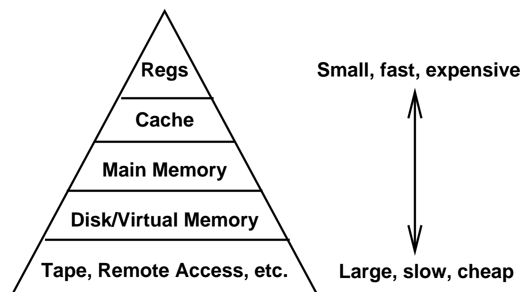
Computer Science 2500
Computer Organization
Rensselaer Polytechnic Institute
Spring 2009

Topic Notes: Memory Hierarchy

Our next topic is one that comes up in both architecture and operating systems classes: memory hierarchies.

We have thought of memory as a single unit – an array of bytes or words. From the perspective of a program running on the CPU, that's exactly what it looks like.

In reality, what we think of as "main memory" is just part of a hierarchy:



We have already considered how the use of registers is different from main memory, in particular for the MIPS ISA, where all computation must operate on values from, and store results in, registers.

From P&H, we have the technologies, access times, and costs as of 2008 for these types of memory:

Technology	Access time	Cost per GB
SRAM	0.5-2.5 ns	$2000-$5000
DRAM	50-70 ns	$20-75
disk	5-20 ms	$0.20-$2

Note in the last line that this is the equivalent of 5,000,000 to 20,000,000 ns.

We will consider:

- the layer (or layers, in most cases) of *cache*: faster memory between registers and main memory
- *virtual memory*, which allow us to pretend to have more memory than we really have by using disk space to store parts of memory not currently in use

Caches

We have noted from time to time in the course that memory access is very slow compared to registers and the ALU.

In reality, the difference between processor speeds and memory speeds is even greater than we have considered and growing.

- For decades, CPU performance (clock speed) doubled every 18-24 months
- Memory speeds have increased linearly, at best
- Improvements in memory technology have focused on increased density rather than speed: address space increases by 1.5 bits per year—a factor of four every 18-24 months

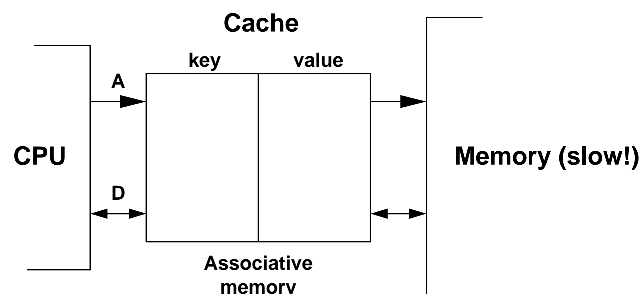
Early vonNeumann architectures could rely on memory producing a value nearly as quickly as the CPU could operate on it. Today, we are very far from that.

Caches help keep the necessary values close to the CPU for faster access:

- A small, fast memory is placed between the CPU and main memory (likely multiple levels)
- whenever a value is needed for an address, we first look for it in the cache – only if it is not there do we look to main memory and copy it to cache
- when we write a value, we write it to cache, then update it in memory later, concurrently with other computations
- terminology: data found in cache is a *cache hit*, not found is a *cache miss*

Our goal is to avoid memory accesses as much as possible.

The cache is an *associative memory*, where we can look up a memory value by its address in a table.



The key is the address, value is the contents of memory at that address.

When the CPU makes a memory request, the cache needs to find the appropriate value very quickly.

- Ideally, we can look up the etries that match the key instantly
- Realistically, there is a hash function that maps addresses to a set of possible cache entries
- Depending on the hash function, we may or may not need to store the key itself in the cache
- The hash function depends on how we organize the cache
- Generally there is an organization into *lines* of cache – groups of addresses that are stored (and read/written) in the cache together
  - typical: 16 or 32 bytes
  - this works well with the idea of a wider memory bus (recall discussion when we built memory circuits and organized it into banks) – it is very convenient if the memory can read/write cache-line size chunks

---

Direct Mapped Caches

The first cache organization we consider allows for direct access to a cache line from memory by breaking the address down into three parts:

```
101 10111 1011
```

- the first chunk is the key
- the middle chunk is the cache line (or *cache address* or *index*)
- the last chunk specifies the entry within the cache line

In this case, we would have 16 addresses per line in the cache, and 32 lines of cache.

$\frac{1}{8}$ of our memory actually fits in cache, but each address, if it is in the cache, is in a specific location.

- we need only check if the entry at the desired line is the actual address we want – this is a fast comparison
- if the desired line is currently in the cache, we just use it
- if not, we have to go get it, evicting the old value from the cache line
- if we repeatedly use values from two different parts of memory that happen to map to the same cache line, we will have a lot of cache misses

---

Fully Associative Caches

Here, a cache entry is not restricted to a single location. In fact, it can be in any line.

We break our address into only two chunks for a fully associative cache:

```
10110111 1011
```

The key is all bits above the number of bits to specify an entry within a line.

- when looking up, we have to check *every* entry in the cache to see if it's in there
- this sounds like a search (think: expensive), but we can build a circuit for this
- if we find it, use it, otherwise, we need to go get it
- But then which line do we replace? It can go anywhere!

  We can consider many approaches to select a "victim" for eviction –

  1. LRU – least-recently used
  2. LRU approximations
  3. random
  4. saturating counters – keep track of frequency of recent access
  5. replace non-dirty (unmodified) lines

  The decision must be fast but a poor decision will lead to another cache miss in the near future – also expensive
- we are much less susceptible to an unfortunate access pattern compared to direct mapping since the associativity means flexibility

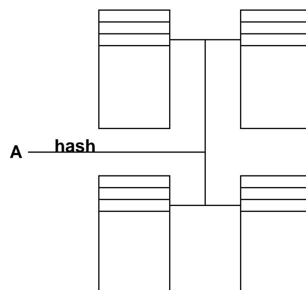
---

Set Associative Caches

In between the other approaches – each address maps to a small set of possible lines.

- for 2-way, each address could be in one of two places
- easier to check than in the fully associative case, but if we are using 3 or 4 lines repeatedly, we will end up with lots of cache misses
- but still better off than the direct mapped

We will almost definitely do lookups in parallel, as in this 4-way example:



There are tradeoffs and all three approaches (and various levels of associativity within set associative) are really used.

Cache Discussion

There are three ways that a memory value will fail to be found in cache:

1. a *compulsory* miss—the first time a value is needed, it must be retrieved from memory
2. a *conflict* miss—since our last access another value that maps to the same cache location has evicted the value we want
3. a *capacity* miss—the value we want has been evicted not because of a direct conflict, but because it was the least recently used value

Notice that, looking at a stream of memory references, it is difficult to identify the reasons for misses; they are equally difficult to avoid.

See diagrams of cache organization for direct-mapped (Figure 5.7) and 4-way set associative (Figure 5.17).

Some cache statistics:

- Most primary caches hold a small amount of data—8 to 32K bytes – amazingly this value has been fairly constant for 40+ years
- a good primary cache delivers better than 95% of requests
  - Why? **Locality, locality, locality.**
  - *Spatial locality* and *temporal locality* – it's why caches work so well (and why computers are anywhere near as fast as they are today given the discrepancy between CPU speeds and memory speeds)
  - spatial locality: when we access a memory location we're likely to access other items in nearby memory locations in the near future (groups of local variables, arrays, code for a subroutine or loop)
  - temporal locality: when we access a memory location, we're likely to access that location again in the near future (code in a loop, loop index or accumulator variables)
- missing 2% of requests on to a memory that is 50 times slower (and it is often much worse) means the average access time is $0.98 + 0.02 * 50 = 1.98$ cycles, or half of the desired speed – we must have a high *hit rate* (i.e., low *miss rate*)
- Secondary (and tertiary) caches can be effective at reducing miss rate, but they must be much larger: secondary caches might be 256K bytes or larger, while tertiary caches are measured in megabytes

Comparing the cache organizations:

- Fully Associative is much more flexible, so the miss rate will be lower
- Direct Mapped requires less hardware (cheaper) and will also be faster!
- Tradeoff of miss rate vs. hit time
- Levels of associativity are the compromises

Consider this extreme example:

```
for (i=0;i<10000;i++) {
    a[i] = a[i] + a[i+64] + a[i+128];
    a[i+64] = a[i+64] + a[i+128];
}
```

`a[i]`, `a[i+64]` and `a[i+128]` belong to the same set for a direct-mapped cache with 64 entries, but if we instead organize as a 4-way set associative cache, we're OK, since we can hold all 3 in the cache at the same time.

Cache management

- instructions are read-only – motivation for *Harvard* cache, where data and instructions are separated
  - avoid need to worry about write policies (see below), at least for instructions
- data is mutable (usually) – need to keep track of *dirty* lines – those that have been changed
- if a value has changed, it must be written back to memory some time (at least before it gets evicted)
- the line never changes, no need to write it back – avoid unnecessary writebacks?
- Write policies:
  - Write-through – memory is always written back but usually only when the MMU is not otherwise busy
  - Write-back – memory is updated only when the line is evicted
  - probably need some sort of "write queue" in either case
  - and what if you write before you read? (variable initialization)
    - read it into the cache first?
    - perhaps "write-around" for writes to lines not in cache?

    * how likely are we to be reading them soon anyway?

- Another technique: *victim cache*
  - a place to keep recently evicted lines
  - these are good candidates for reinsertion
  - at least keep them here until written back
  - especially useful for direct-mapped caches, can benefit set associative caches

---

Multiprocessor issues – *cache coherency*

For example, the Power 4 architecture has 4 cores, each with its own L1 cache, but the 4 share an L2 cache

- if a line is in 2 L1 caches, how do we mantain consistency?
- do lines in an L1 also exist in L2?

This topic would be investigated extensively in an advanced architecture course (and somewhat in operating systems).

Handling Cache Misses in the Datapath and Control

When we have a cache hit, our datapath and control (be it single-cycle or pipelined) works as we saw previously.

In the case of a cache miss, a requested memory value (instruction or data) is not immediately available. We must wait an appropriate amount of time for the value to be placed into the cache before we can obtain the desired value.

The basic idea in either datapath/control approach is to stall the processor when a cache miss is detected. We wait for the memory values to populate the appropriate cache line, then restart the instruction in question.

The processor can continuously attempt to fetch the value until it works (it's a hit). During this time, bubbles are inserted into the pipeline.

Virtual Memory

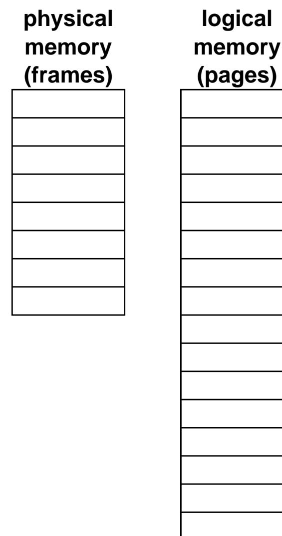
Another place caching is very common is to use main memory as a cache for data being read from and written to a disk. Main memory is slow compared to registers and cache, but is very fast compared to disks (where we have to wait for *gasp* actual mechanical parts).

But we'll now consider just the opposite – the use of disk to store values that we want to pretend exist in main memory, but for which there is not enough main memory.

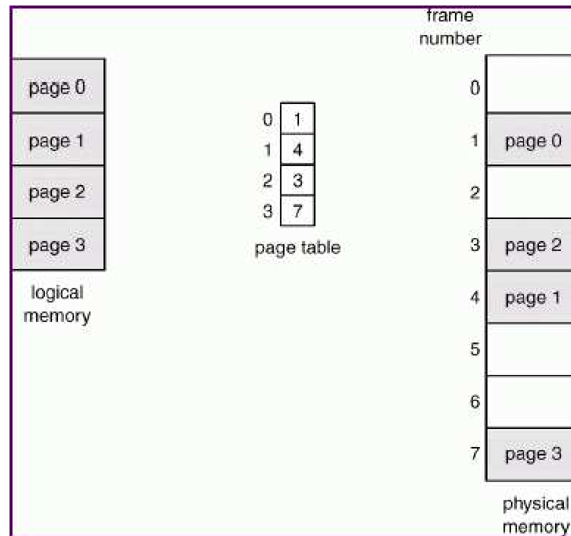
When the memory requirements of a program (or the sum of the requirements of the many programs which may be running on a system) are larger than physical memory, we depend on this idea – *virtual memory*.

We will just consider the basics – look forward to Operating Systems for a more thorough discussion.

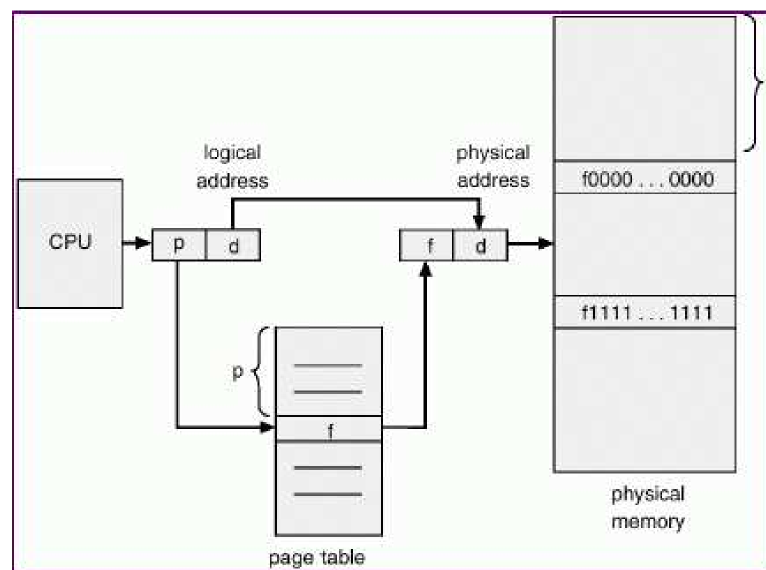
- Programs generate (that is, the CPU puts onto the address bus) *logical addresses*
- Each program in a multiprogrammed environment may have its own private *address space*
- Even a single program can have a larger address space than the available physical memory
- Basic goal: assign logical addresses to physical addresses, translate logical addresses to physical addresses when accessed
- Logical memory is broken into *pages*, usually around 4K in size, often stored on secondary store (disk)
- Pages are brought into *frames* of physical memory of the same size



- Operating system keeps track of a map called a *page table*

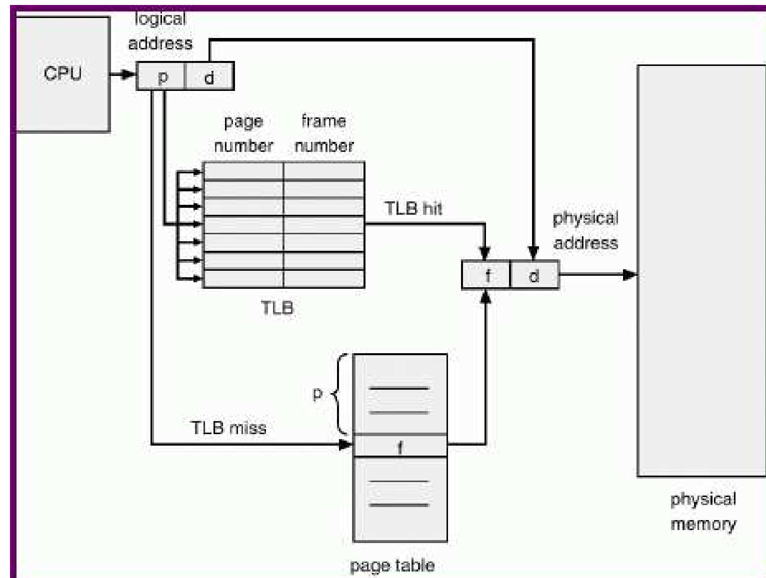


- Entries in a page table either give a disk address for a non-resident page or the information to translate the logical address to the appropriate physical address



- Accessing a non-resident page causes a *page fault*
  - analogous to a cache miss but much more costly (we're talking millions of CPU cycles)
  - bring in the page of logical memory into a frame of physical memory, possibly displacing a page currently in physical memory
  - takes long enough that the OS is likely to let a different process use the CPU while the page fault is being serviced
- Looking up page table entries is expensive (it is, after all, an additional memory access for each "real" memory access), and often is supported by special hardware

- Specifically, we usually keep a (very) small cache of translations called a *translation lookaside buffer (TLB)* that is responsible for servicing a majority of lookup requests



A TLB hit is still more expensive than a direct memory access (if we had no paging at all) but much better than the two references from before.

A TLB is typically around 64 entries: tiny, but good enough to get a reasonable hit rate (locality is your friend!).

- In large systems, the page table might not be memory resident, which leads to things like multilevel page tables, inverted page tables, and other ideas – come back for OS