Translation Lookaside Buffers

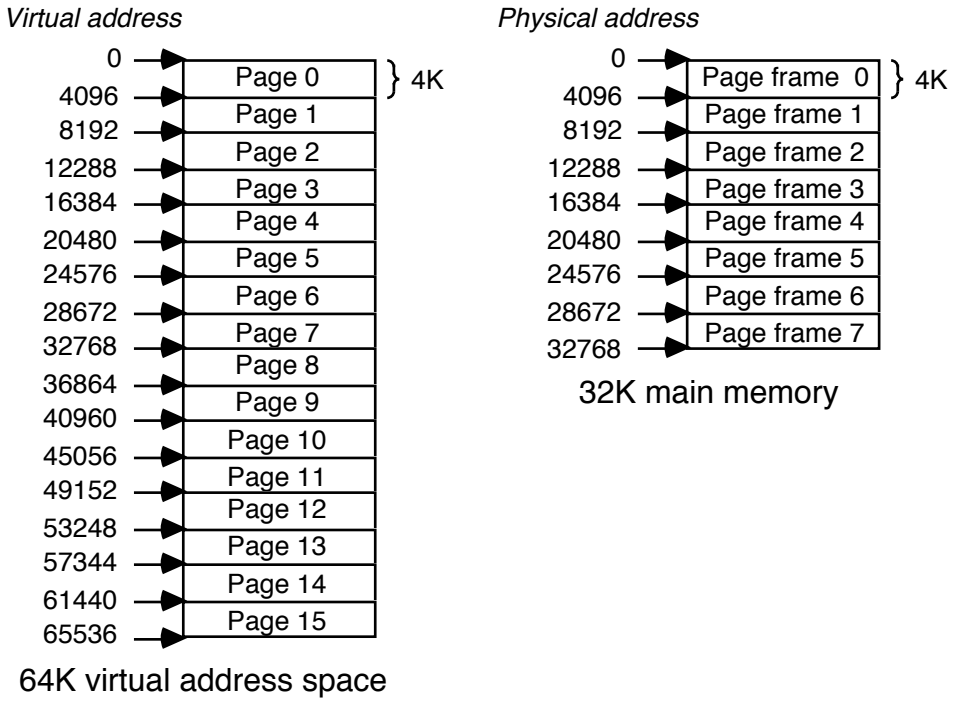
When (paged) virtual memory is in use, addresses must be *translated* before being used. As we shall see, address translation makes use of a *translation lookaside buffer* (TLB) that is structured very much like an L1 cache.

Motivation for virtual memory: Allow programs to be run concurrently that are too large to fit completely into main memory at the same time.

(We could just swap a program out of main memory and bring another program in. Why don't we do this?

Pages are units of virtual memory. Physical memory is divided into page-sized units that are called *page frames.* Here is a diagram of—

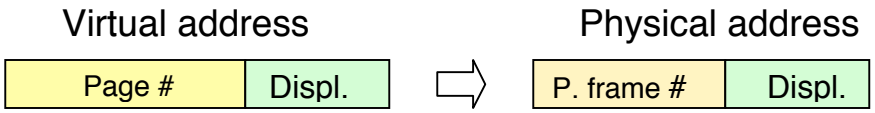
- a 64K virtual address space divided into sixteen pages of 4K each
- a 32K main memory divided into eight page frames of 4K each



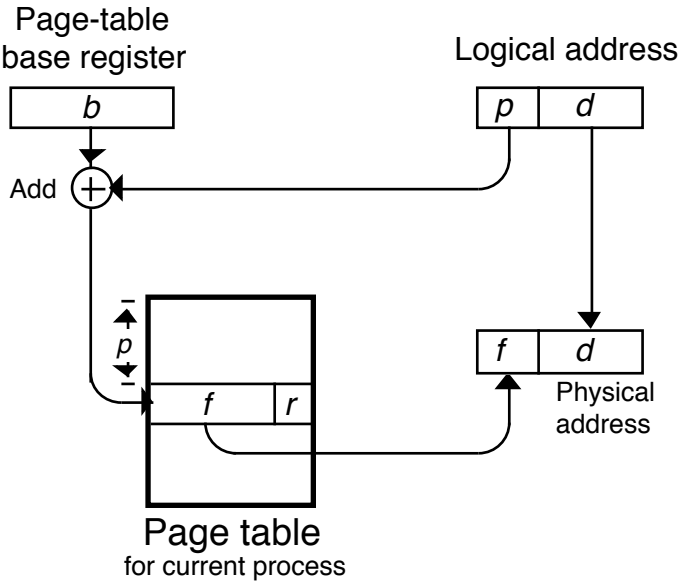
(Of course, today's memories are much larger than this.)

A virtual address consists of a page # and displacement within the page.

The page number is *translated* to a page-frame number.



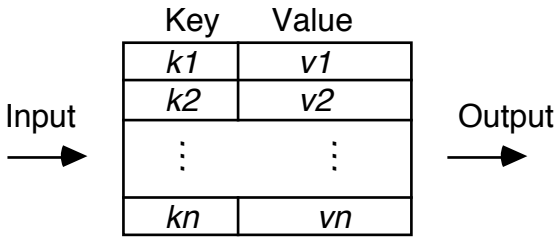
This translation is done through a *page table*.



A page table is a data structure in main memory. Do you see a problem with this?

Instead, a form of cache is used.

- This “cache” is smaller than a page table.
- The input is compared against all keys, using direct-mapped, set-associative, or fully associative logic. If the input matches a key, then the corresponding value is output.



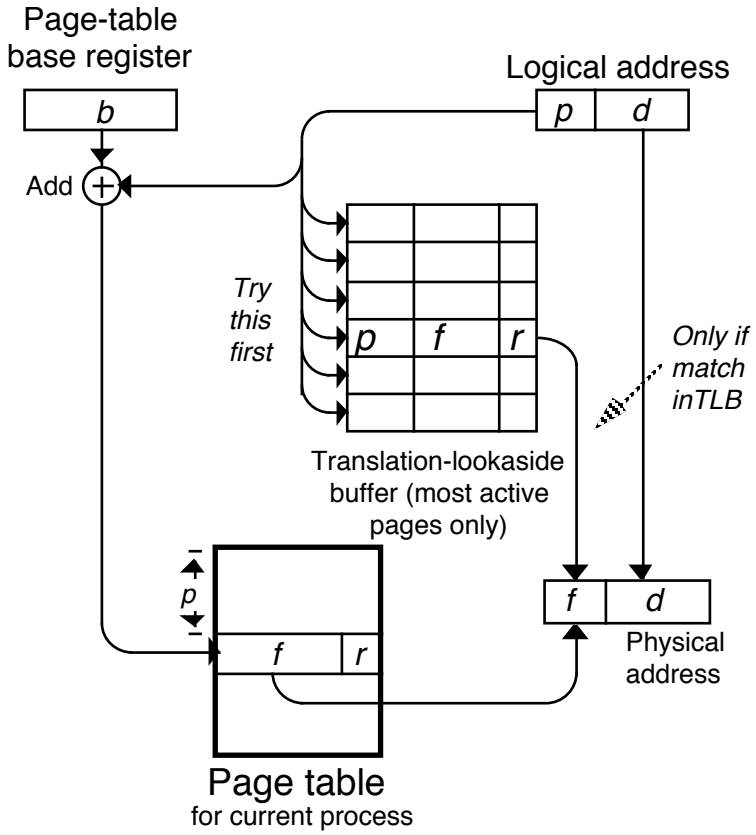
This memory is called a *translation lookaside buffer*, or TLB, because the processor “looks aside” to it as it is translating an address.

Below is a diagram of address translation using a TLB.

Each time the TLB is accessed, this procedure is followed:

- If an entry matching the page number is found, the page-frame address is returned.
- If no matching entry is found, a *victim* entry is chosen from among the TLB slots.

The victim is chosen based upon the replacement policy of the TLB.



A TLB's *hit ratio* is the number of entries found / the number of times searched.

Improving Cache Performance

There are three basic approaches to improving cache performance.

- Reducing the miss penalty [H&P §5.4]

  Hybrid-access caches (victim, MRU)
  Multilevel caches
  Write buffers
  Early restart
  Critical word first
  Subblocking

- Reducing the miss ratio [H&P §5.5]

  Block size, cache size, associativity
  Prefetching: Hardware, Software
  Data layout: Instructions, Data

- Reducing the hit time [H&P §5.7]

  Avoiding address translation (TLB accesses in parallel)
  Using simple caches, small caches
  Pipelining writes

Topics listed in dark blue have already been covered in Lectures 5 and 6.

Miss-penalty reduction

Let's take a look at some other techniques for reducing the miss penalty.

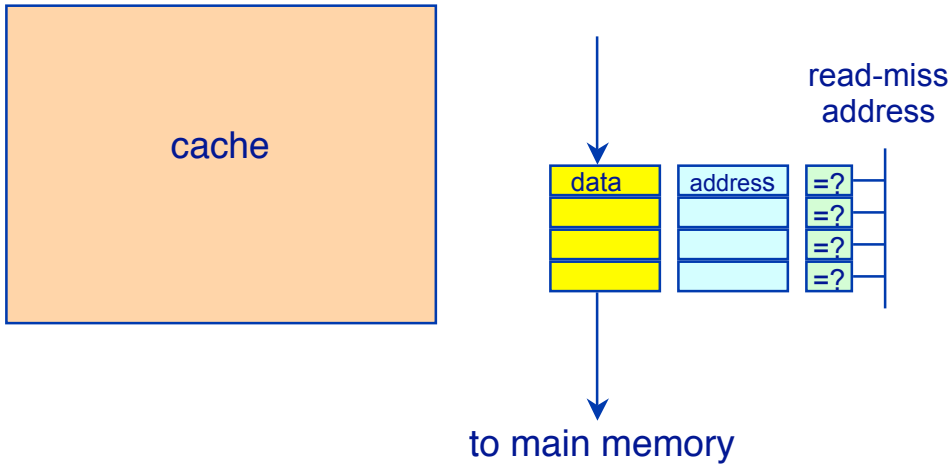
Write buffers

This technique is used with write-through or write-back.

The idea is not to make the CPU wait for the write to complete in memory.

Instead, data is written to a *write buffer*, and the processor can continue while it is being written to memory.

How must handling of a read miss change when a write buffer is in use?



Going one step further, we can use a *merging* write buffer. In this case, a buffer entry will be several words long, like a cache line.

Writes to multiple words in this line can share space in the same buffer entry, and be written to memory at the same time. (See H&P, p. 412 for details).

Early restart and critical-word first

Early restart: It's not necessary to wait until the cache block is completely read before restarting the processor.

As soon as the requested word arrives, it can be forwarded to the CPU.

The miss penalty is now the time to fetch the requested word, not the entire block.

Critical-word first: Early restart takes care of restarting the processor when the requested word arrives. But that may still be quite a long time. Why?

To handle this case better, we can start fetching the block with the required (critical) word, and fill in the rest later.

Example: Suppose a computer has a 128-byte cache block (containing sixteen 64-bit words). It takes 12 clock cycles to get the critical word (or the *first* word, if critical-word first is not in use), and then two clock cycles per word to fetch the rest of the block.

- How long does it take to read a whole block, without critical-word first?
- How long does it take to read a whole block, assuming critical-word first?

With critical word first, it takes ___ cycles to get the critical word.

Without critical word first,

To get the rest of the block,

- with critical-word first,
- without critical-word first,

Subblocking

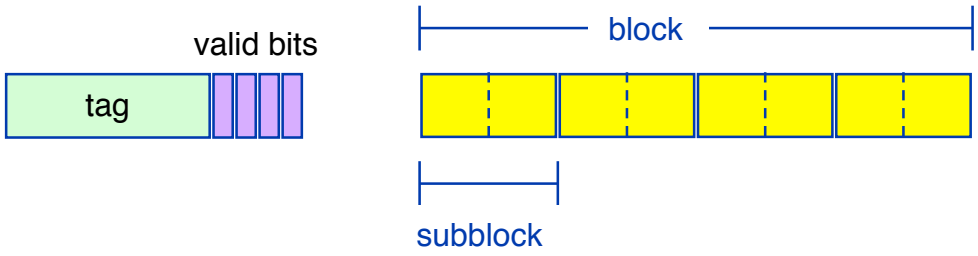
Problem: Tags are overhead; they take up extra space.

Partial solution: Large blocks reduce the amount of tag storage.

- Say we keep cache size fixed, but double block size
- Then the number of blocks is halved
- *The number of tags is also halved* (# tags = # blocks)
- So, we've reduced amount of tag overhead, conserving space on the chip.
- *but* large blocks increase the cache miss penalty

Complete solution: Use large blocks, but also divide blocks into smaller "subblocks."

- Fetch only 1 subblock on a miss
- Keep valid bits for subblocks
- Better if other subblocks are prefetched in the background (that is, combine subblocking with *early restart* and *critical word first*).



We have seen an example of subblocking before in this course. What is it?