozer](#)<sup>[2]</sup> processor shipping in Q3 2011. AVX provides new features, new instructions and a new coding scheme.

**AVX2** (also known as **Haswell New Instructions**) expands most integer commands to 256 bits and introduces fused multiply-accumulate ([FMA](#)) operations. They were first supported by Intel with the Haswell processor, which shipped in 2013.

**AVX-512** expands AVX to 512-bit support using a new [EVEX prefix](#) encoding proposed by Intel in July 2013 and first supported by Intel with the [Knights Landing](#) processor, which shipped in 2016.<sup>[3][4]</sup>

### ❖ AVX = Advanced Vector Extensions

- AVX (128-bit)
- AVX2 (256-bit)
- AVX-512

# x86 Register Chart

ZMM0

YMM0

XMM0

ZMM1

YMM1

XMM1

ZMM2

YMM2

XMM2

ZMM3

YMM3

XMM3

ZMM4

YMM4

XMM4

ZMM5

YMM5

XMM5

ZMM6

YMM6

XMM6

ZMM7

YMM7

XMM7

ZMM8

YMM8

XMM8

ZMM9

YMM9

XMM9

ZMM10

YMM10

XMM10

ZMM11

YMM11

XMM11

ZMM12

YMM12

XMM12

ZMM13

YMM13

XMM13

ZMM14

YMM14

XMM14

ZMM15

YMM15

XMM15

ZMM16

ZMM17

ZMM18

ZMM19

ZMM20

ZMM21

ZMM22

ZMM23

ZMM24

ZMM25

ZMM26

ZMM27

ZMM28

ZMM29

ZMM30

ZMM31

ST(0)

MM0

ST(1)

MM1

ST(2)

MM2

ST(3)

MM3

ST(4)

MM4

ST(5)

MM5

ST(6)

MM6

ST(7)

MM7

CW

FP\_IP

FP\_DP

FP\_CS

SW

TW

FP\_DS

FP\_OPC

FP\_DP

FP\_IP

AL

AH

AX

EAX

RAX

BL

BH

BX

EBX

RBX

CL

CH

CX

ECX

RCX

DL

DH

DX

EDX

RDX

BPL

BP

EBP

RBP

SIL

SI

ESI

RSI

SPL

SP

ESP

RSP

R8B

R8W

R8D

R8

R12B

R12W

R12D

R12

R9B

R9W

R9D

R9

R13B

R13W

R13D

R13

R10B

R10W

R10D

R10

R14B

R14W

R14D

R14

R11B

R11W

R11D

R11

R15B

R15W

R15D

R15

DIL

DI

EDI

RDI

IP

EIP

RIP

MSW

CR0

CR4

CR1

CR5

CR2

CR6

CR3

CR7

MXCSR

CR8

CR9

CR10

CR11

CR12

CR13

CR14

CR15

DR0

DR6

DR1

DR7

DR2

DR8

DR3

DR9

DR4

DR10

DR12

DR14

DR5

DR11

DR13

DR15

FLAOS

EFLAOS

RFLAGS

8-bit register

32-bit register

80-bit register

256-bit register

16-bit register

64-bit register

128-bit register

512-bit register

</

- ❖ 8/16/32/64 bit basic registers
- ❖ 128/256/512 bit MMX extended registers



# Parallel Data: AVX

AVX

## New instructions [\[ edit \]](#)

These AVX instructions are in addition to the ones that are 256-bit extensions of the legacy 128-bit SSE instructions; most are usable on both 128-bit and 256-bit operands.

Instruction	Description
<code>VROADCASTSS</code> , <code>VROADCASTSD</code> , <code>VROADCASTF128</code>	Copy a 32-bit, 64-bit or 128-bit memory operand to all elements of a XMM or YMM vector register.
<code>VINSERTF128</code>	Replaces either the lower half or the upper half of a 256-bit YMM register with the value of a 128-bit source operand. The other half of the destination is unchanged.
<code>VEXTRACTF128</code>	Extracts either the lower half or the upper half of a 256-bit YMM register and copies the value to a 128-bit destination operand.
<code>VMASKMOVPS</code> , <code>VMASKMOVPD</code>	Conditionally reads any number of elements from a SIMD vector memory operand into a destination register, leaving the remaining vector elements unread and setting the corresponding elements in the destination register to zero. Alternatively, conditionally writes any number of elements from a SIMD vector register operand to a vector memory operand, leaving the remaining elements of the memory operand unchanged. On the AMD Jaguar processor architecture, this instruction with a memory source operand takes more than 300 clock cycles when the mask is zero, in which case the instruction should do nothing. This appears to be a design flaw. <sup>[8]</sup>
<code>VPERMILPS</code> , <code>VPERMILPD</code>	Permute In-Lane. Shuffle the 32-bit or 64-bit vector elements of one input operand. These are in-lane 256-bit instructions, meaning that they operate on all 256 bits with two separate 128-bit shuffles, so they can not shuffle across the 128-bit lanes. <sup>[9]</sup>
<code>VPERM2F128</code>	Shuffle the four 128-bit vector elements of two 256-bit source operands into a 256-bit destination operand, with an immediate constant as selector.
<code>VZEROALL</code>	Set all YMM registers to zero and tag them as unused. Used when switching between 128-bit use and 256-bit use.
<code>VZERoupper</code>	Set the upper half of all YMM registers to zero. Used when switching between 128-bit use and 256-bit use.

# Parallel Data: AVX

AVX

COMP222

**CPUs with AVX** [ [edit](#) ]

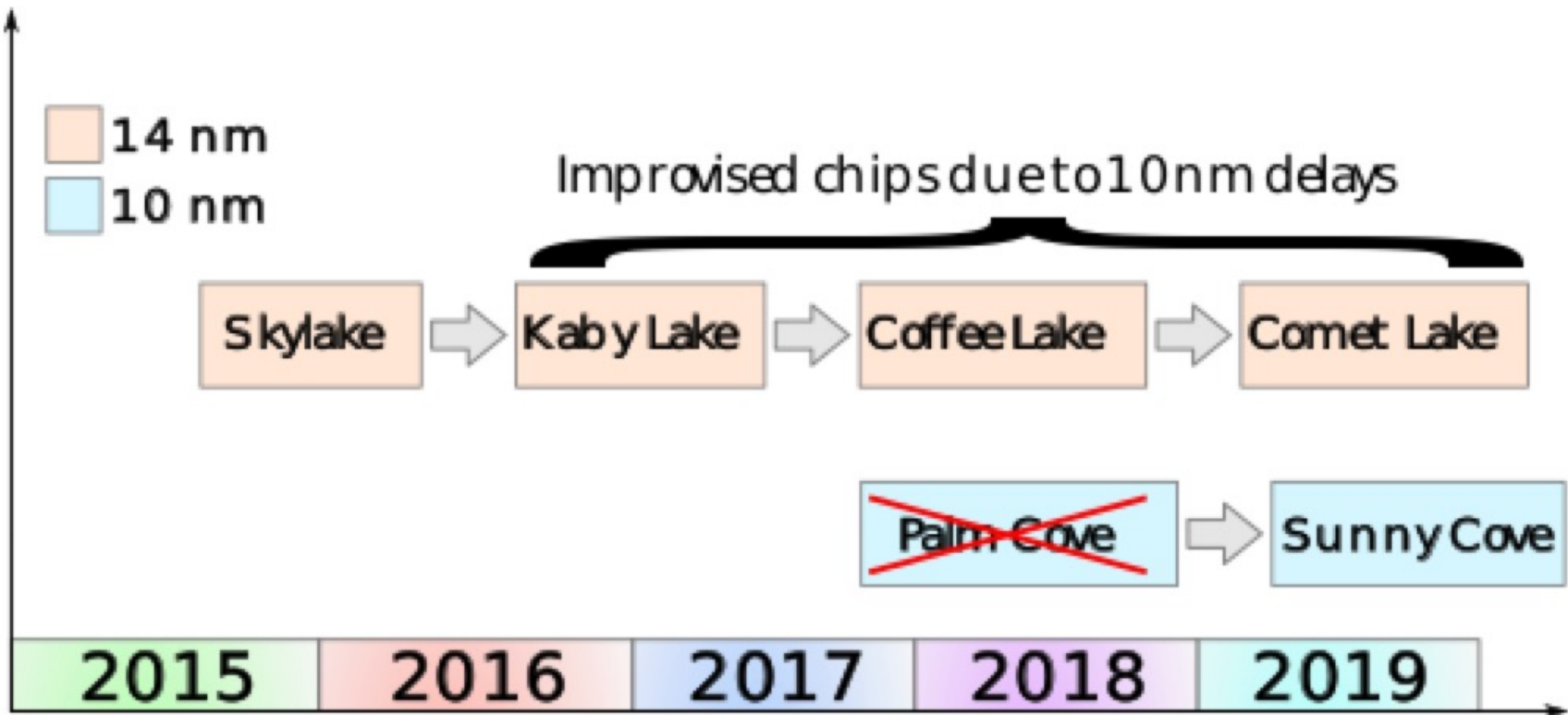
- [Intel](#)

## Intel

- [Sandy Bridge](#) processors, Q1 2011<sup>[10]</sup>
- [Sandy Bridge E](#) processors, Q4 2011<sup>[11]</sup>
- [Ivy Bridge](#) processors, Q1 2012
- [Ivy Bridge E](#) processors, Q3 2013
- [Haswell](#) processors, Q2 2013
- [Haswell E](#) processors, Q3 2014
- [Broadwell](#) processors, Q4 2014
- [Skylake](#) processors, Q3 2015
- [Broadwell E](#) processors, Q2 2016
- [Kaby Lake](#) processors, Q3 2016(ULV mobile)/Q1 2017(desktop/mobile)
- [Skylake-X](#) processors, Q2 2017
- [Coffee Lake](#) processors, Q4 2017
- [Cannon Lake](#) processors, Q2 2018
- [Whiskey Lake](#) processors, Q3 2018
- [Cascade Lake](#) processors, Q4 2018
- [Ice Lake](#) processors, Q3 2019
- [Comet Lake](#) processor (only Core branded), Q3 2019
- [Tiger Lake](#) processor, 2020

Not all CPUs from the listed families support AVX. Generally, CPUs with the commercial denomination "Core i3/i5/i7" support them, whereas "Pentium" and "Celeron" CPUs don't.

# Intel "Lake" Models



# Parallel Data: AVX

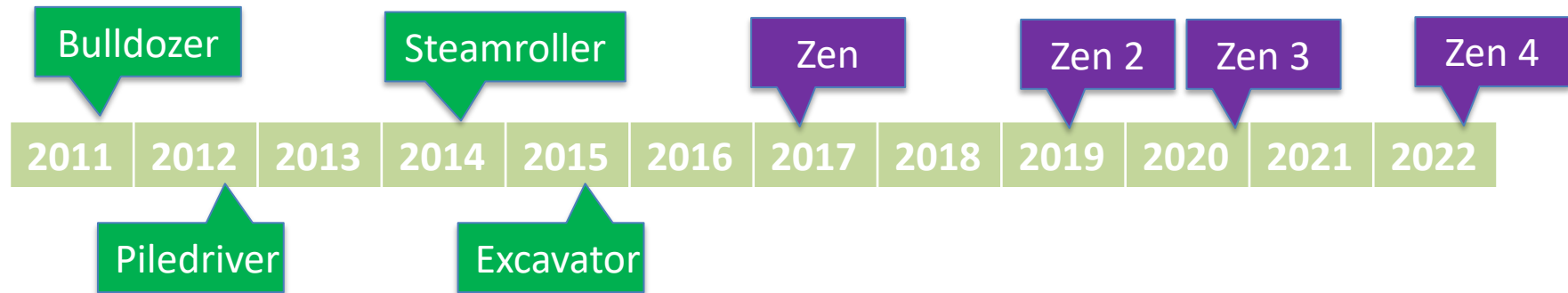
AVX

## AMD

- **AMD:**
  - **Jaguar-based** processors and newer
  - **Puma-based** processors and newer
  - "Heavy Equipment" processors
    - **Bulldozer-based** processors, Q4 2011<sup>[12]</sup>
    - **Piledriver-based** processors, Q4 2012<sup>[13]</sup>
    - **Steamroller-based** processors, Q1 2014
    - **Excavator-based** processors and newer, 2015
  - **Zen-based** processors, Q1 2017
  - **Zen+-based** processors, Q2 2018
  - **Zen 2-based** processors, Q3 2019
  - **Zen 3** processors, 2020

Zen

# AMD uArch Timeline



Advanced Vector Extensions 2 (AVX2), also known as **Haswell New Instructions**,<sup>[5]</sup> is an expansion of the AVX instruction set introduced in Intel's **Haswell microarchitecture**. AVX2 makes the following additions:

- expansion of most vector integer SSE and AVX instructions to 256 bits
- three-operand general-purpose bit manipulation and multiply
- **Gather** support, enabling vector elements to be loaded from non-contiguous memory locations
- DWORD- and QWORD-granularity any-to-any permutes
- vector shifts.

Sometimes another extension using a different cpuid flag is considered part of AVX2; those instructions are listed on their own page and not below:

- three-operand **fused multiply-accumulate** support (FMA3)

## New instructions [\[ edit \]](#)

Instruction	Description
VROADCASTSS , VROADCASTSD	Copy a 32-bit or 64-bit register operand to all elements of a XMM or YMM vector register. These are register versions of the same instructions in AVX1. There is no 128-bit version however, but the same effect can be simply achieved using VINSERTF128.
VPBROADCASTB , VPBROADCASTW , VPBROADCASTD , VPBROADCASTQ	Copy an 8, 16, 32 or 64-bit integer register or memory operand to all elements of a XMM or YMM vector register.
VROADCASTI128	Copy a 128-bit memory operand to all elements of a YMM vector register.
VINSERTI128	Replaces either the lower half or the upper half of a 256-bit YMM register with the value of a 128-bit source operand. The other half of the destination is unchanged.
VEEXTRACTI128	Extracts either the lower half or the upper half of a 256-bit YMM register and copies the value to a 128-bit destination operand.
VGATHERDPD , VGATHERQPD , VGATHERDPS .	<b>Gathers</b> single or double precision floating point values using either 32 or 64-bit indices and scale.



# Parallel Data: AVX

## AVX2

### CPUs with AVX2 [\[ edit \]](#)

- **Intel**
  - **Haswell** processor (only Core branded), Q2 2013
  - **Haswell E** processor (only Core branded), Q3 2014
  - **Broadwell** processor (only Core branded), Q4 2014
  - **Broadwell E** processor (only Core branded), Q3 2016
  - **Skylake** processor (only Core branded), Q3 2015
  - **Kaby Lake** processor (only Core branded), Q3 2016(ULV mobile)/Q1 2017(desktop/mobile)
  - **Skylake-X** processor (only Core branded), Q2 2017
  - **Coffee Lake** processor (only Core branded), Q4 2017
  - **Cannon Lake** processor, Q2 2018
  - **Cascade Lake** processor, Q2 2019
  - **Ice Lake** processor, Q3 2019
  - **Comet Lake** processor (only Core branded), Q3 2019
  - **Tiger Lake** processor, 2020
- **AMD**
  - **Excavator** processor and newer, Q2 2015
  - **Zen** processor, Q1 2017
  - **Zen+** processor, Q2 2018
  - **Zen 2** processor, Q3 2019
  - **Zen 3** processor, 2020
- **VIA:**
  - Nano QuadCore
  - Eden X4

## AVX-512

AVX-512 are 512-bit extensions to the 256-bit Advanced Vector Extensions SIMD instructions for x86 instruction set architecture proposed by [Intel](#) in July 2013, and are supported with Intel's [Knights Landing](#) processor.<sup>[3]</sup>

# Parallel Data: AVX

## AVX512

AVX-512 are 512-bit extensions to the 256-bit Advanced Vector Extensions SIMD instructions for x86 instruction set architecture proposed by Intel in July 2013, and are supported with Intel's [Knights Landing](#) processor.<sup>[3]</sup>

AVX-512 instruction are encoded with the new [EVEX prefix](#). It allows 4 operands, 7 new 64-bit opmask registers, scalar memory mode with automatic broadcast, explicit rounding control, and compressed displacement memory [addressing mode](#). The width of the register file is increased to 512 bits and total register count increased to 32 (registers ZMM0-ZMM31) in x86-64 mode.

AVX-512 consists of multiple extensions not all meant to be supported by all processors implementing them. The instruction set consists of the following:

- AVX-512 Foundation – adds several new instructions and expands most 32-bit and 64-bit floating point SSE-SSE4.1 and AVX/AVX2 instructions with EVEX coding scheme to support the 512-bit registers, operation masks, parameter broadcasting, and embedded rounding and exception control
- AVX-512 Conflict Detection Instructions (CD) – efficient conflict detection to allow more loops to be vectorized, supported by Knights Landing<sup>[3]</sup>
- AVX-512 Exponential and Reciprocal Instructions (ER) – exponential and reciprocal operations designed to help implement transcendental operations, supported by Knights Landing<sup>[3]</sup>
- AVX-512 Prefetch Instructions (PF) – new prefetch capabilities, supported by Knights Landing<sup>[3]</sup>
- AVX-512 Vector Length Extensions (VL) – extends most AVX-512 operations to also operate on XMM (128-bit) and YMM (256-bit) registers (including XMM16-XMM31 and YMM16-YMM31 in x86-64 mode)<sup>[22]</sup>
- AVX-512 Byte and Word Instructions (BW) – extends AVX-512 to cover 8-bit and 16-bit integer operations<sup>[22]</sup>
- AVX-512 Doubleword and Quadword Instructions (DQ) – enhanced 32-bit and 64-bit integer operations<sup>[22]</sup>
- AVX-512 Integer [Fused Multiply Add](#) (IFMA) – fused multiply add for 512-bit integers.<sup>[23]:746</sup>
- AVX-512 Vector Byte Manipulation Instructions (VBMI) adds vector byte permutation instructions which are not present in AVX-512BW.
- AVX-512 Vector Neural Network Instructions Word variable precision (4VNNIW) – vector instructions for deep learning.
- AVX-512 Fused Multiply Accumulation Packed Single precision (4FMAPS) – vector instructions for deep learning.
- VPOPCNTDQ – count of bits set to 1.<sup>[24]</sup>
- VPCLMULQDQ – carry-less multiplication of quadwords.<sup>[24]</sup>
- *AVX-512 Vector Neural Network Instructions (VNNI)* – vector instructions for deep learning.<sup>[24]</sup>
- *AVX-512 Galois field New Instructions (GFNI)* – vector instructions for calculating [Galois field](#).<sup>[24]</sup>
- *AVX-512 Vector AES instructions (VAES)* – vector instructions for [AES](#) coding.<sup>[24]</sup>
- *AVX-512 Vector Byte Manipulation Instructions 2 (VBMI2)* – byte/word load, store and concatenation with shift.<sup>[24]</sup>
- *AVX-512 Bit Algorithms (BITALG)* – byte/word [bit manipulation](#) instructions expanding VPOPCNTDQ.<sup>[24]</sup>

# Parallel Data: AVX

COMP222

CPU with AVX-512 [\[ edit \]](#)

AVX512

AVX-512 Subset	F	CD	ER	PF	4FMAPS	4VNNIW	VL	DQ	BW	IFMA	VBMI	VBMI2	VPOPCNTDQ	BITALG	VNNI	VPCLMULQDQ	GFNI	VAES				
Intel <b>Knights Landing</b> (2016)	Yes		Yes		No																	
Intel <b>Knights Mill</b> (2017)					Yes	No				Yes	No											
Intel <b>Skylake-SP, Skylake-X</b> (2017)			No				Yes				No											
Intel <b>Cannon Lake</b> (2018)											No					Yes	No					
Intel <b>Cascade Lake-SP</b> (2019)											No				Yes		No					
Intel <b>Ice Lake</b> (2019)											Yes											

[\[25\]](#)

As of 2020, there are no AMD CPUs that support AVX-512, and AMD has not yet released plans to support AVX-512.

## Compilers supporting AVX-512 [\[ edit \]](#)

- [GCC](#) 4.9 and newer<sup>[26]</sup>
- [Clang](#) 3.9 and newer<sup>[27]</sup>
- [ICC](#) 15.0.1 and newer<sup>[28]</sup>
- Microsoft Visual Studio 2017 C++ Compiler<sup>[29]</sup>
- Java 9<sup>[30]</sup>
- Go 1.11<sup>[31]</sup>
- [Julia](#)<sup>[32][33]</sup>

# Parallel Data: AVX

AVX

## Applications [\[ edit \]](#)

- Suitable for [floating point](#)-intensive calculations in multimedia, scientific and financial applications (AVX2 adds support for [integer](#) operations).
- Increases parallelism and throughput in floating point [SIMD](#) calculations.
- Reduces register load due to the non-destructive instructions.
- Improves Linux RAID software performance (required AVX2, AVX is not sufficient)<sup>[34]</sup>

## Software [\[ edit \]](#)

- [Blender](#) uses AVX2 in the render engine cycles.
- [Botan](#) uses both AVX and AVX2 when available to accelerate some algorithms, like ChaCha.
- [Crypto++](#) uses both AVX and AVX2 when available to accelerate some algorithms, like Salsa and ChaCha.
- [OpenSSL](#) uses AVX- and AVX2-optimized cryptographic functions since version 1.0.2.<sup>[35]</sup> This support is also present in various clones and forks, like LibreSSL.
- [Prime95/MPrime](#), the software used for [GIMPS](#), started using the AVX instructions since version 27.x.
- dav1d [AV1](#) decoder can use AVX2 on supported CPUs.<sup>[36]</sup>
- [dnetc](#), the software used by [distributed.net](#), has an AVX2 core available for its RC5 project and will soon release one for its OGR-28 project.
- [Einstein@Home](#) uses AVX in some of their distributed applications that search for [gravitational waves](#).<sup>[37]</sup>
- [Folding@home](#) uses AVX on calculation cores implemented with [GROMACS](#) library.
- [Horizon: Zero Dawn](#) Uses AVX1 in the Decima (game engine) and is the engine the game uses.
- [RPCS3](#), an open source [PlayStation 3](#) emulator, uses AVX2 and [AVX-512](#) instructions to emulate PS3 games.
- [Network Device Interface](#), an IP video/audio protocol developed by NewTek for live broadcast production, uses AVX and AVX2 for increased performance.
- [TensorFlow](#) since version 1.6 and tensorflow above versions requires CPU supporting at least AVX.<sup>[38]</sup>
- [Xenia](#) requires AVX instruction set in order to run.



By: **Linus Torvalds** (torvalds.delete@this.linux-foundation.org), July 11, 2020 1

I hope AVX512 dies a painful death, and that Intel starts fixing real problems instead of trying to create magic instructions to then create benchmarks that they can look good on.

I've said this before, and I'll say it again: in the heyday of x86, when Intel was laughing all the way to the bank and killing all their competition absolutely *everybody* else did better than Intel on FP loads. Intel's FP performance sucked (relatively speaking), and it matter not one iota.

And AVX512 has real downsides. I'd *much* rather see that transistor budget used on other things that are much more relevant. Even if it's still FP math (in the GPU, rather than AVX512). Or just give me more cores (with good single-thread performance, but without the garbage like AVX512) like AMD did.

No honestly, I'm not a huge fan of AVX2 either. But then. wasn't a huge fan of MMX or the original AVX. And no, before you ask, it's not like I hold up the original i387 FPU as some shining example either ;)

MMX/SSE was a first-attempt (plus fixes). The i387 was a particularly nasty thing to be compatible with anyway, it's entirely understandable why it was done the way it was done. In hindsight, maybe it could have been done better, but a "in hindsight" argument is always complete BS. So that's not a valid argument. MMX/SSE was fine.

AVX/AVX2 were reasonable cleanups and honestly, I don't think 256 bits is a huge pain even as a baseline. And Intel has been good about keeping AVX always there. Afaik, new CPU's really have gotten AVX reliably. So it hasn't been a fragmentation issue, and while I *think* it has the same state dirtying issue ("helper function using MMX instructions and saves/restores the instructions it modifies will be clearing upper bits in AVX registers and trashing state"), I think it was a fairly reasonable extension.

# Intel AVX/AVX2

## 12th Generation Intel® Core™ Processors

### Intel® Advanced Vector Extensions 2 (Intel® AVX2)

Intel® Advanced Vector Extensions 2.0 (Intel® AVX2) is the latest expansion of the Intel instruction set. Intel® AVX2 extends the Intel® Advanced Vector Extensions (Intel® AVX) with 256-bit integer instructions, floating-point fused multiply-add (FMA) instructions, and gather operations. The 256-bit integer vectors benefit math, codec, image, and digital signal processing software. FMA improves performance in face detection, professional imaging, and high-performance computing. Gather operations increase vectorization opportunities for many applications. In addition to the vector extensions, this generation of Intel processors adds new bit manipulation instructions useful in compression, encryption, and general purpose software. For more information on Intel® AVX, refer to <http://www.intel.com/software/avx>

Intel® Advanced Vector Extensions (Intel® AVX) are designed to achieve higher throughput to certain integer and floating point operation. Due to varying processor power characteristics, utilizing AVX instructions may cause a) parts to operate below the base frequency b) some parts with Intel® Turbo Boost Technology 2.0 to not achieve any or maximum turbo frequencies. Performance varies depending on hardware, software and system configuration and you should consult your system manufacturer for more information.

Intel® Advanced Vector Extensions refers to Intel® AVX or Intel® AVX2 .

**Note:** Intel® AVX and AVX2 Technologies may not be available on all SKUs.



## Intel<sup>®</sup>AVX512-FP16

### Architecture Specification

June 2021

Revision 1.0

#### CHAPTER 3. INSTRUCTION TABLE

IVB	Intel <sup>®</sup> Xeon <sup>®</sup> processors based on Ivy Bridge microarchitecture
KNL	Intel <sup>®</sup> Xeon Phi <sup>™</sup> Processor 3200, 5200, 7200 Series based on Knights Landing microarchitecture
SKX	Intel <sup>®</sup> Xeon <sup>®</sup> Processor Scalable Family based on Skylake microarchitecture
SPR	Future Intel <sup>®</sup> Xeon <sup>®</sup> processors based on Sapphire Rapids microarchitecture

Figure 3.1: Microarchitecture abbreviations used in instruction table

# intel<sup>®</sup>AVX/AVX2

## 12th Generation Intel<sup>®</sup> Core™ Processors

### Datasheet

### 5 INSTRUCTIONS

5.1	VADDPH . . . . .	21
5.2	VADDSH . . . . .	24
5.3	VCMPPH . . . . .	26
5.4	VCMPSH . . . . .	29
5.5	VCOMISH . . . . .	32
5.6	VCVTDQ2PH . . . . .	34
5.7	VCVTPD2PH . . . . .	36
5.8	VCVTPH2DQ . . . . .	38



5.72	VSCALEFPH . . . . .	195
5.73	VSCALEFSH . . . . .	198
5.74	VSQRTPH . . . . .	200
5.75	VSQRTPH . . . . .	202
5.76	VSUBPH . . . . .	203
5.77	VSUBSH . . . . .	205
5.78	VUCOMISH . . . . .	207

### 5.1.3 Operation

Src2 = **register**

```

1  VADDPH (EVEX encoded versions) when src2 operand is a register
2  VL = 128, 256 or 512
3  KL := VL/16
4  IF (VL = 512) AND (EVEX.b = 1):
5      SET_RM(EVEX.RC)
6  ELSE
7      SET_RM(MXCSR.RC)
8
9  FOR j := 0 TO KL-1:
10     IF k1[j] OR *no writemask*:
11         DEST.fp16[j] := SRC1.fp16[j] + SRC2.fp16[j]
12     ELSE IF *zeroing*:
13         DEST.fp16[j] := 0
14     // else dest.fp16[j] remains unchanged
15
16  DEST[MAXVL-1:VL] := 0

```

## 5.1. VADDPH

## CHAPTER 5. INSTRUCTIONS

Src2 = **memory**

```
1  VADDPH (EVEX encoded versions) when src2 operand is a memory source
2  VL = 128, 256 or 512
3  KL := VL/16
4
5  FOR j := 0 TO KL-1:
6      IF k1[j] OR *no writemask*:
7          IF EVEX.b = 1:
8              DEST.fp16[j] := SRC1.fp16[j] + SRC2.fp16[0]
9          ELSE:
10             DEST.fp16[j] := SRC1.fp16[j] + SRC2.fp16[j]
11     ELSE IF *zeroing*:
12         DEST.fp16[j] := 0
13     // else dest.fp16[j] remains unchanged
14
15  DEST[MAXVL-1:VL] := 0
```

Quora



Search Quora



**Yowan Rajcoomar**, Computer Technician (2008-present)



Answered 18h ago

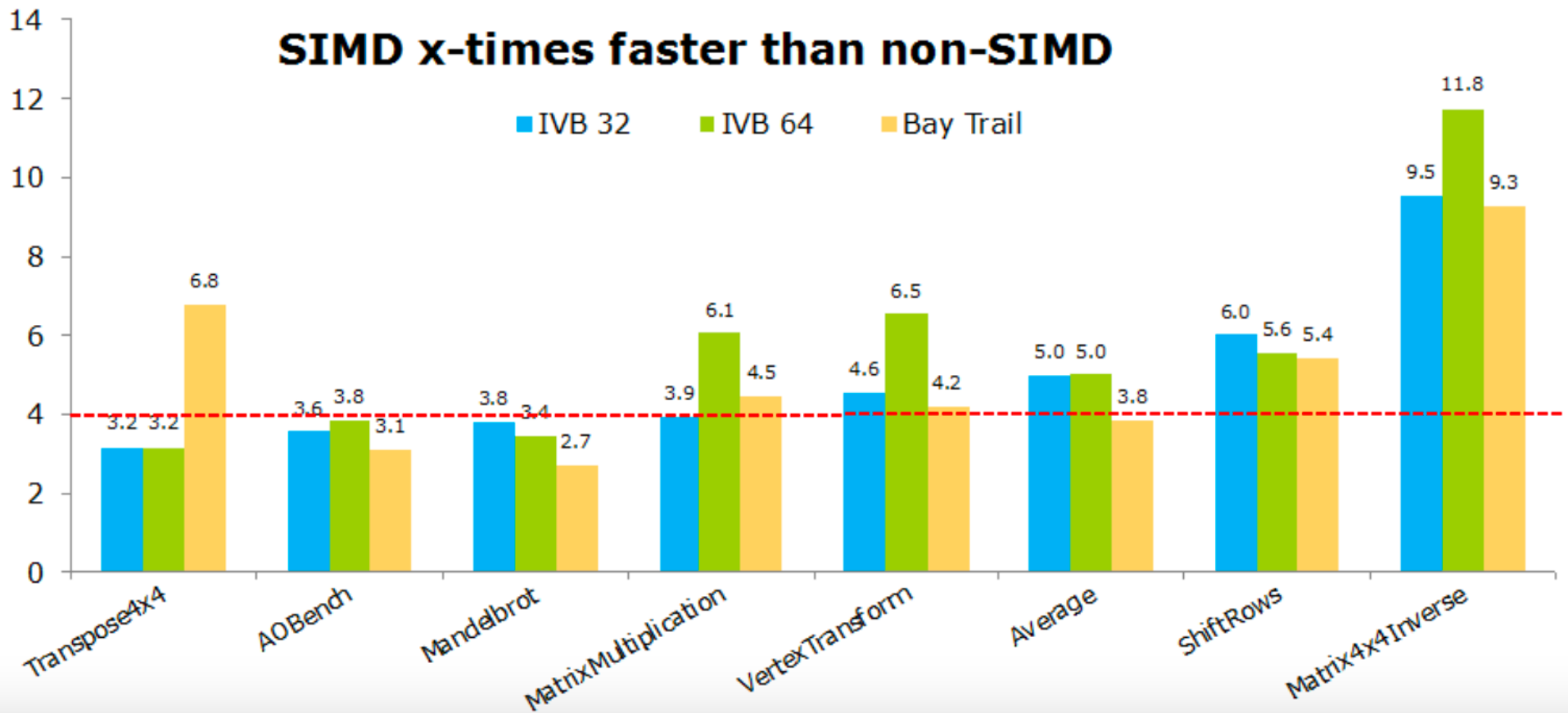
There is no current use cases which requires general purposes registers larger than 64-bits.

The current trend is the extension of **vector and SIMD capabilities** and the addition of **domain-specific accelerators** which will help demanding applications such as CAD, rendering, scientific workloads, machine/deep learning and AI. There, a lot of data needs to be crunched at the same time so companies such as Intel have introduced the following:

- AVX-512F which bumps the number of FP/SIMD registers from 16 to 32 while providing additional features (at the cost of die space and thermals)
- AVX-512 VNNI (Vector Neural Instructions) for handling 8 and 16-bit values in convolutional neural networks. This was made available in Cascade and Ice Lake-based chips.
- AVX-512 BF16 which provides a speedup when dealing with dot products on bfloat16 pairs.
- x86 AMX or Advanced Matrix Extension. This is literally an accelerator built into the x86 core with its own register file and instructions. This will debut sometime later this year with Intel's next HEDT lineup.

ARM also went through similar changes with the addition of Scalable Vector Extensions (SVE and SVE2) which is flexible in terms of width, ranging from 128 to 2048-bits. The RISC-V spec also provisions a 128-bit mode but no actual hardware implements it because it would be a waste of resources, die space and would add

# SIMD Benchmark





# Section

## AMX Mat Mult

# Mat Mult Example

P&H

 Present

 Note

## 1.10 Going faster: Matrix multiply in Python

To demonstrate the impact of the ideas in this book, every chapter has a "Going Faster" section that improves the performance of a program that multiplies a matrix times a vector. We start with this Python program:

```
for i in xrange(n):
    for j in xrange(n):
        for k in xrange(n):
            C[i][j] += A[i][k] * B[k][j]
```

We are using the n1-standard-96 server in Google Cloud Engine, which has two Intel Skylake Xeon chips, and each chip has 24 processors or cores and running Python version 3.1. If the matrices are 960x960, it takes about 5 minutes to run using Python 2.7. Since floating point computations go up with the cube of the matrix dimension, it would take almost 6 hours to run if the matrices were 4096x4096. While it's quick to write the matrix multiply in Python, who wants to wait that long to get the answer?

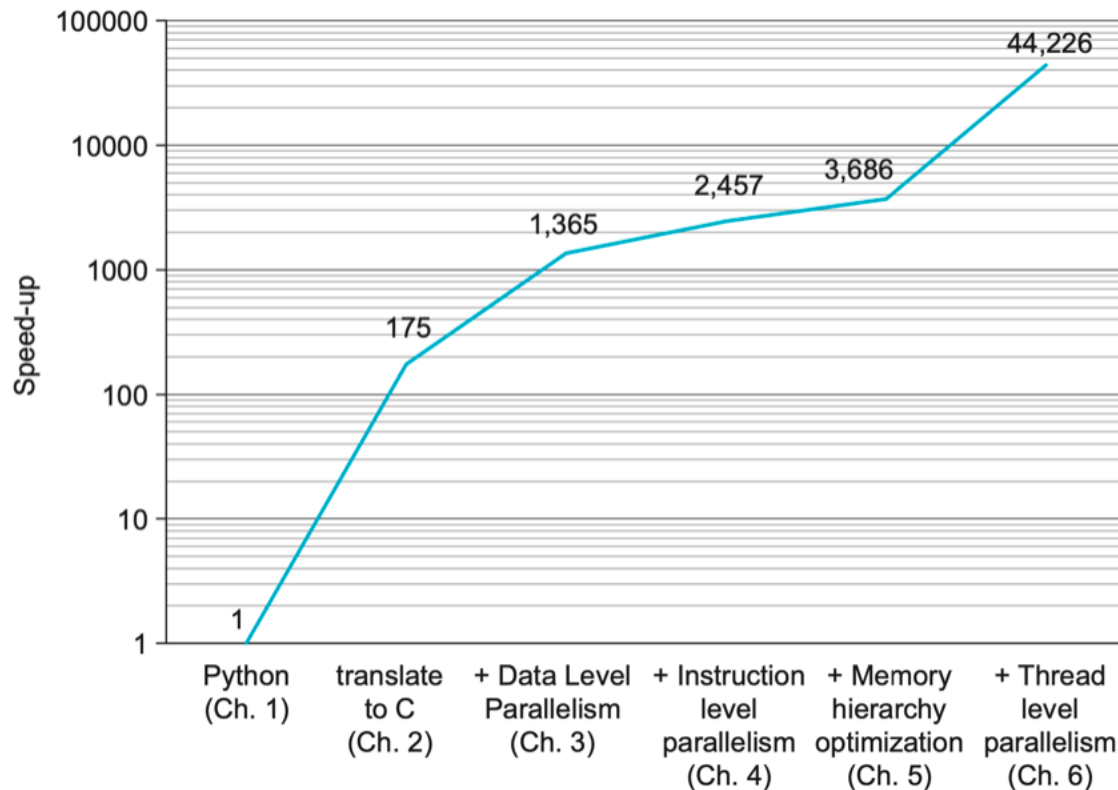
In COD Chapter 2 (Instructions: Language of the Computer), we are converting the Python version of matrix multiply to a C version increases performance by a factor of 200. The C programming abstraction is much closer to the hardware than Python, which is why we use it as the programming example in this book. Closing the abstraction gap also makes it much faster than Python [Leiserson, 2020].

- In the category of data level parallelism, in COD Chapter 3 (Arithmetic for Computers) we use subword parallelism via C intrinsics to increase performance by a factor of about 8.
- In the category of instruction level parallelism, in COD Chapter 4 (The Processor) we use loop unrolling to exploit multiple instruction issue and out-of-order execution hardware to increase performance by another factor of about 2.
- In the category of memory hierarchy optimization, in COD Chapter 5 (Large and Fast: Exploiting Memory Hierarchy) we use cache blocking to increase performance on large matrices by another factor of about 1.5.
- In the category of thread level parallelism, in COD Chapter 6 (Parallel Processor from Client to Cloud) we use parallel for loops in OpenMP to exploit multicore hardware to increase performance by another factor of 12 to 17.

# Mat Mult Example

P&H

Figure 1.10.1: Optimizations of matrix multiply program in Python in the chapters of this book. (COD Figure 1.20).



# Apple M1 MatMult

It's amazing to me that there are four separate pieces of hardware in M1 devices that can do **matrix multiplies**. In addition to running on the **CPU**, M1 Max devices have three separate kinds of hardware-accelerated `gemm`: the **GPU**, the **ANE** (Apple Neural Engine), and this special **matrix coprocessor**.

Since there is no summary, these are the benchmark findings:

AMX co-processor 2 TFLOPS FP32  
GPU 8 TFLOPS FP32  
Neural Engine 5.5 TFLOPS FP16

# Apple M1 MatMult



Timothy Liu's Blog

## Benchmarking the Apple M1 Max

Understanding the Hardware Capabilities of  
Apple's flagship SOC

### CPU



Timothy Liu · Nov 14, 2021 · 13 min read

The CPU in the M1 Max is a 10-core CPU, with 2 efficiency cores at 2.1 GHz and 8 performance cores at 3.0 GHz during all-core load. There should be no difference between the CPU performance on the M1 Max and M1 Pro, barring slightly higher memory bandwidth available to the CPU complex on the M1 Max. As a reference point for comparison purposes, I have my desktop AMD Ryzen 5600X CPU with DDR4 memory and not overclocked. The desktop is running Ubuntu 20.04.

❖ 8 **P** (3.0 GHz)

❖ 2 **E** (2.1 GHz)

➤ vs AMD Ryzen 5600X

# Apple M1 MatMult

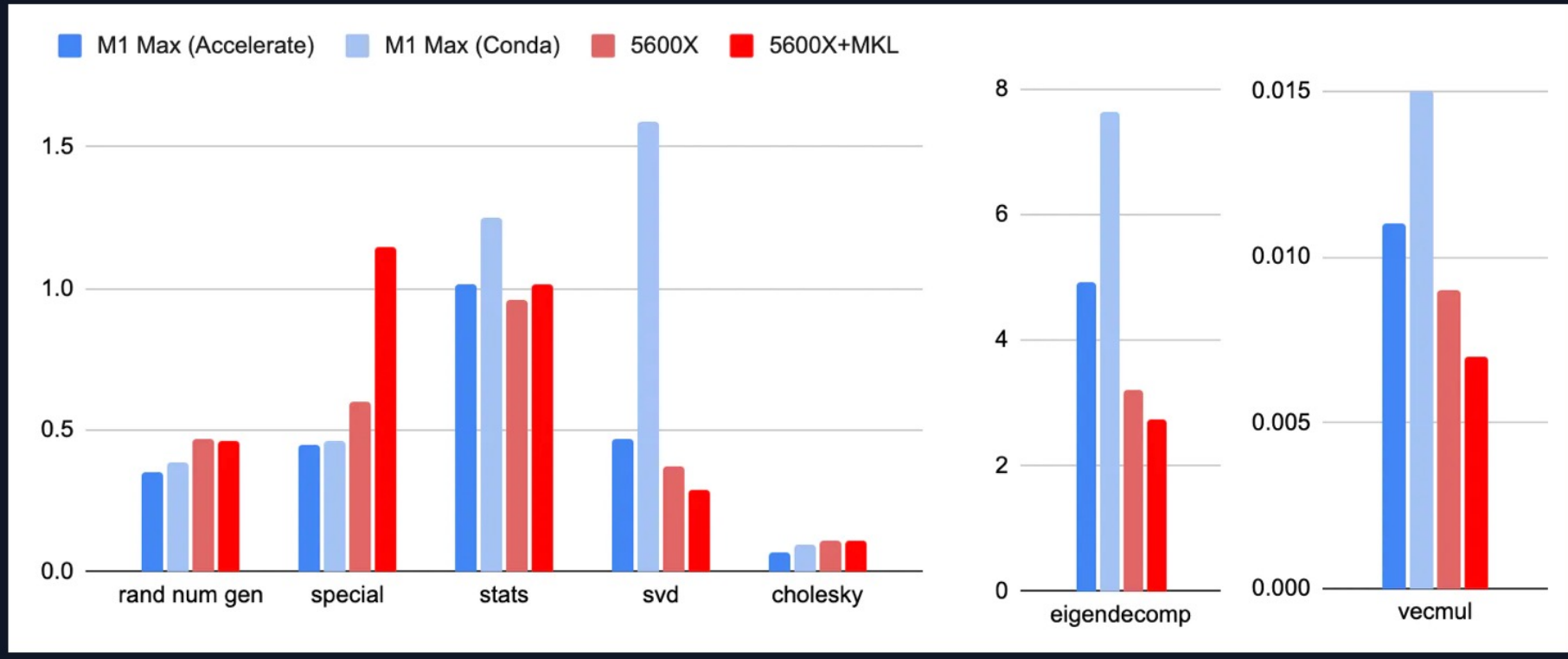
## Matrix Multiplication (GEMM) Performance

We already know that the M1 Max CPU should have really strong matrix multiplication performance due to Apple's "hidden"/undocumented AMX co-processor embedded in the CPU complex, and that it is leveraged when you use Apple's Accelerate framework. What I didn't know is that you can compile NumPy to work with Accelerate, which allows you to easily leverage the AMX instructions via normal NumPy code (NumPy installed via conda does **not** include Accelerate support, and instead uses cblas). The results are quite stunning, especially for single precision, or FP32 (which is commonly used in machine learning applications), giving us about **almost 2 TFLOPS** for large enough matrix sizes (about the level of a GTX 1050 Ti). Mind you, this is via normal Python NumPy code. Presumably, if you use Accelerate directly via a lower-level language, you can get even better performance. Compared to the 5600X, the M1 Max CPU is generally at least 2x faster any data type and any matrix size, even with MKL enabled on the 5600X.



# Apple M1 MatMult

(Execution timings for function reported, lower is better)



- M1 Max, with Accelerate, is faster on 3 tasks (RNG, special, Cholesky)
- Both are about the same on 1 task (stats)
- 5600X, with MKL is faster on 3 tasks (SVD, VecMul, eigendecomp)

# Apple AMX MatMult

COMP222 by [@bwasti](#) ([mastodon](#))

---

## How to Get 1.5 TFlops of FP32 Performance on a Single M1 CPU Core

by [@bwasti](#) ([mastodon](#))

If you're in the market for training large modern neural networks, this post won't really be relevant, since that's 100x slower than an A100 (156TFlops).

So, how on earth is 1.5 TFlops interesting?

- this is running on a single core of a battery powered 2020 MacBook Air
- this is running with a ~0.5 *nanosecond* latency per instruction

We are not in the realm of beefy accelerators or GPU tensor cores. We are talking about real-world linear algebra performance that lives **one cycle** away from CPU registers.

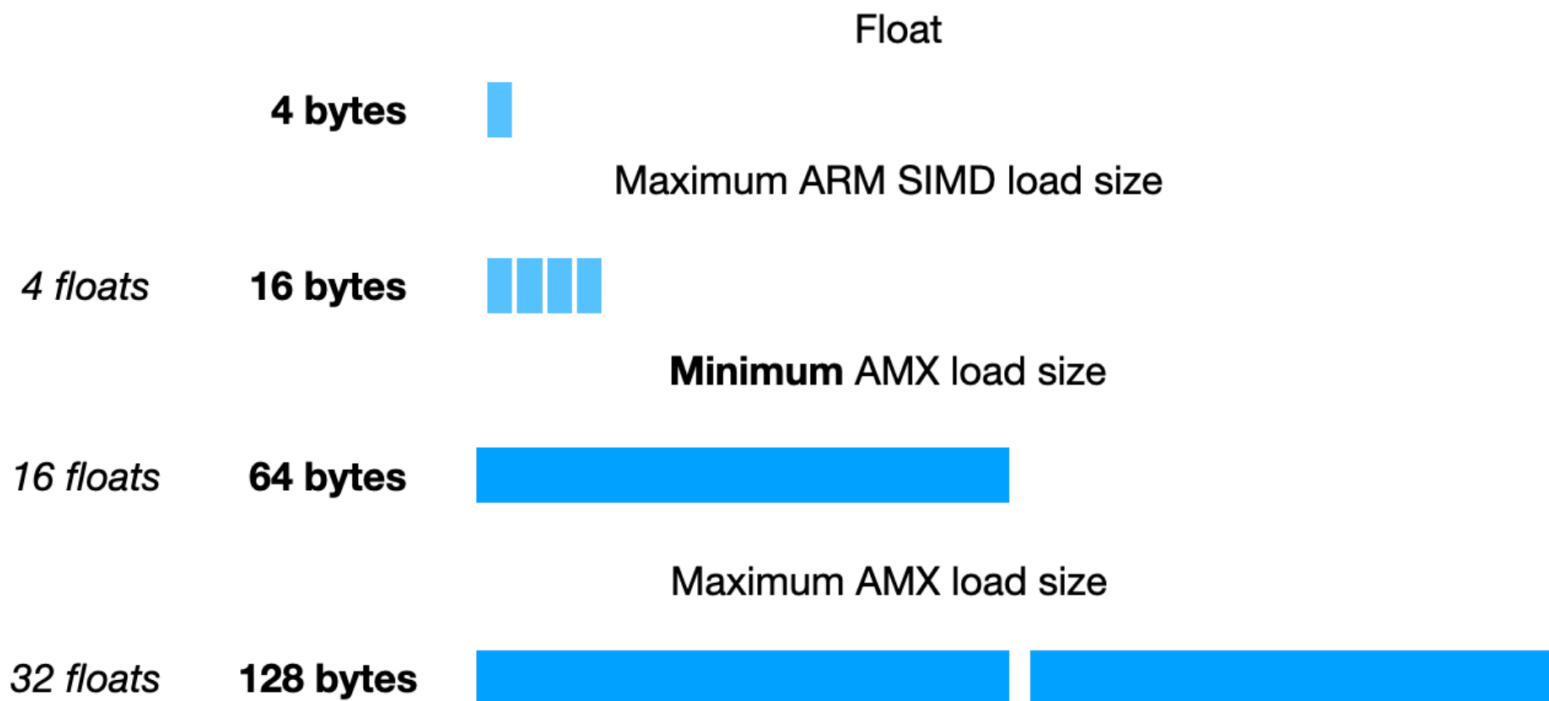
# Apple AMX MatMult

COMP222 by @bwasti (mastodon)

## What is an AMX Co-Processor?

It's basically SIMD on steroids. An important distinction is that the AMX:CPU ratio is not 1:1; not every core has its own AMX co-processor.

Here are the sizes one might use to load or store values:



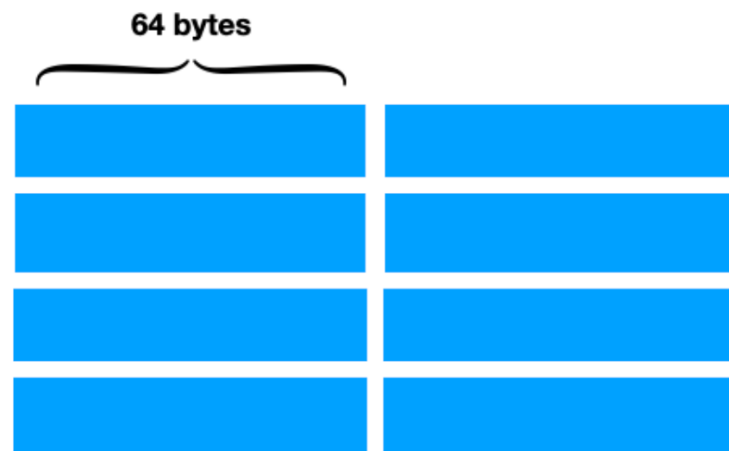
The *minimum* is as wide as a full AVX512 register.

# Apple AMX MatMult

COMP222 by @bwasti (mastodon)

The registers are segmented into groups: X, Y and Z. For every instruction, the X and Y groups hold inputs and the Z group holds outputs.

**X**  
  
128 floats    512 bytes



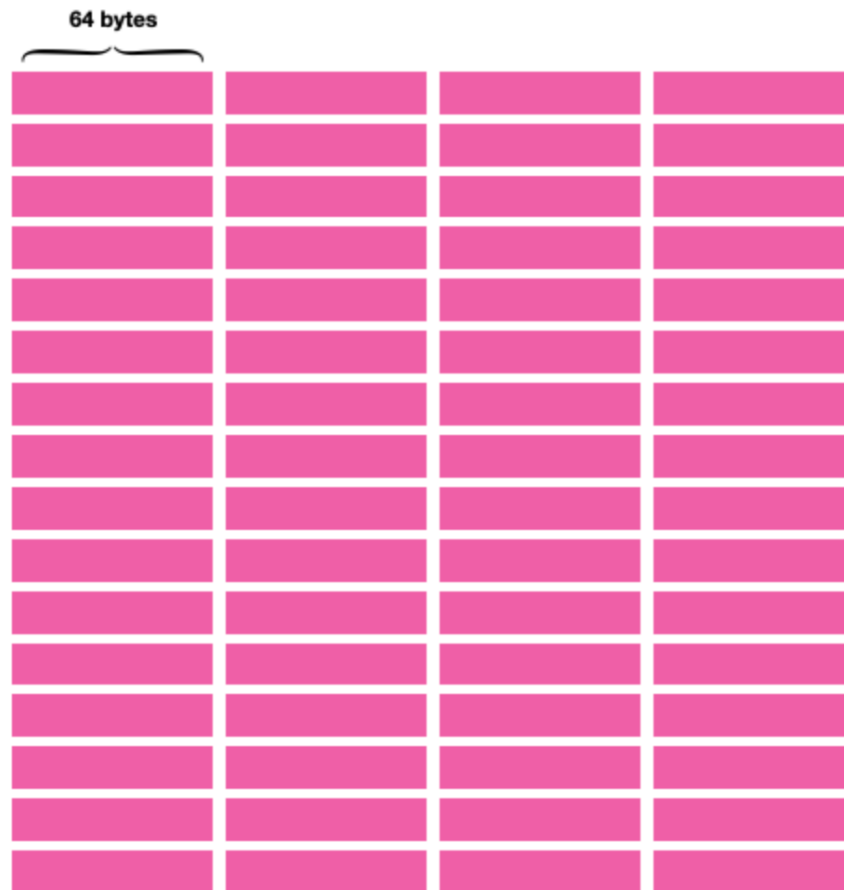
**Y**  
  
128 floats    512 bytes



# Apple AMX MatMult

COMP222 by @bwasti (mastodon)

**Z**  
**4096 bytes**  
*1024 floats*



*(Spoiler: a full 1024 bytes (1/4 of the Z registers) can be populated with a single AMX instruction.)*

# Apple AMX MatMult

COMP222 by @bwasti (mastodon)

What is an outer product? Assuming you have two input vectors  $\mathbf{u}$  and  $\mathbf{v}$ :

$$\mathbf{u} = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_m \end{bmatrix}, \quad \mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$$

The outer product is the matrix containing a product of every possible pairwise combination of their elements. (This gives some hints as to why the Z register group is so much bigger than X and Y.)

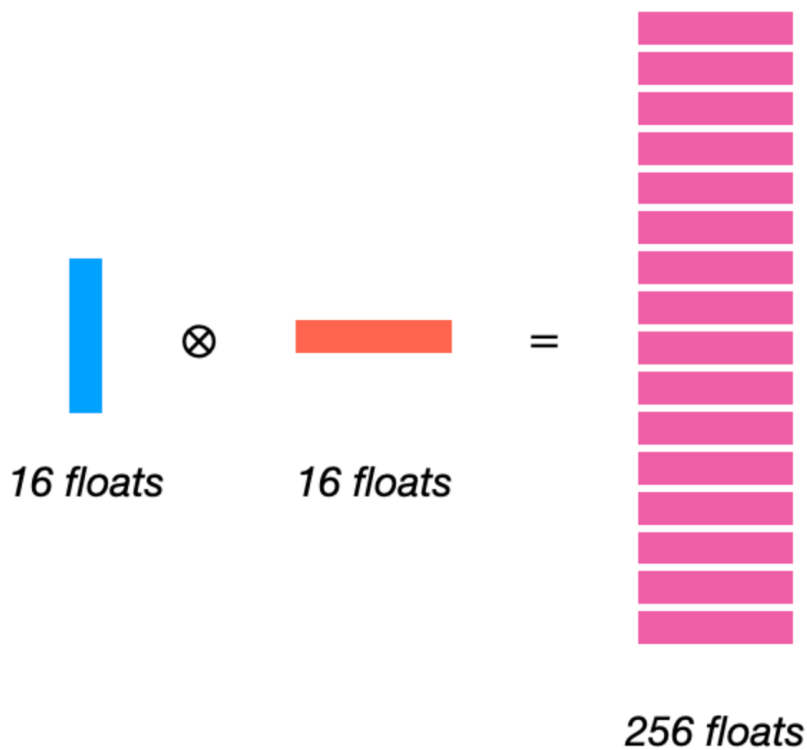
$$\mathbf{u} \otimes \mathbf{v} = \begin{bmatrix} u_1 v_1 & \dots & u_1 v_n \\ u_2 v_1 & \dots & u_2 v_n \\ \vdots & \ddots & \vdots \\ u_m v_1 & \dots & u_m v_n \end{bmatrix}$$



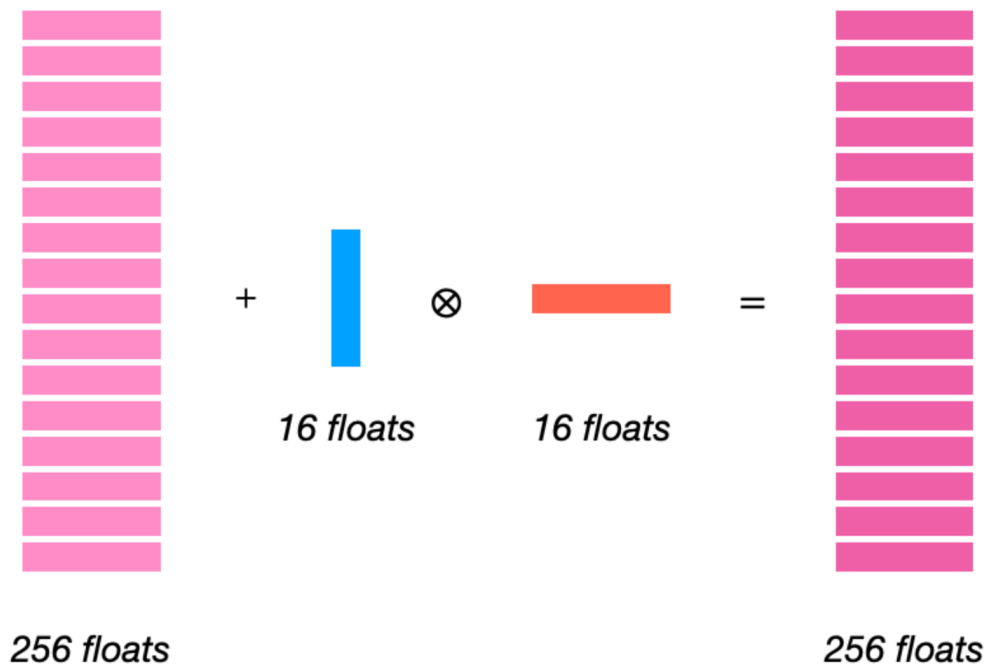
# Apple AMX MatMult

COMP222 by @bwasti (mastodon)

On the AMX chip, this boils down to a very simple instruction that looks a lot like this:



And there's a flag you can set to also make it accumulate from the previous result:



# Apple AMX MatMult

COMP222

by @bwasti (mastodon)

➤ Using **AMX** in C

And here is how we might do it in AMX:

```
// only set for k == 0
uint64_t reset_z = 1ull << 27;

for (uint32_t k = 0; k < K; ++k) {
    uint64_t idx = k % 4;
    // 64 bytes = 16 floats
    AMX_LDX((uint64_t)A + k * 64);
    AMX_LDY((uint64_t)B + k * 64);

    // now we do 4 indepedent outer products (avoidi
    AMX_FMA32(reset_z);
    reset_z = 0;
}
```

# Section

VLIW

IMC

# VLI and VLIW

## What are the pros and cons of variable length instructions?



**Jeff Drobman**, Lecturer at California State University, Northridge (2016-present)

Answered just now

I would say that VLI and VLWI (very long word instructions) have died the same death that CISC did when the smart decision was made to switch to a simpler, scalable RISC architecture that is deeply pipelined. Variations of any kind can leave holes in the instruction pipeline due to multi-cycle instruction fetching.



**Greg Harp**, former Retired, 45 Years With Mainframe IT and Z Systems

Answered 2h ago



From the mainframe perspective, there are 2, 4 and 6 byte instructions. I have seen people do 6 instructions of 2 bytes each (they just have to use the smallest instruction) when a single 6 bytes would do the same. Bottom line, is use the instruction that best meets the need and do not get all hung up about the length of the instruction. But think ahead as you may want to use an extra instruction in one place to set you up for a future use of a register someplace else.

## What is the advantage of VLIW architecture?



**Jeff Drobman**

Lecturer at California State University, Northridge (2016–present) · Just now

simple: parallel computation via multiple operations per instruction. but we now have been using a simpler architecture with SMT and superscalar that provide multi-issue instructions. so no need to load them up into a single long instruction.

## Is VLIW better than X86?



**Jeff Drobman**

Lecturer at California State University, Northridge (2016–present) · Just now

No. VLIW creates parallel operations in a single instruction. but now in modern CPU's, we have better techniques for parallelism: multi-threading, multi-core, multi-issue.

# IMC: in Mem Compute

SemiWiki.com

## Survey paper on SRAM-based In-Memory Computing Techniques

👤 sparsh · 🕒 Wednesday at 1:24 AM



sparsh

MEMBER

Wednesday at 1:24 AM

🔗 #1

As von Neumann computing architectures become increasingly constrained by data-movement overheads, researchers have started exploring in-memory computing (IMC) techniques to offset data-movement overheads.

We present a survey of 90+ papers on in-memory computing using SRAM memory. We review the use of SRAM-IMC for implementing Boolean, search and arithmetic operations, and accelerators for machine learning (especially neural networks), image processing and automata computing. Paper is [here](#), accepted in Elsevier Journal of Systems Architecture 2021.



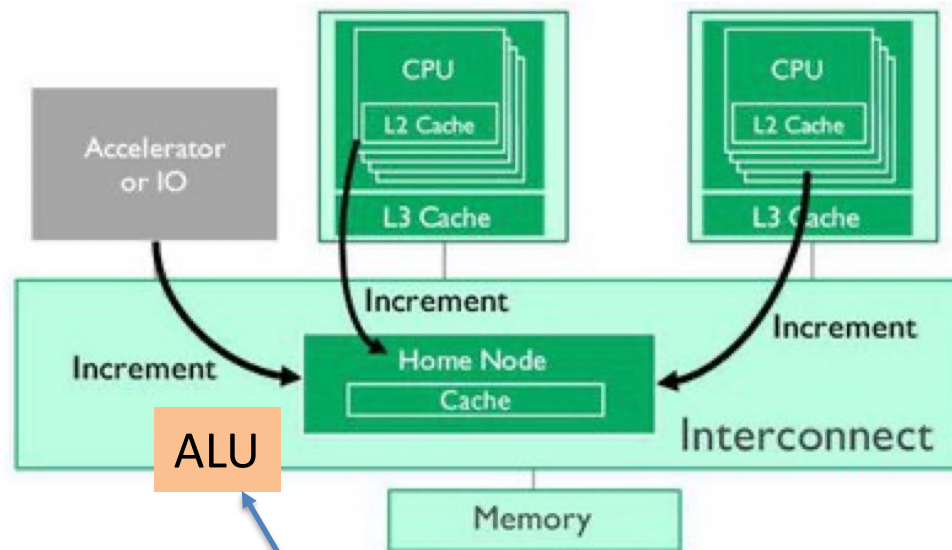
# ARM IMC Extension

COMP222

The atomic instructions offer a rather rich set of functions including: add, sub, \*, and, or, xor, min, max, swap, and compare-and-swap.

## In-Memory Computation

These are executed by the cache/memory controller and not the CPU. A non-CPU device can even issue these request types.



(Source.) [↗](#)

Now, within the controller, there's a 64-bit ALU, and most likely some internal registers to hold arguments and intermediate results. Registers are everywhere.

But, these aren't CPU registers—architectural or microarchitectural—and they're not user visible. They're microarchitectural registers within the memory system implementation.

# ARM IMC Extension

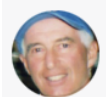
Enter the ARM v8.1 Large System Extensions. Specifically, their new atomic instructions. These make use of special bus support in the [AXi 5 and CHI bus protocols](#). ↗

With these extensions, you can issue special atomic transactions that execute *within the memory system*. [Depending on where the cache line resides, it could execute in L1D, or way out in the L3 cache, last-level cache, or memory controller](#). ↗

The atomic instructions offer a rather rich set of functions including: add, sub,\*, and, or, xor, min, max, swap, and compare-and-swap.

These are executed by the cache/memory controller and not the CPU. A non-CPU device can even issue these request types.

RISC → Load-Store



**Jeff Drobman** · Just now

hey Joe, good stuff, but ... "RISC" is always a "Load-Store" architecture by definition. if there are any other instructions which access memory then those are not "RISC". you introduce here an architectural extension generally called "in-memory computation" — a whole other thing.

# Loop Unrolling



**Lawrence Stewart** · [Follow](#)

Research Computer Scientist · 4h



Loop unrolling is a bit oversold. Out of order processors with any sort of decent branch prediction can run not-unrolled loops at full speed and keep multiple iterations in flight.

Instead, unrolling a largish loop blows up the size of the code in the loop, which may cause performance to be limited by the instruction cache and decode parts of the processor. Small loops will fit inside the "loop buffer" or whatever it is called, but large loops won't.

Loop unrolling is useful for in-order processors that don't do register renaming, it is not clear whether any speedup is achieved for a newish CPU.

Instead, it makes more sense for the compiler to vectorize the loop, using SSE, AVX, and AVX512 if the CPU has those things.

# Loop Unrolling



**Lawrence Stewart** · [Follow](#)

Research Computer Scientist ·

And if your problem is loops without data dependencies between iterations, may I suggest using the GPU? CUDA or SYCL are pretty easy. Or if you have a multicore, then take a look at OpenMP for parallelizing your loops.

As usual, any screed about performance will be incomplete without a suggestion to use performance tools to find out where are the bottlenecks before you start making assumptions about loops.

Loops with any memory references in them can easily miss for large loops and be memory bandwidth limited rather than CPU limited.



**Jeff Drobman** · Just now

very good analysis. seems to me that branch prediction is the most significant factor, so for large loops, prediction efficiency is very high (>99%). next comes L1 cache performance. D-cache should not change, as the data remains the same. but there will be much higher cache miss rate in the I-cache, since loops will re-use code (high locality).

# Section

## Micro Architecture

- ☐ Operand forwarding
- ☐ Branch prediction
- ☐ Speculation/OOE
- ☐ Intel & AMD

➤ See separate slide set

# Section

---

## GPU Graphics

(See separate slide set “GPU”)



# Section



DSP

# 64-Bit Integer Multiply

**How do you write an ALP (assembly language) to multiply the word 3421H by the double word 57412236H and store the result in locations starting in 89000H?**



**Jeff Drobman**, Lecturer at California State University, Northridge (2016-present)

Answered just now

in words not code: 1. load the operands into registers. 2. choose the best multiply instruction (MIPS and ARM have several versions), and use it for the low word. 3. store the product in a pair of 32-bit registers. 4. multiply the high word and store it into a pair of 32-bit registers. 5. finally, sum the 2 partial products after 1st shifting the more significant one left 32 bits (which may be done without actually shifting).

# DSP: 64-Bit Multiply

Quora



**Joe Zbiciak**, I have been programming since grade school

Answered 1m ago

```
1      MVK.S1    0x3421, A0
2  ||  MVK.S2    0x2236, B0
3      MVK.S2    0x5741, B0
4  ||  MPYU.M1X  A0, B0, A2
5  ||  MVK.S1    0x89,   A1
6      MPYU.M2X  A0, B0, B0
7  ||  SHL.S1    A1, 12, A1
8      SHRU.S1   A2, 16, A0
9  ||  STH.D1T1  A0, *A1++
10     ADD.L1X   B0, A0, A0
11     SHRU.S1   A0, 16, A0
12  ||  STH.D1T1  A0, *A1
13     STH.D1T1  A0, *++A1
```

DSP (VLIW) ISA

Multi-issue

# DSP: 64-Bit Multiply

Quora

DSP (VLIW)



**Joe Zbiciak**, I have been programming since grade school

Answered 1m ago

This is actually assembly code for an exposed pipeline VLIW DSP. It has registers A0 through A15 and B0 through B15, and can issue up to 8 instructions in parallel every cycle.

It has two clusters of functional units (A side and B side), and a cross-path that allows each side to read one register from the other side. Also, the load/store units can associate addresses on either side with data from either side.

The 8 units are:

- L1, L2: *Logical/Long*. 32-bit add/sub, 40-bit add/sub, booleans, compares (aka. "logicals").
- S1, S2: *Shift*. 32-bit add/sub, 32- & 40-bit shifts, booleans, 16-bit constant generation, branches. Branches have 6 exposed delay slots.
- D1, D2: *Data*. 32-bit add/sub, 32-bit address arith, load/store. Loads have 4 exposed delay slots.
- M1, M2: *Multiply*. 16×16 → 32 multiplies with 1 exposed delay slot

# DSP: 64-Bit Multiply

Quora

DSP (VLIW)



**Joe Zbiciak**, I have been programming since grade school

Answered 1m ago

The 1/2 in the name specifies A side vs. B side. An X indicates using the cross-path. For LD/ST, the T1/T2 suffix indicates which side the data is on, while D1/D2 indicates which side the address is on.

And, the parallel bars || specify that the instruction issues in parallel with the previous instruction. Example:

```
MPY.M1X A1, B2, A3 ; A3 = A1 × B2
|| ADD.L1 A2, A1, A0
```

This issues an MPY to the M1 unit in parallel with an ADD on the L1 unit. The multiply reads a single operand from the B side.

You could swap two registers in one cycle w/out a temporary like this:

```
MV.L2 B1, B2
|| MV.S2 B2, B1
```

That takes advantage of the fact they issue in parallel.

The first ~20 years of my career involved members of that DSP family. 😊

It's a beast to program, but I was able to wring quite a lot out of it.

# DSP: 64-Bit Multiply

Quora

DSP (VLIW)



**Joe Zbiciak**, I have been programming since grade school

Answered 1m ago

Yes, indeed it was. I was at TI for just shy of 20 years. I'm only avoiding naming it for OP's sake.

Before I left at the end of 2015, we were working on a 13-issue, 64-bit VLIW DSP with 512-bit vectors. It did finally end up in products just before COVID-19 hit, with press just as COVID was taking off:

[www.eetimes.com/isscc-2020-chipl...](http://www.eetimes.com/isscc-2020-chipl...)

[training-ti-com/sites/default/fi...](http://training-ti-com/sites/default/fi...)

- ❖ 13-issue
- ❖ 64-bit VLIW

I was the lead architect for the multidimensional vector streaming engine.

(The diagram shows 12 units; however, there were a variety of "unitless" instructions that effectively formed a 13th unit.)

I told my former team I'd come back to buy everyone a round of drinks once it released, but COVID-19 put the kibosh on that. :-(



# DSP Chip (TI)

COMP222 **Quora**

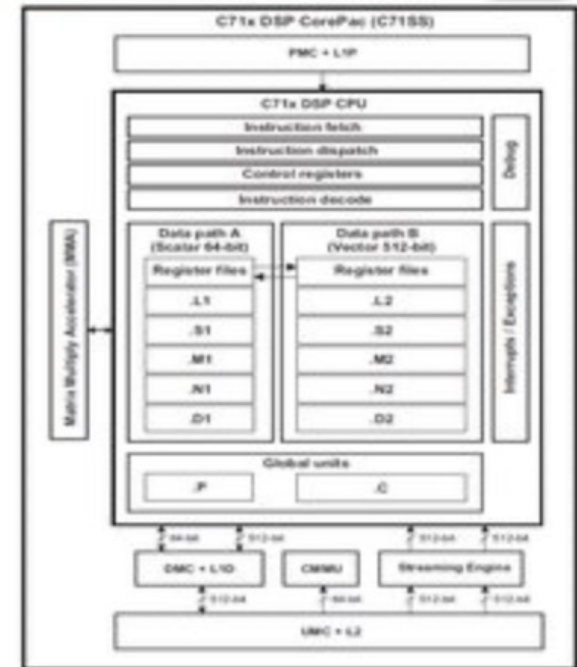
## Heterogeneous processing cores

### C71x DSP (up to 1 GHz):

- Next-generation, TI-true 64b DSP core:
  - 512b SIMD processing
  - Dual-data path CPU
    - 64-bit scalar + 512-bit vector
  - Vision processing enhancements
  - OpenVX support for computer vision processing
- Matrix Multiply Accelerator (MMA) for deep learning
- Memory system:
  - 32kB L1 program cache
  - 48kB L1 data cache or RAM
  - 512kB L2 unified cache or RAM
  - Access to L3 with IO and CPU cache coherency
  - CorePac Memory Management Unit (CMMU)
    - ARM®v8-A compliant
- Multi-dimensional Streaming Engine (SE) provides high-speed synchronous access to L3 memory

Assuming 1 GHz as clock for frequency for C6x, C7x and MMA.

		C6x	C7x	MMA
GMAC	Fixed 8-bit	32	144	4096
	Fixed 16-bit	32	144	1024
GOPS	Fixed 8-bit	96	496	8192
	Fixed 16-bit	80	392	2048
GFLOPS	Float	16	88	NA



**Joe Zbiciak**, I have been programming since grade school

Answered 1m ago

I was the lead architect for the multidimensional vector streaming engine.

(The diagram shows 12 units; however, there were a variety of "unitless" instructions that effectively formed a 13th unit.)