Advanced Computer Architecture

Department of Computing, Imperial College London

Chapter 4: Caches and Memory Systems

Part 4: hit time reduction – and address translation

November 2022

Paul H J Kelly

These lecture notes are partly based on the course text, Hennessy and Patterson's *Computer Architecture, a quantitative approach (3rd, 4th, 5th and 6th eds),* and on the lecture slides of David Patterson and John Kubiatowicz's Berkeley course

Average memory access time:

$$\text{AMAT} = \text{HitTime} + \text{MissRate} \times \text{MissPenalty}$$

There are three ways to improve AMAT:

1. Reduce the miss rate,
2. Reduce the miss penalty, or
3. Reduce the time to hit in the cache

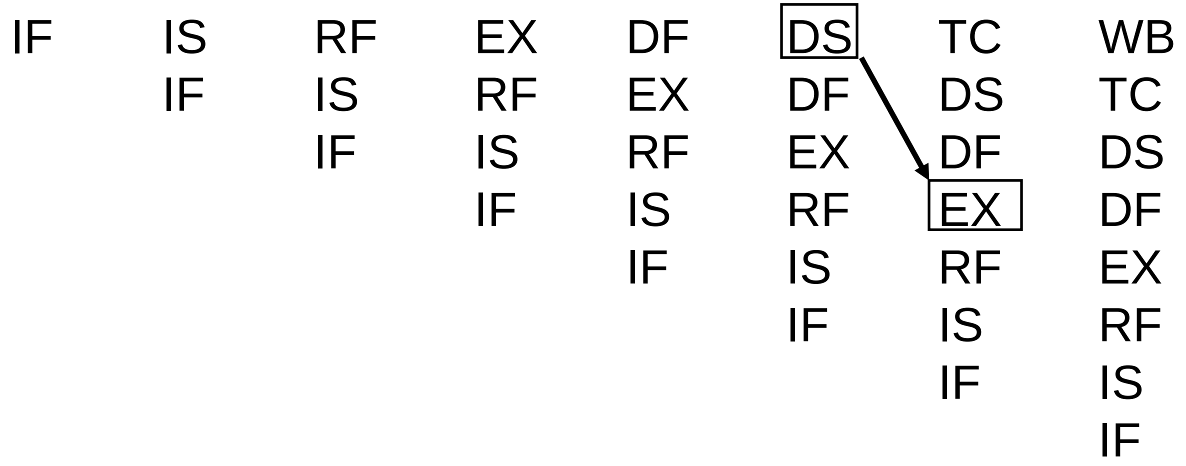
We now look at each of these in turn…

Fast Hits by pipelining Cache Case Study: MIPS R4000

- 8 Stage Pipeline:
  - IF–first half of fetching of instruction; PC selection happens here as well as initiation of instruction cache access.
  - IS–second half of access to instruction cache.
  - RF–instruction decode and register fetch, hazard checking and also instruction cache hit detection.
  - EX–execution, which includes effective address calculation, ALU operation, and branch target computation and condition evaluation.
  - DF–data fetch, first half of access to data cache.
  - DS–second half of access to data cache.
  - TC–tag check, determine whether the data cache access hit.
  - WB–write back for loads and register-register operations.
- What is impact on Load delay?
  - Need 2 instructions between a load and its use!

Case Study: MIPS R4000

TWO Cycle Load-use Latency between load instruction and arithmetic instruction

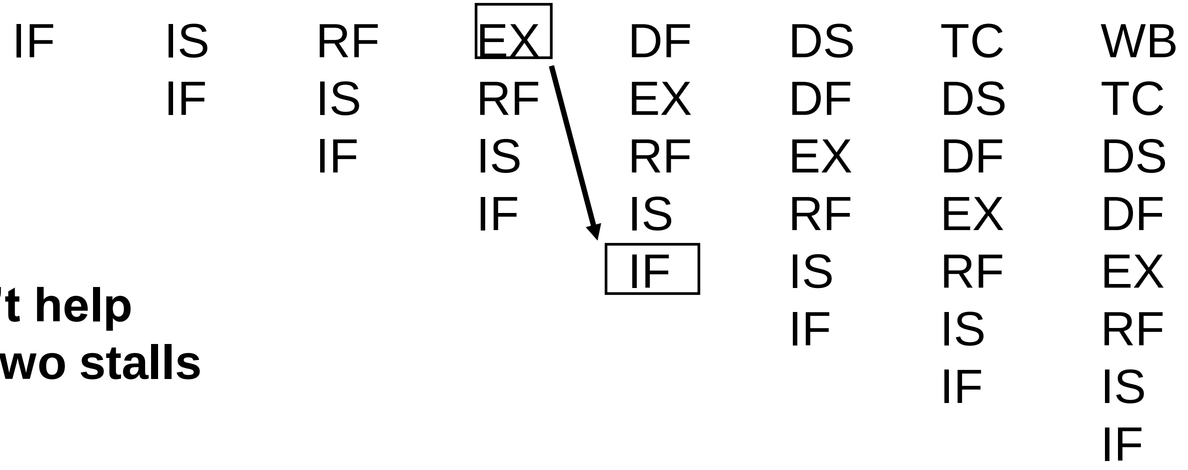


IF	IS	RF	EX	DF	DS	TC	WB
	IF	IS	RF	EX	DF	DS	TC
		IF	IS	RF	EX	DF	DS
			IF	IS	RF	EX	DF
				IF	IS	RF	EX
					IF	IS	RF
						IF	IS
							IF

THREE Cycle Branch Latency

(conditions evaluated during EX phase)

Delayed branch doesn't help much: delay slot plus two stalls



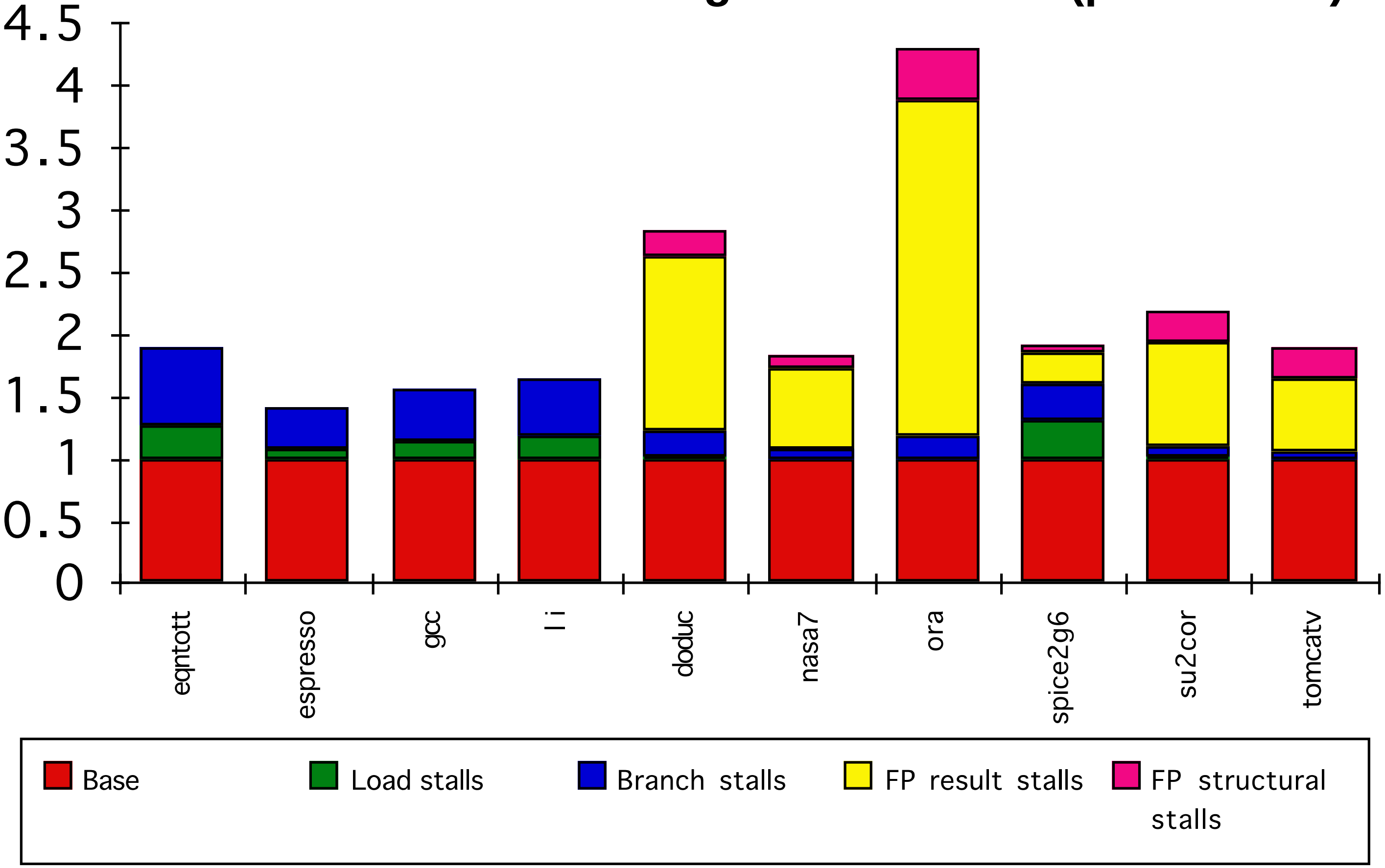
IF	IS	RF	EX	DF	DS	TC	WB
	IF	IS	RF	EX	DF	DS	TC
		IF	IS	RF	EX	DF	DS
			IF	IS	RF	EX	DF
				IF	IS	RF	EX
					IF	IS	RF
						IF	IS
							IF

- **Cache is *pipelined: 1* cache access per cycle, but with a 2-cycle access *latency***

  (Q: does this reduce AMAT?)

R4000 Performance

- Not ideal CPI of 1:
  - Load stalls (1 or 2 clock cycles)
  - Branch stalls (2 cycles + unfilled slots)
  - FP result stalls: RAW data hazard (latency)
  - FP structural stalls: Not enough FP hardware (parallelism)



R4000 was an in-order processor – this shows the potential importance of dynamically-scheduled “out-of-order” microarchitectures

MIPS next architecture was the o-o-o R10000

Cache bandwidth

H&P 6th ed pages 99-100

- What if we want to support multiple parallel accesses to the cache?
  - Divide the cache into several banks
  - Map addresses to banks in some way (low-order bits? Hash function?)

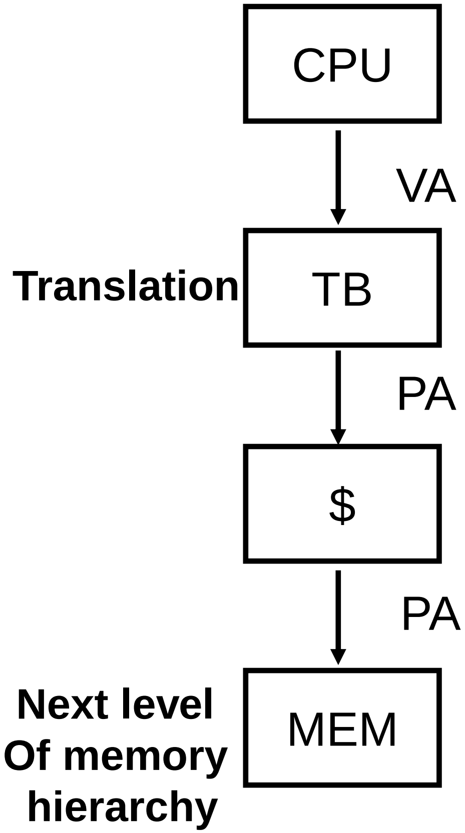
  - Other options are possible…
    - Duplicate the cache!
    - Multi-ported RAM: support two reads, to different addresses, every cycle
      - RAM array has two wordlines per row, and two bitlines per column
      - (See https://inst.eecs.berkeley.edu/~cs250/sp16/lectures/lec07-sp16.pdf slide 67)

Virtual memory, and address translation

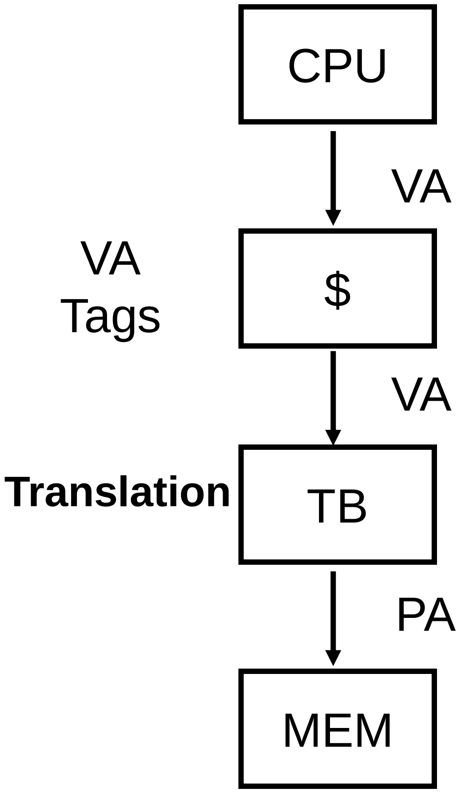
H&P 6th ed page B-36ff

- Simple processors access memory directly
  - Addresses generated by the processor are used directly to access memory
- What if you want to
  - Run some code in an isolated environment
    - So that if it fails it won’t crash the whole system
    - So that if it’s malicious it won’t have total access
  - Run more than one application at a time
    - So they can’t interfere with each other
    - So they don’t need to know about each other
  - Use more memory than DRAM
- An effective solution to this is to *virtualise* the addressing of memory
- By adding address translation

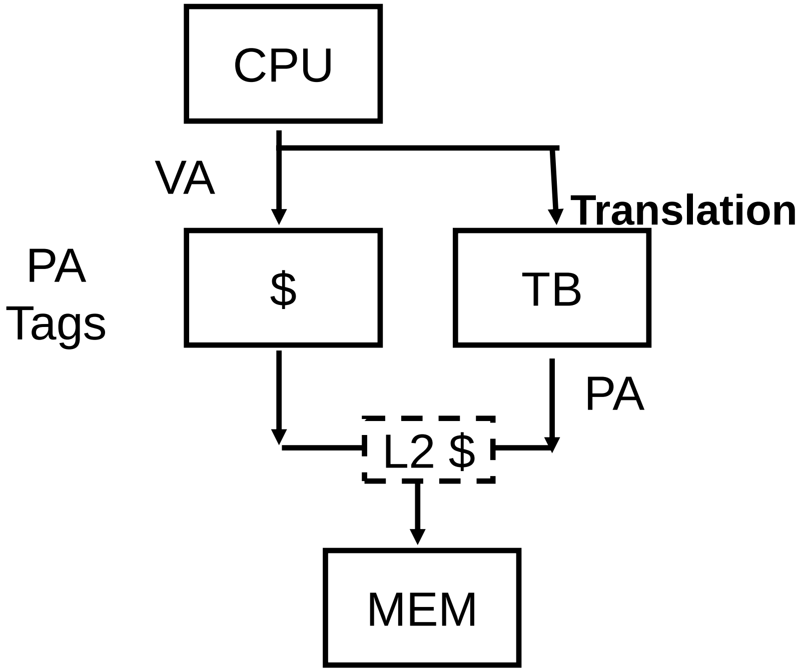
Fast hits by removing address translation from critical path



Physically-indexed, Physically-tagged (**PIPT**)



Virtually-indexed, virtually tagged (**VIVT**): Translate only on miss
Synonym/homonym problems



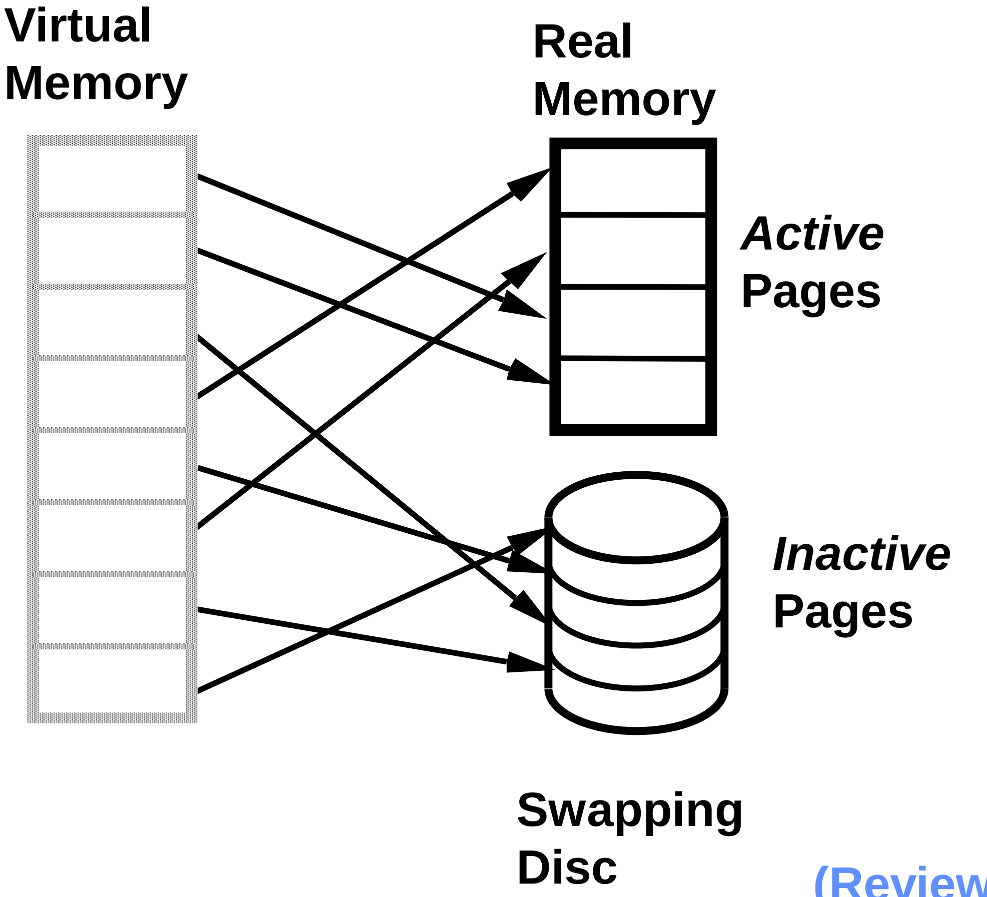
Virtually-indexed, physically-tagged (**VIPT**)
Overlap $ access with VA translation:
requires $ index to remain invariant
across translation

- ***CPU issues Virtual Addresses (VAs)***
- ***TB translates Virtual Addresses to Physical Addresses (PAs)***

Paging

Virtual address space is divided into ***pages*** of equal size.

Main Memory is divided into ***page frames*** the same size.



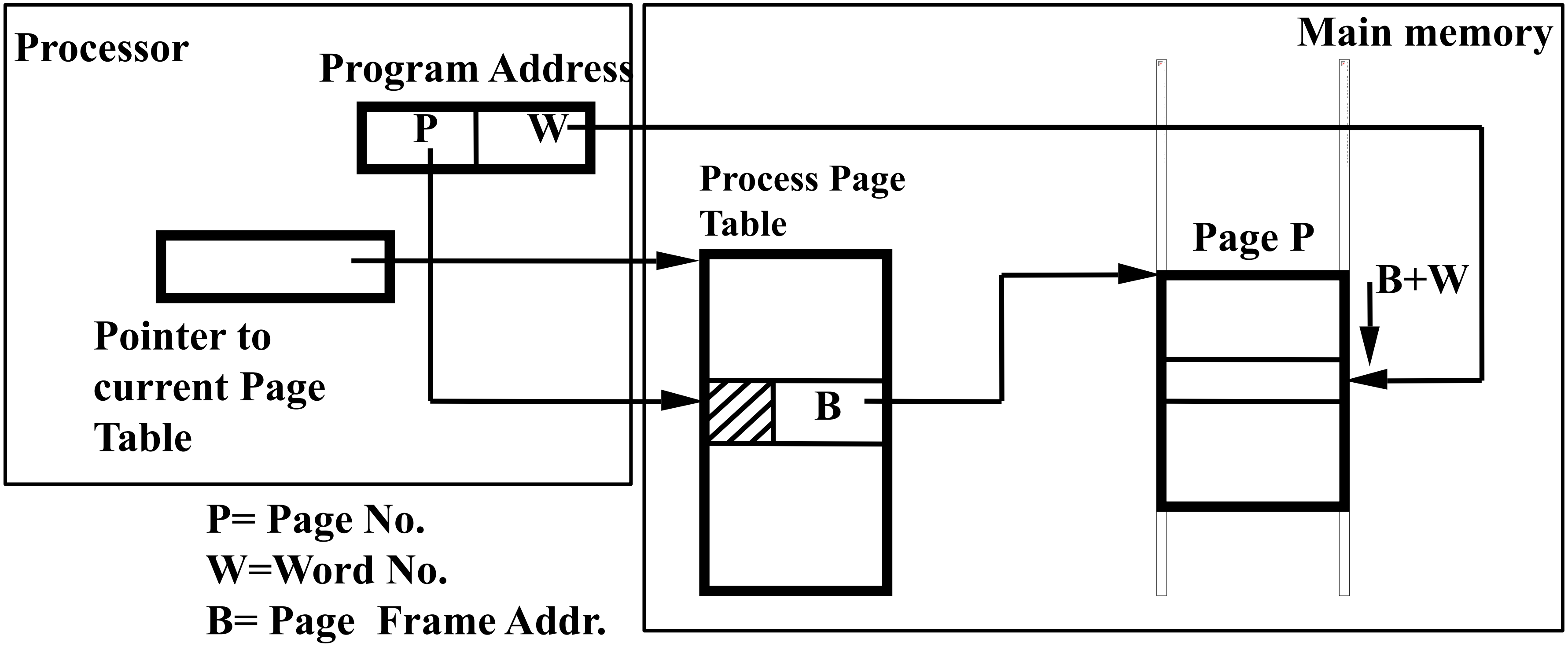
- Running or ready process
  - some pages in main memory
- Waiting process
  - all pages can be on disk
- Paging is transparent to programmer

Paging Mechanism

(1) Address Mapping
(2) Page Transfer

(Review introductory operating systems material for students lacking CS background)

Paging - Address Mapping



P= Page No.
W=Word No.
B= Page Frame Addr.

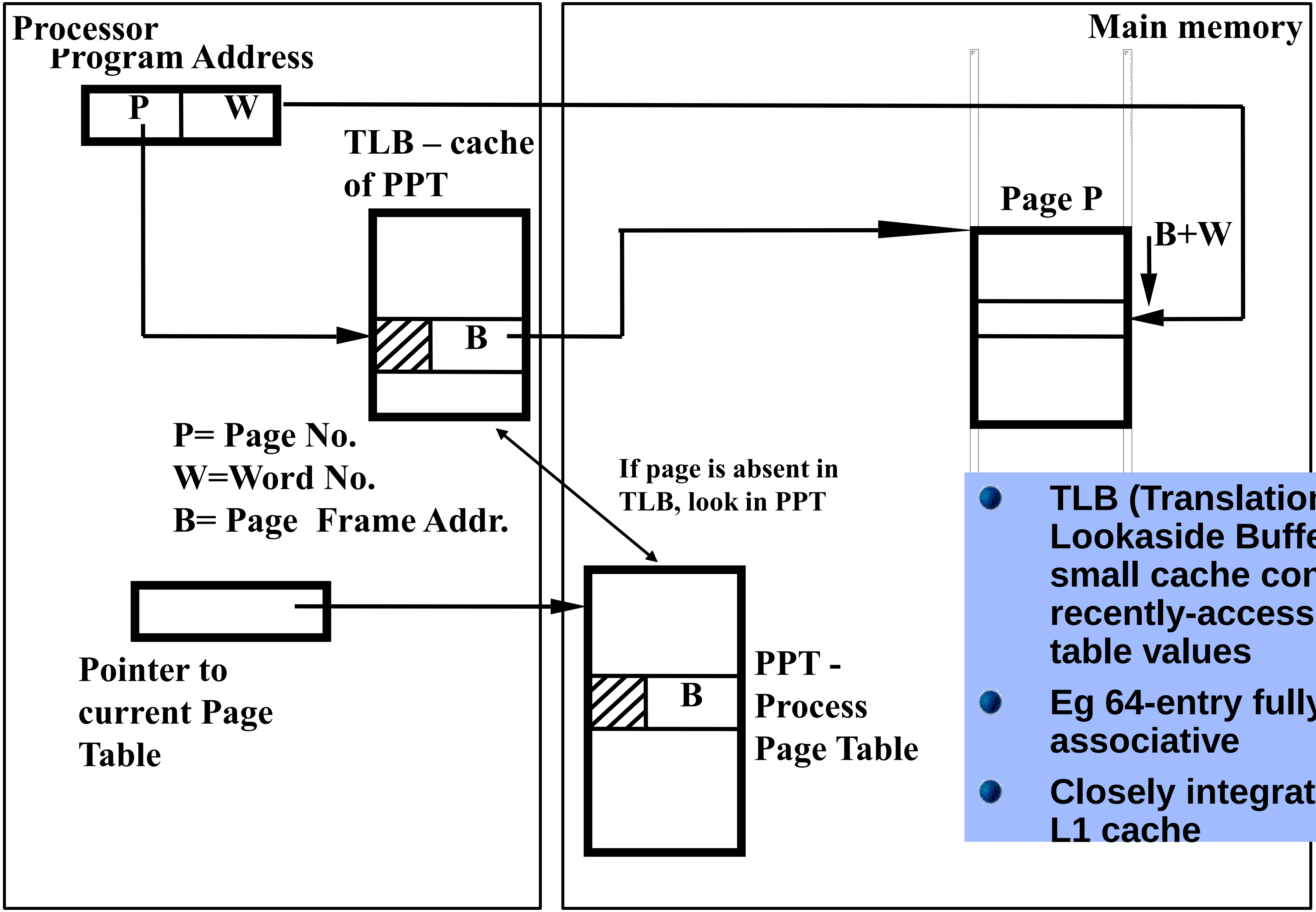
Example: Word addressed machine, W = 8 bits, page size = 4096

$$Amap(P,W) := PPT[P] * 4096 + W$$

Note: The Process Page Table (PPT) itself can be paged

(Review introductory operating systems material for students lacking CS background)

Paging - Address Mapping



Processor

Program Address

P

W

TLB – cache of PPT

B

Main memory

Page P

B+W

P= Page No.

W=Word No.

B= Page Frame Addr.

If page is absent in TLB, look in PPT

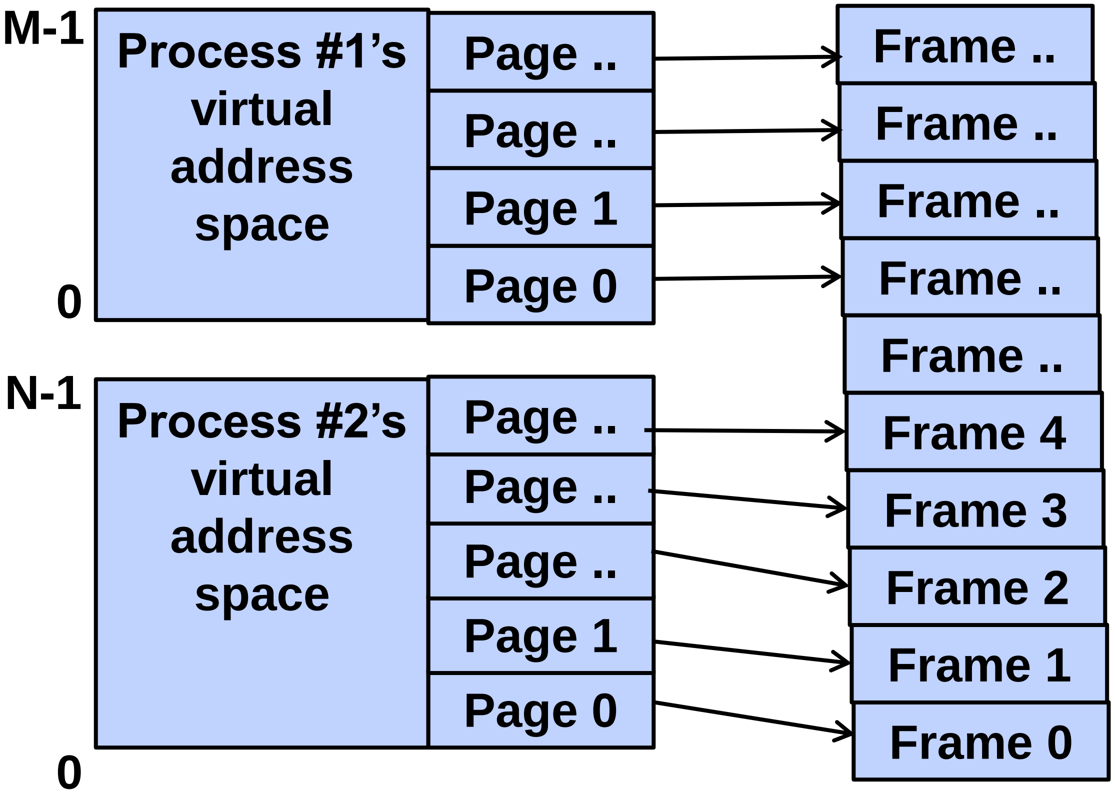
Pointer to current Page Table

B

PPT - Process Page Table

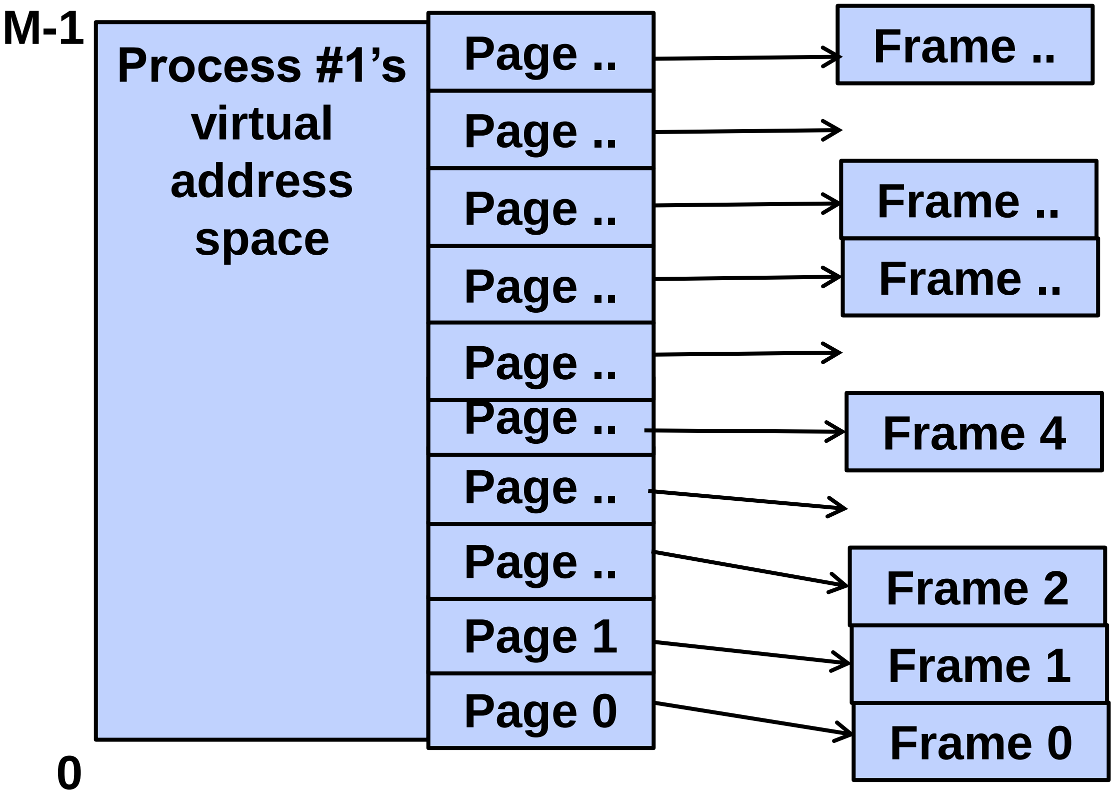
- TLB (Translation Lookaside Buffer) is small cache containing recently-accessed page table values
- Eg 64-entry fully-associative
- Closely integrated with L1 cache

What address translation is for



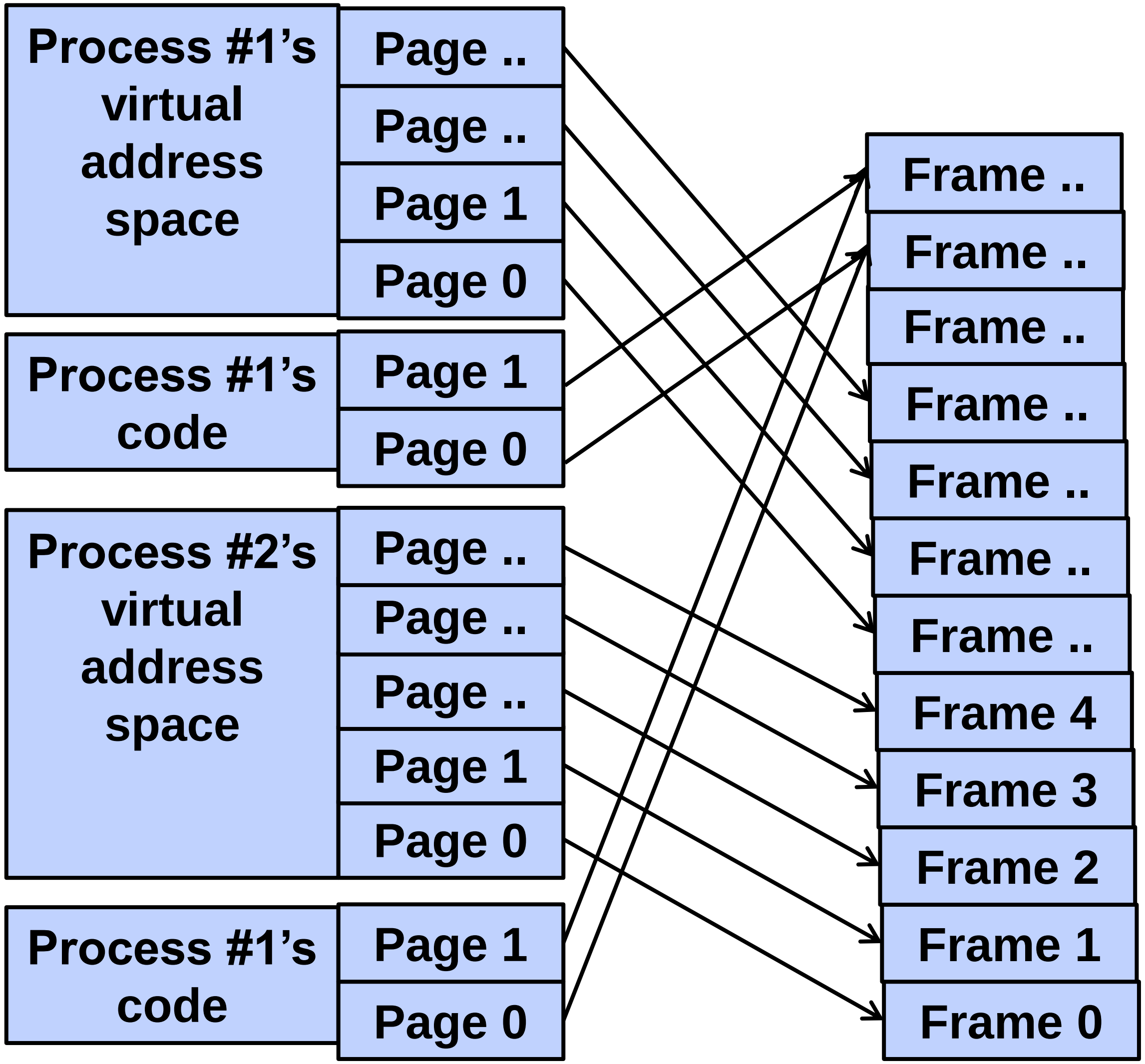
- Two different processes sharing the same physical memory
- Both processes have a virtual address starting at zero

What address translation is for



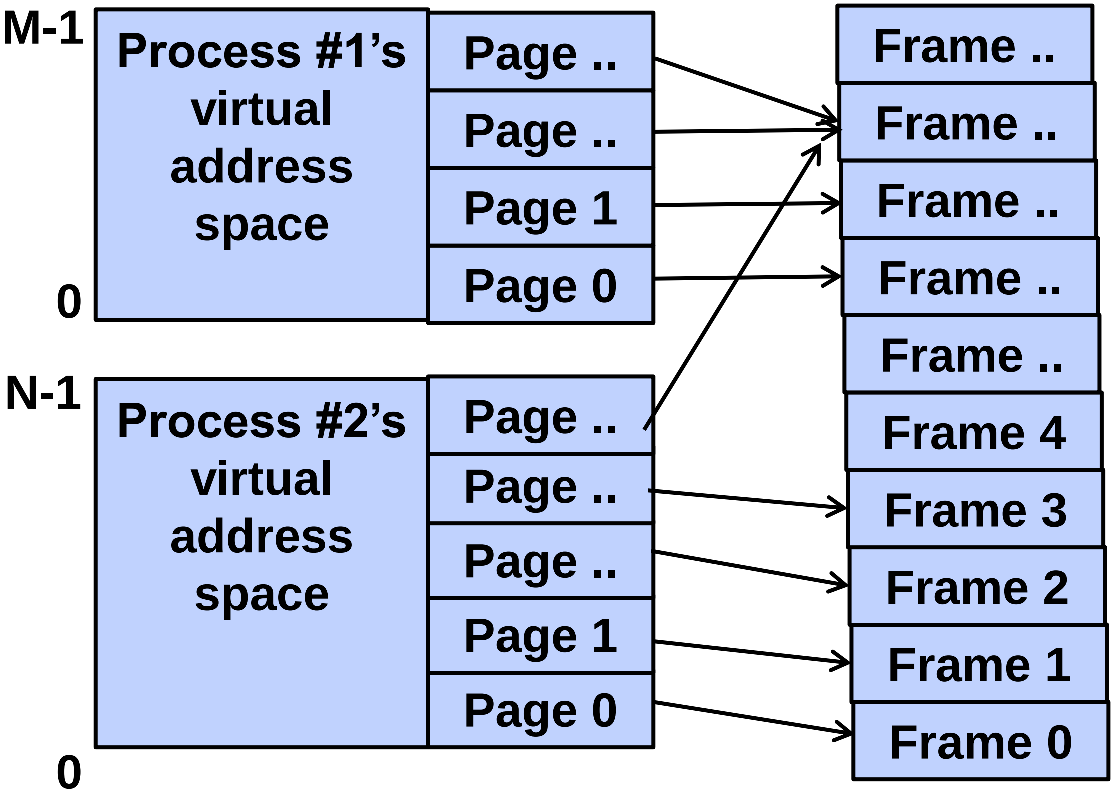
- Virtual address space may be larger than physical address space
- Some pages may be absent – OS (re-)allocates when fault occurs
- Virtual address space may be *very* large

What address translation is for



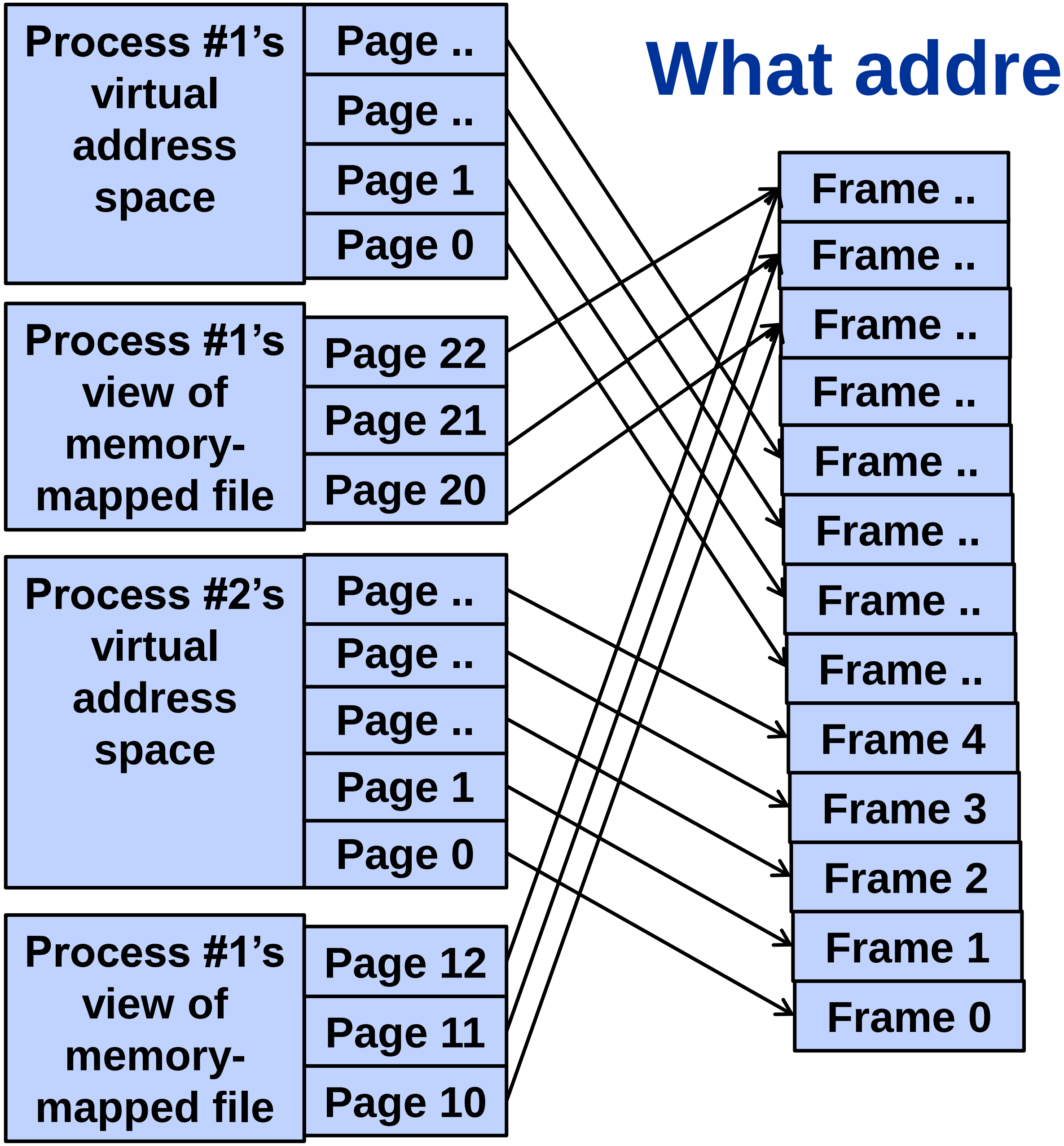
- Two different processes sharing the same code (read only)

What address translation is for



- When virtual pages are initially allocated, they all share the same physical page, initialised to zero
- When a write occurs, a page fault results in a fresh writable page being allocated (“Copy-on-Write”)

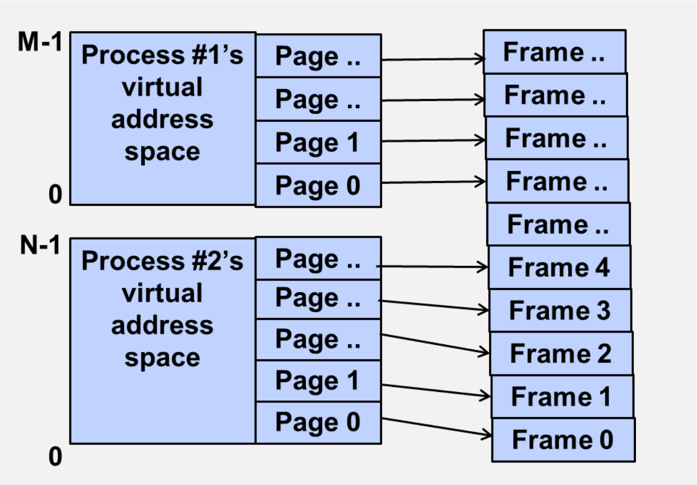
What address translation is for

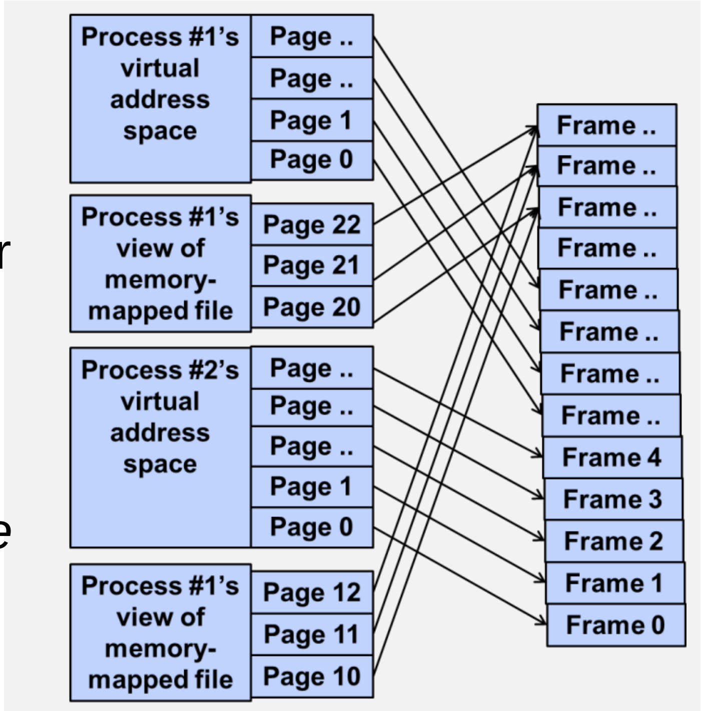


- Two different processes sharing the same memory-mapped file (with both having read and write access permissions)

Synonyms and homonyms in address translation

- Homonyms *(same sound different meaning)*
  - **same virtual address points to two different physical addresses in different processes**
  - If you have a virtually-indexed cache, flush it between context switches - or include a process identifier (PID) in the cache tag
- Synonyms *(different sound same meaning)*
  - **different virtual addresses (from the same or different processes) point to the same physical address**
  - in a virtually-indexed cache
    - a physical address could be cached twice under different virtual addresses
    - updates to one cached copy would not be reflected in the other cached copy
    - solution: make sure synonyms can't co-exist in the cache, *e.g., OS can force synonyms to have the same index bits in a direct mapped cache* (sometimes called page colouring)





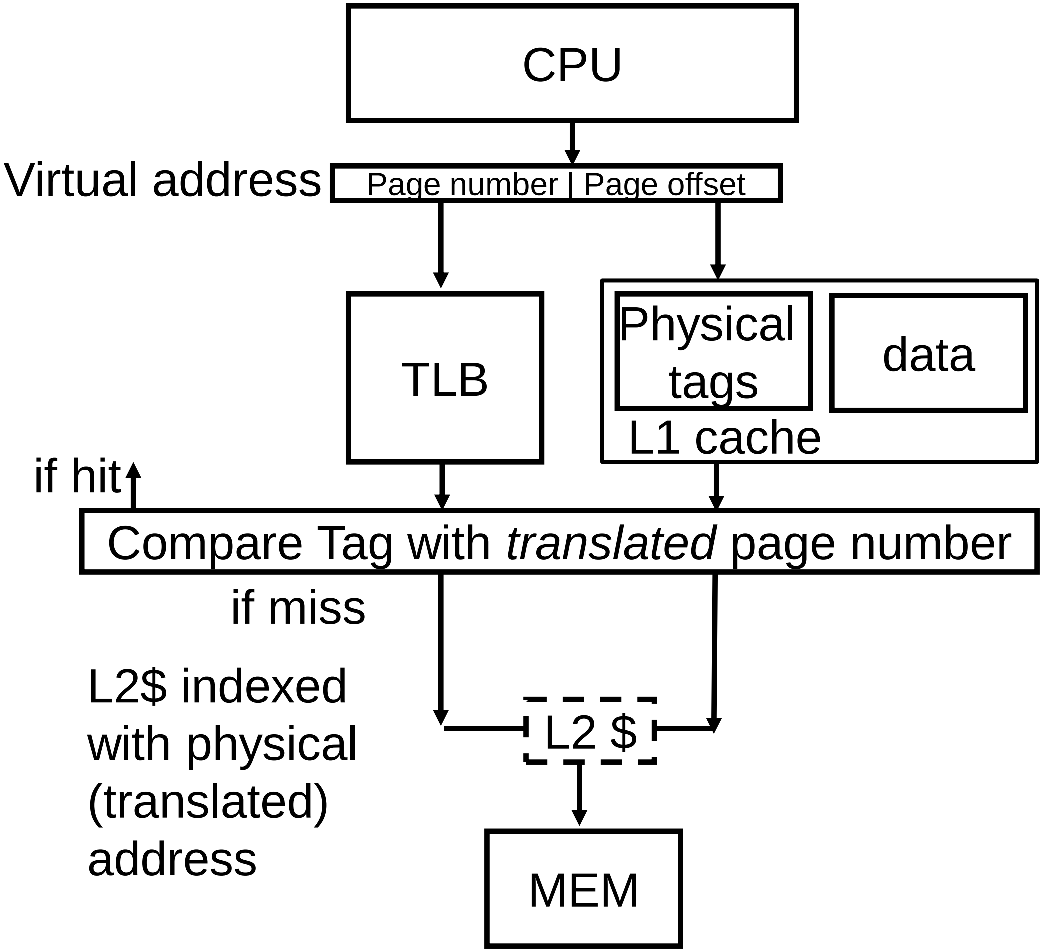
(a nice explanation in more detail can be found at http://www.ece.cmu.edu/~jhoe/course/ece447/handouts/L22.pdf)

Maybe also see http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0344h/BEIBFJEA.html

Fast cache hits by avoiding translation:

Index with physical (untranslated) portion of address

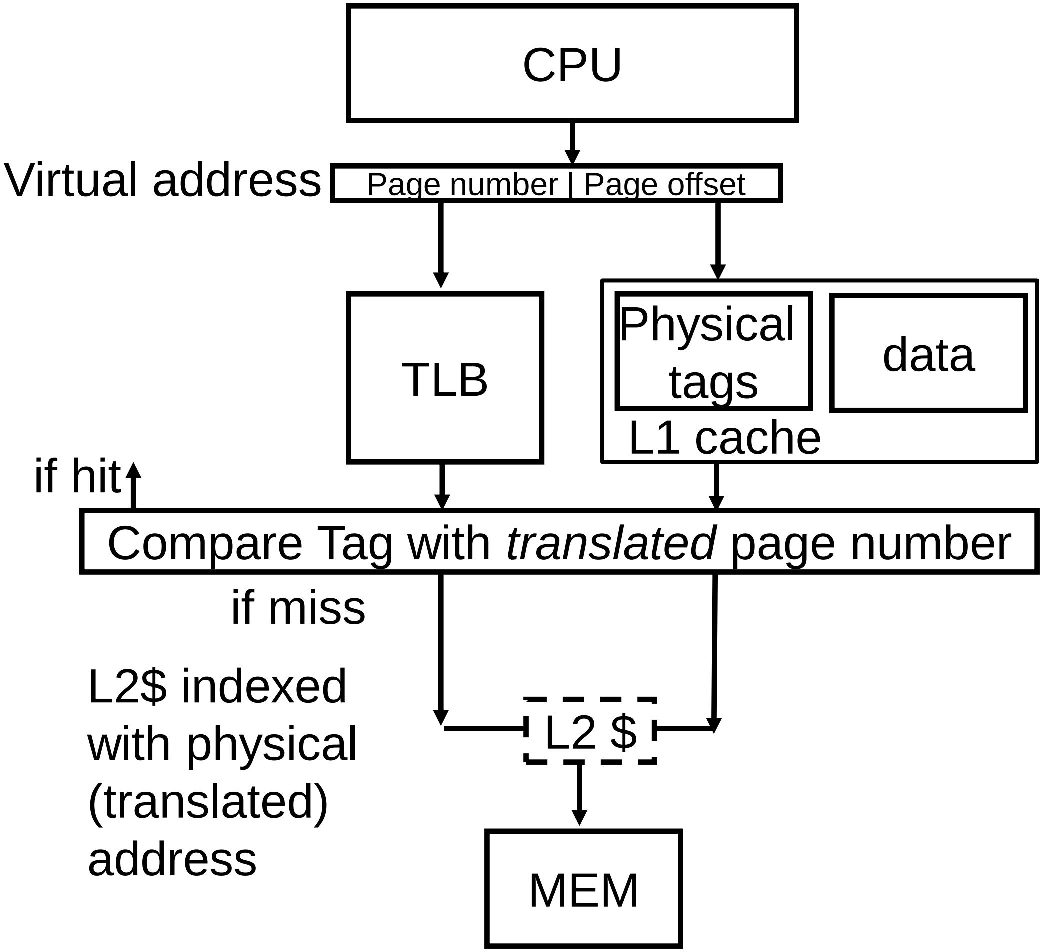
- **If the cache index consists only of physical (untranslated) bits of the address,**
  - **We can start tag access in parallel with translation**
  - **so we can compare to physical tag**


CPU
Virtual address
Page number | Page offset
TLB
Physical tags
data
L1 cache
if hit
Compare Tag with *translated* page number
if miss
L2$ indexed with physical (translated) address
L2 $
MEM

Fast cache hits by avoiding translation:

Index with physical (untranslated) portion of address

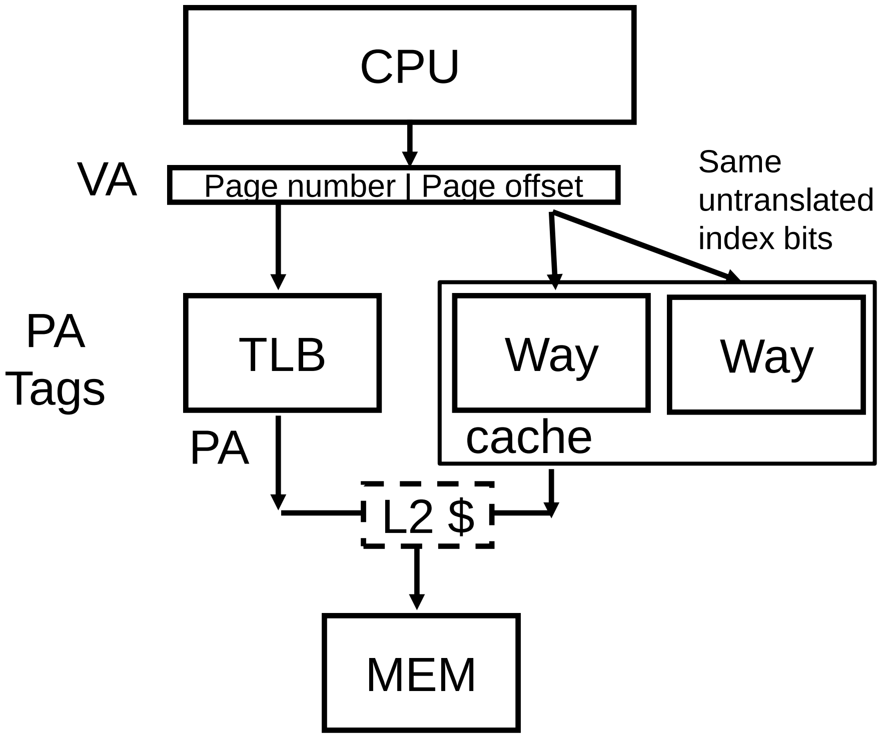
- **If the cache index consists only of physical (untranslated) bits of the address,**
  - **We can start tag access in parallel with translation**
  - **so we can compare to physical tag**
- **Limits cache to page size: what if we want bigger caches and still use same trick?**



Fast cache hits by avoiding translation:

Index with physical (untranslated) portion of address

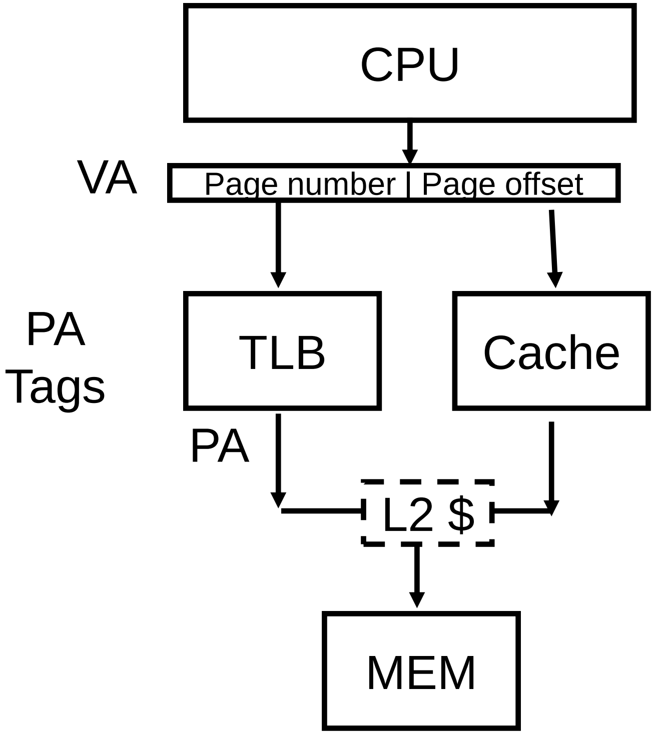
- If the cache index consists only of physical (untranslated) bits of the address,
  - We can start tag access in parallel with translation
  - so we can compare to physical tag
- Limits cache to page size: what if want bigger caches and still use same trick?
  - Option 1: Higher associativity
    - This is an attractive and common choice
    - Consequence: L1 caches are often highly associative



Fast cache hits by avoiding translation:

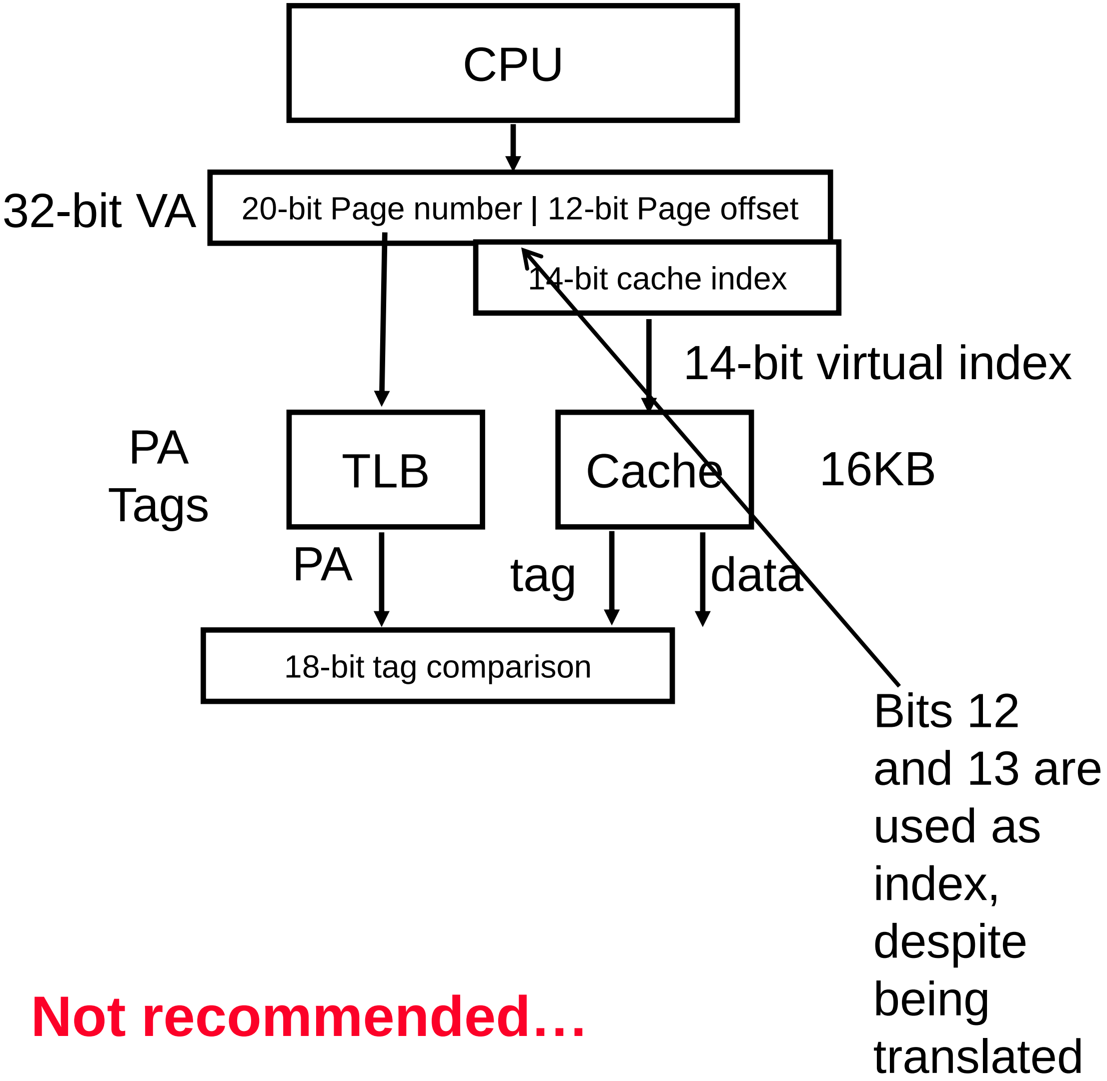
Index with physical (untranslated) portion of address

- **If the cache index consists only of physical (untranslated) bits of the address,**
  - **We can start tag access in parallel with translation**
  - **so we can compare to physical tag**
- **Limits cache to page size: what if want bigger caches and still use same trick?**
  - **Option 1: Higher associativity**
  - **Option 2: Page coloring**
    - **Get the operating system to help – see next slide**
    - **A cache conflict occurs if two cache blocks that have the same tag (physical address) are mapped to two different virtual addresses**
    - **Make sure OS never creates a page table mapping with this property**



What if you *insist* on using some translated bits as index bits?

- Page colouring for **synonym consistency**:
- "A cache synonym conflict occurs if two cache blocks that have the same tag (physical address) are mapped to two different virtual addresses"
- So if the OS needs to create two virtual memory regions, A and B within the same process, mapping the **same** physical address region
  - So A[0] and B[0] have *different* VAs
  - But after translation refer to the same location
  - We need to ensure that the virtual addresses that we assign to A[0] and B[0] match in bits 12&13
  - So the map to the same location in the cache
  - So they have only one value!

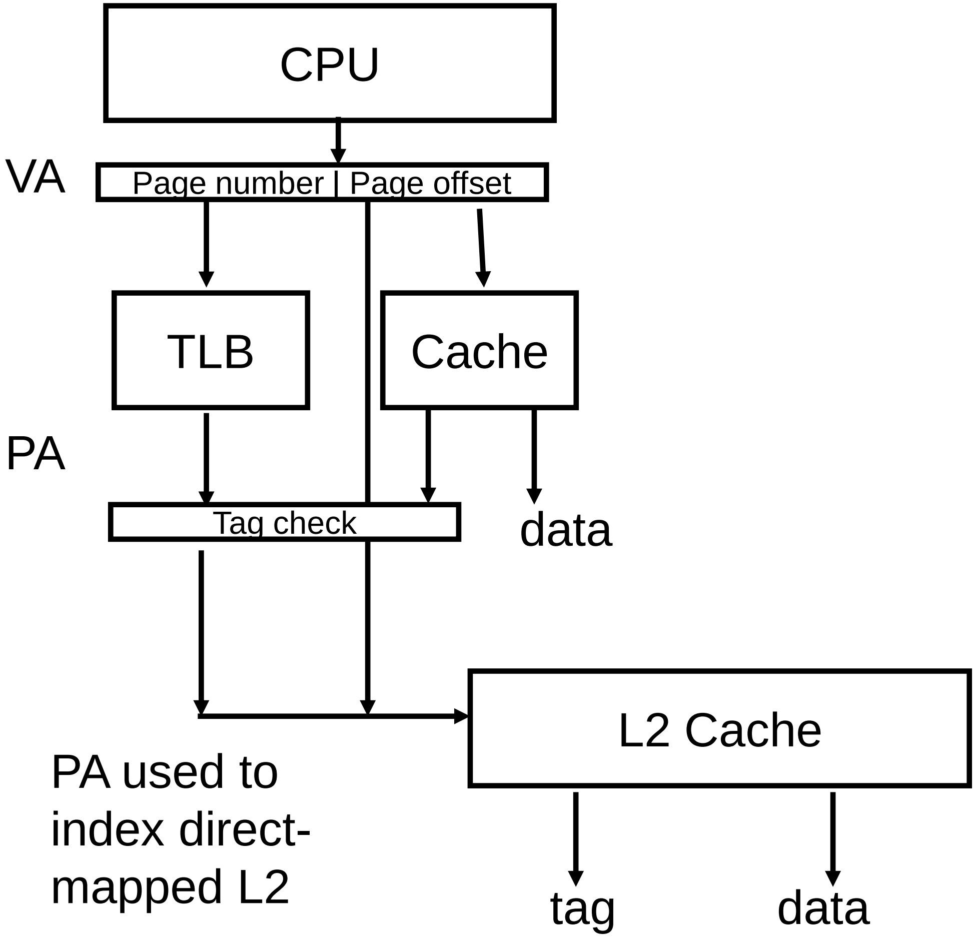


Not recommended…

The Linux mmap system call for creating a memory region shared between two processes *chooses* the address of the region in order to allow the OS to do this if necessary

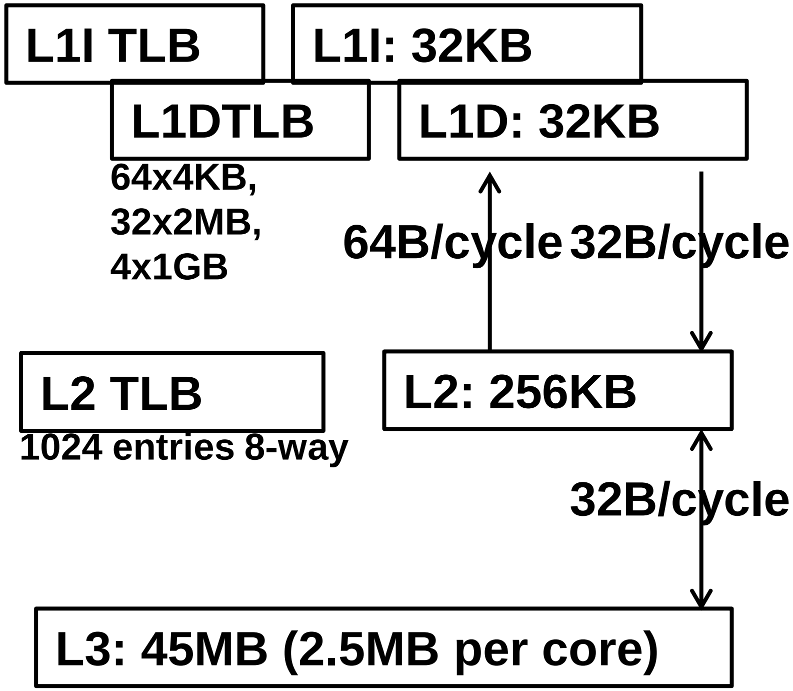
Associativity conflicts depend on address translation

- The L2 cache is indexed with *translated* address
- So the L2 associativity conflicts depend on the virtual-to-physical mapping
- It would be helpful if the OS could choose non-conflicting pages for frequently-accessed data! **(page colouring for conflict avoidance)**
- Or at least, make sure adjacent pages don't map to the same L2 index



- **Running the same program again on the same data may result in different associativity conflicts**
- **Because you may get a different virtual-to-physical mapping**

TLBs in Haswell



L1I TLB

L1I: 32KB

L1DTLB

64x4KB,
32x2MB,
4x1GB

L1D: 32KB

64B/cycle

32B/cycle

L2 TLB

1024 entries 8-way

L2: 256KB

32B/cycle

L3: 45MB (2.5MB per core)

L1: 32KB, 8-way associative I and D
L1D: writeback, two 256-bit loads and a 256-bit store every cycle

So each L1 way is 32/8=4KB
Virtually indexed, Physically Tagged (VIPT)

L2 and L3 are physically indexed

TLBs support three different page sizes – 4KB, 2MB, 1GB

L1 ITLB:128 mappings for 4KB pages – 4-way set associative
and 8 2MB-page mappings
L1 DTLB: 64 mappings for 4KB pages – 4-way set associative
(fixed partition between two threads)
and 8 2MB-page mapping
and 4 mappings for 1GB pages

- **Example: Intel Haswell e5 2600 v3**

Summary

We can reduce the hit time…..

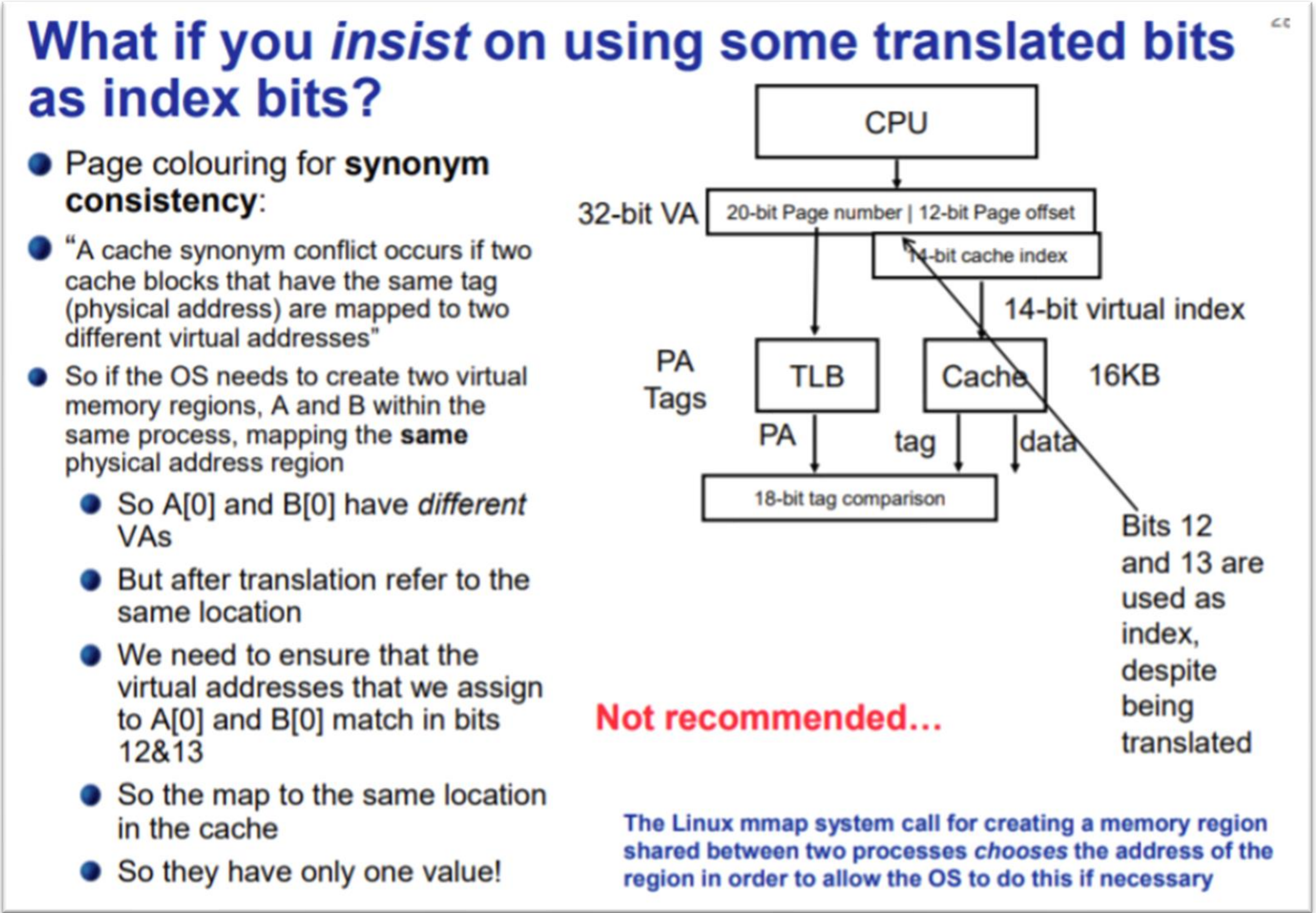
- Using a really small cache (and a larger next-level cache)
- With a pipelined cache (really only improves bandwidth)
- With a multi-bank cache (only increases bandwidth)
- By using a direct-mapped cache
- By passing data forward while checking tags in parallel
- Using way prediction (not covered in slides)

- By taking address translation off the critical path
  - Access the TLB in parallel with the L1 cache
    - Do not use translated bits as index bits if you can help it!
  - The TLB is a cache of the address translation
    - Consider a two (multi?)-level TLB
    - Pay attention to TLB miss penalty (beyond this lecture)

Cache Optimization Summary

	Technique	*MR*	*MP*	*HT*	*Complexity*
miss rate	Larger Block Size	+	–		0
	Higher Associativity	+		–	1
	Victim Caches	+			2
	Pseudo-Associative Caches	+			2
	HW Prefetching of Instr/Data	+			2
	Compiler Controlled Prefetching	+			3
	Compiler Reduce Misses	+			0
miss penalty	Priority to Read Misses		+		1
	Early Restart & Critical Word 1st		+		2
	Non-Blocking Caches		+		3
	Second Level Caches		+		2

Edstem questions



What if you *insist* on using some translated bits as index bits?

- Page colouring for **synonym consistency**:
- "A cache synonym conflict occurs if two cache blocks that have the same tag (physical address) are mapped to two different virtual addresses"
- So if the OS needs to create two virtual memory regions, A and B within the same process, mapping the **same** physical address region
  - So A[0] and B[0] have *different* VAs
  - But after translation refer to the same location
  - We need to ensure that the virtual addresses that we assign to A[0] and B[0] match in bits 12&13
  - So the map to the same location in the cache
  - So they have only one value!

Not recommended…

The Linux mmap system call for creating a memory region shared between two processes *chooses* the address of the region in order to allow the OS to do this if necessary

- **Q:** "Hi, I don't understand why page colouring only requires bits 12 and 13 of A[0] and B[0] being the same? Isn't A[0] and B[0] mapping to the same physical address still having different virtual addresses? I.e. there is still a synonym conflict?"
- **A:** The objective is to ensure that when A[0] is allocated into the cache, and then later B[0] is loaded, the two words map to the same cache line in the cache. If they didn't (which would happen if bits 12 and 13 were different) then we would have a consistency problem. A store to A[0] would update one cached copy of the line, but a load from B[0] would load the original, unchanged data.