

Random access memory (RAM) comes in two varieties—static and dynamic. Static RAM (SRAM) is faster and significantly more expensive than dynamic RAM (DRAM). SRAM is used for cache memories, both on and off the CPU chip. DRAM is used for the main memory plus the frame buffer of a graphics system. Typically, a desktop system will have no more than a few tens of megabytes of SRAM, but hundreds or thousands of megabytes of DRAM.

Accessing Main Memory

Data flows back and forth between the processor and the DRAM main memory over shared electrical conduits called *buses*. Each transfer of data between the CPU and memory is accomplished with a series of steps called a *bus transaction*. A *read transaction* transfers data from the main memory to the CPU. A *write transaction* transfers data from the CPU to the main memory.

the same set of wires or can use different sets. Also, more than two devices can share the same bus. The control wires carry signals that synchronize the transaction and identify what kind of transaction is currently being performed. For example, is this transaction of interest to the main memory, or to some other I/O device such as a disk controller? Is the transaction a read or a write? Is the information on the bus an address or a data item?

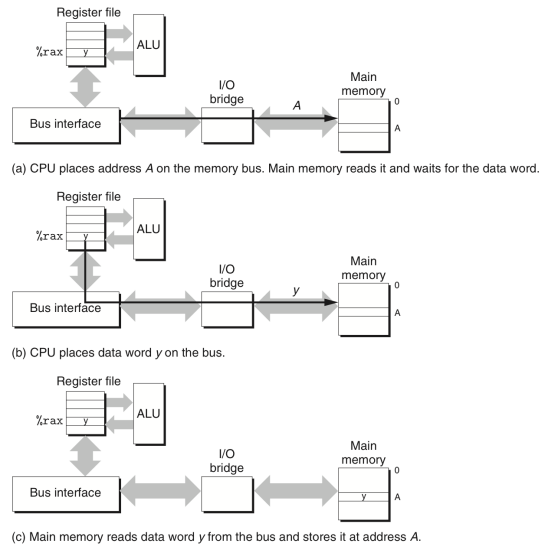
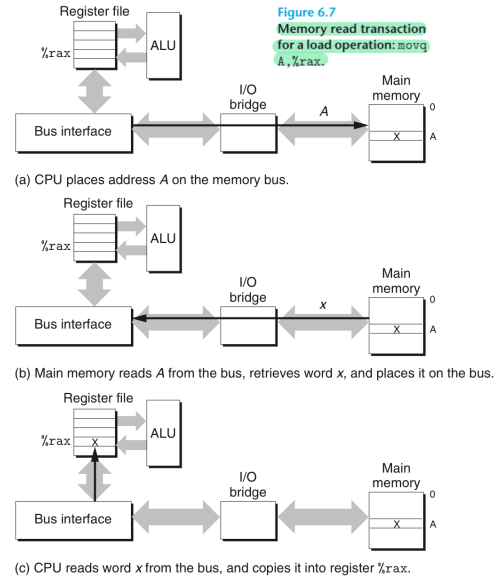


Figure 6.8 Memory write transaction for a store operation: `movq %rax, A`.

6.2 Locality

Well-written computer programs tend to exhibit good *locality*. That is, they tend to reference data items that are near other recently referenced data items or that were recently referenced themselves. This tendency, known as the *principle of locality*, is an enduring concept that has enormous impact on the design and performance of hardware and software systems.

Locality is typically described as having two distinct forms: *temporal locality* and *spatial locality*. In a program with good temporal locality, a memory location that is referenced once is likely to be referenced again multiple times in the near future. In a program with good spatial locality, if a memory location is referenced once, then the program is likely to reference a nearby memory location in the near future.

Cache Hits

When a program needs a particular data object d from level $k + 1$, it first looks for d in one of the blocks currently stored at level k . If d happens to be cached at level k , then we have what is called a *cache hit*. The program reads d directly from level k , which by the nature of the memory hierarchy is faster than reading d from level $k + 1$. For example, a program with good temporal locality might read a data object from block 14, resulting in a cache hit from level k .

Cache Misses

If, on the other hand, the data object d is not cached at level k , then we have what is called a *cache miss*. When there is a miss, the cache at level k fetches the block containing d from the cache at level $k + 1$, possibly *overwriting* an existing block if the level k cache is already full.

This process of overwriting an existing block is known as *replacing* or *evicting* the block. The block that is evicted is sometimes referred to as a *victim block*. The decision about which block to replace is governed by the cache's *replacement policy*. For example, a cache with a *random replacement policy* would choose a random victim block. A cache with a *least recently used (LRU)* replacement policy would choose the block that was last accessed the furthest in the past.

Kinds of Cache Misses

It is sometimes helpful to distinguish between different kinds of cache misses. If the cache at level k is empty, then any access of any data object will miss. An empty cache is sometimes referred to as a *cold cache*, and misses of this kind are called *compulsory misses* or *cold misses*. Cold misses are important because they are often transient events that might not occur in steady state, after the cache has been *warmed up* by repeated memory accesses.

Whenever there is a miss, the cache at level k must implement some *placement policy* that determines where to place the block it has retrieved from level $k + 1$. The most flexible placement policy is to allow any block from level $k + 1$ to be stored in any block at level k . For caches high in the memory hierarchy (close to the CPU) that are implemented in hardware and where speed is at a premium, this policy is usually too expensive to implement because randomly placed blocks are expensive to locate.

Thus, hardware caches typically implement a simpler placement policy that restricts a particular block at level $k + 1$ to a small subset (sometimes a singleton) of the blocks at level k . For example, in Figure 6.22, we might decide that a block i at level $k + 1$ must be placed in block $(i \bmod 4)$ at level k . For example, blocks 0, 4, 8, and 12 at level $k + 1$ would map to block 0 at level k ; blocks 1, 5, 9, and 13 would map to block 1; and so on. Notice that our example cache in Figure 6.22 uses this policy.

Restrictive placement policies of this kind lead to a type of miss known as a *conflict miss*, in which the cache is large enough to hold the referenced data objects, but because they map to the same cache block, the cache keeps missing.

Programs often run as a sequence of phases (e.g., loops) where each phase accesses some reasonably constant set of cache blocks. For example, a nested loop might access the elements of the same array over and over again. This set of blocks is called the *working set* of the phase. When the size of the working set exceeds the size of the cache, the cache will experience what are known as *capacity misses*. In other words, the cache is just too small to handle this particular working set.

Cache Management

As we have noted, the essence of the memory hierarchy is that the storage device at each level is a cache for the next lower level. At each level, some form of logic must *manage* the cache. By this we mean that something has to partition the cache storage into blocks, transfer blocks between different levels, decide when there are hits and misses, and then deal with them. The logic that manages the cache can be hardware, software, or a combination of the two.

For example, the compiler manages the register file, the highest level of the cache hierarchy. It decides when to issue loads when there are misses, and determines which register to store the data in. The caches at levels L1, L2, and L3 are managed entirely by hardware logic built into the caches. In a system with virtual memory, the DRAM main memory serves as a cache for data blocks stored on disk, and is managed by a combination of operating system software and address translation hardware on the CPU. For a machine with a distributed

Type	What cached	Where cached	Latency (cycles)	Managed by
CPU registers	4-byte or 8-byte words	On-chip CPU registers	0	Compiler
TLB	Address translations	On-chip TLB	0	Hardware MMU
L1 cache	64-byte blocks	On-chip L1 cache	4	Hardware
L2 cache	64-byte blocks	On-chip L2 cache	10	Hardware
L3 cache	64-byte blocks	On-chip L3 cache	50	Hardware
Virtual memory	4-KB pages	Main memory	200	Hardware + OS
Buffer cache	Parts of files	Main memory	200	OS
Disk cache	Disk sectors	Disk controller	100,000	Controller firmware
Network cache	Parts of files	Local disk	10,000,000	NFS client
Browser cache	Web pages	Local disk	10,000,000	Web browser
Web cache	Web pages	Remote server disks	1,000,000,000	Web proxy server

Figure 6.23 The ubiquity of caching in modern computer systems. Acronyms: TLB: translation lookaside buffer; MMU: memory management unit; OS: operating system; NFS: network file system.