Cache memories (Stone §2.2): A cache is a small, fast memory which is transparent to the processor and to the programmer.

- The cache duplicates information that is in main memory. It contains the subset of main memory data that is likely to be needed by the processor in the near future.

- It is needed because the speed of dynamic RAMs (main memory) have not kept pace with the improvements in logic speed.

We actually have a memory hierarchy, where the application programmer only sees a very fast and large virtual memory space that is implemented as:

- Very fast but small cache within the processor chip, which contains a subset of the

- Fast external cache, which contains a subset of the

- Main memory, which contains a subset of the

- Virtual memory
We want to structure the cache to achieve a high *hit ratio*.

- **Hit** — the referenced information is in the cache.
- **Miss** — the referenced information is not in cache, and must be read in from main memory.

\[
\text{Hit ratio} = \frac{\text{Number of hits}}{\text{Total number of references}}
\]

If \( h \) is the hit ratio, then we can also define \((1 - h)\) as the *miss ratio*.

Using the hit ratio, we can express the *effective access time* of a memory system using a cache as:

\[
t_{\text{eff}} = t_{\text{cache}} + (1 - h) t_{\text{main}}
\]

So, for a hit ratio of 90\% (0.9), a cache access time of 10 ns, and a main memory access time of 60 ns, we have:

\[
t_{\text{eff}} = 10 + (1 - 0.9) \times 60 = 16 \text{ ns}
\]
A cache can be organized according to four different strategies:

- Direct
- Fully associative
- Set associative
- Sectored

A cache implements several different policies for retrieving and storing information, one in each of the following categories:

- **Fetch policy**—determines when information is loaded into the cache.
- **Replacement policy**—determines what information is purged when space is needed for a new entry.
- **Write policy**—determines how soon information in the cache is written to lower levels in the memory hierarchy.

**Cache memory organization:** Information is moved into and out of the cache in **blocks** (block locations in the cache are called **lines**). Blocks are usually larger than one byte (or word),

- to take advantage of locality in programs, and
- because memory can be organized so that it can optimize transfers of several words at a time.

A **placement policy** determines where a particular block can be placed when it goes into the cache. E.g., is a block of memory eligible to be placed in any line in the cache, or is it restricted to certain lines or to a single line?
In the following examples, we assume—

- The cache contains 2048 words, with 16 words per line. Thus it has 128 lines.

- Main memory is made up of 256K words, or 16384 blocks. Further, if we have 4 bytes per word, we have a total of 1 Mbyte.

As noted above, there are four different placement policies. The simplest of these is:

**Direct Mapping:** The main memory block is eligible to be placed into only one line of the cache.

\[
\text{block } i \rightarrow \text{ line } \ mod \ 128
\]

Each line has its own tag associated with it.

When the line is in use, the tag contains the high-order seven bits of the main-memory address of the block.
Direct Mapped Cache:

**Cache Memory**

- Tag
- Line 0
- Line 1
- Line 2
- ... (Line 127)

**Main Memory**

- Block 0
- Block 1
- Block 2
- ... (Block 127, 128, 129, 130, 16,382, 16,383)

**Main Memory Address**

- 7 bits
- 7 bits
- 4 bits

To search for a word in the cache,

1. Determine what line to look in – easy; just index into the cache using the *line* bits of the address.

2. Compare the *tag* bits (leading seven bits) of the address with the tag of the line. If it matches, the block is in the cache.

3. Select the desired word from the line using the *word* bits.

**Advantages:**

- Fast lookup (only one comparison needed).
- Cheap hardware (no associative comparison).
- Easy to decide where it is or where to place it.
**Fully associative**: The block can be placed into *any* line in the cache.

block $i \rightarrow$ any free (or purgeable) cache location

Each line has its own tag associated with it.

When the line is in use, the tag contains the high-order *fourteen* bits of the main-memory address of the block.

To search for a word in the cache,

1. Simultaneously compare the *tag* bits of the address (leading 14 bits) with the tag of all lines in the cache. If it matches any one, the block is in the cache.

2. Select the desired word from the line using the *word* bits.

**Advantages**:

- Minimal contention for lines.
- Wide variety of replacement algorithms feasible.
**Set associative:** $1 < n < 128$ choices of where to place a block.

A *compromise* between direct and fully associative strategies.

The cache is divided into $s$ sets, where $s$ is a power of 2.

$$ \text{block } i \rightarrow \text{any line in set } i \mod s $$

Each line has its own tag associated with it.

For a *two-way set-associative* cache organization, ($s = 64$) the tag contains the high-order **eight** bits of the main-memory address of the block. The next six bits is the set number (64 sets).

**Set-associative Cache**

![Cache Diagram](attachment:cache-diagram.png)
To search for a word in the set-associative cache,

1. Index into the set using the set bits of the address \((i \mod s)\).

2. Simultaneously compare the tag bits (leading 8 bits) of the address with the tag of all lines in the set. If it matches any one, the desired block is in the cache.

   **Concurrent with comparing the tag bits**, begin reading out the selected data words of all lines in the set (using the word bits) so the data words will be available at the end of the compare cycle.

3. If a tag match is found, gate the data word from the selected line to the cache-output buffer and on to the processor. Discard the other words.

A different diagram can also be found in Stone, Fig. 2.7, page 38.
Set-associative cache size

For a set-associative cache, Stone (page 39) defines size parameters:

- $L$ – bytes per line
- $K$ – lines per set
- $N$ – sets

The total number of bytes in the cache is then:

$$\text{Cache size} = LKN$$

We can actually use this for the direct-mapped and fully-associative cache as well.

- for the direct-mapped cache, $K = 1$ – there is one line per set.
- for the fully-associative cache, $N = 1$ – there is only one set.
**Sectored Cache**: Another (little used) cache organization is the sectored cache. The main memory is partitioned into sectors, each containing several blocks.

The cache is partitioned into sector frames, each containing several lines. (The number of lines/sector frame = the number of blocks/sector.)

When block $b$ of a new sector $c$ is brought in,

- it is brought into line $b$ within some sector frame $f$, and
- the rest of the lines in sector frame $f$ are marked invalid (and not immediately filled).

Thus, if there are $S$ sector frames, there are $S$ choices of where to place a block.

- There are 128 lines in the (example) cache.
- If we have 16 blocks per sector, then there are only 8 sector frames in the cache.

When a frame is in use, its tag contains the high-order 8 bits of the main-memory address of the block.

There are relatively few tags, and all tags are compared simultaneously.

The main difference with a sectored cache is that we first perform an associative lookup on the high-order (sector) bits to see if we have allocated a frame for this sector. If the sector frame is allocated, we then index into the sector to access the proper line within the sector, which may or may not be valid.

A problem with this organization is that we must complete the parallel compares for the sector before we can begin reading out the selected word in the sector. Compare this with the set-associative organization where we can overlap the tag compare with reading out data from each of the lines in the selected set.
**Sectored Cache**

To search for a word in the sectored cache,

1. Simultaneously compare the sector bits of the address with the sector tags of the cache. If it matches any one, the desired sector is in the cache.

2. Index into the selected sector using the block bits of the address and test the line valid bit. If the valid bit is on, the desired block is in the cache.

3. Select the desired word from the line using the word bits.

The assumption with the sectored cache is that a program has a few, large areas of memory that it is using.
Factors influencing cache line lengths:

♦ Long lines ⇒ higher hit ratios.

♦ Long lines ⇒ less memory devoted to tags.

♦ Long lines ⇒ longer memory transactions (undesirable in a multiprocessor).

♦ Long lines ⇒ more write-backs (explained below).

For most machines, line sizes between 16 and 64 bytes perform best.

Factors influencing associativity

If there are \( b \) lines per set, the cache is said to be \( b \text{-way} \) set associative.

♦ Almost all caches built today are either direct mapped, or 2- or 4-way set-associative.

As cache size is increased, a high degree of set associativity becomes less important.

♦ A high degree of set associativity gives more freedom of where to place a block in the cache. So it tends to decrease the chance of a conflict requiring the purge of a line.

♦ A large cache also tends to decrease the chance that two blocks will map to the same line. So it tends to decrease the chance of a conflict requiring the purge of a line.
**Diminishing returns**: As the cache size grows larger, it becomes more and more difficult to improve performance by increasing the size of the cache.

The 30% rule of thumb: Doubling cache size reduces the number of misses by 30% (Stone page 42).

![Graph showing cache size vs. miss ratio]

Cache memory need not grow linearly with physical memory. (Program locality doesn’t grow linearly with physical memory.)

**Most systems today have at least two levels of cache.**

- The first level captures most of the hits.
- The second level is still much faster than main memory.
- Performance is similar to what would be obtained with a cache as fast as the first level.
- Cost is similar to that of the second level.
Two-level caches: In recent years, the processor cycle time has been decreasing much faster than even the cache access times.

- Processor cycle times are now 5 to 10 times shorter than external cache access times.

Caches must become faster to keep up with the processor.

- This means they must be on chip.
- But on-chip caches cannot be very big.

The only way out of this dilemma is to build—

- a small first-level (L1) cache, which is on the processor chip and fast,
- a larger second-level (L2) cache, which is external to the processor.

A miss in the L1 cache is serviced in the L2 cache, a miss in the L2 cache is serviced by the main memory.

To analyze the effect of a second-level cache, we can refine the definition of the hit ratio for any level of cache to be the number of hits to the cache divided by the number of accesses by the processor. Note that the number of accesses by the processor is not necessarily the number of accesses to the particular level of cache.

\[
\text{Global hit ratio} \equiv \frac{\text{Number of hits}}{\text{Total number of accesses by the processor}}
\]
Using the global hit ratio for each level of cache, we can express the effective access time of a memory system with a two-level cache hierarchy as:

\[ t_{eff} = t_{L1cache} + (1 - h_{L1cache}) t_{L2cache} + (1 - h_{L2cache}) t_{main} \]

So, for a cache hierarchy with:

- L1 hit ratio of 80%
- L2 hit ratio of 90%
- L1 cache access time of 2 ns
- L2 cache access time of 10 ns
- main memory access time of 60 ns

\[ t_{eff} = 2 + (1 - 0.8) 10 + (1 - 0.9) 60 = 10 \text{ ns} \]
Cache fetch and replacement policies: The fetch policy determines when information should be brought into the cache.

- Demand fetching—
- Prefetching—

Prefetching policies:

- Always prefetch. Prefetch block \( i + 1 \) when a reference is made to block \( i \) for the first time.
- Prefetch on miss. If a reference to block \( i \) misses, then fetch block \( i \) and \( i + 1 \). (i.e., fetch two blocks at a time.)

To minimize the processor’s waiting time on a miss, the read-through policy is often used:

- Forward requested word to processor.
- Fetch rest of block in wraparound fashion.
**Write policies:** There are two cases for a write policy to consider.

**Write-hit policies:** What happens when there is a write hit.

- **Write-through** (also called store-through). Write to main memory whenever a write is performed to the cache.

- **Write-back** (also called store-in or copy-back). Write to main memory only when a block is purged from the cache.

**Write-miss policies:** What happens when there is a write miss. These policies can be characterized by three semi-dependent parameters.

- Write-allocate vs. no-write-allocate. If a write misses, do/do not allocate a line in the cache for the data written.

- Fetch-on-write vs. no-fetch-on-write. A write that misses in the cache causes/does not cause the block to be fetched from a lower level in the memory hierarchy.

- Write-before-hit vs. no-write-before-hit. Data is written into the cache before/only after checking the tags to make sure they match. Hence a write-before-hit policy will displace a block already there in case of a miss. This may be reasonable for a direct-mapped cache.

Combinations of these parameter settings give four useful strategies.

<table>
<thead>
<tr>
<th></th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Fetch-on-write, Write-validate</td>
<td>No</td>
</tr>
<tr>
<td>Yes</td>
<td>Fetch-on-write, Write-validate</td>
<td>Yes</td>
</tr>
<tr>
<td>No</td>
<td>Write-around</td>
<td>No</td>
</tr>
<tr>
<td>No</td>
<td>Write-invalid</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Fetch-on-write?**

**Write-allocate?**

**Write-before-hit?**
The shaded area in the diagram represents a policy that is not useful.

If data is going to be fetched on a write, it doesn’t matter whether or not write-before-hit is used.

If data is not fetched on a write, three distinct strategies are possible.

• If write-allocate is used, the cache line is invalidated except for the data word that is written. (Again, it doesn’t matter whether write-before-hit is used, because the same data winds up in the cache in either case.) This is called *write-validate*.

• If write-allocate is *not* used, then it does matter whether write-before-hit is used.
  – Without write-before-hit, write misses do not go into the cache at all, but rather go “around” it into the next lower level of the memory hierarchy. The old contents of the cache line are undisturbed. This is called *write-around*.
    (Note that in case of a write *hit*, writes are still directed into the cache; only misses go “around” it.)
  – If write-before-hit is used, then the cache line is corrupted (as data from the “wrong” block has just been written into it). The line must therefore be invalidated. This is called *write-invalidate*.

Typically, lines to be written are directed first to a write buffer (fast register-like storage), then later to main memory.

This avoids stalling the processor while the write is completed.
**Cache relationship to the I/O subsystem**

We can consider two ways to place the processor cache in relation to the I/O subsystem:

- All I/O goes through the cache.
- All I/O bypasses the cache and goes directly to the main memory.
It is clearly simpler to have the I/O go through the cache:

- No problems with consistency between the cache and main memory

It is clearly preferable for performance to have the I/O bypass the cache:

- I/O doesn’t need the speed of the cache
- I/O data would take up cache lines and lower the hit rate to the processor
- I/O data would cause contention for access to the cache

But with I/O bypassing the cache, we have to deal with the problem of cache consistency with main memory. One of two ways:

- The I/O processor can keep a copy of the cache directory (Stone page 84) and notify the cache when it is writing into memory at a location held in the cache.
- The cache can monitor the main memory bus and detect when data is being stored into memory at a location that it holds in the cache – bus snooping.

Bus snooping is the most common way used for uniprocessors. We will spend much more time on this subject in the discussion of multiprocessors.
Replacement policies:

Least Recently Used (LRU) is a good strategy for cache replacement.

In a set-associative cache, LRU is reasonably cheap to implement.

With the LRU algorithm, the lines can be arranged in an LRU stack, in order of recency of reference. Suppose a string of references is:

\[ a \ b \ c \ d \ a \ b \ e \ a \ b \ c \ d \ e \]

and there are 4 lines. Then the LRU stacks after each reference are:

\[
\begin{align*}
&\text{a} \ b \ c \ d \ &\text{a} \ b \ c \ d \ &\text{a} \ b \ c \ d \ &\text{a} \ b \ c \ d \ &\text{a} \ b \ c \ d \\
&\text{a} \ b \ c \ d \ &\text{a} \ b \ c \ d \ &\text{a} \ b \ c \ d \ &\text{a} \ b \ c \ d \ &\text{a} \ b \ c \ d \\
&\text{a} \ b \ c \ d \ &\text{a} \ b \ c \ d \ &\text{a} \ b \ c \ d \ &\text{a} \ b \ c \ d \ &\text{a} \ b \ c \ d \\
&\text{a} \ b \ c \ d \ &\text{a} \ b \ c \ d \ &\text{a} \ b \ c \ d \ &\text{a} \ b \ c \ d \ &\text{a} \ b \ c \ d \\
&\text{*} \ &\text{*} \ &\text{*} \ &\text{*} \ &\text{*} \\
\end{align*}
\]

Notice that at each step:

- The line that is referenced moves to the top of the LRU stack.
- All lines below that line keep their same position.
- All lines above that line move down by one position.

Implementation 1 — age counters: Associate a two-bit counter with each line, and update them according to the above procedure. (When a line is referenced, set its counter to 0.)

If a line must be purged from the cache, purge the line whose counter is 3.

For larger sets,

- the set of lines is partitioned into several groups.
- the LRU group is determined.
- the LRU element in this LRU group is replaced.
LRU for a two-way set-associative cache is very simple to implement using a single bit per set.

- If line zero is referenced, set the bit to zero.
- If line one is referenced, set the bit to one.
- When a line needs to be replaced, use the line that is the complement of the bit.

There is also the discussion of the theoretical replacement algorithm OPT in Stone page 70.

**Split cache:** Multiple caches can be incorporated into a system design.

Most processors today have separate instruction and data caches:

- I-cache
- D-cache

This is also called a *Harvard cache* after work done at Harvard University.

A split cache can improve processor execution rate because we need to fetch *both instructions and data* from the memory system. If we can fetch them in simultaneously from two different areas of memory (instruction area and data area), we double the execution rate.

However, the hit rate may not be as high with the split cache because we have two smaller caches instead of one large cache.

A simplification with the instruction cache is that we don’t have to include logic to write into the cache from the processor. We just *forbid writing into the instruction stream* (a good practice in any event).
Cycles per Instruction – CPI

Stone §2.2.9 argues that MIPS is not a fair measure of performance, and that *Cycles per Instruction* (CPI) is better. He then defines CPI:

\[
CPI = CPI_{\text{Ideal}} + CPI_{\text{Finite Cache}} + CPI_{\text{Trailing Edge}} + CPI_{\text{Contention}}
\]

♦ **CPI\text{Ideal}** is the rate assuming a 100% cache hit rate.

♦ **CPI\text{Finite Cache}** is the additional cycles due to cache misses that require a full main memory access delay.

♦ **CPI\text{Trailing Edge}** is the additional cycles due to cache misses that require only a partial main memory access delay because the line is already being read due to a miss on a prior word in the line.

♦ **CPI\text{Contention}** is the additional cycles due to contention for the main memory bus.

Note that here we are talking about cycles per instruction, *not cycles per memory access*. Each instruction may have several memory accesses – one for the instruction itself, two to fetch operands and one to store results back to memory.

We will also find later that, in many cases, we can *overlap instruction execution with memory accesses*, effectively hiding some part of the memory access time.

I prefer seeing performance of processors expressed as MIPS, given a clock rate. It’s easier to relate to the real world.

\[
\text{MIPS} = \frac{\text{MHz}}{\text{CPI}}
\]