# CS M151B: Computer Systems Architecture

Week 9

## Memory

- Last week, we covered caches and why it was necessary to have them.
- Recap: We need DRAM/Main Memory since we need some “large memory” in order to adequately contain our program's addressable space.
- Unfortunately, this large memory is too slow to be our primary source of satisfying memory accesses.
- Cache to the rescue.

## Memory

- We now have an adequate model for describing how a program is executed.
- Instruction addresses are fed into the processor and memory accesses generally must be satisfied by the cache.
- Memory addresses range from 32, 48, and 64 (?) bits. If we consider the 32 bit case, this implies that our memory must have a capacity of 2^32 bytes.

## Memory

- That suggests that main memory must be at least 2^32 or approximately 4 GB.
- But that can't be right, computers can function just fine with less memory than that.
- ...for that matter even if you had 4 GB of memory, how could you run multiple processes if each process expects to have 4 GB of personal memory space.
- What if a computer was running 100 processes?

## Virtual Memory: Basics

- Addresses that are known to a program are actually virtual addresses in a scheme known as virtual memory.
- Each process expects to be the sole owner of 2^32 bytes of memory, but it would be impractical to have actually that much DRAM
- Simply allow each process to believe that they own the entire addressable space.

## Virtual Memory: Basics

- In reality, the physical memory must be shared between all of the processes.
- The actual contents of each process' memory must actually be stored on the much larger (but much slower) disk.
- Main memory thus, becomes a “cache” for the virtual memory on disk.

## Virtual Memory: Basics

- Virtual memory doesn't use “blocks”. It uses “pages”.
- ...which are pretty much the same idea (ways of dividing memory into contiguous chunks).
- When a process accesses memory, it uses a virtual address. If the physical memory contains the “page” that the virtual address belongs to, get the value from memory.
- If not, page fault, you'll have to find the page on disk, then copy it to physical memory.

## Virtual Memory: Basics

- Wait... so each process expects ~4 GB of memory which is stored on disk instead. That means if I have 100 processes, I need ~400 GB of disk space, just to store the memory spaces of all of the programs?

## Virtual Memory: Basics

- Not all pages are actually “allocated”. This means they don't exist anywhere. This allows processes to only take up as much space as they need.
- For example, if pages are 4KB and the program has not defined any use for memory address 0x01000000 to 0x01001000, then there's no need to have that page allocated in memory.

# Virtual Memory: Basics

- If physical memory is a “cache” for the virtual memory on disk, how does the processor know where the virtual page is?
- Physical memory is like a fully associative cache where any virtual page can belong to any spot in physical memory.

## Virtual Memory: Basics

- Like a cache, if main memory is full and you need to bring a different page from disk into main memory, some page is going to have to be a victim.
- Keep this in mind for later.
- When looking up a virtual page in physical memory, it would be obnoxious to have to search every page in physical memory.

## Virtual Memory: Page Table

- When we need to look through main memory for a virtual page, we don't want to have to manually search every page.
- Instead, we use a Page Table, where we will use (part of) the virtual address to index into the table. Each entry in the page table will contain (part of) the physical address that it maps to.

## Virtual Memory: Page Table

- Consider a page size of ~4 KB or 2^12 bytes. If the addressable space is 32 bits. How many pages could there possibly be in the virtual address space?

## Virtual Memory: Page Table

- Consider a page size of ~4 KB or 2^12 bytes. If the addressable space is 32 bits. How many pages could there possibly be in the virtual address space?
- 2^32 / 2^12 = 2^20 pages, each of size 2^12 bytes.

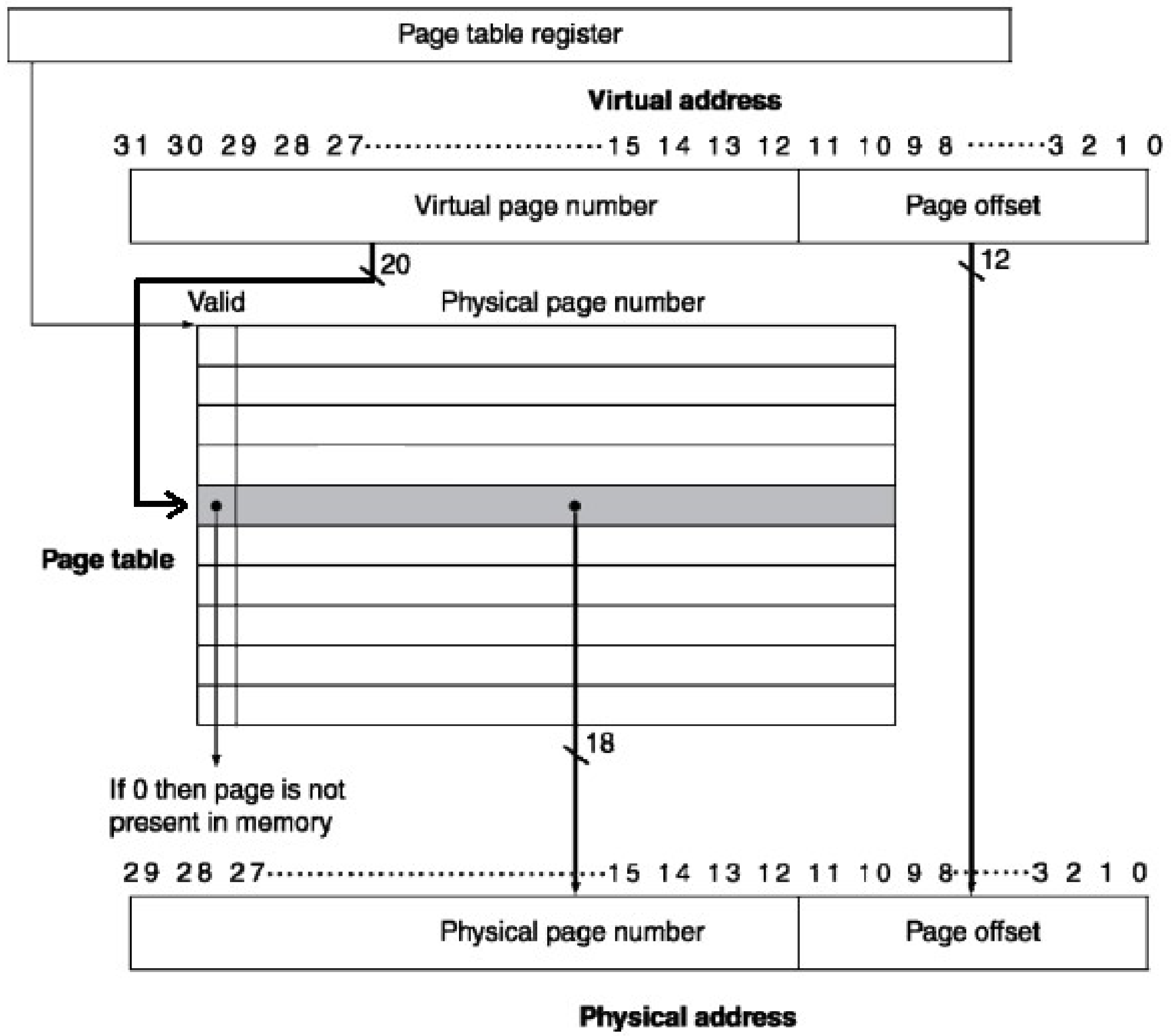
## Virtual Memory: Page Table

- Virtual Address decomposition.
- [ VPN ][ VPO ]
- VPN: Virtual Page Number
  - The index into the page table
- VPO: Virtual Page Offset
  - The byte offset into the page
- Index into the page table with the VPN. The value stored in the page table is the PPN or physical page number.

# Virtual Memory: Page Table

- Virtual Address decomposition.
- [ VPN ][ VPO ]
- Physical Address decomposition.
- [ PPN ][ PPO ]
- The VPO and PPO are same since page size are consistent in both virtual and physical address space.

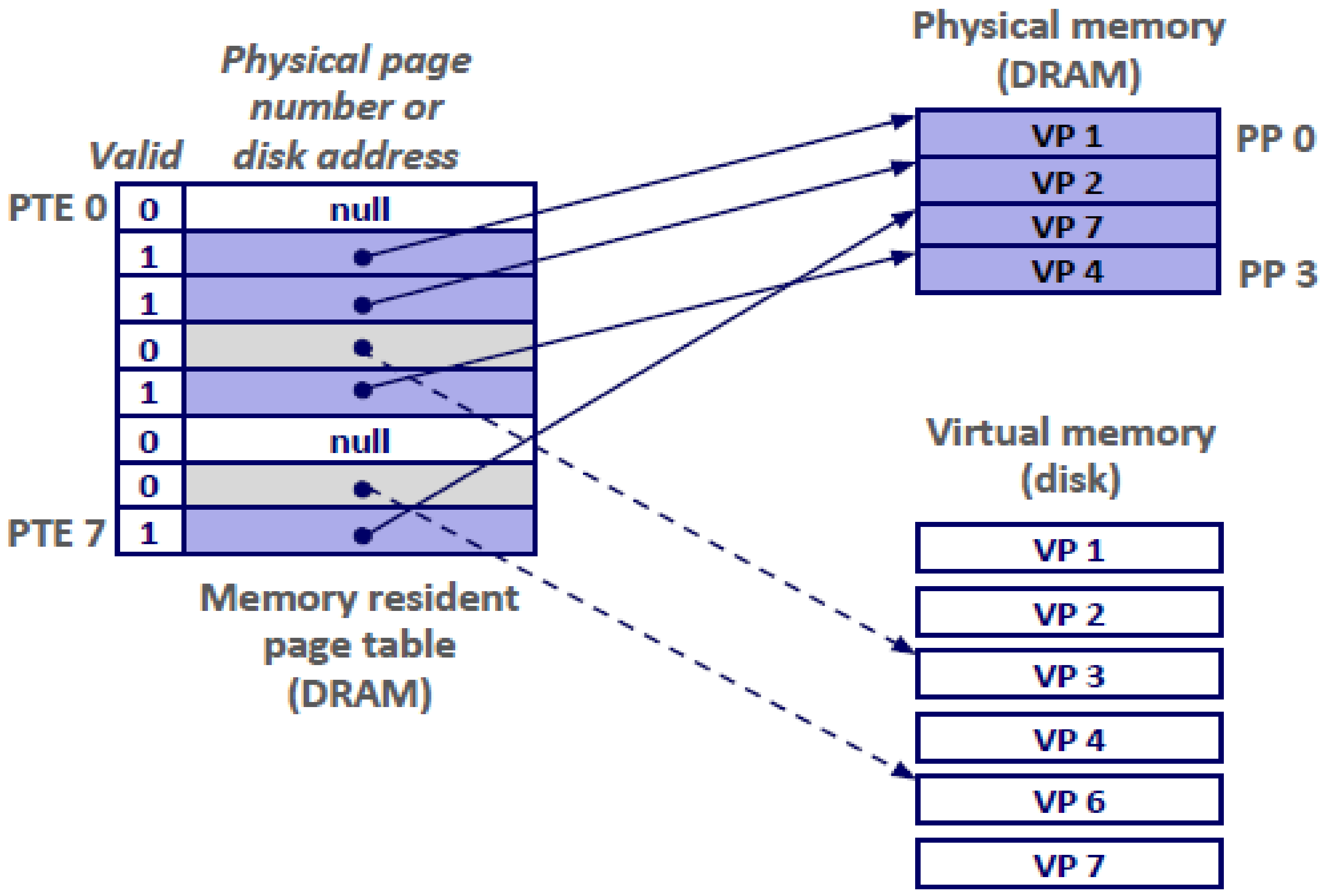
# Virtual Memory: Page Table



## Virtual Memory: Page Table

- Consider 8 bit virtual addresses with a page size of 2^5 bytes. This mean 5 bits for the virtual page offset and 3 bits for the virtual page number.
- There are 2^3 pages in this addressable space and thus, the there are 8 page table entries indexed from 0 (000) to 7 (111).
- Each page table entry would contain the mapping to the physical address (PPN).

# Virtual Memory: Page Table



# Virtual Memory: Page Table

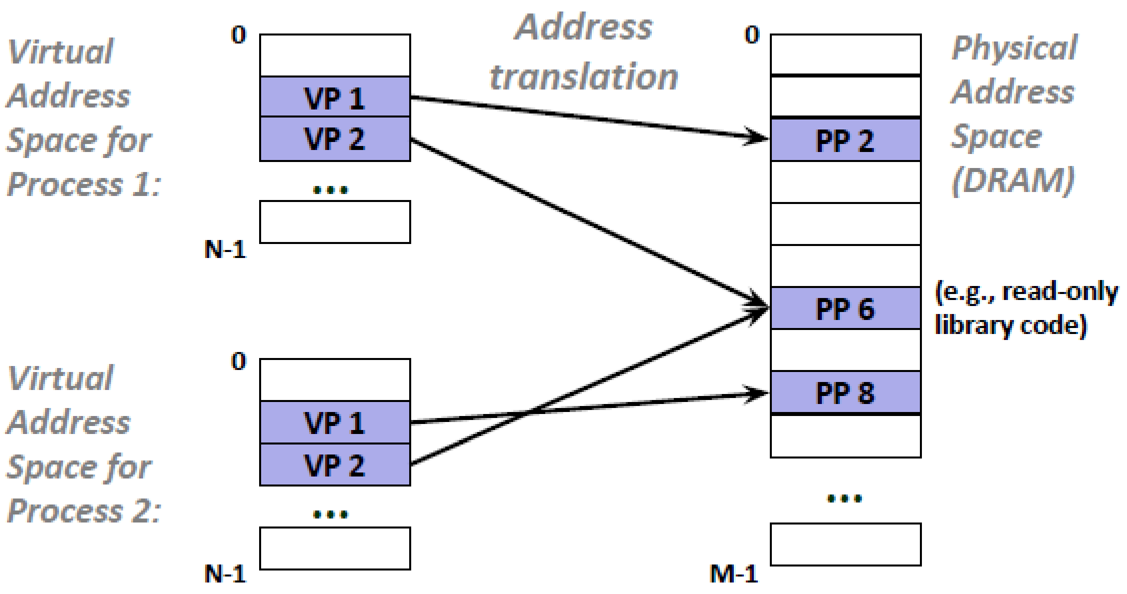
- Some notes:
- The Page Table is an entity stored in memory, it's not hardware.
- The physical address of each page table is stored in a register.
- Each process has its own page table.
- A page table entry will either indicate:
  - Virtual page is in memory at location (valid = 1) [PPN : VPO]
  - Virtual page is not in memory (valid = 0). Go to disk
    - Virtual page is either allocated in disk or unallocated.

# Virtual Memory: Page Table

- More notes:
- The page table look-up process inherently provides one of the other great benefits of virtual memory: isolation between processes.
- Processes can't see the memory of any other process by virtue of the fact that all processes have the same virtual address space.
- Ex. Process 1 has a variable at address 0x10. For process 2, address 0x10 refers to something completely different.

# Virtual Memory: Page Table

- More notes:
- Virtual memory paging, however, also allows for simple explicit memory sharing.
- If you want two processes to share some value of memory, have their page tables point to the same physical address.


Virtual Address Space for Process 1:
0
VP 1
VP 2
...
N-1
Address translation
0
PP 2
PP 6
PP 8
...
M-1
Physical Address Space (DRAM)
(e.g., read-only library code)
Virtual Address Space for Process 2:
0
VP 1
VP 2
...
N-1

## Virtual Memory: Page Table

- More notes:
- If a virtual page isn't in main memory, when the page is pulled in from disk, the page table of the process is updated.
- However, consider the case when we need to evict a page. The OS can choose to throw out any page belonging to any process.
- This means that evicting a page potentially means updating multiple or all page tables.

# Virtual Memory: Page Table

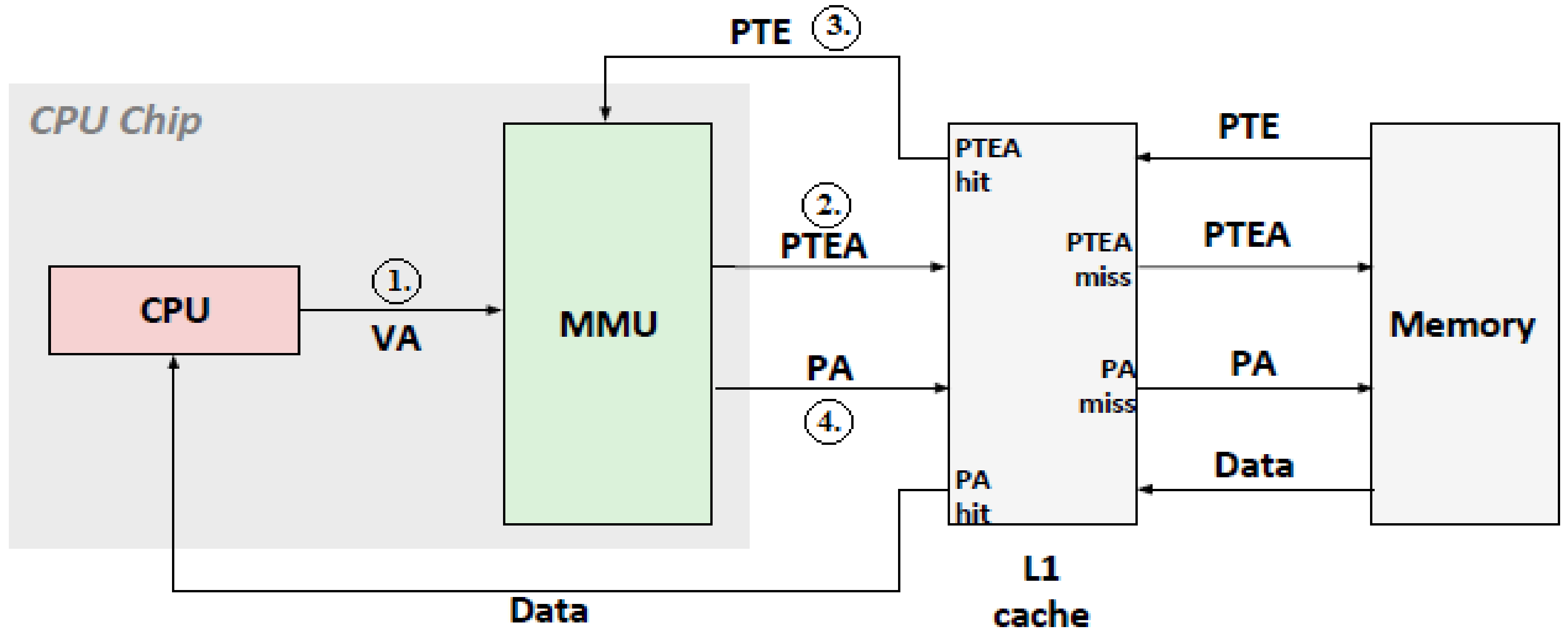
- More notes:
- Consider the previous example where the page size was 4KB.
- The page table consisted of 2^20 entries. Each entry contained the PPN + valid bit. Let's say PPN is 15 bits for 16 bits total
- The page table is then 2^20 * 2 bytes or 2^21 bytes. That's about 2 MB per process. That can be a pretty considerable amount of memory if you have many running processes.
- Use hierarchical page tables to help alleviate this problem.

## Virtual Memory: Page Table

- The fact that the page table is stored in memory should be a red flag.
- For every memory access, we actually need to make two memory accesses?!
  - One to access page table to find where the memory is actually located.
  - One to actually access memory.
  - Actually.
- What does this process look like?

# Virtual Memory: Page Table

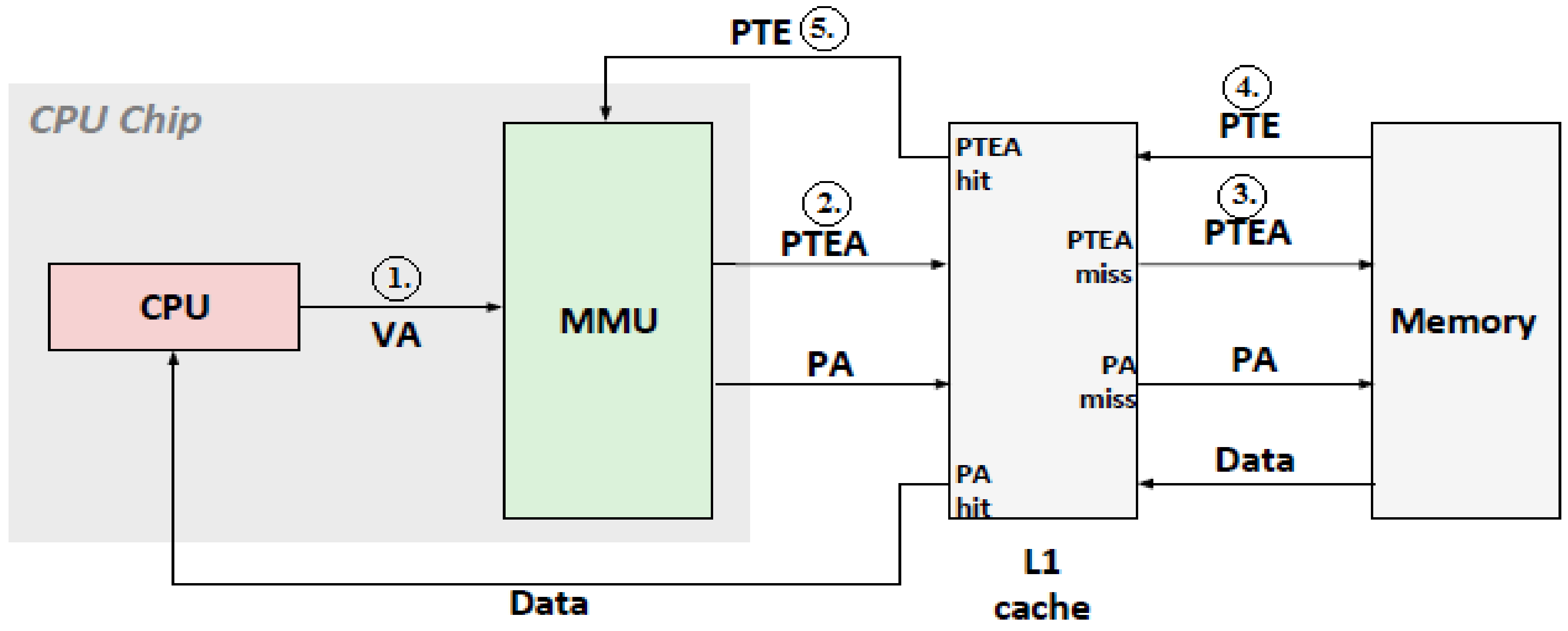
- Page Table Entry in cache



1. CPU issues virtual address to memory management unit.
2. MMU issues address of relevant page table entry to cache.
3. L1 cache return the page table entry to MMU.
4. MMU constructs physical address and sends it to cache.

# Virtual Memory: Page Table

- Page Table Entry **not** in cache



1. CPU issues virtual address to memory management unit.
2. MMU issues address of relevant page table entry to cache.
3. Page table entry not in cache. Cache passes page table entry address to memory.
4. Memory sends page table entry to cache. PTE is cached in... cache.
5. Cache sends page table entry to MMU... and so on.

## Virtual Memory: Page Table

- Because the page table is simply a data structure in memory, it can be cached just like anything else, sharing the instruction or data cache.
- But recall the mantra of this class:

## Virtual Memory: Page Table

- Because the page table is simply a data structure in memory, it can be cached just like anything else, sharing the instruction or data cache.
- But recall the mantra of this class:

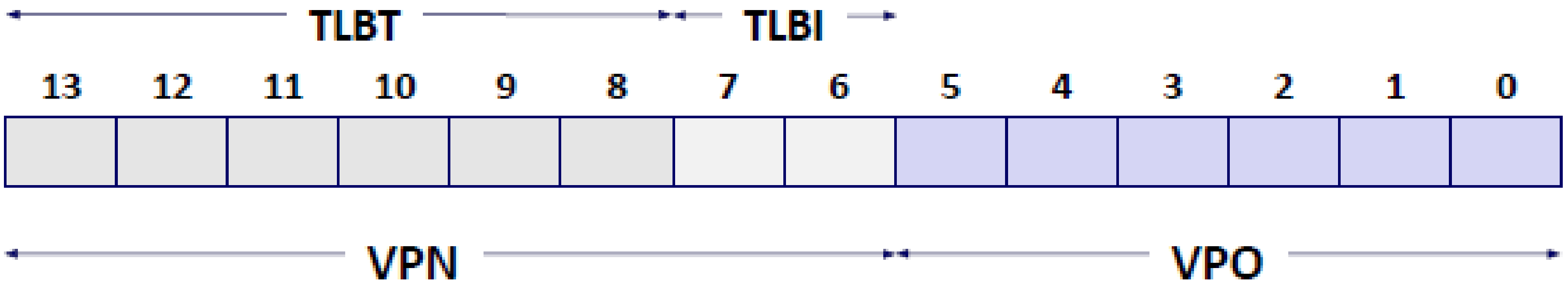
“No, actually that's not good enough”

- Every CS M151B lecture

## Virtual Memory: TLB

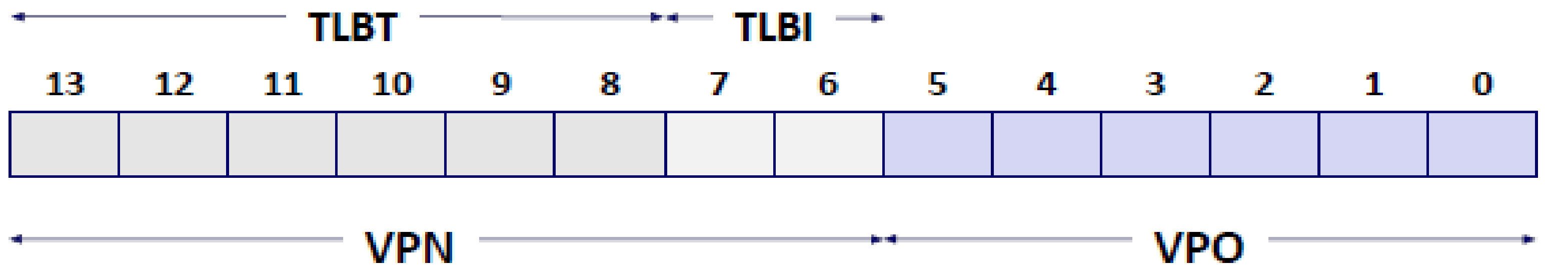
- We would like a specialized cache that will only hold page table entries.
- Enter... the translation lookaside buffer.
- Instead of having to go to the cache and then memory to get the page table, lets have a special cache for just the page table.

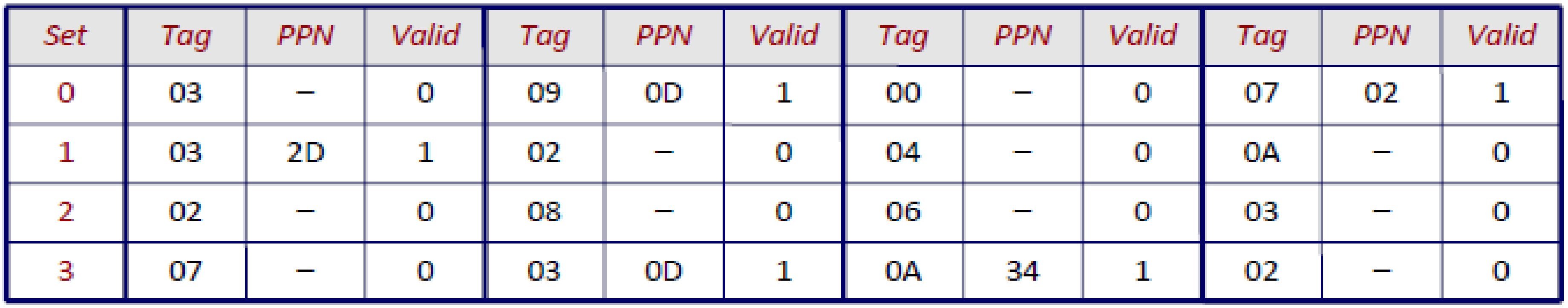
# Virtual Memory: TLB



- Accessing the TLB is the same as accessing a cache; the (virtual) address is split into three components Tag (TLB), Index (TLB), and Offset (VPO).
- Instead of memory data, the TLB simply contains the page table entry corresponding to that virtual address.

# Virtual Memory: TLB



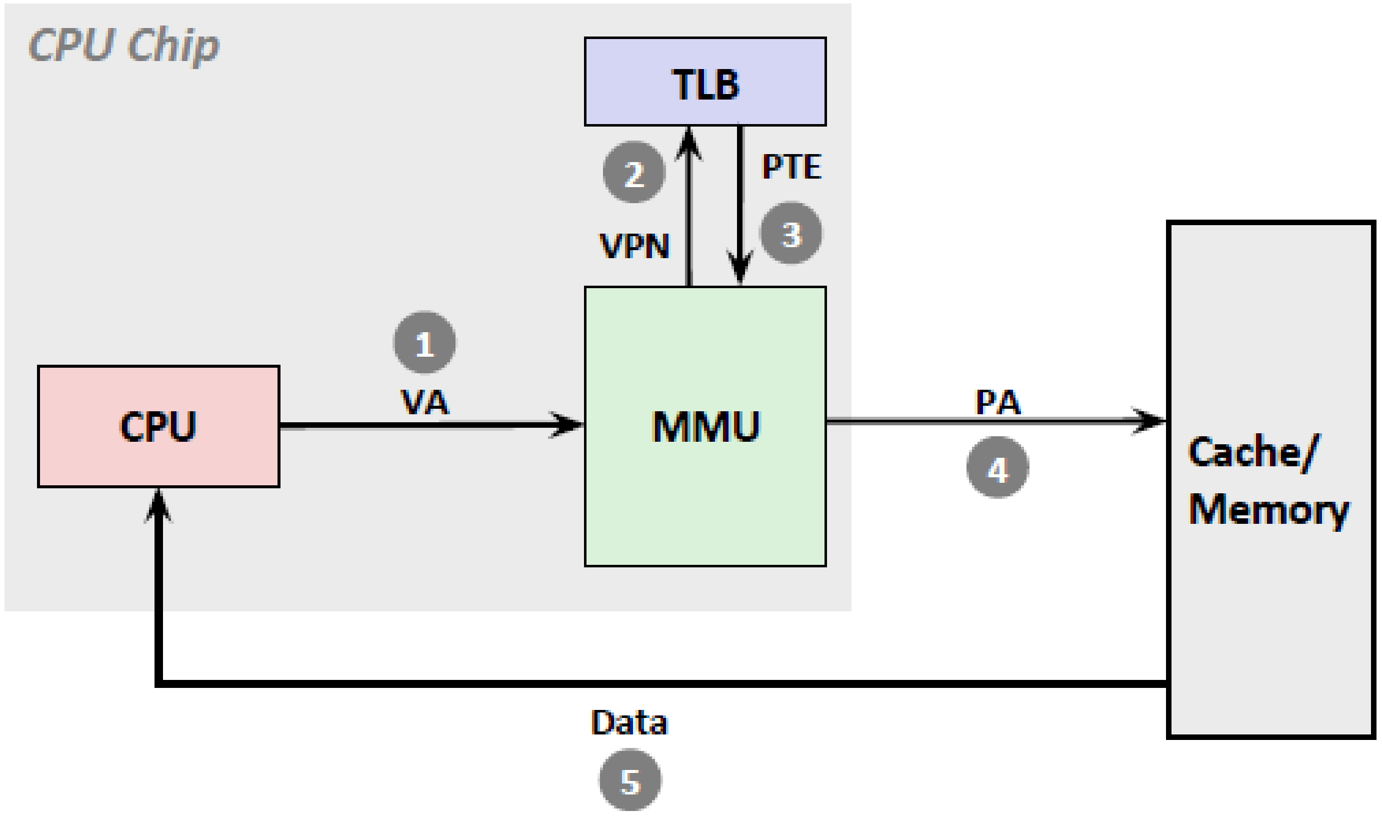


| Set | Tag | PPN | Valid | Tag | PPN | Valid | Tag | PPN | Valid | Tag | PPN | Valid |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 03 | – | 0 | 09 | 0D | 1 | 00 | – | 0 | 07 | 02 | 1 |
| 1 | 03 | 2D | 1 | 02 | – | 0 | 04 | – | 0 | 0A | – | 0 |
| 2 | 02 | – | 0 | 08 | – | 0 | 06 | – | 0 | 03 | – | 0 |
| 3 | 07 | – | 0 | 03 | 0D | 1 | 0A | 34 | 1 | 02 | – | 0 |

- Each entry contains the tag, a valid bit, and a PPN.
- At a high level, how does this look?

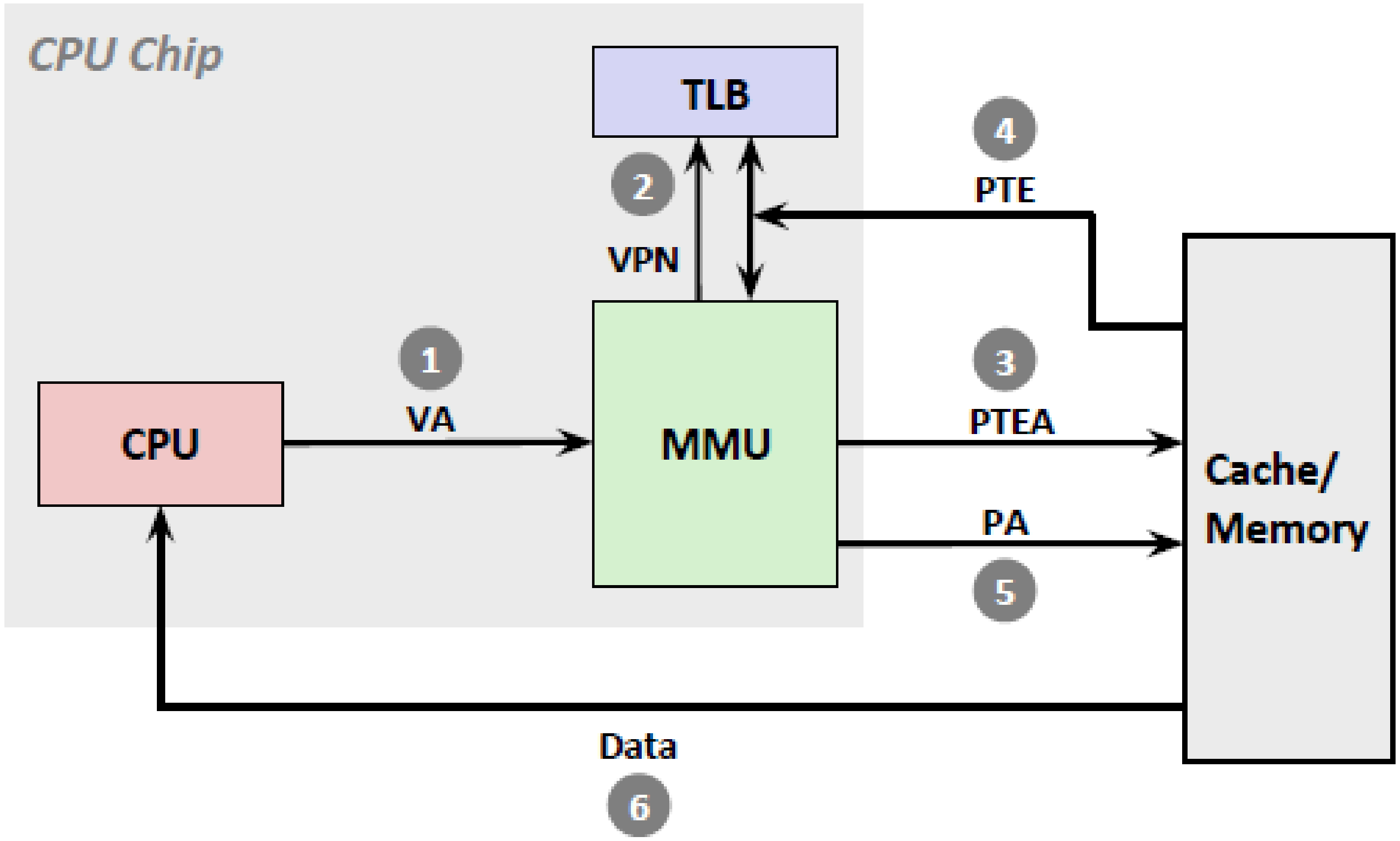
# Virtual Memory: TLB

- TLB Hit (the page table is in the TLB)
- Assume cache is PIPT



# Virtual Memory: TLB

- TLB Miss (the page table is not in the TLB)



## Virtual Memory: TLB

- Some notes:
- Unlike the page table, the translation lookaside buffer actually *is* a hardware component. This means there is a single TLB shared among each process run by the processor.
- TLB's are usually highly associative.
- “Translation lookaside buffer” is a silly name.

# Virtual Memory: Cache Indexing

- Now that we know that accessing memory is actually a mish-mash of various memory combinations, what addressing type matches each memory construct?
- Virtual Addresses correspond to the pages that are actually stored on disk or cached in memory. Notice that even on disk, the virtual address isn't the real location for the values. Thus, addresses are “virtual”.

## Virtual Memory: Cache Indexing

- We know how virtual pages are mapped to locations in physical memory. How are virtual pages mapped to locations on disk? That's for the paging supervisor to know.

## Virtual Memory: Cache Indexing

- What about the cache?
- In our previous examples, we would index into the cache using only physical addresses, but... it doesn't have to be this way...
- In fact we can have hybrid systems.

# Virtual Memory: Cache Indexing

- PIPT: Physically indexed physically tagged
  - Index into the cache using the physical address obtained from page table or TLB.
  - Compare the tag of cache with the tag of the physical address.
  - Simple, intuitive, straightforward.
  - Weakness?

# Virtual Memory: Cache Indexing

- PIPT: Physically indexed physically tagged
  - Weaknesses:
  - There is no choice but to first obtain the physical address that a virtual address maps to before accessing memory.
  - At best, this means going to the TLB first, then getting the physical address. At worst, you'll miss in the TLB and the cache, meaning an additional memory access.
  - (of course, the very worst is a page fault... but that's also true in other indexing methods)
  - Maybe let's cache based on virtual addresses

# Virtual Memory: Cache Indexing

- VIVT: Virtually indexed, virtually tagged
  - When you pull something from physical memory to the cache, insert it into the cache based on the virtual address that the processor originally issued.
  - Potentially much faster than PIPT. It's possible that the TLB and cache doesn't have the page table entry, but the relevant data could still be cached.
  - At best, CPU can immediately get data from cache without doing VA $\rightarrow$ PA mapping. At worst, same as PIPT.
  - Weaknesses?

## Virtual Memory: Cache Indexing

- VIVT: Virtually indexed, virtually tagged
- Weaknesses: Aliasing
  - Multiple processes have different virtual addresses that point to the same spot in physical memory (sharing memory).
  - If caching is based on virtual addresses each entry would be cached in different locations.
  - This is redundant (same physical location in multiple cache locations) and even worse could cause coherency problems if the cache is write-back.
  - If multiple processes tried to write and both processes had copies of the data in different locations of the cache, no process would see each others changes.

# Virtual Memory: Cache Indexing

- VIVT: Virtually indexed, virtually tagged
- Weaknesses: Homonym
  - Each process has the same addressable space range (ex. 0 to 2^32 – 1).
  - That means, each process would likely map a particular virtual address to different physical memory addresses.
  - Unless the cache associates the cache block with a particular process, this would be a problem.

# Virtual Memory: Cache Indexing

- VIVT: Virtually indexed, virtually tagged
- Weaknesses: What happens when a VA → PA mapping changes, for example, by eviction?
- This would mean a virtual address would not necessarily map to a particular block of data and existing cache lines which map a VA to data may be invalid.
- Additional complexity in having to flush cache lines when this happens.
- So it looks like VIVT isn't the way to go.
- But wait...

## Virtual Memory: Cache Indexing

- VIVT suffers from homonyms because a single cache is shared among multiple processes that index into it with the same address space.
- That TLB is also shared among multiple process that index into it with the same address space?
- Does it suffer from homonyms?
- Yup. In practice you can either flush the TLB after every context switch or maintain pid.

## Virtual Memory: Cache Indexing

- VIPT: Virtually indexed, physically tagged
  - A compromise between PIPT and VIVT.
  - Upon receipt of the VA, you can immediate start looking into the cache.
  - Simultaneously, you can look into TLB for physical address. This allows you to overlap the TLB access and cache access.
  - In the best case, the virtual address is used to index into the cache and the physical address is found in TLB, leading to VIVT-like latency.

# Virtual Memory: Cache Indexing

- VIPT: Virtually indexed, physically tagged
  - Does it suffer from VIVT problems?
  - It suffers from aliasing, but not homonyms. Why?

# Virtual Memory: Cache Indexing

- VIPT: Homonym safe?
- Consider the following:
  - Virtual address space: 8-bits
  - Physical address space: 7-bits
  - Page size: 16 bytes
  - Block size: 8 bytes
  - Cache size: 64 bytes
  - Cache associativity: direct mapped
- How would the virtual and physical addresses be interpreted as page numbers and page offsets?
- How would the addresses be split into cache tag, index, and offset?

# Virtual Memory: Cache Indexing

- VIPT: Homonym safe?
- VA : [ VPN: 4-bits ][ VPO: 4-bits ]
- PA: [ PPN: 3-bits ][ PPO: 4-bits ]
- The cache has 8 sets. We index into the cache with the VA.
- VA cache: [ T: 2-bits ] [ I: 3-bits ][ O: 3-bits ]
- We use the index bits of the VA to get the correct set, but we compare tags with the PA which is retrieved from the TLB/Page Table.
- PA cache?: [ T: 1-bit ][ I: 3-bits ][ O: 3-bits]

# Virtual Memory: Cache Indexing

- If the PA was split as such, homonyms would be a problem. However, the PA does not need to maintain set bits. As a result, it is split as follows:
- PA cache: [ T: 4 bits ][ O: 3-bits ]
- Now homonyms will not be a problem.
- Note that this technically requires more bits to store as tag bits (4-bits) when compared to an equivalent PIPT (1-bits) or VIVT (2-bits).

## The story so far..

- We have covered a pretty robust view of Computer Architecture
  - Pipelined, multi-issue, superscalar processors
- We started out with a simple Single Datapath as a simple method of executing a set of instructions and improved upon the idea immensely.

## The story so far..

- Pipelining was introduced and refined to reduce the clock time and increase utilization.
- Then we thought about further reducing the ET by making the pipeline deeper.
- We also thought about increasing the IPC by introducing multi-issue pipelined processors by (ideally) completing more instructions per cycle.

## The story so far..

- Despite all of this, we've limited our thinking.
- In all of our tinkering, we've only really considered one thread of execution and one processor.
- In terms of memory (virtual memory), we've considered what it may take to run multiple processes concurrently, but we haven't really thought about how that's handled by the processor.
- Let's think about that now.

## The story so far..

- A quick recap of running concurrent processes and threads on a single CPU.
- What pieces of hardware are “reserved” by a process or thread.
- That is, what does a thread or process think that it owns when it is executing?

# The story so far..

- PC to keep track of current execution.
- Registers for computations.
- A full 2^32 (48, 64, etc) addressable space. Each process believes it owns the whole addressable space. The threads of a program share the virtual address space of a single process.
- Each thread has it's own call stack.
- ...and of course, the CPU itself.

## The story so far..

- Realistically hundreds of threads and processes are expected to run concurrently.
- This is handled via context switching.
- Only one thread/process actually has control of the CPU at any point in time. The OS schedules which thread or process to run.
- Context switching will require storing and restoring registers (may require flushing cache or TLB). This is expensive.

## The story so far...

- Recall multi-issue processors
- Static multi-issue is known as VLIW and dynamic multi-issue is known as “superscalar”.
- This allows us to exploit ILP. What's the difficulty of trying continue to exploit ILP?

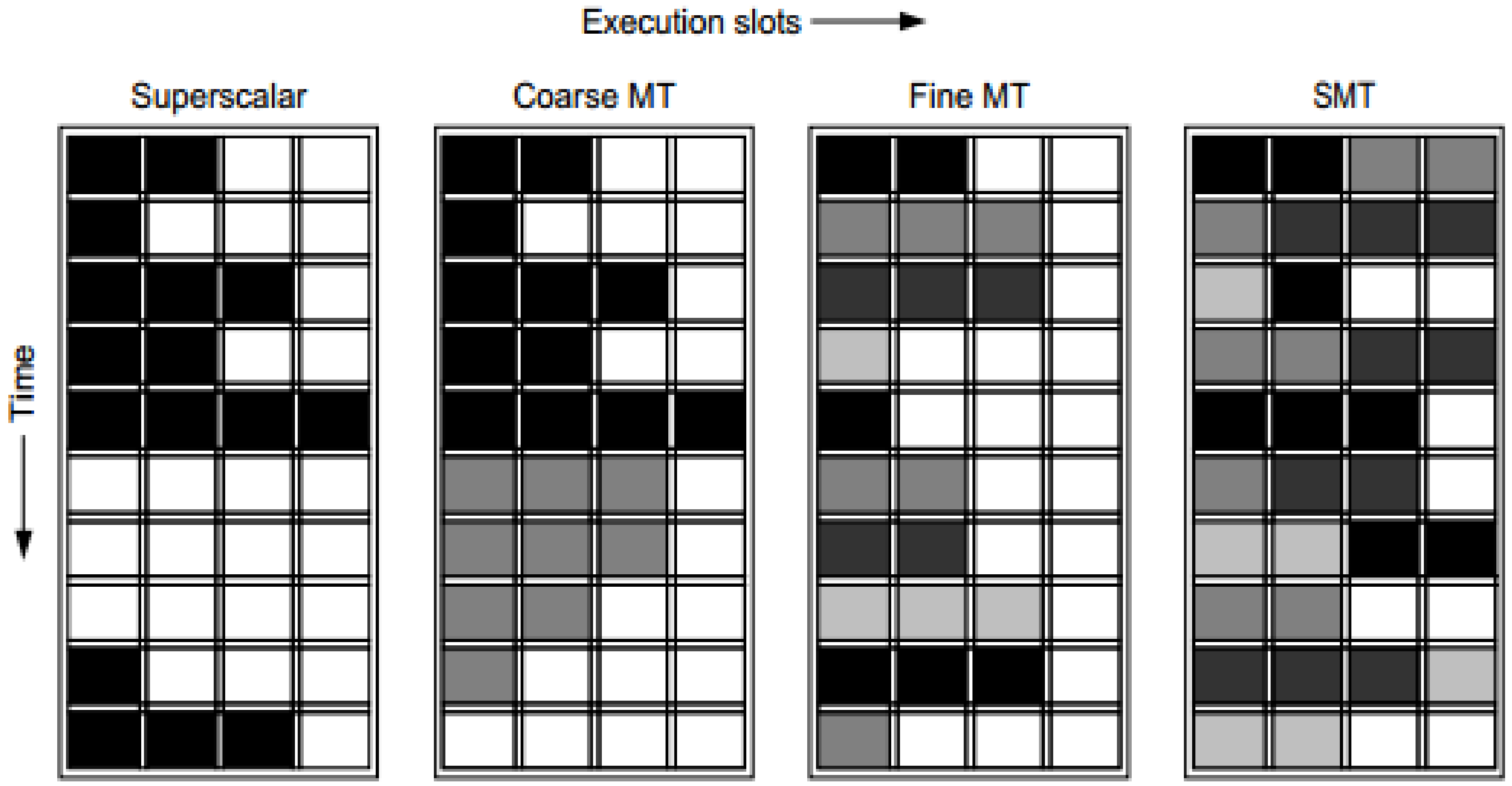
## The story so far...

- There are limits to how much ILP can be squeezed out of code. If it's inherently serial, there's no room for improvement in terms of ILP.
- The tradeoffs strike. As IPC increases, CT does as well. Diminishing returns.
- Ultimately, when you have a highly depended upon memory access, you've got no choice but to stall.

## The story so far...

- Memory access is extremely costly and potentially means stalling a thread for a long time.
- However, in a real system, we're running more than one thread.
- When we stall for a memory access, we can hide the latency by switching to another thread.
- However, OS-controlled context switching is slow. Ideally we want to do this in hardware.

# Multithreading



•

## Multithreading

- Note: In this image, there are four “lanes” (theoretical IPC of 4)
- Superscalar: Only switch contexts according to OS.
- Multi-threading requires additional register sets and hardware thread scheduling.
- Coarse-grain MT: Only switch contexts during long latency (page fault, memory miss).

## Multithreading

- Fine-grain MT: Switch threads after every cycle. Can hide latency of memory accesses AND data/control hazard stalls.
- Coarse and fine grain MT both eliminate “vertical waste” (unused clock cycles), but don't resolve “horizontal waste” (unused issue slots per cycle)
- Finally... Simultaneous Multithreading: Interleave the executions of multiple threads into the issues slots of individual cycles.

# Multithreading

- ...sounds like a clear winner, but what are the tradeoffs of each?
- Fine grain multithreading can hide latencies of memory access, data and control hazards, but at the cost of single thread performance. A thread might be ready to go, but must artificially wait for additional threads.

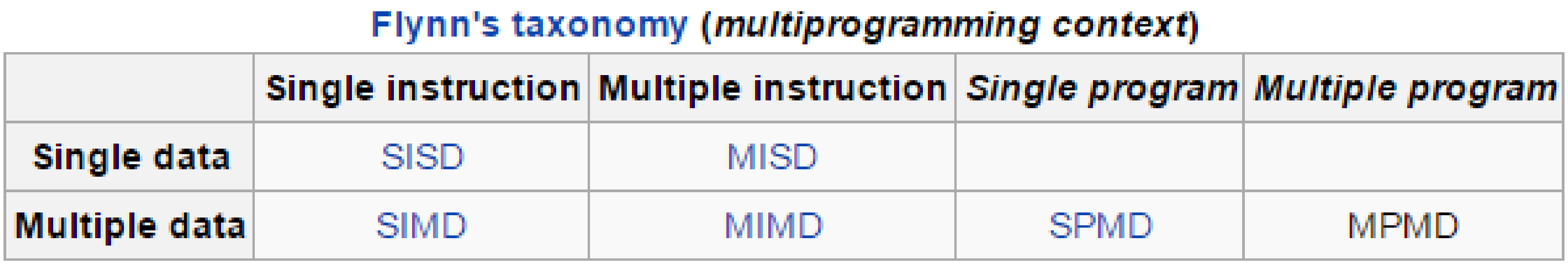
## Multithreading

- Coarse grain multithreading was actually developed as an alternative to fine-grain multithreading that doesn't sacrifice the performance of a single thread.
- Unlike fine-grain, coarse grain issues instructions from a single thread by default. Switching to the other thread incurs a start-up penalty.
- As a result, coarse grain multi-threading is ineffective in hiding shorter stalls.

# Multithreading

- What about SMT?
- Performance gains via SMT have been modest.
- Like fine-grain, can hurt single thread performance.
- In general, more threads means worse cache performance.
- Instead of increasing the complexity of a single processor to do advanced SMT, moving more towards having more cores that are simple.

# Flynn's Taxonomy



Flynn's taxonomy (*multiprogramming context*)

| | Single instruction | Multiple instruction | *Single program* | *Multiple program* |
|---|---|---|---|---|
| **Single data** | SISD | MISD | | |
| **Multiple data** | SIMD | MIMD | SPMD | MPMD |

- A classification of different types of parallel computing.

## SISD

- Single instruction, single data.
- Basically everything we've considered in this class, pipelined and superscalar processors and logic that operates on a single instruction stream.
- Simple and straightforward.
- Focus on instruction level parallelism

## MISD

- Multiple Instruction Single Data
- Only theoretical
  - ...because it's probably only useful to fill out the taxonomy grid.
- Not really useful to implement a machine that does MISD, especially when that sort of task could be done as easily on MIMD.
- Speaking of...

## MIMD

- Multiple Instruction Multiple Data
- A very general term that refers to different threads of instruction operating on different data. Independent tasks operating at the same time.
- A more viable way to improve thread level parallelism (or task-level parallelism) by using multi-processors. More on that later.
- Can be tightly coupled (multiple threads on a single program cooperating, SPMD) or loosely (clusters, cloud computing doing completely independent tasks).

## SIMD

- Single Instruction Multiple Data
  - Doing the same or nearly the same operations on multiple data. Exploits data-level parallelism
- Ex:

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

- This code will do 100 adds sequentially, but they could be done in parallel.

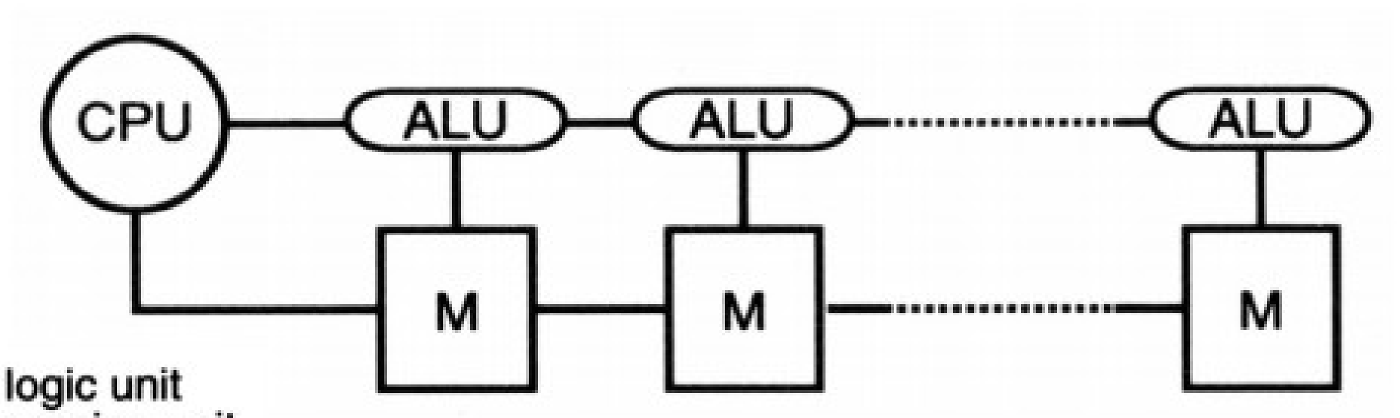
## SIMD

- Why SIMD when we can have more freedom with MIMD?
- SIMD exploits very fine-grain parallelism and in cases where SIMD is applicable (matrix operations, signal processing, graphics, vectors), SIMD can be efficient and optimized.
- Needs only one control unit since each data is operated upon by the same instructions.
- If appropriate, don't need to worry about coordination/synchronization.

# Parallel vs. Pipelined SIMD

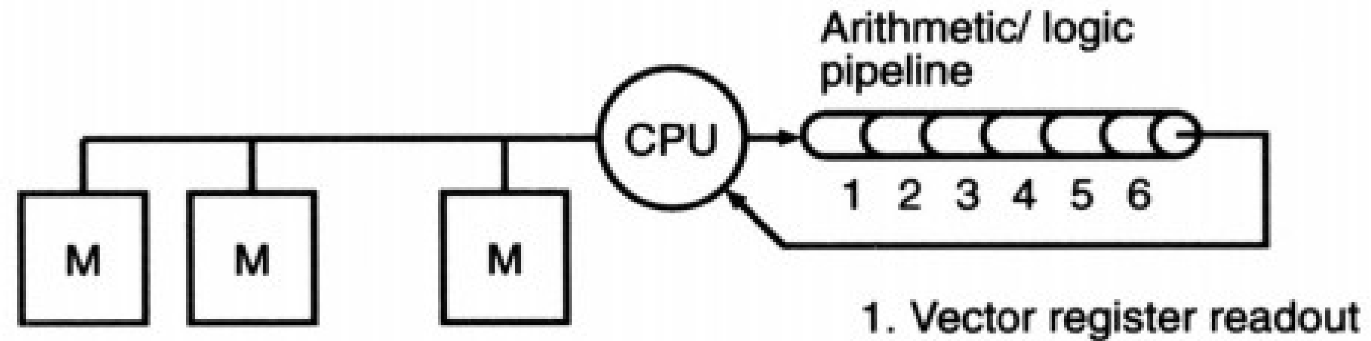
"true SIMD"

array computer



AU - Arithmetic/ logic unit
CPU - Central processing unit
M - Memory module

vector processor



1. Vector register readout
2. Compare exponents
3. Shift
4. Add
5. Normalize
6. Vector register write

## Vector Processors

- “True SIMD” is generally referred to as having multiple processing elements individually operating or accessing memory.
  - Ex. For processing elements 0 to 15, each processing element 'k' operates on element 'k' of some array.
- Vector processing is a mechanism for doing SIMD by streaming large vectors into a few highly pipelined processing elements.

## Vector Processors

- Vector processors might not be able to do things completely in parallel like “true SIMD”, but it can do things with simple hardware, simple memory layouts, and with few instructions.

## Vector Processors

- for(int i = 0; i < 64; i++)

  c[i] = a[i] + b[i]
- Rather than have to do 64 loads, 64 adds, and 64 stores, 64 branches, 64 increments, we can (potentially) do it in three:
  - lv (load vector)
  - addvv.d (add vector to vector, double precision)
  - sw (store vector)
- Clearly the main downside is in the bonkers instruction names (wtf is VFMSUBPD?)

# SIMD + Vector Processors

- SIMD and Vector processors work best when there is little divergence in instructions (ie, when all data are subjected to the same instructions)
- Diverging instructions is for MIMD.
- However, even in highly SIMD-esque code, there's going to be divergence.

## SIMD + Vector Processors

- Vector processors can handle divergence by using mechanisms such as bit mask registers.
- Given vector instructions of length 64 elements, have a 64-bit register where each bit corresponds to whether or not to do the operation on that element.
- Ex. If mask reg is 011110... means that you don't do the operation on the first (index 0) element and the sixth element (index 5) and etc.

## SIMD + Vector Processors

- True SIMD with multiple processing elements can handle divergence by running code that conditions on which processing element is running it.
- “If processing element id % 2 == 0, shut off processing element for next instruction”.
- Despite these workarounds, the more divergence, the less performance benefit you're getting.

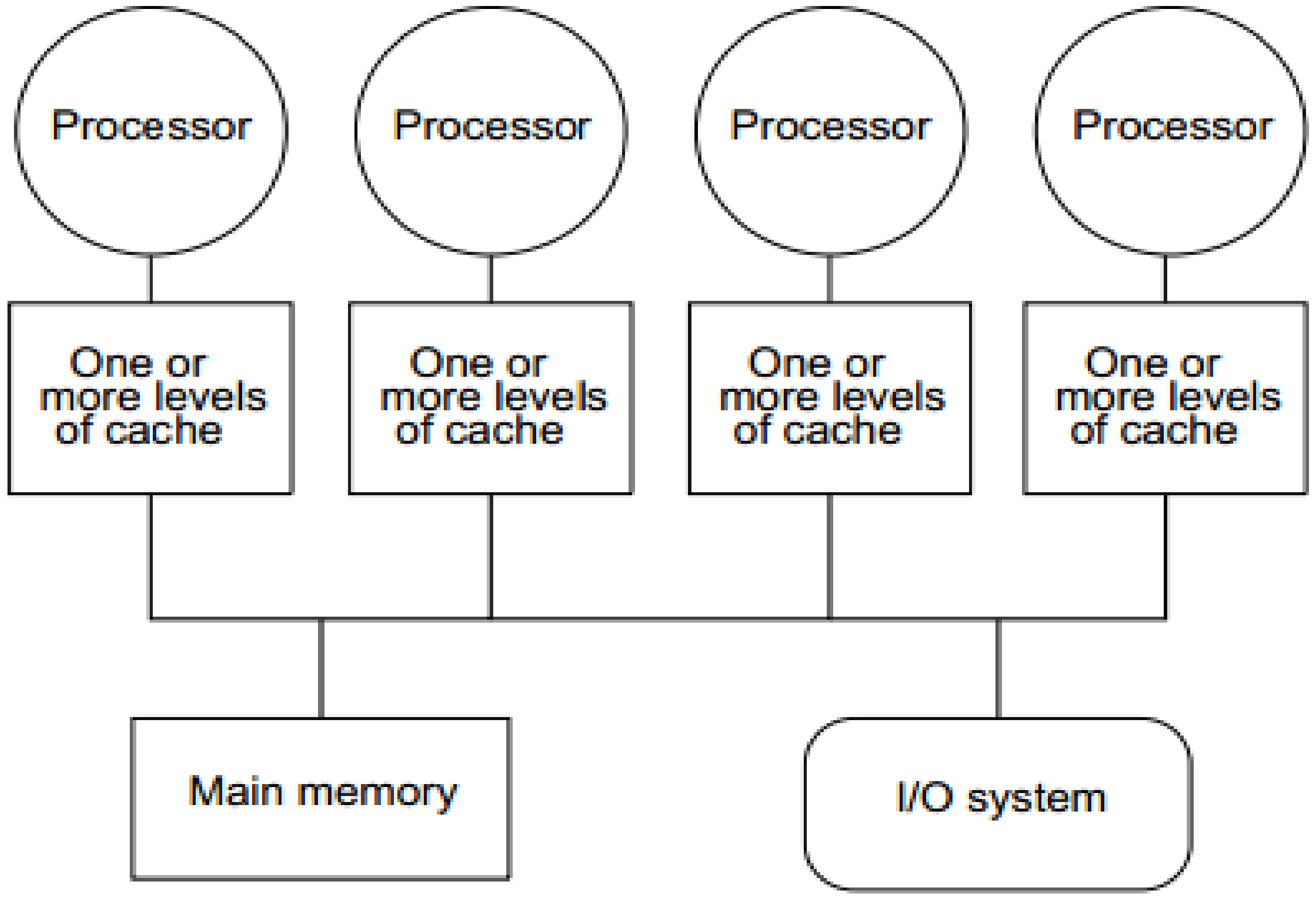
## Memory Systems

- SIMD and MIMD both have implementations that have multiple processing elements or multiprocessors that need to share data.
- This could be done by explicitly passing/copying data between one processing element to another.
- However, another way of doing this is by having a shared address space.

## Memory Systems

- There are many ways of implementing a shared address space but we'll focus on two:
- Shared memory, where each processor is physically connected to the same physical memory.
- Distributed memory, where each processor is responsible for a personal chunk of physical memory.

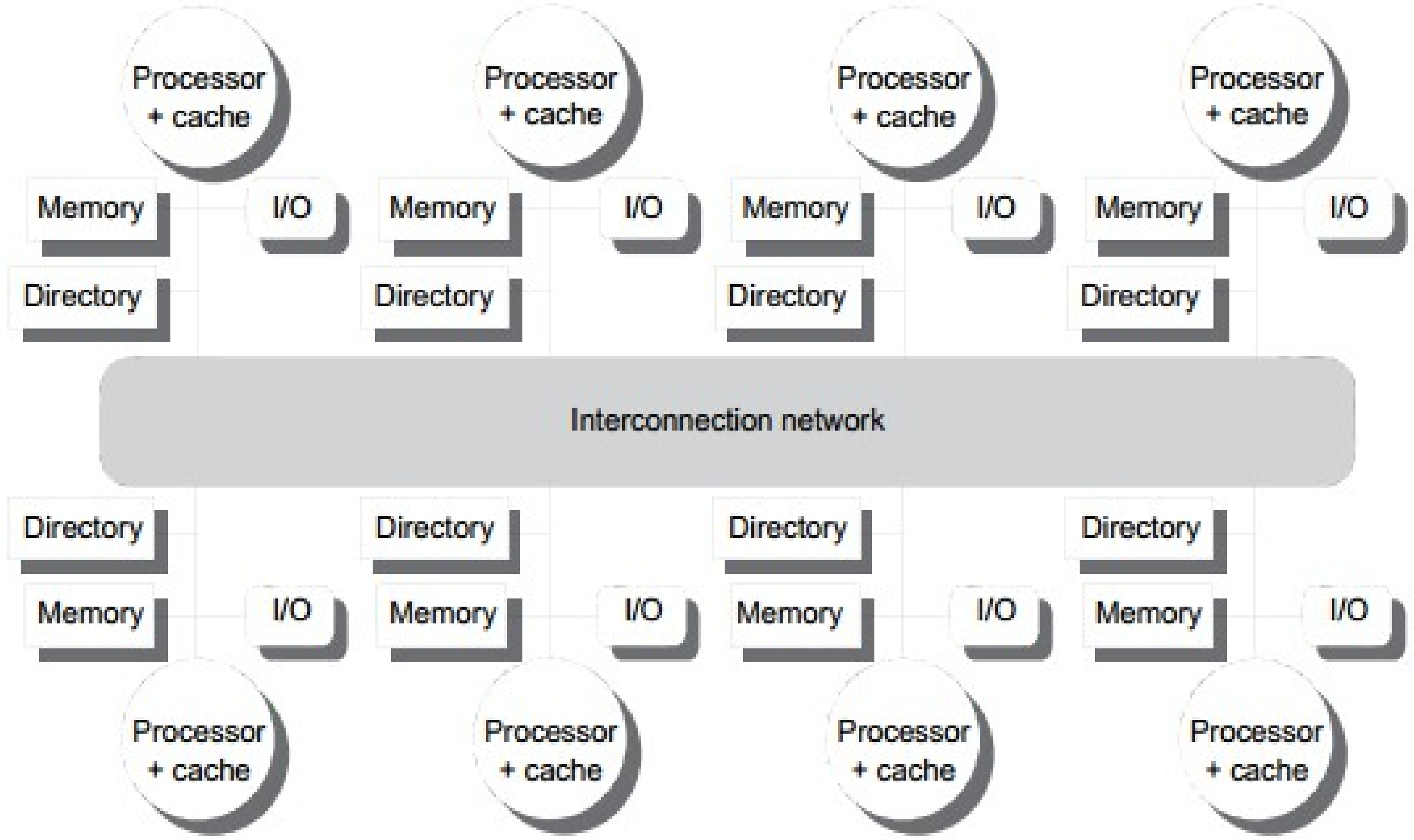
# Shared Memory


Processor
Processor
Processor
Processor
One or more levels of cache
One or more levels of cache
One or more levels of cache
One or more levels of cache
Main memory
I/O system

## Shared Memory

- Pros:
  - UMA: Uniform memory access. All processors have uniform latency to access memory.
  - Simple to actually share data. A processor writes to a location in memory. Another processor reads it.
- Cons:
  - The single shared memory could become a structural hazard or bottleneck if many processors want to access it.
  - Not very scalable, difficult to add another processor.
  - This generally requires a bus. A slow bus.

## Distributed Memory



# Distributed Memory

- Pros:
  - Reduce latency to local memory.
  - More scalable.
- Cons:
  - NUMA: Non-uniform memory access. Each processor accessing the local memory will be faster than accessing a non-local memory.
  - Have to explicitly communicate across the interconnection network to access shared information.
  - The interconnection network could be a bottleneck.

End of

# The Nine Week

-One Week Remains-