Imperial College London

Lecture 9

Memory Hierarchy - Cache Memory

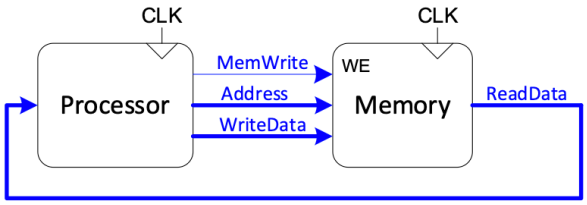
Peter Cheung
Imperial College London

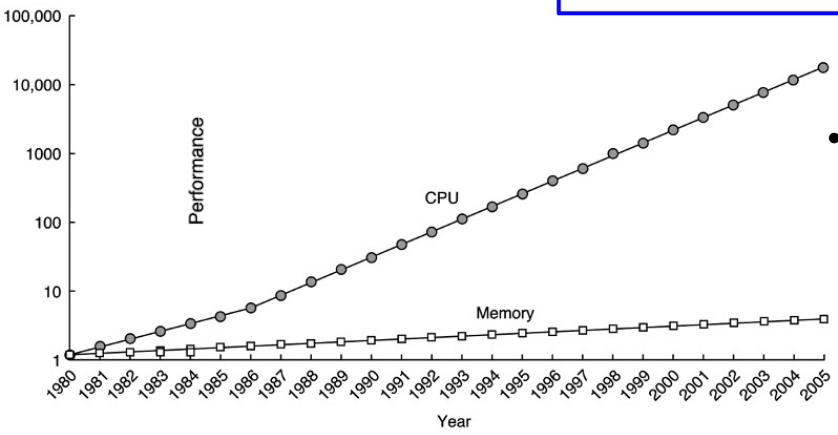
URL: www.ee.imperial.ac.uk/pcheung/teaching/EE2_CAS/
E-mail: p.cheung@imperial.ac.uk

This lecture is about how memory is organized in a computer system. In particular, we will consider the role play in improving the processing speed of a processor.

Processor to Memory Interface

- **Computer performance depends on:**
  - **Processor** performance
  - **Memory system** performance





- **Ideal memory**
  - **Fast**
  - **Cheap**
  - **Large**

Based on: *"Digital Design and Computer Architecture (RISC-V Edition)"* by Sarah Harris and David Harris (H&H),

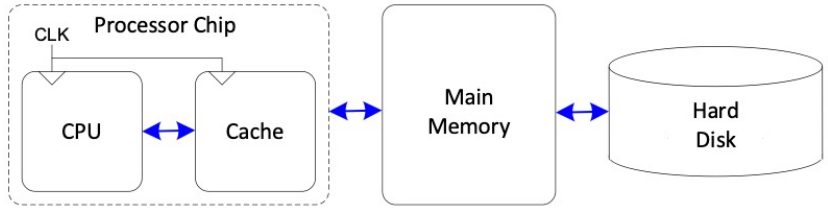
In our single-cycle instruction model, we assume that memory read operations are asynchronous, immediate and also single cycle. In reality, memory devices are synchronous both for read and write, and often take many cycles to complete. In many embedded processors using microcontrollers, some memory may not even use parallel data bus. For example, many microcontrollers such as the ESP32-C3 processor used with Vbuddy, it uses SPI memory that communicates with the processor in serial format.

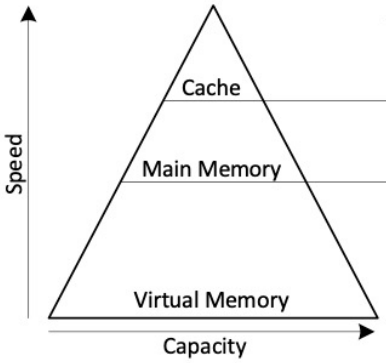
Ideally we want memory to be as fast as the processor, very low cost (because we use plenty of them in a system) and has large capacity. However, technological developments over decades have seen processor speed improving as a much faster than memory (as shown in the graph above).

Of the three characteristics: high speed, low-cost and large capacity, we can achieve at best two out of the three. For example, we can have very large storage at reasonably low cost, but it will be slow. Hard disk is one such example. We may have fast main memory on a computer with large capacity (say 32GB), but it will cost a lot.

What do we mean by "fast" in memory? It can mean access time of the memory from address or control signals to data being valid. In the case of synchronous memory it also refers to the latency of access, i.e. the number of clock cycles needed before data is valid.

Memory Hierarchy





Technology	Price / GB	Access Time (ns)	Bandwidth (GB/s)
SRAM	$100	0.2 - 3	100+
DRAM	$3	10 - 50	30
SSD	$0.10	20,000	0.05 - 3
HDD	$0.03	5,000,000	0.001 - 0.1

Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

PYKC 19 Nov 2024 EIE2 Instruction Architectures & Compilers Lecture 9 Slide 3

Memory does not need to be ”fast” provided that the delay or latency in access does not affect the performance of the processor. A way to mitigate the disparity between processor speed and the relatively slow memory access of very large storage is to introduce memory hierarchy.

Computers are organized such that the CPU has access to very fast storage very near to the ALU itself. The fastest is the Register File, which is used for temporary storage. Then we have very fast on-chip memory called cache memory which supply recently used instruction and/or data. External to the processor is the main processor memory which is usually dynamic memory (DRAM). For example, your laptop will have say at least 8GB if not 16GB of RAM as the main memory external to the processor. Finally, your files are stored in the disk storage, which nowadays are less likely to be hard disk, but solid state disk based on silicon flash memory chips. If you back up your laptop, the back up disk is likely to be hard disk which of course is much lower. Alternative, instead of the hard disk, you may use cloud storage, which is even slower with wifi communication.

Big idea: Principle of Locality

Exploit locality to make memory accesses fast:

- **Temporal Locality:**
  - Locality in time
  - If data used recently, likely to use it again soon
  - **How to exploit:** keep recently accessed data in higher levels of memory hierarchy
- **Spatial Locality:**
  - Locality in space
  - If data used recently, likely to use nearby data soon
  - **How to exploit:** when access data, bring nearby data into higher levels of memory hierarchy too

Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

All prcoessors now include cache memory to mitigate against the disparity between processor and memory speed. The reason why cache works is based on two principles:

1. **Temporal Locality** – any instruction or data used previously is more likely to be used again. In the case of instruction memory, programs are often executing tight loops. That means recently executed instructions are likely to be executed again.

2. **Spatial Locality** – any instruction or data accesses are likely to be close to each other in address space. Instructions are executed in sequence unless that is a jump or branch. Therefore accessing instructions are likely to be spatially close. Data accesses are often to arrays or other data structures that are in consecutive address locations.

Memory Performance

- **Hit:** data found in that level of memory hierarchy
- **Miss:** data not found (must go to next level)

Hit Rate = # hits / # memory accesses
= 1 – Miss Rate

Miss Rate = # misses / # memory accesses
= 1 – Hit Rate

- **Average memory access time (AMAT):** average time for processor to access data

AMAT $= t_{\text{cache}} + MR_{\text{cache}}[t_{MM} + MR_{MM}(t_{VM})]$

Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

Effectiveness of cache memory is dependent on how often a memory access is found in the cache memory in a given level of memory hierarchy. A cache "hit" happens when the data or instruction required is found in the cache and therefore there is no need to go to the next level to fetch from memory. A miss happens when the data or instruction is not in cache, and a fetch from the main memory is required, thus incurring extra latency.

Hit and miss rate is therefore the number of hit or miss divided by the total number of memory accesses in a program. The average memory access time is given by the formula above with the following meaning:

t_{cache} – time taken to access cache memory

t_{MM} – time taken to access main memory

t_{VM} – time taken to access virtual memory on disk

MR_{cache} – cache miss rate

MR_{MM} – main memory miss rate

Memory Performance Example 1

- A program has 2,000 loads and stores
- 1,250 of these data values in cache
- Rest supplied by other levels of memory hierarchy
- What are the **cache hit and miss rates?**

Hit Rate = 1250/2000 = **0.625**

Miss Rate = 750/2000 = **0.375** = 1 – Hit Rate

Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

Here is a simple example to calculate the hit and miss rate of a program.

Memory Performance Example 2

- Suppose processor has 2 levels of hierarchy: cache and main memory
- t_{cache} = 1 cycle, t_{MM} = 100 cycles
- What is the **AMAT** (average memory access time) of the program from Example 1**?**

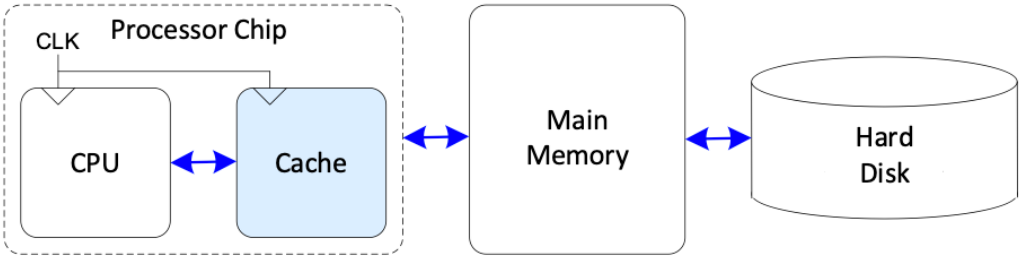
$$\text{AMAT} = t_{cache} + MR_{cache}(t_{MM})$$
$$= [1 + 0.375(100)] \text{ cycles}$$
$$= \textbf{38.5 cycles}$$

Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

Here is another example to calculate the average memory access time (AMAT). Note that t_{cache} and t_{MM} are specified in latency, i.e. number of clock cycles.

Cache Memory

- Highest level in memory hierarchy
- Fast (typically ~ 1 cycle access time)
- Ideally supplies most data to processor
- Usually holds most recently accessed data



Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

Cache memory is the most important feature of modern processor design that ensure high performance. It is usually very fast – taking only 1 clock cycle to access. It usually holds the most recently accessed data or instruction, thus exploiting the temporal locality property of programs.

Cache Design Questions

- What data is held in the cache?
- How is data found?
- What data is replaced?

- Ideally, cache anticipates needed data and puts it in cache
- But impossible to predict future
- Use past to predict future – temporal and spatial locality:
  - **Temporal locality:** copy newly accessed data into cache
  - **Spatial locality:** copy neighboring data into cache too

Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

When considering the design of a cache memory system, we ask the three most important questions shown above.

The design needs to determine if a memory access is a hit or a miss, i.e. is the data or instruction stored in the cache. If it is a read cycle and a hit, then a read from cache must be initiated. If it is a miss, then the hardware must go and fetch the data from the next level of the memory hierarchy and fill the cache with it. Finally, if the operation is a write, it must write to the cache AND to the next level of memory hierarch (i.e. in this case, the main memory). Since the cache has limited capacity, the design must determine which cache location is to be overwritten with the new data.

Ideally we want all memory access to be found in cache. This is of course impossible. However, we can used temporal and spatial locality property of memory accesses to maximize hit rate. So, newly accessed data are stored in cache. Furthermore, each time we fetch a new data and store to cache, we also read and store neighborough data to exploit spatial locality.

Cache Terminology

- **Capacity (*C*):**
  - number of data bytes in cache
- **Block size (*b*):**
  - bytes of data brought into cache at once
- **Number of blocks (*B = C/b*):**
  - number of blocks in cache: $B = C/b$
- **Degree of associativity (*N*):**
  - number of blocks in a set
- **Number of sets (*S = B/N*):**
  - each memory address maps to exactly one cache set

Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

Here are a number of terms used when characterizing cache memory. Exactly what they mean and how they affect the cache performance will become clear later.

Meanings of Capacity, Block size and number of blocks are obvious.

The meaning of sets and degrees of associativity will be explained in the next few slides.

How to find data in cache?

- Cache organized into *S* sets
- Each memory address maps to exactly one set
- Caches categorized by # of blocks in a set:
  - **Direct mapped:** 1 block per set
  - ***N*-way set associative:** *N* blocks per set
  - **Fully associative:** all cache blocks in 1 set
- Examine each organization for a cache with:
  - Capacity (***C*** = 8 words)
  - Block size (***b*** = 1 word)
  - So, number of blocks (***B*** = 8)

Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

Cache memory is organized into sets. Each set holds one or more blocks of data. The relationship between the address of the data in main memory and the location of that data in the cache is called *mapping*

Each memory address maps to exactly one set in the cache. But a cache location can be mapped tom many memory addresses. Some of the address bits are used to determine which cache set contains the data.

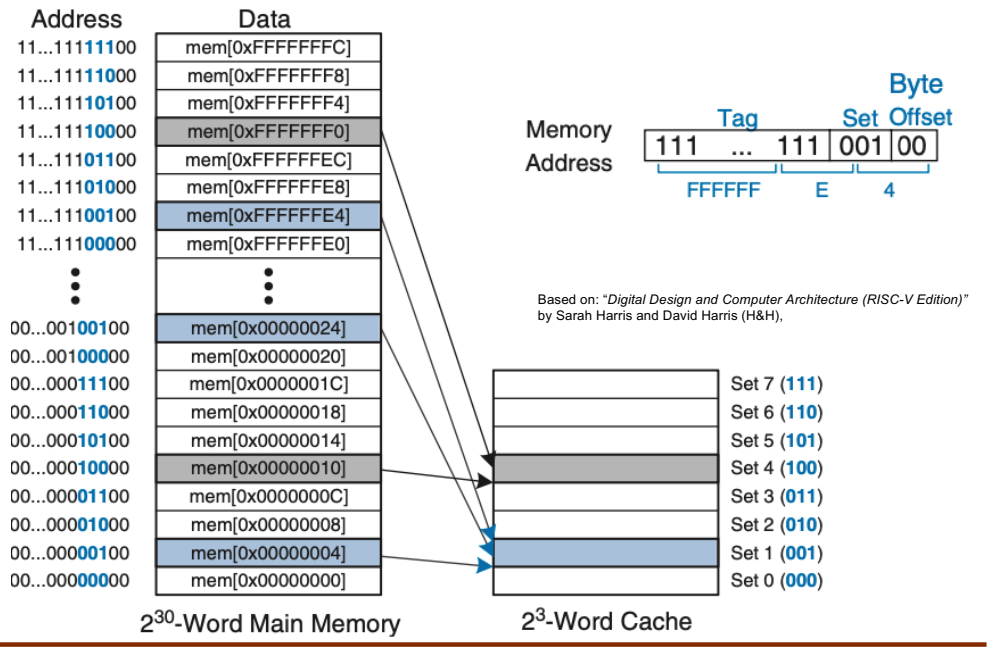
Caches are categorized based on the number of blocks in a set, as will be seen in the next few slides.

Essentially, if the cache has only 1 block per set, it is calle DIRECT MAPPED cache.

If the cache has 2 block per set, it is called a 2-WAY SET ASSOCIATIVE cache. Similar, N block per set cache is called N-way set associative cache.

Fully associative cache refers to all cache blocks belong to 1 set meaning that data can go in any of the blocks in the set. It is not used in practice.

Direct Mapped Cache



Here is an example of direct mapped cache. The assumption is that we have a cache capacity (C) of 8 words (32-bit). Block size is 1, therefore there are 8 sets (S).

Let us assume that the memory space is like RISC-V, 32-bit address and it is byte addressing. The bottom 2-bits of the address is the byte offset – it specifies which bytes is addressed.

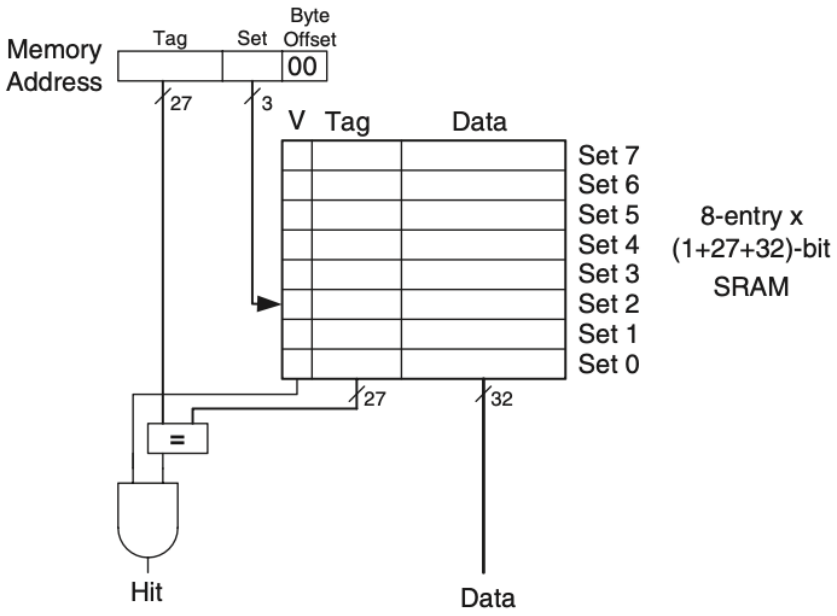
The next 3 bits of the address specify which cache location belongs to the set. As shown in the diagram, ALL MEMORY LOCATIONS with AD[4:2] = 3'b001 map to Set 1 cache as shown in BLUE.

Similar, Set 4 has the cache address 3'b100. All memory locations with AD[4:2] = 3'b100 map to this cache word.

It is called direct mapped cache because the mapping of a memory address to which cache location is **directly** derived from the address itself. There is no other decision to be made.

This of course has a problem – a cache location could be occupied by the data from MANY possible main memory addresses. How do we know if this one location in cache contains the actual data we want?

Direct Mapped Cache hardware



Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

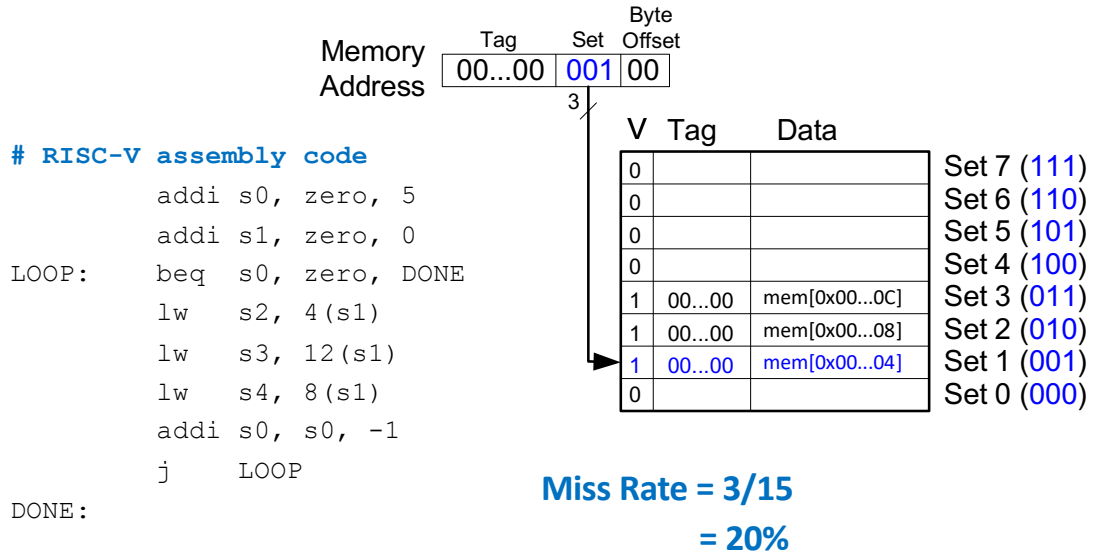
To identify exactly which memory content is currently in the cache, each cache entry has the 32-bit data, and the most-significant 27 bits of the address stored along side. This part of the address is called a "Tag" – it identify which memory location the cache is holding the data for.

One problem remains. Before anything is read, the cache is empty. Therefore we need to know if a cache location has anything meaningful stored in it or not. Therefore there is an extra bit called the "V" or **valid bit**. When the computer first powerup, the cache is empty and all V bits are 0. When a data is read from main memory and is stored in its direct mapped cache location, the corresponding V bit is set to 1 indicating that it has valid data. This valid data may be however be for a different address than the one that you want to read. So the top 27 bits of the memory address is compared to the tag stored in the cache location. If V is '1' and the Tag fields match, then there is hit and the signal "Hit" goes high.

Otherwise, it is a miss. The cache control must then read the relevant data from main memory, update the Tag and Data field of the cache.

Therefore each entry of the cache must have 32 bits for the data, 27 bits for the tag and 1 bit for valid flag, or 60 bits all together.

Direct Mapped Cache Miss Rate Example




```
# RISC-V assembly code
          addi s0, zero, 5
          addi s1, zero, 0
LOOP:     beq  s0, zero, DONE
          lw   s2, 4(s1)
          lw   s3, 12(s1)
          lw   s4, 8(s1)
          addi s0, s0, -1
          j    LOOP
DONE:
```

Miss Rate = 3/15
= 20%

Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

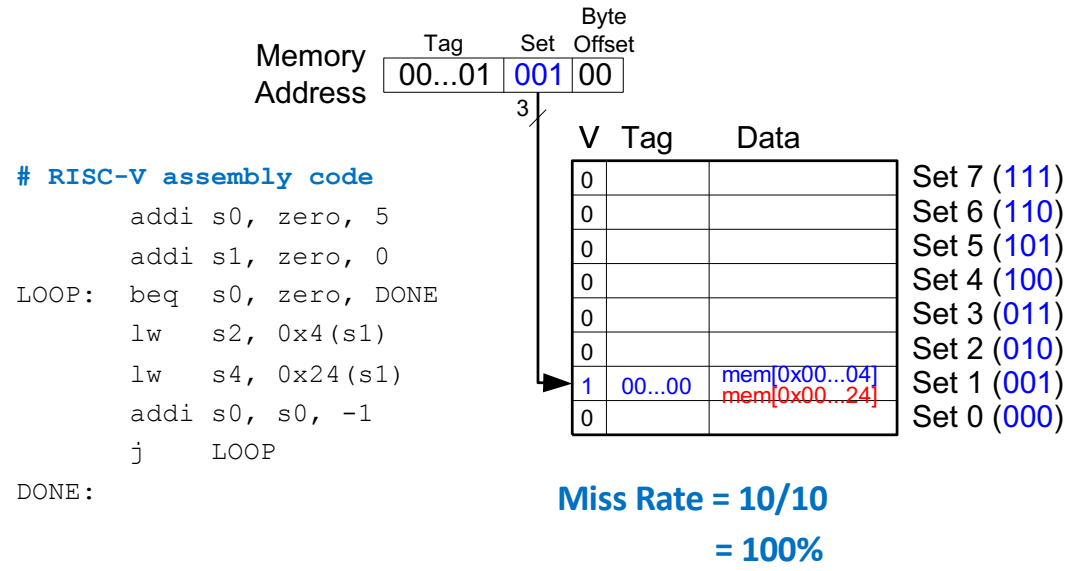
Consider what happens when the processor is executing this snippet of assembly code. Register s0 is used as a loop counter, counting down from 5 to zero. Register s1 is the address register into memory which has associated with it 8 words of direct mapped cache as shown.

The loop reads word data from addresses 0x4, 0x8 and 0xC (byte addressing), again and again for 5 times. This program does not do anything useful – it is just an example of cache memory.

The instruction lw s2, 4(s1) load a word from memory addrss 0x4 to Regsiter s2. The first time it happens, the cache does not contain the data, therefore it is a miss. However, each time round the loop for the remaining 4 times, the data is in the cache and therefore they are all hits. The same is true for the next two memory read instructions.

So the total number of memory access is 3 x 5 = 15, and there are 3 misses. Hence the miss rate is 20%.

Direct Mapped Cache Conflict Miss Example




```
# RISC-V assembly code
        addi s0, zero, 5
        addi s1, zero, 0
LOOP:   beq  s0, zero, DONE
        lw   s2, 0x4(s1)
        lw   s4, 0x24(s1)
        addi s0, s0, -1
        j    LOOP
DONE:
```

Miss Rate = 10/10
= 100%

Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

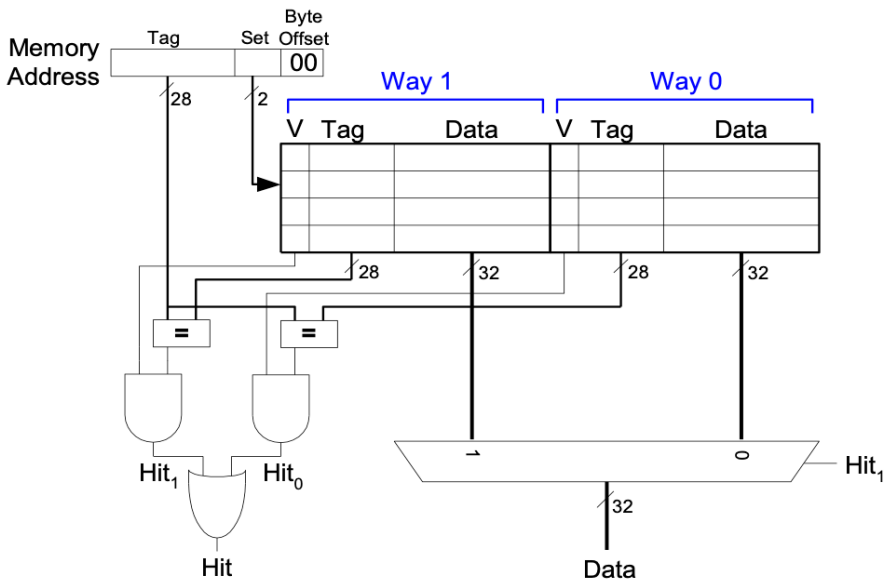
Here is another example. This time there are two memory read operations, one from 0x4 and a second from 0x24. Here both memory address map to the same cache location of 3'b001 (Set 1). Therefore each time around the loop, the WRONG data is stored in the cache location. This is called a conflict because the same cache is mapped to both memory locations being read in the loop.

Therefore all memory access in the loop are misses. The miss rate is there 100%.

It is clear that if we only have a block size of 1, as is the case here, the chance of a conflict miss is high. This is because MANY memory addresses are mapped to the same cache location.

One way to reduce conflict misses is to have block size more than 1 in a given set.

2-Way Set Associative Cache



Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

This is the structure of a cache organization where each set has TWO possible entries, so that instead of having 8 set of 1, we now have 4 sets, each have two storage locations. This is also called a **2-way set associative cache**. Similar to before, two of the address bits are used to identify which set the memory address is mapped to. However, each cache set can store TWO data from two different members of the set. The Tag comparison determines which if any of the two-way cache contains the actual data.

Now the number of locations mapped to each cache line is doubled. This is because the Tag is now 28 bits instead of 27 bits. However, each time the cache mapping identify a given cache location, there are two places that the data could go. This has the potential of reducing conflict misses.

2-Way Set Associative Cache Miss Rate Example

```
# RISC-V assembly code
        addi s0, zero, 5
        addi s1, zero, 0
LOOP:   beq  s0, zero, DONE
        lw   s2, 0x4(s1)
        lw   s4, 0x24(s1)
        addi s0, s0, -1
        j    LOOP
DONE:
```

Miss Rate = 2/10
= 20%

	Way 1			Way 0		
V	Tag	Data	V	Tag	Data	
0			0			Set 3
0			0			Set 2
1	00...10	mem[0x00...24]	1	00...00	mem[0x00...04]	Set 1
0			0			Set 0

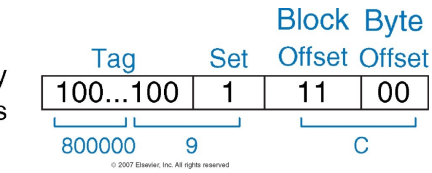
Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

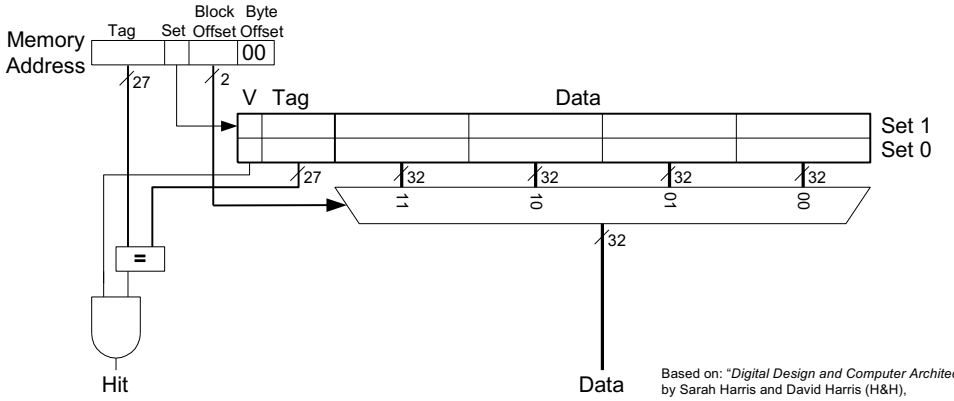
Using the example earlier where we had 100% miss rate due to conflict (i.e. successful memory access maps to the same cache location).

In this case, there are two locations in cache that can be mapped to the main memory in Set 1, the cache now stores the data from 0x4 AND 0x24 simultenously. There would be two misses when the cache was first filled. Thereafter, there is no need to read from main memory again. So the miss rate goes down from 100% to 20%.

Design for Spatial Locality

- Increase block size:
  - Block size, ***b* = 4 words**
  - *C* = 8 words
  - Direct mapped (1 block per set)
  - Number of blocks, ***B* = 2** (*C*/*b* = 8/4 = 2)



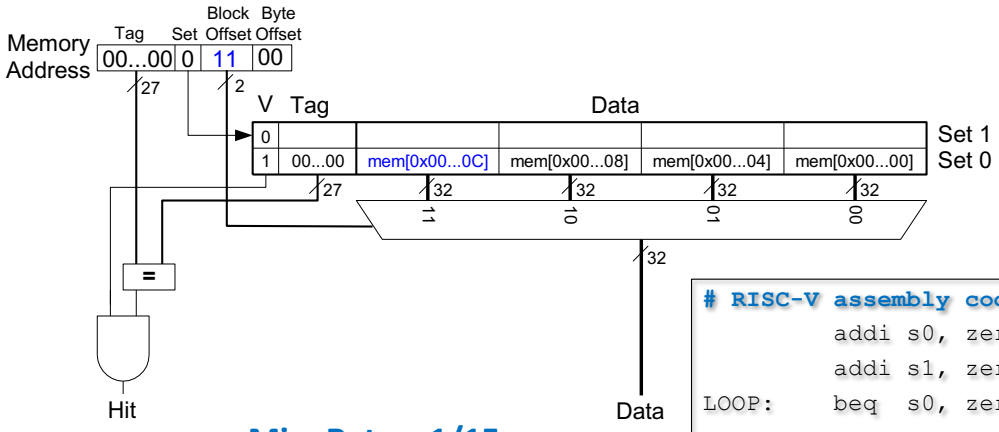


Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

So far, the cache organization does not exploit the property of spatial locality because neighbouring address locations are mapped to different cache locations since the block size was 1. However, we can increase the block size, say, to 4 as shown here.

That is everything a cache miss occurs, we fetch not only the missed data, but its closest neighbours as well in anticipation that they may be accessed next. The same 8 cache word storage is now dividied into two set, each stores one block of 4 words as shown in the diagram here.

Cache Performance with b = 4



Miss Rate = 1/15

= 6.67%

```
# RISC-V assembly code
        addi s0, zero, 5
        addi s1, zero, 0
LOOP:   beq  s0, zero, DONE
        lw   s2, 4(s1)
        lw   s3, 12(s1)
        lw   s4, 8(s1)
        addi s0, s0, -1
        j    LOOP
DONE:
```

Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

Here is again the example we considered earlier where we read from addresses 0x4, 0x8 and 0xC. However, all these three locations belong to the same set and have the same tag value. They are all stored in the same block such that each time one of these locations is accessed, the rest of the block will be read into the cache. Assuming that there is enough time that the first read fills the entire cache block, then there is only 1 miss in 15, and the miss rate drops to 6.67%.

Cache Organisation Recap

- **Type of misses:**
  - **Compulsory:** first time data accessed
  - **Capacity:** cache too small to hold all data of interest
  - **Conflict:** data of interest maps to same location in cache
- **Miss penalty:** time it takes to retrieve a block from lower level of hierarchy

- **Capacity:** C
- **Block size:** b
- **No. of blocks in cache:** $B = C/b$
- **No. of blocks in a set:** N
- **No. of sets:** $S = B/N$

Organization	Number of Ways (N)	Number of Sets (S)
Direct Mapped	1	B
Set Associative	$1 < N < B$	B/N
Fully Associative	B	1

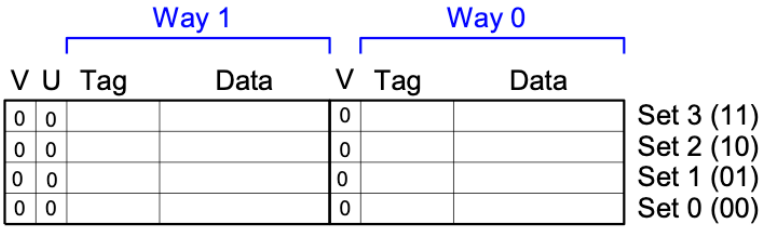
Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

Here is a re-cap fo the different organization of caches: direct mapped, set associative and fully associative (not used). They have three types of misses: first time (or compulsory), capacity and conflict. A combination of N-way set associative and larger block size provide the best compromise to reduce cache misses for a given capacity.

Replacement Policy

- Cache is too small to hold all data of interest at once
- If cache full: program accesses data X and evicts data Y
- ***Capacity miss*** when access Y again
- How to choose Y to minimize chance of needing it again?
  - **Least recently used (LRU) replacement**: the least recently used block in a set evicted

```
# RISC-V assembly
lw s1, 0x04(zero)
lw s2, 0x24(zero)
lw s3, 0x54(zero)
```



Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

When one uses N-way set associative cache, each cache set can store data with N different tags. The example here shows a 2-way associative cache. The code snippet accesses memory at 0x04, 0x24 and 0x54. All these three locations map to the same Set 1.

When 0x54 is to be read, there will be a miss because the cache contains the data from 0x04 and 0x24 already and is full. We now must kick out one of these two so that data from 0x54 can be stored. Which one should be kicked out and be replaced?

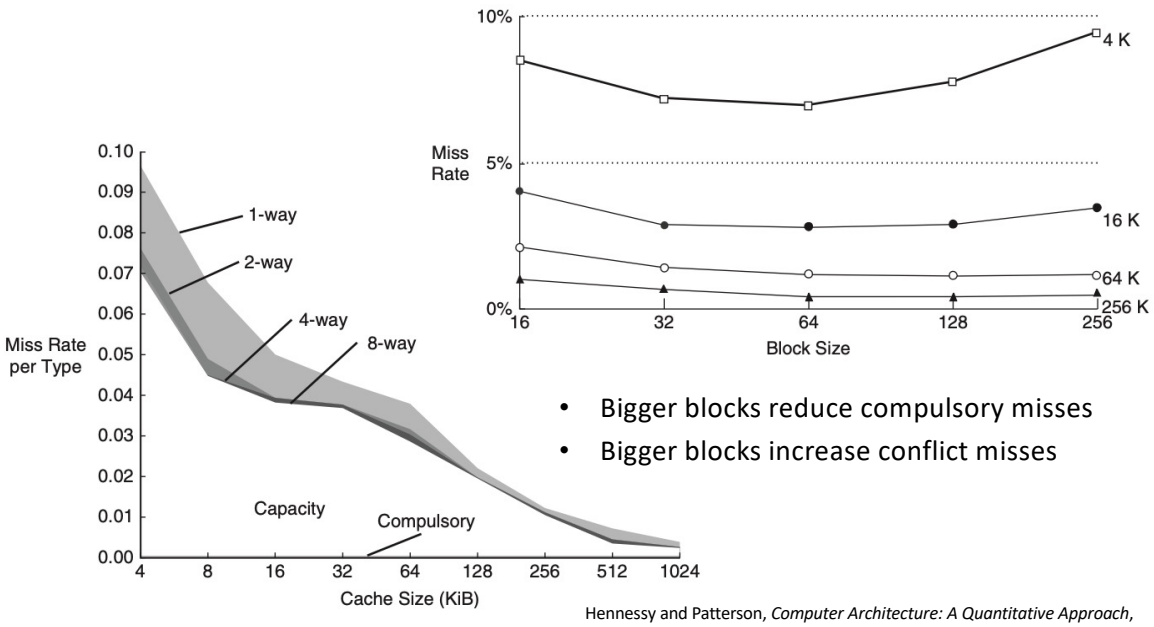
The most common method is called least recently used (LRU) replacement method. That is, we kick out the one that was least recently used and keep the one that was most recently used.

In the example here, the U-bit (use bit) indicates which way (i.e. Way 0 or Way 1) within the set was least recently used.

For set associative cache with more than two ways, tracking the least recently used way becomes very complicated. To make things simpler, the ways (say 4-way) are divided into two groups (each has 2 ways). U indicates which group of the wasy was least recently used.

When a replacement is needed, the new data replaces a random data within the least recently used group. Such a policy is called pseudo-LRU and is a frequently used policy in practice.

Cache Miss Rate vs Sizes



- Bigger blocks reduce compulsory misses
- Bigger blocks increase conflict misses

Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed., Morgan Kaufmann, 2012

Here is a plot of miss rate versus the size of cache for different ways of associativeness when running SPEC2000 benchmark. It shows the following:

1. Increase cache capacity (i.e. size) reduces miss rate.
2. Going from 1-way associative (i.e. direct mapped) to 2-way associative cache gives the biggest gain.
3. Increasing beyond 4 or 8 way associative yield diminishing return – so no point to go beyond 4 or 8 ways.

The plot of miss rate versus block size also produces good insights. The miss rate is minimum for block size of 64 bytes (x-axis is block size in bytes). Increasing the block size beyond this actually increases the miss rate.

Write policy for cache memory

- Same as cache read for hit/miss detection
- If cache misses, cache block is fetched from main memory, the word written to the cache block
- If cache hits, word is simply written to cache block – no need to fetc
- **Problem** – once a word is written, the cache contains data DIFFERENT from the main memory. This is known **as cache coherency** problem.
- There are two ways to handle this:
  - **Write-through cache** – data is written to cache and simultaneously to the main memory
  - **Write-back cache** – data is written to cache and a **dirty bit (D)** associated with the cache block is set. It is written back to main memory ONLY when the block is evicted from the cache.

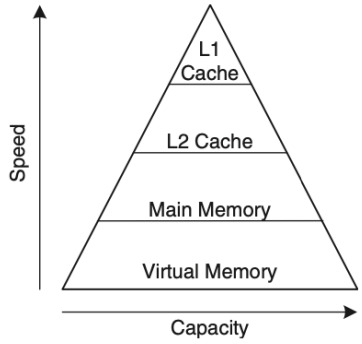
Here is a plot of miss rate versus the size of cache for different ways of associativeness when running SPEC2000 benchmark. It shows the following:

1. Increase cache capacity (i.e. size) reduces miss rate.
2. Going from 1-way associative (i.e. direct mapped) to 2-way associative cache gives the biggest gain.
3. Increasing beyond 4 or 8 way associative yield diminishing return – so no point to go beyond 4 or 8 ways.

The plot of miss rate versus block size also produces good insights. The miss rate is minimum for block size of 64 bytes (x-axis is block size in bytes). Increasing the block size beyond this actually increases the miss rate.

Multilevel Cache

- Larger caches have lower miss rates, longer access times
- Expand memory hierarchy to multiple levels of caches
- Level 1: small and fast (e.g. 16 KB, 1 cycle)
- Level 2: larger and slower (e.g. 256 KB, 2-6 cycles)
- Most modern PCs have L1, L2, and L3 cache



Modern processors do not only have ONE level of cache. Instead, they have multiple level of caches. For example if a processor has two level of cache, L1 and L2, L1 would be smaller in size, hence faster to access. If there is a miss in L1, then data if fetch from L2 (if there), which is larger, and have relatively slower access time. If it again is a miss, then main memory is accessed etc.

Intel core i7 processors use THREE level of caching. This is a multicore processor with several i7 cores. Each core has its own L1 and L2 cache.

L1 cache is broken up into two halves, one for instruction and a second for data.

L2 is a combined cache for both instruction and data.

L3 cache is shared among all cores and is an inclusive cache, meaning that its stored data is present in lower level L1 or L2 cache of the cores.

Intel i7 Cache Characteristics

Characteristic	L1	L2	L3
Size	32 KB I/32 KB D	256 KB	2 MB per core
Associativity	4-way I/8-way D	8-way	16-way
Access latency	4 cycles, pipelined	10 cycles	35 cycles
Replacement scheme	Pseudo-LRU	Pseudo-LRU	Pseudo-LRU but with an ordered selection algorihtm

Here is the layout of the cache in an Intel i7 processor. Each of the four cores of this i7 variant has its own L1 and L2 caches.

The L1 cache is 32kB each for instruction and for data. While the instruction cache is 4-way associative, the data cache is 8-way associative. The latency of the L1 cache is 4 cycles and it is pipelined. The replacement policy is pseudo-LRU.

The L2 cache is combined data and instruction, and is 256kB in capacity and it is 8-way associative. It takes 10 cycles to access.

Finally the large L3 cache is common to all cores, It is 8MB in size, 160way associative and takes a relatively long 35 cycles to access.

Summary on Cache

- **What data is held in the cache?**
  - Recently used data (temporal locality)
  - Nearby data (spatial locality)
- **How is data found?**
  - Set is determined by address of data
  - Word within block also determined by address
  - In associative caches, data could be in one of several ways
- **What data is replaced?**
  - Least-recently used way in the set

Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

This slide summarises the content of this lecture on cache memory organization and design.