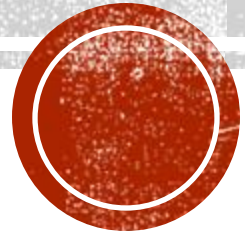
ADVANCED CACHE OPTIMIZATIONS

Prof Debdeep Mukhopadhyay
Computer Sc &. Engg,
Indian Institute of Technology Kharagpur



CACHE PERFORMANCE

- Average Memory Access Time (AMAT)=HIT TIME + MISS RATE x MISS PENALTY
  - Reduce Hit Time
  - Reduce Miss Rate
  - Reduce Miss Penalty

REDUCE HIT TIME

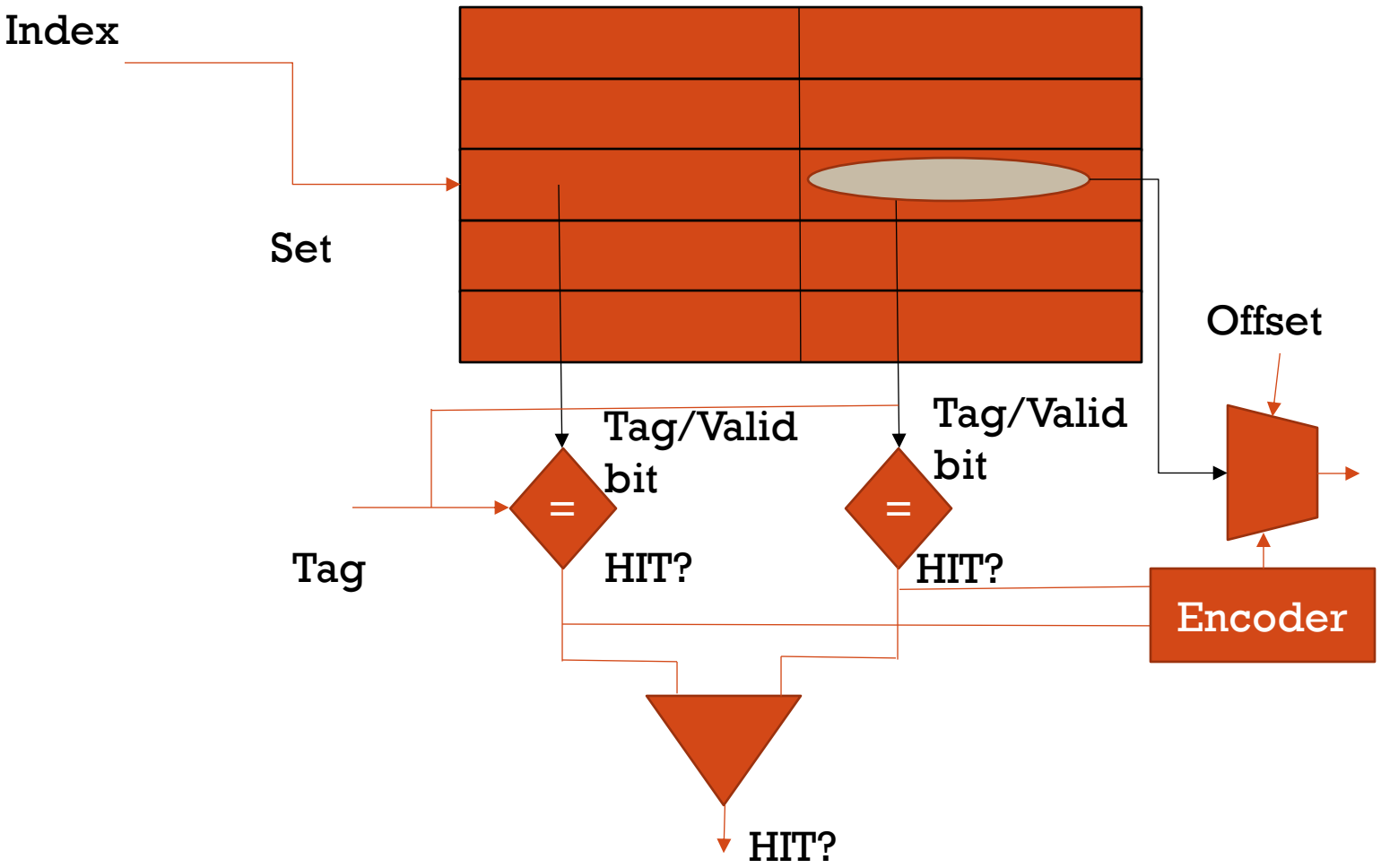
- Reduce Cache Size (But bad for Miss Rate)
- Reduce Cache Associativity (Bad for Miss Rate)
- Overlap Cache Hit with Another Hit
- Overlap Cache Hit with TLB Hit
- Optimize Lookup for Common Case
- Maintain Replacement State more efficient

PIPELINED CACHE

- Basic Idea is to overlap one hit with another.
  - Introduce pipelining in cache
- Consider an access to the cache in cycle N, results in Hit.
- If another access comes in cycle N+1, and also should result in Hit.
- But if the access to the cache takes multiple cycles, then the second access has to wait.
- Thus,

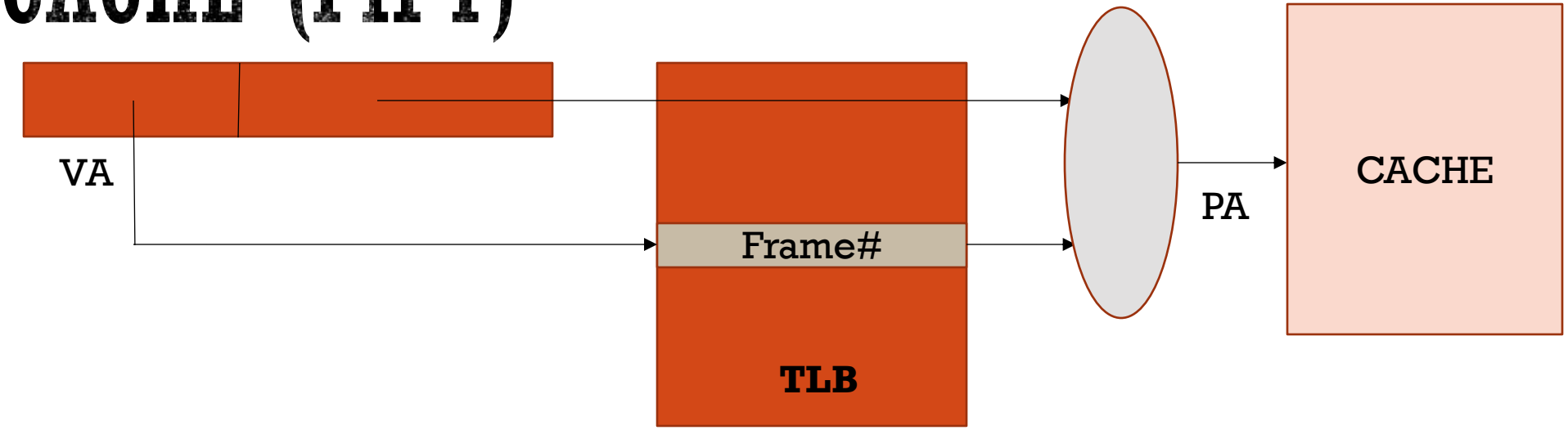
Hit Time (for second access)=Actual Hit Time + Wait Time

PIPELINING THE CACHE



- Index→Set
- Reading out the tags and valid bits to know if the access is a hit
- Combining the respective hits to determine whether we have an overall hit
- Using the offset to select the corresponding byte.
- Usually, the access is divided into stages.
- The stages can be forming the different cycles in the pipelined cache.
- An L1 cache is usually pipelined into 3 stages.

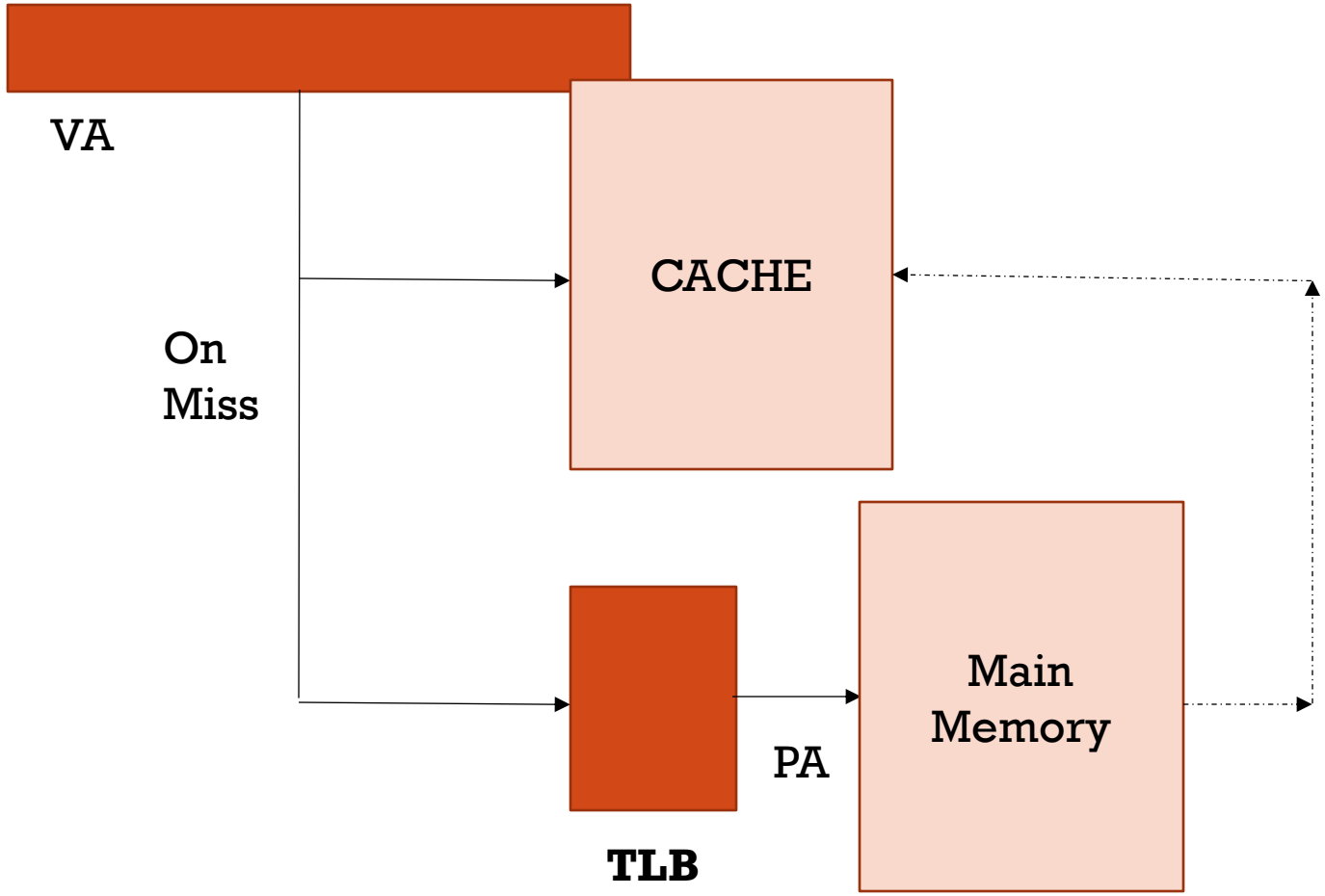
PHYSICALLY INDEXED PHYSICALLY TAGGED CACHE (PIPT)



- Hit time is affected because we have to access the TLB before the cache.
- Assuming the TLB and cache each takes 1 clock cycle, we need 2 cycles to get the data.

A cache that is accessed with the Physical Address (PA) is called Physically Accessed Cache or Physically Indexed Physically Tagged (PIPT) cache.

VIRTUALLY ACCESSED CACHE



On Miss a VA→PA mapping is obtained to access the main memory and update the cache.

Hit Time is same as that of Cache Hit Time

On Cache Hit, it is expected there is no VA to PA translation.

PROBLEMS WITH VIRTUALLY ACCESSED CACHE: PROBLEM 1

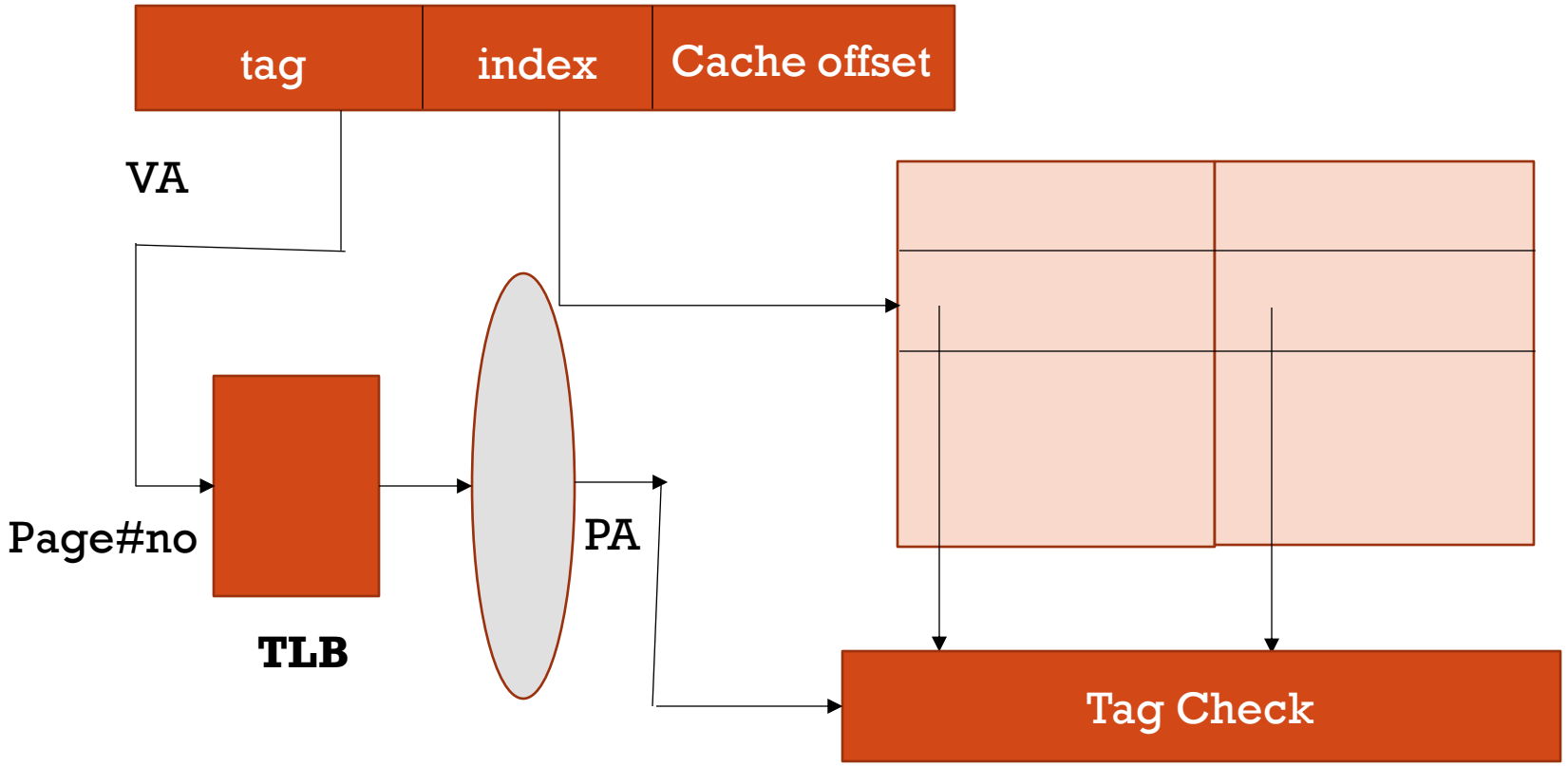
- TLB in addition to the frame nos. also has certain bits to store permissions, like valid bits etc.

- So, even though we do not need to get the VA→PA mapping through the TLB, the processor needs to access the TLB to get the permissions that tell us whether we are allowed to read, write or execute that location.

PROBLEMS WITH VIRTUALLY ACCESSED CACHE: PROBLEM 2

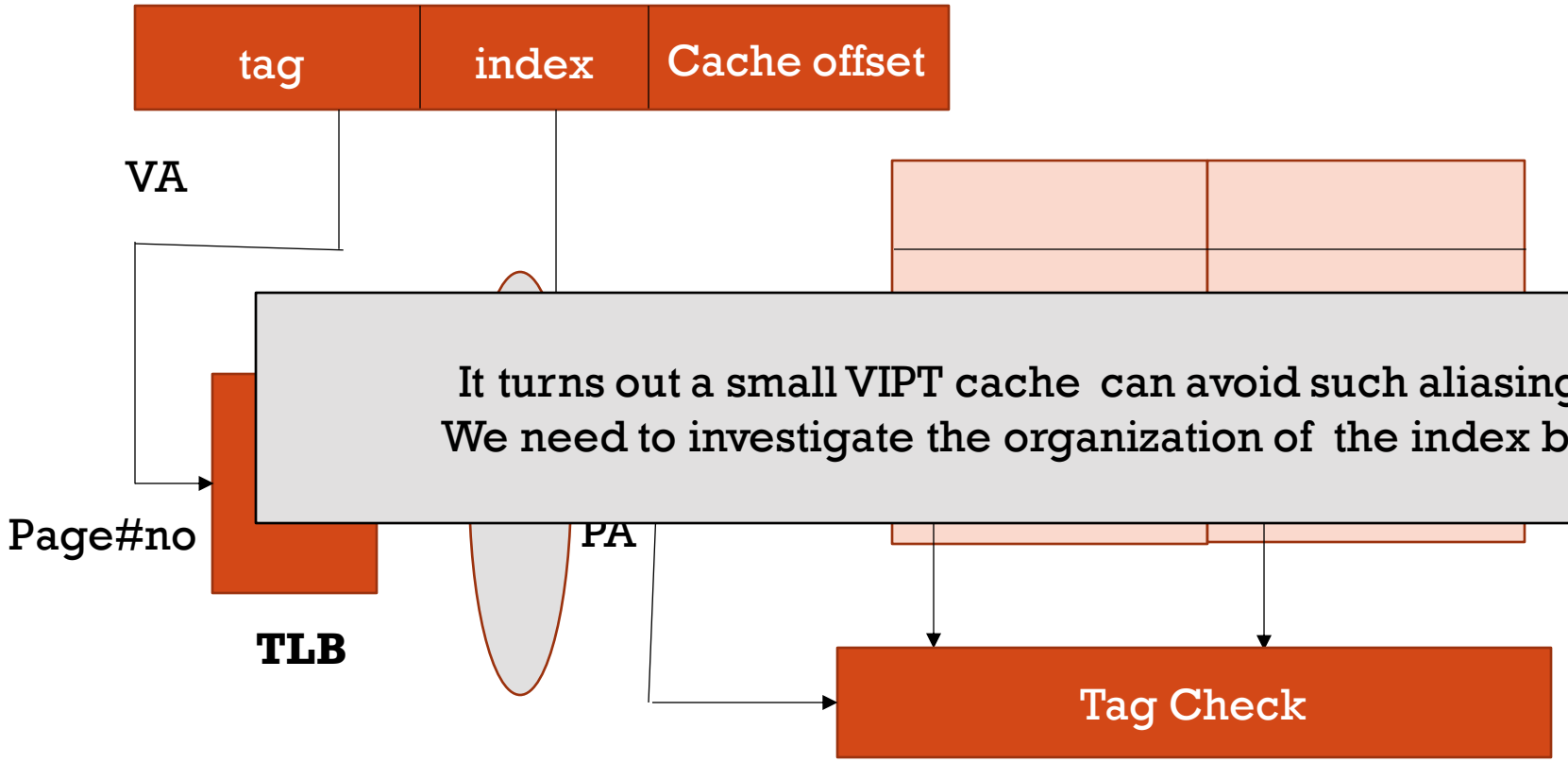
- A bigger problem is in the case of context switch.
  - VA is specific to a process.
  - One process runs and we fill the cache with some data
  - Another process executes, the virtual addresses that may overlap with the addresses from the previous process, should be targeted to different data.
  - It should be translated to different physical address in the TLB etc.
  - But the cache only sees the VA!
- Thus, we need to flush the cache on context switch.
- Every time there is a context switch there is a burst of cache misses.

VIRTUALLY INDEXED PHYSICALLY TAGGED (VIPT) CACHE



- Hit Time=Cache Hit Time (as the TLB access is typically faster)
- Context Switch?
  - The tag check is done with the physical tag.
  - So, the VA for a different process may map to the same cache set.
  - But, the actual tag will not match, and we will have a miss.
  - Data would be brought from the main memory to the cache.
  - Thus, it will operate like the physical cache.

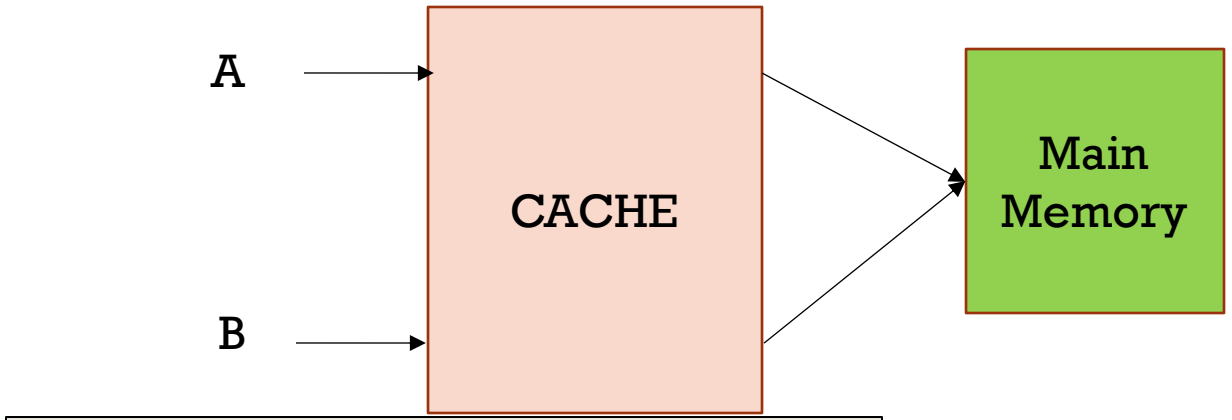
VIRTUALLY INDEXED PHYSICALLY TAGGED (VIPT) CACHE – ALIASING?



It turns out a small VIPT cache can avoid such aliasings!
We need to investigate the organization of the index bits.

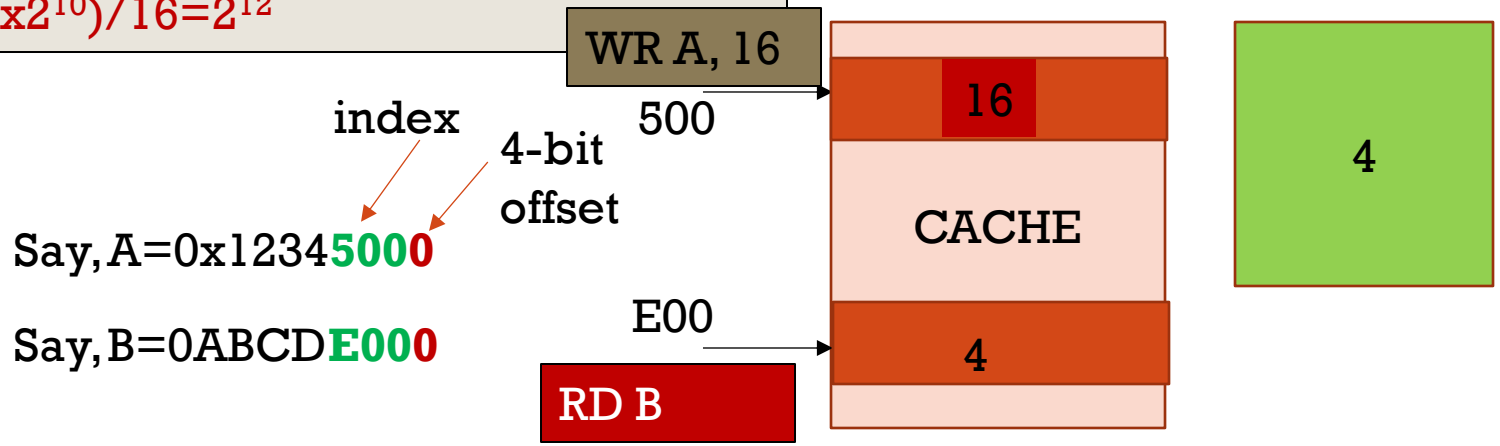
- But what about aliasing?
- Two different VAs can map to the same PA.
- But because we access
  with the VAs,
  nt end up in
  sets of the
- Thus, write to one address will not be seen by the other.

ALIASING EXAMPLE IN A VIPT



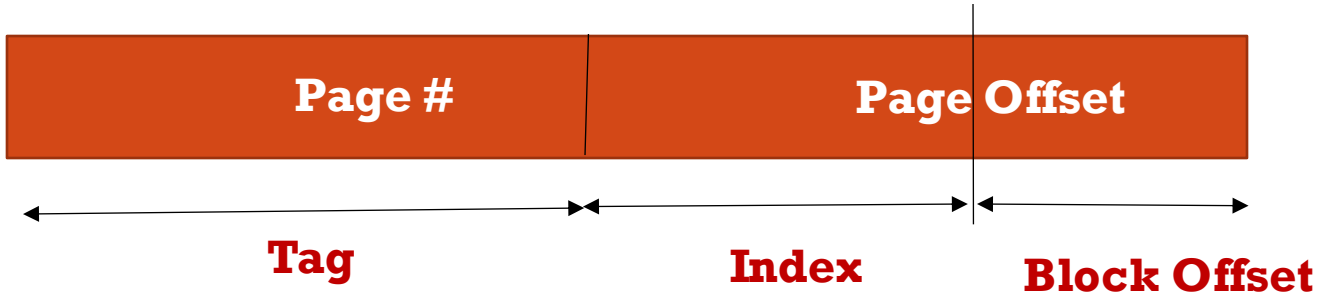
mmap(A,4096,fd,0) – creates a new mapping in the VA space of the calling process. It puts physical memory at a given location in program virtual memory, and optionally copies a file's contents there.
See:
https://www.cs.uaf.edu/2016/fall/cs301/lecture/11_02_mmap.html
Likewise, mmap(B,4096,fd,0)

Consider, a 64kB, Direct Mapped Cache with 16-byte Block Size.
Hence, no of index bits=12, as $(64x2^{10})/16=2^{12}$



Write Back Cache

AVOIDING THE ALIASING



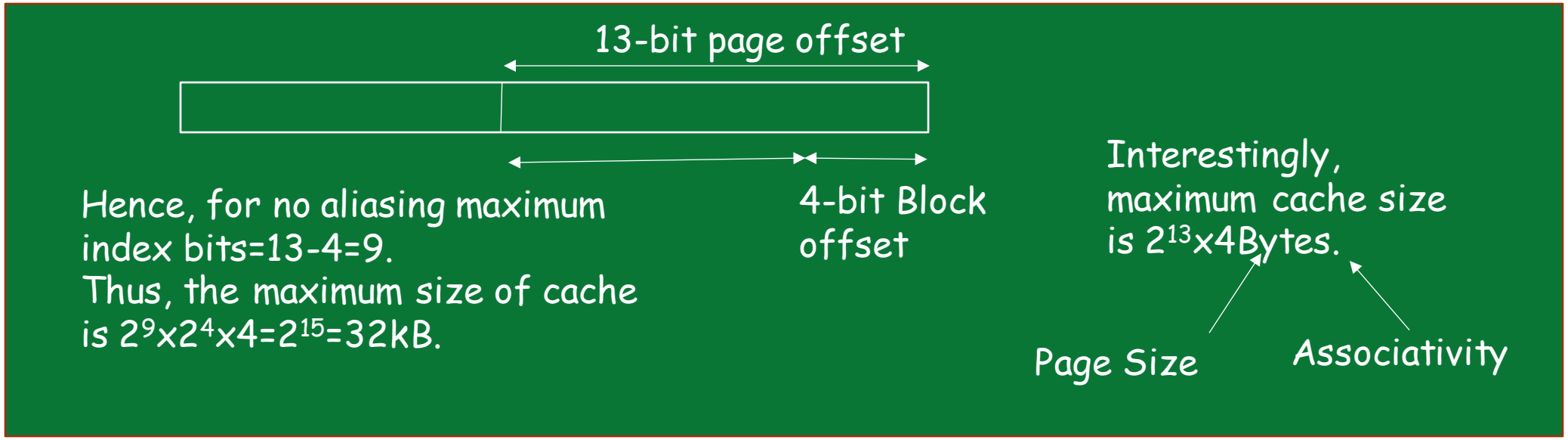
- Consider, 4 kB page size => 12-bit page offset
- 32 Byte Cache Block => 5-bit block offset
- Thus, if the cache has less than (12-5)=7 bits, ie. 128 sets then the index bits all come from the "Page Offset".
- For aliasing we are considering two VAs whose PAs are same => Page Offsets are also same.
- Thus, here the index is also same, and both the VAs map to the same set in the cache, hence there is no aliasing!

VIPT QUIZ

- 4-way Set Associative Cache
- 16 Byte Block Size
- 8 kB Page Size, ie. $2^3 x 2^{10} = 2^{13}$. So, 13 bit is for Page Offset.
- What is the maximum size of the cache for no aliasing?

VIPT QUIZ

- 4-way Set Associative Cache
- 16 Byte Block Size
- 8 kB Page Size, ie. $2^3 x 2^{10} = 2^{13}$. So, 13 bit is for Page Offset.
- What is the maximum size of the cache for no aliasing?

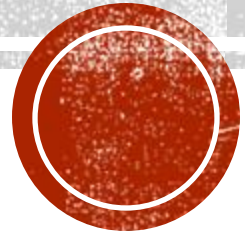


REAL VIPT CACHES

CACHE SIZE≤ASSOCIATIVITY X PAGE SIZE
PENTIUM 4: 4 WAY SET ASSOCIATIVE, WITH PAGE SIZE=4KB. THUS, THE. L1 CACHE IS OF SIZE 16KB.
CORE-2, NEHHALEM, SANDY BRIDGE, HASWELL: 8 WAY SET ASSOCIATIVE, PAGE SIZE=4KB. THUS, L1 CACHE IS OF SIZE 32KB.
SKY LAKE: 16 WAY SET ASSOCIATIVE. THUS CACHE SIZE IS 64KB.

ADVANCED CACHE OPTIMIZATIONS

Prof Debdeep Mukhopadhyay
Computer Sc &. Engg,
Indian Institute of Technology Kharagpur



ASSOCIATIVY AND HIT TIME

Relationship between Associativity and Hit Time?

High Associativity Implies:
1. Fewer Conflicts => Miss Rate ↓
2. Larger VIPT Cache => Miss Rate ↓
3. Slower Hits

Low Associativity Implies:
1. Miss Rate ↑
2. Hit Time ↓

So. How to combine the two approaches?
The idea is called Way Prediction…

WAY PREDICTION

Way Prediction

Let us consider a Set Associative Cache: The Miss rate is lower than a direct mapped cache.

The idea of way prediction is based on an additional feature to predict/guess which line in a set is the most likely to result in a HIT

If we guessed correctly, then access time would be like a direct-mapped cache, hence HIT time ↓

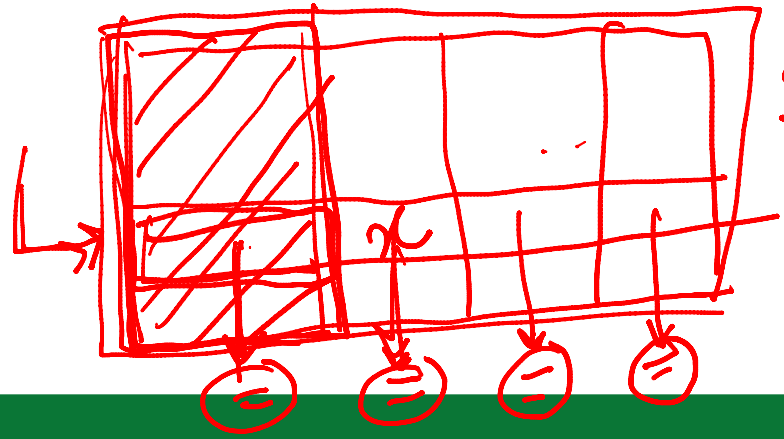
However if we guessed wrongly, then normal checks as in set associative cache is performed => miss rate is not affected!

WAY PREDICTION QUIZ

Way Prediction Quiz

	32 kB, 8-way Set Associative	**4 kB Direct Mapped**	**32 kB, 8-way Set Associative, way prediction**
HIT Rate	90%	70%	
HIT Latency	2	1	
MISS Penalty	20		
AMAT			

WAY PREDICTION QUIZ



Way Prediction Quiz

	32 kB, 8-way Set Associative	4 kB Direct Mapped	32 kB, 8-way Set Associative, way prediction
HIT Rate	90%	70%	90%
HIT Latency	2	1	1 or 2
MISS Penalty	20	20	20
AMAT	4	7	1·3 + (0·1 x 20) = 1·3 + 2 = 3·3

2+0.1x20=4

1+0.3x20=7

0·7x1 + 0·3x2 = 0·7+0·6 = 1·3

Hit time + Miss rate x Miss penalty.

WAY PREDICTION QUIZ

Way Prediction Quiz

	32 kB, 8-way Set Associative	**4 kB Direct Mapped**	**32 kB, 8-way Set Associative, way prediction**
HIT Rate	90%	70%	90%
HIT Latency	2	1	1 or 2
MISS Penalty	20	20	20
AMAT	4	7	3.3

2+0.1x20=4

1+0.3x20=7

0.7x1+0.3x2+0.1x20=3.3

REPLACEMENT POLICY AND HIT TIME

Replacement Policy and Hit Time

Random: Nothing to update on cache hit. On cache hit there is no replacement, and no data-structure to update. So lesser hit time! Only on cache miss, we randomly select a data from the cache and evict to make room => This results in Miss Rate↑

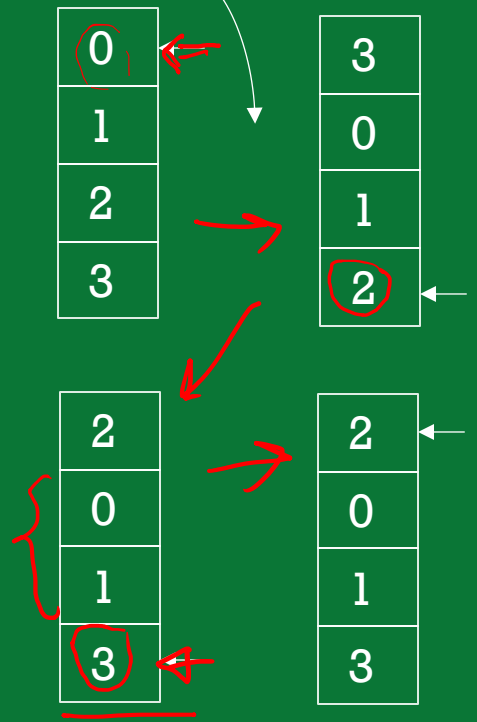
LRU (Least Recently Used): => Miss Rate↓

However, on every cache hit we need to update many counters to keep track on the recency. This will consume power! Even if we don't update any counter, we still need to check them to see if the ordering is correct and whether any update is to be done.

LRU EXAMPLE

A 4-way set associative cache

LRU Example

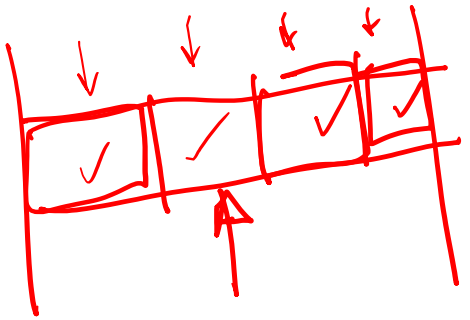


A Cache Hit can result in updating up to n counters, where n is the associativity.

So, for LRU, for a cache hit even if most recently used data is accessed, we need to find for every location whether lesser than or greater than the accessed data's counter value. This creates lot of unnecessary cache activity!

We want a replacement policy which evicts data which has less chance of being subsequently accessed, but further the policy should do have less activity on hit.

NMRU REPLACEMENT



8 bit n log n.
2 bit log n.

Not Most Recently Used

An LRU approximation:

1. We track which block in a set is MRU (Most Recently Used). How many bits are needed?
   - For a 4-way set associative cache, in a set there are 4 blocks. So, we need one 2-bit MRU pointer. For NRU policy, we need 4 2-bit counters.
2. On replacement, we pick a non-MRU block randomly.

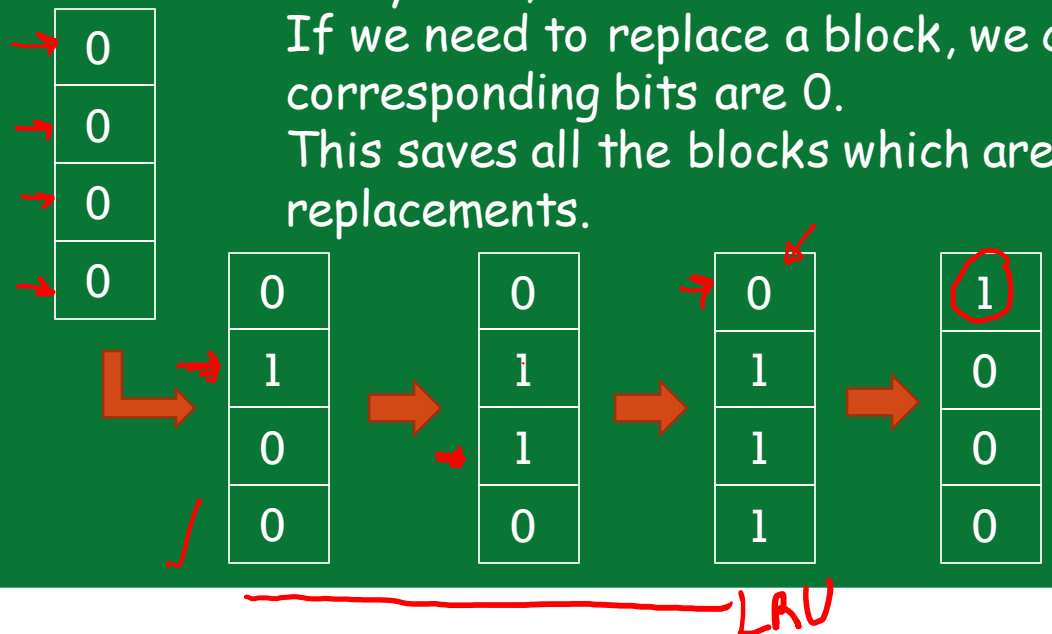
For n-way set associative cache, the NMRU policy requires n-times less states/counters than that of a LRU policy! However, it has a disadvantage in that we have no idea of the remaining blocks. So, we would like a policy less costly than LRU, but keeps more track than NMRU.

PLRU REPLACEMENT

Pseudo LRU

Here, we keep one bit per line in the set.
Consider, a 4-way set associative cache again. Here, we keep one bit for every block. Thus, we need to keep 4-bit (as follows), associating 1-bit per line/block.

Here, all the bits are initialized to 0.
Every time, a line is accessed it becomes a 1.
If we need to replace a block, we choose among the blocks whose corresponding bits are 0.
This saves all the blocks which are recently accessed from replacements.



We have a situation like the NMRU. We only know which block is MRU. For rest, we pick at random.

NMRU REPLACEMENT

Not Most Recently Used

As we start accessing some other blocks, we get a better idea of which are more and which are less recently used. So, it is between a NMRU and LRU policies.

When 1 bit is set, it is like NMRU.
When more than 1-bits are set (but not all-but-one) it is in between NMRU and LRU.
When all-but-one are set, we are like LRU: we know precisely which block is least recently used.

Note that the technique is very simple, it has very less cache activity on a cache hit. This helps to reduce hit time of the cache.

NMRU QUIZ

NMRU Quiz

- Fully Associative Cache with 4 lines
- NMRU replacement
- Initially empty

Processor accesses blocks:
A A B A C A D A E A A A A B

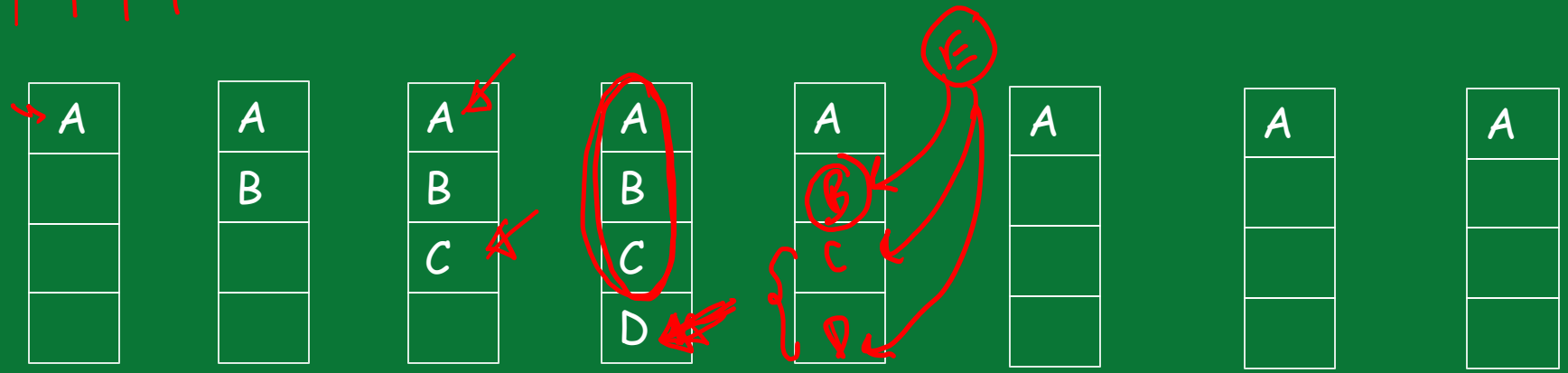
How many. Misses do we have? Give the minimum and maximum values.

NMRU QUIZ

NMRU Quiz

- Fully Associative Cache with 4 lines
- NMRU replacement
- Initially empty

Processor accesses blocks:
A A B A C A D A E A A A A B
M H M H M H M H M H H H H ?



NMRU QUIZ

NMRU Quiz

- Fully Associative Cache with 4 lines
- NMRU replacement
- Initially empty

Processor accesses blocks:
A A B A C A D A E A A A A B
M H M H M H M H M H H H H M

No of misses (maximum)=6

A	A	A	A	A
	B	B	B	E
		C	C	C
			D	D

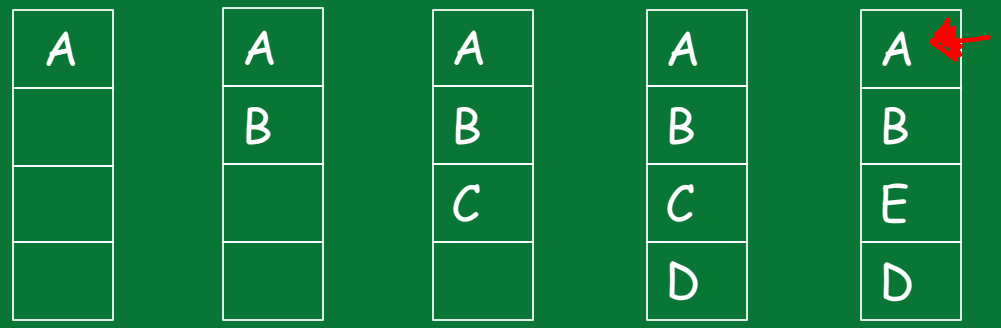
NMRU QUIZ

NMRU Quiz

- Fully Associative Cache with 4 lines
- NMRU replacement
- Initially empty

Processor accesses blocks:
A A B A C A D A E A A A A B
M H M H M H M H M H H H H H

No of misses (minimum)=5



TYPES OF CACHE MISSES

Types of Cache Misses

- Compulsory Misses:
  - First time when a block is accessed
  - Would be a miss even in infinite cache
- Capacity Misses:
  - Block evicted because of limited cache size
  - Would be a miss even in fully associative cache of that size
  - So, if we have an 8 kB direct mapped cache, an access would be treated as a cache miss if it would be a miss even on a fully associative cache of 8kB.
- Conflict Misses:
  - Due to conflict between sets.
  - Due to block which has been evicted due to limited associativity.
  - Would not be a miss in fully associative cache.

REDUCING MISS RATE

- We have seen the 3Cs: compulsory, capacity, and conflict misses which tells us the reasons for the cache misses.

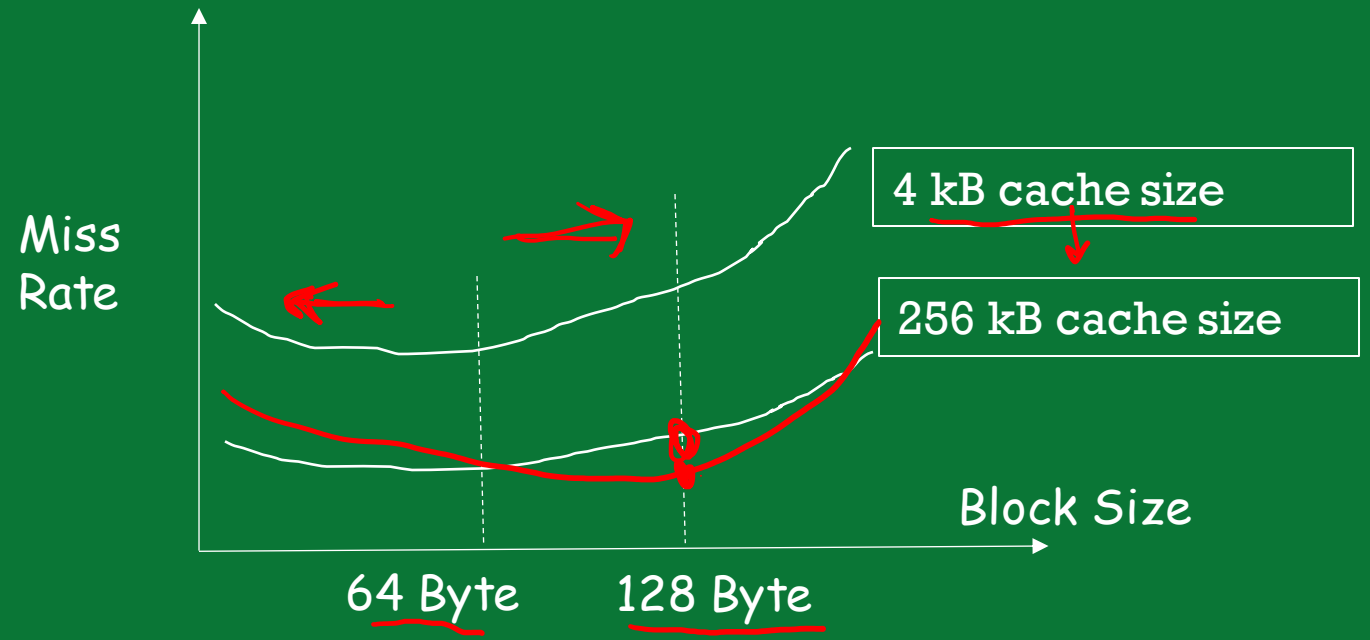
- So, how can we reduce the Miss rate?
  - A large cache?: Capacity miss should reduce.
  - What about the associativity?: Conflict misses should reduce.
  - A better replacement policy?: Should reduce conflict misses.

However, all these also would increase the hit time of the cache!

REDUCING MISS RATE

- Have Larger Cache. Blocks:
  - More words brought in on a miss.

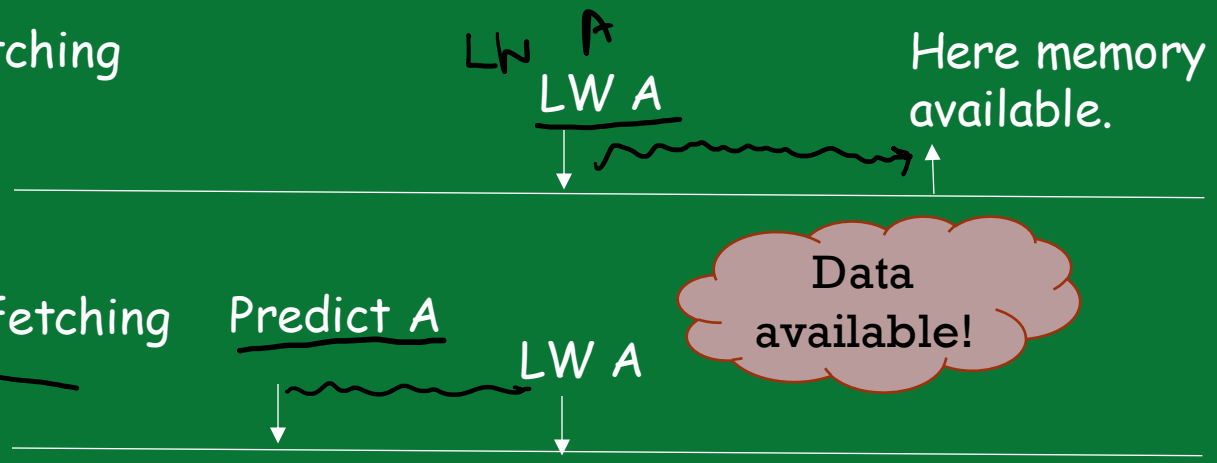
- Miss rate ↓ when spatial locality is Good.
- Miss rate ↑ when spatial locality is Poor.
  - Because our cache has more junk then.



PREFETCHING

Predictive technique attempting to bring data or. Instructions to the cache before their usages, to reduce delay

Guesses which blocks will be accessed soon
Brings them into the cache ahead of tim



Good guesses ⇒ Eliminate the miss penalty
Bad guesses ⇒ Cache Pollution ⇒ other misses (as the garbage data will evict other useful things/data)

CRITERIA

- Accuracy: ratio of useful prefetches to number of prefetches generated
- Coverage: Ratio of useful prefetches to number of misses when prefetching is turned off

100 → 70 70%

- Timeliness: Whether the prefetch was too early or too late.

PREFETCH INSTRUCTIONS

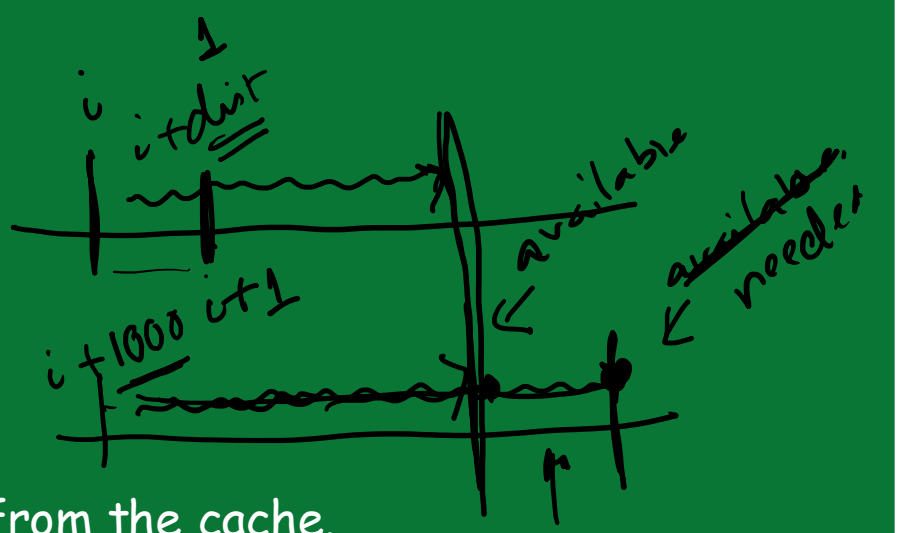
Software-

Powerful microprocessors have prefetch instructions in their ISA

```
for (i=0;i<10; i++)
    sum += a[i];
```

⇒

```
for (i=0;i<10; i++)
    prefetch a[i+pdist]
    sum += a[i];
```



What to use as pdist?

If it is too small, data will arrive later than needed

If it is large, the prefetched data may be evicted from the cache.

Further the same code with same pdist value will not work for faster generation processors

The challenge is related to timeliness.

QUIZ

```
for (i=0;i<1000;i++)
    for(j=0;j<1000;j++)
        a[i]+=b[i][j];
```

prefetch a[i+x]

prefetch b[i][j+y]

10 cycles

Consider, the elements are 8-byte elements, and the cache is 16KB fully associative LRU cache

Each inner iteration takes 10 cycles if there is no miss

The miss penalty is 200 cycles (memory latency)

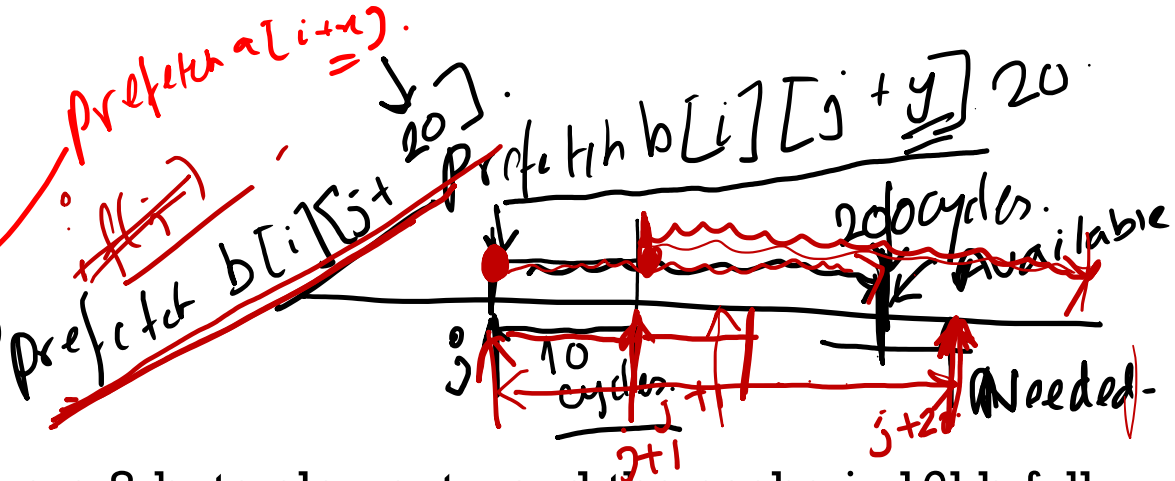
Which are better options:

1) prefetch a[i+?] or No prefetching

2) prefetch b[i][j+?] or No prefetching

QUIZ

- for (i=0;i<1000;i++)
  for(j=0;j<1000;j++)
    a[i]+=b[i][j];



Consider, the elements are 8-byte elements, and the cache is 16kb fully associative LRU cache

Each inner iteration takes 10 cycles if there is no miss

The miss penalty is 200 cycles (memory latency)

Which are better options:

1) prefetch b[i][j+y] or No prefetching

2) prefetch a[i+x] or No prefetching

y=20

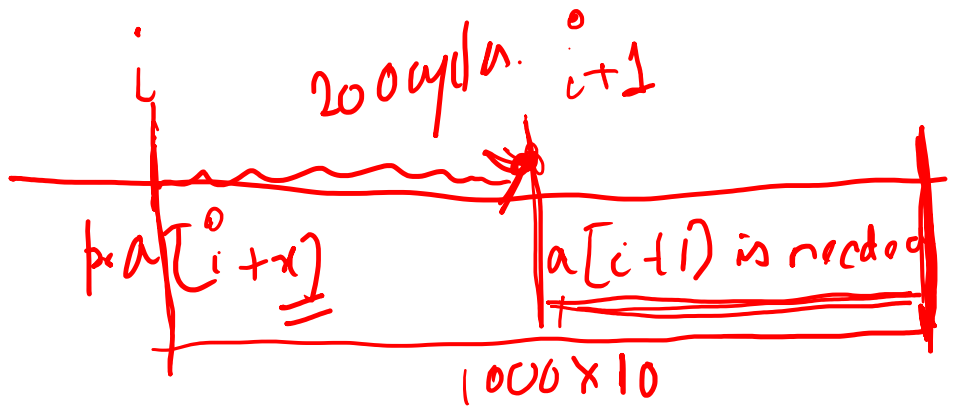
1) b[i][j+y] is ready by 200 cycles. Inner iteration takes min 10 cycles. So, if we set y=20, then the corresponding data will not be needed before 20x10=200 cycles. So data is available by then, and the prefetching helps.

QUIZ

- for (i=0;i<1000;i++)

  for(j=0;j<1000;j++)

  a[i]+=b[i][j];



Consider, the elements are 8-byte elements, and the cache is 16kb fully associative LRU cache

Each inner iteration takes 10 cycles if there is no miss

The miss penalty is 200 cycles (memory latency)

Which are better options:

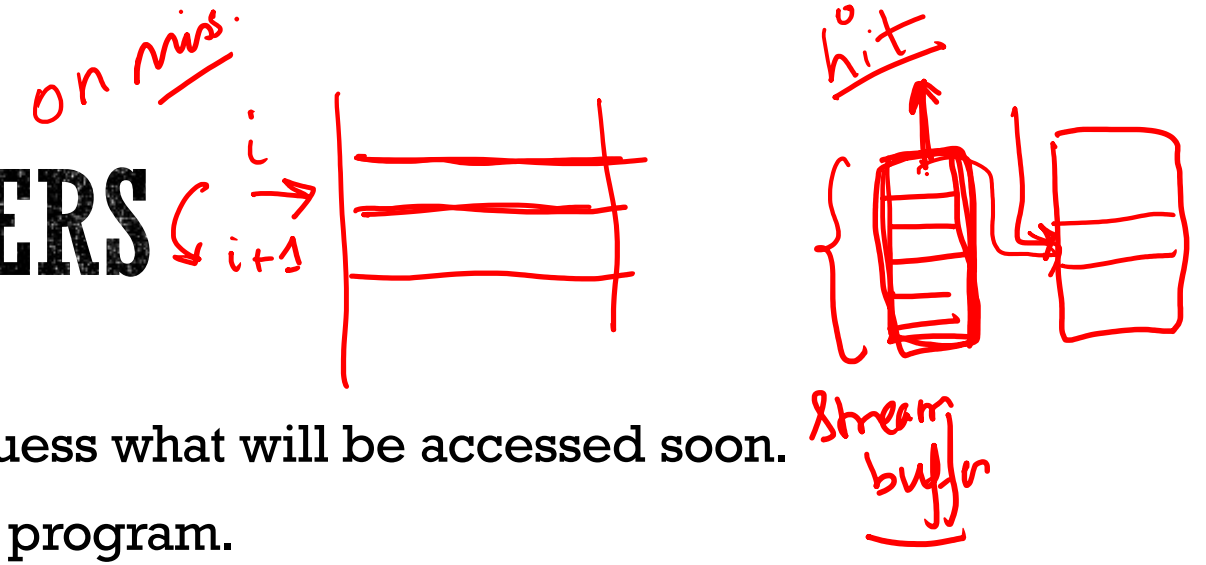
1) prefetch b[i][j+y] or No prefetching
2) prefetch a[i+x] or No prefetching

a[i+1]

x=1

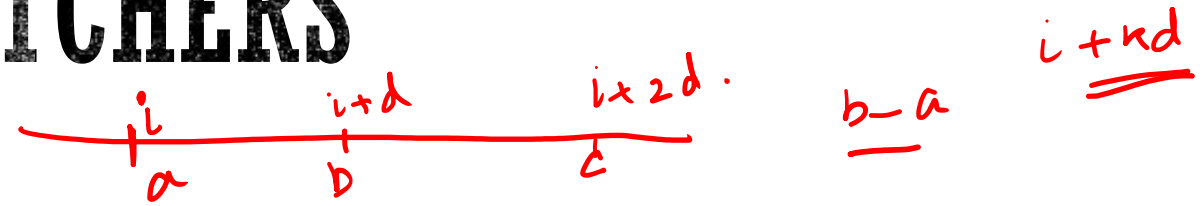
1) a[i+x] is ready after 200 cycles. If we set x=1, a[i+1] will be useful after the 1st iteration. This implies cache accesses for 8x1000 bytes<16KB of the cache. So, the prefetched data is not evicted
2) However, if set x=2, we need it after accessing 16x1000 bytes, which implies that the prefetched data can get removed before used. So, we set x=1.

HARDWARE PREFETCHERS

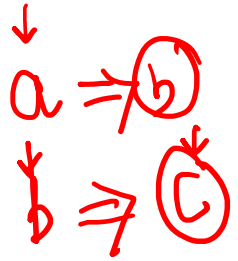


- The architecture supports prefetching to guess what will be accessed soon.
- We do not need to make any change to the program.
- Popular techniques:
  - Sequential prefetcher: prefetch of a line adjacent to the one that has just been referenced.
  - For I-caches, prefetch on miss is followed. This improves the accuracy.
  - Can be adaptive: IBM power4 upon a miss in line i, line i+x are prefetched with different x's and different number of lines depending on the level in the cache hierarchy.
  - In order to prevent cache pollution, the sequential lines are brought to FIFO structures of fixed sizes which are called stream buffers.
    - On a memory reference, both the regular cache and the head of the stream buffer are checked.
    - In case of a hit in the stream buffer, the line is transferred to the cache.
    - In case of a miss in both, the latter is flushed.

HARDWARE PREFETCHERS



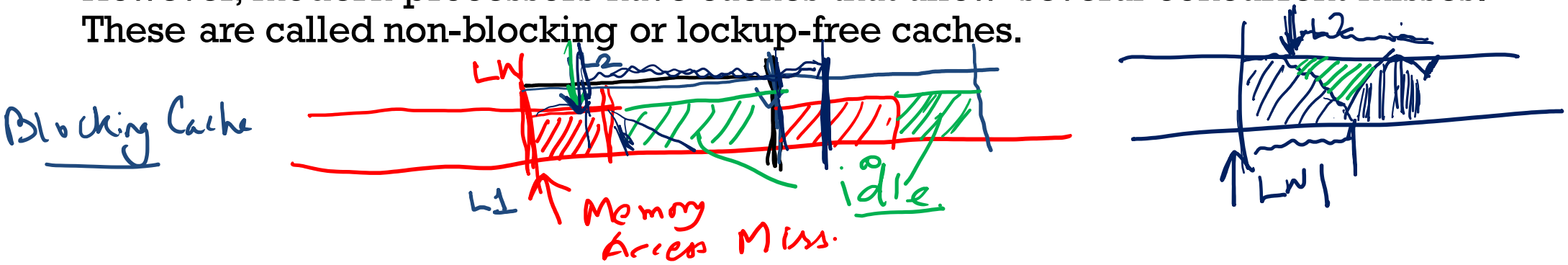
- Stride Buffer: In a program with nested loops the memory access patterns can exhibit constant strides.
  - If the stride is large, dedicated 'stride prefetchers' are used.
  - Given, a string of references: a, b, c, …, here b-a is called the stride.
- Correlating prefetchers: useful for data structures which are not in order like linked lists. It remebers, say a=>b and b=>c. so, next time a is accessed, b is, likewise, when b is accessed a is accessed.



NON-BLOCKING CACHES: REDUCE MISS PENALTY

lockup-free caches.

- For an out of order processor, instructions that follow the load that missed can enter the reservation stations and execute if the operands are ready.
- So, some part of the latency of the time to resolve the miss is overlapping with the processor activity.
- During the time of handling the 1st miss, and while executing the following instructions, processor may encounter another load, which also is a miss.
- In blocking caches, the second load cannot be done before 1st load is finished.
- However, modern processors have caches that allow several concurrent misses. These are called non-blocking or lockup-free caches.



NON-BLOCKING CACHES: REDUCE MISS PENALTY

- Hit under Miss: Allowing cache accesses resulting in hits while a single miss is outstanding. HP-PA 1700 implemented the same.

- Miss under Miss: This introduces Memory level Parallelism, which helps significantly to reduce the miss penalty.

Miss
L1

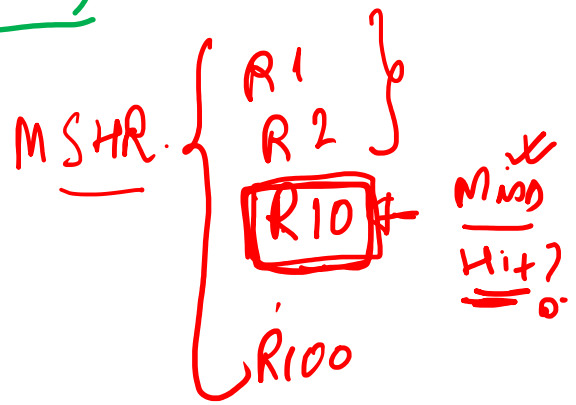
Hit
L2

Miss.

MISSING STATUS HANDLING REGISTERS

(MSHR).

- The idea is to encode the information related to a miss in MSHR
- When a cache miss occurs an MSHR is associated with it.
- They remember the misses that are in progress
- An MSHR for L1 cache contains a bit indicating whether it is free.
  - If there is no free MSHR the processor stalls until an MSHR becomes free.
- When there is a miss the processor checks the MSHRs (in parallel) if there is a match.
  - We intend to know if this is a new miss or a previous miss which has been requested to memory.



MISSING STATUS HANDLING REGISTERS

- In case of no-match: we allocate any MSHR, and remember which instructions to wakeup
  - This is a miss
- Alternatively, we have a miss to a block which was previously accessed and data has not yet arrived.
  - This is called a half miss
  - In a blocking cache this would have been a hit
  - However, as the memory is already requested the present memory access should not be again done.
  - Instead we add the instruction to MSHR.
  - When the data finally arrives all the waiting instructions are woken up.
  - Then the registers are released.

MULTI-LEVEL CACHE-(HIERARCHY)

- Miss in L1 cache goes to another cache. Thus,
  - L1 miss penalty is not the memory latency
- Consider, n-levels, L1, L2, …,Ln
- AMAT=L1 Hit Time + L1 miss rate x L1 miss penalty
- L1 miss penalty=L2 Hit time + L2 miss rate x L2 miss penalty
- L2 miss penalty=L3 hit time + L3 miss rate x L3 miss penalty
- …
- Ln miss penalty = Main Memory Latency

QUIZ

- True or False?
  - L1 capacity < L2 capacity
  - L1 latency < L2 latency
  - L1 # of accesses < L2 # of accesses
  - L1 Associativity < L2 Associativity

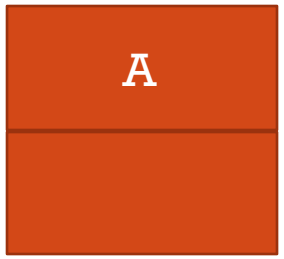
QUIZ

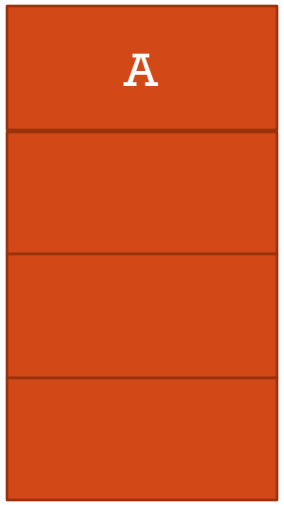
- True or False?
  - L1 capacity < L2 capacity
  - L1 latency < L2 latency
  - L1 # of accesses < L2 # of accesses (False)
  - L1 Associativity < L2 Associativity

INCLUSION PROPERTY

- Block is in L1;
  - May or may not be in L2
  - Has to be in L2 (inclusion)
  - Cannot be in L2 (exclusion)

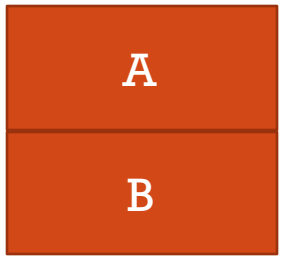
A	B	A	C	A	D	A	E

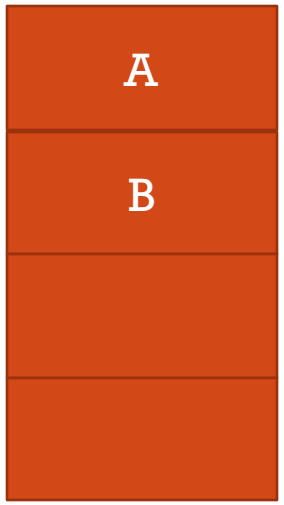




INCLUSION PROPERTY

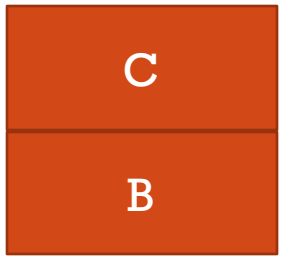
- Block is in L1;
  - May or may not be in L2
  - Has to be in L2 (inclusion)
  - Cannot be in L2 (exclusion)

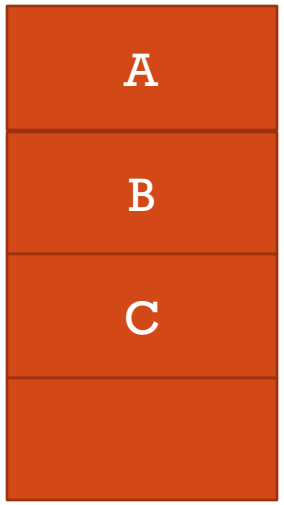




INCLUSION PROPERTY

- Block is in L1;
  - May or may not be in L2
  - Has to be in L2 (inclusion)
  - Cannot be in L2 (exclusion)

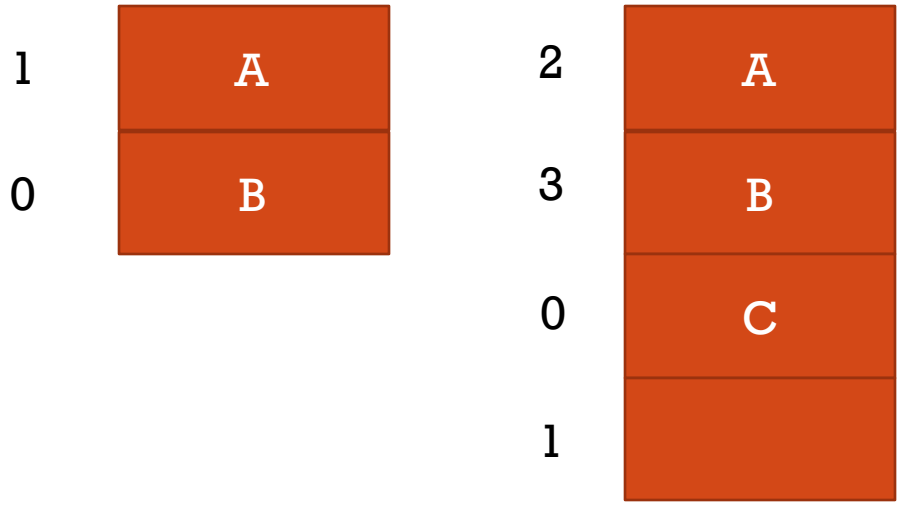




INCLUSION PROPERTY

- Block is in L1;
  - May or may not be in L2
  - Has to be in L2 (inclusion)
  - Cannot be in L2 (exclusion)

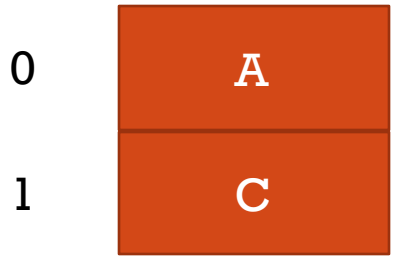
A	B	A	C	A	D	A	E

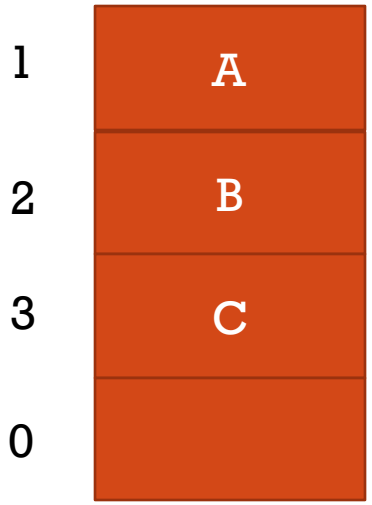


THIS A IS NOT ACCESSED IN L2

INCLUSION PROPERTY

- Block is in L1;
  - May or may not be in L2
  - Has to be in L2 (inclusion)
  - Cannot be in L2 (exclusion)

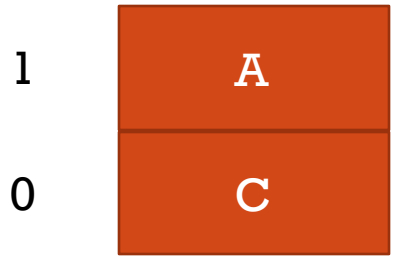


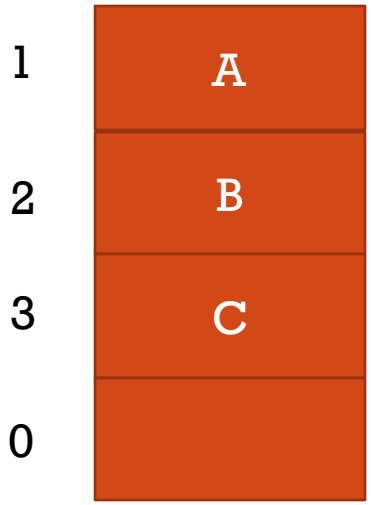


INCLUSION PROPERTY

- Block is in L1;
  - May or may not be in L2
  - Has to be in L2 (inclusion)
  - Cannot be in L2 (exclusion)

A	B	A	C	A	D	A	E

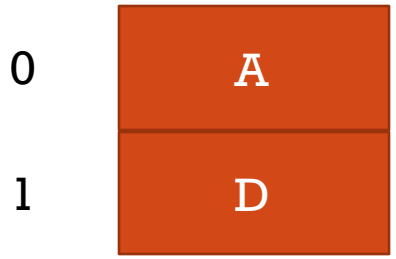


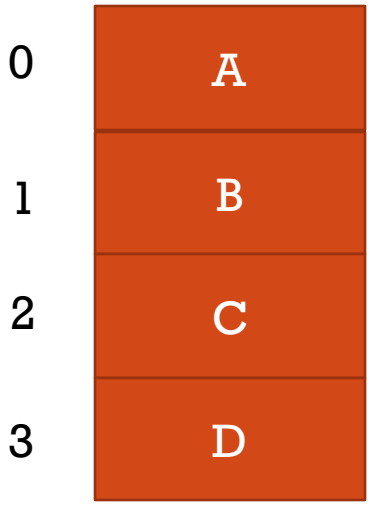


NO ACCESS IN L2

INCLUSION PROPERTY

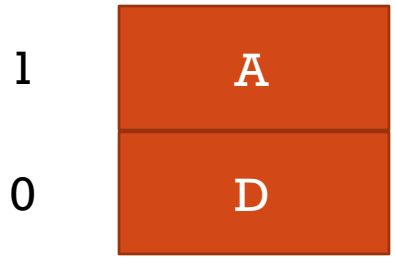
- Block is in L1;
  - May or may not be in L2
  - Has to be in L2 (inclusion)
  - Cannot be in L2 (exclusion)

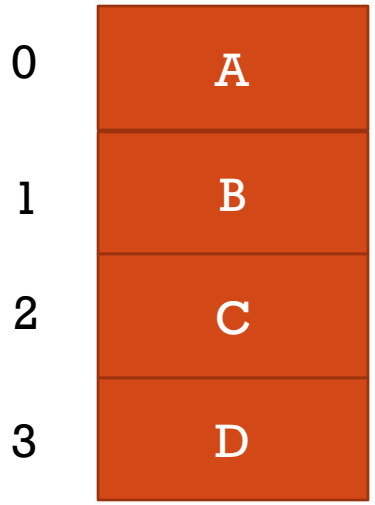




INCLUSION PROPERTY

- Block is in L1;
  - May or may not be in L2
  - Has to be in L2 (inclusion)
  - Cannot be in L2 (exclusion)

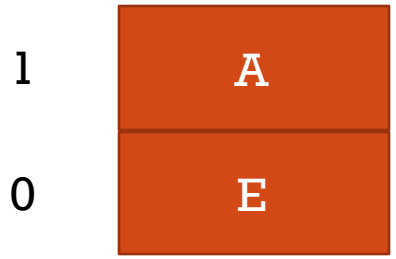


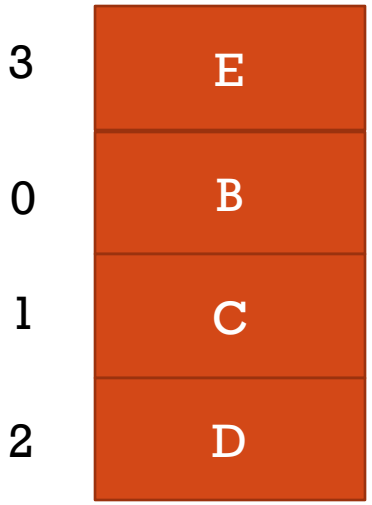


INCLUSION PROPERTY

- Block is in L1;
  - May or may not be in L2
  - Has to be in L2 (inclusion)
  - Cannot be in L2 (exclusion)

A	B	A	C	A	D	A	E





This violates inclusion property!

For inclusion the cache has extra dedicated bits in L2 cache. If that is set they are not evicted from L2.