Advanced Computer Architecture

Department of Computing, Imperial College London

Chapter 4: Caches and Memory Systems

Part 4: hit time reduction – and address translation

November 2025

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:

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

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

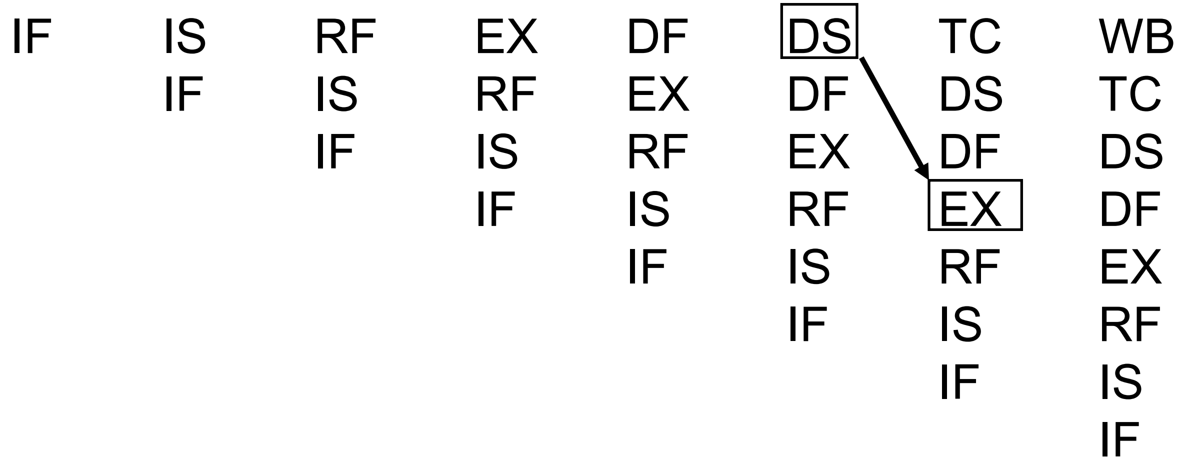
We now look at each of these in turn…

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

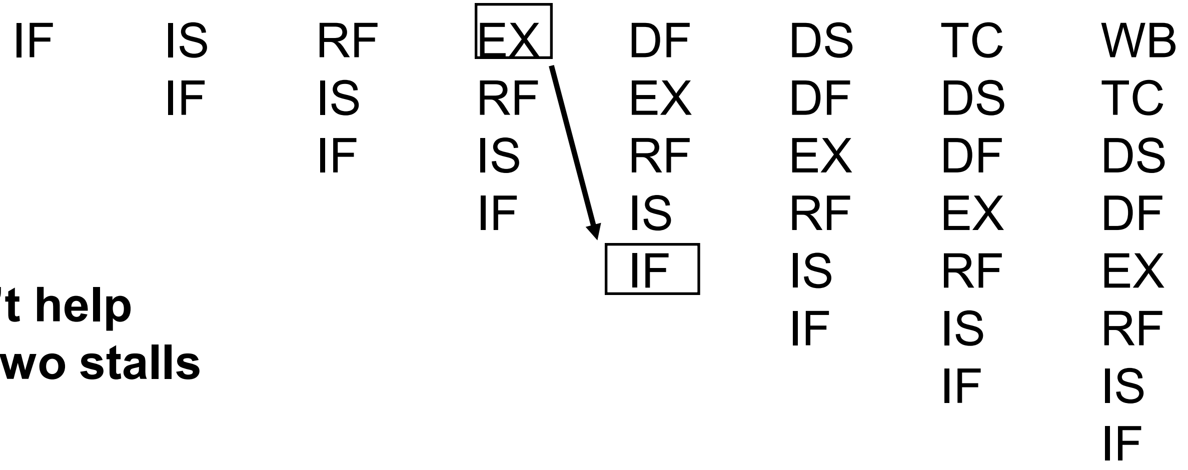


IF	IS	RF	EX	DF	DS	TC	WB
	IF	IS	RF	EX	DF	DS	TC
		IF	IS	RF	EX	DF	DS
			IF	IS	RF	EX	DF
				IF	IS	RF	EX
					IF	IS	RF
						IF	IS
							IF

THREE Cycle Branch Latency

(conditions evaluated during EX phase)

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



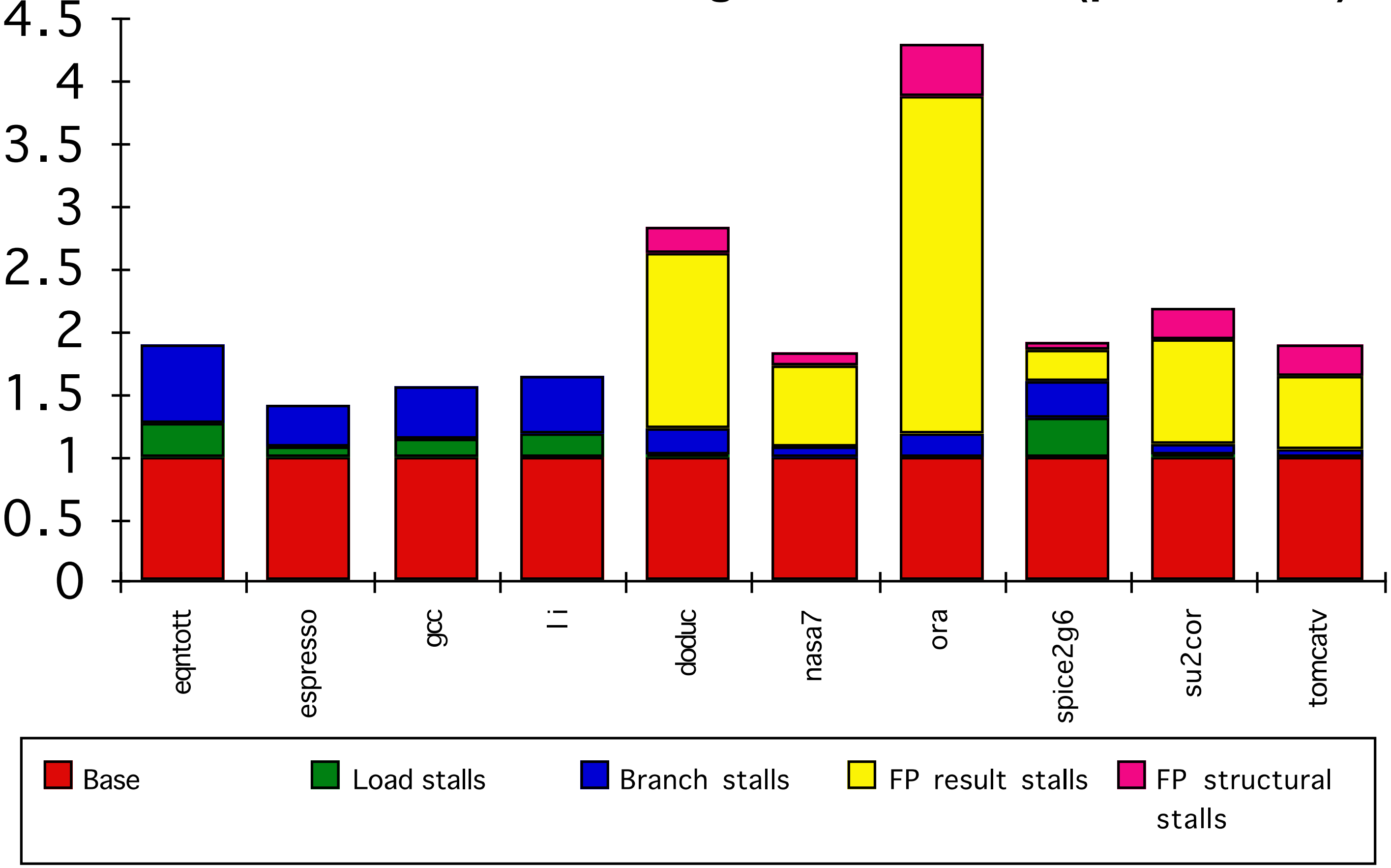
IF	IS	RF	EX	DF	DS	TC	WB
	IF	IS	RF	EX	DF	DS	TC
		IF	IS	RF	EX	DF	DS
			IF	IS	RF	EX	DF
				IF	IS	RF	EX
					IF	IS	RF
						IF	IS
							IF

- **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

- 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?)

H&P 6th ed pages 99-100

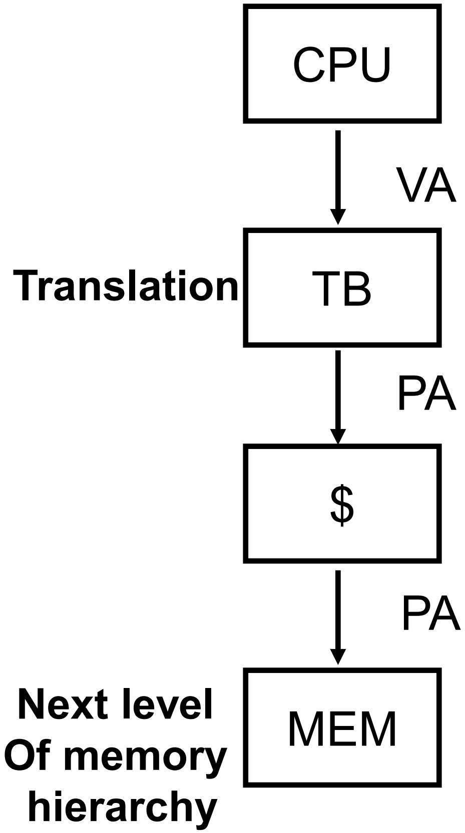
  - Other options are possible…
    - Duplicate the cache!
    - Multi-ported RAM: support two reads, to different addresses, every cycle
      - RAM array has two wordlines per row, and two bitlines per column
      - (See for example *Single-Ended 8T SRAM cell with high SNM and low power/energy consumption* Mohagheghi et al (2023) https://www.tandfonline.com/doi/full/10.1080/00207217.2022.2118848#d1e440 Fig 4)

Virtual memory, and address translation

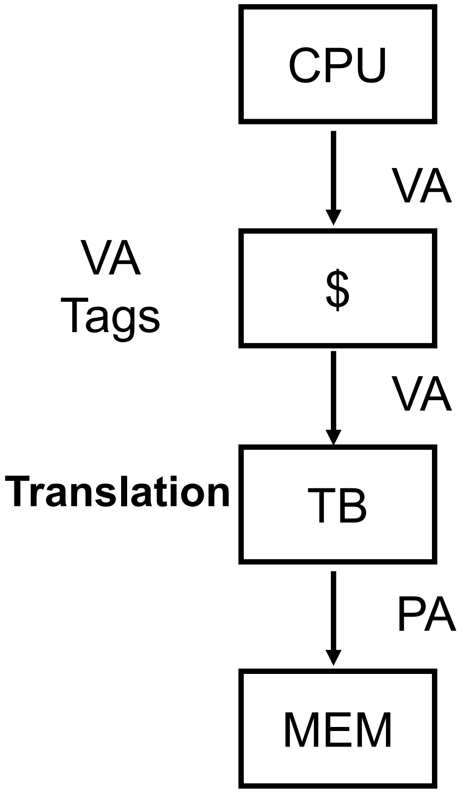
H&P 6th ed page B-36ff

- 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

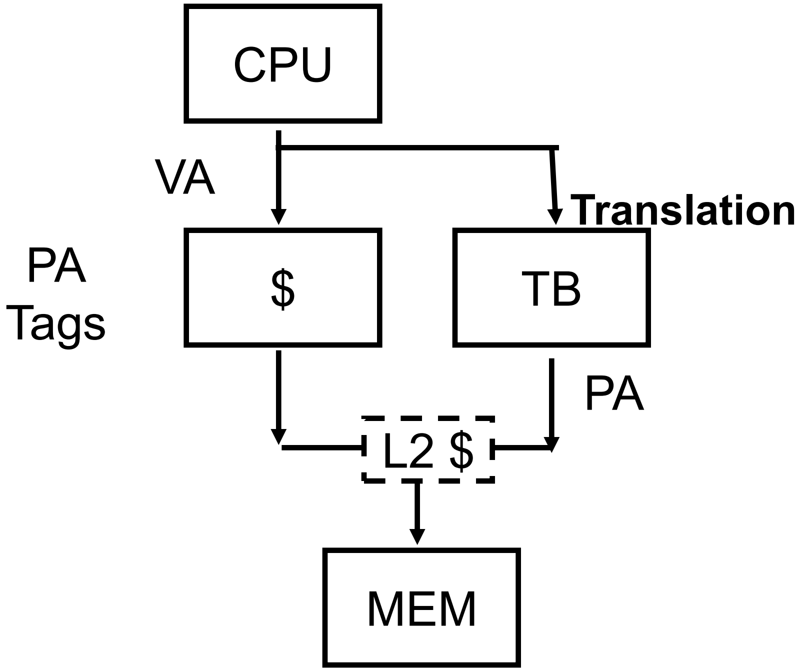
Fast hits by removing address translation from critical path


CPU
VA
Translation
TB
PA
$
PA
Next level
Of memory
hierarchy
MEM

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


CPU
VA
VA
Tags
$
VA
Translation
TB
PA
MEM

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


CPU
VA
Translation
PA
Tags
$
TB
PA
L2 $
MEM

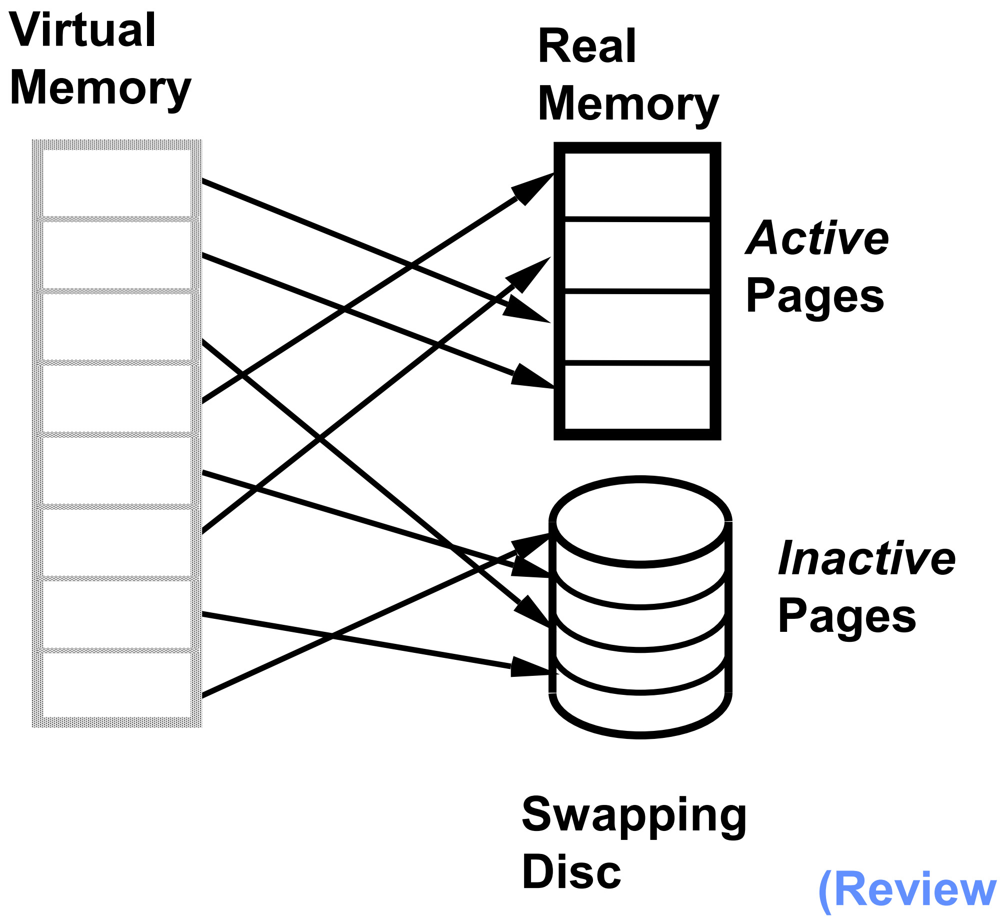
Virtually-indexed, physically-tagged (**VIPT**)
Overlap $ access with VA translation:
requires $ index to remain invariant
across translation

- ***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.



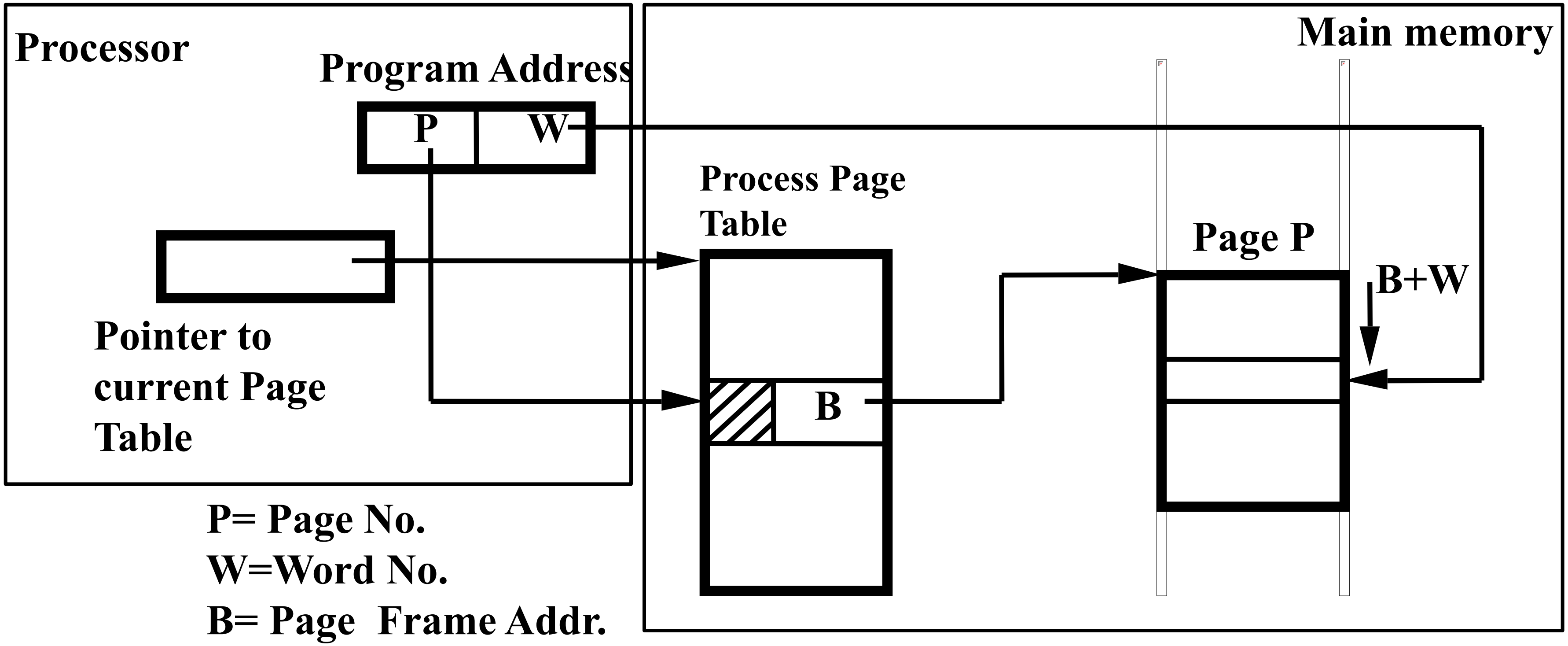
- Running or ready process
  - some pages in main memory
- Waiting process
  - all pages can be on disk
- Paging is transparent to programmer

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.

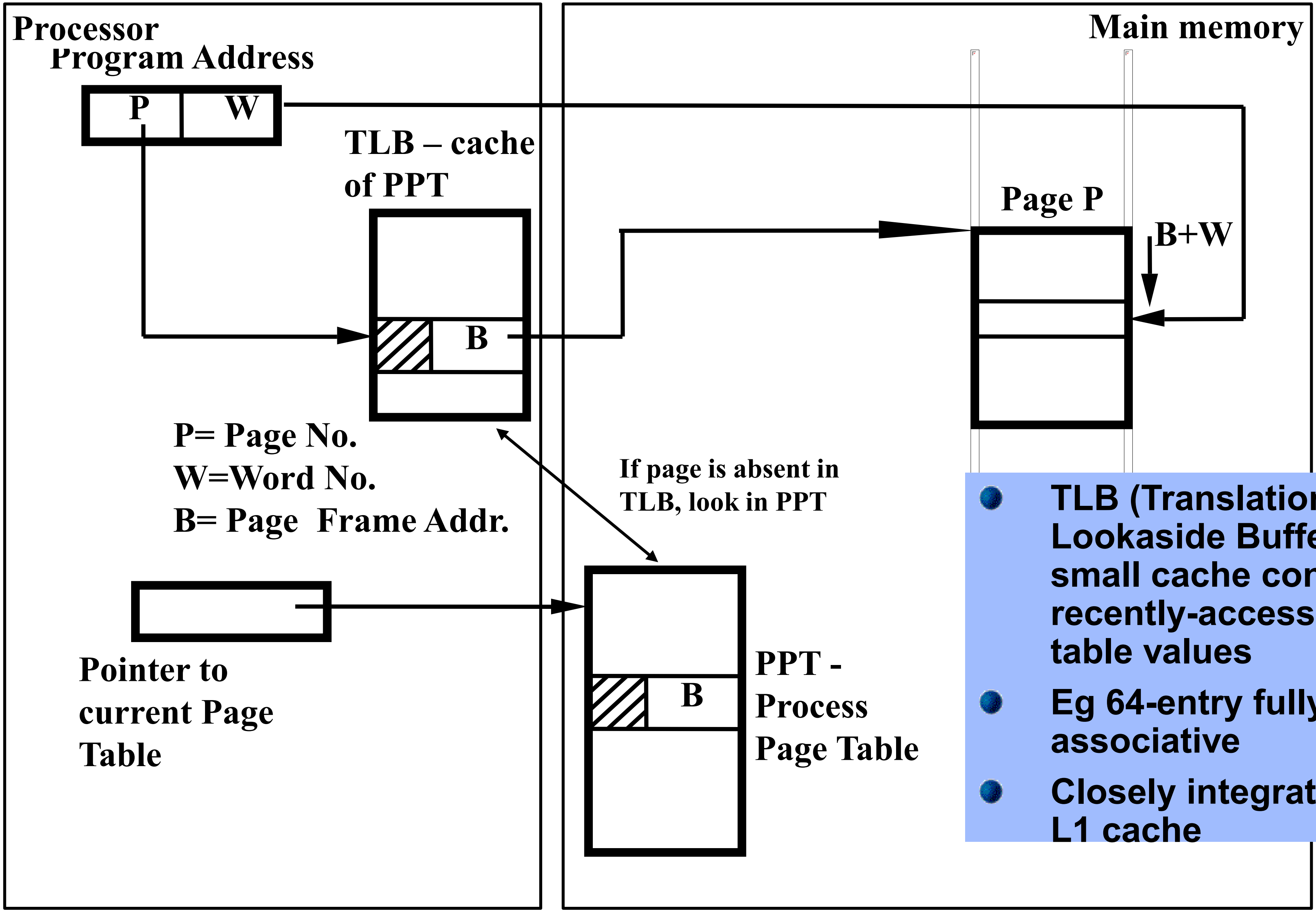
Example: Word addressed machine, W = 8 bits, page size = 4096

$$\text{Amap}(P,W) := \text{PPT}[P] * 4096 + W$$

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

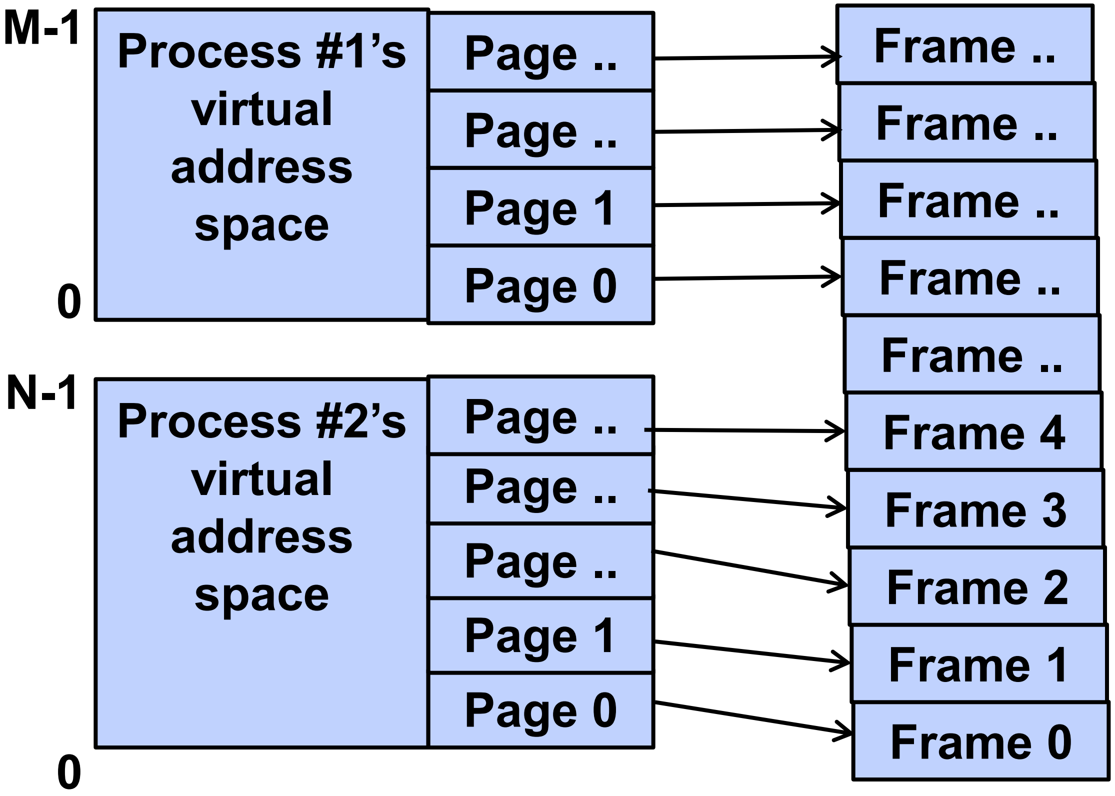
(Review introductory operating systems material for students lacking CS background)

Paging - Address Mapping



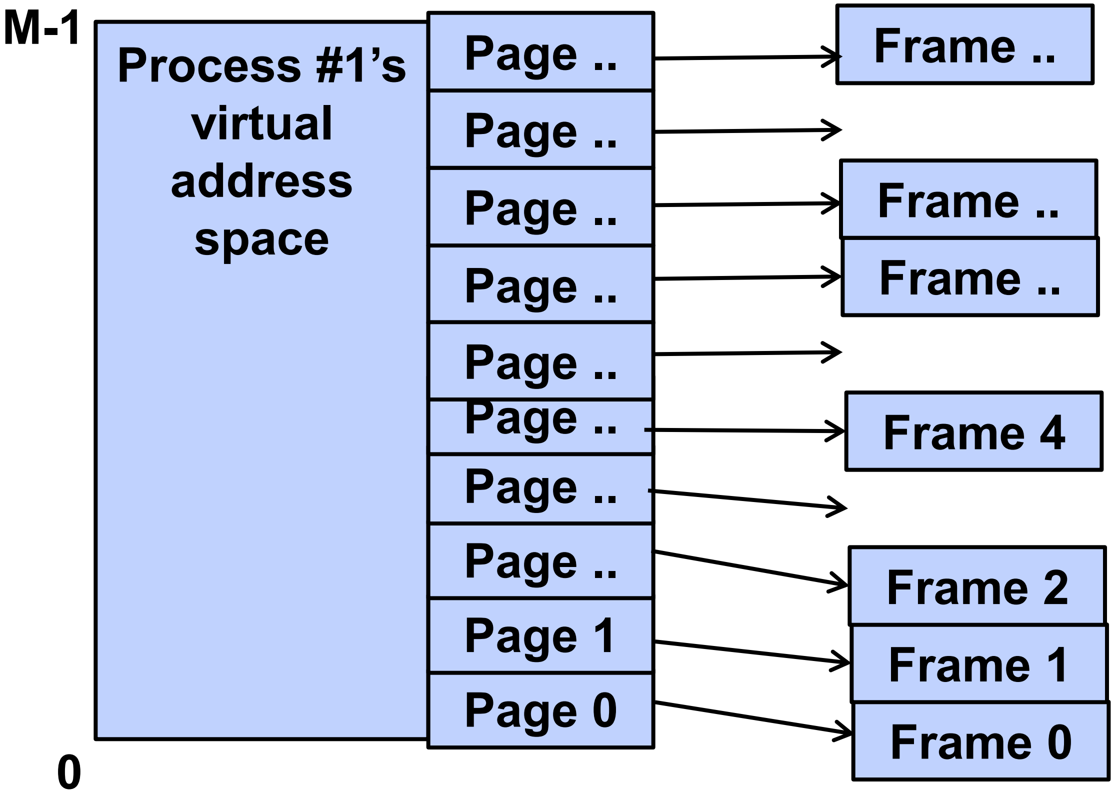
- TLB (Translation Lookaside Buffer) is small cache containing recently-accessed page table values
- Eg 64-entry fully-associative
- Closely integrated with L1 cache

What address translation is for



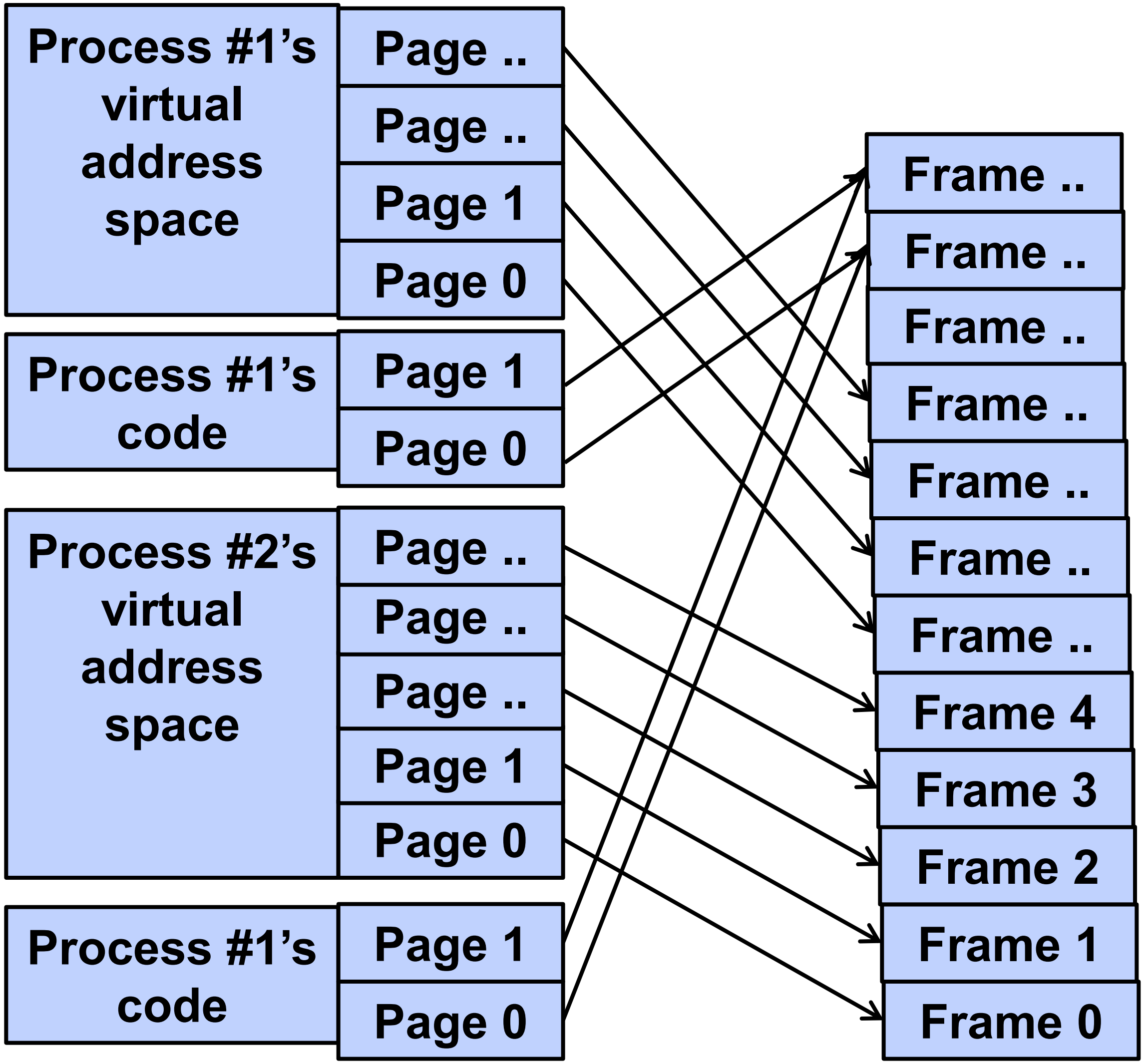
- Two different processes sharing the same physical memory
- Both processes have a virtual address starting at zero

What address translation is for



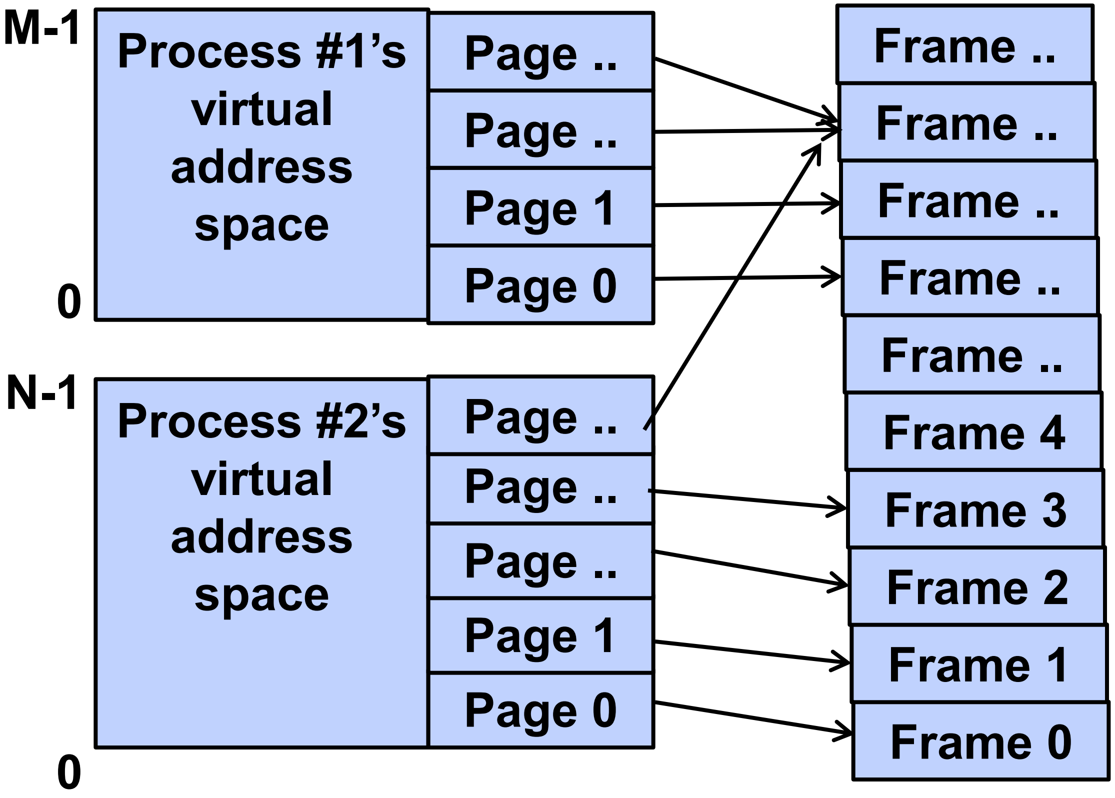
- Virtual address space may be larger than physical address space
- Some pages may be absent – OS (re-)allocates when fault occurs
- Virtual address space may be *very* large

What address translation is for



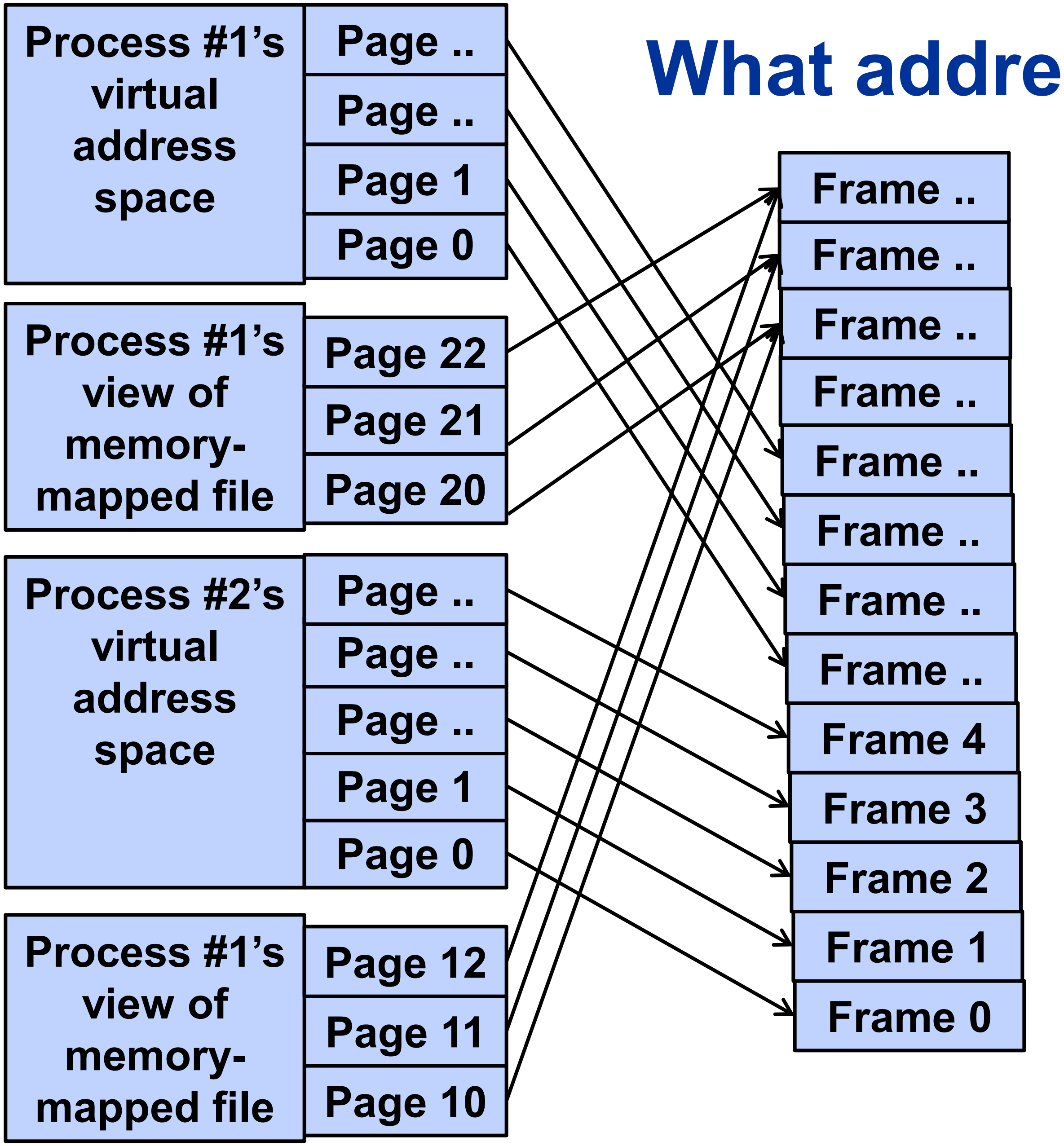
- Two different processes sharing the same code (read only)

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")

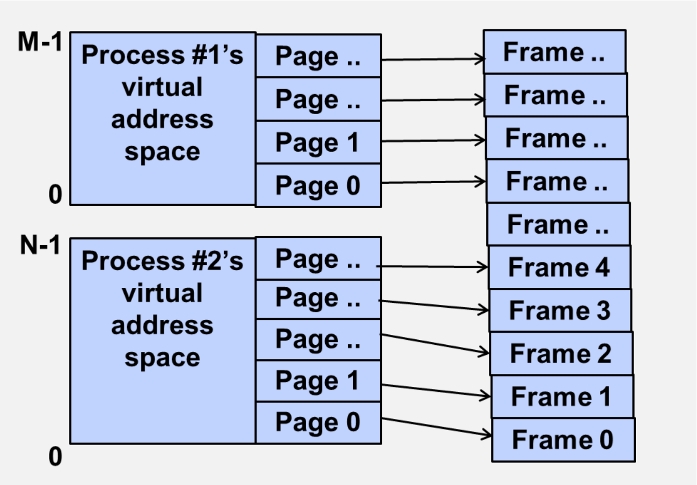
What address translation is for

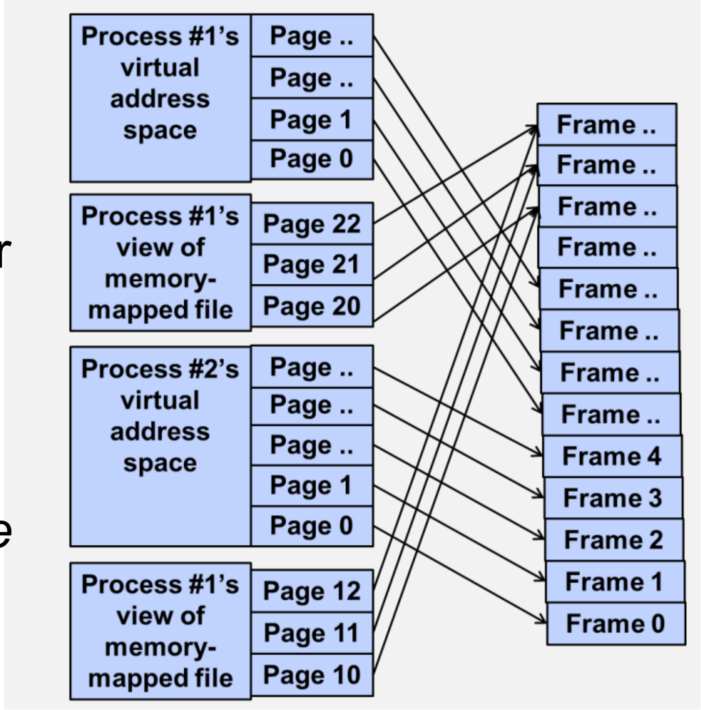


- Two different processes sharing the same memory-mapped file (with both having read and write access permissions)

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)





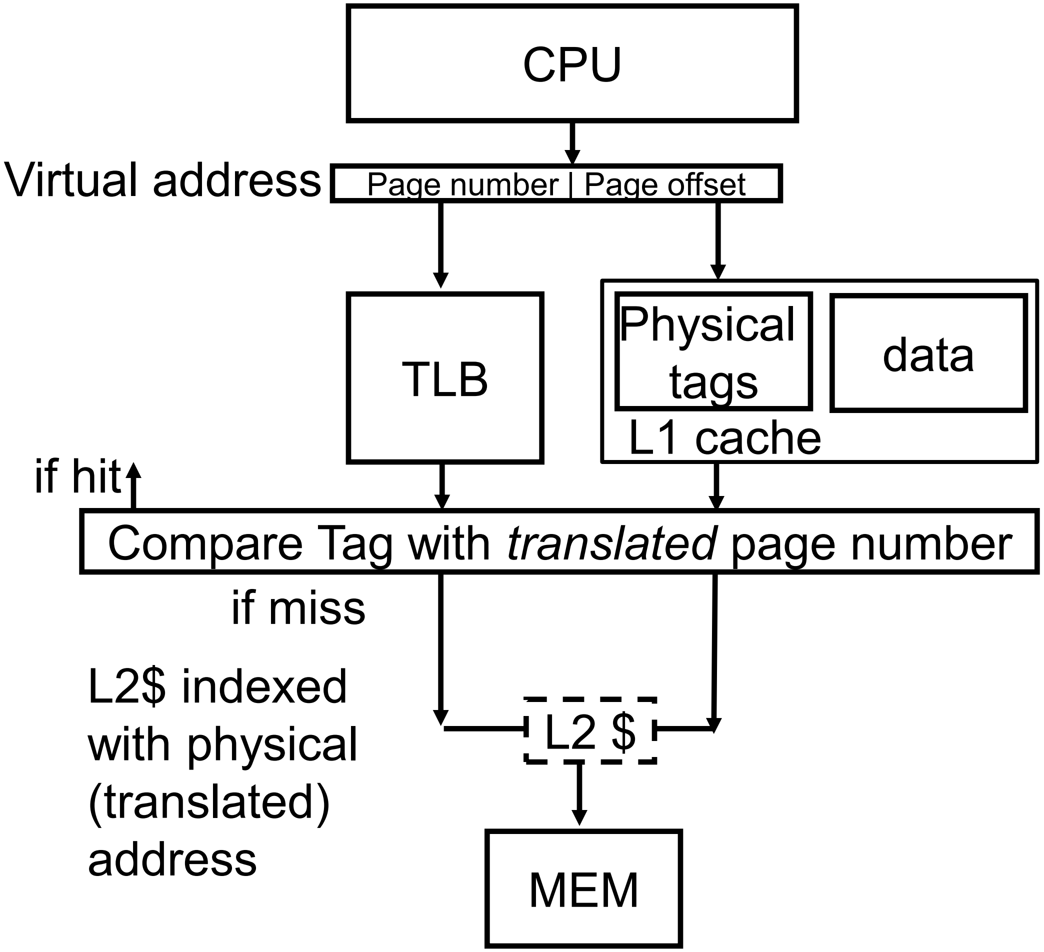
(a nice explanation in more detail can be found at http://www.ece.cmu.edu/~jhoe/course/ece447/handouts/L22.pdf)

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

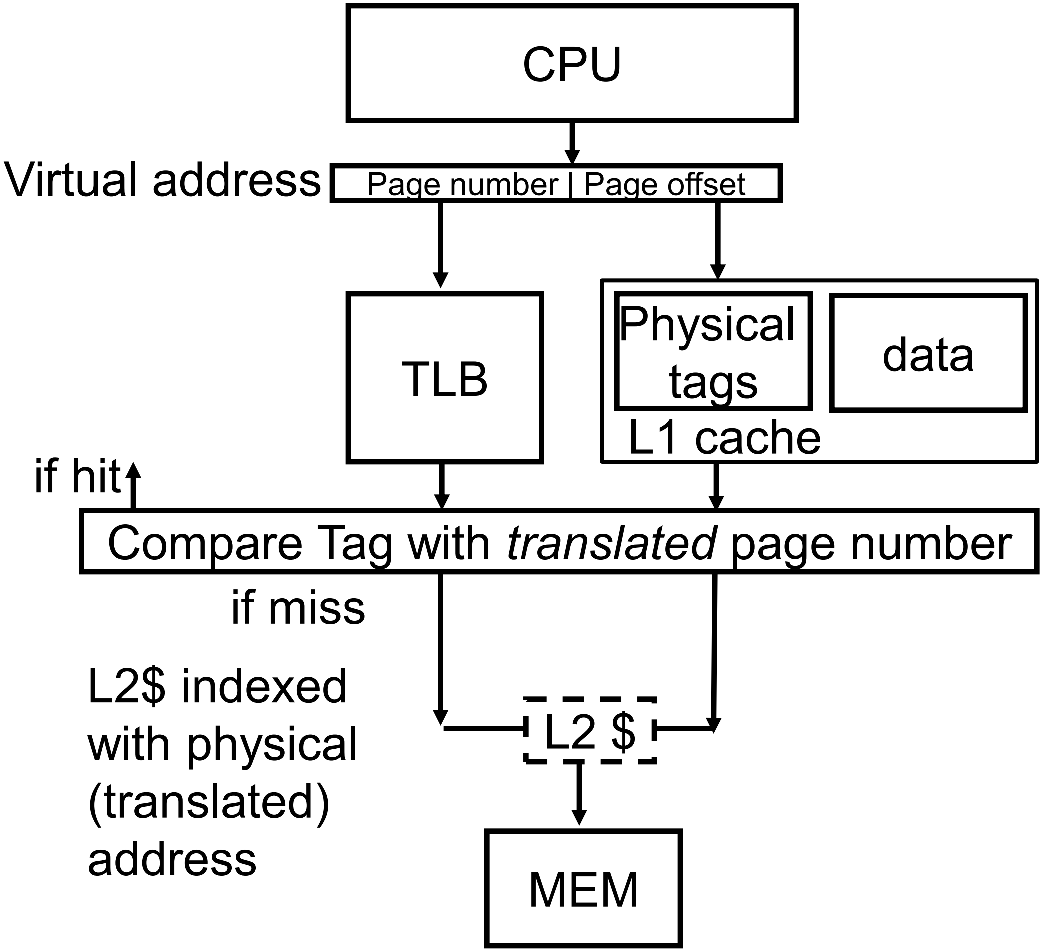
- **If the cache index consists only of physical (untranslated) bits of the address,**
  - **We can start tag access in parallel with translation**
  - **so we can compare to physical tag**


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

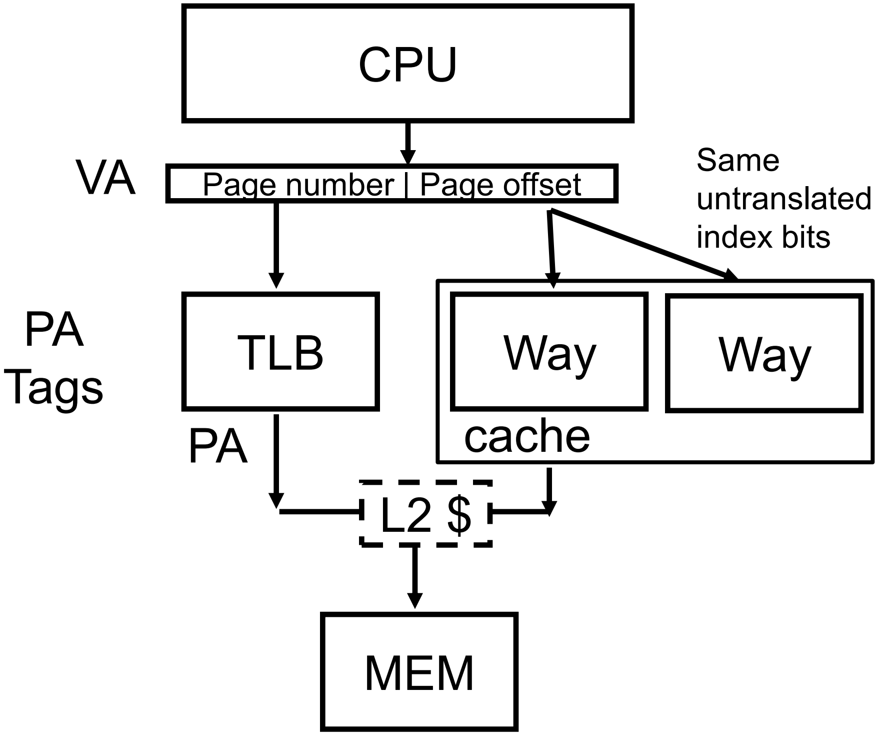
- **If the cache index consists only of physical (untranslated) bits of the address,**
  - **We can start tag access in parallel with translation**
  - **so we can compare to physical tag**
- **Limits cache to page size: what if we want bigger caches and still use same trick?**


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

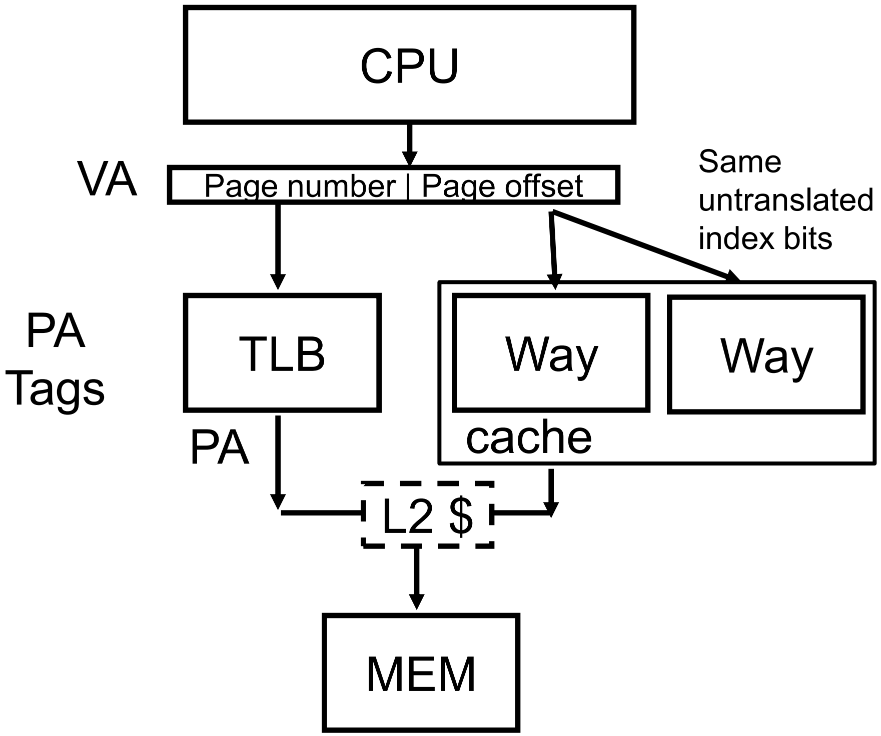
- **If the cache index consists only of physical (untranslated) bits of the address,**
  - **We can start tag access in parallel with translation**
  - **so we can compare to physical tag**
- **Limits cache to page size: what if want bigger caches and still use same trick?**
  - **Option 1: Higher associativity**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

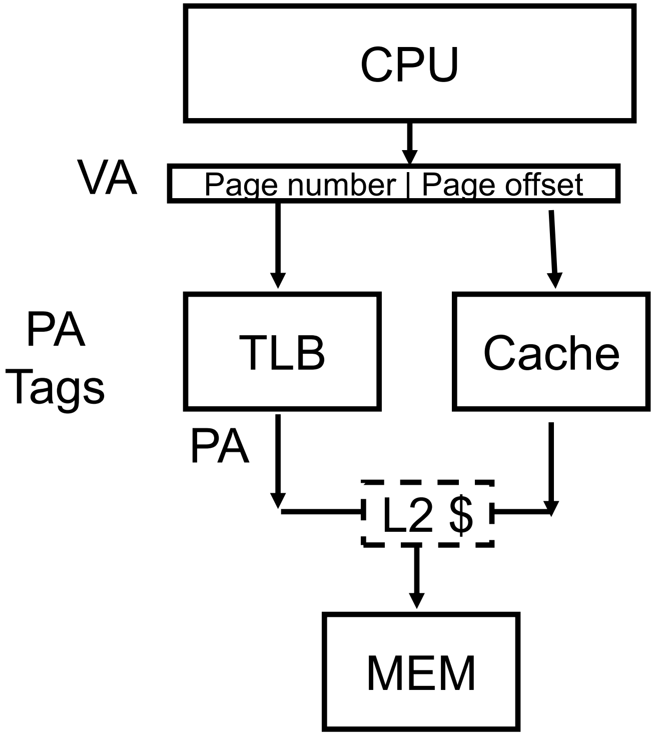


Example: Apple M1 has a 128KB L1 data cache
The page size is 16KB
It is 8-way set associative
128/8=16 so only the untranslated address bits are used to index the cache

Fast cache hits by avoiding translation:

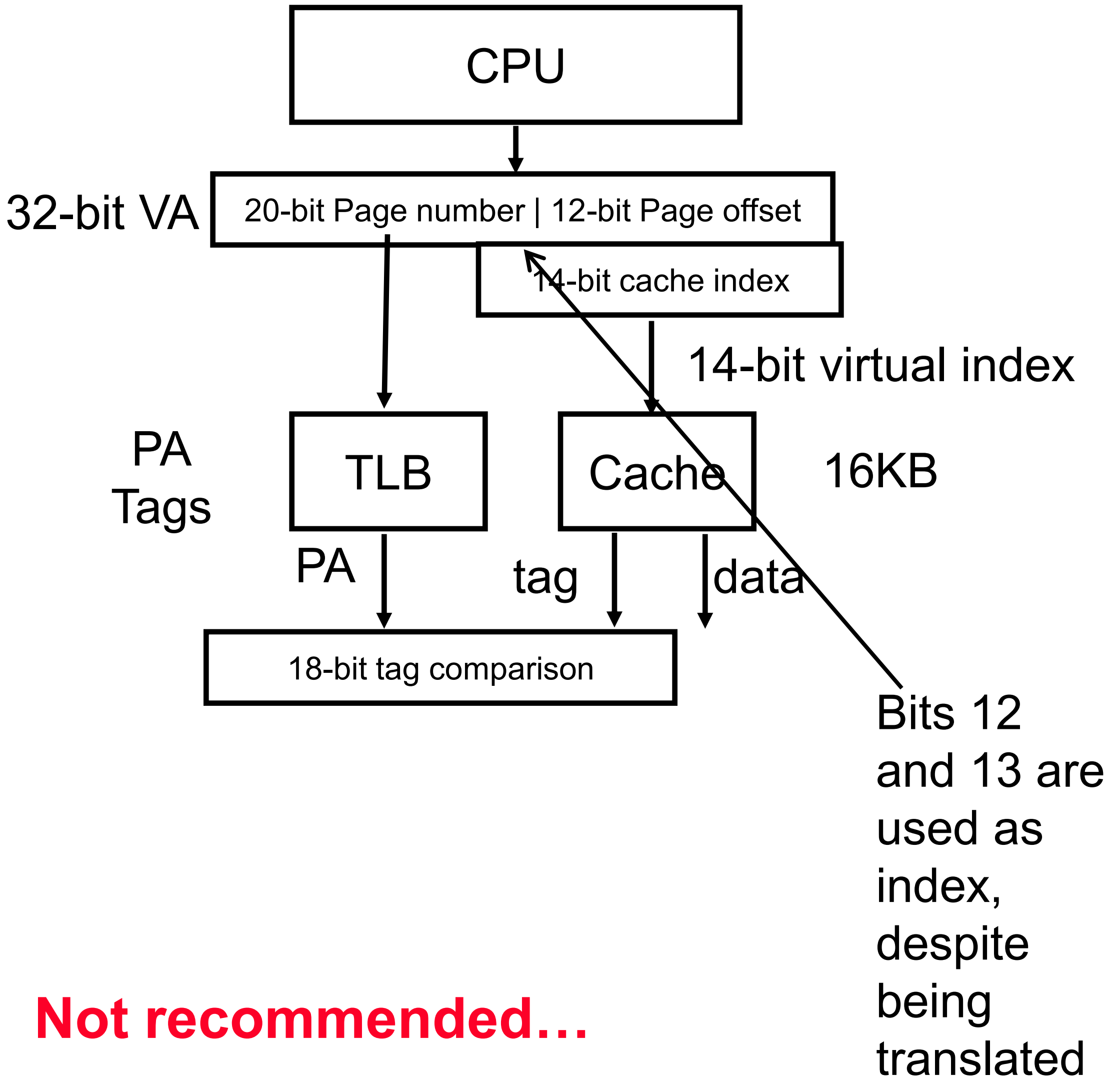
Index with physical (untranslated) portion of address

- **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!

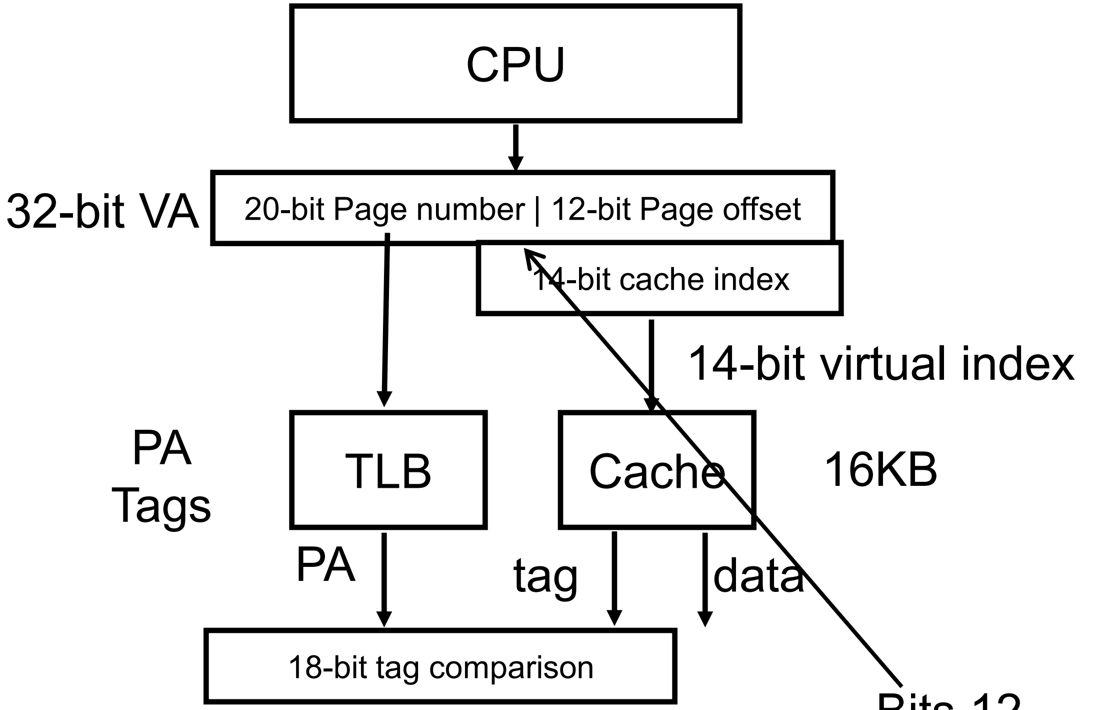


Not recommended…

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

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!



Bits 12 and 13 are used as index, despite being translated

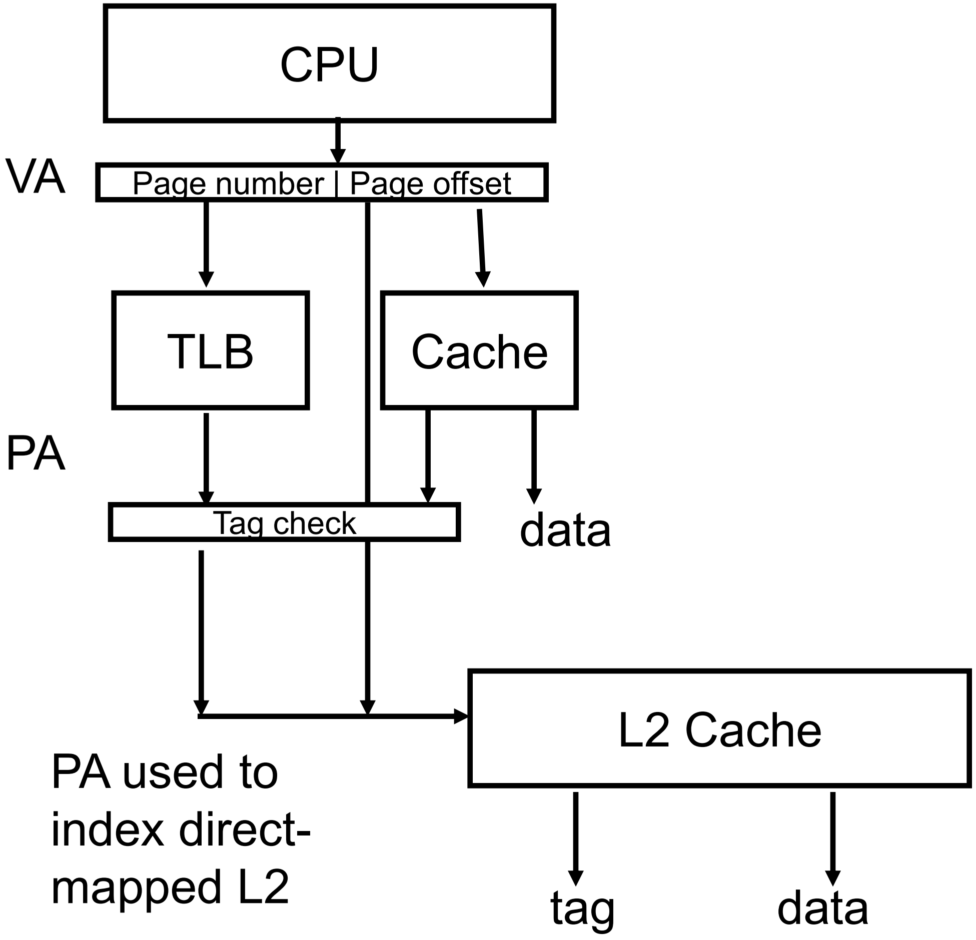
Not recommended…

https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/page-colouring-on-armv6-and-a-bit-on-armv7

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

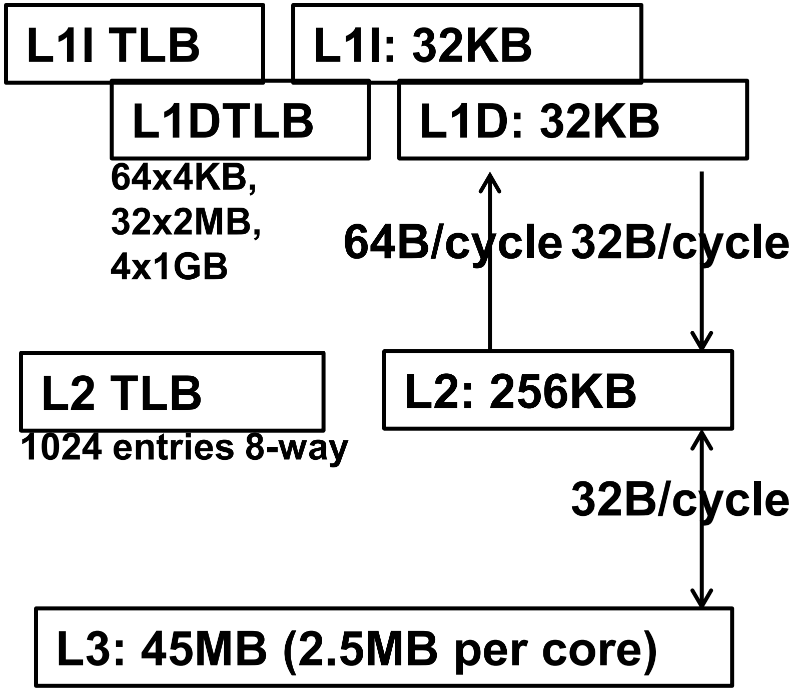
- The L2 cache is indexed with *translated* address
- So the L2 associativity conflicts depend on the virtual-to-physical mapping
- It would be helpful if the OS could choose non-conflicting pages for frequently-accessed data! **(page colouring for conflict avoidance)**
- Or at least, make sure adjacent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



PA used to index direct-mapped L2

- **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



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…..

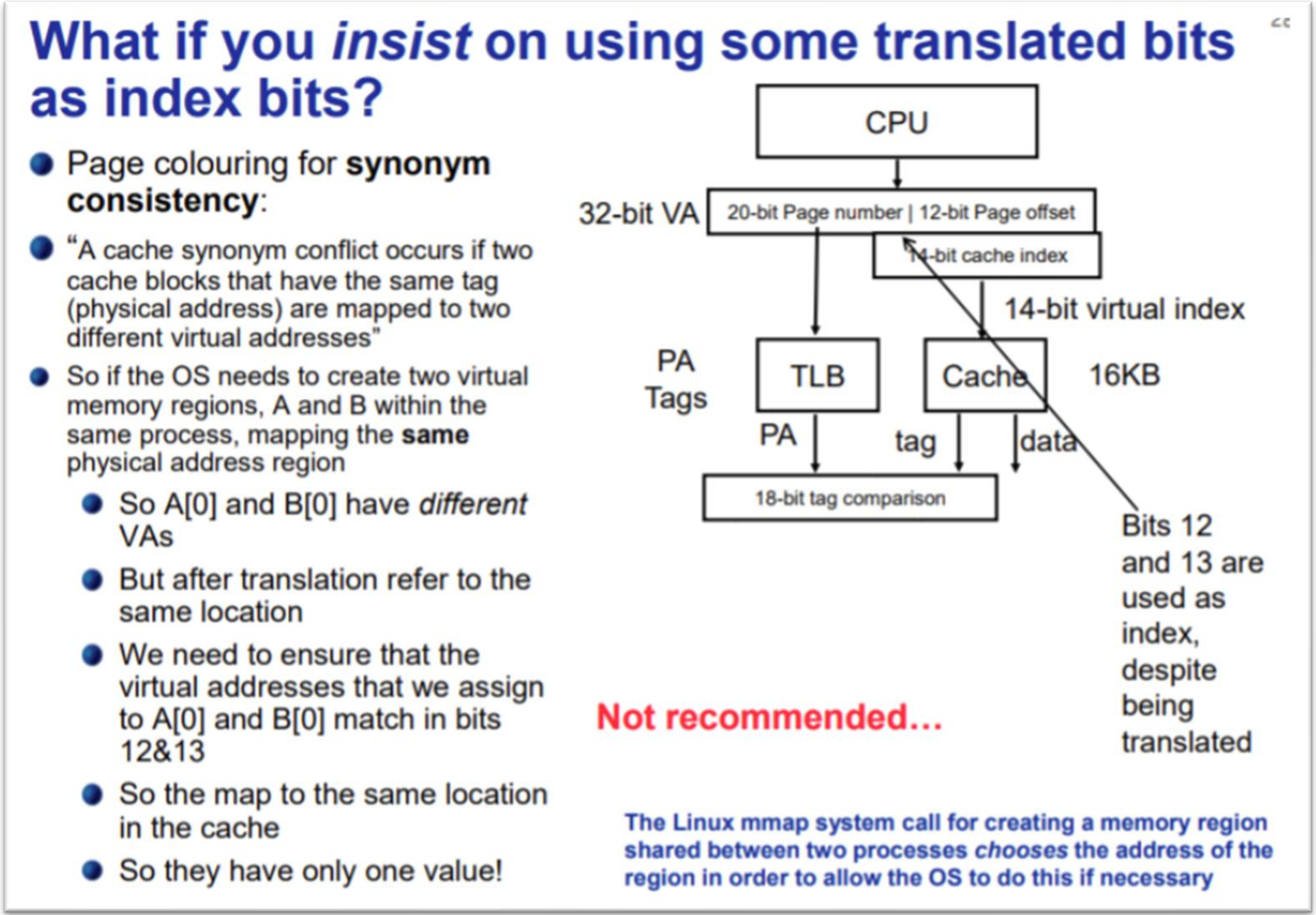
- Using a really small cache (and a larger next-level cache)
- With a pipelined cache (really only improves bandwidth)
- With a multi-bank cache (only increases bandwidth)
- By using a direct-mapped cache
- By passing data forward while checking tags in parallel
- Using way prediction (not covered in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



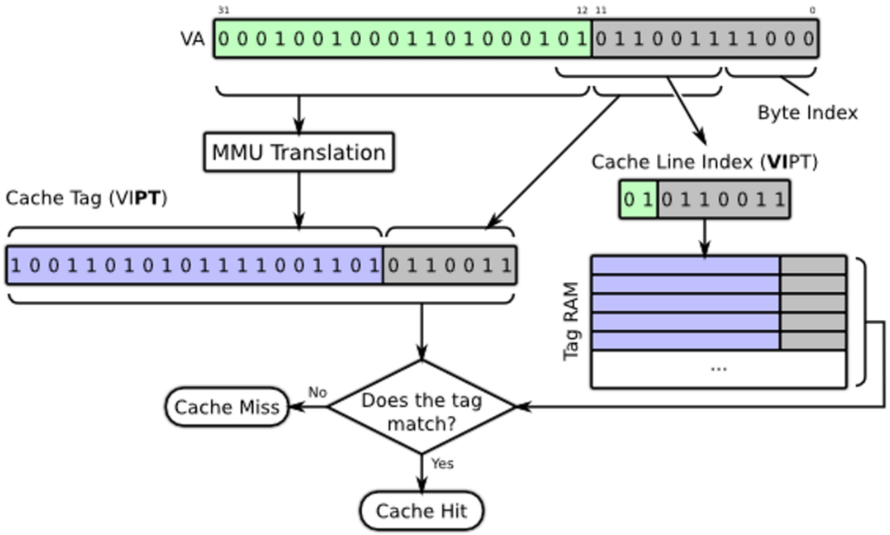
What if you *insist* on using some translated bits as index bits?

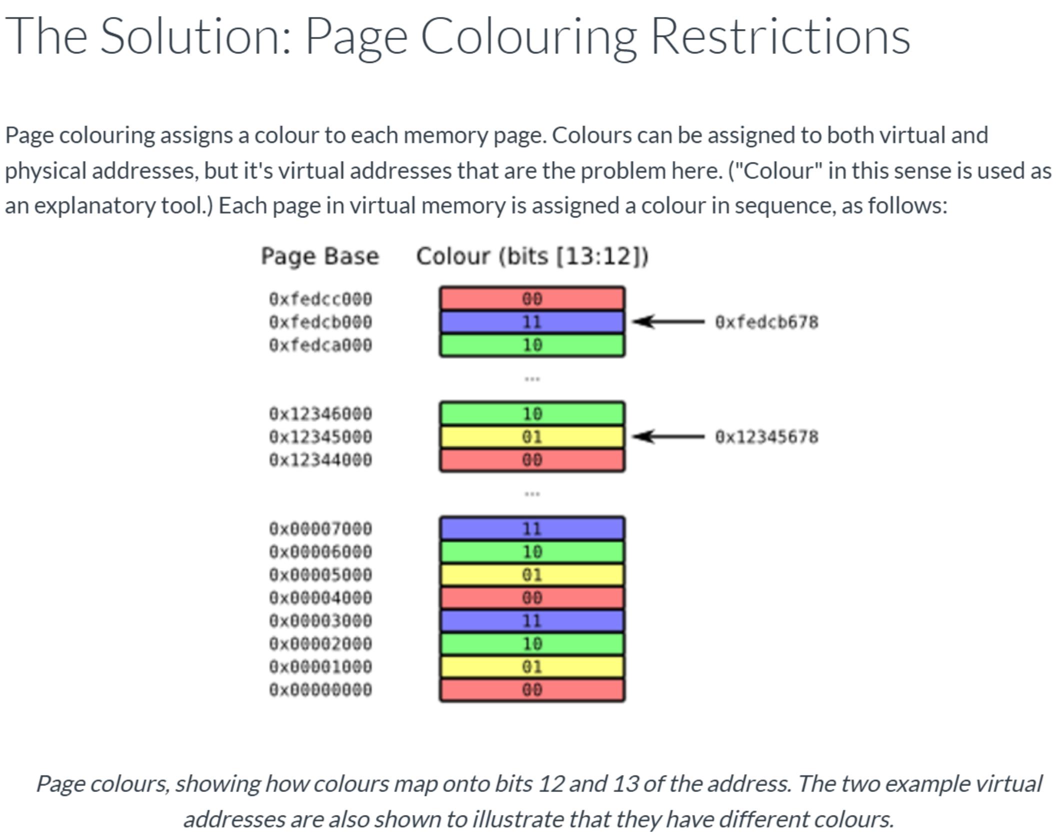
- Page colouring for **synonym consistency**:
- "A cache synonym conflict occurs if two cache blocks that have the same tag (physical address) are mapped to two different virtual addresses"
- So if the OS needs to create two virtual memory regions, A and B within the same process, mapping the **same** 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.





The Solution: Page Colouring Restrictions

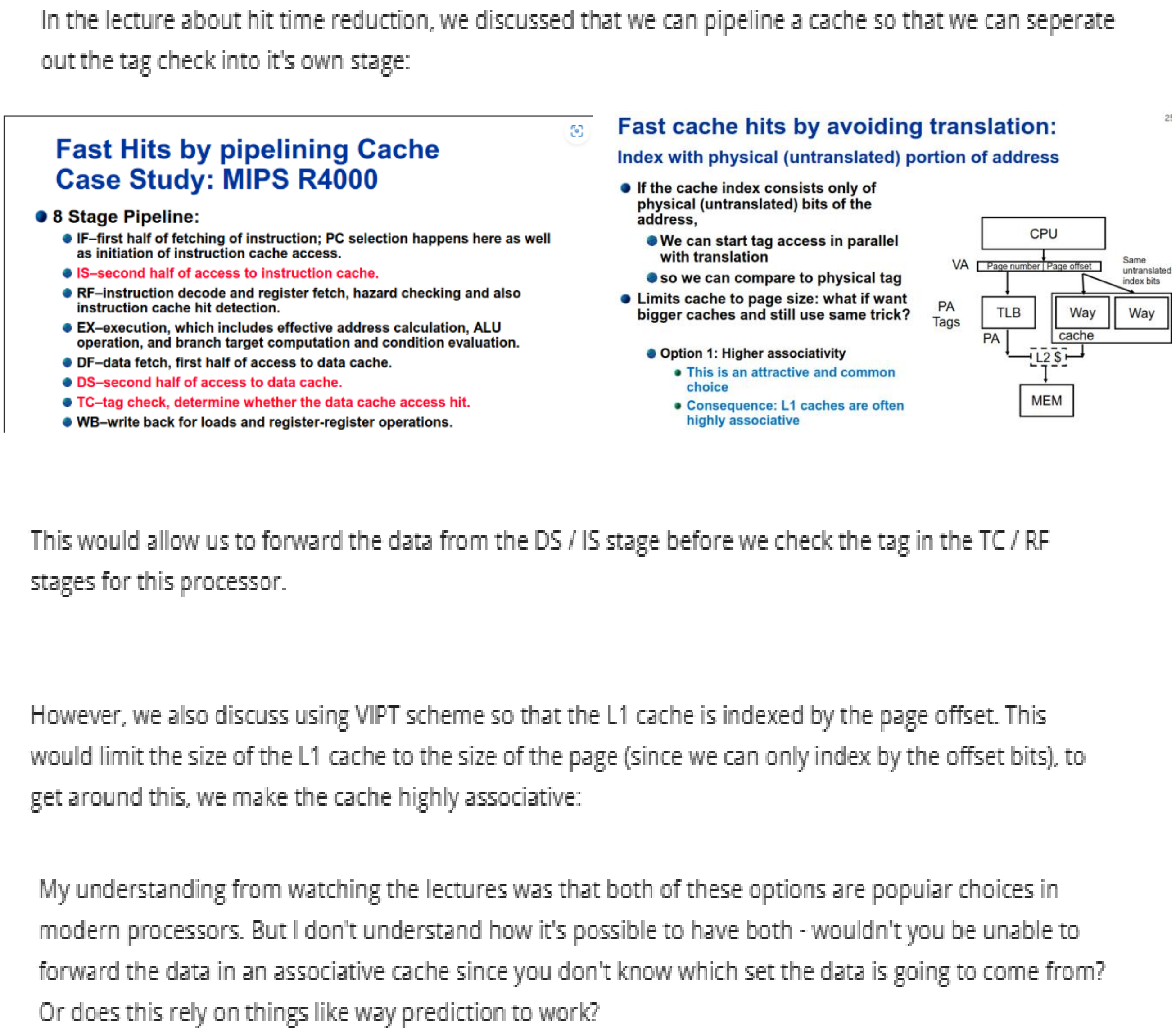
Page colouring assigns a colour to each memory page. Colours can be assigned to both virtual and physical addresses, but it's virtual addresses that are the problem here. ("Colour" in this sense is used as an explanatory tool.) Each page in virtual memory is assigned a colour in sequence, as follows:

Page Base	Colour (bits [13:12])	
0xfedcc000	00	
0xfedcb000	11	← 0xfedcb678
0xfedca000	10	
	...	
0x12346000	10	
0x12345000	01	← 0x12345678
0x12344000	00	
	...	
0x00007000	11	
0x00006000	10	
0x00005000	01	
0x00004000	00	
0x00003000	11	
0x00002000	10	
0x00001000	01	
0x00000000	00	

Page colours, showing how colours map onto bits 12 and 13 of the address. The two example virtual addresses are also shown to illustrate that they have different colours.

Page Colouring on ARMv6 (and a bit on ARMv7) - Architectures and Processors blog - Arm Community blogs - Arm Community https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/page-colouring-on-armv6-and-a-bit-on-armv7

Student question: way prediction



In the lecture about hit time reduction, we discussed that we can pipeline a cache so that we can seperate out the tag check into it's own stage:

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.

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

This would allow us to forward the data from the DS / IS stage before we check the tag in the TC / RF stages for this processor.

However, we also discuss using VIPT scheme so that the L1 cache is indexed by the page offset. This would limit the size of the L1 cache to the size of the page (since we can only index by the offset bits), to get around this, we make the cache highly associative:

My understanding from watching the lectures was that both of these options are popular choices in modern processors. But I don't understand how it's possible to have both - wouldn't you be unable to forward the data in an associative cache since you don't know which set the data is going to come from? Or does this rely on things like way prediction to work?

Yes this is exactly right:

•VIPT makes you really really want high associativity in the L1 data cache (actually also the instruction cache)

•The tag check is likely done a cycle after the cache provides the data (because only then do we have the tag, and also only then do we have the translated (physical) address against which to compare the tag.

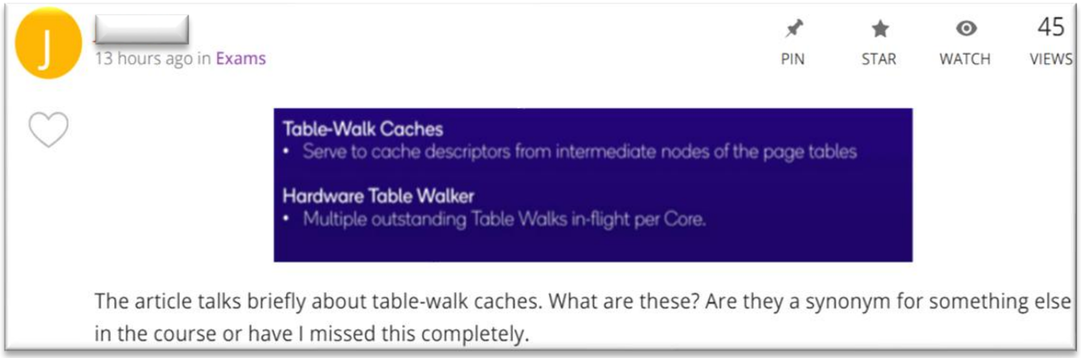
•With a direct-mapped cache we could still forward the data, before the tag check is done, and squash if the tag check fails (in which case we have a cache miss).

•With an associative cache, you don't know which way's data to forward

So a popular solution to this is way prediction: we have a small fast predictor that guesses which is going to be the right way, so we can forward it right away.

See Hennessy and Patterson 6th edition section 2.2 pp81.
Sarangi's book provides a bit more detail, see section 7.4.4 pp307 (https://www.cse.iitd.ac.in/~srsarangi/advbook/chapters/caches.pdf)

Student question: page-table walk



[redacted]

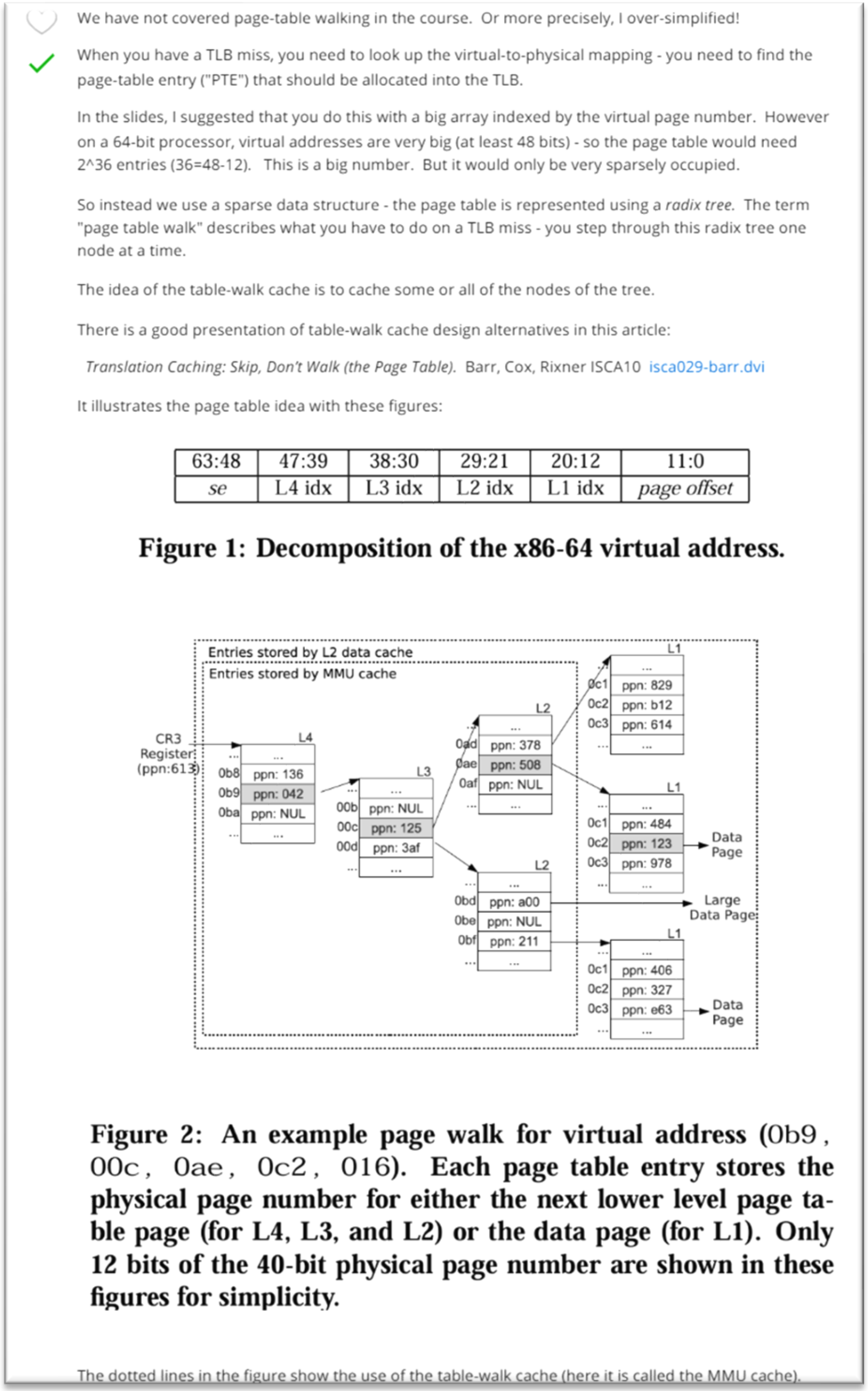
13 hours ago in Exams

Table-Walk Caches
- Serve to cache descriptors from intermediate nodes of the page tables

Hardware Table Walker
- Multiple outstanding Table Walks in-flight per Core.

The article talks briefly about table-walk caches. What are these? Are they a synonym for something else in the course or have I missed this completely.

This question is about the Qualcomm Snapdragon X Elite (Oryon) architecture which was the subject of the 2024 exam.



We have not covered page-table walking in the course. Or more precisely, I over-simplified!

When you have a TLB miss, you need to look up the virtual-to-physical mapping - you need to find the page-table entry ("PTE") that should be allocated into the TLB.

In the slides, I suggested that you do this with a big array indexed by the virtual page number. However on a 64-bit processor, virtual addresses are very big (at least 48 bits) - so the page table would need 2^36 entries (36=48-12). This is a big number. But it would only be very sparsely occupied.

So instead we use a sparse data structure - the page table is represented using a *radix tree*. The term "page table walk" describes what you have to do on a TLB miss - you step through this radix tree one node at a time.

The idea of the table-walk cache is to cache some or all of the nodes of the tree.

There is a good presentation of table-walk cache design alternatives in this article:

Translation Caching: Skip, Don't Walk (the Page Table). Barr, Cox, Rixner ISCA10 isca029-barr.dvi

It illustrates the page table idea with these figures:

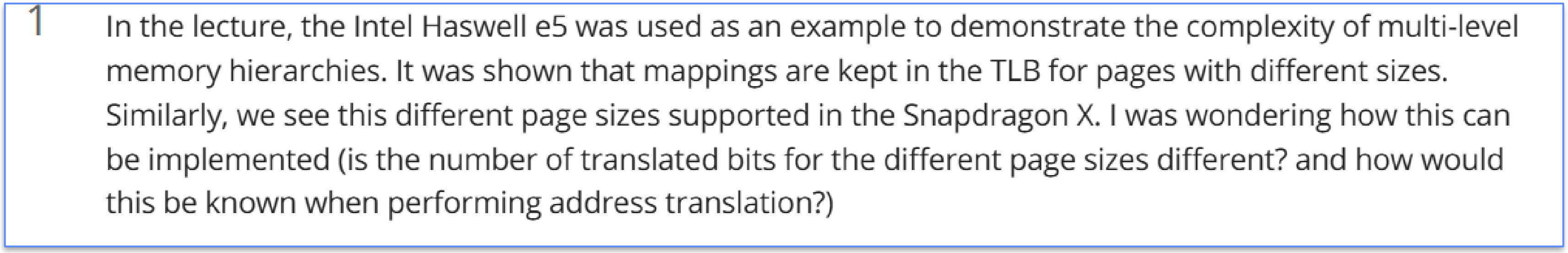
63:48	47:39	38:30	29:21	20:12	11:0
se	L4 idx	L3 idx	L2 idx	L1 idx	*page offset*

Figure 1: Decomposition of the x86-64 virtual address.

Figure 2: An example page walk for virtual address (0b9, 00c, 0ae, 0c2, 016). Each page table entry stores the physical page number for either the next lower level page table page (for L4, L3, and L2) or the data page (for L1). Only 12 bits of the 40-bit physical page number are shown in these figures for simplicity.

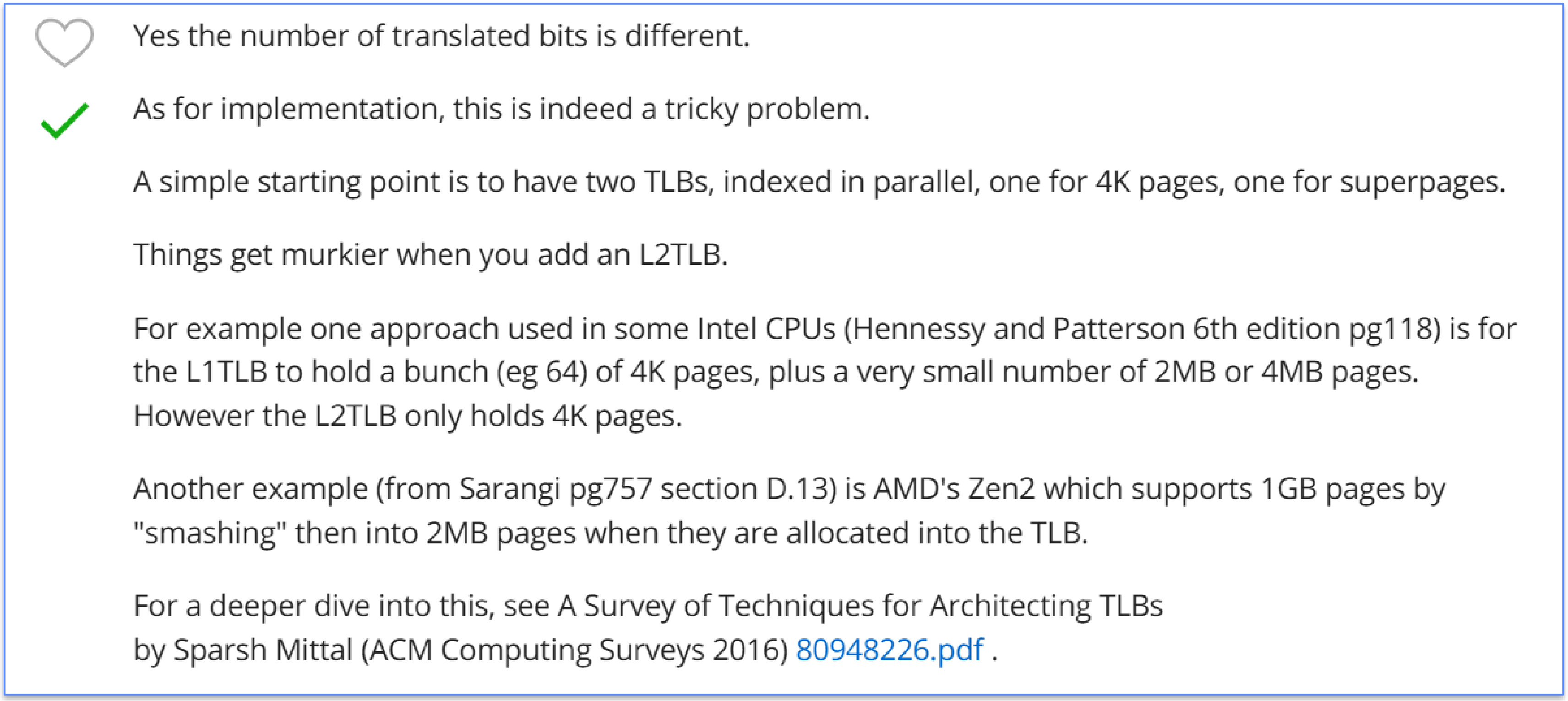
The dotted lines in the figure show the use of the table-walk cache (here it is called the MMU cache).

Student question: multiple page sizes



1

In the lecture, the Intel Haswell e5 was used as an example to demonstrate the complexity of multi-level memory hierarchies. It was shown that mappings are kept in the TLB for pages with different sizes. Similarly, we see this different page sizes supported in the Snapdragon X. I was wondering how this can be implemented (is the number of translated bits for the different page sizes different? and how would this be known when performing address translation?)



Yes the number of translated bits is different.

As for implementation, this is indeed a tricky problem.

A simple starting point is to have two TLBs, indexed in parallel, one for 4K pages, one for superpages.

Things get murkier when you add an L2TLB.

For example one approach used in some Intel CPUs (Hennessy and Patterson 6th edition pg118) is for the L1TLB to hold a bunch (eg 64) of 4K pages, plus a very small number of 2MB or 4MB pages. However the L2TLB only holds 4K pages.

Another example (from Sarangi pg757 section D.13) is AMD's Zen2 which supports 1GB pages by "smashing" then into 2MB pages when they are allocated into the TLB.

For a deeper dive into this, see A Survey of Techniques for Architecting TLBs
by Sparsh Mittal (ACM Computing Surveys 2016) 80948226.pdf .