Prefetching and Writing to Caches

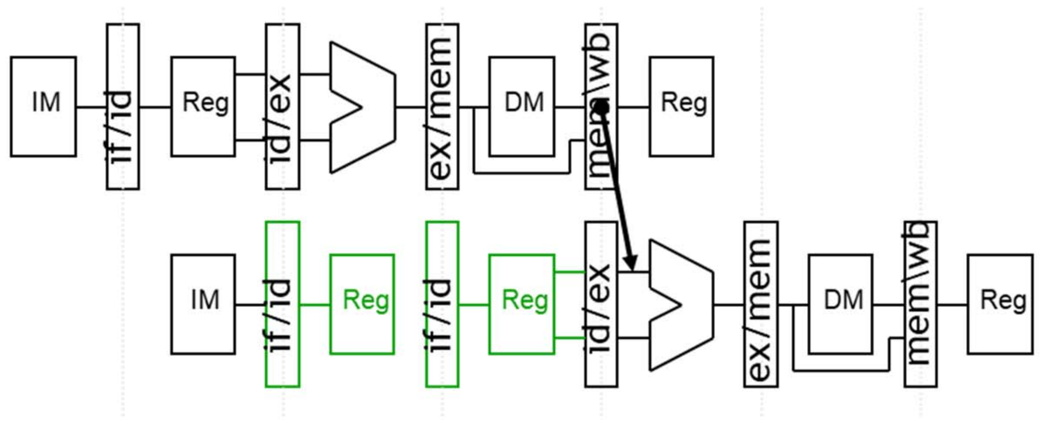
Today’s Lecture

- We’ll also see how to mitigate cache misses through pre-fetching.
- Writing to caches
  - Minimize accesses to main memory
  - keeping memory consistent & write-allocation.

How do we avoid stalls from cache misses?

lw $2, 20($3)

and $12, $2, $5



Hit: Stall 1 cycle

Miss: Stall many (~16-200) cycles

Non-blocking caches remember cache misses and the dependencies they affect

- This cache design stalls only when
  - Data is needed
  - Or too many misses outstanding

- Programmer can exploit by “hoist”ing loads up from their uses, but…
  - Uses up a register
  - For potentially many cycles (~100 to memory)
  - Might be guessing what will be accessed.

```
lw    $t0, 64($a0)
…
…
…
…
add   $v0, $t0, $t1
```

Software prefetching lets the programmer predict cache misses and fill cache in advance

- Code prefetch as a load without a destination register
  - e.g., on SPIM, `lw   $0, 64($a0)` *# write to the zero register.*
- Prefetches are hints to the processor:
  - "I think I might use cache block containing this address"
  - Hardware will try to move the block into the cache.
  - But, hardware can ignore (if busy)

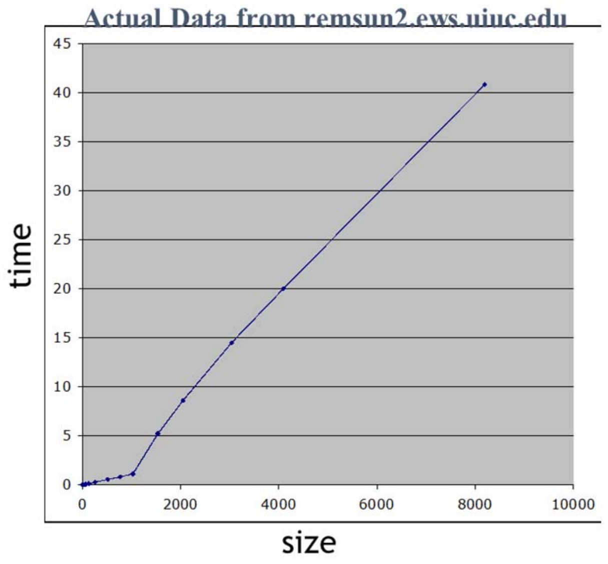
```
for (int i = 0 ; i < LARGE ; i ++) {
 prefetch A[i+16];   // prefetch 16 iterations ahead.
 computation A[i];
}
```

Prefetching may improve performance by removing or reducing stalls from cache misses

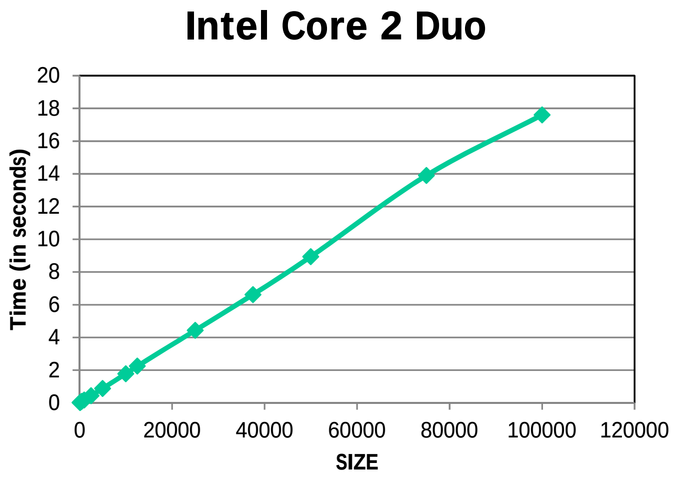
```
int array[SIZE];
int A = 0;

for (int i = 0 ; i < 200000 ; ++ i) {
  for (int j = 0 ; j < SIZE ; ++ j) {
      A += array[j];
  }
}
```

Without Pre-fetching



With Pre-fetching

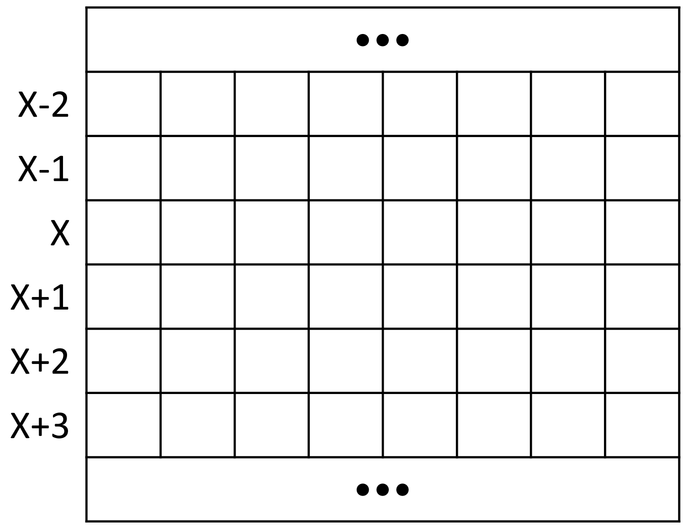


Hardware can prefetch by tracking patterns in cache hits and misses

- Inner loop has simple access streams
  - A, A+4, A+8, A+12, …
- Example: If we get a pair of sequential misses (blocks X, X+1), predict a stream
  - Fetch the next two blocks (X+2, X+3)
  - Fetch more blocks if predictions come true
- Can learn strides as well (X, X+16, X+32, …) and (X, X-1, X-2, …)

```
int array[SIZE];
int A = 0;

for (int i = 0 ; i < 200000 ; ++ i) {
   for (int j = 0 ; j < SIZE ; ++ j) {
        A += array[j];
   }
}
```



Use the program counter to track and prefetch multiple streams of addresses

```
for (int i=0 ; i < LARGE ; i ++) {
     C[i] = A[i] + B[i];
}
```

SW LW LW

A, B C A+4 B+4 C+4

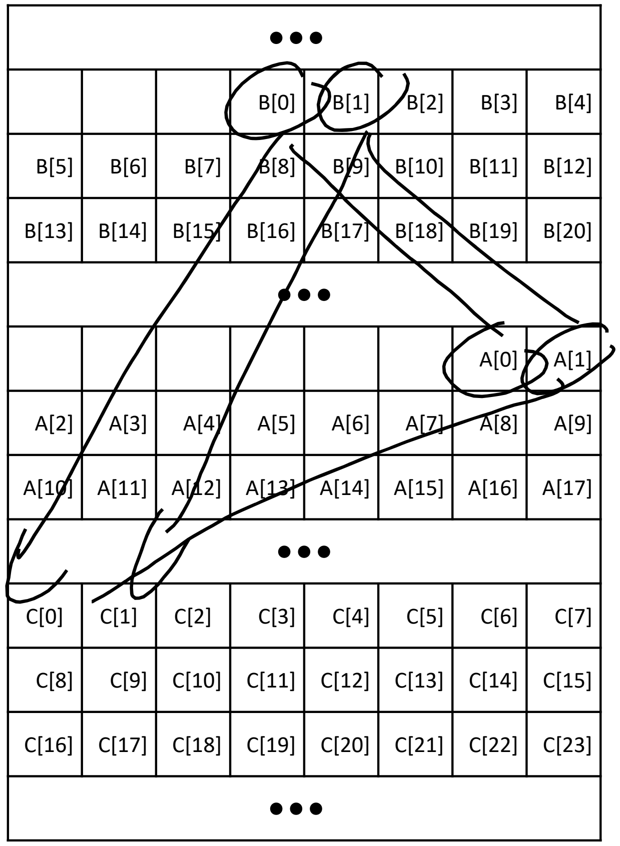
- 3 streams might confuse naïve prefetcher.

PC LW A, A+4, A+8

PC LW B, B+4, B+8

- A, B, and C accessed by different instructions

PC SW C, C+4, ...

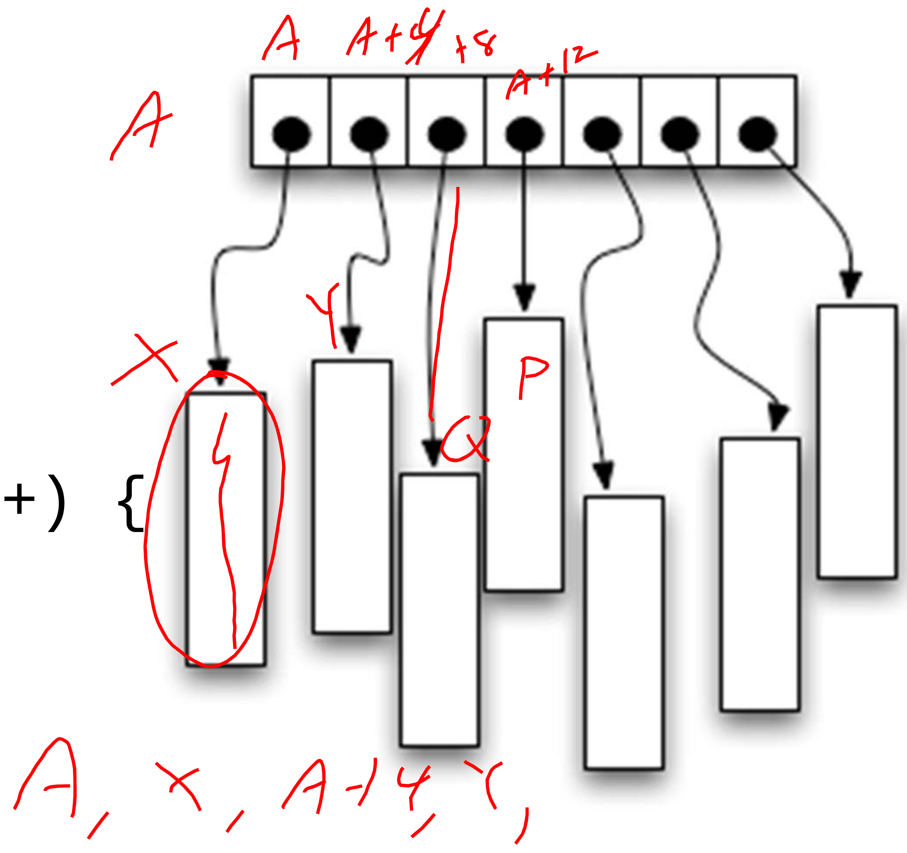


•••
B[0] B[1] B[2] B[3] B[4]
B[5] B[6] B[7] B[8] B[9] B[10] B[11] B[12]
B[13] B[14] B[15] B[16] B[17] B[18] B[19] B[20]
•••
A[0] A[1]
A[2] A[3] A[4] A[5] A[6] A[7] A[8] A[9]
A[10] A[11] A[12] A[13] A[14] A[15] A[16] A[17]
•••
C[0] C[1] C[2] C[3] C[4] C[5] C[6] C[7]
C[8] C[9] C[10] C[11] C[12] C[13] C[14] C[15]
C[16] C[17] C[18] C[19] C[20] C[21] C[22] C[23]
•••

Software prefetching is primarily used for non-strided memory access patterns

- Example: linked data structures:
  - Lists
  - Arrays of pointers

```
element_t *A[SIZE];
for (i = 0; i < SIZE ; i ++) {
   process(A[i]);
}
```



Four important questions

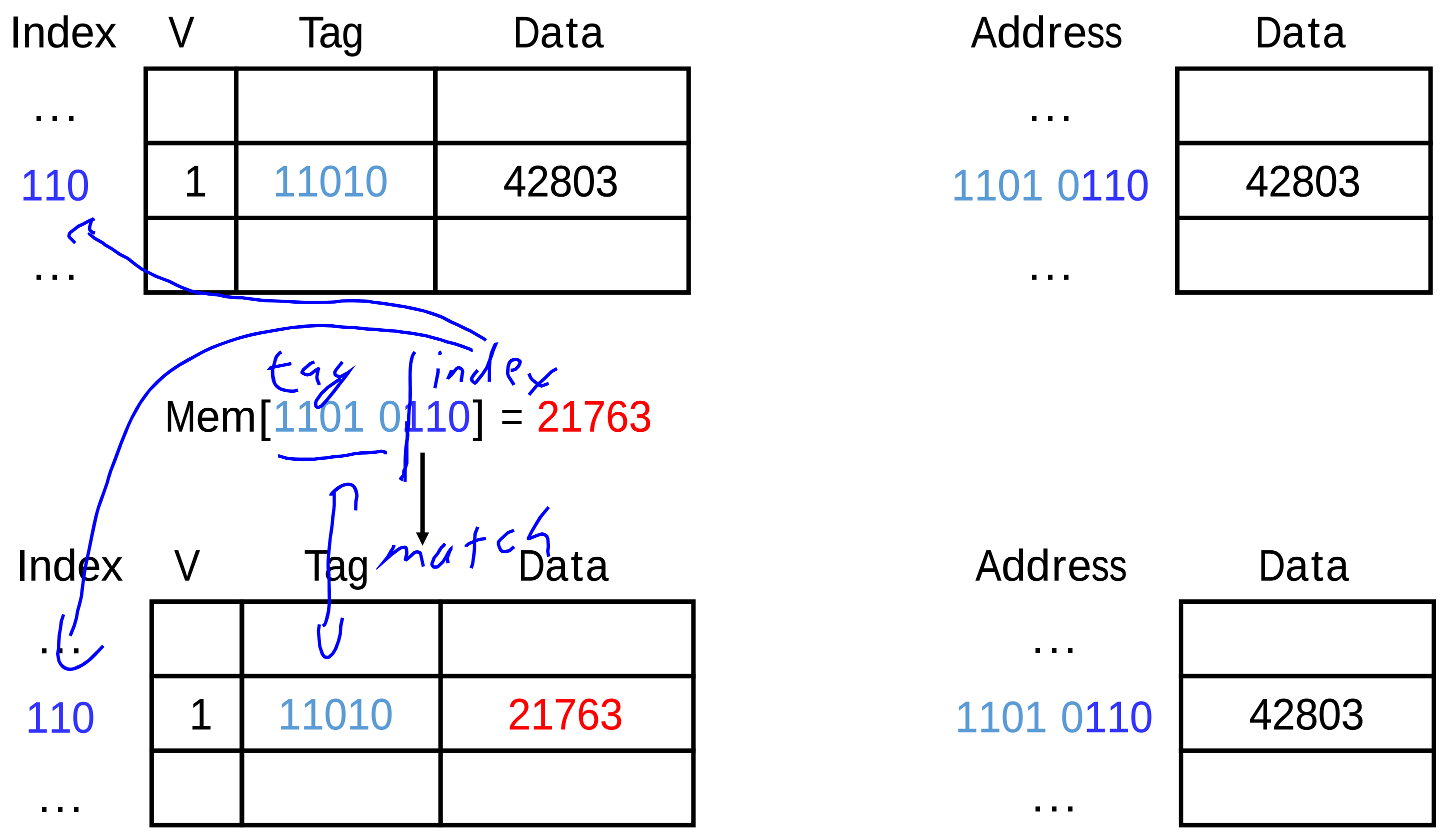
1. When we copy a block of data from main memory to the cache, where exactly should we put it?

2. How can we tell if a word is already in the cache, or if it has to be fetched from main memory first?

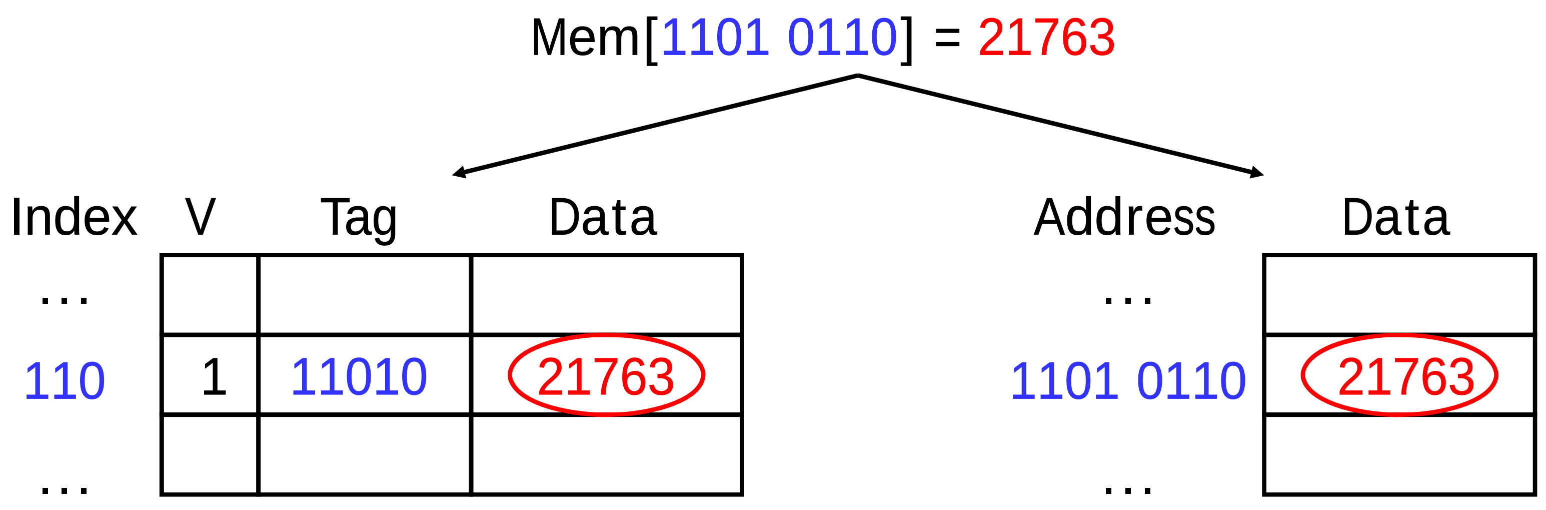
3. Eventually, the small cache memory might fill up. To load a new block from main RAM, we'd have to replace one of the existing blocks in the cache... which one?

4. How can *write* operations be handled by the memory system?

Writing to the cache can cause the cache to be inconsistent with main memory



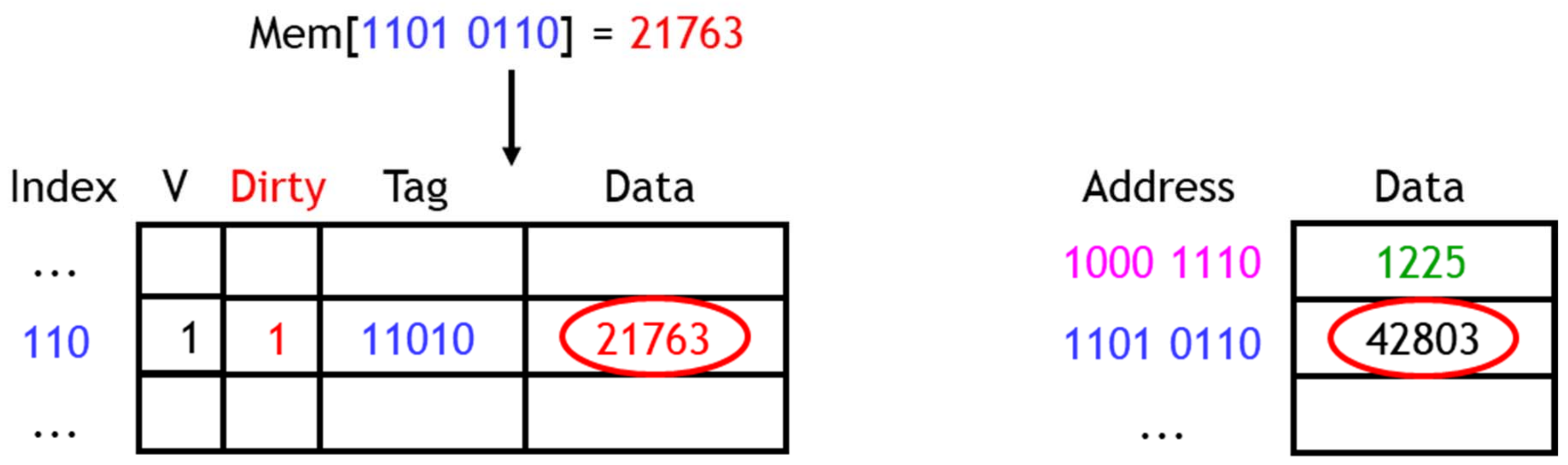
Write-through cache maintains consistency by writing to both main memory and the cache



- Why is this not so good?

Write-back cache updates main memory only when a cache block needs to be replaced

- Use a dirty bit to track where the cache is inconsistent with memory



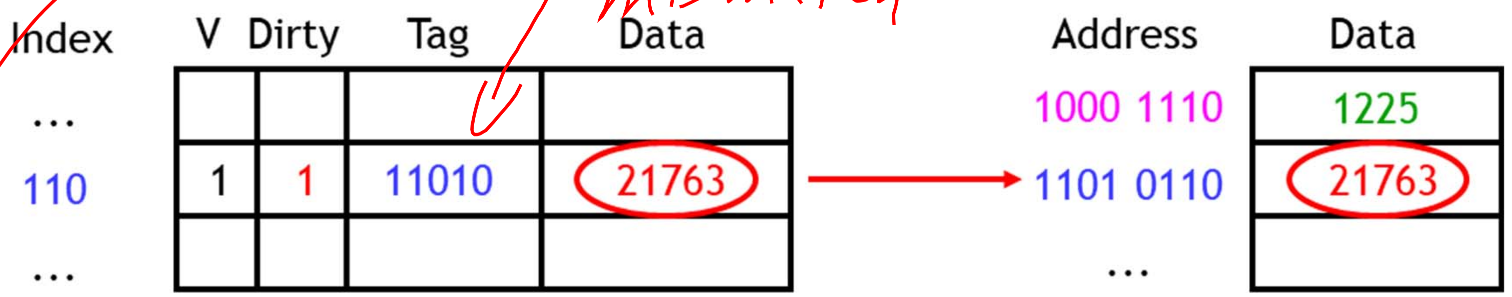
- Okay, since loads read from the cache, not main memory

Write back to main memory when replacing a dirty cache block

Example: Load from Mem[$\overset{\text{tag}}{1000\ 1}\overset{\text{index}}{110}$], collides with current data

1) Dirty bit is 1 -> Update main memory

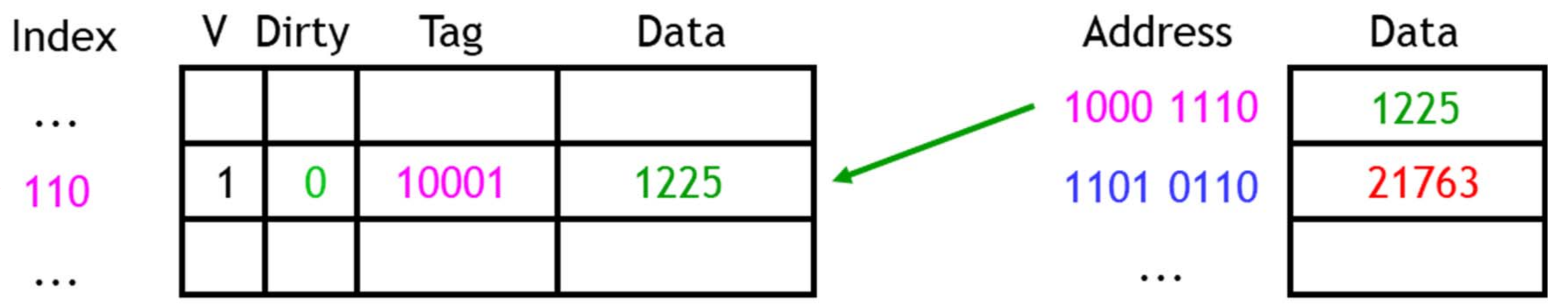
mismatch



Index	V	Dirty	Tag	Data
…				
110	1	1	11010	21763
…				

Address	Data
1000 1110	1225
1101 0110	21763
…	

2) Load new data -> new data is "clean"



Index	V	Dirty	Tag	Data
…				
110	1	0	10001	1225
…				

Address	Data
1000 1110	1225
1101 0110	21763
…	

A write miss occurs when we try to write to an address that is not already in the cache

Example: Store 21763 into Mem[1101 0110]

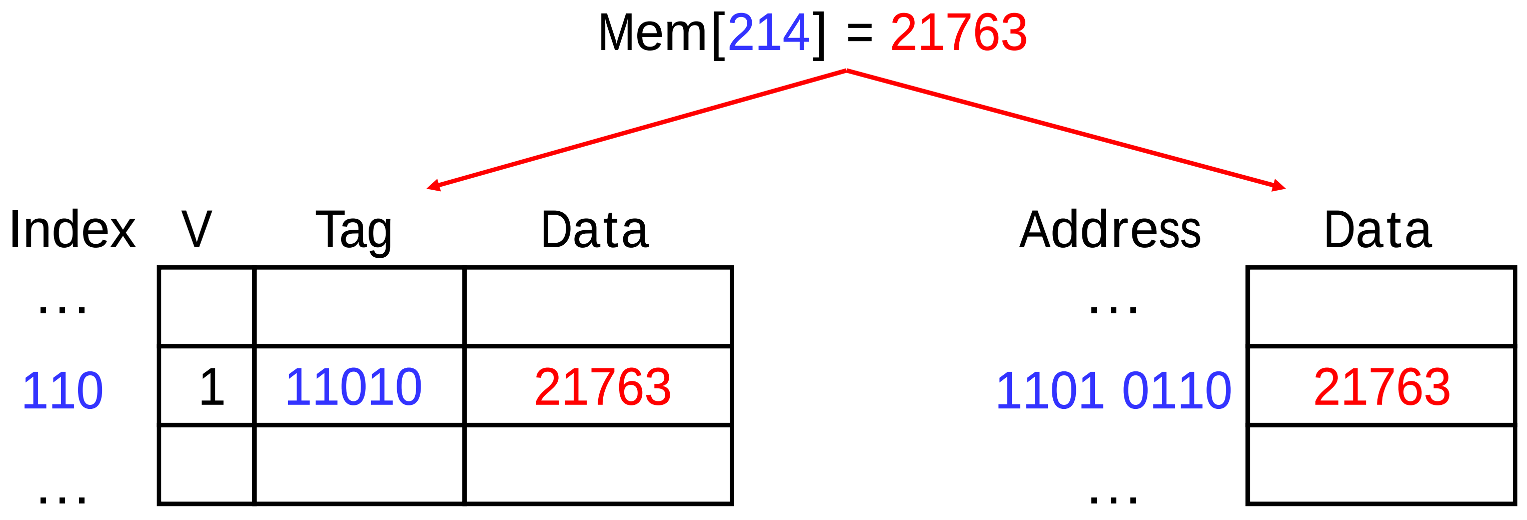
tag index

Index	V	Tag	Data
…			
110	1	00010	123456
…			

Address	Data
…	
1101 0110	6378
…	

- When we update Mem[1101 0110], should we *also* load it into the cache?

Allocate on write strategy loads new cache block on write misses

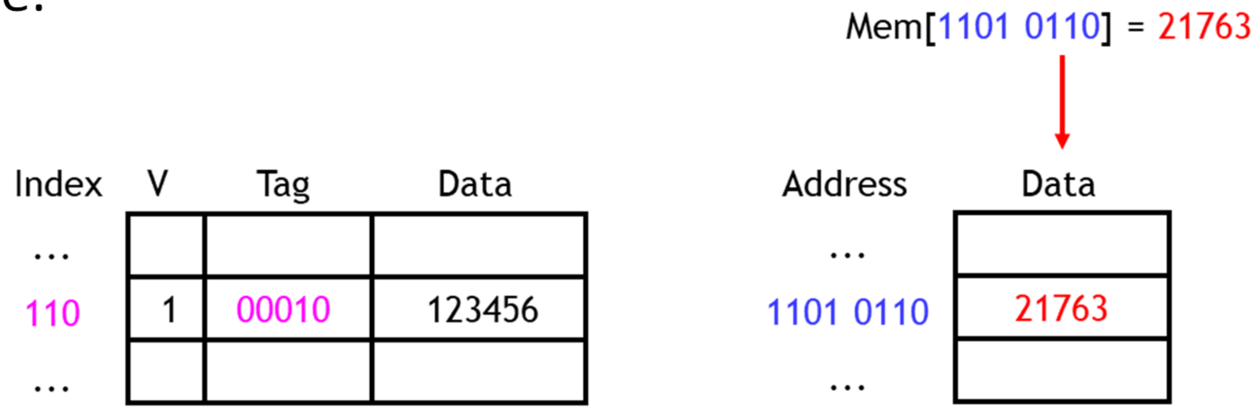


- If that data is needed again soon, it will be available in the cache
- This is the baseline behavior of processors.

Use non-temporal stores (write-around/write-no-allocate) if stored data will not be used soon

```
for (int i = 0; i < LARGE; i++) {
    a[i] = i;
}
```

- No temporal locality
- write around policy, writes directly to main memory *without* affecting the cache.



Summary

- Hardware prefetching handles most streams and strides.
- Software prefetching is useful for linked data structures
  - Must be added by programmer (or very smart compiler)

- Writing to a cache poses a couple of interesting issues.
  - Write-through and write-back policies keep the cache consistent with main memory in different ways for write hits.
  - Write-around and allocate-on-write are two strategies to handle write misses, differing in whether updated data is loaded into the cache.