

Memory Hierarchy Design

(Appendix B and Chapter 2)

Outline

- ◆ Review of the ABCs of Caches
- ◆ Cache Performance
- ◆ Reducing Cache Miss Penalty
- ◆ Reducing Miss Rate
- ◆ Reducing Cache Miss Penalty or Miss Rate via Parallelism
- ◆ Reducing Hit Time
- ◆ Memory Technology
- ◆ Main Memory and Organization for Improving Performance
- ◆ Virtual Memory

Review of the ABCs of Caches

When the CPU finds a requested data item in the cache, it is called a cache **hit**. Otherwise, a cache **miss** occurs.

A fixed-size collection of data containing the requested word, called a **block**, is retrieved from the main memory and placed into the cache.

◆ Cache Performance Review

Recall that CPU time can be computed by

$$\text{CPU time} = \frac{\text{IC}}{\text{Programs}} \times \frac{\text{Clock Cycles}}{\text{IC}} \times \frac{\text{Seconds}}{\text{Clock Cycles}}$$

Total CPU clock cycles Clock cycle time

IC= Instruction Count

This equation is valid only when CPU is not stalled waiting for a memory access.

When we take the memory stall cycles into account, the modified CPU time is given by

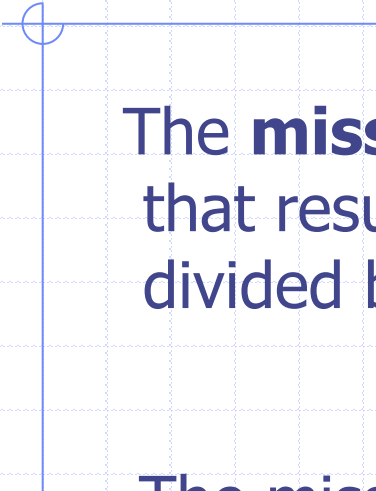
$$\text{CPU time} = (\text{CPU clock cycles} + \text{Memory stalled cycles}) \times \text{Clock cycle time}$$

The number of memory stalled cycles depends on both the number of misses and the cost per miss, which is called the miss penalty:

$$\text{Memory stalled cycles} = \text{Number of misses} \times \text{Miss Penalty}$$

$$= \text{IC} \times \frac{\text{Misses}}{\text{IC}} \times \text{Miss Penalty}$$

$$= \text{IC} \times \frac{\text{Memory accesses}}{\text{IC}} \times \text{Miss rate} \times \text{Miss Penalty}$$



The **miss rate** is defined as the fraction of cache accesses that result in a miss (i.e., number of accesses that miss divided by the number of accesses).

The miss rate is one of the most important measures of cache design.

Example:

Assume we have a computer where the CPI is 1.0 when all memory accesses hit in the cache. The only data accesses are loads and stores, and these total 50% of the instructions. If the miss penalty is 25 clock cycles and the miss rate is 2 %, how much faster would the computer be if all instructions were cache hits ?

Sol:

- First compute the performance for the computer that always hit:

$$\begin{aligned}\text{CPU time} &= (\text{CPU clock cycles} + \text{Memory stalled cycles}) \times \text{Clock cycle time} \\ &= (\text{IC} \times \text{CPI} + 0) \times \text{Clock cycle} = \text{IC} \times 1 \times \text{Clock cycle}\end{aligned}$$

Now for the computer with real cache, first we compute memory stall cycles:

$$\begin{aligned}\text{Memory stalled cycles} &= \text{IC} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss rate} \times \text{Miss Penalty} \\ &= \text{IC} \times (1 + 0.5) \times 0.02 \times 25 = \text{IC} \times 0.75\end{aligned}$$

one instruction access and 0.5 memory access per instruction


$$\text{CPU time}_{\text{cache}} = (\text{IC} \times 1.0 + \text{IC} \times 0.75) \times \text{Clock cycle} = 1.75 \times \text{IC} \times \text{Clock cycle}$$

$$\text{Speed up} = \frac{\text{CPU time}_{\text{cache}}}{\text{CPU time}} = 1.75$$

The computer with no cache misses is 1.75 times faster

The number of misses per instruction is also a common measure used in the cache design.

$$\frac{\text{Misses}}{\text{Instruction}} = \frac{\text{Misses}}{\text{Memory access}} \times \frac{\text{Memory access}}{\text{Instruction}}$$

Therefore,

$$\frac{\text{Misses}}{\text{Instruction}} = \text{Miss rate} \times \frac{\text{Memory access}}{\text{Instruction}}$$

◆ Four memory hierarchy questions

- **Where can a block be placed in a cache ?**

(a) If each block has only one place it can appear in the cache, the cache is said to be **direct mapped**. The mapping is usually

$(\text{Block address}) \text{ MOD } (\text{Number of blocks in cache})$

(b) If a block can be placed anywhere in the cache, the cache is said to be **fully associative**.

(c) If a block can be placed in the restricted set of places in the cache, the cache is set associative. A set is a group of blocks in the cache. A block is first mapped onto a set, and then the block can be placed anywhere within the set.

The set is usually chosen by

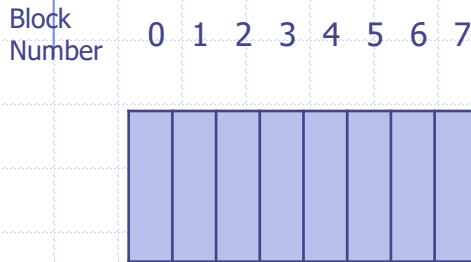
$(\text{Block address}) \text{ MOD } (\text{Number of sets in cache})$

If there are n blocks in a set, the cache is called **n-way set associative**.

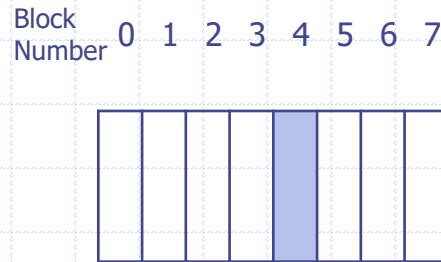
Example:

Assume there are 32 blocks in the memory. The following shows the possible location of the block 12 in each of the cache containing 8 blocks.

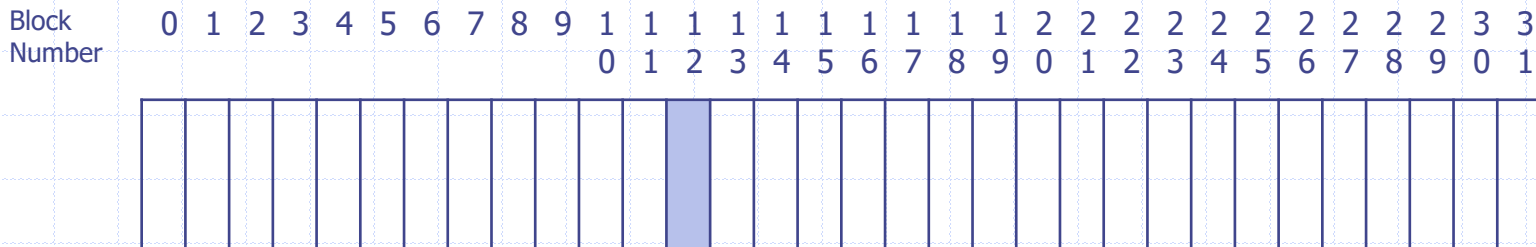
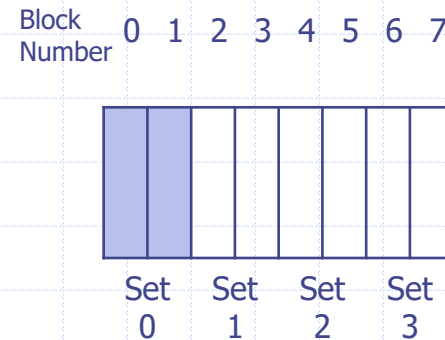
Fully Associative:
Block 12 can go
anywhere.



Direct-Mapped:
Block 12 can go only
block 4
($12 \bmod 8 = 4$)



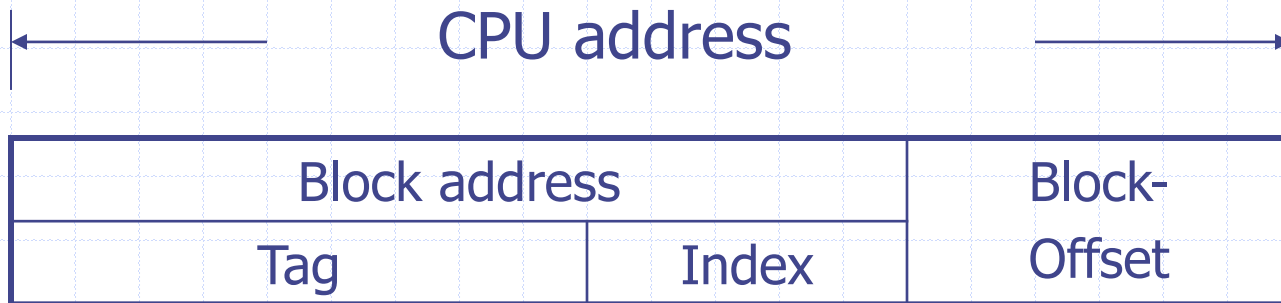
Set Associative (2-way):
Block 12 can go
anywhere in Set 0
($12 \bmod 4 = 0$)



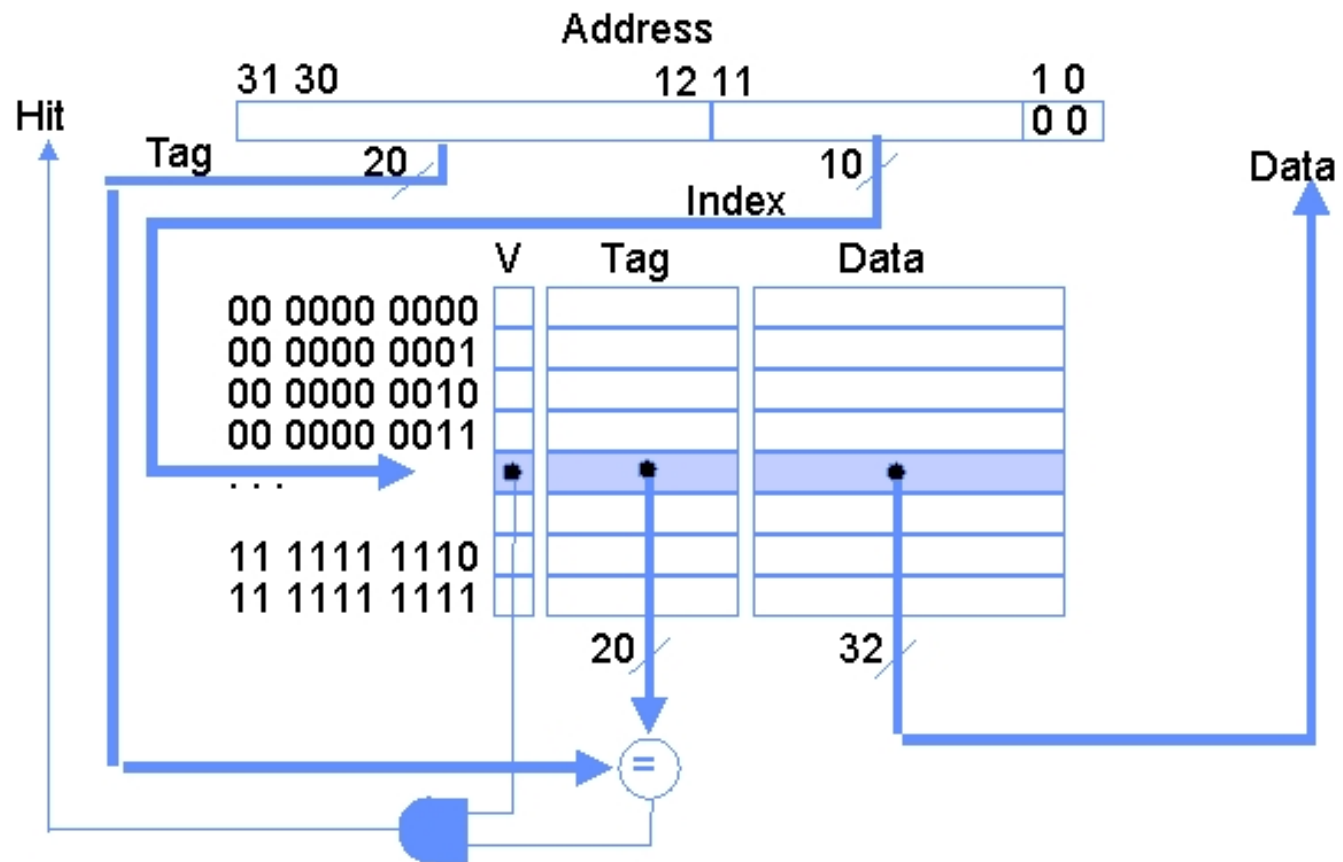
•How is a block found if it is in the cache ?

We divide a CPU address into three portions:
block-offset, index and tag.

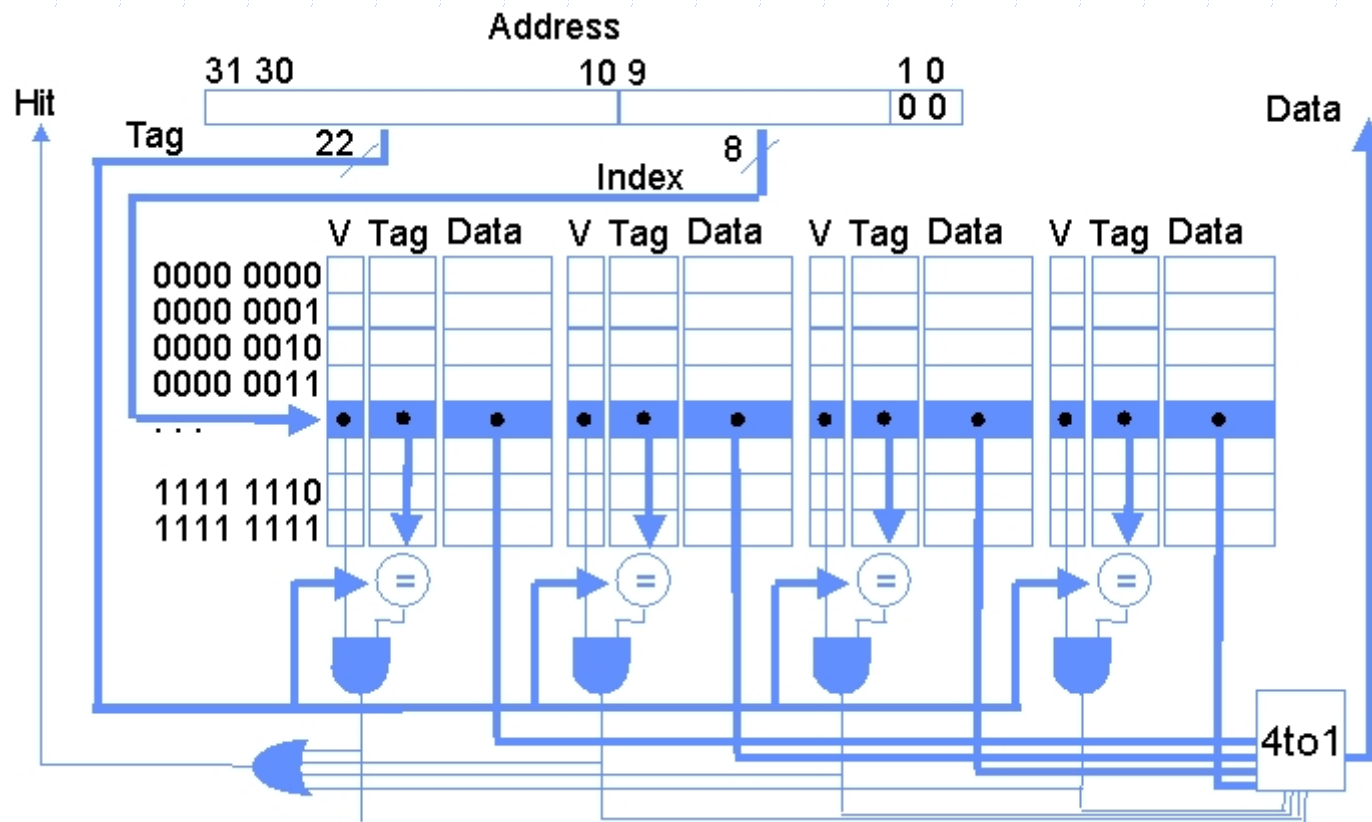
- (a) The block-offset select the desired data from a block.
- (b) The index field selects a **set** in the cache.
- (c) The tag field then compare against the **set** for a hit.



Example: Here is a direct-mapped cache with 1024 blocks. Each block contains only a word.



Example: This is a 4-way set associative cache consisting of 256 sets. Each block also contains 1 word.



•Which block should be replaced on a cache miss ?

There are 3 primary strategies employed for selecting which block to replace:

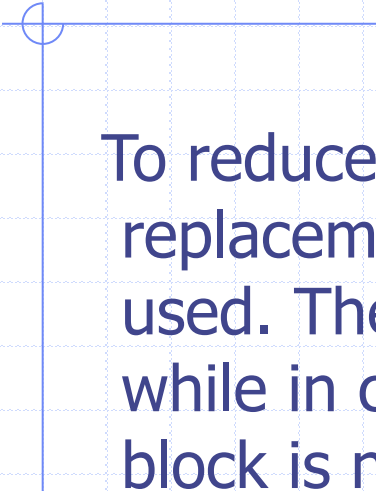
1. Random
2. Least recently used (LRU)
3. First in, first out (FIFO)

•What happens on a write ?

Reads dominate processor cache accesses. All instruction accesses are reads, and most instructions don't write. However, Amdahl's law reminds us that high-performance designs cannot neglect the speed of writes.

The write policies often distinguish cache designs. There are two basic options when writing to the cache:

1. Write through → The information is written to both the block in the cache and to the block in the lower-level memory.
2. Write back → The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.



To reduce the frequency of **writing back** blocks on replacement, a feature called the **dirty bit** is commonly used. The bit indicates whether the block is dirty (modified while in cache) or clean (not modified). If it is clean, the block is not written back when it is replaced.

When the CPU must wait for writes to complete during **write through**, the CPU is said to write stall. A common optimization to reduce write stalls is a **write buffer**, which allows the processor to continue as soon as the data are written to the buffer.



Since the data are not needed on a write, there are two options on a write miss:

1. Write allocate → The block is allocated on a write miss.
2. No-write allocate → The block is modified only in the lower-level memory.

Either write miss policy could be used with write through or write back. Normally, write-back caches use write allocate. Write-through caches often use no-write allocate.

Cache Performance

Average memory access time =
Hit time + Miss rate \times Miss penalty

Example:

Which has the lower miss rate: a 16 KB instruction cache or a 32 KB unified cache ? Assume **36%** of the instructions are data transfer instructions. Assume a hit takes 1 clock cycle and the miss penalty is 100 clock cycles. A load or store hit takes 1 extra cycle on a unified cache if there is only one cache port to satisfy two simultaneous requests. What is the average memory access time in each cache ?

Sol:

Assume the following miss rates for different caches:

$$\text{miss rate}_{16\text{KB instruction}} = 0.004$$

$$\text{miss rate}_{16\text{KB data}} = 0.114$$

$$\text{miss rate}_{32\text{KB unified}} = 0.0318$$

$$\begin{aligned} \text{Percentage of memory accesses are instruction references} \\ = 100/(100+36) = 74\% \end{aligned}$$

$$\begin{aligned} \text{Percentage of memory accesses are data references} \\ = 36/(100+36) = 26\% \end{aligned}$$

Since 74 % of the memory accesses are instruction references. Thus, the overall miss rate for the split caches is

$$(74\% \times 0.004) + (26\% \times 0.114) = 0.0324$$

Thus, a 32 KB unified cache has a slightly lower effective miss rate than two 16 KB caches.

The average memory access time formula can be divided into instruction and data accesses:

$$\text{Average memory access time} = \\ \% \text{instruction} \times (\text{Hit time} + \text{Miss rate} \times \text{Miss penalty}) + \\ \% \text{data} \times (\text{Hit time} + \text{Miss rate} \times \text{Miss penalty})$$

Therefore, the time for each organization is

$$\text{Average memory access time}_{\text{split}} = \\ \%74 \times (1 + 0.004 \times 100) + \%26 \times (1 + 0.114 \times 100) = 4.24$$

$$\text{Average memory access time}_{\text{unified}} = \\ \%74 \times (1 + 0.0318 \times 100) + \%26 \times (1 + 1 + 0.0318 \times 100) = 4.44$$

due to
structural
hazard

Hence, the split caches in this example have a better average memory access time than the unified cache.

◆ Miss penalty and out-of-order execution processors

For an out-of-order execution processor, the miss penalty should be re-defined. It is not the **full** latency of the miss to memory. Since it is not always necessary to stall a CPU when a miss occurs, we re-define the miss penalty as the **non-overlapped** latency when the processor must stall.

Therefore, in this case,

$$\begin{aligned} \text{Average memory access time} = & \\ & \text{Hit time} + \text{Miss rate} \times \\ & (\text{Total Miss Latency} - \text{Overlapped miss latency}) \end{aligned}$$

◆ Improving Cache Performance

Average memory access time =
Hit time + Miss rate \times Miss penalty

Based on the average memory access time formula,
we can organize cache optimizations into 4 categories:

1. Reducing the miss penalty
2. Reducing the miss rate
3. Reducing the miss penalty or miss rate via parallelism.
4. Reducing the time to hit in the cache.

Reducing Cache Miss Penalty

◆ First miss penalty reduction technique: multilevel caches

The multilevel cache scheme can be used to extend the cache size and match the CPU speed.

In the scheme, the 1st level cache can be small enough to match the clock cycle time of the fast CPU. The 2nd level cache can be large enough to capture many accesses that would go to main memory, thereby lessening the effective miss penalty.

Average memory access time =
 $\text{Hit time}_{L1} + \text{Miss rate}_{L1} \times \text{Miss penalty}_{L1}$

$\text{Miss Penalty}_{L1} = \text{Hit time}_{L2} + \text{Miss rate}_{L2} \times \text{Miss penalty}_{L2}$

where local miss rate

Miss rate_{Li} = the number of misses in the cache Li divided
by the number of memory accesses to the cache Li .

Therefore,

Average memory access time = $\text{Hit time}_{L1} + \text{Miss rate}_{L1} \times (\text{Hit time}_{L2} + \text{Miss rate}_{L2} \times \text{Miss penalty}_{L2})$

Example:

Given the data below, what is the impact of second-level cache associativity on its miss penalty ?

1. Hit time_{L2} for direct mapped = 10 clock cycles.
2. Two-way set associativity increases hit time by 0.1 clock cycles to 10.1 clock cycles.
3. Local miss rate_{L2} for direct mapped = 25 %.
4. Local miss rate_{L2} for two-way associative = 20 %.
5. Miss penalty_{L2} = 100 clock cycles.

Sol:

Because

$$\text{Miss Penalty}_{L1} = \text{Hit time}_{L2} + \text{Miss rate}_{L2} \times \text{Miss penalty}_{L2}$$

We have

$$\text{Miss Penalty}_{1\text{-way } L1} = 10 + 25\% \times 100 = 35.0 \text{ clock cycles}$$

$$\text{Miss Penalty}_{2\text{-way } L1} = 10.1 + 20\% \times 100 = 30.1 \text{ clock cycles}$$

In reality, second level caches are almost always synchronized with the first-level cache and CPU. Accordingly, the second level hit time must be an integral number of clock cycles. Here assume the second-level hit time becomes 11 cycles. The miss penalty then becomes

$$\text{Miss Penalty}_{2\text{-way } L1} = 11 + 20\% \times 100 = 31 \text{ clock cycles}$$

◆ Second miss penalty reduction technique: critical word first and early restart

The technique is based on the observation that the CPU normally needs just one word of the block at a time. Here are two specific strategies:

1. Critical word first → Request the missed word first from memory and send it to CPU as soon as it arrives; let the CPU continue execution while filling the rest of the words in the block.
2. Early restart → Fetch the words in normal order, but as soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution.

◆ Third miss penalty reduction technique: giving priority to read misses over writes

With a write-through cache the most important improvement is a write buffer of the proper size. Write buffers, however, do complicate memory accesses because they might hold the updated value of a location needed on a read miss.

Example:

Look at this code sequence:

```
SW R3, 512(R0) ;M[512] ← R3 (cache index 0)
LW R1, 1024(R0) ;R1 ← M[1024] (cache index 0)
LW R2, 512(R0) ;R2 ← M[512] (cache index 0)
```

Assume a direct-mapped, write-through cache that maps 512 and 1024 to the same block, and a four-word write buffer. Will the value in R2 always be equal to the value in R3 ?

Sol:

1. SW R3, 512(R0)

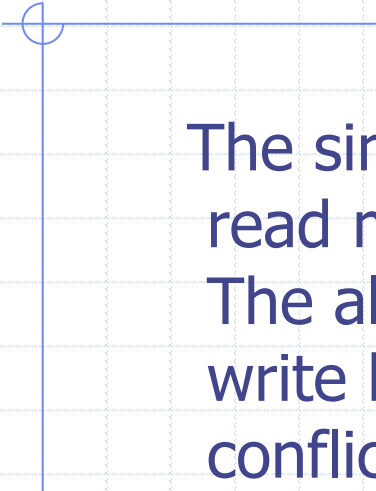
The data in R3 are placed into the write buffer after the store.

2. LW R1,1024(R0)

We have a read miss here. The content of MEM[1024] therefore is also loaded into the cache block 0.

3. LW R2,512(R0)

We again have a read miss here. Note that, if the write buffer has **not completed** writing to location 512 in memory, the read of location 512 will put the **old, wrong** value into the cache block, and then into R2. Without proper precaution, R3 would not be equal to R2 !!



The simplest way out of this dilemma is for the read miss to wait until the write buffer is empty. The alternative is to check the contents of the write buffer on a read miss, and if there are no conflicts and the memory system is available, let the read miss continue.

◆ Fourth miss penalty reduction technique: merging write buffer

This technique combines writes to sequential words into a single block to create a more efficient transfer to memory.

Write
Address

100
108
116
124

MEM[100]			
MEM[108]			
MEM[116]			
MEM[124]			



without
write-merging

Write
Address

100

MEM[100]	MEM[108]	MEM[116]	MEM[124]



with
write-merging

◆ Fifth miss penalty reduction technique: victim caches

One approach to lower the miss penalty is to remember what was discarded in case it is needed again. Since the discarded data has already been fetched, it can be used again at small cost.

Such “recycling” requires a small, fully associative cache, termed victim cache. The cache contains only blocks that are discarded from a cache because of a miss, and are checked on a miss to see if they have the desired data before going to the next lower level memory.

Reducing Miss Rate

We first start with a model that sorts all misses into 3 simple categories:

1. Compulsory → The very first access to a block cannot be in the cache, so the block must be brought into the cache.
2. Capacity → If the cache cannot contain all the blocks needed during execution of a program, capacity misses will occur because of blocks being discarded and later fetched.
3. Conflict → If the block placement strategy is set associative or direct mapped, conflict misses will occur because a block may be discarded and later retrieved if too many blocks map to its set.



Compulsory misses are those that occur in an infinite cache.

Capacity misses are those that occur in a fully associative cache.

Conflict misses are those that occur going from fully associative to 8-way associative, 4-way associative, and so on.

◆ First miss rate reduction technique: larger block size

The simplest way to reduce miss rate is to increase the block size. Larger block sizes will reduce compulsory misses. This reduction occurs because larger block size take advantage of spatial locality.

However, they reduce the number of blocks in the cache. Therefore, larger blocks may increase conflict misses and even capacity misses if the cache is small.

Example:

The following table shows the actual miss rates for 5 different-sized caches. Assume the memory system takes 80 clock cycles of overhead and then delivers 16 bytes every 2 clock cycles. Thus, it can supply 16 bytes in 82 clock cycles, 32 bytes in 84 clock cycles, and so on. Which block has the smallest average memory access time ?

Block Size	Cache Size			
	4K	16K	64K	256K
16	8.57%	3.94%	2.04%	1.09%
32	7.24%	2.87%	1.35%	0.70%
64	7.00%	2.64%	1.06%	0.51%
128	7.78%	2.77%	1.02%	0.49%
256	9.51%	3.29%	1.15%	0.49%

Sol: Recall that

Average memory access time = Hit time + Miss rate \times Miss penalty

Assume the hit time = 1 clock cycle. Then, for a 256-byte block in a 256 KB cache, we have

Average memory access time = $1 + 0.49\% \times 112 = 1.549$ clock cycles.

The following table shows the average memory access time for all block and cache sizes. The boldfaced entries show the fastest block size for a given cache size.

Block Size	Miss Penalty	Cache Size			
		4K	16K	64K	256K
16	82	8.027	4.231	2.673	1.894
32	84	7.082	3.411	2.134	1.588
64	88	7.160	3.323	1.933	1.449
128	96	8.469	3.659	1.979	1.470
256	112	11.651	4.685	2.288	1.549

◆ Second miss rate reduction technique: larger caches

The obvious way to reduce capacity misses is to increase capacity of the cache. The obvious drawback is longer hit time and higher cost.

◆ Third miss rate reduction technique: higher associativity

Miss rate can be reduced by increasing associativity.

There are two rules of thumb for increasing the associativity:

1. 8-way set associative is as effective as fully associative.
2. A direct –mapped cache of size N has about the same miss rate as a 2-way set associative cache of size $N/2$.
(2:1 cache rule of thumb)

The drawback of this method is that the hit time may be increased as the associativity becomes higher.

◆ Fourth miss rate reduction technique: way prediction and pseudoassociative caches

- way prediction

In way prediction, extra bits are kept in the cache to predict the block (i.e., way) within the set of the next cache access. A miss results in checking the other blocks for matches in subsequent clock cycles.

In addition to improving performance, way prediction can reduce power for embedded applications. By only supplying power to the half of tags that are expected to be used, the MIPS R4300 series lowers power consumption with the same benefits.

- Pseudoassociative

In this technique, accesses proceed just as in the direct-mapped cache for a hit. On a miss, however, before going to the next lower level of the memory hierarchy, a second entry is checked to see if it matches there. A simple way is to invert the most significant bit of the index field to find another block in the “pseudoset.”

In fact, pseudoassociative have one fast and one slow hit time — corresponding to a regular hit and a pseudohit — in addition to the miss penalty.



◆ Fifth miss rate reduction technique: compiler optimizations

This technique reduces miss rates without any hardware changes. A simple example is the loop exchange.

Some programs have nested loops that access data in memory in nonsequential order. Simply exchanging the nesting of the loops can make the code access the data in the order they are stored.

```
for (j=0;j<100;j=j+1)
  for(i=0;i<5000;i=i+1)
    x[i][j]=2*x[i][j]
```



```
for (i=0;i<5000;i=i+1)
  for(j=0;j<100;j=j+1)
    x[i][j]=2*x[i][j]
```


Reducing Cache Miss Penalty or Miss rate via Parallelism

Here we describe three techniques that **overlap** the **execution** of instructions with **activity** in memory hierarchy.

◆ First miss penalty/rate reduction technique: nonblocking caches to reduce stalls on cache misses

For pipelined computers that allow out-of-order completion, the CPU need not stall on a cache miss.

A **nonblocking cache** escalates the potential benefits of such a scheme by allowing the data cache to continue to supply cache hits during a miss.

A subtle and complex option is that the cache may further lower the effective miss penalty if it can overlap multiple misses: a “hit under multiple miss” optimization.

◆ Second miss penalty/rate reduction technique: hardware prefetching of instructions and data

Instruction prefetching is frequently done in hardware outside of cache. Typically, the processor fetches **two** blocks on a miss: the requested block and the next consecutive block. The requested block is placed in the instruction cache when it returns, and the prefetched block is placed into the **instruction stream buffer**.

If the requested block is present in the instruction stream buffer, the original cache request is canceled, the block is read from the stream buffer, and the next prefetch request is issued.

Example:

What is the effective miss rate of UltraSPARC III using prefetching ? How much bigger a data cache would be needed in the UltraSPARC III to match the average access time if prefetching were removed ? It has a 64 KB data cache. Assume prefetching reduces the data miss rate by 20%. The following table gives the number of misses per 1000 instructions for 64KB, 128KB and 256KB data caches. Also assume the number of data references out of 1000 instructions is 220. In addition, hit time = 1 clock and miss penalty=15 clocks.

Data cache size	64KB	128KB	256KB
Miss per 1000 instructions	36.9	35.3	32.6

Sol:

Based on the table, we see that the number of misses per 1000 instructions for a 64KB data cache is 36.9. Since the number of data accesses per 1000 instructions is 220, from

$$\frac{\text{Misses}}{\text{Instruction}} = \text{Miss rate} \times \frac{\text{Memory access}}{\text{Instruction}}$$

we know that the miss rate for a 64 KB data cache is given by

$$\text{Miss rate} = \frac{36.9}{1000 \times 220 / 1000} = 16.7\%$$

We assume it takes 1 extra clock cycle if the data miss the cache but are found in the prefetch buffer. Here is the revised formula for computing the average memory access time.

$$\begin{aligned} \text{Average memory access time}_{\text{prefetch}} = & \text{Hit time} + \\ & \text{Miss rate} \times \text{Prefetch hit rate} \times \text{1} + \\ & \text{Miss rate} \times (1 - \text{Prefetch hit rate}) \times \text{Miss penalty} \end{aligned}$$

Extra clock
cycle for
prefetch buffer

Therefore,

$$\begin{aligned} \text{Average memory access time}_{\text{prefetch}} = & 1 + 16.7\% \times 20\% \times 1 + \\ & 16.7\% \times (1 - 20\%) \times 15 = 3.046 \end{aligned}$$

To find the effective miss rate when the average memory access time is 3.046, we first note that

$$\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

Therefore,

$$\text{Effective Miss rate} = (3.046 - \text{Hit time}) / \text{Miss penalty} = (3.046 - 1) / 15 = 13.6\%$$

The table gives the misses per 1000 instructions of a 256 KB data cache as 32.6, yielding a miss rate of $32.6 / (22\% \times 1000) = 14.8\%$

If the prefetching reduces the miss rate by 20%, then a 64KB data cache with prefetching outperforms a 256 KB cache without it.

◆ Third miss penalty/rate reduction technique: compiler-controlled prefetching

An alternative to hardware prefetching is for the compiler to insert prefetch instructions to request the data before they are needed. There are two flavors of prefetch:

1. Register prefetch will load the value into a register
2. Cache prefetch loads data only into the cache and not register.

Reducing Hit Time

Hit time is critical because it affects the clock rate of the processor.

A fast hit time is multiplied in importance beyond the average memory access time formula because it helps everything.

◆ First hit time reduction technique: small and simple caches

Small cache →

Smaller hardware is faster, and a small cache certainly helps the hit time.

Simple cache →

The direct-mapped cache can reduce the hit time because the tag check and the data transmission can be overlapped in the scheme.

◆ Second hit time reduction technique: avoiding address translation during indexing of the cache

Even a small and simple cache must cope with the translation of a virtual address from the CPU to a physical address to access memory.

The hit time therefore can be reduced if we use the virtual address directly for indexing cache and tag comparison. Such caches are termed virtual caches.

Using the virtual address directly is not popular because of the following reasons:

1. Protection: Page-level protection is checked as the part of virtual to physical address translation, and it must be enforced no matter what.
2. Process-dependent: Every time a process is switched, the virtual addresses refer to different physical addresses, requiring the virtual cache to be flushed.
3. Aliasing: Operating systems and user programs may use two different virtual addresses for the same physical address. These duplicate addresses may result in two copies of the same data in a virtual cache; if one is modified, the other will have the wrong value.

◆ Third hit time reduction technique: pipelined cache access

This technique is simply to pipeline cache access so that the effective latency of a first-level cache hit can be multiple clock cycles, giving fast cycle time and slow hit.

This split increases the number of pipeline stages, leading to greater penalty on mispredicted branches and more clock cycles between the issue of the load and the use of the data.

Memory Technology

There are two basic memory technologies considered here: static RAMs (SRAMs) and dynamic RAMs (DRAMs).

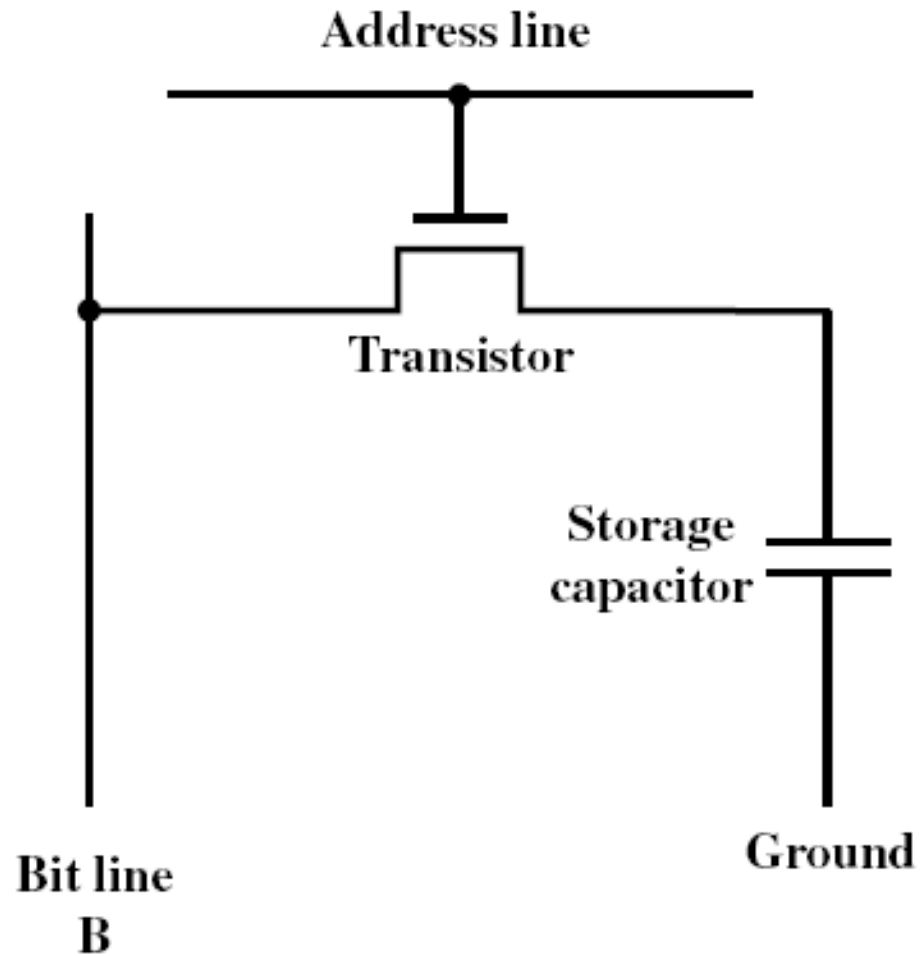
Virtually all desktop or server computers since 1975 used DRAMs for main memory, and virtually all used SRAMs for cache.



In DRAM designs the emphasis is on cost per bit and capacity.

SRAM designs are concerned with speed and capacity.

◆ DRAM Storage Cell



◆ DRAM Operation

1. Address line active when bit read or written
2. Logic '1' closes transistor switch (i.e., current flows)
3. Write
 - 1) Voltage to bit line – High for 1 low for 0
 - 2) Signal address line – Controls transfer of charge to capacitor
4. Read
 - 1) Address line selected – transistor turns on
 - 2) Charge from capacitor fed via bit line to sense amplifier
 - 3) Compares with reference value to determine 0 or 1

◆ DRAM Refreshing

○ Two things discharge a DRAM capacitor:

1. Data read
2. Leakage current

We therefore need refreshing (or **pre-charging**) capacitor even when powered and idle (once every few milliseconds).

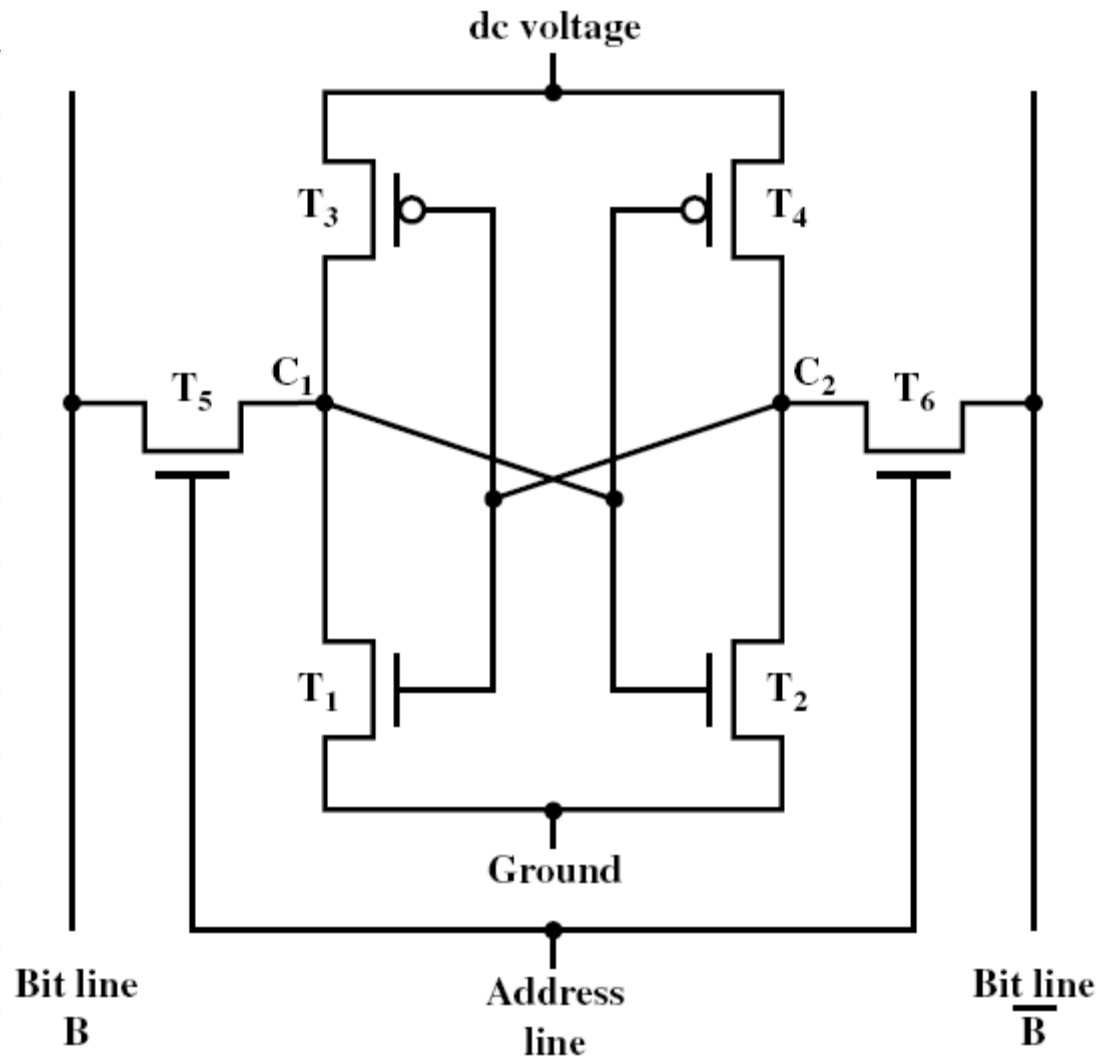
Refresh circuit included on chip:

- Even with added cost, still cheaper than SRAM cost.

Refresh process involves disabling chip, then reading data and writing it back.

The refreshing process will slow down the performance of DRAM.

◆ SRAM Storage Cell



◆ SRAM Operation

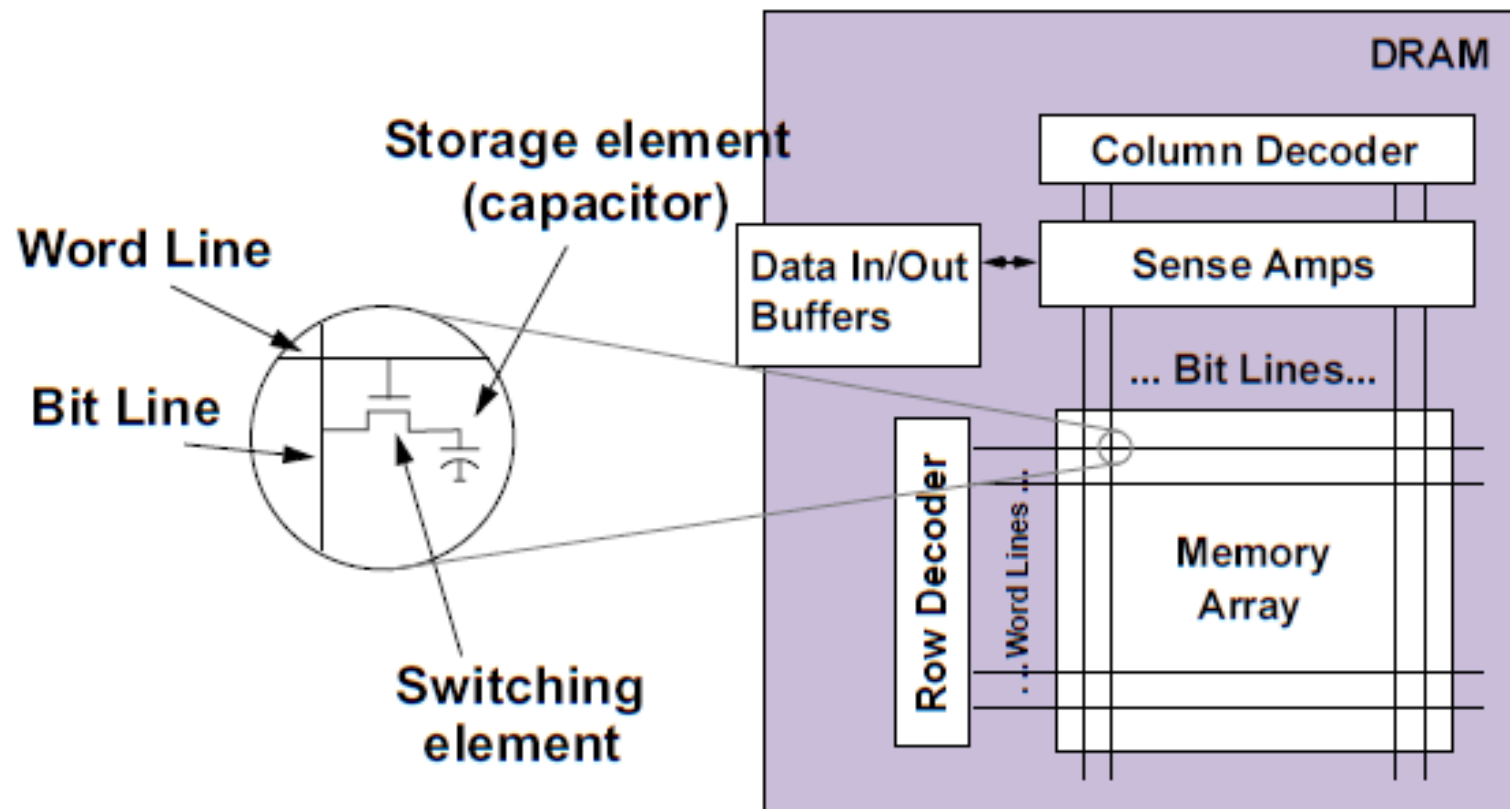
1. Transistor arrangement gives stable logic state
2. State 1
 - 1) C1 high, C2 low
 - 2) T1 T4 off, T2 T3 on
3. State 0
 - 1) C2 high, C1 low
 - 2) T2 & T3 off, T1 & T4 on
4. Address line transistors
 - 1) T5 & T6 act as switches connecting cell
5. Write – apply value to B & compliment to B
6. Read – value is on line B
7. It is not necessary to refresh the storage cell in SRAM.

◆ DRAM Internal Organization

As early DRAM grew in capacity, the cost of a package with all the necessary lines was an issue.

The solution was to multiplex the address lines, thereby cutting the address pins in half.

One-half of the address is sent first, called the row access strobe (RAS). The other half of the address, sent during the column access strobe (CAS), follows it.

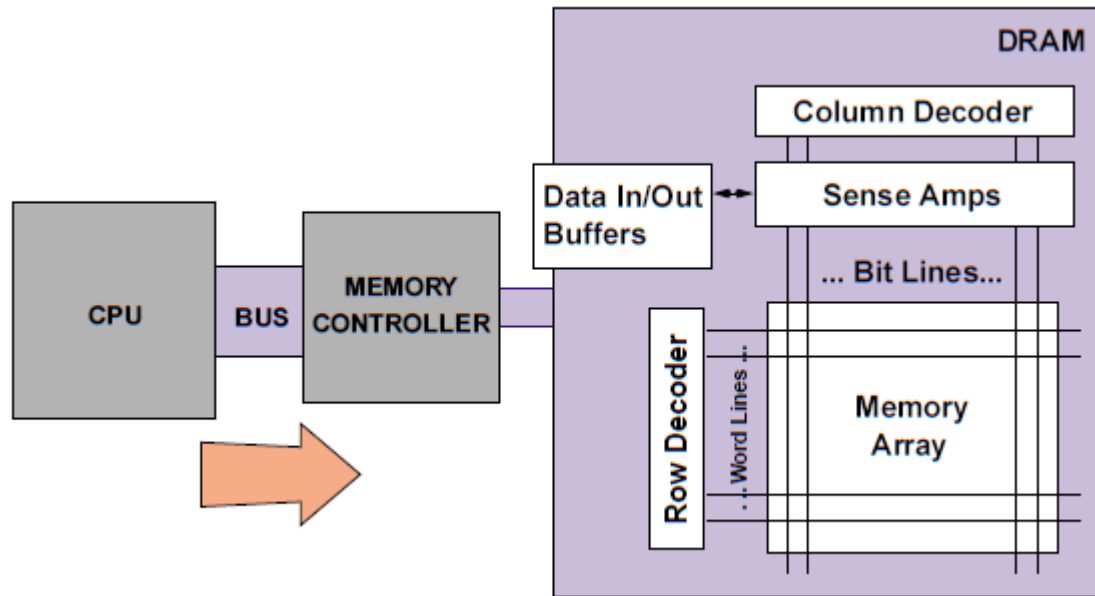


◆ Accessing DRAM

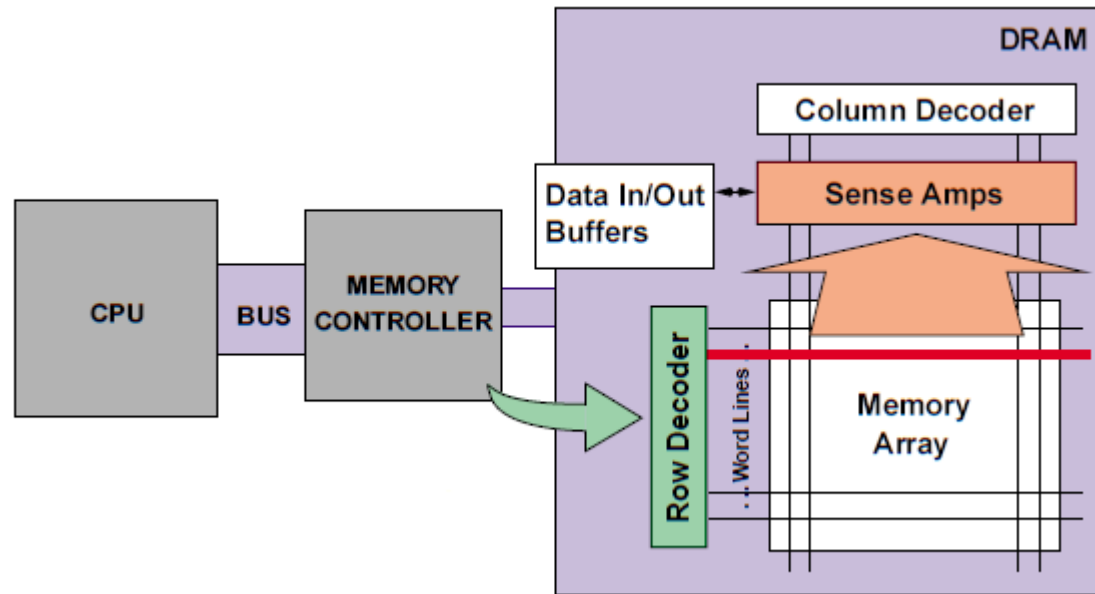
To accessing DRAM, a memory controller is required.

Use a read operation as an example. The memory controller first accepts an address from CPU. It then translates the address to RAS and CAS for accessing DRAM. After that, the memory controller obtains data From DRAM, and delivers data back to CPU.

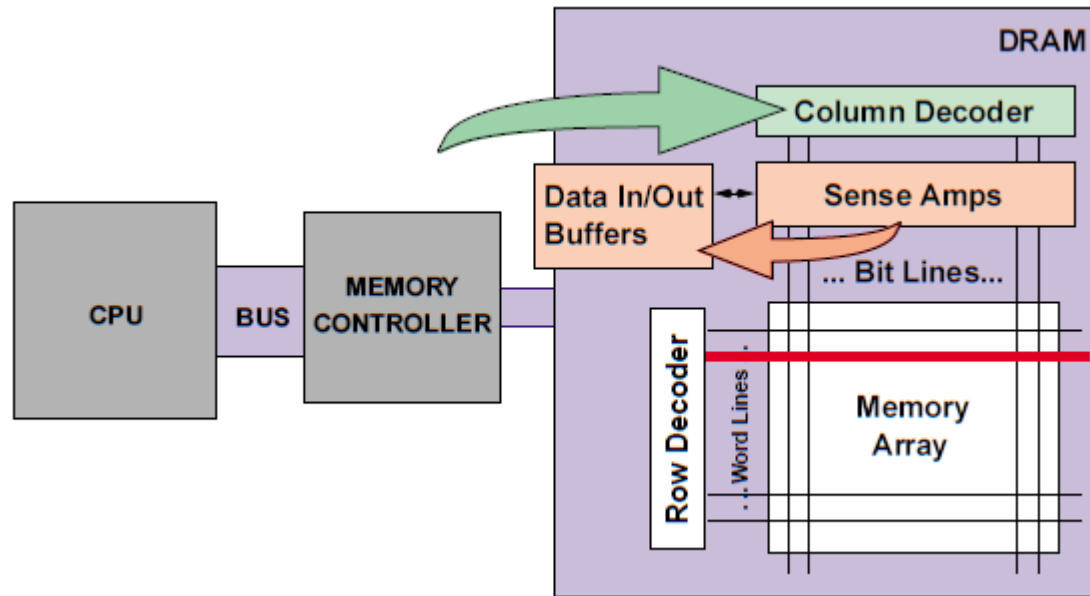
Step-by-step illustration of this example is shown below.



CPU sends an address to memory controller.

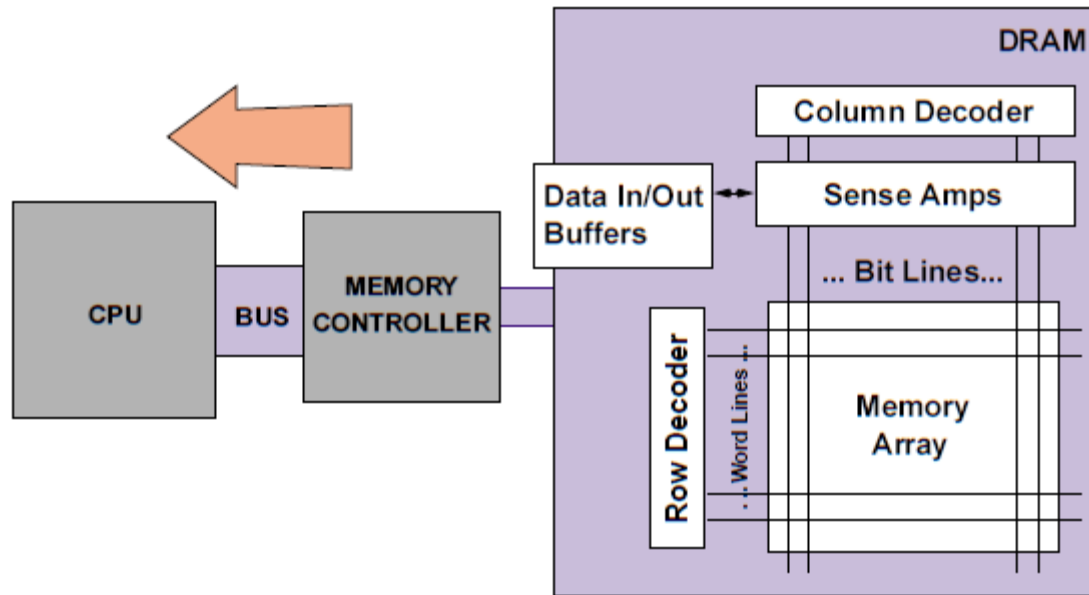


Memory controller sends a RAS to DRAM.



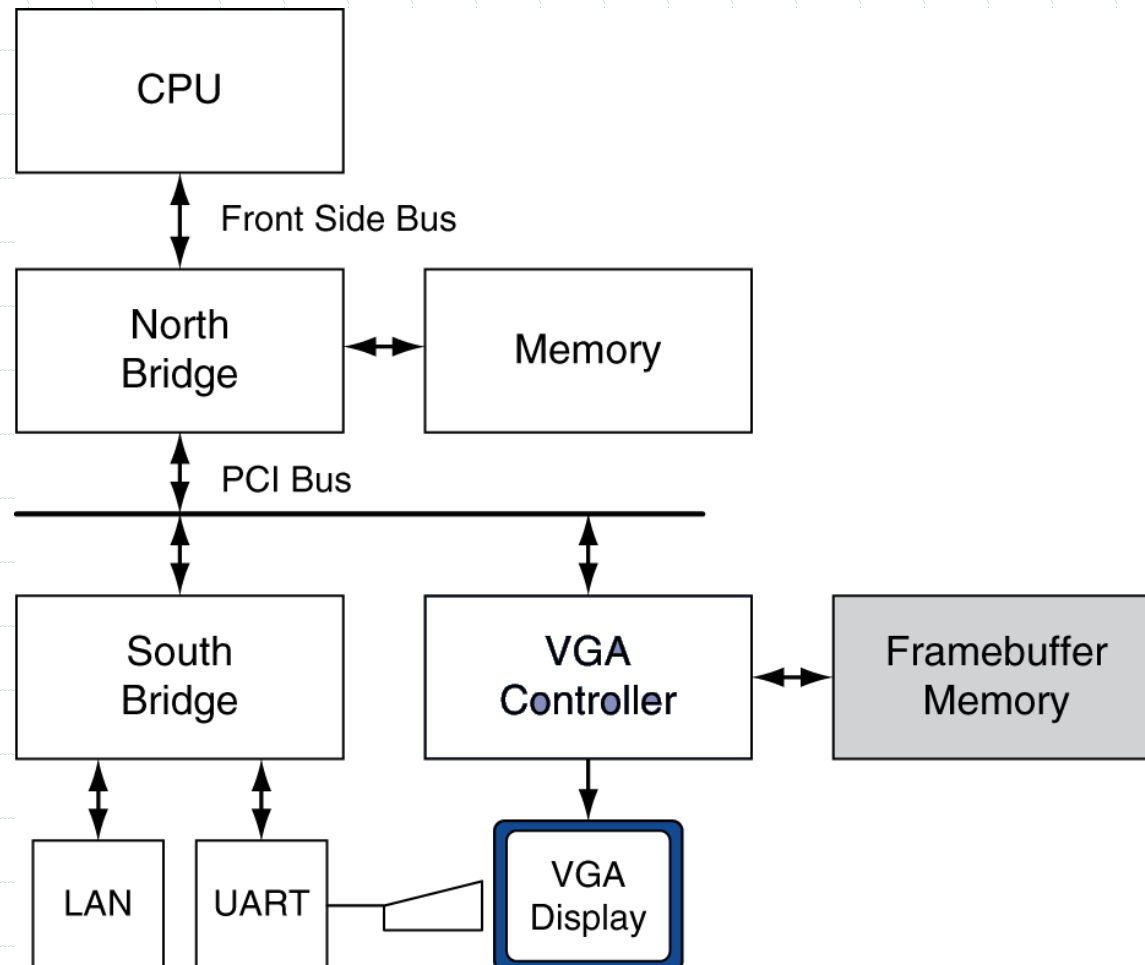
Memory controller sends a CAS to DRAM.



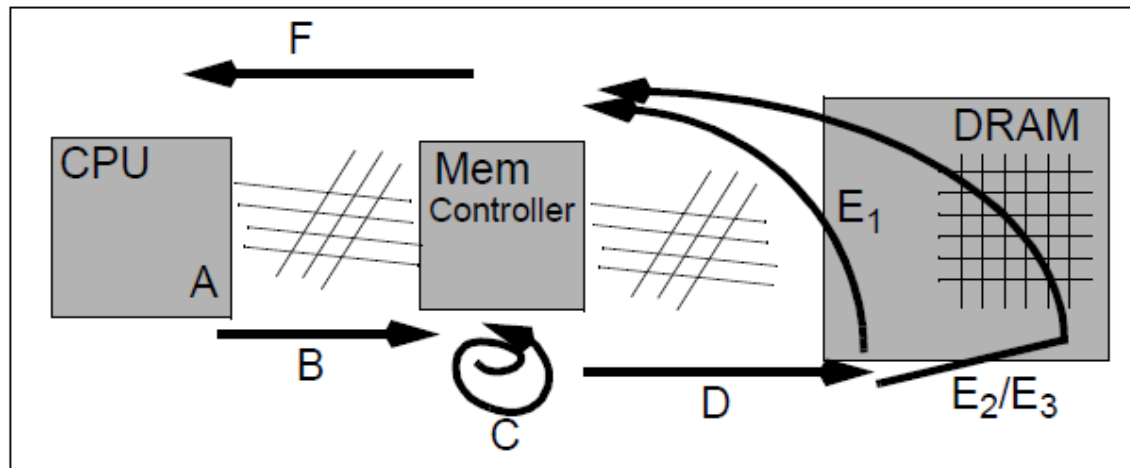


Memory controller sends the data to CPU.

The memory controller is usually included in the northbridge chip in the PC mother board.



◆ DRAM Latency



A: Transaction request may be delayed in Queue
B: Transaction request sent to Memory Controller
C: Transaction converted to Command Sequences
(may be queued)

D: Command/s Sent to DRAM

E₁: Requires only a **CAS** or

E₂: Requires **RAS + CAS** or

E₃: Requires **PRE + RAS + CAS**

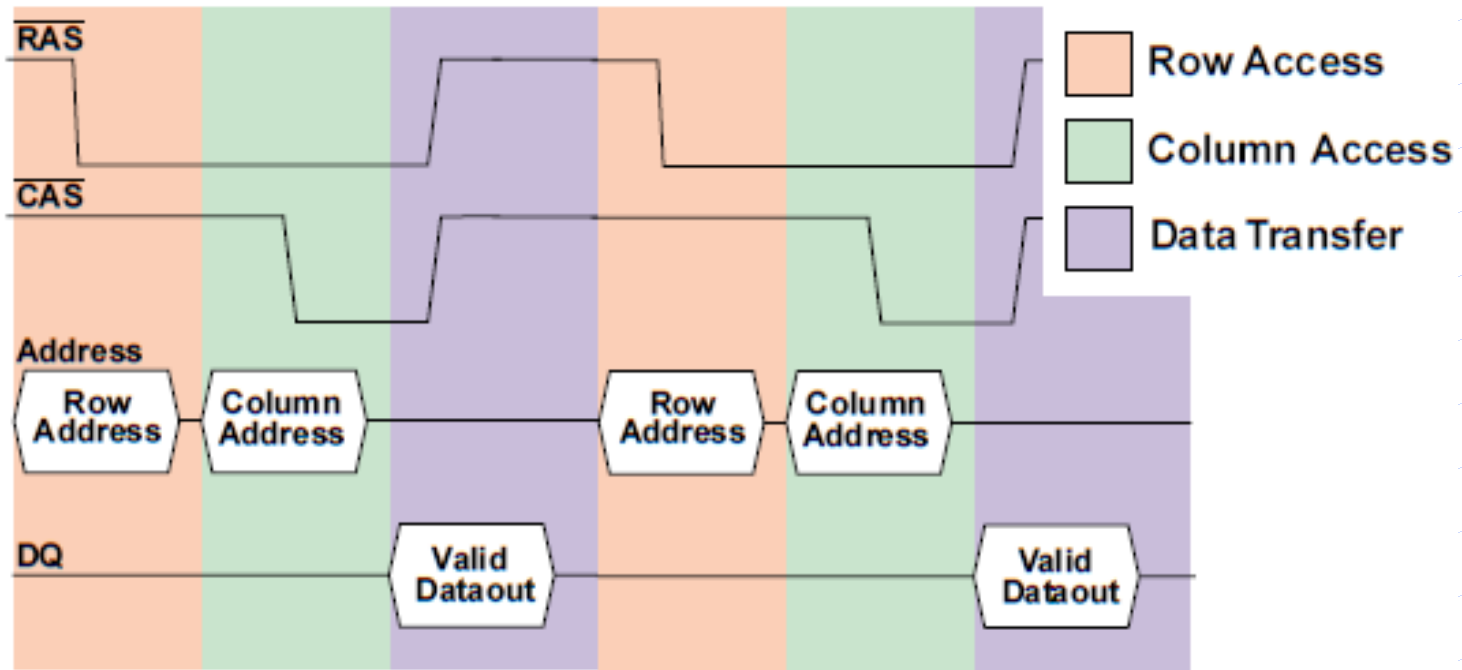
F: Transaction sent back to CPU

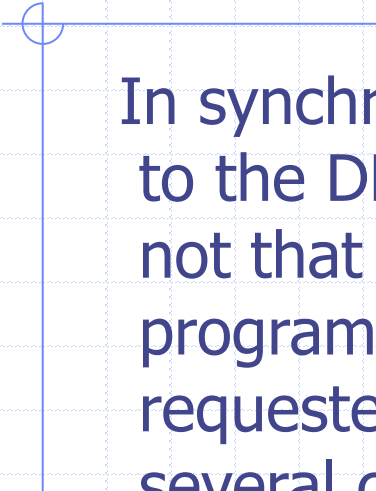
“DRAM Latency” = A + B + C + D + E + F

◆ DRAM Improvement

Basic DRAM has an asynchronous interface to the memory controller, and hence every transfer involved overhead to synchronized with the memory controller.

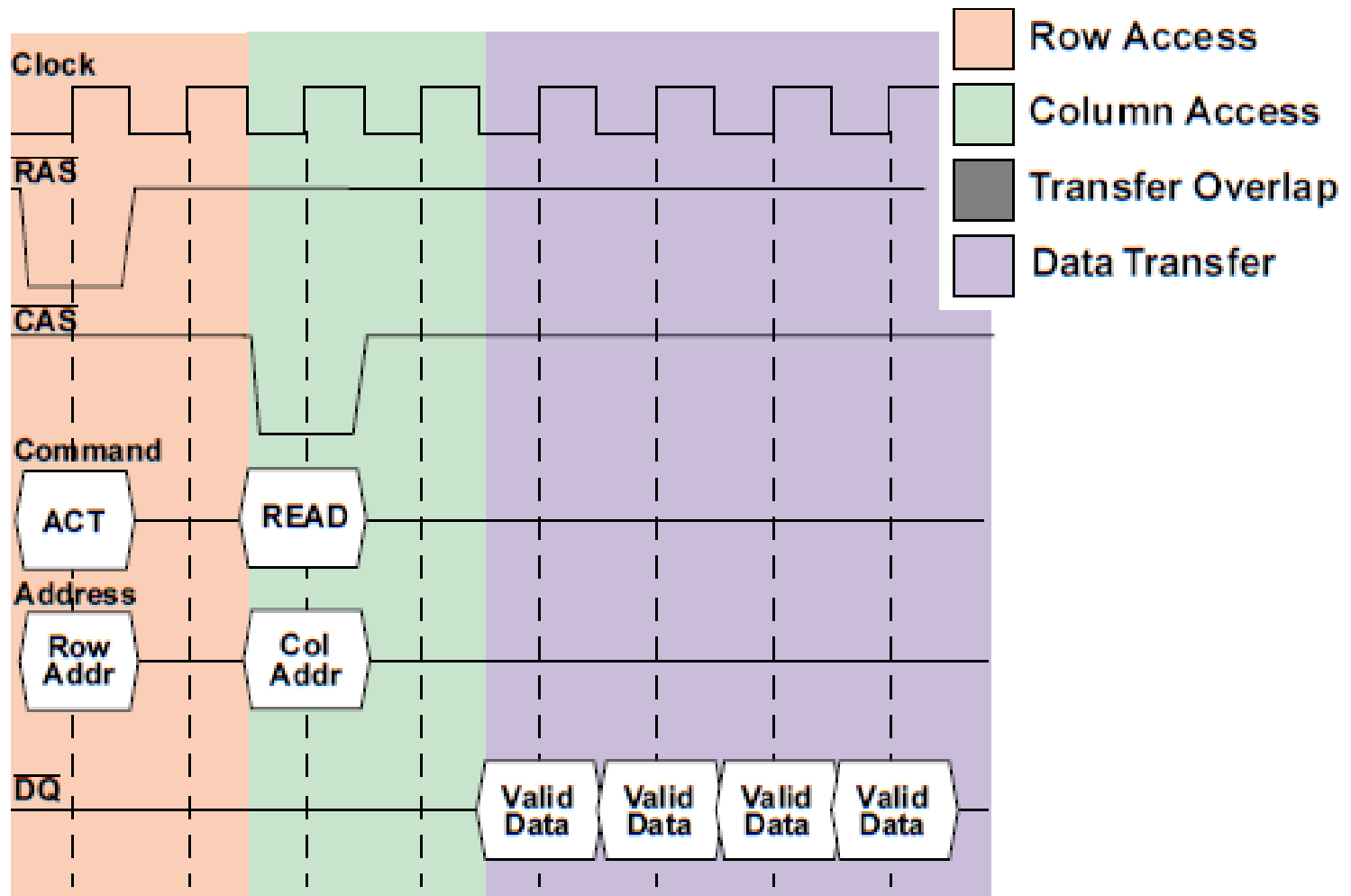
Read timing for basic DRAM.

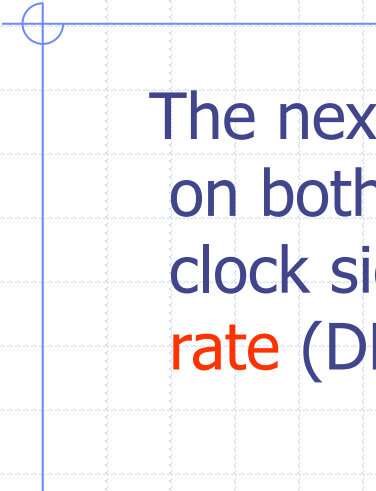




In synchronous DRAM (SDRAM), a clock signal is added to the DRAM interface so that repeated transfers would not that overhead. SDRAMs typically have a programmable register to hold the number of bytes requested, and hence can send many bytes over several cycles per request.

Read timing for synchronous DRAM (SDRAM).

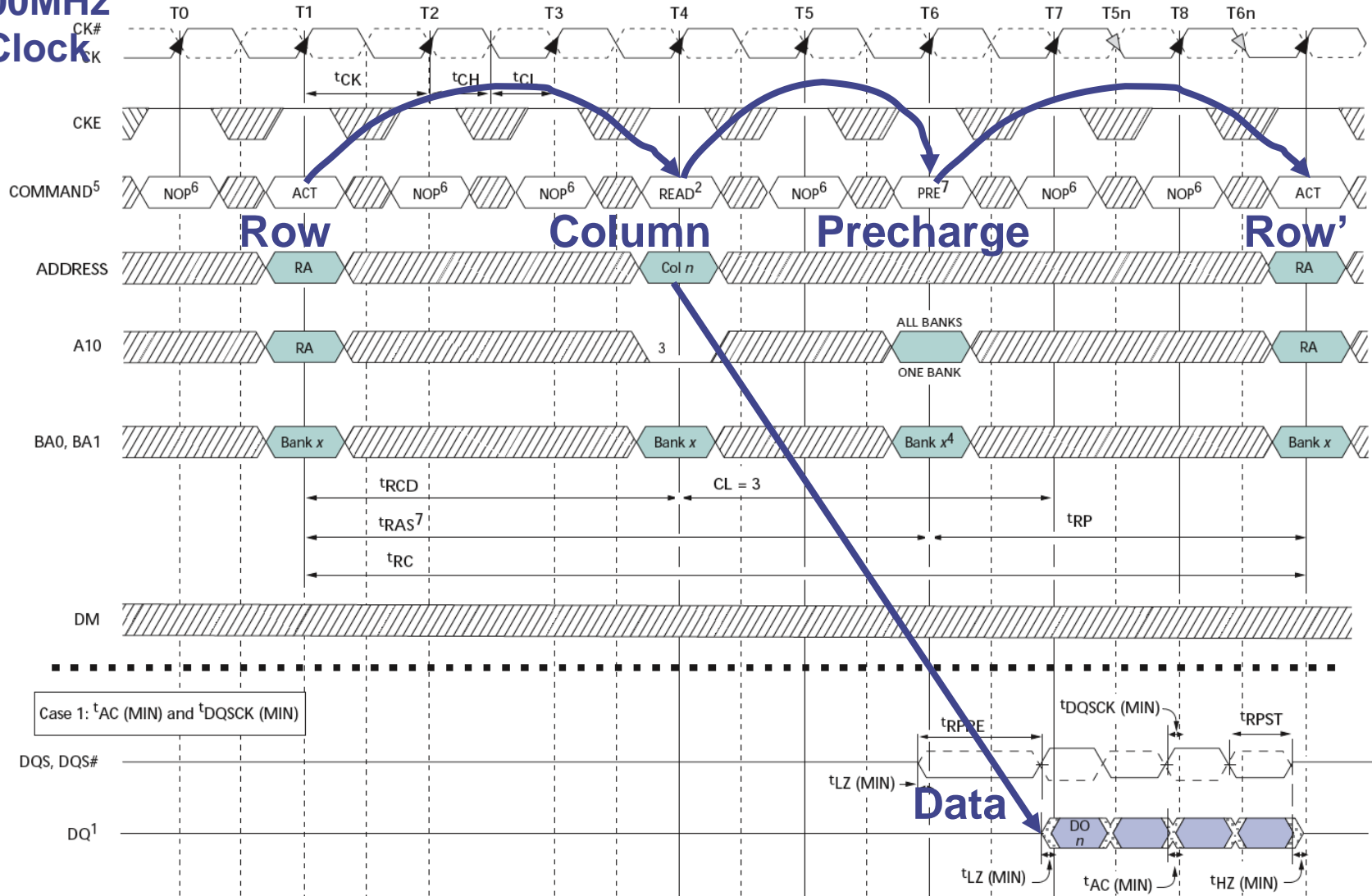




The next major DRAM innovation is to transfer data on both rising edge and falling edge of the DRAM clock signal. This optimization is called **double data rate** (DDR).

Double-Data Rate (DDR2) DRAM

200MHz
Clock



400Mb/s
Data Rate

[Micron, 256Mb DDR2 SDRAM datasheet]

Main Memory and Organizations for Improving Performance

Performance measures of main memory emphasize both latency and bandwidth.

The memory bandwidth is the number of byte read or written per unit time.

Although caches are interested in low-latency memory, it is generally easier to improve memory bandwidth with new organizations than it is to reduce latency. With popularity of L2 caches and their larger block sizes, main memory bandwidth becomes important to caches as well.

In this section we examine techniques for organizing memory to improve bandwidth.

Assume the performance of basic memory organization is

1. 4 clock cycles to send the address
2. 56 clock cycles for the access time per word
3. 4 clock cycles to send a word of data

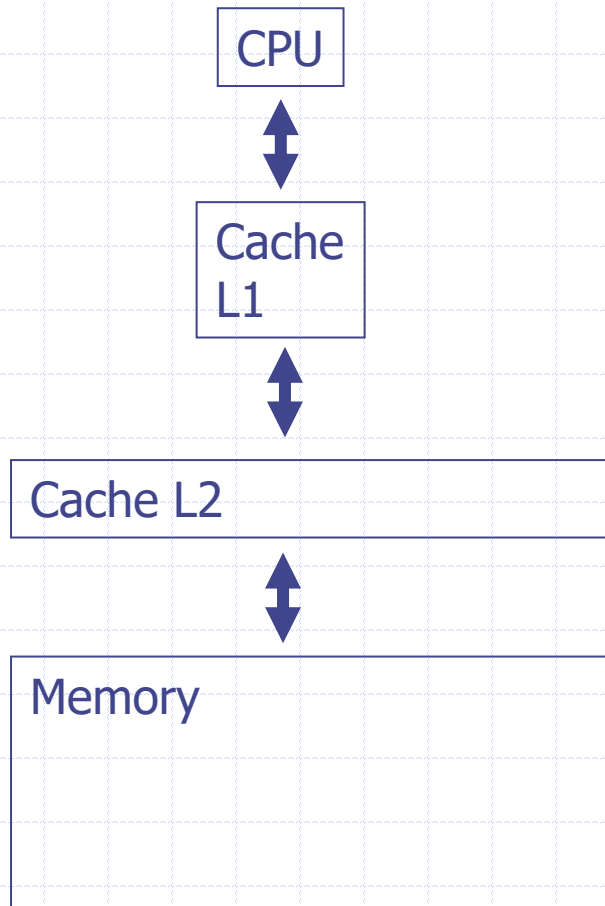
Given a cache block of 4 words, and that a word is 8 bytes, the miss penalty is $4 \times (4 + 56 + 4) = 256$ cycles, with a memory bandwidth of $(8 \times 4) / 256 = 1/8$ byte per clock cycle.

Three different memory systems.

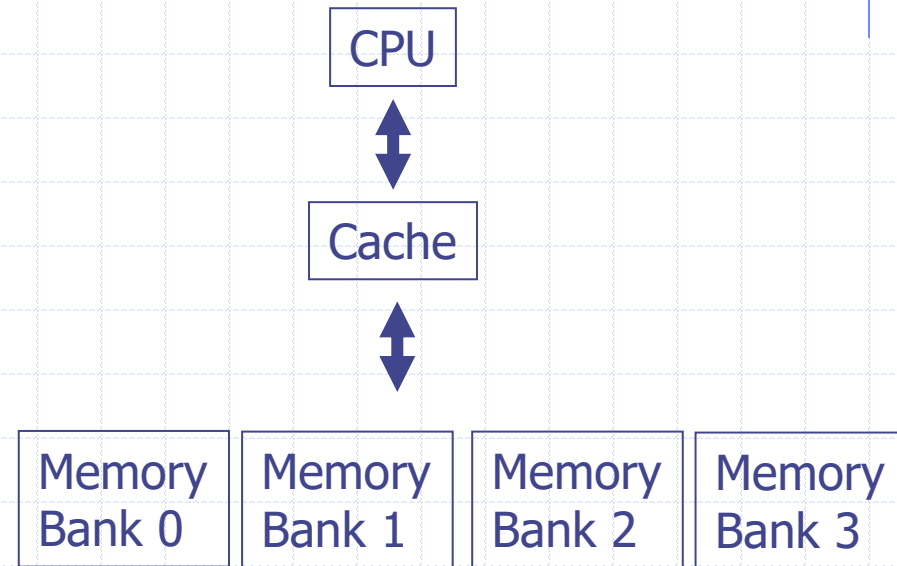
One-word-wide
memory organization



Wide memory
organization



Interleaved memory
organization



◆ First technique for higher bandwidth: wider main memory

First-level caches are often organized with a physical width of 1 word because most CPU accesses are that size. Doubling or quadrupling the width of the cache and memory will therefore doubling or quadrupling the memory bandwidth.

With a main memory width of 2 words, the miss penalty in our example will drop from $4 \times (4 + 56 + 4) = 256$ cycles to $2 \times (4 + 56 + 4) = 128$ cycles. At 4 word wide the miss penalty is just $1 \times (4 + 56 + 4) = 64$ cycles.

Since main memory is traditionally expandable, a drawback to wide memory is that the minimum increment is doubled or quadrupled when the width is doubled or quadrupled.

◆ Second technique for higher bandwidth: simple interleaved memory

Memory chips can be organized in **banks** to read or write multiple words at a time rather than a single word.

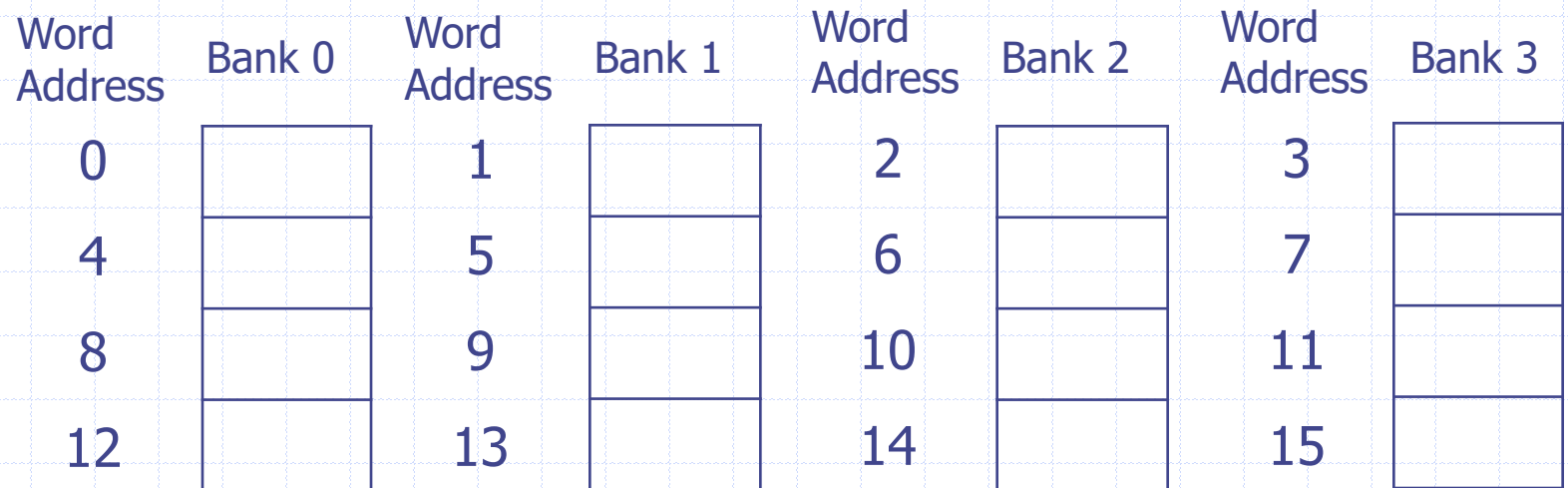
The banks are often 1 word wide so that the width of the bus and the cache need not change, but sending addresses to several banks permits them all to read simultaneously.

For example, sending an address to 4 banks yields a miss penalty of $4 + 56 + (4 \times 4) = 76$ clock cycles.

Banks are also valuable to writes. Banks allow 1 clock cycle for each write, provided the writes are not destined to the same bank. Such a memory organization is especially important for write through.

We assume the addresses of the four banks are interleaved at the word level.

Bank i has all words whose address modulo 4 is i , $i=0,1,2,3$.



4-way interleaved memory

Example:

Consider the following description of a computer and its cache performance :

Block size=1 word

memory bus width = 1 word

miss rate = 3%

memory access per instruction = 1.2

cache miss penalty = 64 cycles

average CPI (ignoring cache misses)=2

If we change the block size to 2 words, the miss rate falls to 2%, and a 4-word block has a miss rate of 1.2%.

What is the improvement in performance of interleaving 2 ways and 4 ways versus doubling the width of memory and bus ?

Sol: **memory access per instruction**

The CPI for the base computer using 1-word block is

$$2 + (1.2 \times 3\% \times 64) = 4.30$$

miss rate (points to 3%)
miss penalty (points to 64)
miss penalty for two words (points to 64)

Increasing the block size to 2 words gives the following options:

1 word = 8 bytes = 64 bits

64-bit bus and memory, no interleaving = $2 + (1.2 \times 2\% \times 2 \times 64) = 5.07$
64-bit bus and memory, interleaving = $2 + (1.2 \times 2\% \times (4 + 56 + 8)) = 3.63$
128-bit bus and memory, no interleaving = $2 + (1.2 \times 2\% \times 1 \times 64) = 3.54$

If we increase the block size to four, the following is obtained:

64-bit bus and memory, no interleaving = $2 + (1.2 \times 1.2\% \times 4 \times 64) = 5.69$
64-bit bus and memory, interleaving = $2 + (1.2 \times 1.2\% \times (4 + 56 + 16)) = 3.09$
128-bit bus and memory, no interleaving = $2 + (1.2 \times 1.2\% \times 2 \times 64) = 3.84$

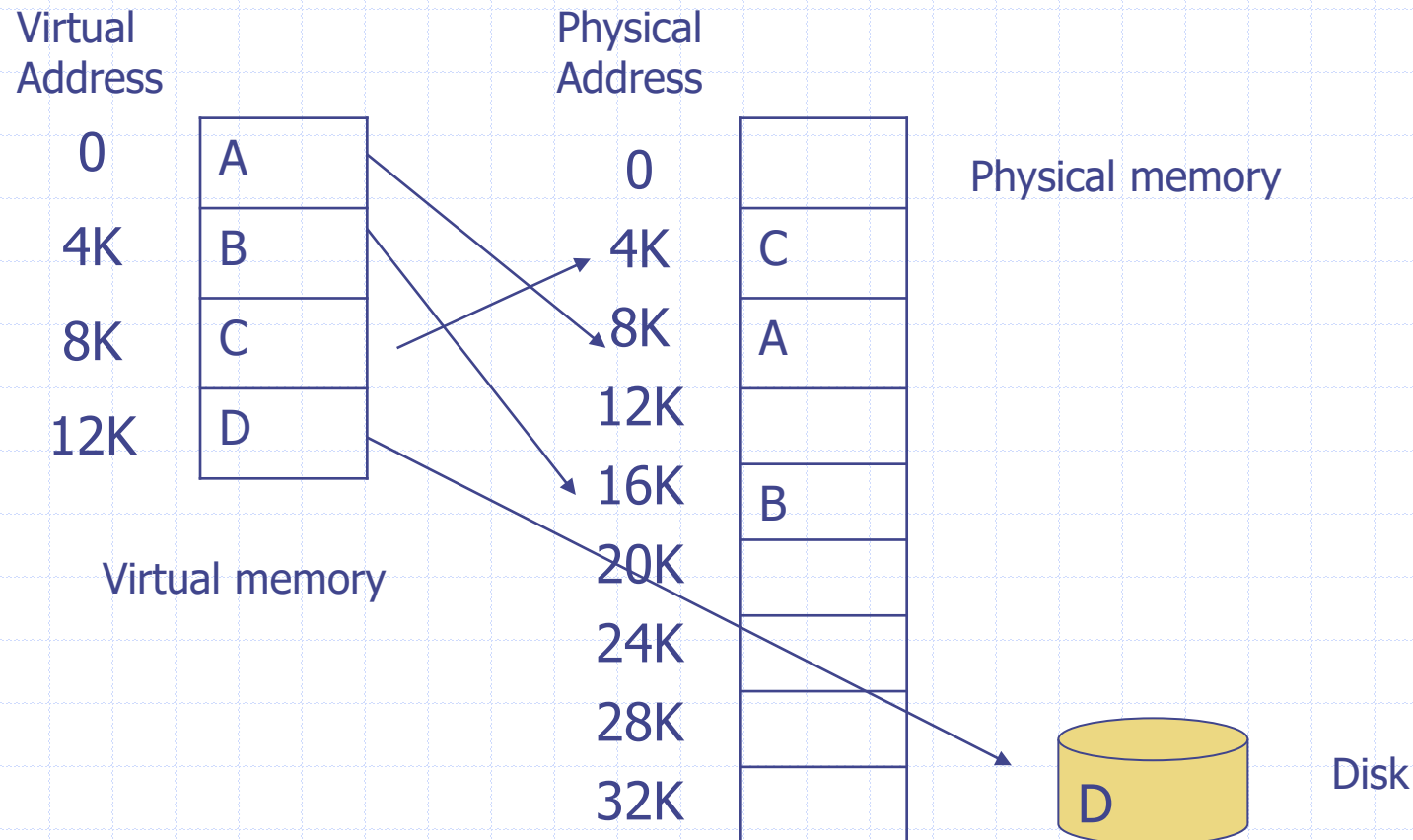
Virtual memory

The main memory can act as a “cache” for the secondary storage, usually implemented with magnetic disks. This technique is called virtual memory. There are two motivations for virtual memory: to allow efficient and safe sharing of memory among multiple programs, and to remove the programming burdens of a small, limited amount of main memory.

Several general memory ideas about caches are analogous to virtual memory, although many of the terms are different.

Page or **segment** is used for **block**, and **page fault** or **address fault** is used for **miss**.

With virtual memory, the CPU produces virtual addresses that are translated by a combination of software and hardware to physical addresses, which access the memory. This process is called **memory-mapping** or **address translation**.



◆ Four memory hierarchy questions revisited

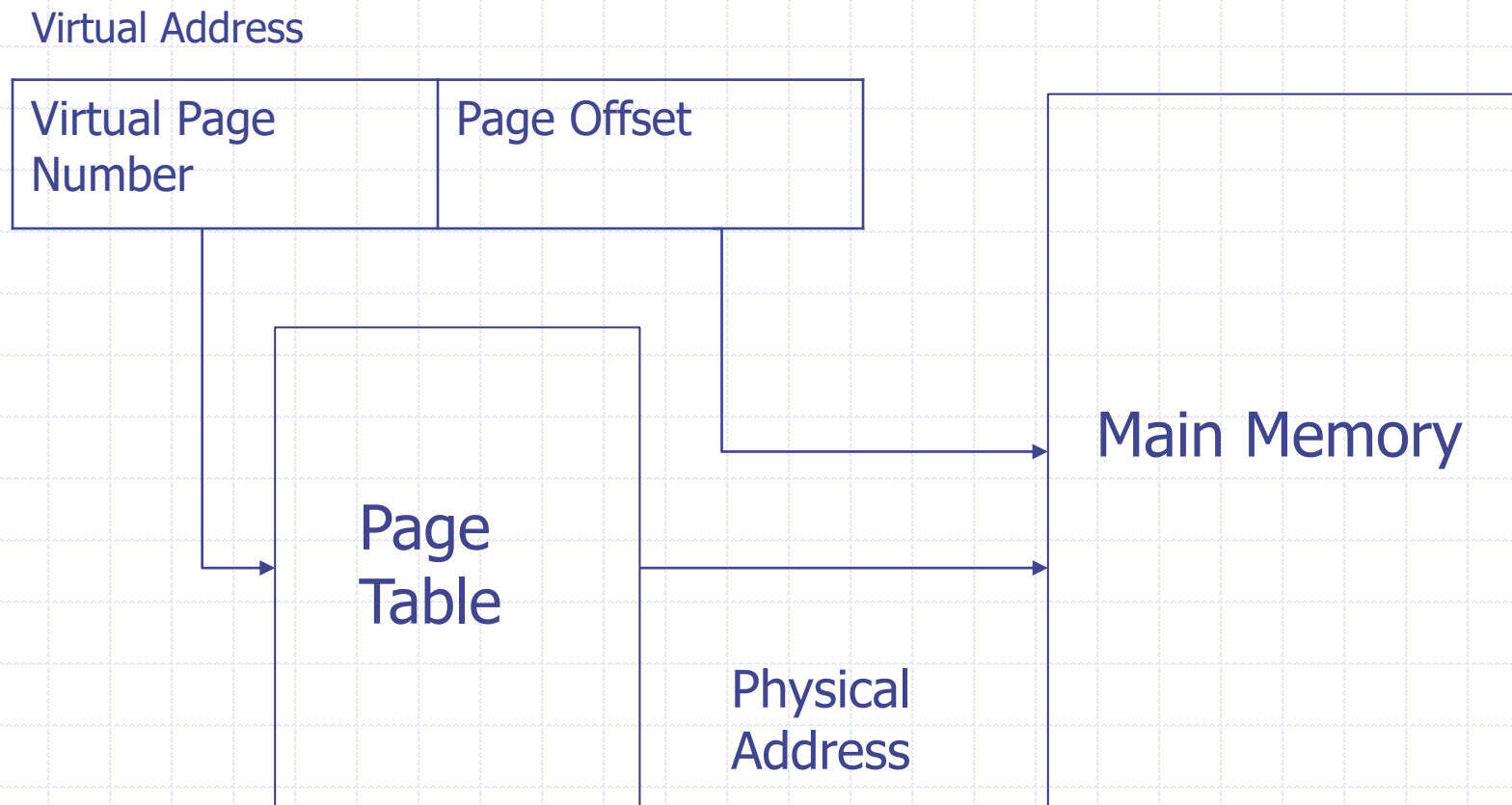
- **Where can a block be placed in a main memory ?**

The miss penalty for virtual memory is very high.

To reduce the miss rate, the operating systems allow blocks to be placed anywhere in main memory.

•How is a block found if it is in the main memory ?

We need to translate the virtual address to physical address. This can be done using the page table.



•Which block should be replaced on a virtual memory miss ?

The LRU is usually used to minimize page faults. To help the operating system estimate LRU, many processors provide a **use bit** or **reference bit**, which is logically set whenever a page is accessed.

•What happens on a write ?

Since the cost of an unnecessary access to the disk is very high, the write strategy is always **write back**. However, virtual memory systems usually include a **dirty bit**. It allows blocks to be written to disk only if they have been altered since being read from disk.

◆ Techniques for fast address translation

Page tables usually have large size. For example, given a 32-bit virtual address, 4 KB pages, and 4 bytes per page table entry, the size of the page table is $(2^{32}/2^{12}) \times 4 = 4\text{MB}$. Therefore, the address translation may be very slow.

One solution to this problem is also rely on the principle of locality;
if the accesses have locality, then the address translations for the accesses must also have locality.

We then can use a special cache, called **translation lookaside buffer (TLB)**, for address translation.



A TLB entry is like a cache entry where the tag holds portions of the virtual address. The data portion holds:

1. physical page number
2. protection field (for page-level protection)
3. valid bit
4. use bit (optional, for LRU)
5. dirty bit (optional, indicates whether the corresponding page is dirty or not, for write back).

The overall picture of a hypothetical memory hierarchy going from virtual address to L2 cache access

