

CSE 4305
Computer Organization and Architecture

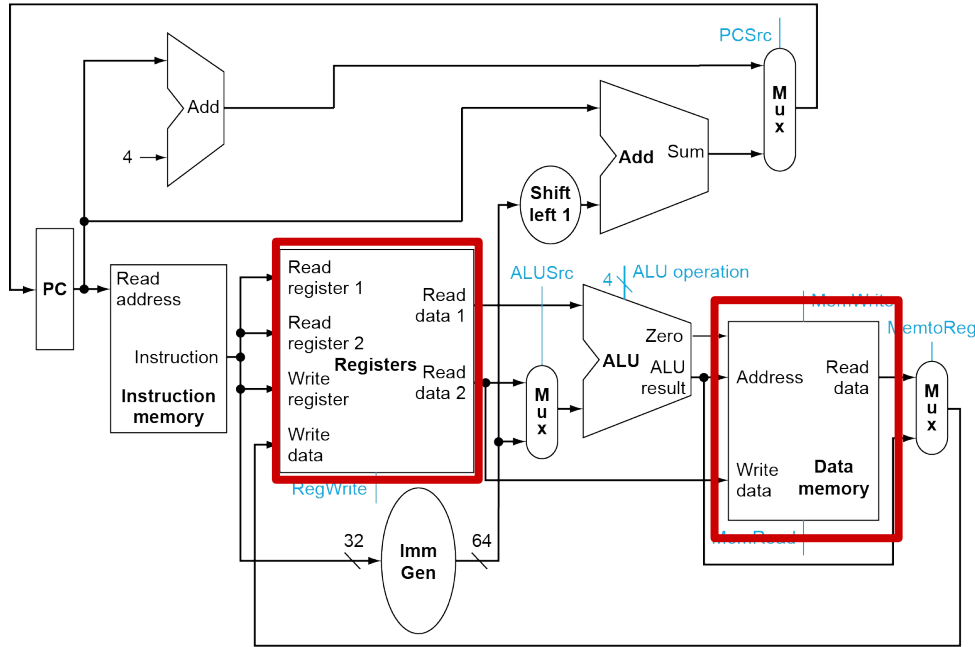
Lecture 6

Memory

Sabrina Islam
Lecturer, CSE, IUT
Contact: +8801832239897
E-mail: sabrinaislam22@iut-dhaka.edu

The big picture

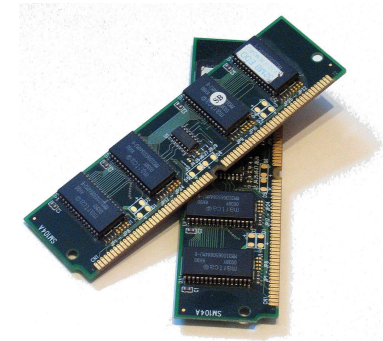
Where are we now?



add x9, x1, x2

ld|sd x3, 16(x5)

beq x3, x9, offset



Background

Recall one of the 8 great ideas:

- **Hierarchy of Memories**

Programmers want
memory to be



Unlimited
Fast



However, fast memory
technology is more
expensive per bit than
slower memory

Solution: Organize memory system into a hierarchy

- **store** always in **large memory**
- **access** always from **fast memory**



Memory Hierarchy

Hierarchy of Memories

- The closer to the CPU
 - faster and smaller
- The farther from the CPU
 - slower and larger

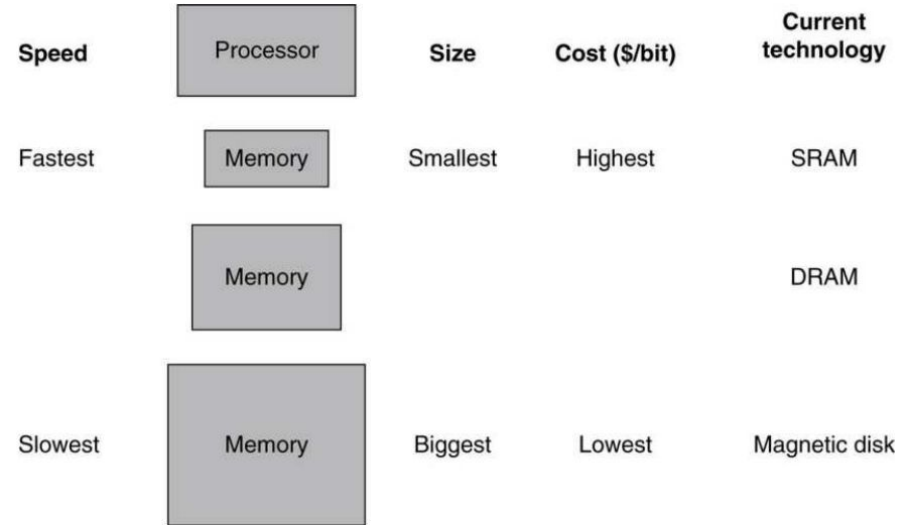


Fig: The basic structure of a memory hierarchy

Why Memory Hierarchy works?

The Principle of Locality

- Programs access a relatively small portion of their address space at any instant of time

There are two different types of locality

- **Temporal locality** (locality in time)
 - Items accessed recently are likely to be accessed again soon
 - e.g., instructions in a loop, induction variables
- **Spatial locality** (locality in space)
 - Items near those accessed recently are likely to be accessed soon
 - e.g., sequential instruction access, array data

Principle of Locality: Understanding with an analogy

Scenario: You need to write a term paper on a topic. You are in a library and gathering the necessary books and keeping them on your desk to get information from them relevant to your topic.



- Getting one book from the shelves, writing down the information and return it to the shelf.
- Need to search the whole library over and over again.
- The continuous back and forth takes a lot of time

Slow



- Collection of books relevant to the topic on the desk.
- High probability that most of the topics needed can be found in them.
- No need to return to the shelves and search whole library continuously.

Faster

Principle of Locality: Understanding with an analogy

If we need lots of books, we need to borrow from the library, put on our bookshelf and on our study desk.

- **Books are Data**
- **Study is the Program**
- **Desk/Bookshelf/Library are Memory Hierarchy**
- **You are the CPU**

In this scenario:

- **Temporal locality:** If you recently looked at a book for some information, you will probably need to look at it again soon.
- **Spatial locality:** If you find a book useful in a library shelf, it's likely that you'll find similar books that may help you nearby in that shelf.

Temporal locality in Data

Example: Sum each iteration

- The sum variable is accessed over and over again.
- When a variable is accessed for data store or retrieve over and over again, it is temporal locality.

```
sum = 0;  
for(i=0; i<n; i++)  
    sum += a[i];  
return sum;
```


Temporal locality in Program

Example: Cycle through loop repeatedly

- The **instructions** for this loop block will be executed again and again until it ends.
- So the processor will keep accessing the same location multiple times.

```
sum = 0;  
for(i=0; i<n; i++)  
    sum += a[i];  
return sum;
```

Spatial locality in Data

Example: Array elements in succession

- Some data are often accessed serially.
- When we take input for an array, the values are stored side by side. So they are accessed from nearby locations as well.

3	5	6	4	5	9
---	---	---	---	---	---

Spatial locality in Program

Example: Reference instructions in sequence

- The **instructions** are usually executed sequentially. If we execute an instruction at memory location i , then we will probably also execute the next instruction, at memory location $i+1$.
- Code fragments such as loops exhibit both **temporal** and **spatial** locality.

```
ld x10, 40(x1)
sub x11, x2, x3
add x12, x3, x4
ld x13, 48(x1)
add x14, x5, x6
```

Memory Hierarchy: Explained with Desk, Bookshelf and Library

You, as a processor study and read books of a topic (a program)	Speed	Processor	Size	Cost (\$/bit)	Current technology
Books that are used often are on your desk; fast access, small # of books	Fastest	Memory	Smallest	Highest	SRAM
Your bookshelf can hold more books, less often used than those on the desk. you need to stand up to grab a book and you often grab more than one books a time		Memory			DRAM
Library has more books, but you do not go often since it is far. Each time you go, you borrow a bag of books.	Slowest	Memory	Biggest	Lowest	Magnetic disk

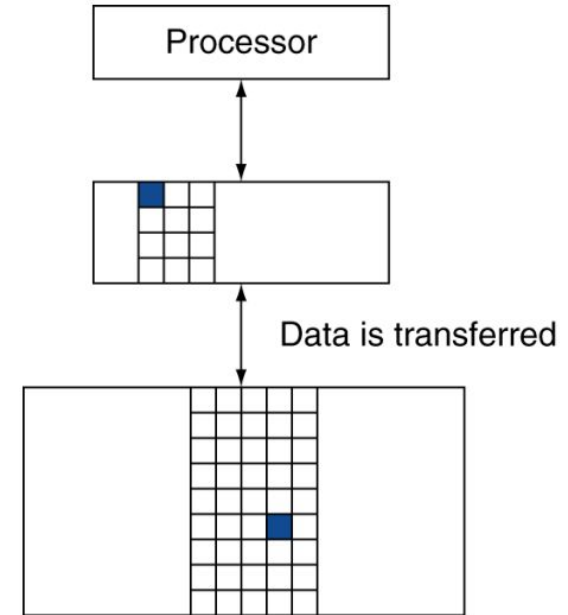
Taking Advantage of Locality

- We can build a memory hierarchy
- Store everything on **disk** (library)
- Copy recently accessed (and nearby) items from disk to smaller **DRAM memory** (bookshelf)
 - **Main memory**
- Copy more recently accessed (and nearby) items from DRAM to smaller SRAM memory (desk)
 - **Cache** memory attached to CPU

Speed	Processor	Size	Cost (\$/bit)	Current technology
Fastest	Memory	Smallest	Highest	SRAM
	Memory			DRAM
Slowest	Memory	Biggest	Lowest	Magnetic disk

Memory Hierarchy Levels

- Data are copied between only two adjacent levels at a time.
 - Upper level: the one closer to the processor.
- **Block (or line):** unit of copying, minimum unit of information
 - In our library analogy, a block of information is one book.
 - May be multiple words, just like we move multiple books a time.
- **Hit**
 - If the data requested appears in the upper level (found on desk)
 - **Hit ratio:** hits/accesses
- **Miss**
 - If the data requested is absent in the upper level. The lower level is then accessed for data. (Go to shelf)
 - **Time taken:** miss penalty
 - **Miss ratio:** misses/accesses = $1 - \text{hit ratio}$
 - Then accessed data supplied from upper level



Memory Hierarchy Levels

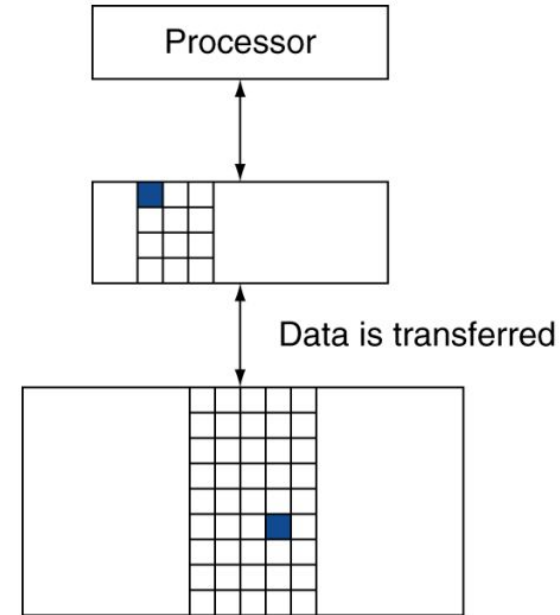
Hit rate or hit ratio is often used as a **measure of the performance** of the memory hierarchy.

Hit time:

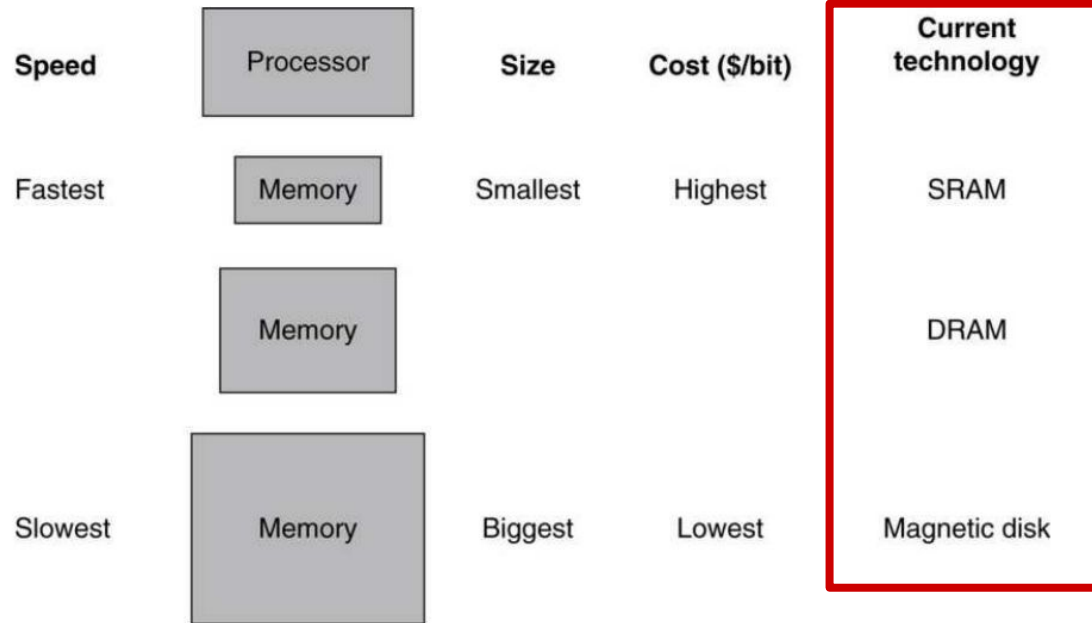
- The time to access the upper level of the memory hierarchy
- Including the time needed to determine whether the access is a hit or a miss.
- From the analogy, the time needed to look through the books on the desk.

Miss penalty:

- The time to replace a block in the upper level with the corresponding block from the lower level,
- Plus the time to deliver this block to the processor
- From the analogy, the time to get another book from the shelves and place it on the desk.



Memory Hierarchy Levels



Need to learn about these technologies!

Memory Technologies

Current technologies in memory hierarchy:

- SRAM Technology
- DRAM Technology
- Flash Memory
- Disk Memory

**Chapter 5, Section 5.2 from the book
provided in the classroom**

Cache basics

- The level of the memory hierarchy closest to the CPU.
- In our library example, the **desk** acted as a cache.
- It is a small amount of fast, expensive memory.
 - The cache goes between the processor and the slower, dynamic main memory.
 - It keeps a copy of the most frequently used data from the main memory.
- Memory access speed increases overall, because we've made the **common case faster**.
 - Reads and writes to the most frequently used addresses will be serviced by the cache.
 - We only need to access the slower main memory for less frequently used data.

How caches take advantage of temporal locality

- The first time the processor reads from an address in main memory, a copy of that data is also stored in the **cache**.
 - The next time that same address is read, we can use the copy of the data in the cache instead of accessing the slower dynamic memory.
 - So the first read is a little slower than before since it goes through both main memory and the cache, but **subsequent reads are much faster**.
- This takes advantage of **temporal locality**: commonly accessed data is stored in the faster cache memory.

How caches take advantage of spatial locality

- When the CPU reads location i from main memory, a copy of that data is placed in the cache.
- But instead of just copying the contents of location i , we can copy several values into the cache at once, such as the four bytes from locations i through $i + 3$.
 - If the CPU later does need to read from locations $i + 1$, $i + 2$ or $i + 3$, it can access that data from the cache and not the slower main memory.
 - For example, instead of reading just one array element at a time, the cache might actually be loading four array elements at once.
- Again, the initial load incurs a **performance penalty** (why?), but we're gambling on spatial locality and the chance that the CPU will need the extra data.

Cache hits and misses

- A **cache hit** occurs if the cache contains the data that we're looking for.
- A **cache miss** occurs if the cache does not contain the requested data. This is bad, since the CPU must then wait for the slower main memory.
- The **hit rate** is the percentage of memory accesses that are handled by the cache.
- The **miss rate** ($1 - \text{hit rate}$) is the percentage of accesses that must be handled by the slower main RAM.

A simple cache design

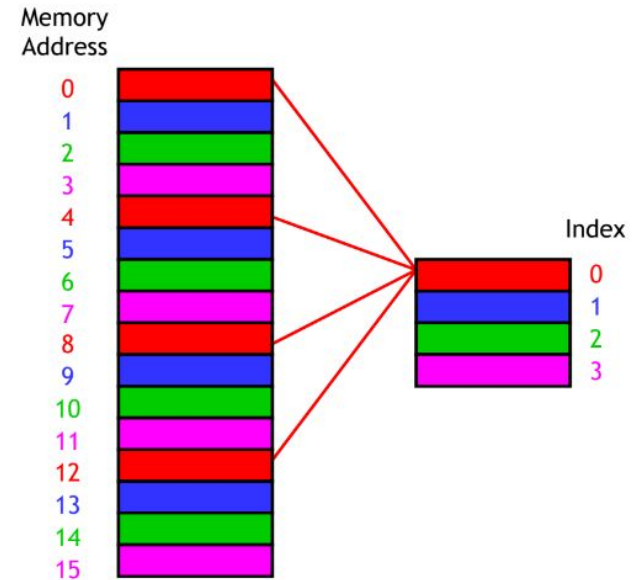
- Caches are divided into **blocks**, which may be of various sizes.
 - The number of blocks in a cache is usually a power of 2.
 - For now we'll assume that each block contains one byte.
- Here is an example cache with 8 blocks, each holding one byte.

Block index	8-bit data
000	
001	
010	
011	
100	
101	
110	
111	

Where should we put data in the cache?

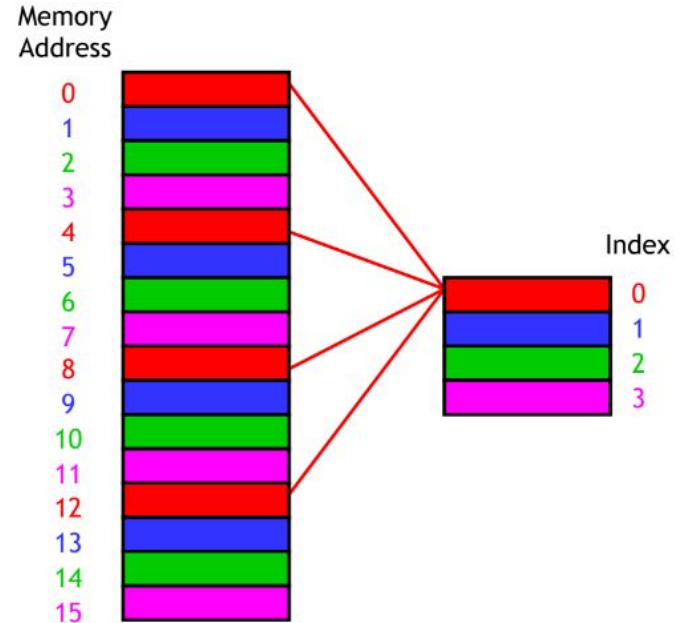
- A **direct-mapped cache** is the simplest approach
 - each main memory address maps to exactly one cache block.
- In the example, there is a 16-byte main memory and a 4-byte cache (four 1-byte blocks).
- Memory locations **0, 4, 8** and **12** all map to cache block **0**.
- Addresses **1, 5, 9** and **13** map to cache block **1**, etc.

How can we compute this mapping?



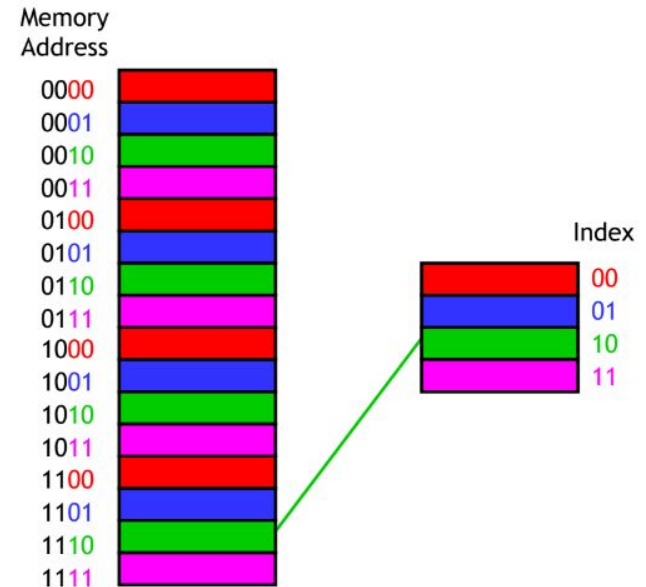
Cache mapping by divisions

- Using the mod (remainder) operator.
- If the cache contains 2^k blocks, then the data at memory address i would go to cache block index $i \bmod 2^k$
- In the example, address 14 would map to cache block 2.
 $14 \bmod 4 = 2$



Cache mapping by least-significant bits

- Another way is to look at the least significant k bits of the address. If the cache contains 2^k blocks.
- With our four-byte cache we would inspect the **two** least significant bits of our memory addresses.
- Address **14** (1110 in binary) maps to cache block **2** (10 in binary).
- $i \bmod 2^k$ = least k bits of a binary value

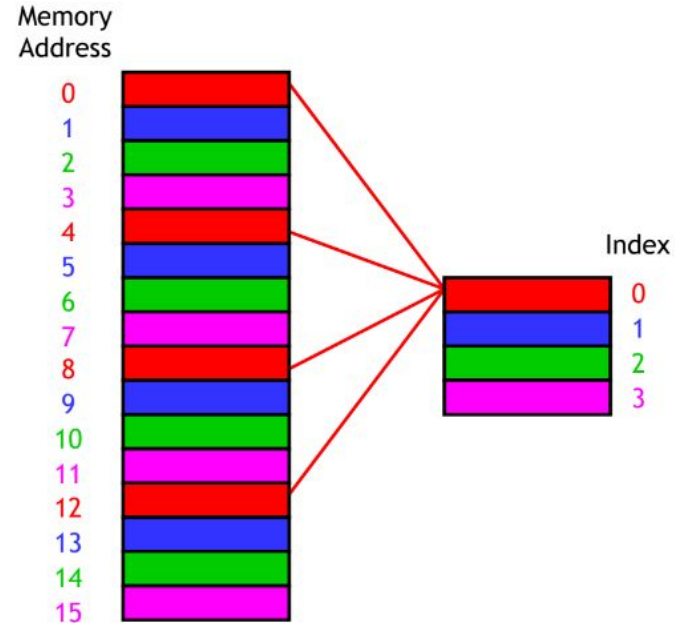


How can we find data in the cache?

- If we want to read memory address i , we can use the **mod trick** to determine which cache block would contain i .
- But other addresses might also map to the same cache block. For example, cache block **2** could contain data from addresses **2, 6, 10** or **14**.

How to distinguish between them?

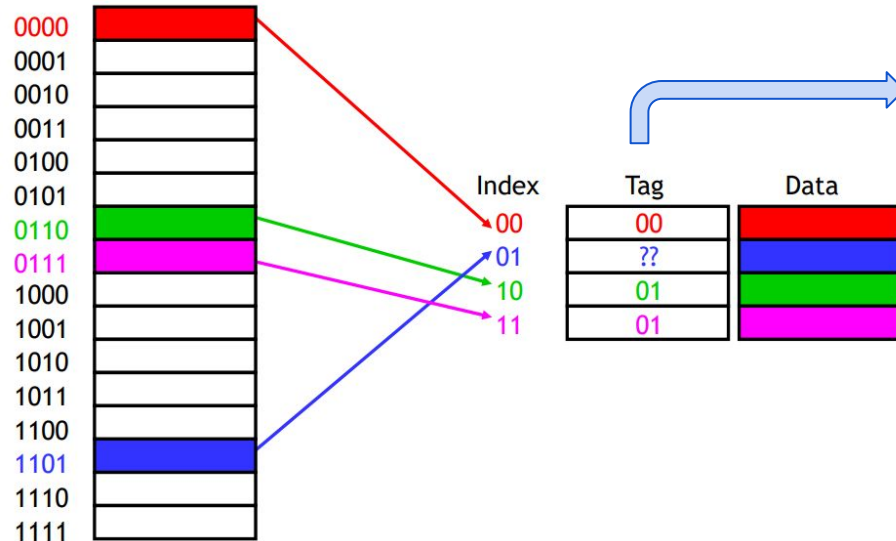
Adding Tags!



How can we find data in the cache?

Adding tags

- **Tag:** Necessary information to identify corresponding memory address.

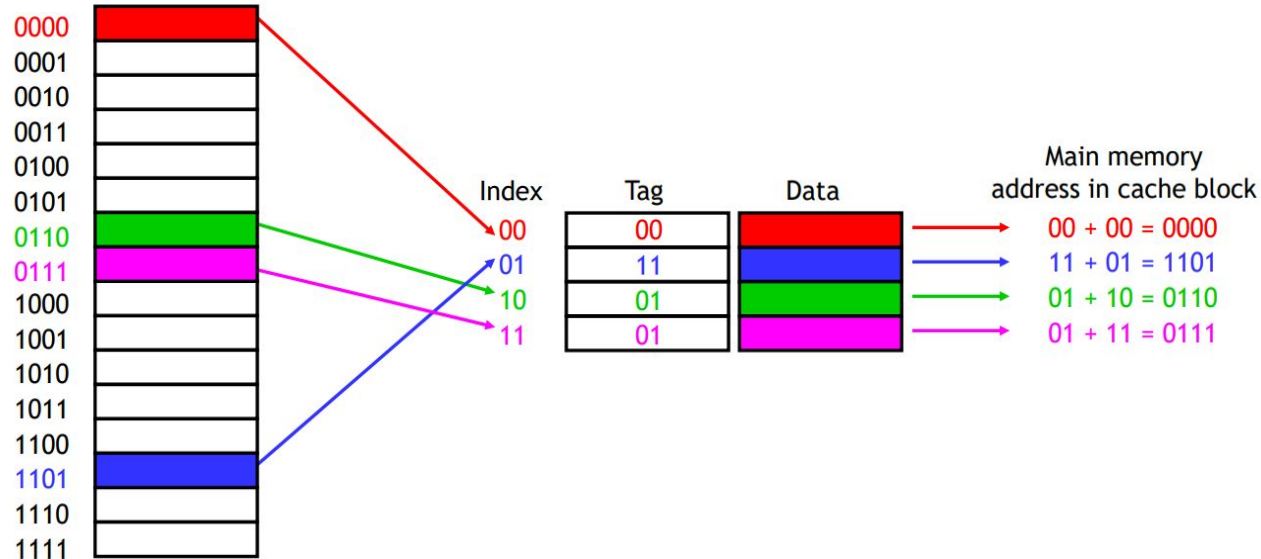


Supplies the **rest of the address bits** to let us **distinguish** between different memory locations that map to the same cache block.

How can we find data in the cache?

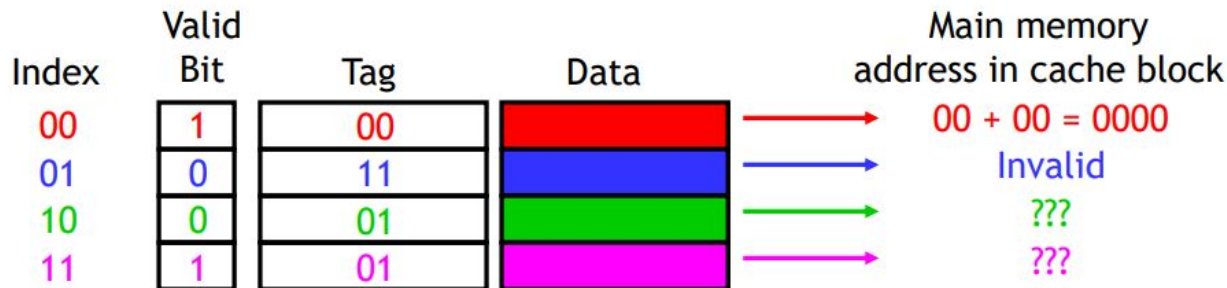
Adding tags

- We can identify which addresses of main memory are stored in the cache, by concatenating the cache block tags with the block indices.



The valid bit

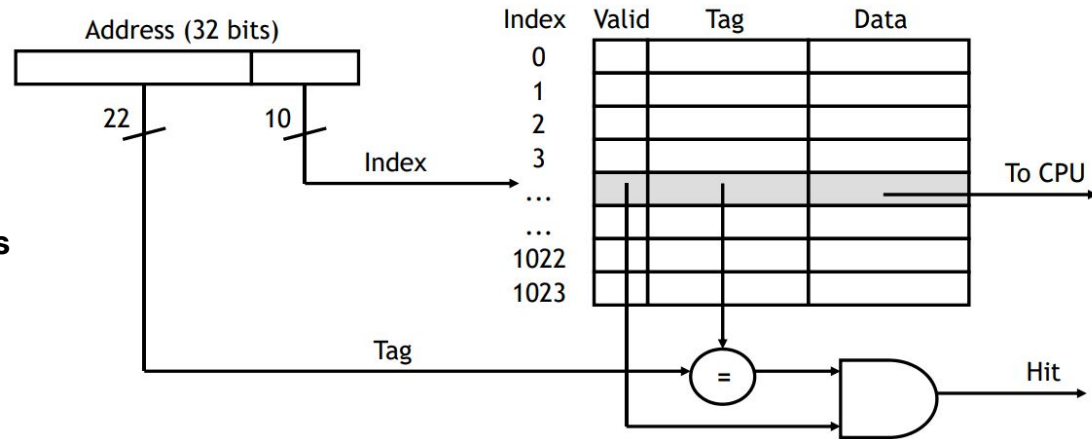
- Initially the cache blocks are empty.
- We need to ignore the values of empty cache blocks and their corresponding tags. How?
 - Adding valid bit**
- When the system is initialized, all the valid bits are set to 0.
- When data is loaded into a particular cache block, the corresponding valid bit is set to 1.



Cache hits

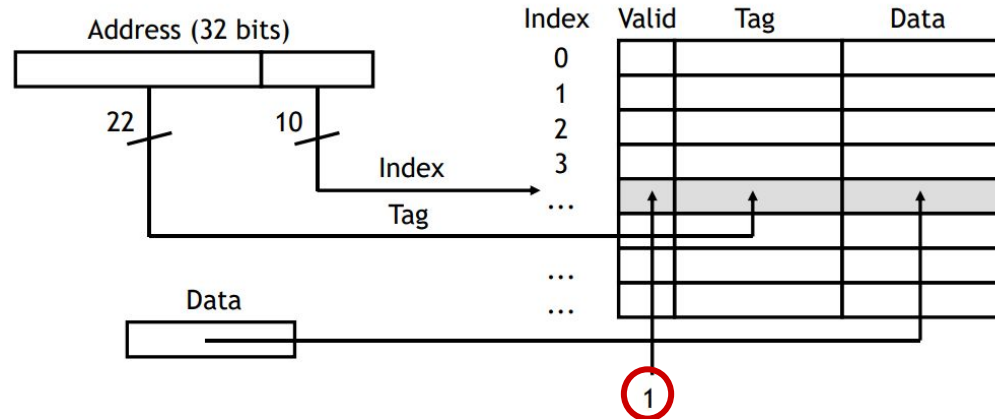
- When the CPU tries to read from memory, the address will be sent to a cache controller.
 - The **lowest k bits** of the address will index a block in the cache.
 - If the block is valid and the tag matches the upper **$(m - k)$ bits** of the m -bit address, then that data will be sent to the CPU.

A 32-bit memory address
and a 2^{10} byte cache.



Cache miss and loading a block into the cache

- After data is read from main memory, putting a copy of that data into the cache is straightforward.
 - The **lowest k bits** of the address specify a cache block.
 - The upper **$(m - k)$ address bits** are stored in the block's **tag field**.
 - The **data** from main memory is stored in the block's **data field**.
 - The **valid bit** is set to **1**.




What if the cache fills up?

- **What to do if we run out of space in our cache? Or, if we need to reuse a block for a different memory address?**
 - A miss causes a new block to be loaded into the cache, automatically **overwriting any previously stored data.**
 - This is a least recently used replacement policy, which assumes that older data is less likely to be requested than newer data.

Example

Address access serial



Decimal address of reference	Binary address of reference
22	10110 _{two}
26	11010 _{two}
22	10110 _{two}
26	11010 _{two}
16	10000 _{two}
3	00011 _{two}
16	10000 _{two}
18	10010 _{two}
16	10000 _{two}

Empty cache, 8 blocks, 1 word/block

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

Example

Cache updates

Decimal address of reference	Binary address of reference	Hit or miss in cache	Assigned cache block (where found or placed)
22	10110_{two}	miss	$(10110_{\text{two}} \bmod 8) = 110_{\text{two}}$
26	11010_{two}	miss	$(11010_{\text{two}} \bmod 8) = 010_{\text{two}}$
22	10110_{two}	hit	$(10110_{\text{two}} \bmod 8) = 110_{\text{two}}$
26	11010_{two}	hit	$(11010_{\text{two}} \bmod 8) = 010_{\text{two}}$
16	10000_{two}	miss	$(10000_{\text{two}} \bmod 8) = 000_{\text{two}}$
3	00011_{two}	miss	$(00011_{\text{two}} \bmod 8) = 011_{\text{two}}$
16	10000_{two}	hit	$(10000_{\text{two}} \bmod 8) = 000_{\text{two}}$
18	10010_{two}	miss	$(10010_{\text{two}} \bmod 8) = 010_{\text{two}}$
16	10000_{two}	hit	$(10000_{\text{two}} \bmod 8) = 000_{\text{two}}$

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Example

Cache updates

Decimal address of reference	Binary address of reference	Hit or miss in cache	Assigned cache block (where found or placed)
22	10110_{two}	miss	$(10110_{\text{two}} \bmod 8) = 110_{\text{two}}$
26	11010_{two}	miss	$(11010_{\text{two}} \bmod 8) = 010_{\text{two}}$
22	10110_{two}	hit	$(10110_{\text{two}} \bmod 8) = 110_{\text{two}}$
26	11010_{two}	hit	$(11010_{\text{two}} \bmod 8) = 010_{\text{two}}$
16	10000_{two}	miss	$(10000_{\text{two}} \bmod 8) = 000_{\text{two}}$
3	00011_{two}	miss	$(00011_{\text{two}} \bmod 8) = 011_{\text{two}}$
16	10000_{two}	hit	$(10000_{\text{two}} \bmod 8) = 000_{\text{two}}$
18	10010_{two}	miss	$(10010_{\text{two}} \bmod 8) = 010_{\text{two}}$
16	10000_{two}	hit	$(10000_{\text{two}} \bmod 8) = 000_{\text{two}}$

Index	V	Tag	Data
000	N		
001	N		
010	Y	11	Mem[11010]
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Example

Cache updates

Decimal address of reference	Binary address of reference	Hit or miss in cache	Assigned cache block (where found or placed)
22	10110_{two}	miss	$(10110_{\text{two}} \bmod 8) = 110_{\text{two}}$
26	11010_{two}	miss	$(11010_{\text{two}} \bmod 8) = 010_{\text{two}}$
22	10110_{two}	hit	$(10110_{\text{two}} \bmod 8) = 110_{\text{two}}$
26	11010_{two}	hit	$(11010_{\text{two}} \bmod 8) = 010_{\text{two}}$
16	10000_{two}	miss	$(10000_{\text{two}} \bmod 8) = 000_{\text{two}}$
3	00011_{two}	miss	$(00011_{\text{two}} \bmod 8) = 011_{\text{two}}$
16	10000_{two}	hit	$(10000_{\text{two}} \bmod 8) = 000_{\text{two}}$
18	10010_{two}	miss	$(10010_{\text{two}} \bmod 8) = 010_{\text{two}}$
16	10000_{two}	hit	$(10000_{\text{two}} \bmod 8) = 000_{\text{two}}$

Index	V	Tag	Data
000	N		
001	N		
010	Y	11	Mem[11010]
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Example

Cache updates

Decimal address of reference	Binary address of reference	Hit or miss in cache	Assigned cache block (where found or placed)
22	10110_{two}	miss	$(10110_{\text{two}} \bmod 8) = 110_{\text{two}}$
26	11010_{two}	miss	$(11010_{\text{two}} \bmod 8) = 010_{\text{two}}$
22	10110_{two}	hit	$(10110_{\text{two}} \bmod 8) = 110_{\text{two}}$
26	11010_{two}	hit	$(11010_{\text{two}} \bmod 8) = 010_{\text{two}}$
16	10000_{two}	miss	$(10000_{\text{two}} \bmod 8) = 000_{\text{two}}$
3	00011_{two}	miss	$(00011_{\text{two}} \bmod 8) = 011_{\text{two}}$
16	10000_{two}	hit	$(10000_{\text{two}} \bmod 8) = 000_{\text{two}}$
18	10010_{two}	miss	$(10010_{\text{two}} \bmod 8) = 010_{\text{two}}$
16	10000_{two}	hit	$(10000_{\text{two}} \bmod 8) = 000_{\text{two}}$

Index	V	Tag	Data
000	N		
001	N		
010	Y	11	Mem[11010]
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Example

Cache updates

Decimal address of reference	Binary address of reference	Hit or miss in cache	Assigned cache block (where found or placed)
22	10110_{two}	miss	$(10110_{\text{two}} \bmod 8) = 110_{\text{two}}$
26	11010_{two}	miss	$(11010_{\text{two}} \bmod 8) = 010_{\text{two}}$
22	10110_{two}	hit	$(10110_{\text{two}} \bmod 8) = 110_{\text{two}}$
26	11010_{two}	hit	$(11010_{\text{two}} \bmod 8) = 010_{\text{two}}$
16	10000_{two}	miss	$(10000_{\text{two}} \bmod 8) = 000_{\text{two}}$
3	00011_{two}	miss	$(00011_{\text{two}} \bmod 8) = 011_{\text{two}}$
16	10000_{two}	hit	$(10000_{\text{two}} \bmod 8) = 000_{\text{two}}$
18	10010_{two}	miss	$(10010_{\text{two}} \bmod 8) = 010_{\text{two}}$
16	10000_{two}	hit	$(10000_{\text{two}} \bmod 8) = 000_{\text{two}}$

Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	Y	11	Mem[11010]
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Example

Cache updates

Decimal address of reference	Binary address of reference	Hit or miss in cache	Assigned cache block (where found or placed)
22	10110_{two}	miss	$(10110_{\text{two}} \bmod 8) = 110_{\text{two}}$
26	11010_{two}	miss	$(11010_{\text{two}} \bmod 8) = 010_{\text{two}}$
22	10110_{two}	hit	$(10110_{\text{two}} \bmod 8) = 110_{\text{two}}$
26	11010_{two}	hit	$(11010_{\text{two}} \bmod 8) = 010_{\text{two}}$
16	10000_{two}	miss	$(10000_{\text{two}} \bmod 8) = 000_{\text{two}}$
3	00011_{two}	miss	$(00011_{\text{two}} \bmod 8) = 011_{\text{two}}$
16	10000_{two}	hit	$(10000_{\text{two}} \bmod 8) = 000_{\text{two}}$
18	10010_{two}	miss	$(10010_{\text{two}} \bmod 8) = 010_{\text{two}}$
16	10000_{two}	hit	$(10000_{\text{two}} \bmod 8) = 000_{\text{two}}$

Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	Y	11	Mem[11010]
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Example

Cache updates

Decimal address of reference	Binary address of reference	Hit or miss in cache	Assigned cache block (where found or placed)
22	10110_{two}	miss	$(10110_{\text{two}} \bmod 8) = 110_{\text{two}}$
26	11010_{two}	miss	$(11010_{\text{two}} \bmod 8) = 010_{\text{two}}$
22	10110_{two}	hit	$(10110_{\text{two}} \bmod 8) = 110_{\text{two}}$
26	11010_{two}	hit	$(11010_{\text{two}} \bmod 8) = 010_{\text{two}}$
16	10000_{two}	miss	$(10000_{\text{two}} \bmod 8) = 000_{\text{two}}$
3	00011_{two}	miss	$(00011_{\text{two}} \bmod 8) = 011_{\text{two}}$
16	10000_{two}	hit	$(10000_{\text{two}} \bmod 8) = 000_{\text{two}}$
18	10010_{two}	miss	$(10010_{\text{two}} \bmod 8) = 010_{\text{two}}$
16	10000_{two}	hit	$(10000_{\text{two}} \bmod 8) = 000_{\text{two}}$

Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	Y	11	Mem[11010]
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Example

Decimal address of reference	Binary address of reference	Hit or miss in cache	Assigned cache block (where found or placed)
22	10110_{two}	miss	$(10110_{\text{two}} \bmod 8) = 110_{\text{two}}$
26	11010_{two}	miss	$(11010_{\text{two}} \bmod 8) = 010_{\text{two}}$
22	10110_{two}	hit	$(10110_{\text{two}} \bmod 8) = 110_{\text{two}}$
26	11010_{two}	hit	$(11010_{\text{two}} \bmod 8) = 010_{\text{two}}$
16	10000_{two}	miss	$(10000_{\text{two}} \bmod 8) = 000_{\text{two}}$
3	00011_{two}	miss	$(00011_{\text{two}} \bmod 8) = 011_{\text{two}}$
16	10000_{two}	hit	$(10000_{\text{two}} \bmod 8) = 000_{\text{two}}$
18	10010_{two}	miss	$(10010_{\text{two}} \bmod 8) = 010_{\text{two}}$
16	10000_{two}	hit	$(10000_{\text{two}} \bmod 8) = 000_{\text{two}}$

Cache updates

Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	Y	10	Mem[10010]
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Value of address 26 is overwritten

Example

Cache updates

Decimal address of reference	Binary address of reference	Hit or miss in cache	Assigned cache block (where found or placed)
22	10110_{two}	miss	$(10110_{\text{two}} \bmod 8) = 110_{\text{two}}$
26	11010_{two}	miss	$(11010_{\text{two}} \bmod 8) = 010_{\text{two}}$
22	10110_{two}	hit	$(10110_{\text{two}} \bmod 8) = 110_{\text{two}}$
26	11010_{two}	hit	$(11010_{\text{two}} \bmod 8) = 010_{\text{two}}$
16	10000_{two}	miss	$(10000_{\text{two}} \bmod 8) = 000_{\text{two}}$
3	00011_{two}	miss	$(00011_{\text{two}} \bmod 8) = 011_{\text{two}}$
16	10000_{two}	hit	$(10000_{\text{two}} \bmod 8) = 000_{\text{two}}$
18	10010_{two}	miss	$(10010_{\text{two}} \bmod 8) = 010_{\text{two}}$
16	10000_{two}	hit	$(10000_{\text{two}} \bmod 8) = 000_{\text{two}}$

Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	Y	10	Mem[10010]
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Reference:

1. Computer Architecture and Design RISC-V Edition, by David A Patterson and John L. Hennessy.
2. ITSC 3181, Spring 2022
3. CSE 378, University of Washington

Thank You!