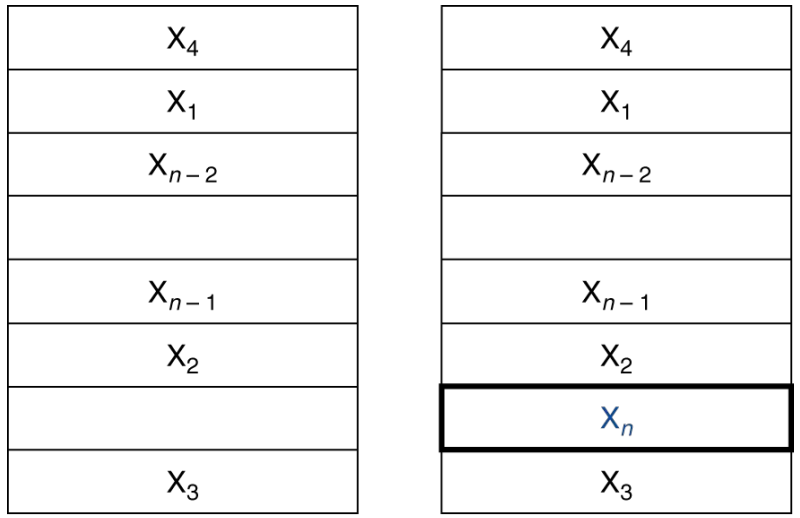
CPU Microarchitecture

Cache Memory

- Cache memory
  - The level of the memory hierarchy closest to the CPU
- Given accesses X_1, …, $X_{n–1}$, X_n

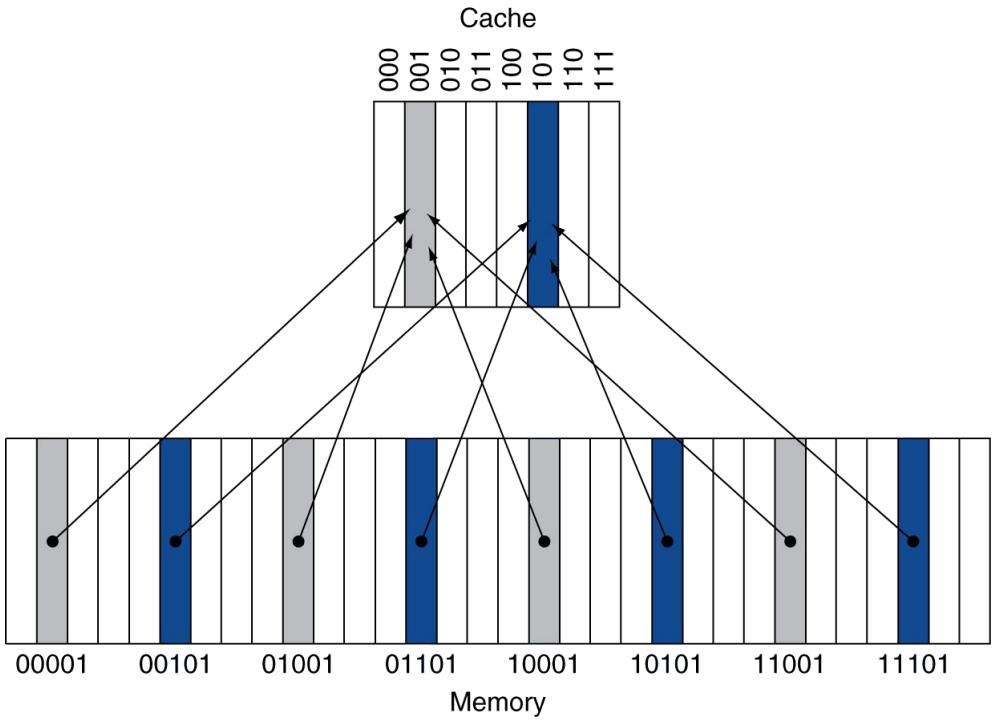


a. Before the reference to X_n b. After the reference to X_n

- How do we know if the data is present?
- Where do we look?

Direct Mapped Cache

- Location determined by address
- Direct mapped: only one choice
  - (Block address) modulo (#Blocks in cache)



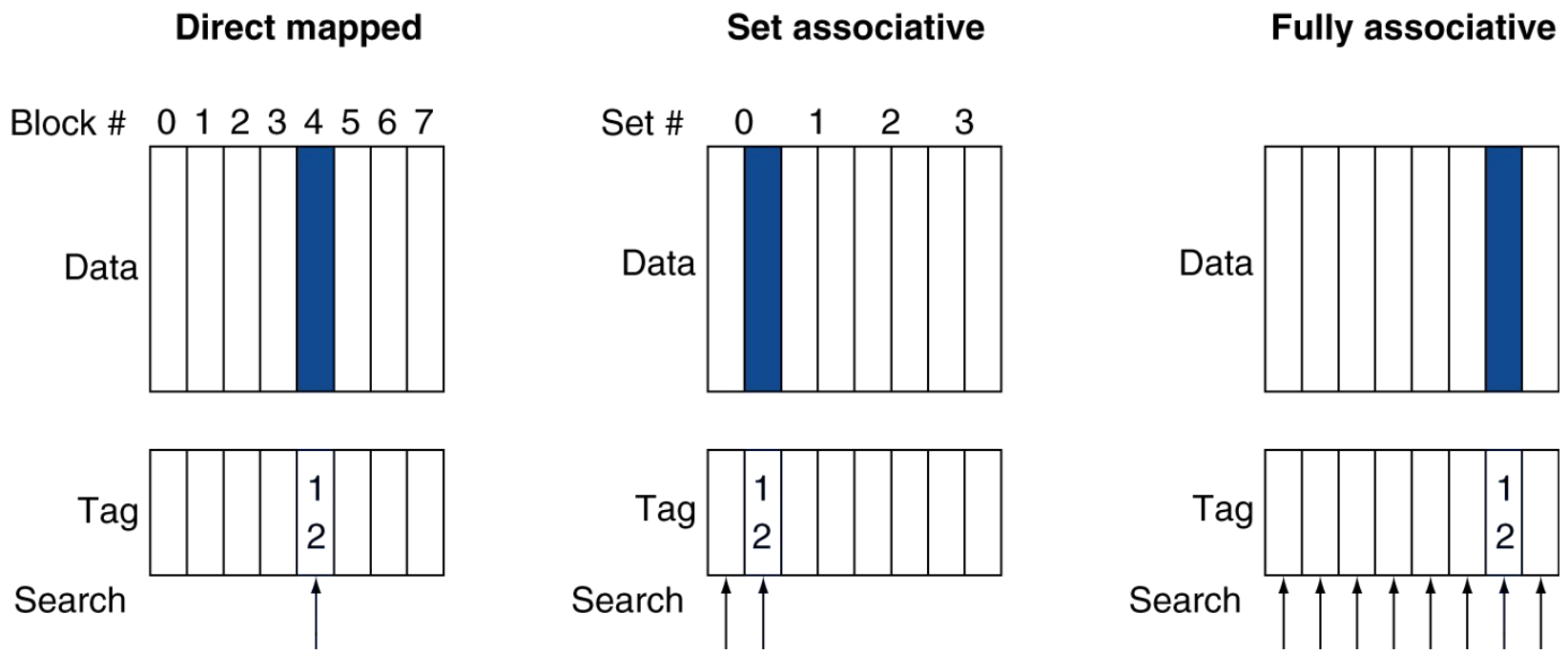
- #Blocks is a power of 2
- Use low-order address bits

Associative Caches

- Fully associative
  - Allow a given block to go in any cache entry
  - Requires all entries to be searched at once
  - Comparator per entry (expensive)

- *n*-way set associative
  - Each set contains *n* entries
  - Block number determines which set
    - (Block number) modulo (#Sets in cache)
  - Search all entries in a given set at once
  - *n* comparators (less expensive)

Associative Cache Example

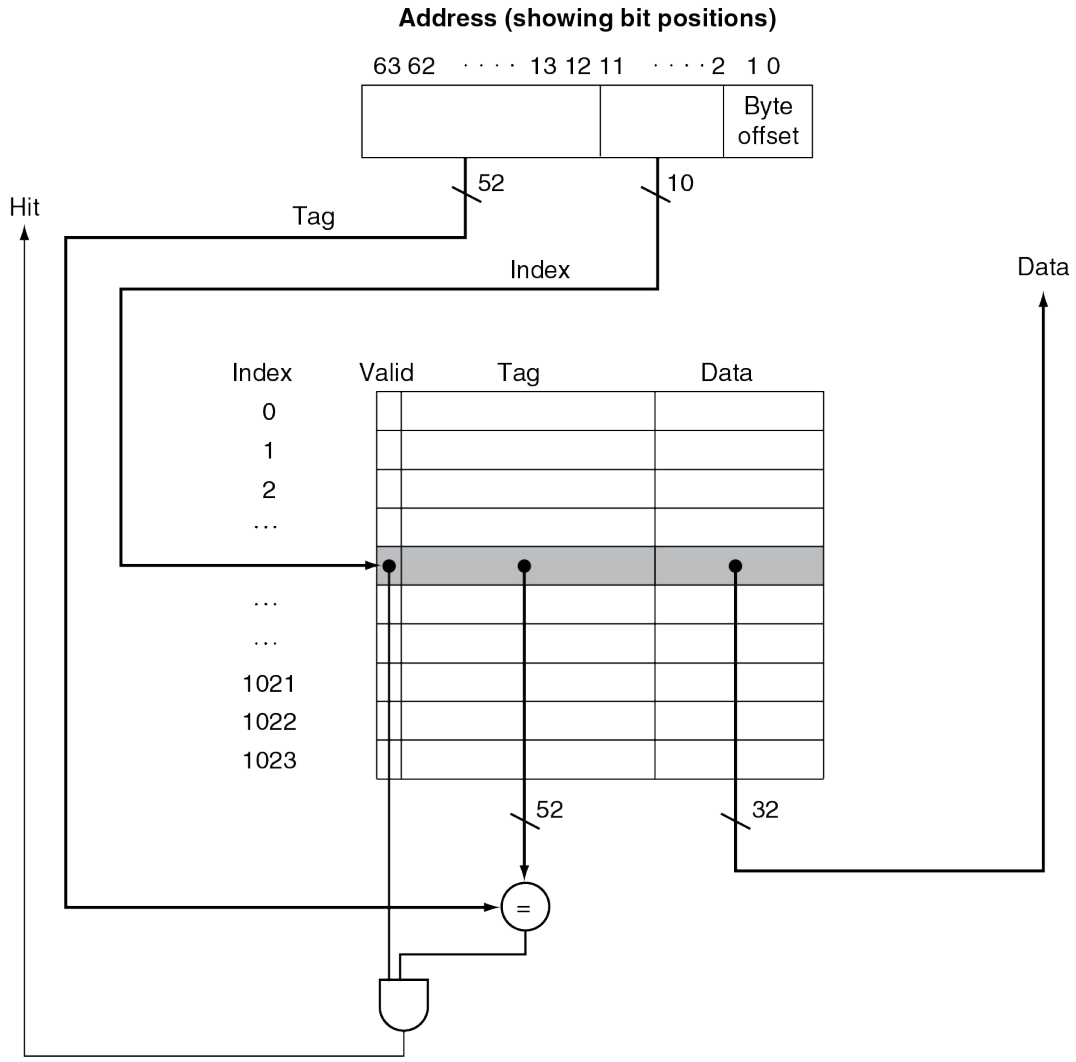


How Much Associativity

- Increased associativity decreases miss rate
  - But with diminishing returns

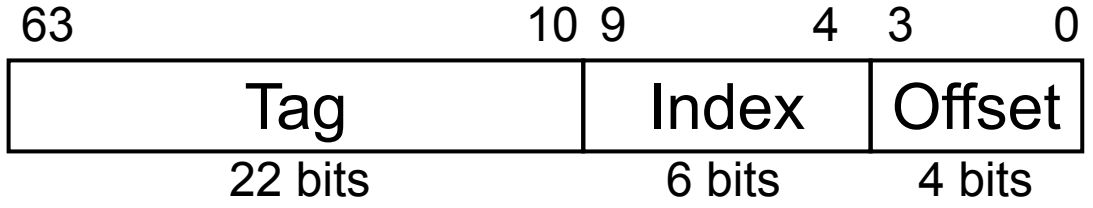
- Simulation of a system with 64KB D-cache, 16-word blocks, SPEC2000
  - 1-way: 10.3%
  - 2-way: 8.6%
  - 4-way: 8.3%
  - 8-way: 8.1%

Address Subdivision



Example: Larger Block Size

- Cache size of 64 **blocks**, 16 bytes/block
  - To what block number does address 1200 map?
- Block address = ⌊1200/16⌋ = 75 □ Tag | Index
- Block number = 75 modulo 64 = 11 □ Index



Block Size Considerations

- Larger blocks should reduce miss rate
  - Due to spatial locality
- But in a fixed-sized cache
  - Larger blocks ⇒ fewer of them
    - More competition ⇒ increased miss rate
  - Larger blocks ⇒ pollution
- Larger miss penalty
  - Can override benefit of reduced miss rate
  - Early restart and critical-word-first can help

Measuring Cache Performance

- Components of CPU time
  - Program execution cycles
    - Includes cache hit time
  - Memory stall cycles
    - Mainly from cache misses
- With simplifying assumptions:

$$\text{Memory stall cycles} = \#\text{Memory accesses} \times \text{Miss rate} \times \text{Miss penalty}$$

Cache Performance Example

- Given
  - I-cache miss rate = 2%
  - D-cache miss rate = 4%
  - Miss penalty = 100 cycles
  - Base CPI (ideal cache) = 2
  - Load & stores are 36% of instructions
- Miss cycles per instruction
  - I-cache: 0.02 × 100 = 2 cc per instruction
  - D-cache: 0.36 × 0.04 × 100 = 1.44 cc per instruction
- Actual CPI = 2 + 2 + 1.44 = 5.44 cc per instruction (3.44 stalled cycles)
  - Ideal CPU is 5.44/2 =2.72 times faster

Average Access Time

- Hit time is also important for performance
- Average memory access time (AMAT)
  - AMAT = Hit time + Miss rate × Miss penalty
- Example
  - CPU with 1ns clock, hit time = 1 cycle, miss penalty = 20 cycles, I-cache miss rate = 5%
  - AMAT = 1 + 0.05 × 20 = 2ns
    - 2 cycles per instruction

Performance Summary

- When CPU performance increased
  - Miss penalty becomes more significant
- Decreasing base CPI
  - Greater proportion of time spent on memory stalls
- Increasing clock rate
  - Memory stalls account for more CPU cycles
- Can’t neglect cache behavior when evaluating system performance

Multilevel Caches

- Primary cache attached to CPU
  - Small, but fast
- Level-2 cache services misses from primary cache
  - Larger, slower, but still faster than main memory
- Main memory services L2 cache misses
- Some high-end systems include L3 (LLC) cache

Multilevel Cache Example

- Given
  - CPU base CPI = 1, clock rate = 4GHz
  - Miss rate/instruction = 2%
  - Main memory access time = 100ns
- With just primary cache
  - Miss penalty = 100ns/0.25ns = 400 cycles
  - Effective CPI = 1 + 0.02 × 400 = 9

Example (cont.)

- Now add L-2 cache
  - Access time = 5ns
  - Global miss rate to main memory = 0.5%
- Primary miss with L-2 hit
  - Penalty = 5ns/0.25ns = 20 cycles
- Primary miss with L-2 miss
  - Extra penalty = 500 cycles
- CPI = 1 + 0.02 × 20 + 0.005 × 400 = 3.4
- Performance ratio = 9/3.4 = 2.6

Multilevel Cache Considerations

- Primary cache
  - Focus on minimal hit time
- L2 cache
  - Focus on low miss rate to avoid main memory access
  - Hit time has less overall impact
- Results
  - L1 cache usually smaller than a single cache
  - L1 block size smaller than L2 block size

Interactions with Advanced CPUs

- Out-of-order CPUs can execute instructions during cache miss
  - Pending store stays in load/store unit
  - Dependent instructions wait in reservation stations
    - Independent instructions continue
- Effect of miss depends on program data flow
  - Much harder to analyse
  - Use system simulation

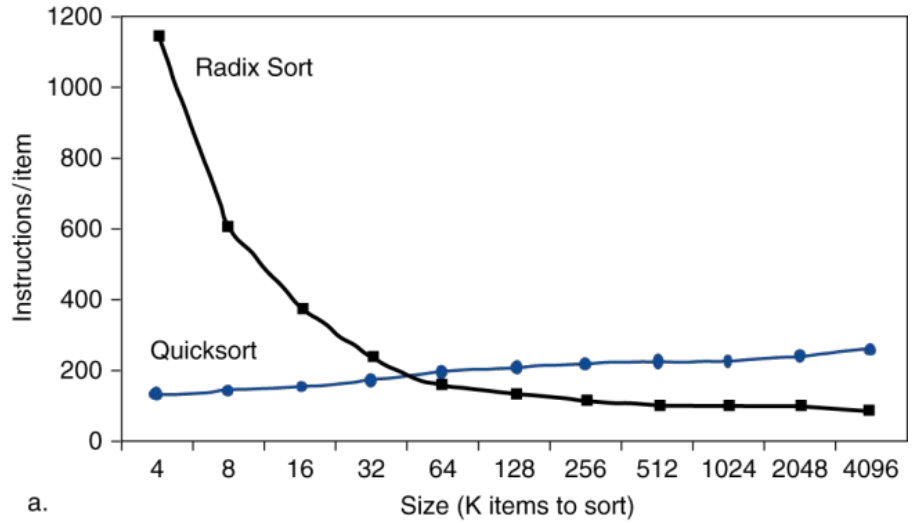
Interactions with Software

- Misses depend on memory access patterns
  - Algorithm behavior
  - Compiler optimization for memory access

Example with sorting algorithms

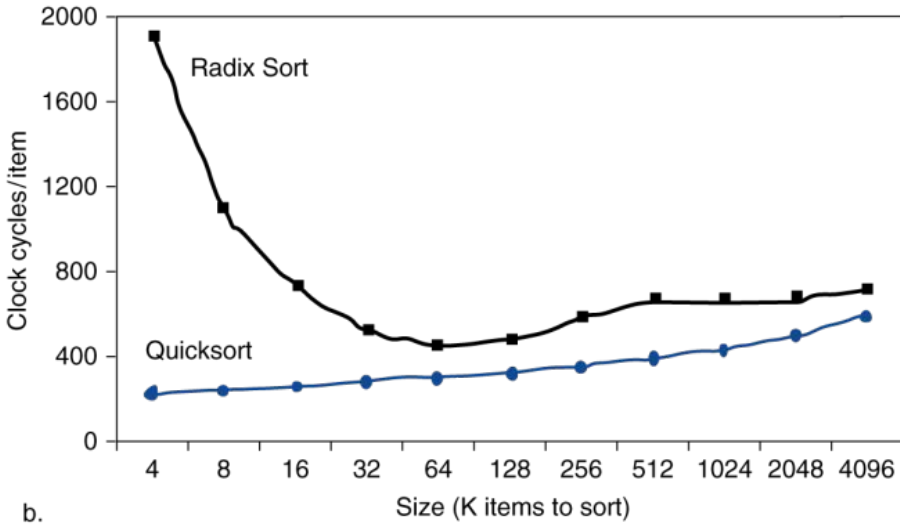
- Quicksort and Radix sort are 2 sorting algorithms

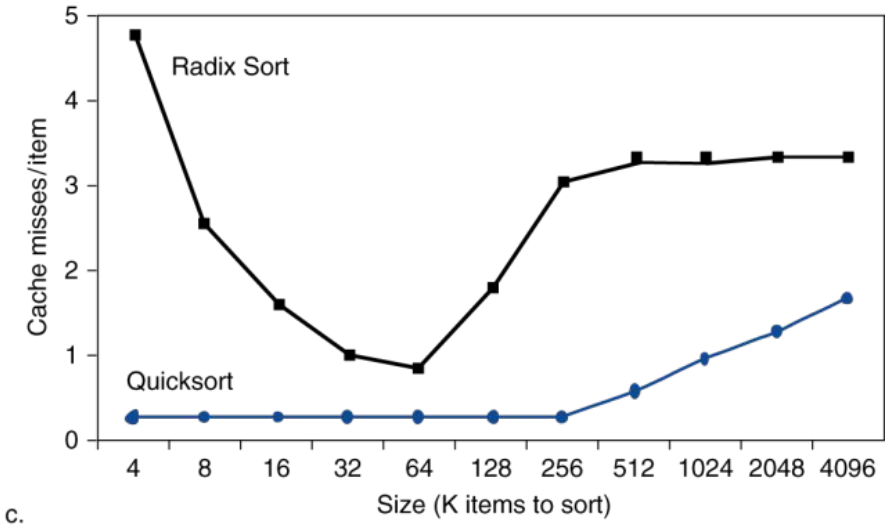
- With a big number of items Radix sort should work better than quicksort



Example with sorting algorithms

- Actually Radix sort is always worse than quick sort
- Motivation is the algorithm miss rate





Sources of Misses

- ***Compulsory misses*** (aka *cold start misses*)
  - First access to a block
- ***Capacity misses***
  - Due to finite cache size
  - A replaced block is later accessed again
- ***Conflict misses*** (aka *collision misses*)
  - In a non-fully associative cache
  - Due to competition for entries in a set
  - Would not occur in a fully associative cache of the same total size

Hardware and Software Prefetching

- One method to avoid cache misses is to prefetch data into caches prior to when the pipeline demands it. The miss penalty can be mostly hidden if the prefetch request is issued sufficiently ahead in the pipeline.
- Most CPUs provide implicit hardware-based prefetching and explicit software prefetching.
- Hardware prefetchers observe the behavior of a running application and initiate prefetching on repetitive patterns of cache misses. However, hardware prefetching works for a limited set of commonly used data access patterns.
- Software memory prefetching complements prefetching done by hardware. Developers can specify which memory locations are needed ahead of time via dedicated instruction. Compilers can also automatically add prefetch instructions into the code to request data before it is required.
- Prefetch techniques need to balance between demand and prefetch requests to guard against prefetch traffic slowing down demand traffic.

Cache Coherence Problem

- Suppose two CPU cores share the address space
  - Write-through caches

Time step	Event	CPU A's cache	CPU B's cache	Memory
0				0
1	CPU A reads X	0		0
2	CPU B reads X	0	0	0
3	CPU A writes 1 to X	1	0	1

Cache Coherence Protocols

- Operations performed by caches in multiprocessors to ensure coherence
  - Migration of data to local caches
    - Reduces bandwidth for shared memory
  - Replication of read-shared data
    - Reduces contention for access
- Snooping protocols
  - Each cache monitors bus reads/writes
- Directory-based protocols
  - Caches and memory record sharing status of blocks in a directory

Invalidating Snooping Protocols

- Cache gets exclusive access to a block when it is to be written
  - Broadcasts an invalidate message on the bus
  - Subsequent read in another cache misses
    - Owning cache supplies updated value

CPU activity	Bus activity	CPU A's cache	CPU B's cache	Memory
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes 1 to X	Invalidate for X	1		0
CPU B read X	Cache miss for X	1	1	1

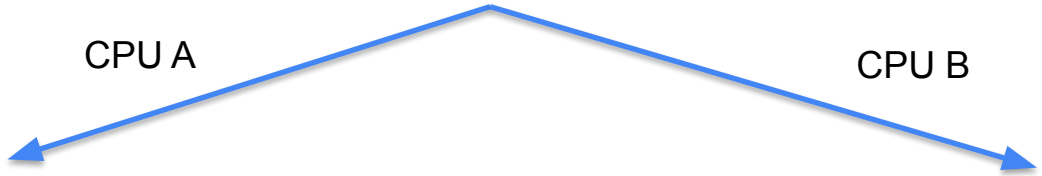
False sharing

- Occurs when two CPUs use different data stored in the same cache line

- Writing in one location invalidate all the line (also the data owned by the other CPU)

False sharing

```
for (i=0; i <n; i=i+1)
   Y[i] = a * X[i] + Y[i];
```




```
for (i=0; i <n; i=i+2)
   Y[i] = a * X[i] + Y[i];
```

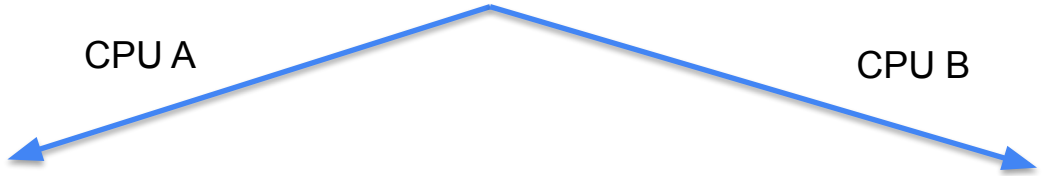
```
for (i=1; i <n; i=i+2)
   Y[i] = a * X[i] + Y[i];
```

CPU cache A							
Y[0]	Y[1]	Y[2]	Y[3]	Y[4]	Y[5]	Y[6]	Y[7]

CPU cache B							
Y[0]	Y[1]	Y[2]	Y[3]	Y[4]	Y[5]	Y[6]	Y[7]

False sharing

```
for (i=0; i <n; i=i+1)
   Y[i] = a * X[i] + Y[i];
```




```
for (i=0; i <n; i=i+2)
   Y[i] = a * X[i] + Y[i];
```

```
for (i=1; i <n; i=i+2)
   Y[i] = a * X[i] + Y[i];
```

After CPU A execution

CPU cache A							
Y[0]	Y[1]	Y[2]	Y[3]	Y[4]	Y[5]	Y[6]	Y[7]

CPU cache B							
~~Y[0]~~	~~Y[1]~~	~~Y[2]~~	~~Y[3]~~	~~Y[4]~~	~~Y[5]~~	~~Y[6]~~	~~Y[7]~~

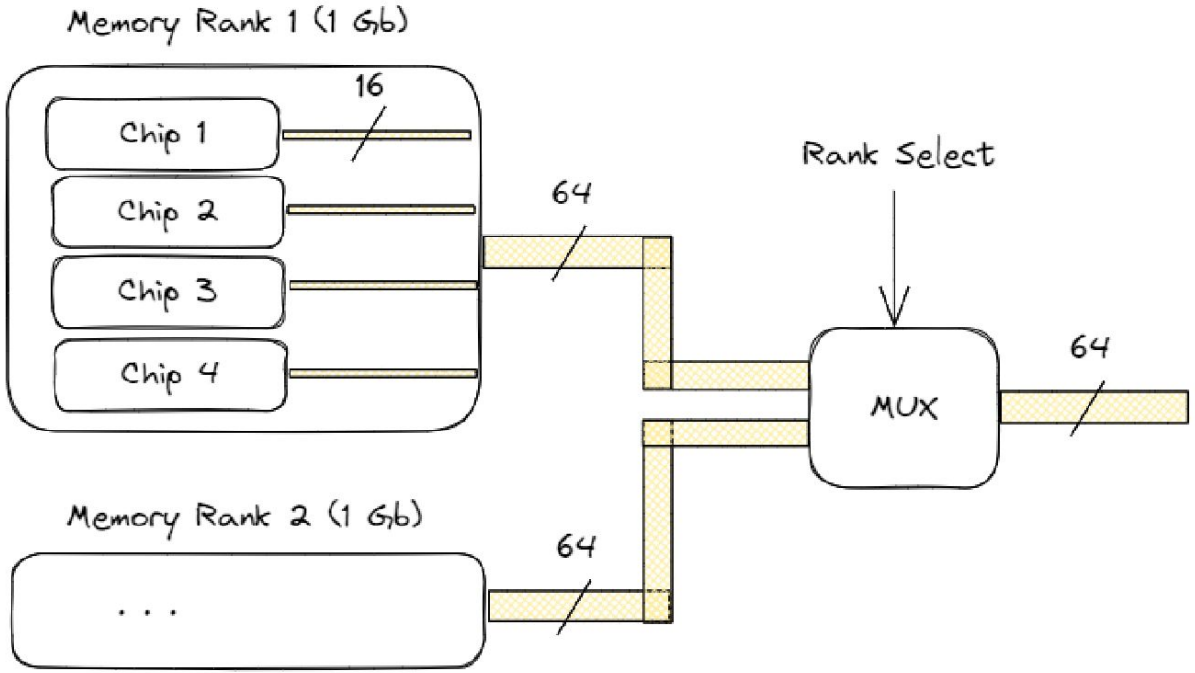
CPU B needs to reload Y[1]

Execution can be slower than using one CPU

Main Memory

Main Memory

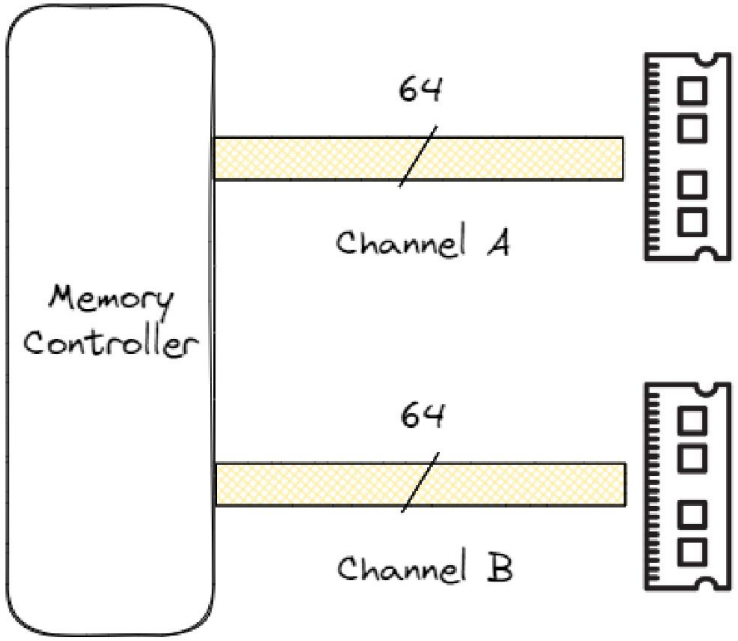
- Main memory uses DRAM (Dynamic Random Access Memory) modules that supports large capacities at reasonable cost points.
- A DRAM module is organized as a set of DRAM chips.
- The memory *rank* describes how many sets of DRAM chips exist on a module



Main Memory

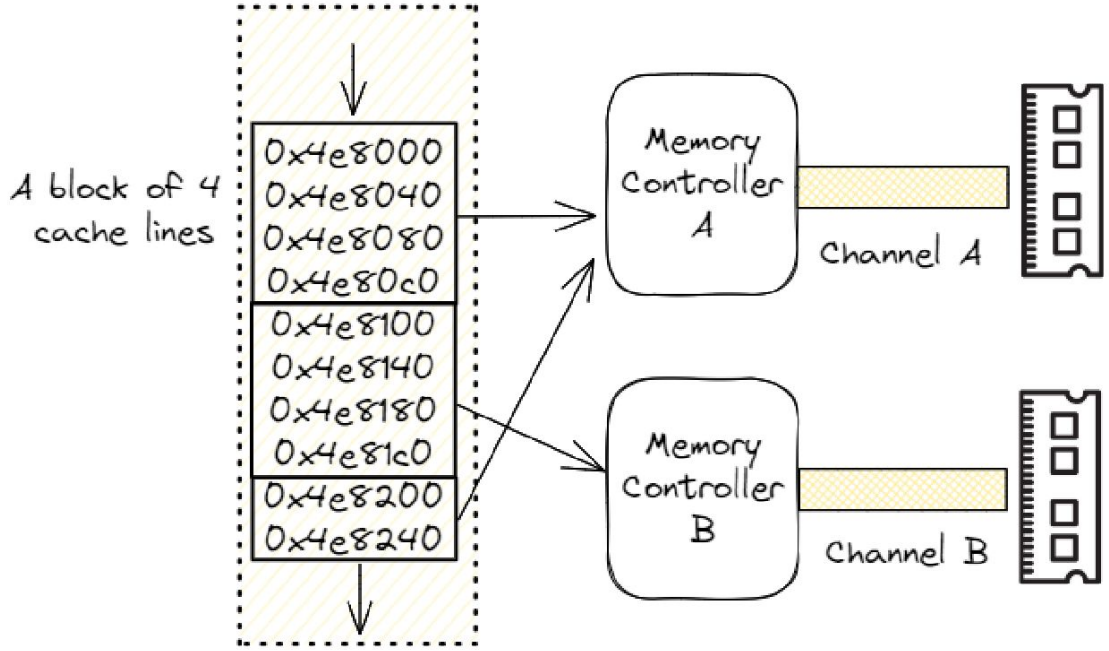
- Multiple DRAM modules can also increase memory bandwidth.

- The multi-channel architectures increase the width of the memory bus, allowing DRAM modules to be accessed simultaneously.



Memory interleaving

- *Interleaving* spreads adjacent addresses within a page across multiple memory devices. Below, an example of a 2-way interleaving for sequential memory accesses.



- Note that memory bandwidth can be the primary bottleneck, i.e. the application stops scaling, and adding more cores doesn’t make it run faster.

Memory interleaving

The *interleaving granularity* controls how big each striped block is before switching to the next channel

- Fine granularity → interleave at small blocks (cache line or 4KB page level).
  - Maximizes parallelism and bandwidth utilization.
  - Best for streaming, bandwidth-heavy HPC workloads (CFD, linear algebra, tensor ops).

- Coarse granularity → interleave at larger blocks (e.g., 2MB, 1GB).
  - Improves **locality** (important for **NUMA-aware** applications like MPI).
  - Best when software is explicitly **NUMA/hugepage-aware** (e.g., HPC MPI jobs pinned to sockets).

Virtual Memory

Virtual Memory

Virtual memory:

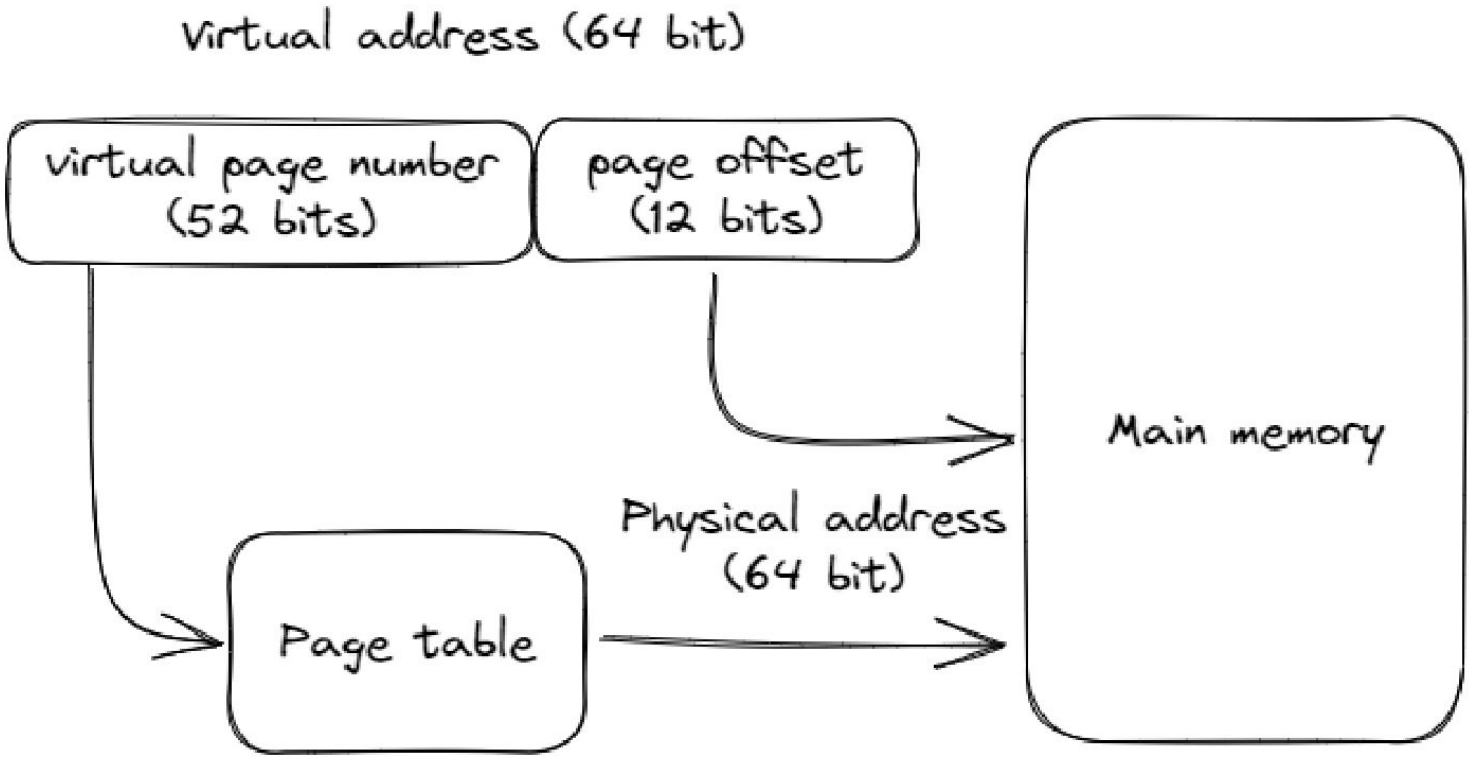
- allows sharing the physical memory attached to a CPU with all the processes executing on it.

- provides a protection mechanism that prevents access to the memory allocated to a given process from other processes.

- provides relocation, which is the ability to load a program anywhere in physical memory without changing the addresses in the program.

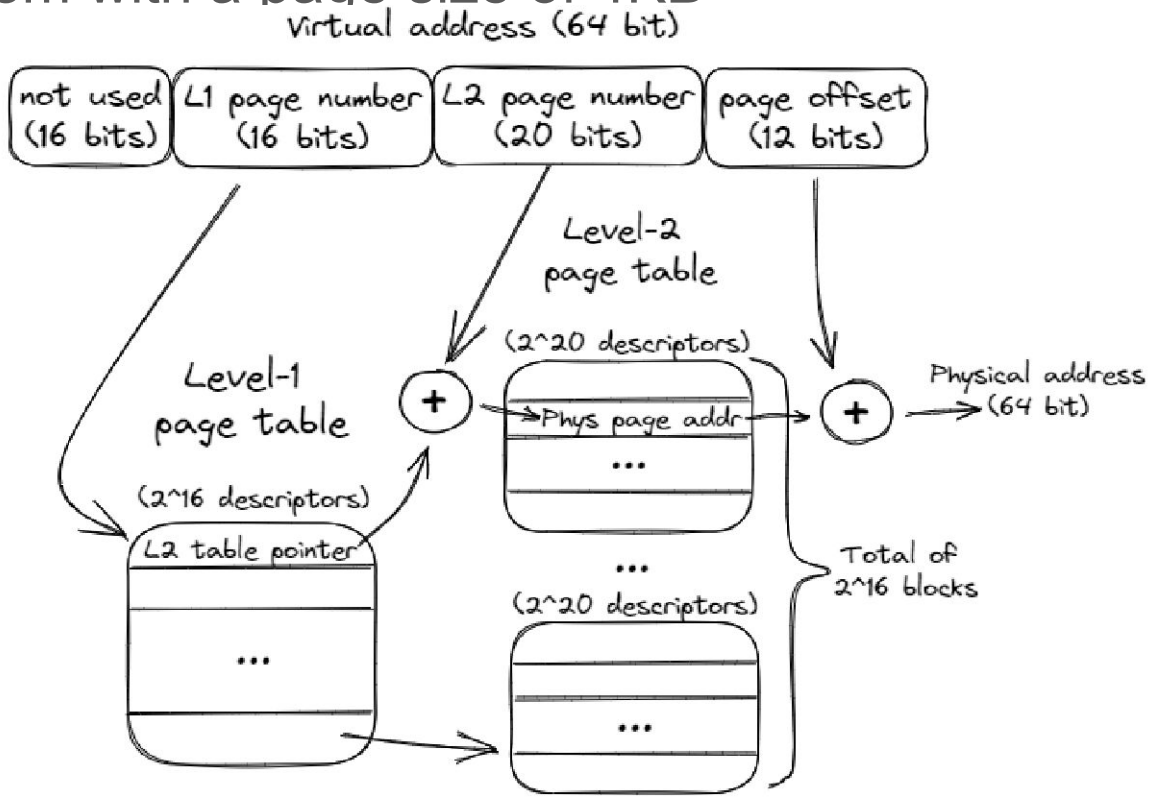
Virtual Memory

Virtual memory is divided into pages.



Virtual Memory

The translation mechanism for a system with a page size of 4KB



- Traversing through the hierarchy potentially requires several indirect accesses. Such a traversal is called a *page walk*.

Virtual Memory

The page table can be either single-level or nested.

- The 16 most significant bits are not used. (Some applications use those unused bits to keep metadata, also known as ***pointer tagging****). Even with the remaining 48 bits we can address 256 TB of total.

- Failure to provide a physical address mapping is called a **page fault**. It occurs if
  - the operating system is committed to allocating a page
  - an accessed page was swapped out to disk and is not currently stored in RAM.

- The exact format of the page table is dictated by the CPU.

*usage example are: to detect memory safety bugs, to manage memory garbage collection, provide pointer reference counters

Translation Lookaside Buffer (TLB)

The translation lookaside buffer (**TLB**) caches the most recently used translations.

- TLBs are often designed as a hierarchy of L1 ITLB (Instructions), and L1 DTLB (Data), followed by a shared (instructions and data) L2 STLB.

- To speed up the handling of TLB misses, CPUs have a mechanism called a *hardware page walker*. Such a unit can perform a page walk directly in hardware by issuing the required instructions to traverse the page table. This is the reason why the format of the page table is dictated by the CPU,

Cache and Virtual Memory

- L1 caches (data + instruction): usually Virtually Indexed, Physically Tagged (VIPT) indexing
  - Cache set selection uses bits from the virtual address → this avoids waiting for the page translation (the page offset bits of the virtual address are the same before and after translation).
  - Tag comparison (to check for a hit/miss) uses the physical address → this ensures correctness even if the same physical page has multiple virtual aliases.
- L2 and L3 caches: almost always Physically Indexed, Physically Tagged (PIPT) → they work only with physical addresses.

Huge Pages

Having a small page size makes it possible to manage the available memory more efficiently and reduce fragmentation. The drawback though is that it requires having **more page table entries** to cover the same memory region.

Huge Pages (2MB or 1GB) drastically reduce the pressure on the TLB hierarchy since fewer TLB entries are required. It greatly **increases the chance of a TLB hit.**

The downsides of using huge pages are **memory fragmentation** and, in some cases, nondeterministic **page allocation latency** because it is harder for the OS to find a contiguous chunk of memory. If this cannot be found, the OS needs to reorganize the pages, resulting in a longer allocation latency.

Compute Node Architecture

Compute Node Architecture

A compute node can be considered a basic building block of computing systems today.

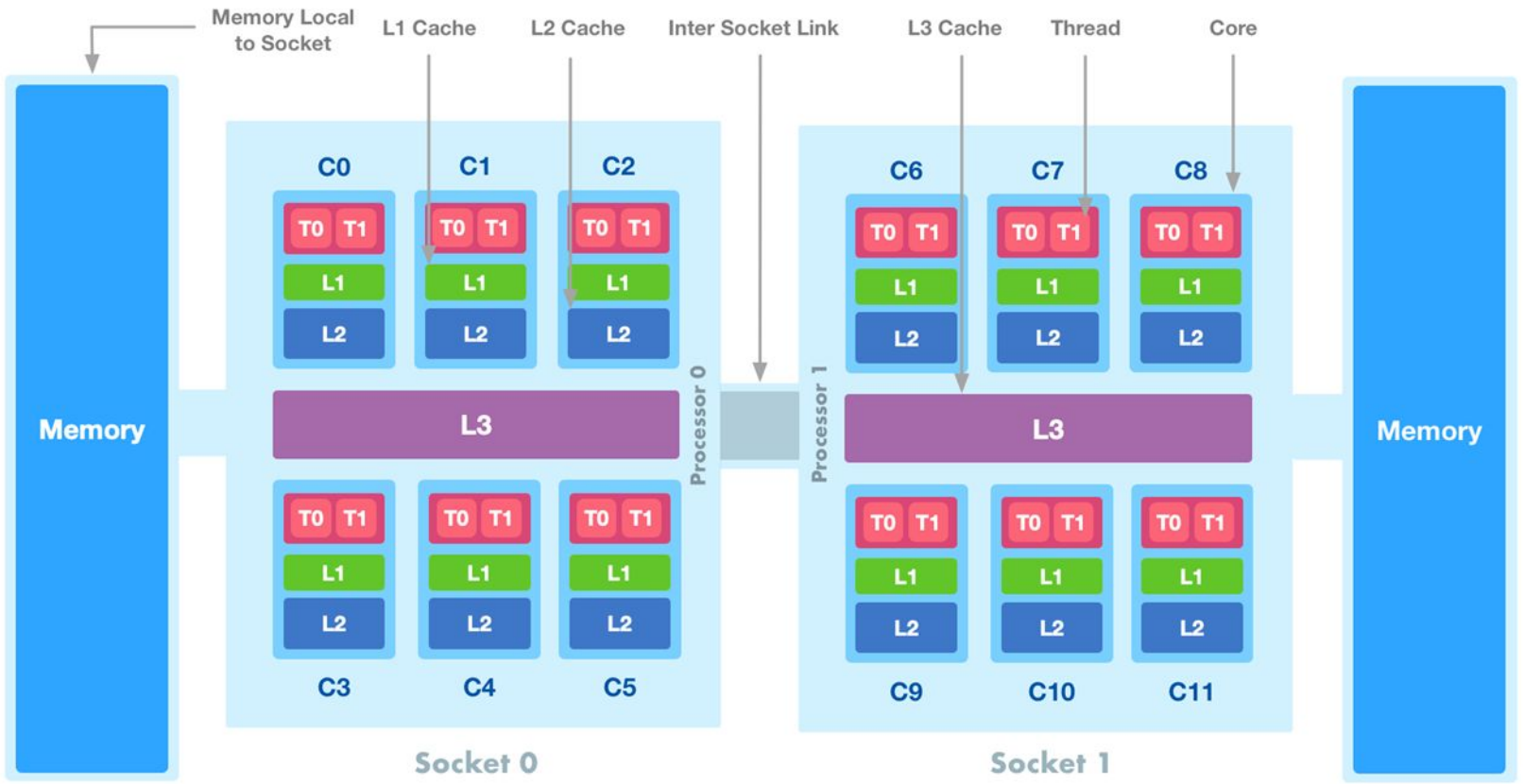
- A node consists of one or more **sockets**. A socket is hardware on the motherboard where **processor** (CPU) is mounted. A processor has components like ALU, FPU, Registers, Control Unit, and Caches are collectively called a **core** of a processor.

- Cores can be replicated on a single chip to form a **multi-core** processor.

- Each socket has main memory which is accessible to all processors in a node through some form of inter-socket link (like HT, QPI or UPI). If the processor can access its own socket memory faster than the remote socket then the design is referred to as Non-Uniform Memory Access.

Compute Node Architecture

Example of a compute node



The uncore

The "**uncore** in an *x86* processor refers to the components and functions of the CPU that are not part of the individual processing cores but are integrated onto the same silicon die.

Heavy memory-bound workloads stress the uncore more than compute-bound workloads.

Key Components and Functions of the uncore are:

Last-Level Cache (LLC): Typically the L3 cache, which is shared among all processor cores. For Intel, the Cache Box (CBox) manages cache coherency and interfacing with the LLC.

Memory Controllers (MBox): These manage how the CPU accesses the system's main memory (RAM).

The uncore

Interconnects (PBox): These provide high-speed communication pathways between the cores themselves, and between the CPU and other system components. Examples include Intel's Ultra Path Interconnect (UPI) and AMD's Infinity Fabric.

PCI Express (PCIe) and I/O Controllers: These handle high-speed peripheral connections, such as graphics cards and storage devices.

System Configuration Controller (UBox): This component, along with a router (RBox), manages connections between various uncore elements like the PBox, CBox, and MBox.

Power Control Logic (PWR): The uncore plays a significant role in power management, including dynamic uncore frequency scaling to optimize for both energy efficiency and performance based on workload. Note that CPUs often allow separate **uncore frequency scaling** from core frequency.

Core Design

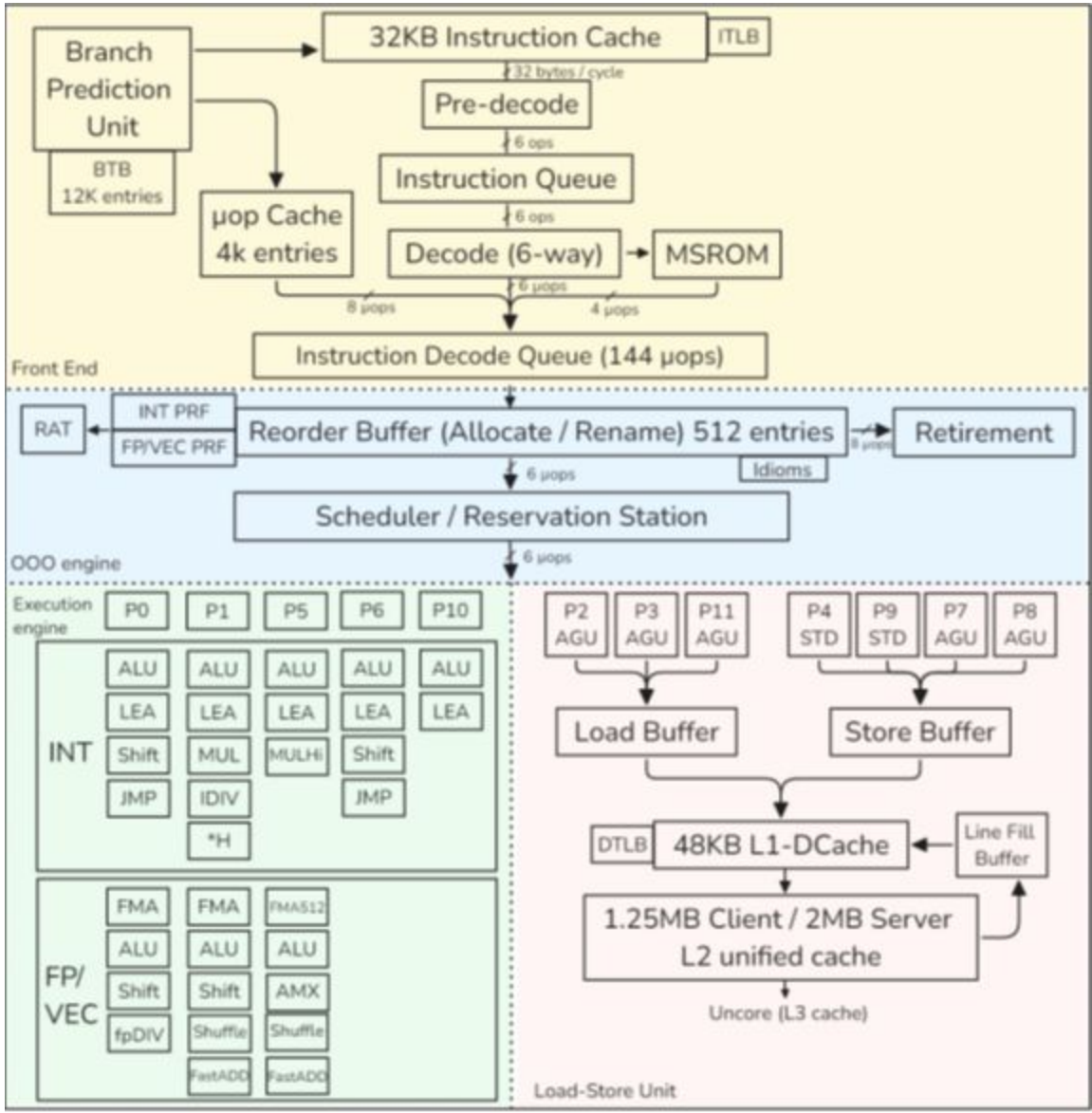
Intel Golden Cove

We take as example the implementation of Intel's 12th-generation core, Golden Cove:

- The core is split into an in-order frontend that fetches and decodes x86 instructions into *μ*ops and a 6-wide superscalar, out-of-order backend.
- The Golden Cove core supports 2-way SMT.
- It has a 32KB L1 I-cache, a 48KB L1 D-cache and a 1.25MB L2 cache.

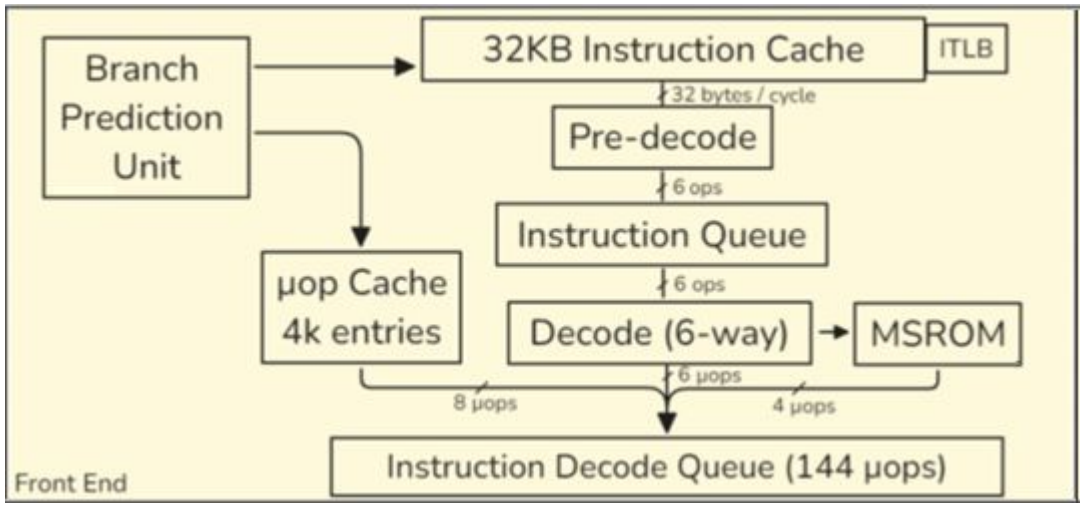
Intel Golden Cove

- In order front-end: fetch and decode instructions
- CPU Backend: employs an OOO engine that executes instructions and stores results.
- Execution engine: perform multiple INT/FP/SIMD operations in parallel
- Load-Store Unit: interact with the CPU memory



Front-end

- The BPU predicts the target of all branch instructions and provide the next instruction to fetch based on this prediction.
- The frontend fetches **32 bytes** per cycle of x86 instructions from the L1 I-cache. This is shared among the two threads.



- The **pre-decode** stage marks the boundaries of the variable length x86. instructions and moves up to 6 instructions (*macroinstructions*) to the **Instruction Queue**.
- The **6-way decoder** converts the complex macro-Ops into fixed-length *µops*. Decoded *µops* are queued into the **Instruction Decode Queue**.

Front-end

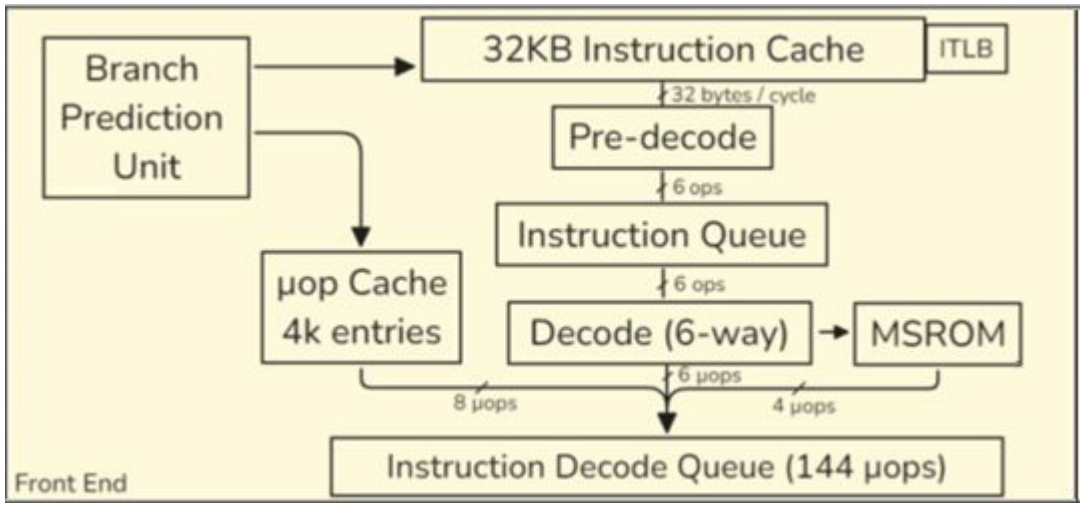
X86 is a CISC architecture, so instructions can be are composed of multiple *µops.*

Examples:

- A simple register-to-register addition instruction such as `ADD rax,rbx` generates only one *µop*,
- `ADD rax,[mem]` generate twos: one for loading from the mem memory location into a temporary unnamed) register, and one for adding it to the rax register.
- The instruction `ADD [mem],rax` generates three *µops*: one for loading from memory, one for adding, and one for storing the result back to memory.

Front-end

- The *µop* cache, also called **Decoded Stream Buffer (DSB)** stores the more recently used decoded instruction. The DSB can provide 8 *µops* per cycle and can hold up to 4K entries.
  - The DSB reduces:
    - Decoder power consumption
    - Latency for instruction fetch
    - Front-end bottlenecks



- The **MSROM (Microcode Sequencer)** manages some “complex” or “rare” x86 instructions (string ops, system instructions, certain FP ops, etc.) that need a longer sequence of *µops*.
- Those sequences are predefined microprograms stored in the MSROM. So the MSROM is essentially a library of *µop* routines burned into the CPU.

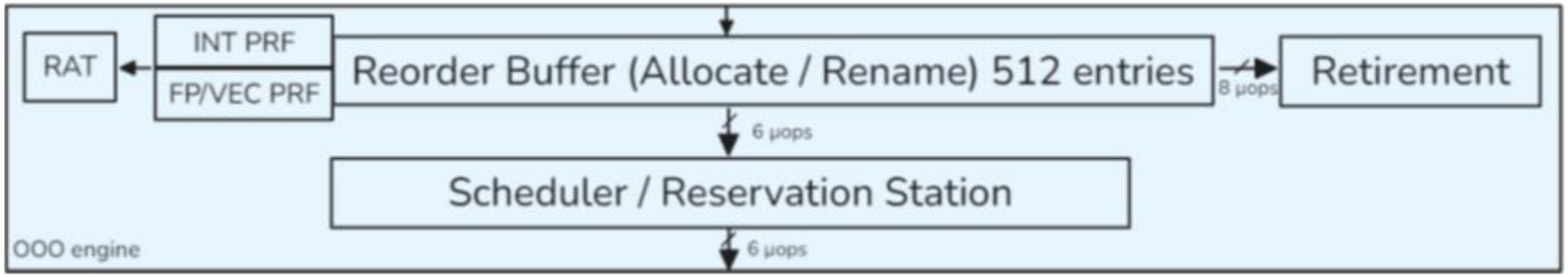
Front-end

Remember that SMT needs to duplicate some hardware blocks and to share other HW blocks. For the front-end

- if SMT is enabled the two SMT threads alternate every cycle to access the instruction fetch, the pre-decode, the instruction queue and the decoder.

- The **DSB** and the **Instruction Decode Queue** is partitioned (72 entries for each thread) between the two threads.

back-end



The CPU Backend employs an OOO engine that executes instructions and stores results.

The ReOrder Buffer (ROB):

- it provides register renaming. There are only 16 general-purpose integer and 32 floating-point/SIMD architectural registers, however, the number of physical registers is much higher (around 300).
    - Physical registers are located in a structure called the Physical Register File (PRF). There are separate PRFs for integer and floating-point/SIMD registers.
    - The mappings from architecture-visible registers to the physical registers are kept in the register alias table (RAT).

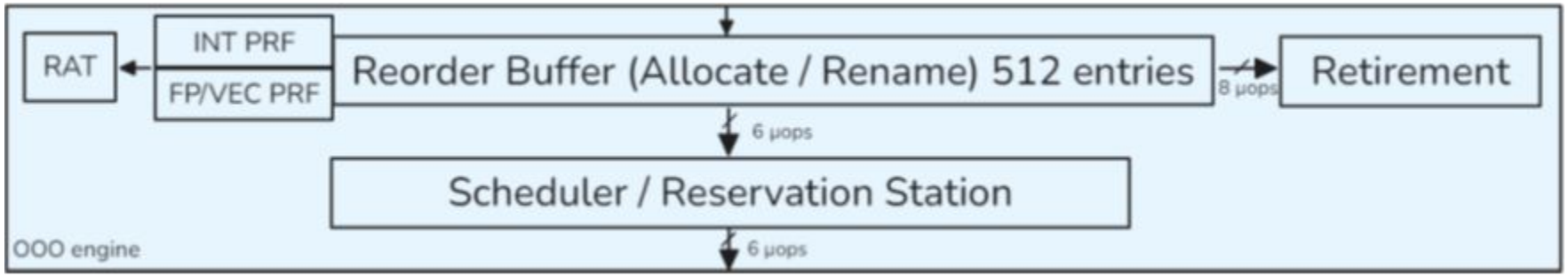
back-end

The ROB:

- allocates execution resources. When an instruction enters the ROB, a new entry is allocated and resources are assigned to it, mainly an execution unit and the destination physical register. The ROB can allocate up to 6 *μops* per cycle.

- The ROB tracks speculative execution. When an instruction has finished its execution, its status gets updated and it stays there until the previous instructions finish. (instructions must retire in program order). The ROB can retire 8 instructions per cycle.

back-end



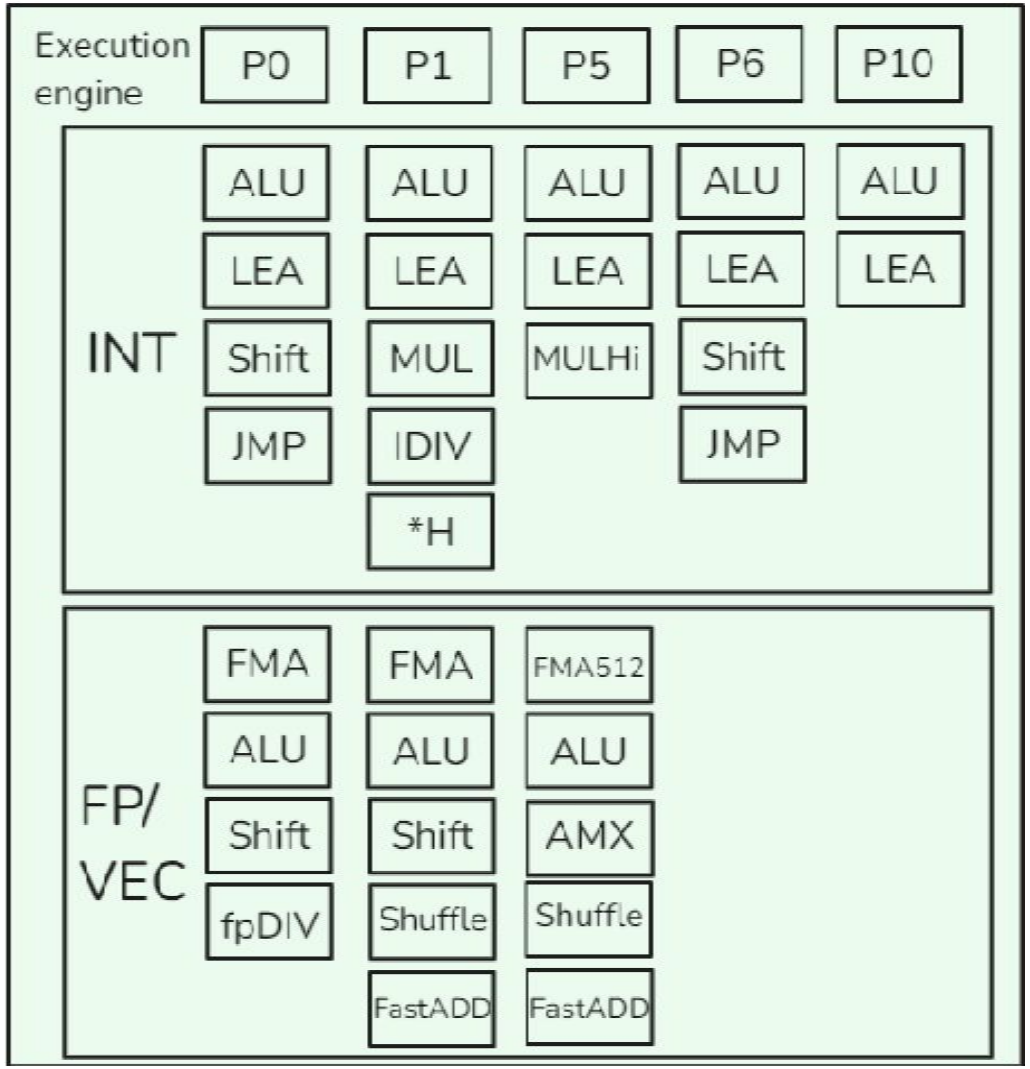
The **Reservation Station (RS)** is the structure that tracks the availability of all resources for a given *µop* and dispatches the *µop* to an execution port once it is ready.

An **execution port** is a pathway that connects the scheduler to its *execution units*. Each execution port may be connected to multiple execution units. When an instruction enters the RS, the scheduler starts tracking its data dependencies. Once all the source operands become available, the RS attempts to dispatch the *µop* to a free execution port. The RS can dispatch up to 6 *µops* per cycle.

execution engine

There are 12 execution ports:

- 5 ports provide integer (INT) operations,
- 3 of them handle floating-point and vector (FP/VEC) operations.
- Instructions that do not require memory operations are handled by will be dispatched to the execution engine (ports 0, 1, 5, 6, and 10). Some instructions may require two μops that must be executed on different execution ports, e.g., load and add.



Load Store Unit

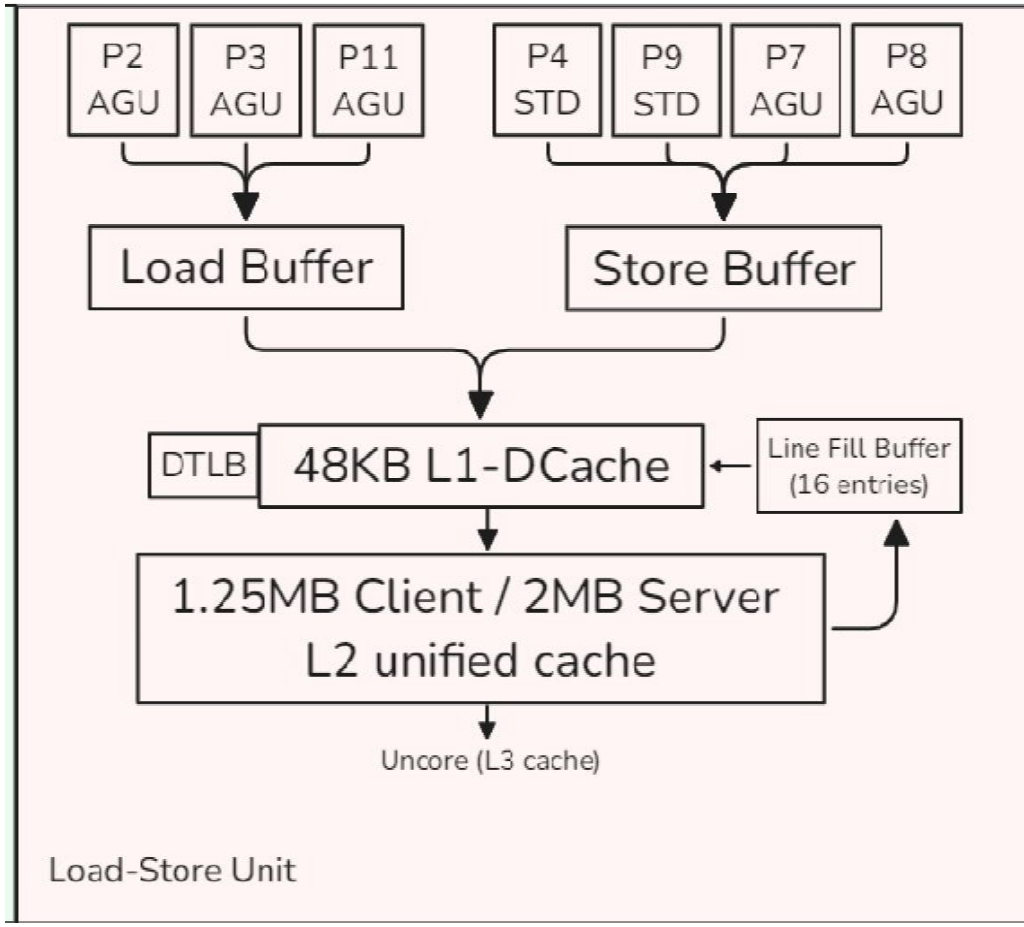
The **Load-Store Unit (LSU)** is responsible for operations with memory. The Golden Cove core can issue up to

- 3 loads (three 256-bit or two 512-bit) by using ports 2, 3, and 11.
- It can also issue up to two stores per cycle via ports 4, 9, 7, and 8.
- AGU (Address Generation Unit) is required for both load and store operations to perform dynamic address calculation.

For example, in the instruction

```
vmovss DWORD PTR [rsi+0x4],xmm0,
```

the AGU will be responsible for calculating `rsi+0x4`, which will be used to store data from `xmm0`.



Load Store Unit

LSU has a **Load Buffer** (also known as Load Queue) and a **Store Buffer** (also known as Store Queue).

- While the data sits in the Store Buffer waiting to be written, other load instructions can read the data straight from the store buffers *(store-to-load forwarding)**
- *non-temporal* memory accesses: if we execute a partial store (e.g., we overwrite 8 bytes in a cache line), we need to read the cache line first. However, if we know that we won't need this data again, then it would be better not to allocate space in the cache for that line. Non-temporal memory accesses are special CPU instructions that do not keep the fetched line in the cache and drop it immediately after use.

* forwarding from a store to a load occurs in real code quite often. In particular, any code that uses read-modify-write accesses to its data structures is likely to trigger these sorts of problems.

Weakly ordered memory model

For optimization purposes, the processor can reorder memory read and write operations.

Consider a situation when a load runs into a cache miss and has to wait until the data comes from memory. The processor allows subsequent loads to proceed ahead of the load that is waiting for data. This allows later loads to finish before the earlier load and doesn't unnecessarily block the execution.

- Just like with dependencies through regular arithmetic instructions, there are memory dependencies through loads and stores:

```
Load R1, MEM_LOC_X
Store MEM_LOC_X, 0
```

- Memory barriers (fences) are required when a strict order matters.

Performance Monitoring Unit

Performance Monitoring Unit

Modern CPU provides facilities to monitor performance, which are combined into the Performance Monitoring Unit (PMU). This unit incorporates features that help developers analyze the performance of their applications:

- **PMC (Performance Monitoring Counter)**: hardware counters that record the number of specific events (e.g., cycles, cache misses, branch mispredictions).
- **LBR (Last Branch Record)**: a stack that logs the source and target addresses of the most recent taken branches.
- **PEBS (Precise Event-Based Sampling)**: an Intel feature that delivers accurate samples of events precise instruction pointers.
- **PT (Processor Trace)**: hardware tracing that records a compressed stream of control-flow information for near-complete program execution reconstruction.

Performance Monitoring Counters

PMC are hardware register implemented as a Model-Specific Register (MSR).

- PMCs are accessible via the RDMSR and WRMSR instructions

- The number of counters and their width can vary from model to model.
  - For example, in the Intel Skylake architecture has 3 fixed and 8 programmable counters. The three fixed counters are set to count core clocks, reference clocks, and instructions retired.

Typically, PMCs are 48-bit wide

- When the value of PMCs overflows, the execution of a program must be interrupted.

Performance Monitoring Events

PMU provides more than 100 events available for monitoring.monitoring.

- It’s not possible to count all the events at the same time, but analysis tools solve this problem by multiplexing between groups of performance events

Maing groups are:

- CPU Core / Execution (cycles, instructions, stalls)
- Branching / Control Flow (mispredictions, branches)
- Caches & Memory (cache and TLB miss/hit)
- Floating-Point / SIMD (instructions, FLOPS)

Biblio

Hennessy, John L., and David A. Patterson. Computer architecture: a quantitative approach. Elsevier, 2011.

Denis, Bakhvalov, Performance Analysis And Tuning On Modern CPUs: Second Edition