**Cache**

- Part of the storage hierarchy
- Tape ➔ HD ➔ RAM ➔ Cache ➔ CPU
- Cheap/Big/Slow ➔ Expensive/Small/Fast
- Cache holds a copy of selected data from RAM
- Data in cache must be tagged with its address to identify where it comes from.
- When CPU issues an address, system first tries to fetch it from cache.
- If data is in cache, this is called a **cache hit**, performance is best.
- If data is not in cache, this is called a **cache miss**, and we must fetch it from RAM into the cache before proceeding, performance is slower.
- Hit ratio: ratio of address references that result in a cache hit.
- Consistently high system performance requires high hit ratios (> 0.9)

**Different Kinds of Cache**

- Cache Levels: L1, L2, L3.
- Typical design is L1 and L2 cache for each core, with L3 cache shared by all cores, all on chip.
- In older designs, cache might be a separate chip on the motherboard.
- Unified Cache: holds both instructions and data
- Split Cache: different caches for instructions and data

**Principle of Program Execution Locality**

- When a program is executed by the CPU, the PC (program counter) maintains the address of the next instruction to fetch from RAM.
- Normal execution is to execute instructions in order from RAM by incrementing the PC.
- But programs with loops and if-else require **branch instructions** that move the PC to a different address that is not simply the next instruction in sequence.
- When a CPU is executing a loop, the PC cycles through a small set of instructions over and over again, for some number of iterations. Instructions inside the loop are in order, but a branch at the end of the loop resets the PC back to the beginning of the loop.
- We say that the program execution exhibits **locality of reference** by fetching instructions from one small area of memory for an extended time.
- If a program is currently focusing on instructions from one small area of memory, not all of the program needs to be in cache at the same time, we can get high performance just by getting the currently executing instructions into cache.
- If the PC randomly picked instructions around memory, there would be no locality. Without the principle of locality, cache would not improve performance because there would be no obvious set of instructions and data to keep in cache.

**Cache Mapping**

- Cache is designed to provide fast access to selected data from RAM.
- But cache is much smaller than RAM so not all data from a program can be stored in cache at the same time.
- When it comes time to copy a block of RAM into cache, we have to have a mapping.
- A mapping defines how we copy a given RAM block into a given cache line.
- Since cache has far fewer lines than RAM, many RAM blocks will map to a single cache line
- When looking up cache contents, we have to be able to determine whether a cache line contains the RAM block we're looking for (a hit) or some other block (a miss).

**Types of Cache Mapping**

- Direct Mapping
- Fully Associative
- Set Associative

**Direct Mapping**

- Easiest to understand and implement
- Each RAM block maps to a specific fixed cache line.
- Multiple RAM blocks map to a single cache line.
- Tag bits are stored to indicate which RAM block is currently occupying a cache line.

**Lookup Algorithm**

- CPU issues an address.
- We first check cache for a copy.
- Middle bits of address are used to index to a specific cache line.
- If the tag stored at that cache line matches the tag of the actual address, we have a hit.
- If the tag doesn't match, we have a cache miss, and must fetch the block from RAM and store it into cache, potentially overwriting whatever data was currently in that cache line.

**Let's Design a Direct Mapped Cache**

**Logical Address Space**

- Determines how many addresses can generated, related to the width of the address bus.
- 32 bit addresses imply $2^{32}$ possible addresses, this is the **logical address space** for the program.
- Note that there is a difference between the number of theoretically possible addresses and the amount of RAM actually and physically present and installed on the system.

**Block or Line Size**

- The **block or line size** determines how much information is moved in a single operation between RAM and cache. Moving data one byte at a time would result in poor performance.
- **Line size** is a fundamental cache design parameter, it impacts the implementation of cache in hardware. However, it is not a fundamental parameter of RAM implementation.
- Historically, a common cache line size is **64 = $2^6$ bytes**.
- Once we know the line size, we can then logically divide both RAM and cache into some number of blocks or lines of the same size, where the number of RAM blocks is much larger than the number of cache lines.
- Each line is like a row in a matrix, while a specific byte within the line is like column entry for that row.

**Cache Size**

- Here is an example implementation for L1, L2, and L3 cache for the Intel Nehalem architecture.
- See Wikipedia article on Nehalem or paper at http://sc.tamu.edu/systems/eos/nehalem.pdf
    - L1: per core, 64 B line, split
        - 32 kiB data, 8 way set associative
        - 32 kiB instruction, 4 way set associative
    - L2: per core, 64 B line, unified, 256 kiB, 8 way set associative
    - L3: shared by all cores, 64 B line, 8 MiB, 16-way set associative

**Example**

- Addresses: 8 bit addresses, or $2^8$ = 256 addressable bytes
- Line size: 4 bytes = $2^2$ bytes per block
- Each 8 bit address identifies a single byte in RAM.
- The first 6 bits identify the line or block
- The last two bits identify the byte within the block (aka byte offset).
- RAM size: 256 bytes, or 256/4 = 64 lines of 4 bytes each.
- Cache size: $2^6$ = 64 bytes, or 64/4 = 16 = $2^4$ lines of 4 bytes each, associativity = direct mapped.

From the point of view of RAM, 8 bit addresses look like this:

| 101110 | 01 |
|---|---|
| Line | Byte within line |
| 6 bits | 2 bits |

From the point of view of cache, they look like this:

| 10 | 1110 | 01 |
|---|---|---|
| Tag | Line | Byte within Line |
| 2 bits | 4 bits | 2 bits |

So if the CPU generates the 8 bit address 10111001 ($377_{10}$), this references RAM block 101110 ($46_{10}$), byte 1. So this is block 46 out of 64 total RAM blocks. All 4 bytes of block 101110 ($46_{10}$) will be moved to cache line 1110 ($14_{10}$), or line 14 out of 16 total cache lines. Since there are other RAM blocks that map to this same cache line, the tag bits must be stored so that we can tell exactly where the cache line came from in RAM.

Other blocks that map to the same cache line:

- 001110 (block $14_{10}$)
- 011110 (block $30_{10}$)
- 101110 (block $46_{10}$)
- 111110 (block $62_{10}$)

RAM:  256 bytes (64 blocks of 4 bytes each)

| | Byte 0 | Byte 1 | Byte 2 | Byte 3 |
|---|---|---|---|---|
| Block 0 | | | | |
| Block 1 | | | | |
| Block 2 | | | | |
| …. | | | | |
| Block 30 | 00 | 13 | 57 | DF |
| … | | | | |
| Block 46 | 13 | FF | 26 | 9A |
| … | | | | |
| Block 62 | | | | |
| Block 64 | | | | |

Cache: 64 bytes (16 lines of 4 bytes each)

Cache line 14 holds contents of RAM block 46

| | Tag | Byte 0 | Byte 1 | Byte 2 | Byte 3 |
|---|---|---|---|---|---|
| Line 0 | | | | | |
| Line 1 | | | | | |
| Line 2 | | | | | |
| …. | | | | | |
| Line 14 | 10 | 13 | FF | 26 | 9A |
| Line 15 | | | | | |

Later on, suppose we bring in RAM block 30 into cache. $30_{10} = 011110_2$. This block will also map to cache line $1110 = 14_{10}$.

Cache line 14 holds contents of RAM block 30, tag = 01

| | Tag | Byte 0 | Byte 1 | Byte 2 | Byte 3 |
|---|---|---|---|---|---|
| Line 0 | | | | | |
| Line 1 | | | | | |
| Line 2 | | | | | |
| …. | | | | | |
| Line 14 | 01 | 00 | 13 | 57 | DF |
| Line 15 | | | | | |

So to recap, when we view the content of cache line 14, it could have come from RAM blocks 14, 30, 46, or 62. But the tag bits will make it clear which one it actually came from.

**Fully Associative Cache**

- Direct mapped cache is the simplest and cheapest cache to implement. There is essentially no associative search required when checking tag bits, only one tag per line to check.
- But the negative is that a RAM block has only one line in the entire cache where it can be stored, even if there are other cache lines not being used at the moment. Storing into that one line erases anything already stored there. In the example above, there's no way blocks 30 and 46 can be stored in cache at the same time since they both are constrained to use line 14.

- Fully associative cache is the opposite extreme to direct mapped cache.
- In fully associative cache, a RAM block can be stored in **any** cache line.
- There is possibly less wasted space in cache, a RAM block can be copied to any available cache line.
- The negative is when we need to look up an address in cache.
- For direct mapped, we can look at the address and immediately index the one line where it might be stored, and do a quick comparison of the stored tag bits.
- For fully associative, we cannot index to a single line, the block could be stored anywhere.
- Instead we must check every tag for every line for a match.
- As a practical matter, we cannot pause and do a linear search for a matching tag (too slow), we must use a special type of hardware storage called **associative memory** to store the tags. Associative memory allows us to search all the tags in parallel for a match. Note: only tag storage requires associative search, not the entire cache.

**Lookup Algorithm**

- CPU issues an address.
- We first check cache for a copy.
- No bits of address can be used to index to a specific cache line.
- Instead, the entire block address is used for the tag.
- All tags for all lines in the entire cache must be searched associatively for a match.
- If the tag stored at that cache line matches the tag of the actual address, we have a hit.
- If the tag doesn't match, we have a cache miss, and must fetch the block from RAM and store it into cache, potentially overwriting whatever data was currently in that cache line.
- But with fully associative, we can use any cache line to store any RAM block.

**Example**

Let's redesign the above cache to use fully associative cache instead of direct mapped.

- Addresses: 8 bit addresses, or $2^8$ = 256 addressable bytes
- Line size: 4 bytes = $2^2$ bytes per block
- Each 8 bit address identifies a single byte in RAM.
- The first 6 bits identify the line or block
- The last two bits identify the byte within the block.
- RAM size: 256 bytes, or 256/4 = 64 lines of 4 bytes each.
- Cache size: $2^6$ = 64 bytes, or 64/4 = 16 = $2^4$ lines of 4 bytes each, associativity = fully associative

From the point of view of RAM, 8 bit addresses look like this:

| 101110 | 01 |
|---|---|
| Line | Byte within line |
| 6 bits | 2 bits |

From the point of view of cache, they look like this:

| 101110 | 01 |
|---|---|
| Tag | Byte within Line |
| 6 bits | 2 bits |

What's different is that since a RAM block can be stored in any cache line, we cannot reserve address bits to use as a cache line index. Instead, all 6 bits that identify the RAM block must be kept as tag.

Cache: 64 bytes (16 lines of 4 bytes each)

Cache line 3 might hold contents of RAM block 46 (it can be in any line, not just line 14):

| | Tag | Byte 0 | Byte 1 | Byte 2 | Byte 3 |
|---|---|---|---|---|---|
| Line 0 | XXXXXX | | | | |
| Line 1 | XXXXXX | | | | |
| Line 2 | XXXXXX | | | | |
| Line 3 | 101110 | 13 | FF | 26 | 9A |
| … | | | | | |
| Line 14 | XXXXXX | | | | |
| Line 15 | XXXXXX | | | | |

RAM block 46 could be stored in any cache line, here it happens to be stored in line 3. When looking it up, to determine that block 46 is stored in line 3, we have to check all tags for all lines. Associative search does this check fast in parallel in hardware, and will report "match for line 3."

**Set Associative Cache**

- The final cache organization is a hybrid of the direct mapped and fully associative organization.
- Set associative means that indexing will be used to map a RAM block to a **set** of cache lines
- Each set contains a small number of lines, any line in the set can be used for that block.

**Examples:**

- 2-way set associative: each set contains two lines
- 4-way set associative: each set contains four lines
- 8-way set associative: each set contains eight lines
- 16-way set associative: each set contains sixteen lines

Note that these schemes only define how many lines per set there are, but not how many sets, that depends on the overall size of the cache.

- When storing a RAM block into cache, we use indexing to pick a set (similar to direct mapped).
- Within a set, we can use any available line (similar to fully associative).
- When searching the tags, we only search the tags within the set.

Let's redesign our example for a simple set associative cache.

- Addresses: 8 bit addresses, or $2^8$ = 256 addressable bytes
- Line size: 4 bytes = $2^2$ bytes per block
- Each 8 bit address identifies a single byte in RAM.
- The first 6 bits identify the line or block
- The last two bits identify the byte within the block.
- RAM size: 256 bytes, or 256/4 = 64 lines of 4 bytes each.
- Cache size: $2^6$ = 64 bytes, or 64/4 = 16 = $2^4$ lines of 4 bytes each, associativity = 4-way set associative.
- The sixteen cache lines will be divided into 4 sets of 4 lines each.

From the point of view of RAM, 8 bit addresses look like this:

| 101110 | 01 |
|---|---|
| Line | Byte within line |
| 6 bits | 2 bits |

From the point of view of cache, they look like this:

| 1011 | 10 | 01 |
|---|---|---|
| Tag | Set | Byte within Line |
| 4 bits | 2 bits | 2 bits |

- Since there are 4 possible cache sets the block could be stored in, we use 2 address bits to index to a specific set.
- Within that set there are 4 lines, any of which can hold the block.
- For our example block 46, the set bits are 10, so the block must be stored in set 2. But within the 4 lines of that set, it can be in any line. The tag bits will need to be searched for all 4 lines in the set to confirm a match.

Here's what the entire 16 lines of cache looks like

| | | Tag | Byte 0 | Byte 1 | Byte 2 | Byte 3 |
|---|---|---|---|---|---|---|
| Set 0 (00) | Line 0 | | | | | |
| | Line 1 | | | | | |
| | Line 2 | | | | | |
| | Line 3 | | | | | |
| | | | | | | |
| Set 1 (01) | Line 0 | | | | | |
| | Line 1 | | | | | |
| | Line 2 | | | | | |
| | Line 3 | | | | | |
| | | | | | | |
| Set 2 (10) | Line 0 | XXXX | | | | |
| | Line 1 | 1011 | 13 | FF | 26 | 9A |
| | Line 2 | XXXX | | | | |
| | Line 3 | XXXX | | | | |
| | | | | | | |
| Set 3 (11) | | | | | | |
| | Line 0 | | | | | |
| | Line 1 | | | | | |
| | Line 2 | | | | | |
| | Line 3 | | | | | |

**Lookup Algorithm for Set Associative Cache**

- CPU issues 8 bit address 10111001
- This divides into a 6 bit block address = 101110 (46), and a two bit byte address = 01
- Block address divides into 4 bit tag = 1011 and 2 bit set 10
- We use the set bits to index to set 10 (2) in cache (indexing is fast, no searching required).
- If block is in cache, it must be in 1 of the 4 lines in this set, it cannot be stored in sets 0, 1, or 3.
- Associatively search the tags of the 4 lines in set 10 (2) for a match with tag bits 1011 from the address.
- If match is found, we have a cache hit, fetch the data from that cache line.
- If no match, must fetch block from RAM and load result into cache, same set but might be a different line (any available line in that set).