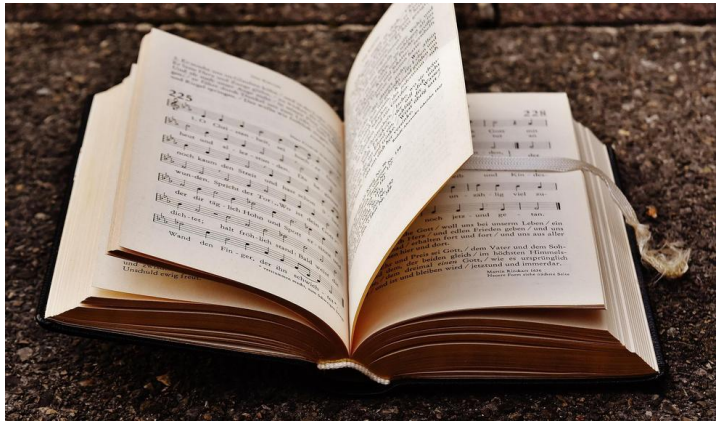


Chapter 7: Caches

Background Required to Understand this Chapter

- RC and CMOS Circuits
- Memory cells, latches and digital logic
- Branch Prediction

Chapter 3

CMOS Logic

Contents

How do we read books?

- Read the books on the desk more frequently.
- Keep books on similar topics on the desk.

Patterns in Real Life

Temporal Locality

- Tend to read the same book over and over again.
- The person will rarely fetch a book from the cabinet

Spatial Locality

- High probability of fetching and reading books on similar topics

Generalize to the memory hierarchy

Temporal locality	• Access the same address over and over again.
Spatial locality	• Access similar (proximate) addresses in the same window of time.

Laptop ⊐ desk ⊐ shelf Principle

Laptop — Fastest

- Most frequently read books and documents are in the laptop.

Desk — Slightly slower

- Slightly less frequent documents are on the desk

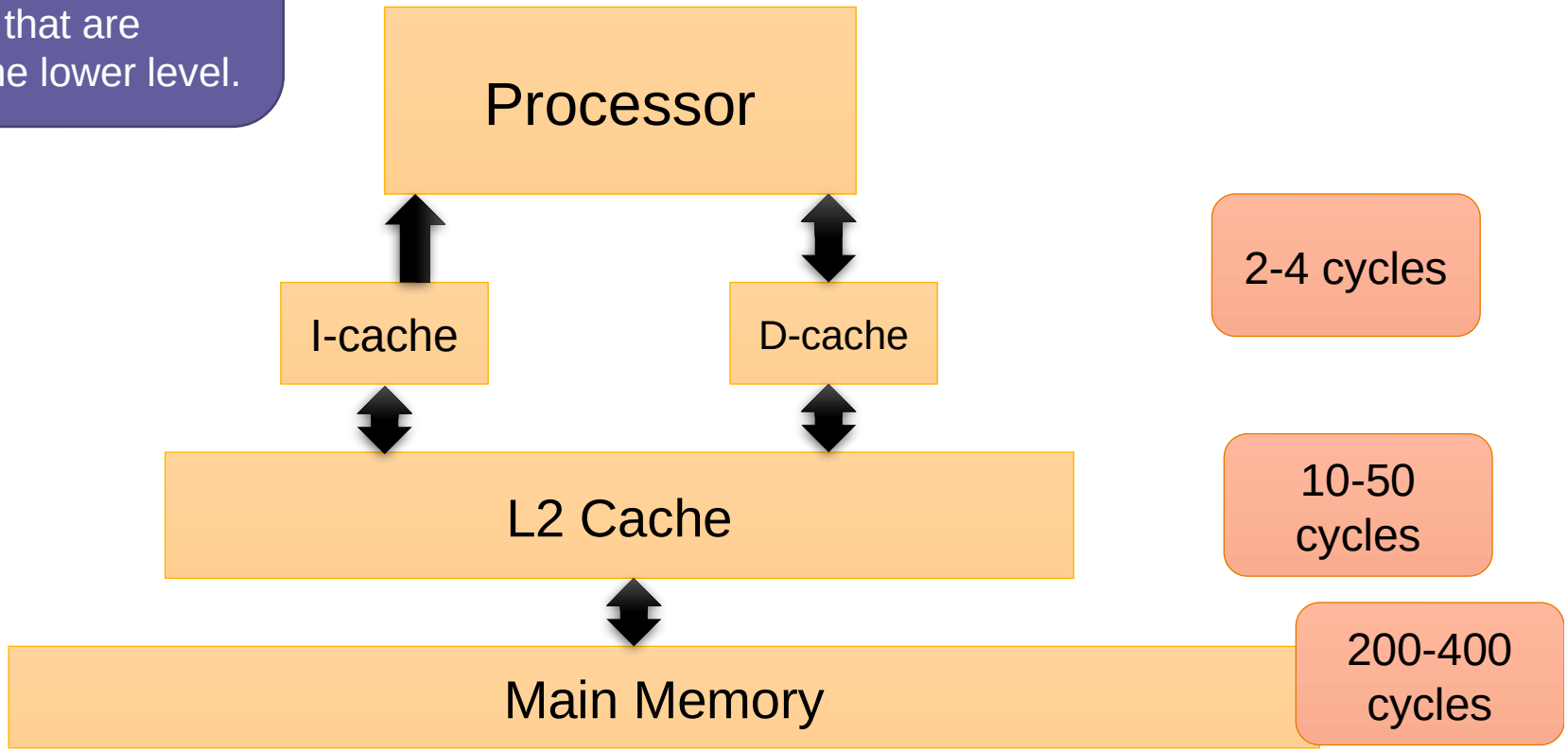
Shelves — Slowest

- Rarely accessed books are in the shelf

Increasing storage space

Apply a similar idea and create a memory hierarchy (use temporal locality)

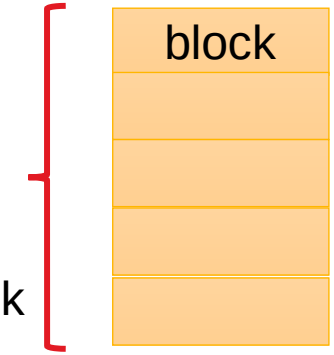
Inclusive cache hierarchy: each cache contains a subset of addresses that are stored in the lower level.



Use Spatial Locality

- Store data at the granularity of blocks: 32 to 128 bytes
  A block is also called a cache line
- We treat the block as an indivisible, atomic unit
- Even if we read or write one byte, we fetch the entire block

A cache simply contains a set of blocks along with some metadata per block

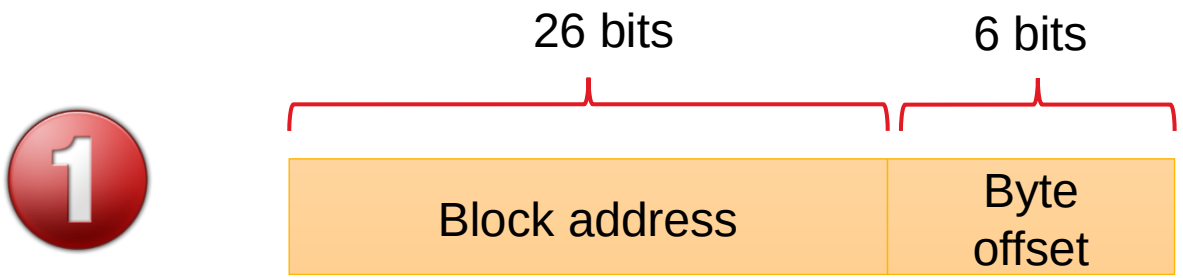


- We don't want to maintain separate per-byte state in a block
- Hence, we never fetch a subset of a block
- While writing we wait for the block to be fist loaded into the cache.

Storing Blocks in a Cache

Problem

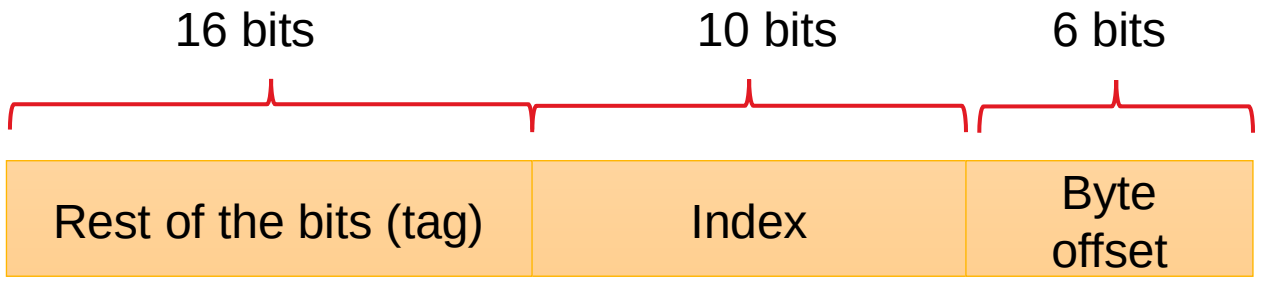
In a 32-bit memory system, design a cache with a 64-byte block size. Total size: 64 KB



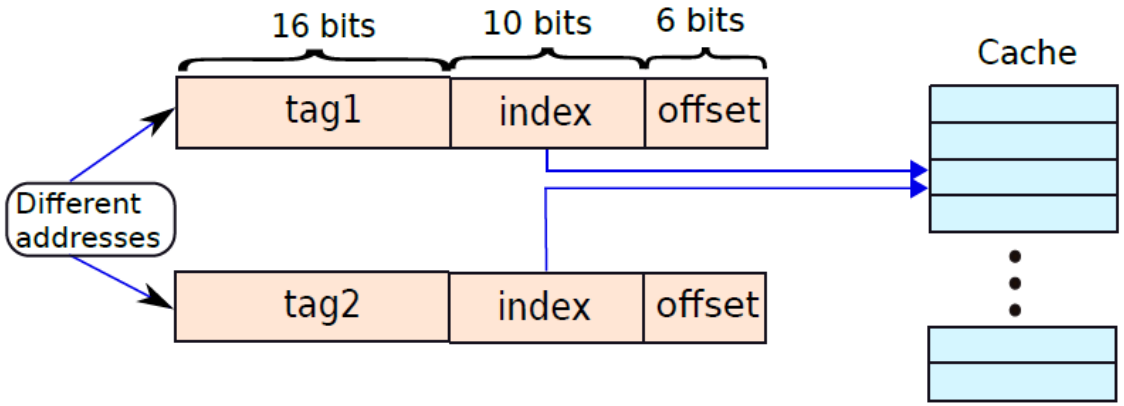
Break the address into two parts: block address and byte offset

A Simple Mapping

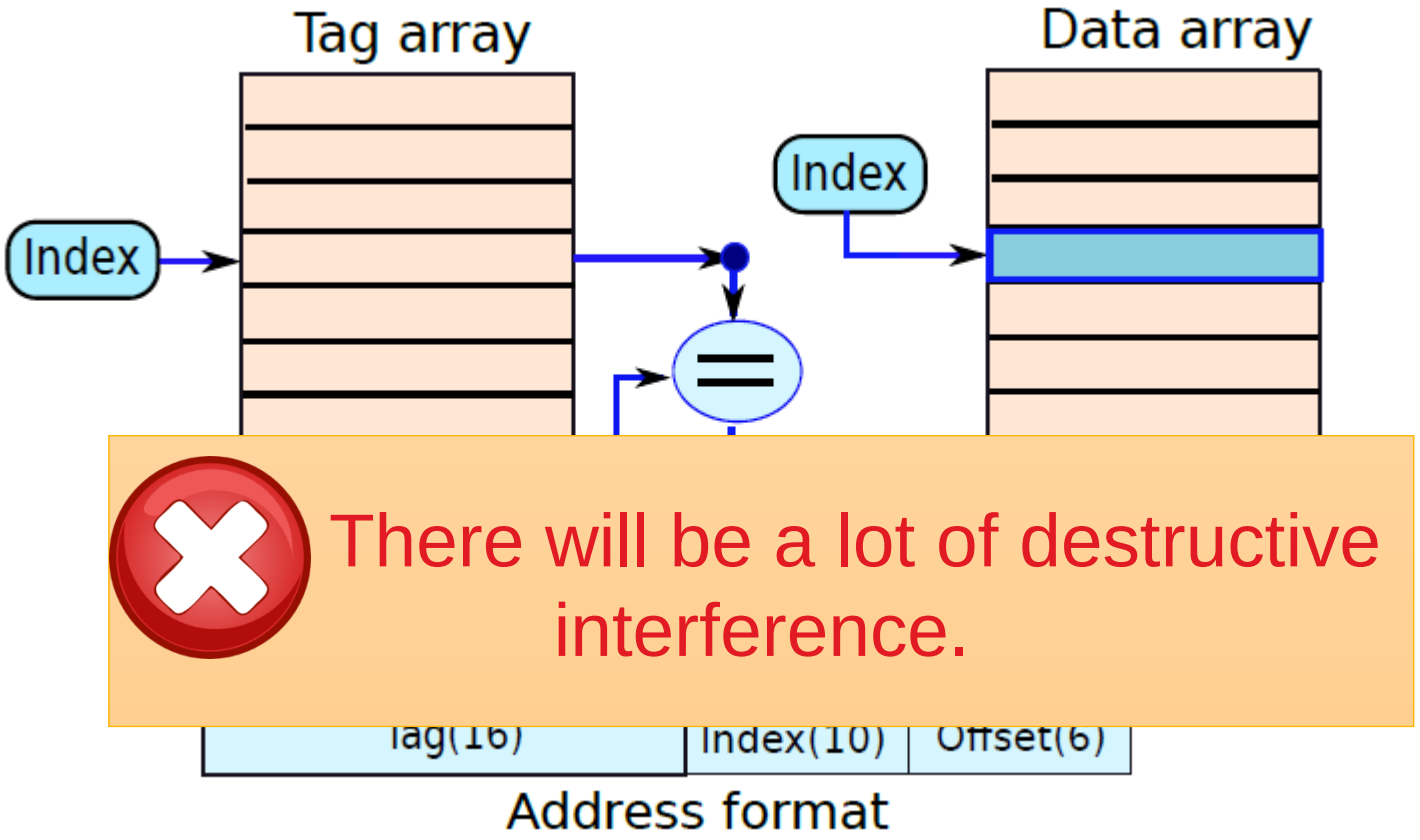
Number of cache lines = 64 KB/ 64 B = 1024



Aliasing is possible



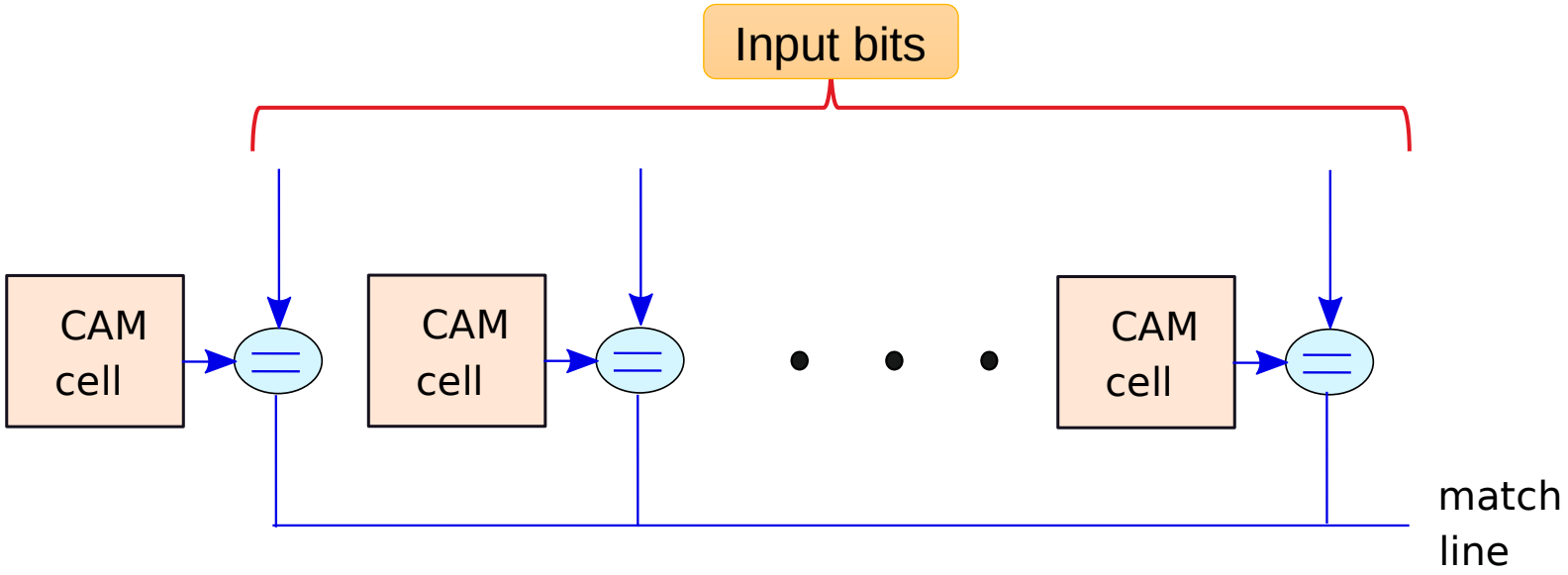
Direct Mapped (DM) Cache: Solves the problem of aliasing



- Have a separate tag and data array
- Access the tag and data array in parallel
- If the tag matches, read or update the corresponding data block

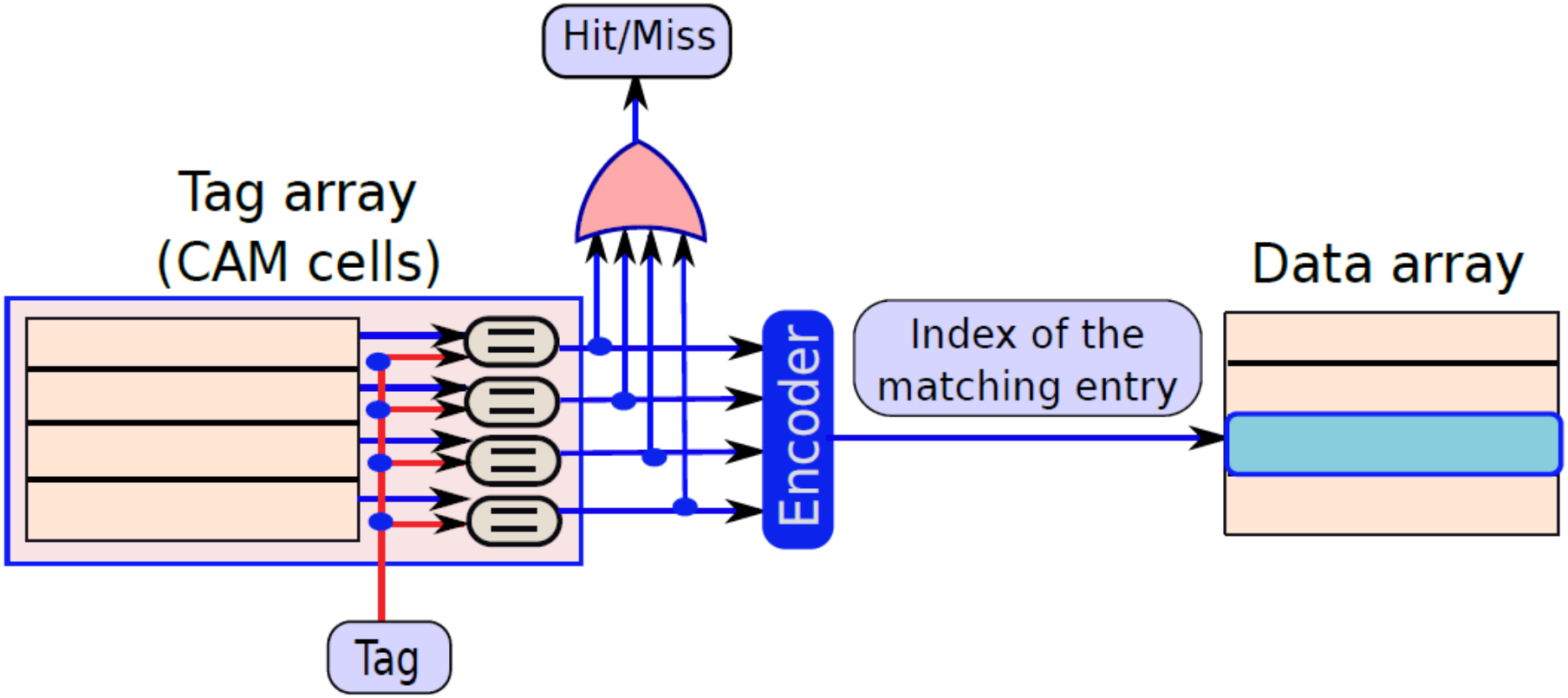
Fully Associative (FA) Cache

- Array of CAM (content addressable memory) cells in each row



- The match line is set to 1 only if all the CAM cells (contain 1 bit) match the corresponding input bit
- Problem with a DM cache ⇒ bind a block to a single cache line
- We can use a CAM array to implement a fully associative (FA) cache where a block can be stored in any cache line

Fully Associative Cache - II

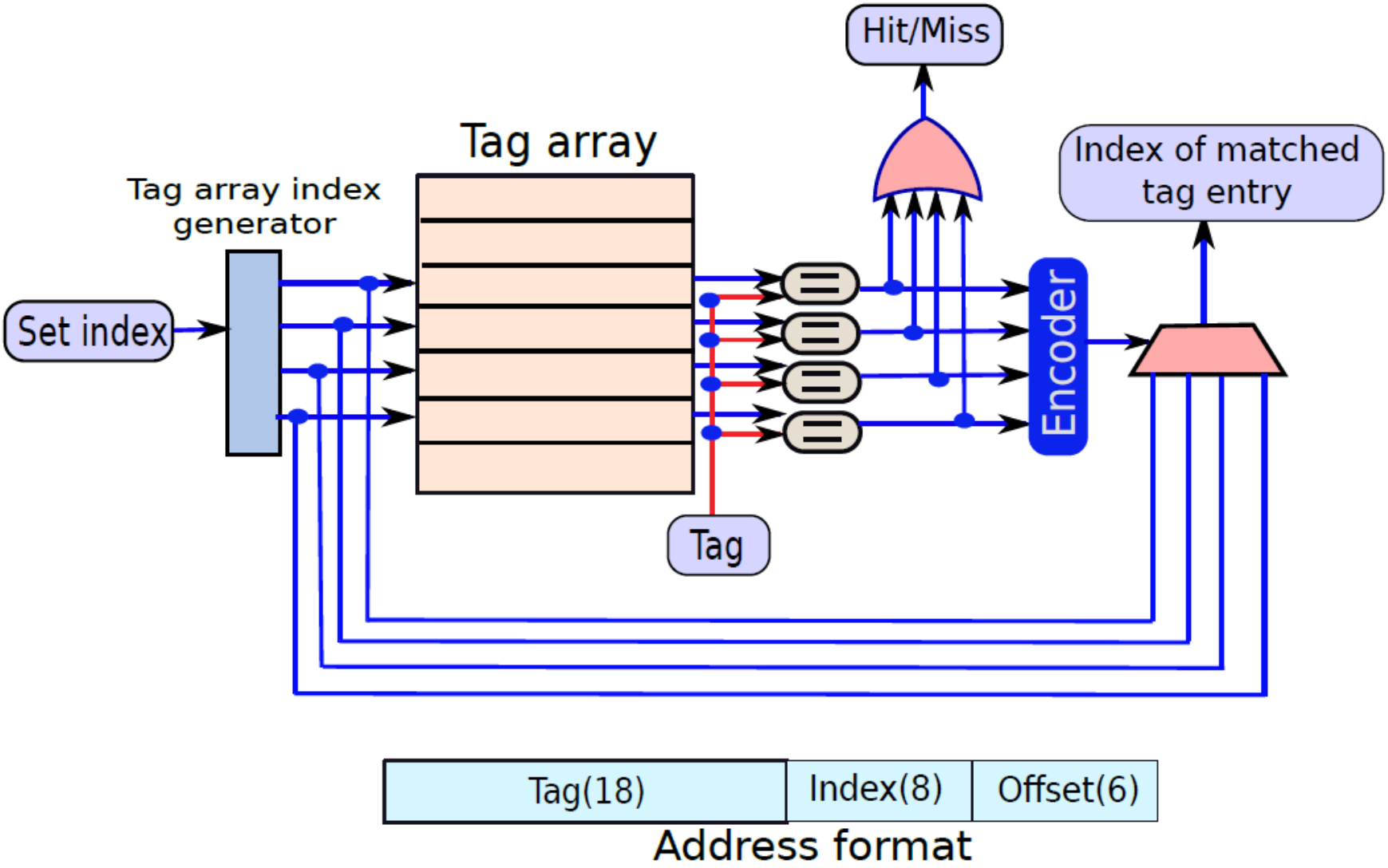


- A block can be stored in any cache line
- Disadvantage: Such caches are slow and power hungry

Compromise Solution: Set Associative Cache

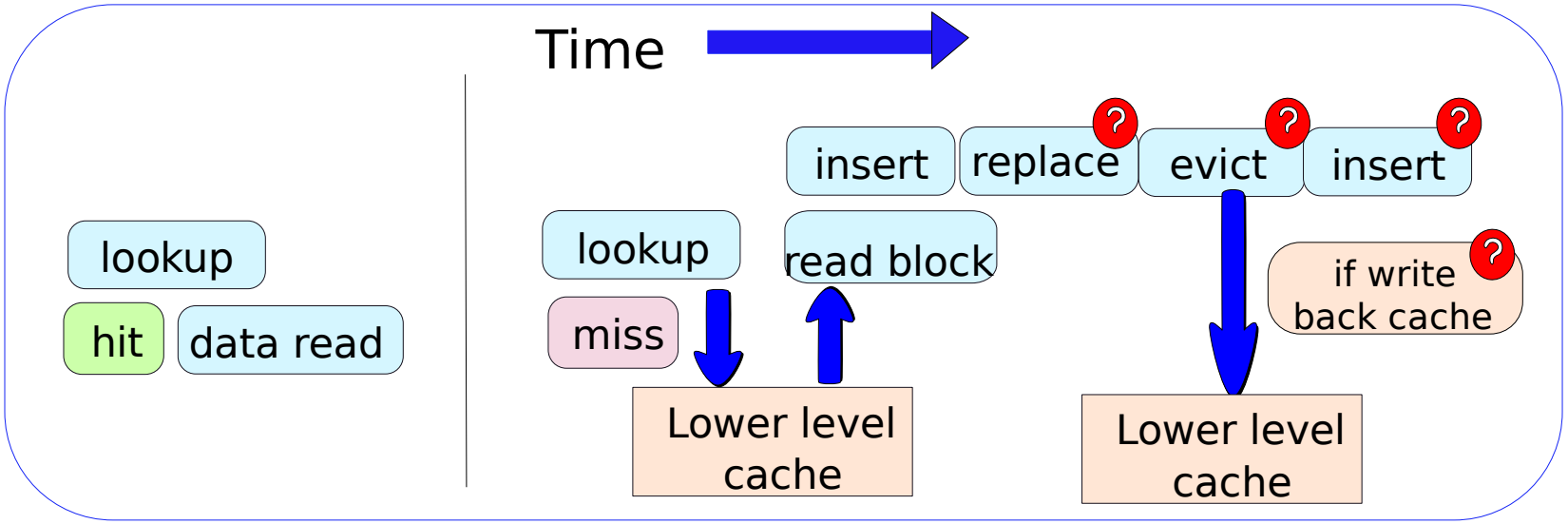
- Divide the 1024 cache lines into 256 sets
  - Each set contains 4 cache lines
- Map a block to a set
  - Within the set it can be stored in any cache line
  - A flexible design that is fast and reduces the probability of aliasing
- We access the tag array
  - Read all the tags in the set
  - If one of them matches the tag part of the address: declare a hit
  - Else it is a miss

Diagram of a Set Associative Cache



Basic Operations in a Cache

The *read* Operation



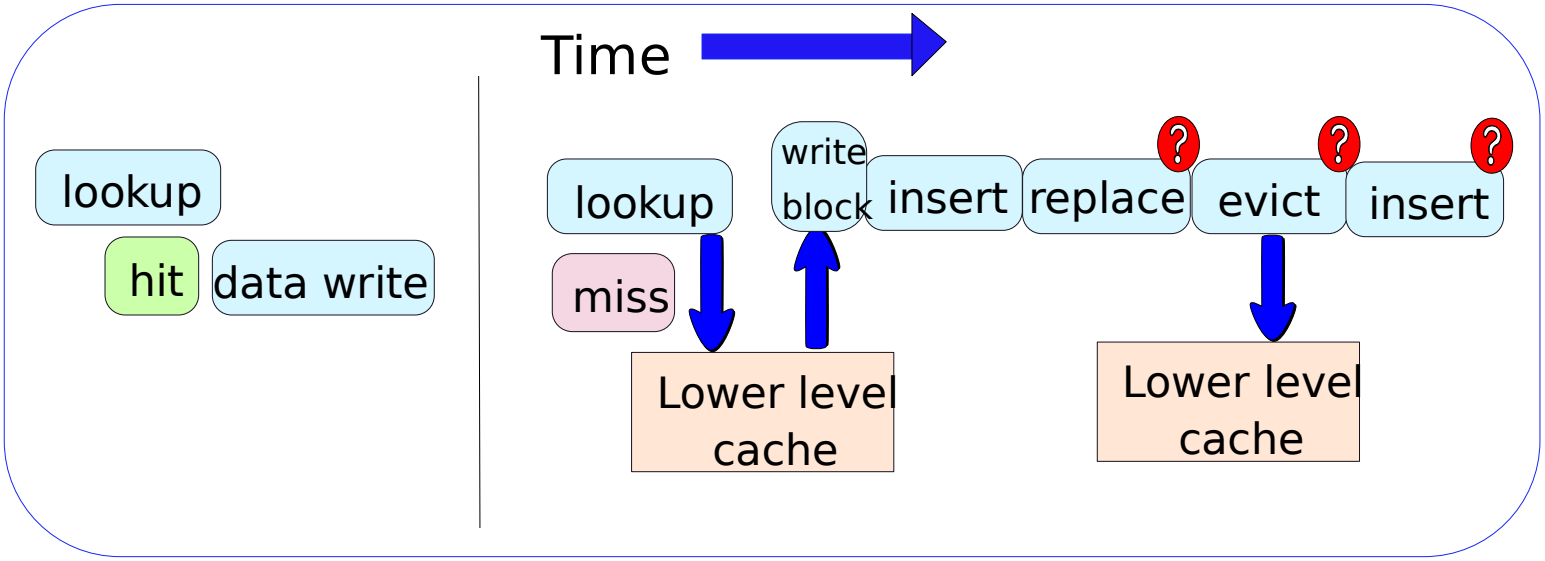
- Along with looking up the tag, read all the blocks of the set in parallel
- If there is a hit, choose one of the read blocks (performance optimization)
- If there is a miss, read the block from the lower level and find an empty line in the set
- If there is no empty line, evict a block in the set to create space for the new block

The Valid bit and Modified bit

- Enhance the tag array to contain two important bits
  - Valid bit ⎵ Does this line contain valid data?
  - Modified bit ⎵ Has any byte in the block been modified?

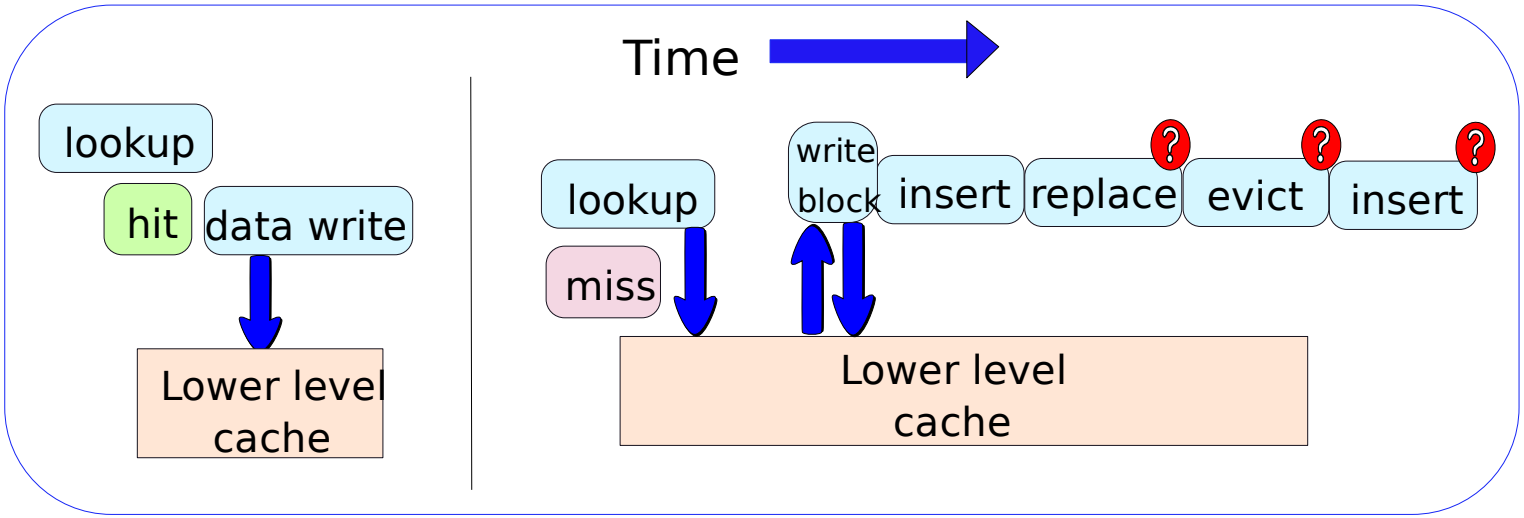
Write-through scheme	Write-back scheme
• Propagate every write to the lower level • We can seamlessly evict the block • Consumes a lot of power	• Set the modified bit, if there is a write access. • At the time of eviction ⎵ If the line is modified, write it back to the lower level.

Write Operation (write-back cache)



- If there is a hit, the access is straight-forward. Set the modified bit.
- If there is a miss, write to the block, and try to insert it.
- If there is no space, find a replacement candidate, and evict it

Write Operation (write-through cache)



- In a write-through cache, we can seamlessly evict a line
- However, at the time of writing, the write needs to be propagated to the lower level.
- Because the hierarchy is inclusive, there will be a hit at the lower level.

Cache Replacement Policies

- Random replacement
- FIFO (first in, first out) replacement
- LRU ⎵ Least recently used
  - Impractical because we need to maintain timestamps
  - Use Psuedo LRU
    - Have a 3-bit saturating counter with each tag
    - Increment on every access
    - Periodically decrement all counters
    - Choose the entry in the set with the least counter

Mathematical Analysis

Average Memory Access Time

$$AMAT = L1_{hit\ time} + L1_{miss\ rate} * L1_{miss\ penalty}$$

$$L3_{miss\ penalty} = main\ memory_{access\ time}$$

Simple Optimizations

Basic Strategy

- Reduce hit time
- Reduce miss rate
- Reduce miss penalty

Three kinds of misses

Cold or compulsory misses

- Misses incurred the first time we read or write a block

Capacity misses

- Misses incurred because of the limited size of the cache

Conflict misses

- Misses incurred because of destructive interference

Simple Optimizations

- Reduce hit time
  -  Small and simple caches
  -  Increases the miss rate
- Reduce compulsory misses
  -  Increase the block size, prefetching
  -  A large block size reduces the number of blocks
- Reduce capacity misses
  -  Prefetching, better compiler algorithms

Simple Optimizations – II

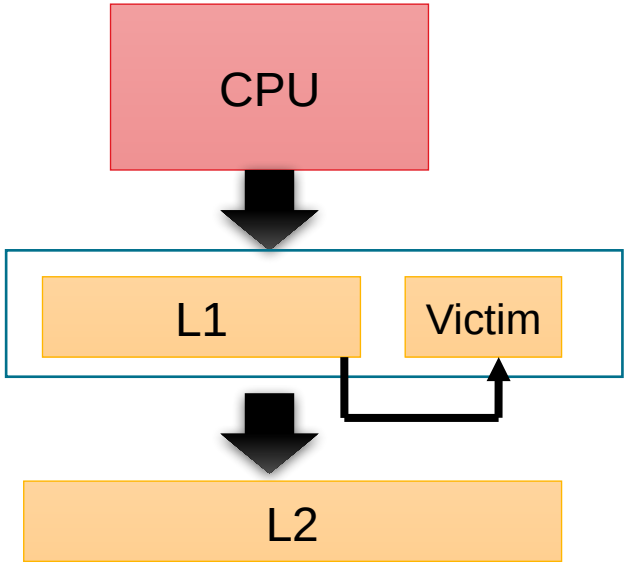
- Conflict misses

- Increase the associativity

Increases the latency and power consumption

- Victim cache

The small victim cache stores frequently evicted blocks. If a block is not found in the L1 cache, it is searched in the victim cache before searching the L2 cache.



Static Optimizations – III

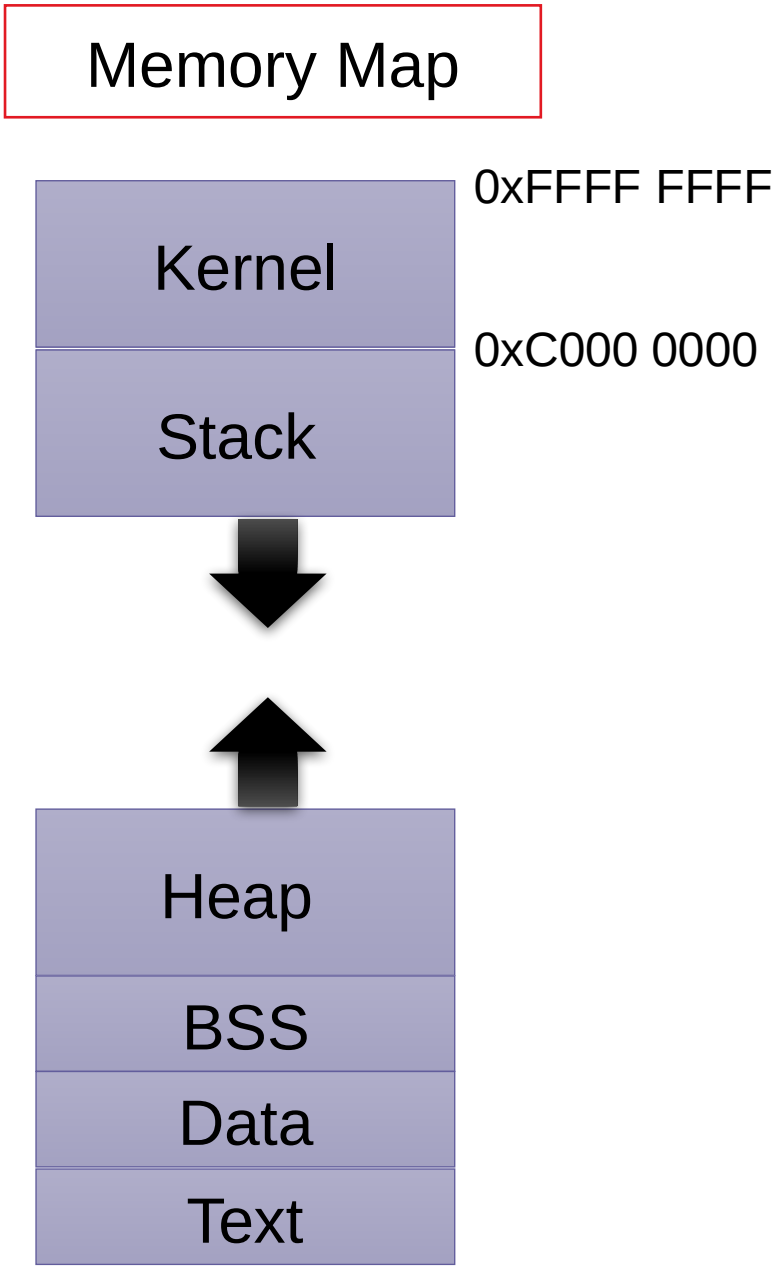
- Reduce the miss penalty
  - We typically request for 4 bytes in a 64-byte block
  - Transfer those 4 bytes first and then transfer the remaining 60 bytes subsequently (critical word first)
  - Let the processor resume when the 4 bytes arrive (early restart)

Virtual Memory

What does a process assume about the memory system?

Process ⊂ Running instance of a program.

It owns a large contiguous chunk of the memory space. Within this chunk it can access and modify any location at will.



Two Important Questions

Overlap Problem

How do we support multiple processes concurrently? A typical machine runs 100s of processes concurrently. Wouldn't they be able to access each other's data?

Size Problem

Can we access memory that is larger than the size of the physical memory?

Solution to the Size and Overwrite Problems

Virtual Memory

- The processor generates virtual addresses
- Convert virtual to actual (physical) addresses
- This solves both the problems

Address Translation

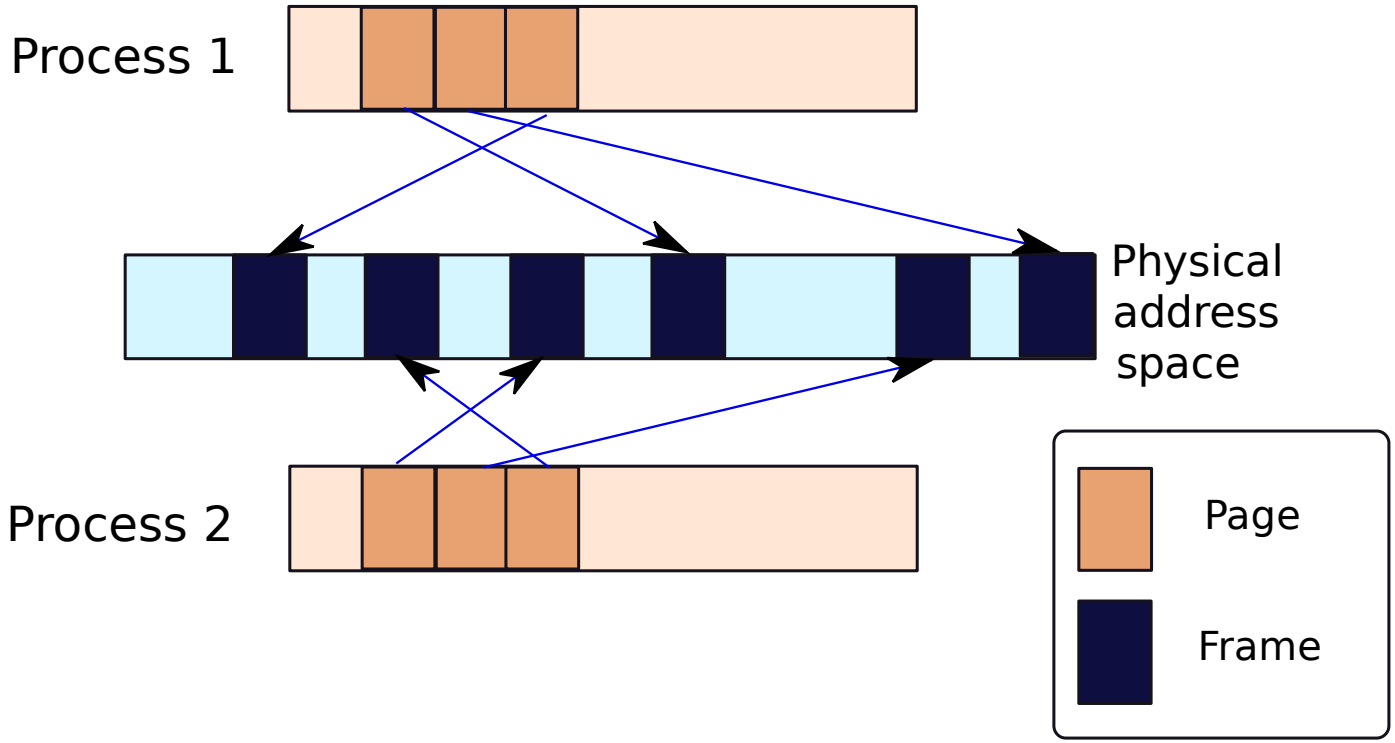
What do we need to ensure?

- Unless intended, two processes never access the same physical address
- The physical address might point to a location outside main memory (solution to the size problem)

Basic Idea

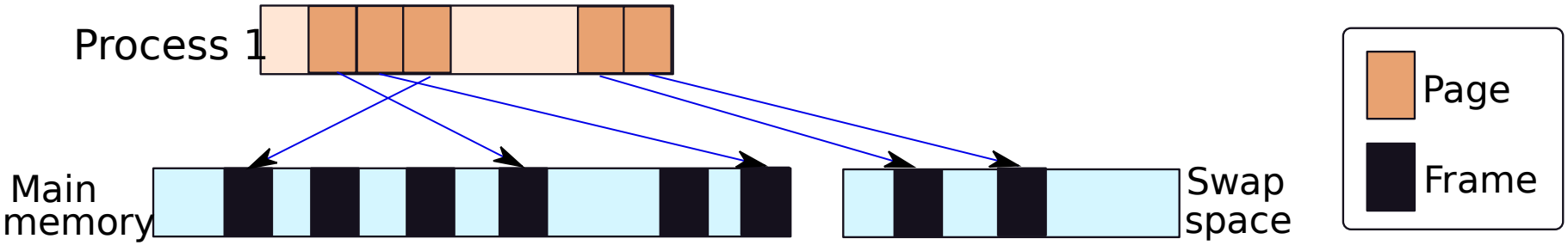
- Divide the virtual memory space (addresses that programmers and compilers see)
  - Into 4 KB pages
- Divide the physical memory into 4 KB frames
- Map pages to frames
- Ensure that the mapping solves the size and overwrite problems

Overlap Problem: Map in such a way that there are no overlaps



Map pages to frames such that there are no overlaps.

Size Problem: Create an array of frames in the hard disk called swap space

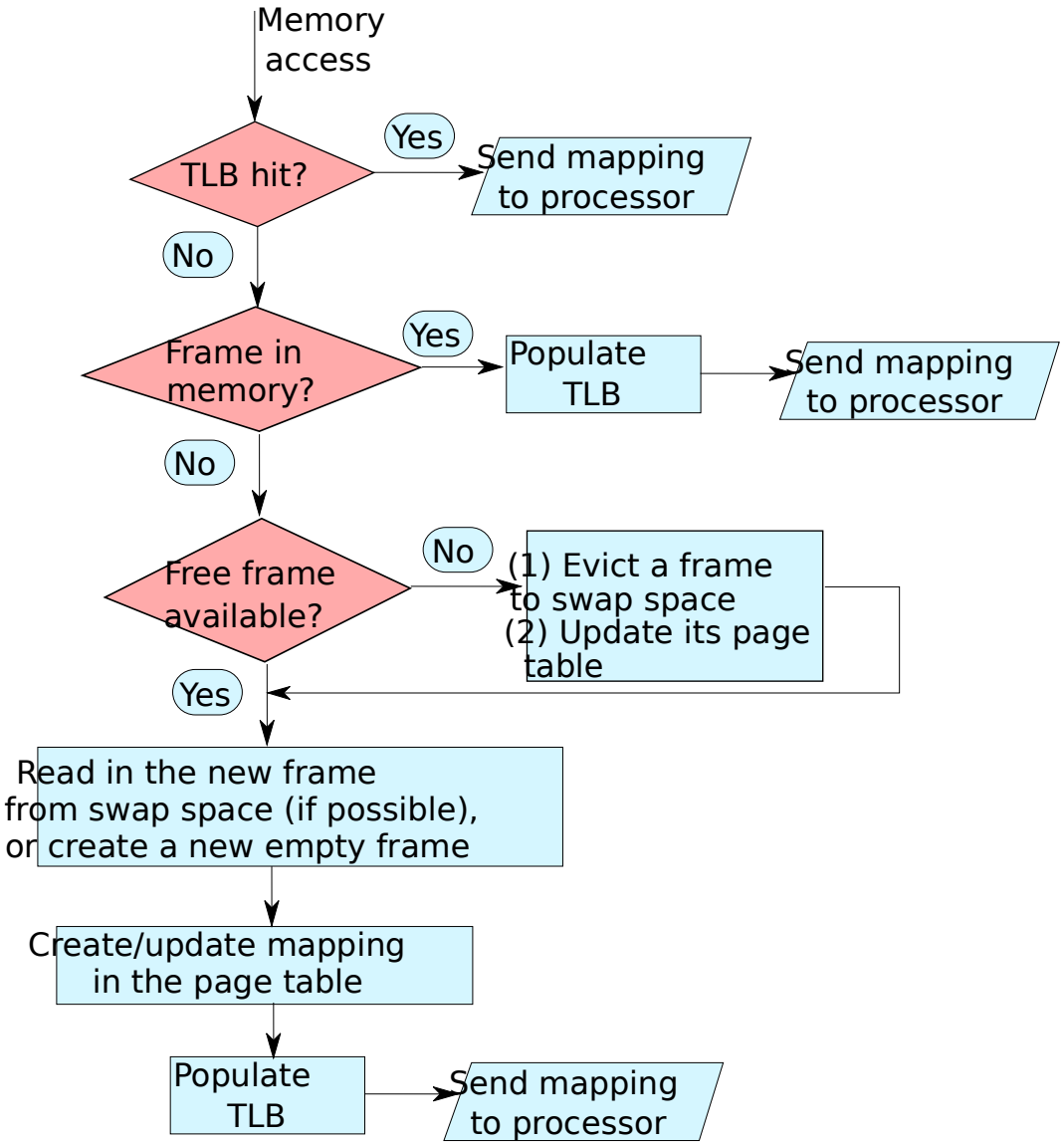


- Create an array in the hard disk or other form of stable storage. This is the swap space.
- If a page points to a frame in the swap space, we need to bring the frame to main memory.

Page Tables and TLBs

- The mappings are stored in a dedicated data structure called the page table.
- The page table takes several cycles to access. This is a slow process.
- Have a fast hardware structure called the TLB to cache the most frequent mappings
- The processor accesses the TLB first, uses the mapping if it is there. Otherwise, it accesses the page table.

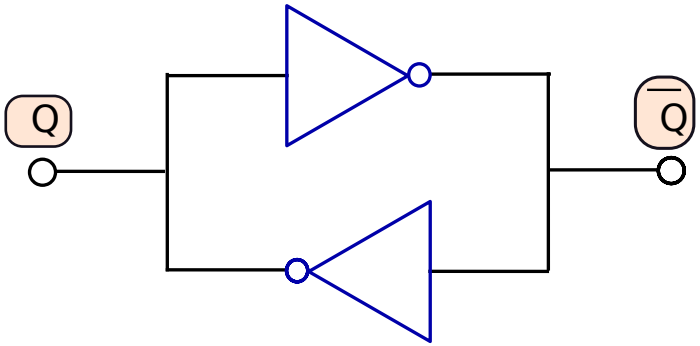
Flowchart for a memory access



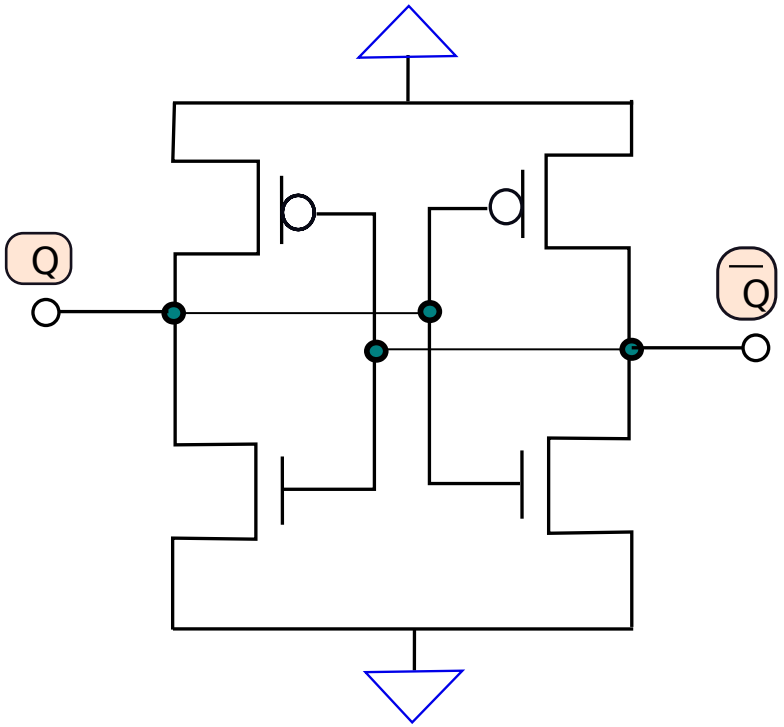
Contents


SRAM Array

Cross-Coupled Inverters

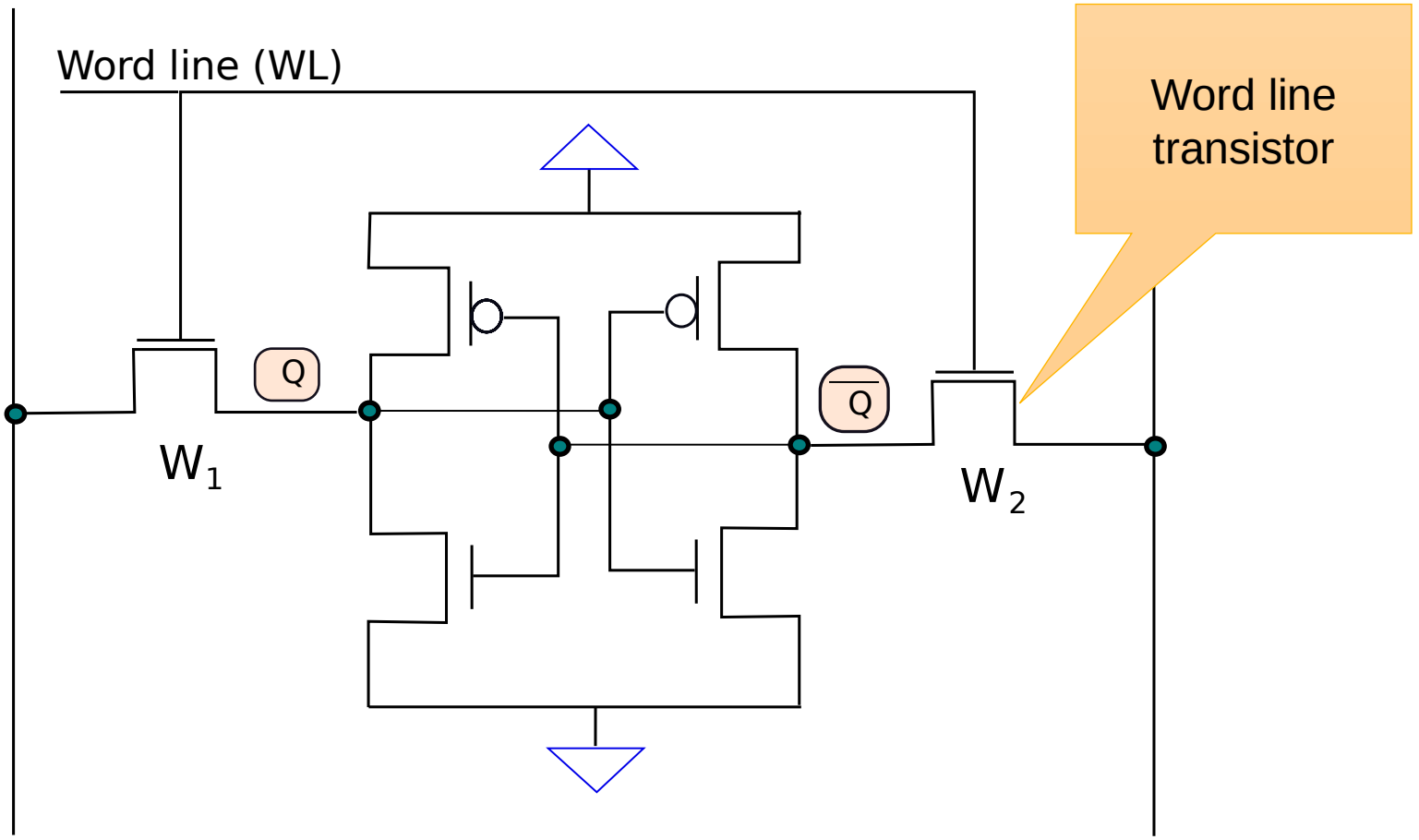


Cross-coupled inverters

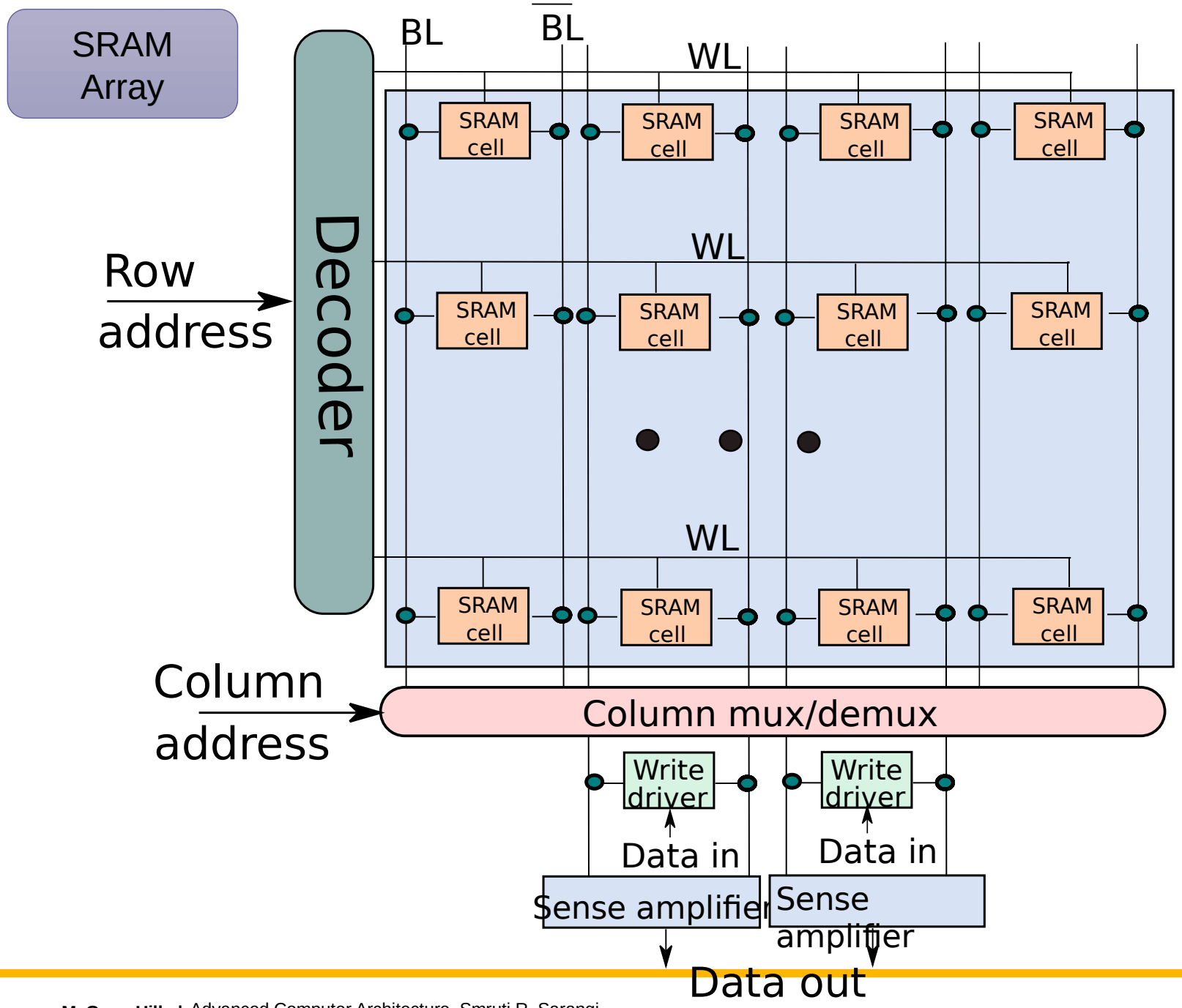


Cross-coupled inverters
implemented using CMOS logic

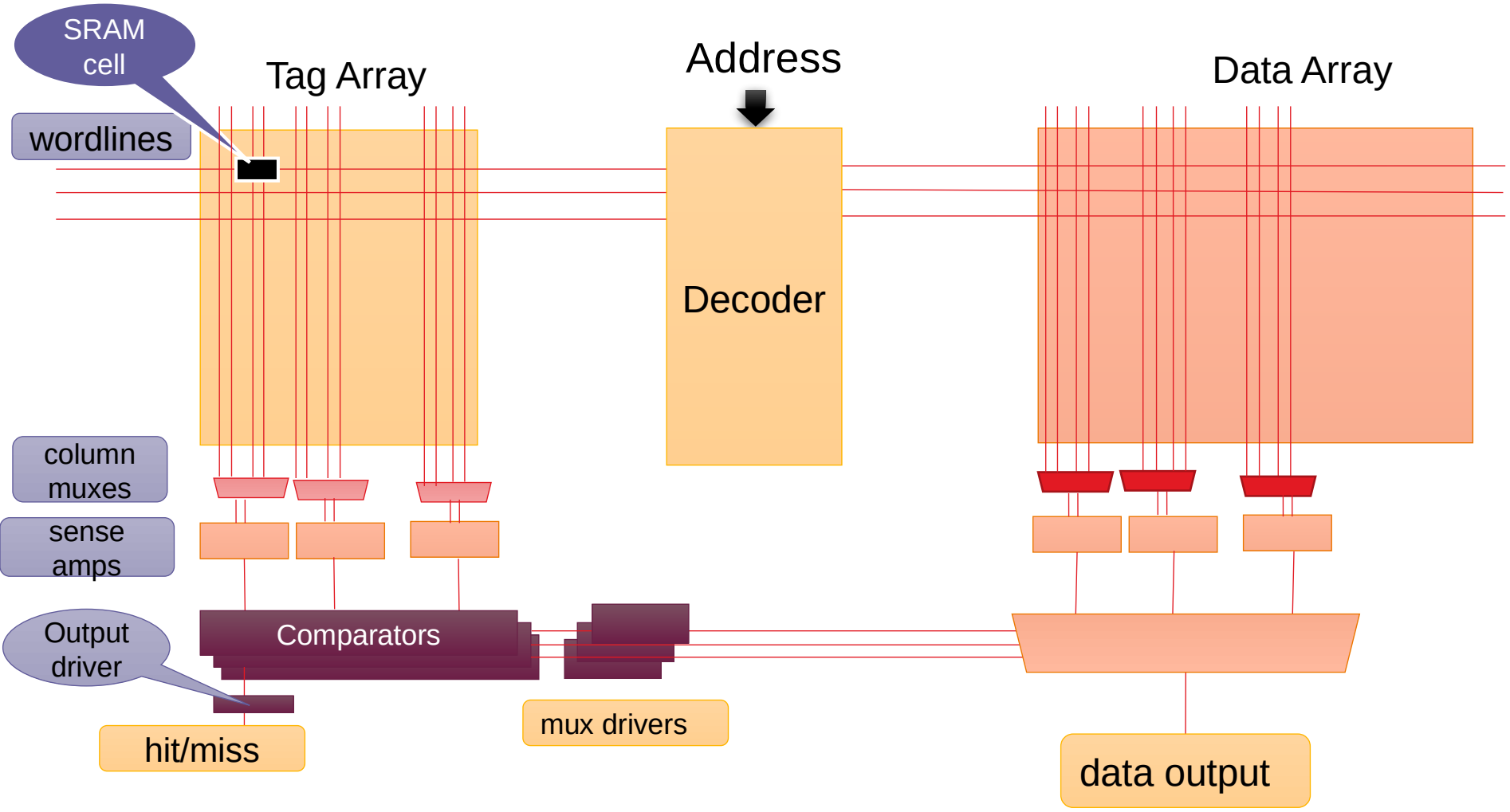
Basic SRAM Cell



Each cell is connected to a bit line pair. They have complementary values.

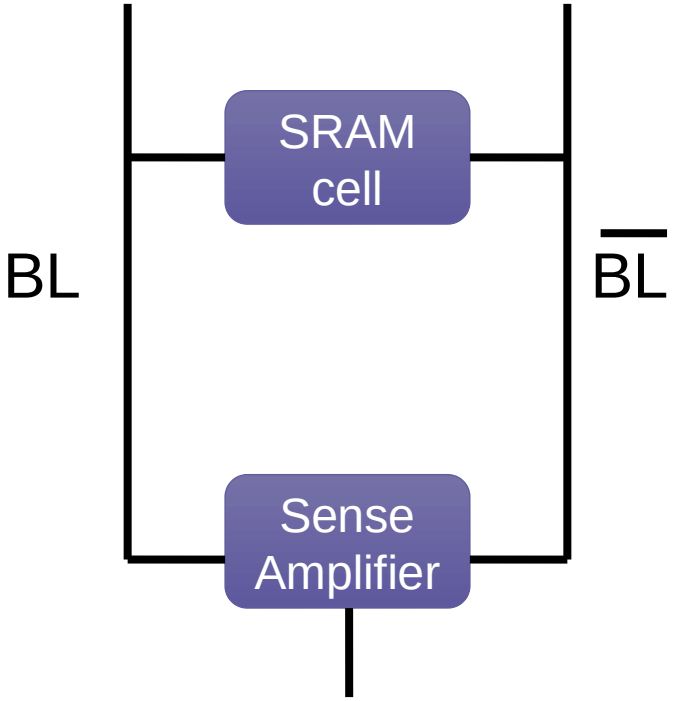

SRAM Array
BL
$\overline{BL}$
WL
SRAM cell
Decoder
Row address
Column address
Column mux/demux
Write driver
Data in
Sense amplifier
Data out

Structure of a Cache



Precharging Trick

Assume ground is 0 V and the supply is 1 V



Do we wait for one bit line to go to 0 V and the other to reach 1 V?

This will take too much time.

Precharging trick – II

To figure out which side is heavier, do we have to wait for one of the stones to hit the ground?

Keep measuring the slope of the rod. Once it is more than a threshold (noise value), declare the result.

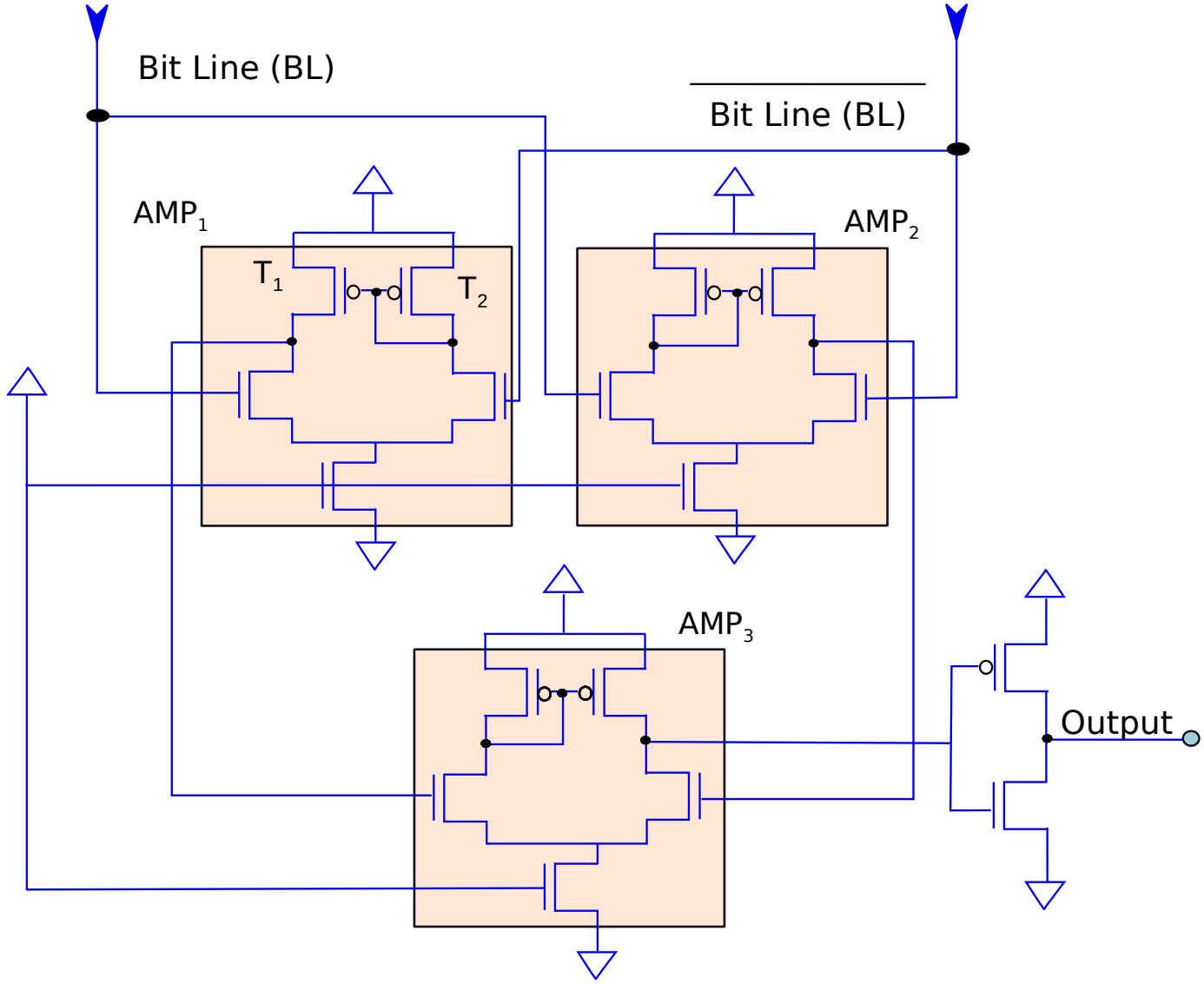
Precharging Trick – III

- We don’t have to wait for the voltage values to reach the final values.

$$value = \begin{cases} 1\,V(BL) - V(\overline{BL}) > \Delta \\ 0\,V(\overline{BL}) - V(BL) > \Delta \end{cases}$$

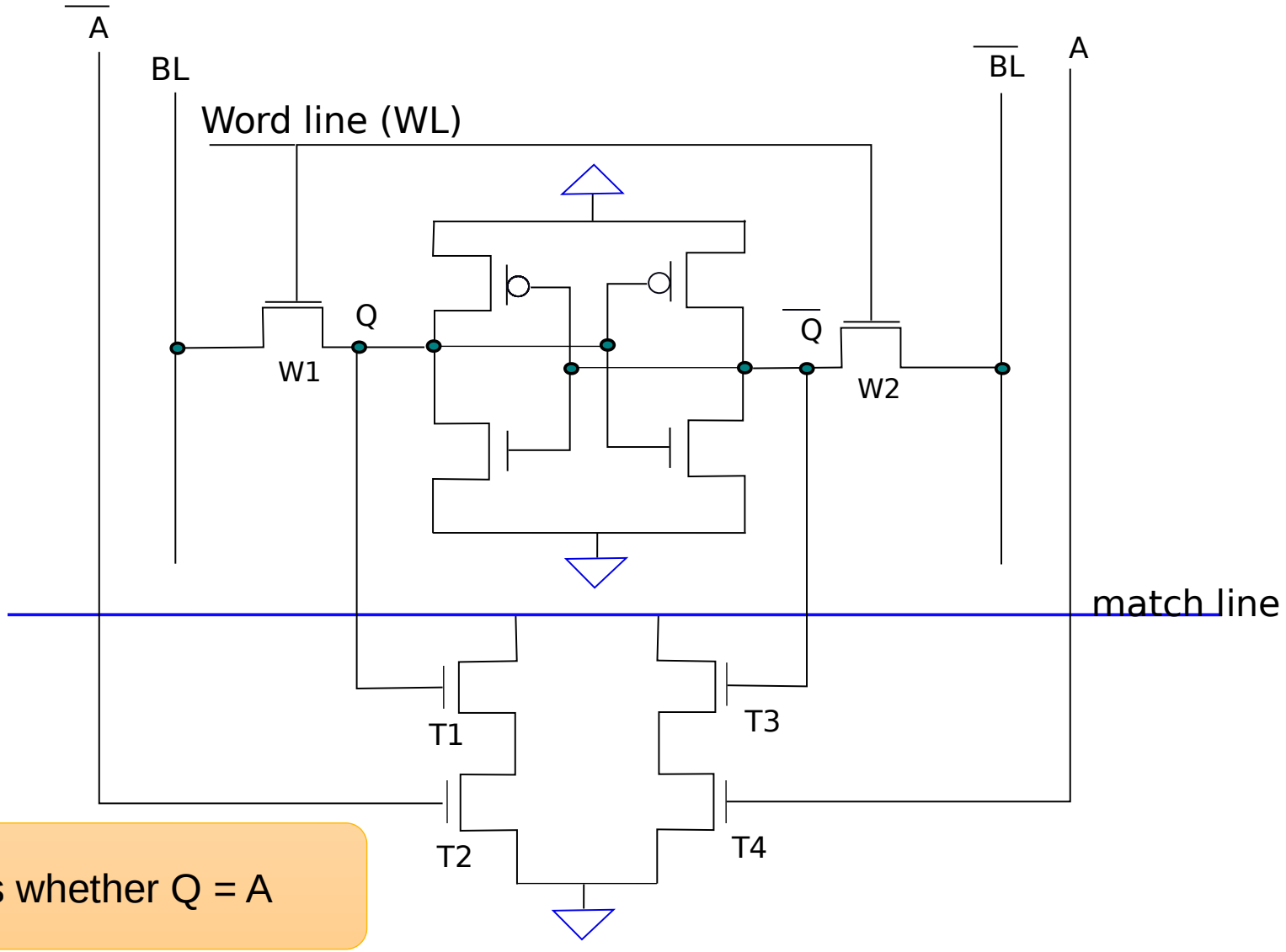
- Here is the noise margin
- Precharging step: Set the voltage of the bit line pair to 0.5 V using powerful precharge drivers
- A dedicated sense amplifier senses the difference in the voltage
- It sets the output, once the difference crosses the threshold

The Sense Amplifier




CAM Array

A CAM Cell



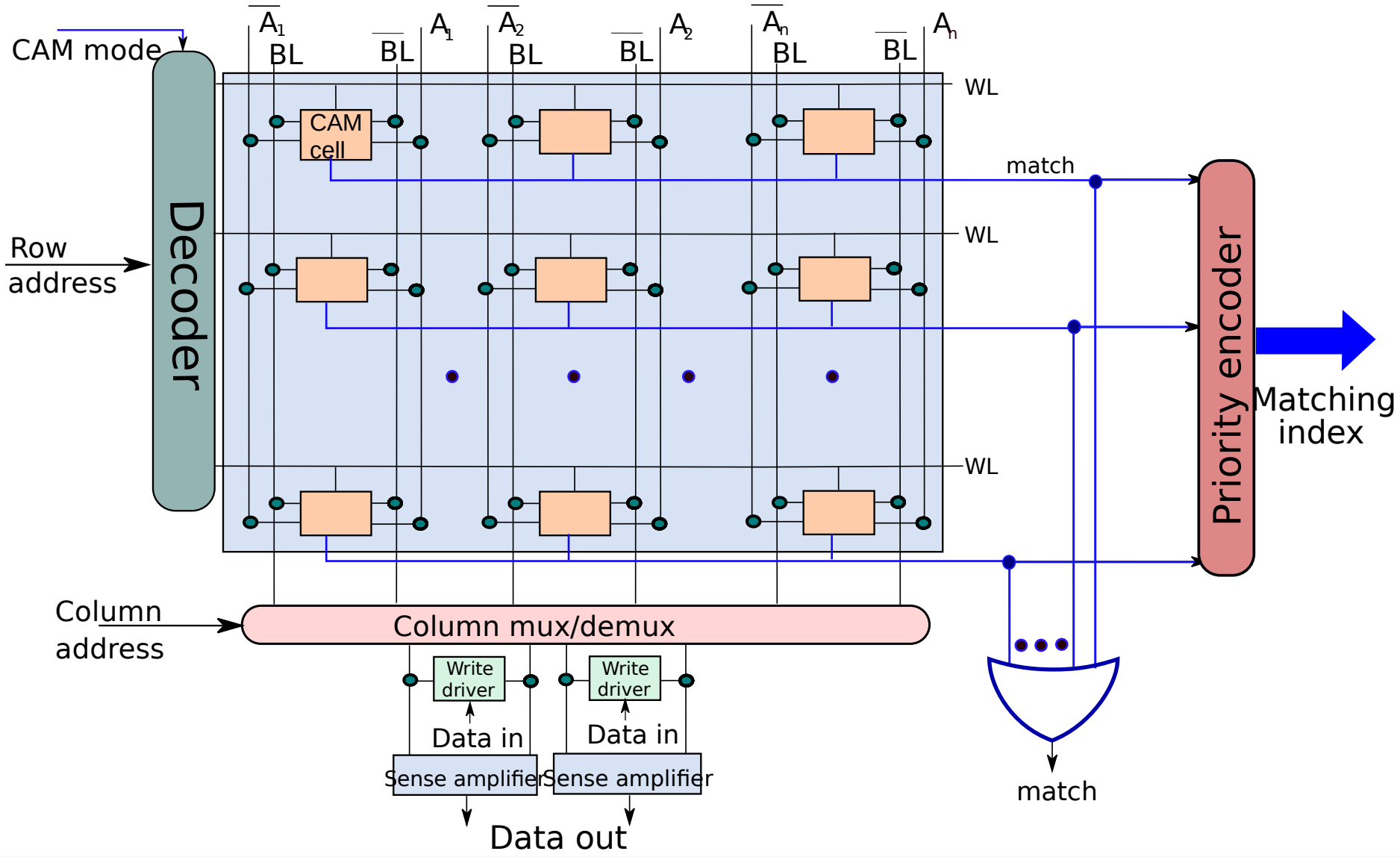
Checks whether Q = A

Truth Tables

Q		T_1	T_2
0	0	off	off
0	1	off	on
1	0	on	off
1	1	on	on

		T_3	T_4
0	0	off	off
0	1	off	on
1	0	on	off
1	1	on	on

CAM Array

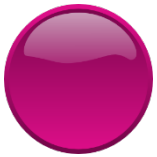

CAM mode
Row address
Decoder
$\overline{A_1}$ BL
BL A_1
$\overline{A_2}$ BL
BL A_2
$\overline{A_n}$ BL
BL A_n
CAM cell
WL
WL
WL
match
Priority encoder
Matching index
Column address
Column mux/demux
Write driver
Write driver
Data in
Data in
Sense amplifier
Sense amplifier
Data out
match

The Cacti Tool

Modeling and Simulating Caches

Should I use a 4 KB, 2-way assoc. cache or an 8 KB, 1-way assoc. cache? Should I allow concurrent accesses?

- First, we need to find the access times of both caches
- Convert access time into clock cycles
- Simulate both the configurations: find the faster one



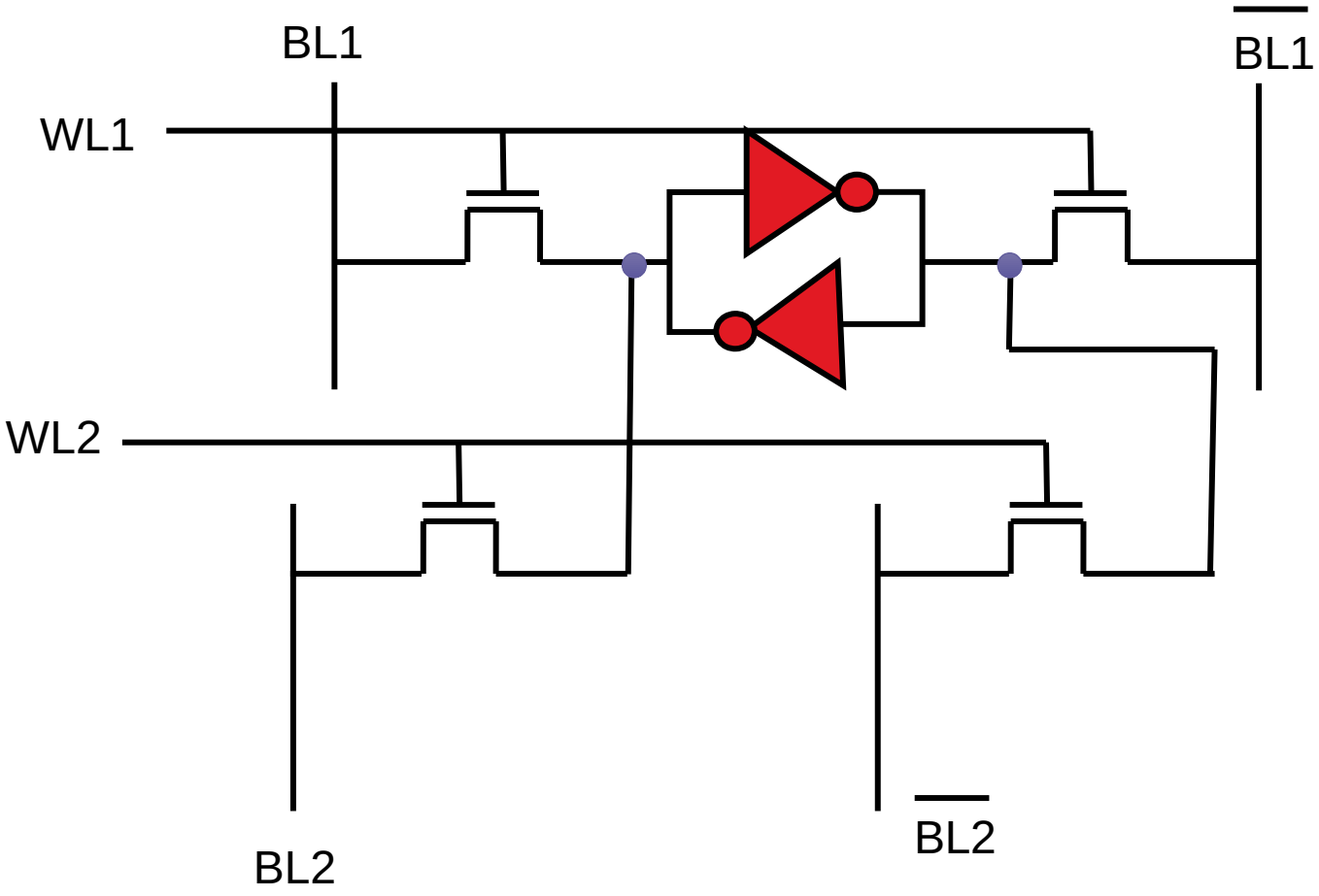
To find the time it takes to access a cache

- We need to use a simulation tool
- Most popular simulation tool ⎵ Cacti (designed by HP labs)

Along with that

- We need power and area data as well
- Cacti 6.0 provides all of these

Idea that will not work: Multi-ported structures



Re-design the SRAM cell ⇒ slower and larger

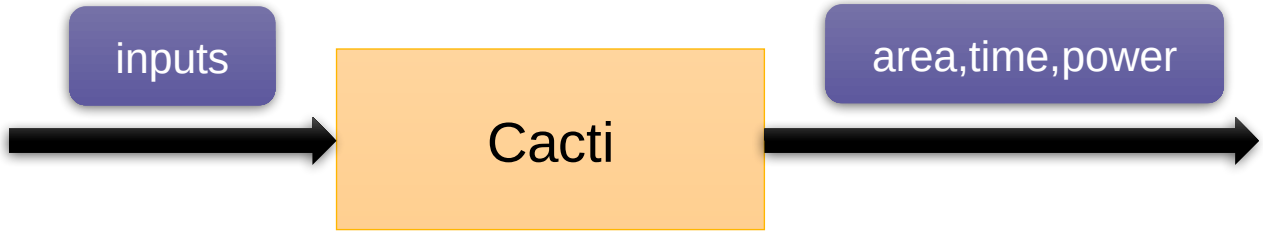
Usage

cacti C B A

- C ▭ cache size in bytes
- B ▭ block size in bytes
- A ▭ associativity
- Number of banks (a bank is a sub-cache) (more later ..., assume 1 for now)

Hidden inputs

- b_o ▭ output width (32 or 64 bits)
- b_{width} ▭ input address width (32 or 64 bits)



Sample input and output

Normal Interface
Detailed Interface
Pure RAM Interface
FAQ

Cache Size (bytes)	16384
Line Size (bytes)	64
Associativity	4
Nr. of Banks	1
Technology Node (nm)	32

Submit

Cache Parameters:

Number of banks:1
Total Cache Size (bytes):16384
Size in bytes of bank:16384
Number of sets per bank:64
Associativity:4
Block Size (bytes):64
Read/Write Ports per bank:1
Read Ports per bank:0
Write Ports per bank:0
Technology Size (nm):32
Vdd:0.9

Access time (ns): 0.614536798032
Random cycle time (ns):0.22095719381
Multisubbank interleave cycle time (of data array) (ns):0.211907965098
Total read dynamic energy per read port(nJ): 0.0779954990649
Total read dynamic power per read port at max freq (W): 0.352989181841
Total standby leakage power per bank (W): 0.00835633730771
Refresh power (percentage of standby leakage power): 0.0
Total area (mm^2): 0.326190837722

How does Cacti work?

- Given the inputs, it tries to create the most optimal (often fastest) cache
- What can happen?
- Caches can get very asymmetric. rows >> columns, or columns >> rows (>> means significantly more than)

How to make it fast?

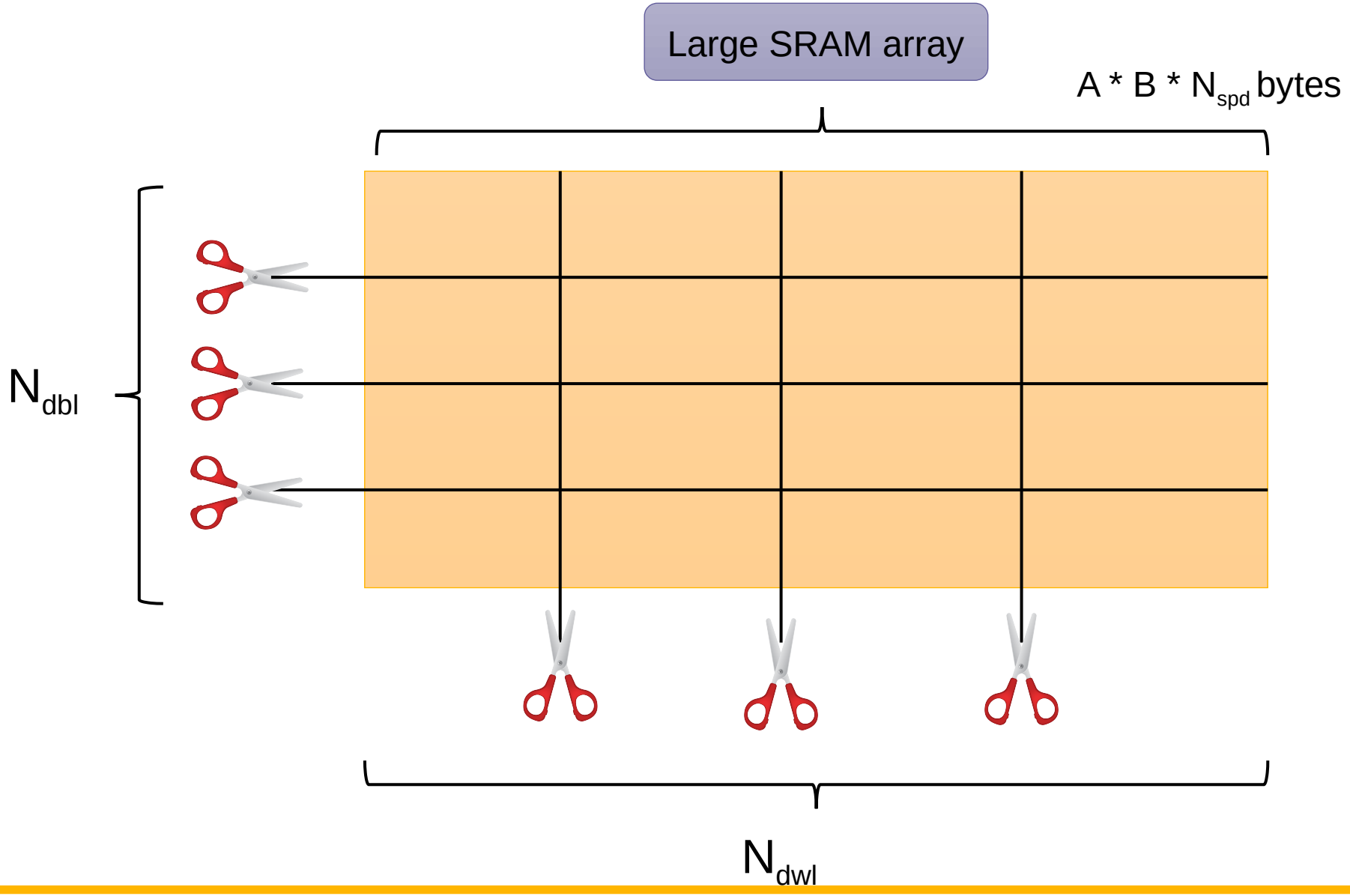
- Set the aspect ratio first:
  - Number of sets mapped to a wordline ▭ N_{spd}
  - A higher N_{spd} will increase the number of column multiplexers ▭ slowdown
  - However, it will decrease the number of rows, and thus the size of the address decoder ▭ speedup

Then create subarrays of SRAM cells

- N_{dwl} ▭ Number of vertical cut lines
- N_{dbl} ▭ Number of horizontal cut lines
- Total number of subarrays created ▭ $N_{dwl} * N_{dbl}$

The optimal values of N_{dwl} , $N_{dbl,}$ and N_{spd} are found out by Cacti

Partitioning an SRAM array



Banks and Subarrays

A large array can be split into multiple banks

- A bank is an independent array.
- Different banks can be accessed simultaneously
- Given an address, we first need to map it to a bank, and then access that specific bank
- Advantage of banking ⇨ faster banks and more parallel access
- Disadvantage of banking ⇨ More area, and additional delay in deciding the bank. Higher wire routing overhead.
- We split each bank into multiple subarrays
  - Compute the three parameters: N_{dwl}, N_{dbl}, N_{spd}

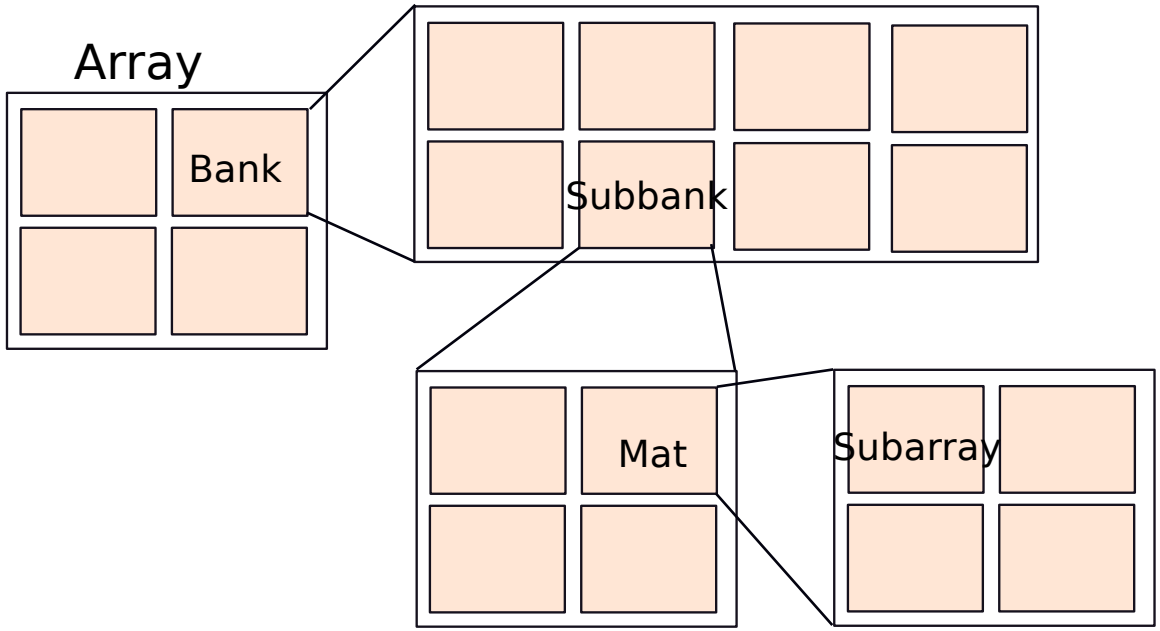
Arrange the Subarrays in a Hierarchy

Each bank can then be split into subbanks

- Subbanks do not allow parallel (concurrent) access
- A subbank stores a full block (Advantage: read the block in one go)

Each subbank is further divided into mats and subarrays

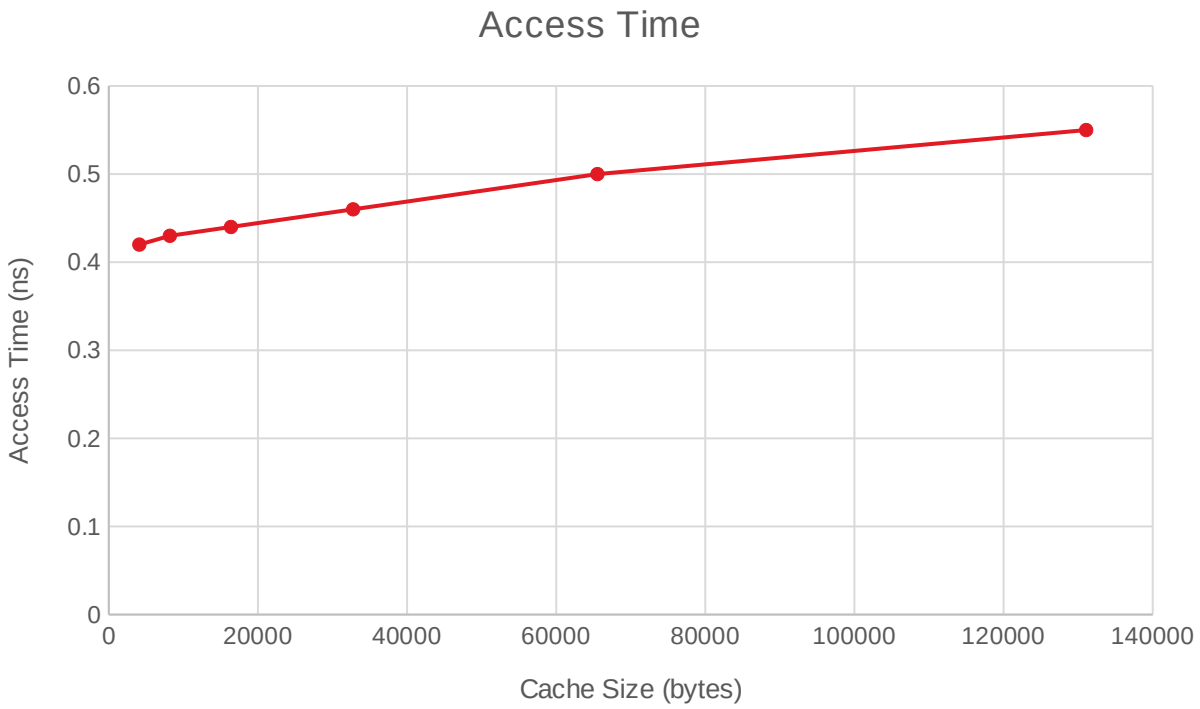
- Each mat supplies a portion of a block (enhanced parallelism)
- Subarrays in a mat share a decoder or predecoder



Summary

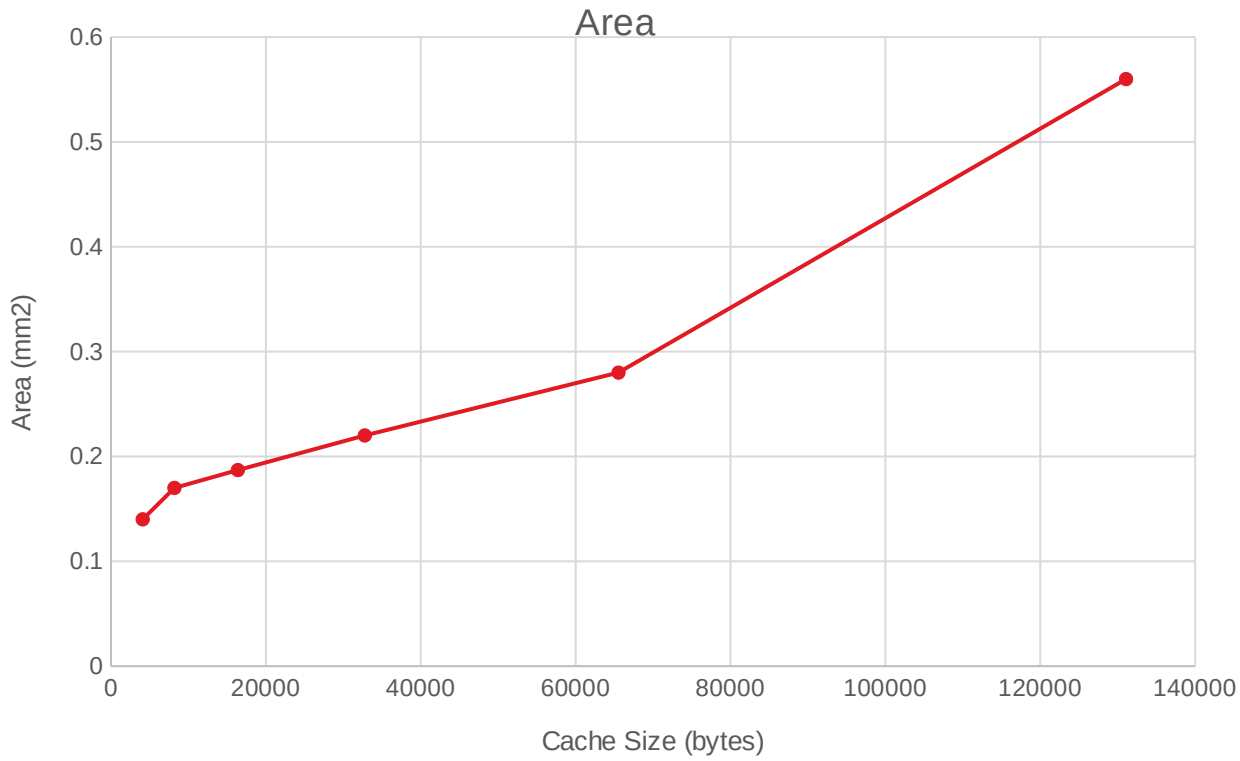
- First set the aspect ratio: N_{spd}
- Then compute N_{dwl} and N_{dbl} ▭ Total $N_{dwl} * N_{dbl}$ subarrays
- Divide the cache into equal-sized banks
  - Keep in mind the probability of bank conflicts
  - A bank is a fully independent array with its separate address and data lines
- Hierarchy: bank ▭ subbank ▭ mat ▭ subarray
- subbanks do not allow independent or concurrent accesses
- Enhanced intra-block parallelism
  - A mat stores a part of a block. It contains multiple subarrays.
  - subarrays share their decoding logic in a mat.

Examples: (2-way assoc, 64B line size, 1 bank, 32 nm)
Access Time vs Cache Size



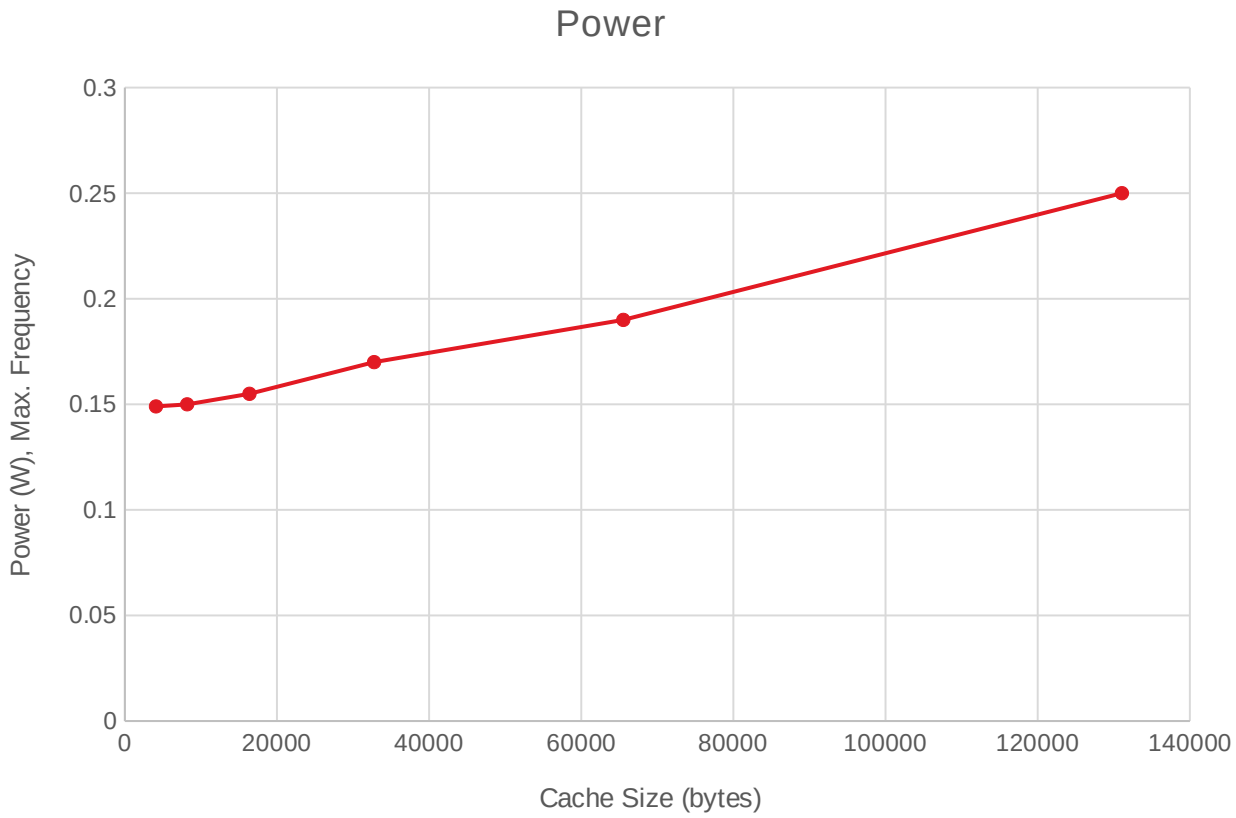
Little variation. Because of Cacti's optimizations.

Area vs Cache Size



Roughly linear scaling

Cache Size vs Power



Little variation. Because of Cacti's optimizations.

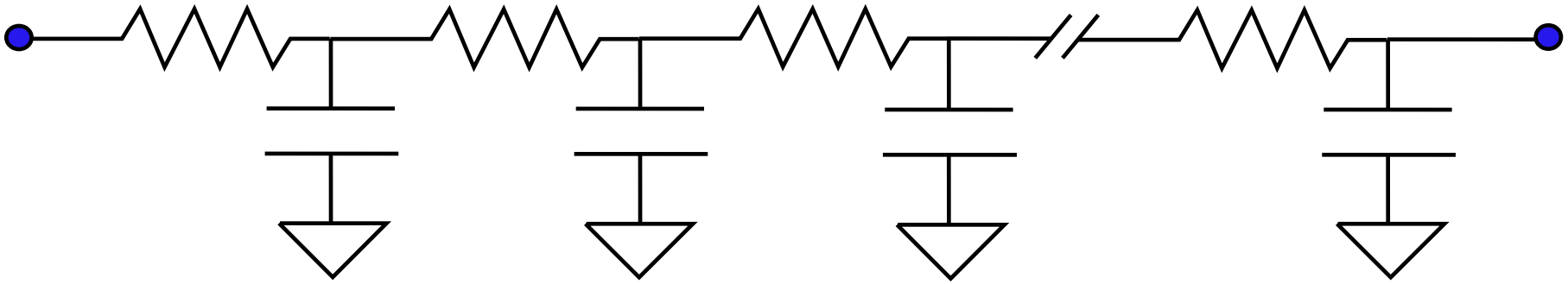
Elmore Delay Model

Time and Power Modelling

How to compute the delay of a cache?

Replace it by its equivalent RC model.

RC model of a wire



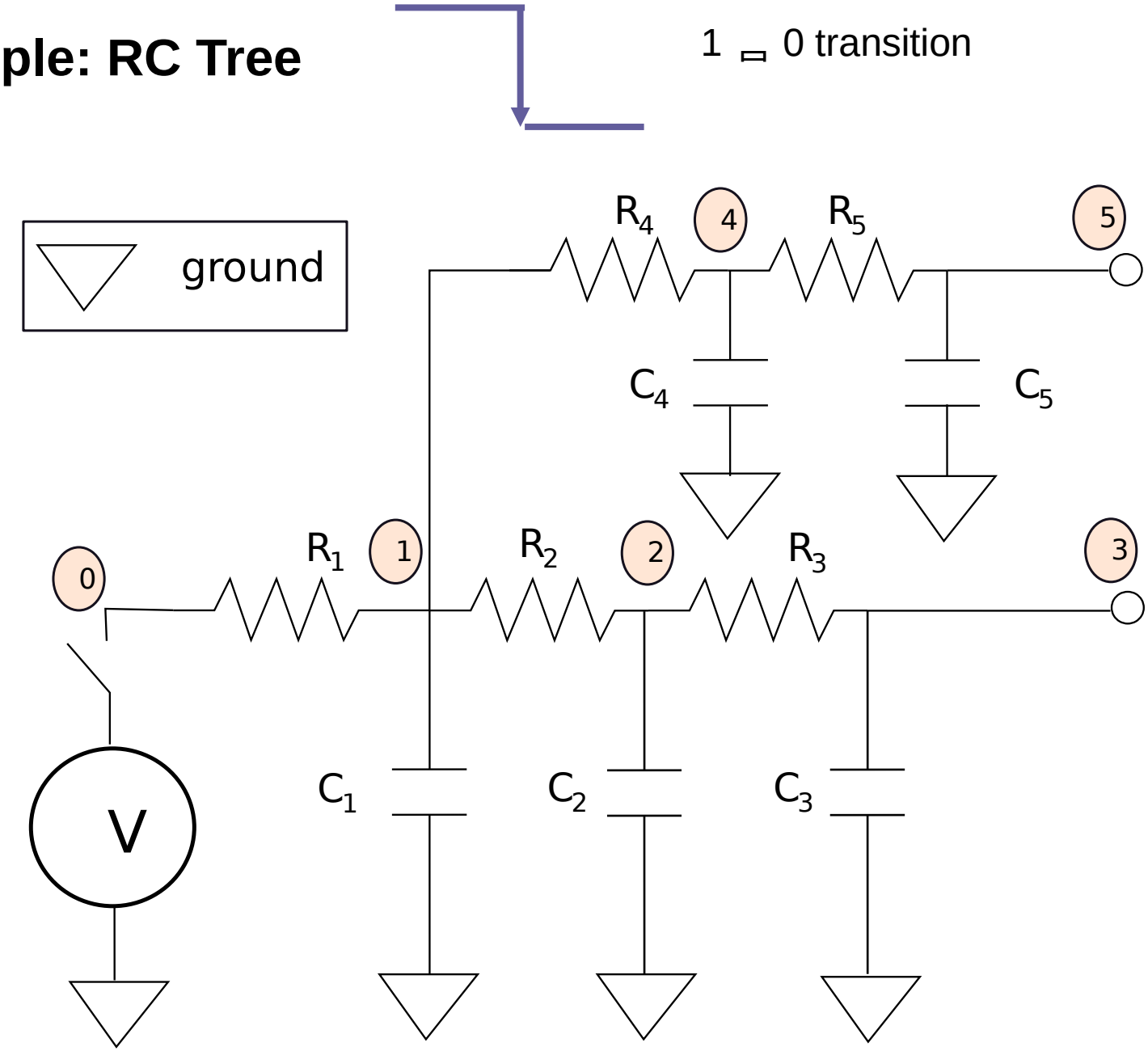
Assumptions

Let us only consider RC trees

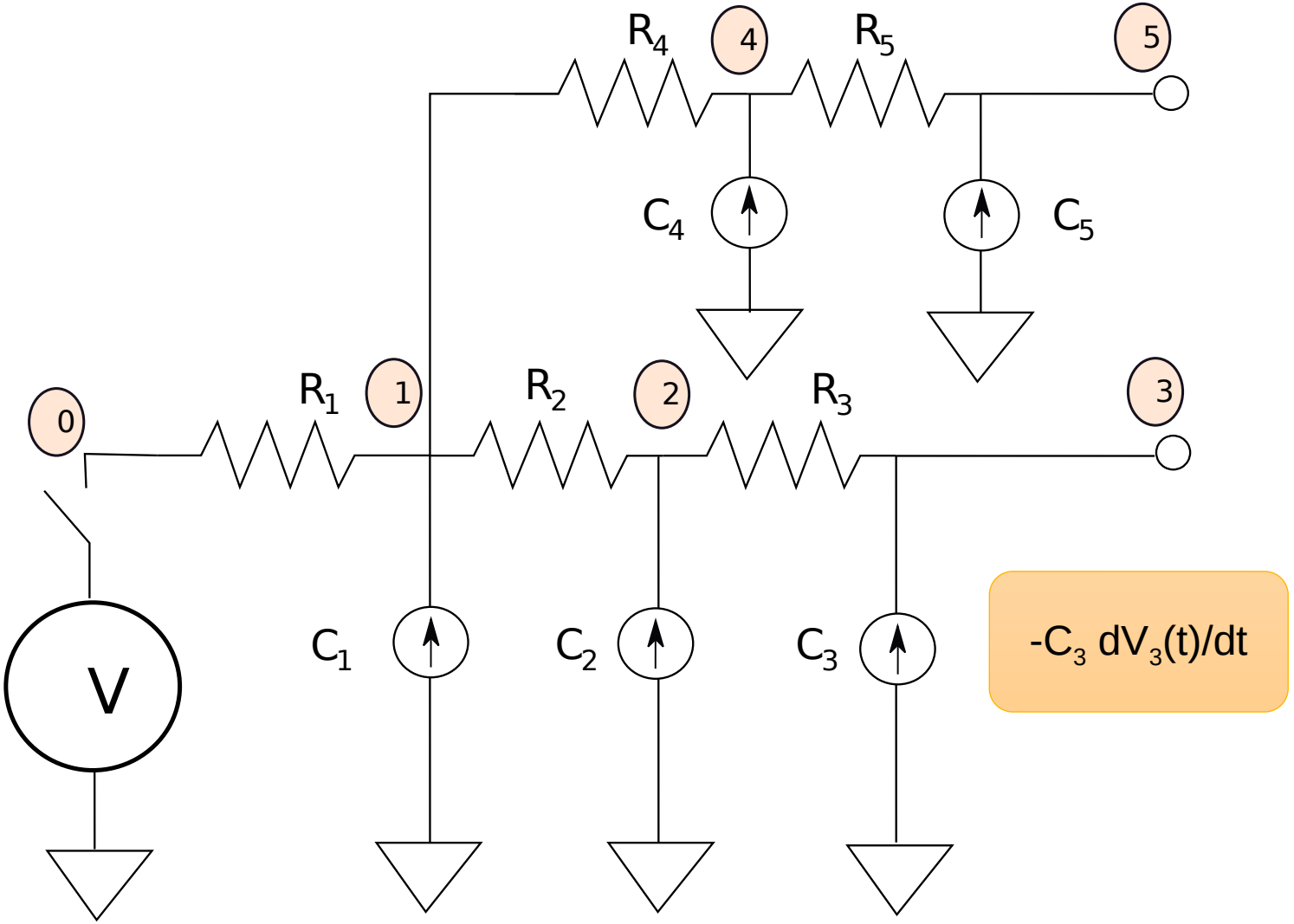
We only consider voltage sources that provide step inputs. 0 $\sqcup$ 1 or 1 $\sqcup$ 0

Elmore Delay Model

Example: RC Tree



Replace Capacitors with Current Sources

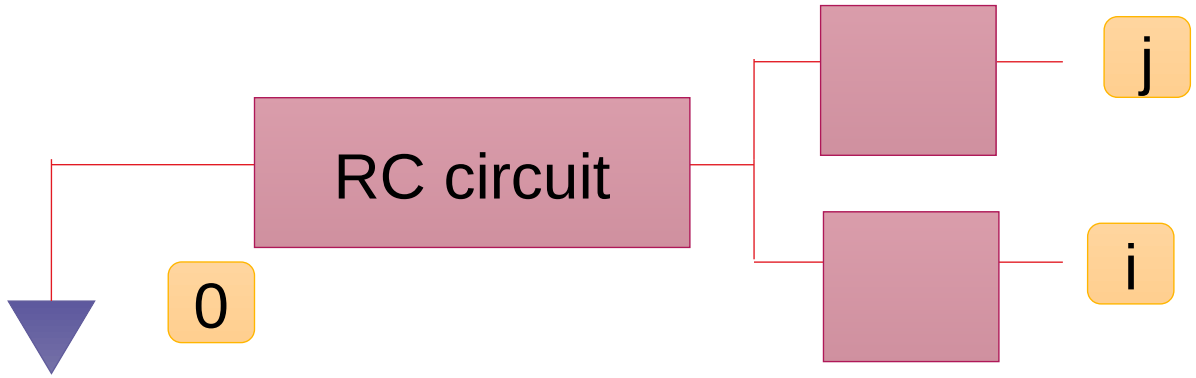


Principle of Superposition

- AIM: Compute the voltage at node V_x at time *t.*
- Methodology:
  - The aim is to find ▭ voltage at terminal *i* because of the current source placed at terminal *j*
  - While considering one current source, disconnect the rest of the current sources (replace by an open circuit)

$$V_3^4 = -R_1 C_4 \frac{d V_4^4}{dt}$$

Generalize the Result



- Let P_{ij} denote the RC circuit in the shared path from from 0 ⎵ i and 0 ⎵ j

Definition

$$R_{ij} = \sum_{R \in P_{ij}} R$$

Elmore's Assumption

$$V_i^j = -R_{ij} C_j \frac{dV_j^j}{dt}$$

Superposition

$$V_i = \sum_j V_i^j = -\sum_j R_{ij} C_j \frac{dV_j^j}{dt}$$

Elmore's Assumption

$$\frac{dV_j^j}{dt} = \frac{dV_i}{dt}$$

Using Elmore's Approximation

$$\tau_i = \sum_j R_{ij} C_j$$

Solution

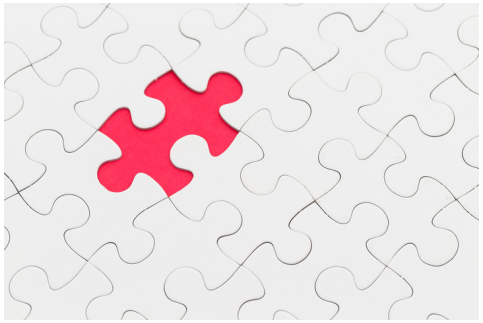
Results Derived by using the Elmore Delay Model

- The latency of a wire is proportional to the square of its length.
- The bit line and word line can be modelled very easily.
- Full SRAM banks can be modeled as a sequence of simple elements.

Contents

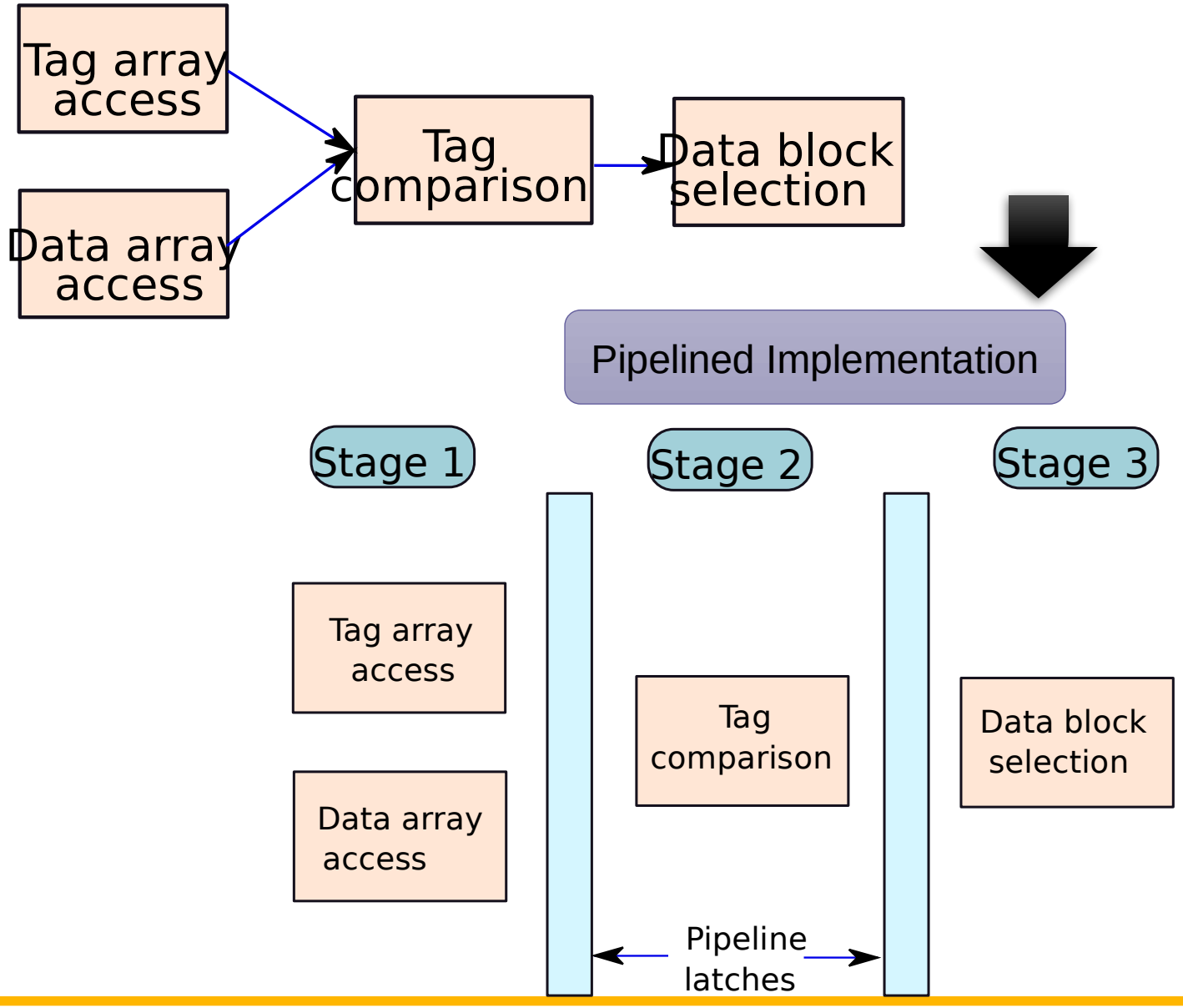
Pipelined Caches

- Let us say that a cache takes 5 cycles to access.
- Do we wait for 5 cycles for the next access?



- Pipeline the cache.
- Even if the latency is high, we can achieve a high throughput.

Tasks Involved in a Read Operation



More on Pipelining

- We simply add new pipeline buffers
- The SRAM access can be pipelined by adding latches after the decoder
  - This is very expensive in terms of area and power.
- The write access can be pipelined
  - Access tag array ⎵ Compare the tags ⎵ Effect the write in the data array

Non-Blocking Caches

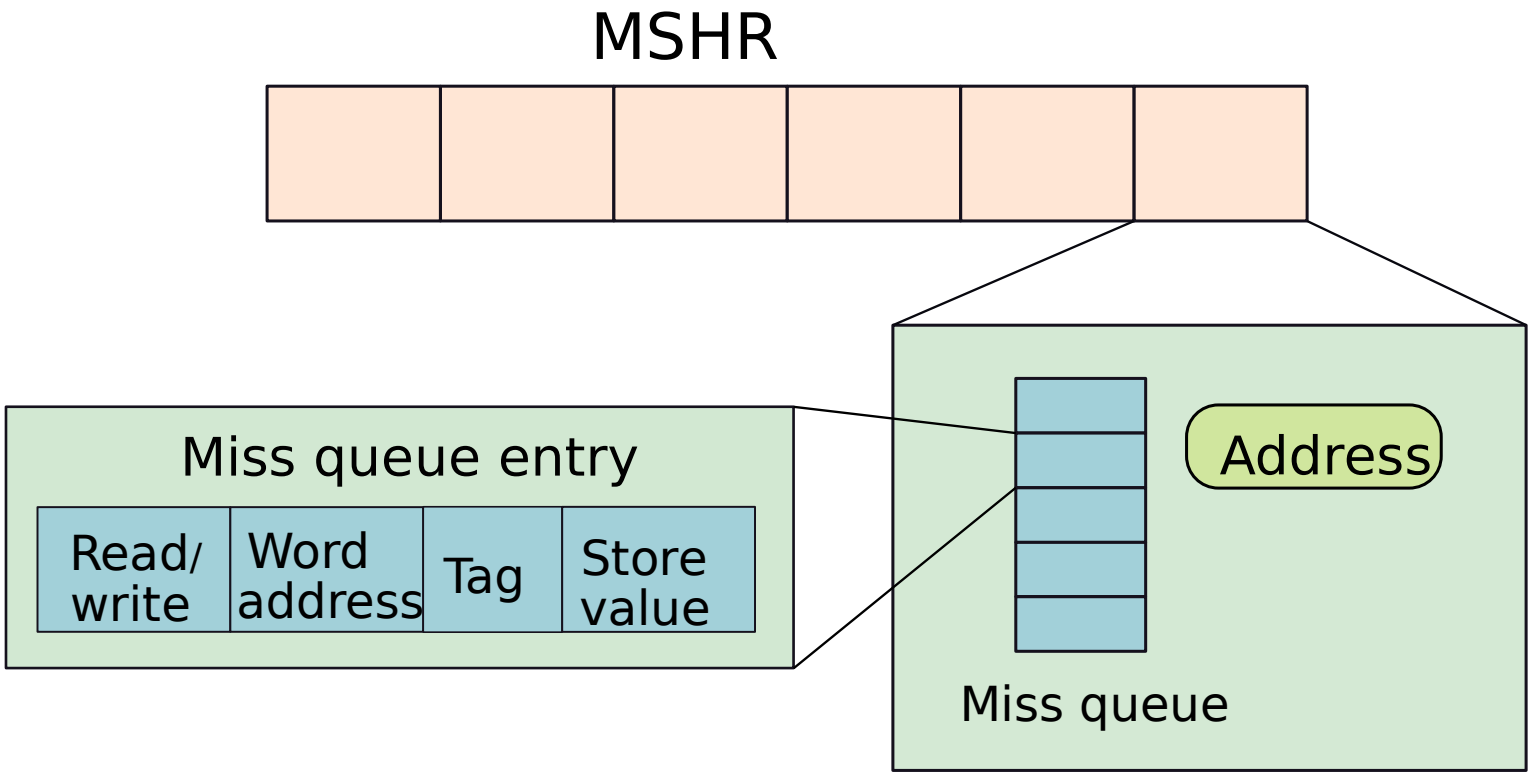
- Let us say that there is a cache miss.
  - Do later cache access stall?

- Cache accesses need not be blocked. We simply need to record all accesses to the missed block.

MSHR

Miss Status Holding Register

Design of an MSHR



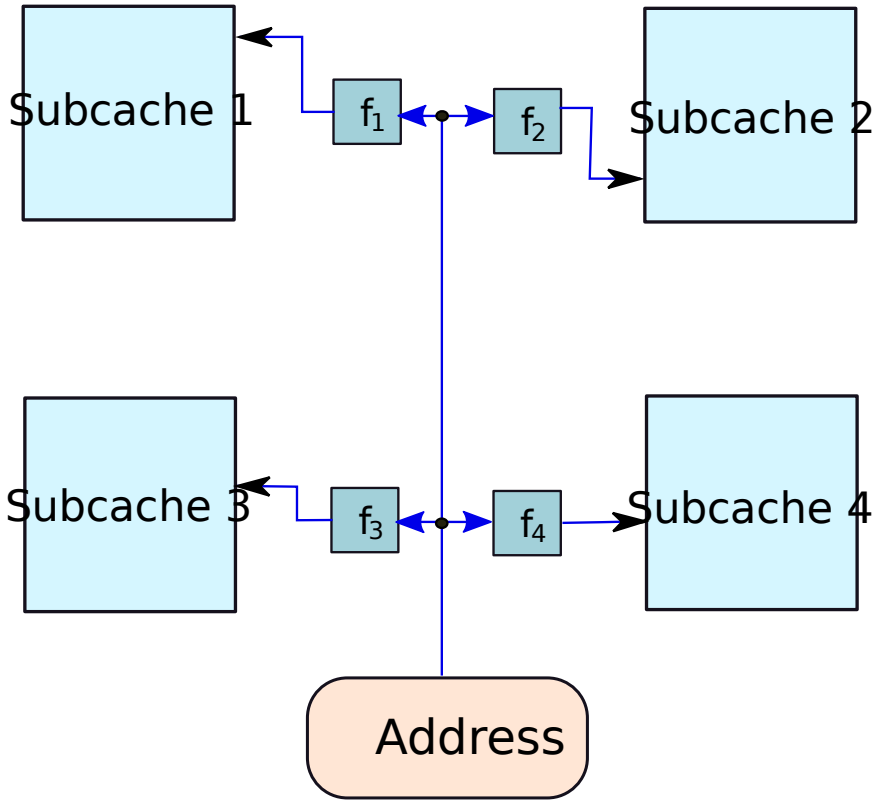
For every address we have an associated miss queue ⎵ Ordered in FIFO order

Operation of an MSHR

- When there is a miss in a cache, an entry is created in its MSHR. If there are no free entries, the request blocks.
  - We also create an entry in the miss queue.
  - This is a primary miss.
- If at the same time, another miss arrives, it will be called a secondary miss.
  - If it is a write, we just append it to the tail of the miss queue.
  - If it is a read, we search for earlier writes in the miss queue to the same set of bytes. If we find an entry, the value is forwarded. Else, we enqueue the entry.

Skewed Associative Caches

- Conflict misses are the main problem
- Solution:
  - Have multiple subcaches
  - Use different indexing functions for each subcache



Basic Insight

Consider two addresses

- If most likely
- If they conflict in one subcache most likely they will not conflict in another subcache
- Reduce conflict misses

Overheads

- These functions need to be computed
- Replacements are tricky

Replacements

Cuckoo search

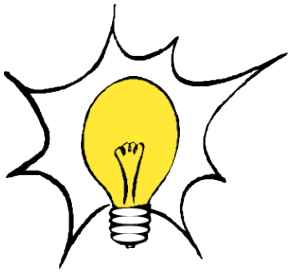
Assume that we want to add the block *B* at address *A*

- We will first check if ... are free
- If not, we evict the block *B'* with address *A'* stored at . We insert block *B* in the first subcache at
- We try to accommodate *B'* at or or
- If none of these locations are free, we choose one of the locations randomly. Let's say, it is We insert *B'* there by evicting block *B''*. Again we try to accommodate *B''.*
- We repeat this process a few times before finally giving up and evicting a block.

Way Prediction

- Default mechanism in set-associative caches
  - Read all the tags and blocks in a set (**simultaneously**)
  - Compare the tag part of the address with each of the tags

- We do more work that what is required.



- Predict the way in which the tag will be found
- Access that way first
- If the tags don't match, access the rest of the ways
- Fast and power-efficient

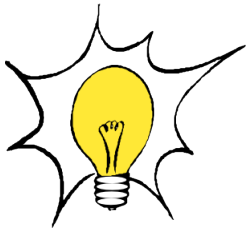
Way Prediction Techniques

If the address is available use it, else use some other piece of information.

Approach 1

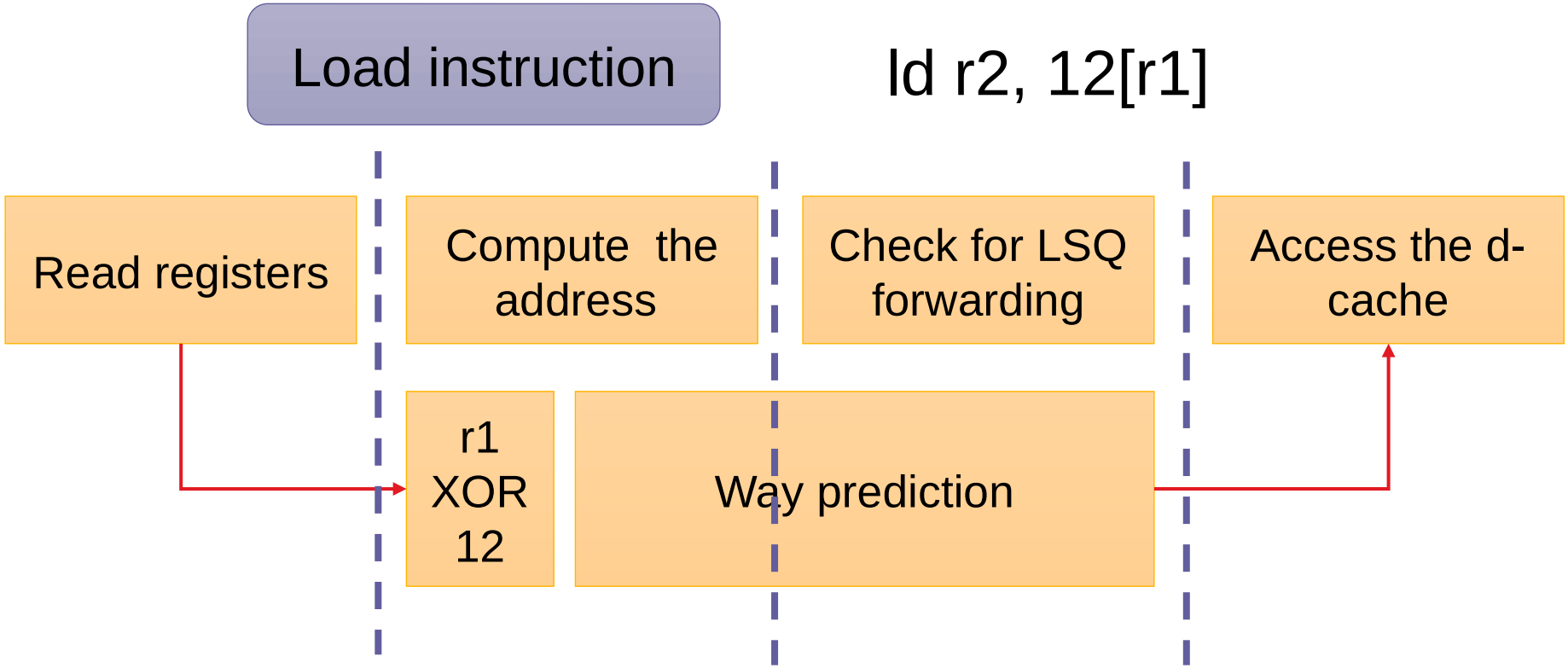
- Predict based on the program counter
- Use a branch predictor-like mechanism
- Problems: Memory addresses for an instruction might keep changing

Approach 2



Use an estimate of the memory address to predict the way

Way Prediction Techniques – II



- Combine the value of the base address with the offset (like GShare). We may use additional PC bits
- Then use a branch predictor-like mechanism to predict the way

Loop Tiling

How do we run large loops that access large arrays on a system with caches?

How is a 2D array stored?

- Row-major order: Store the first row, then the second, then the third and so on.
- Column-major order: first column, second column, and so on

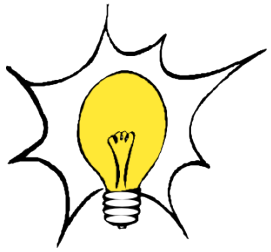
Basic Matrix Multiplication

Matrix Multiplication

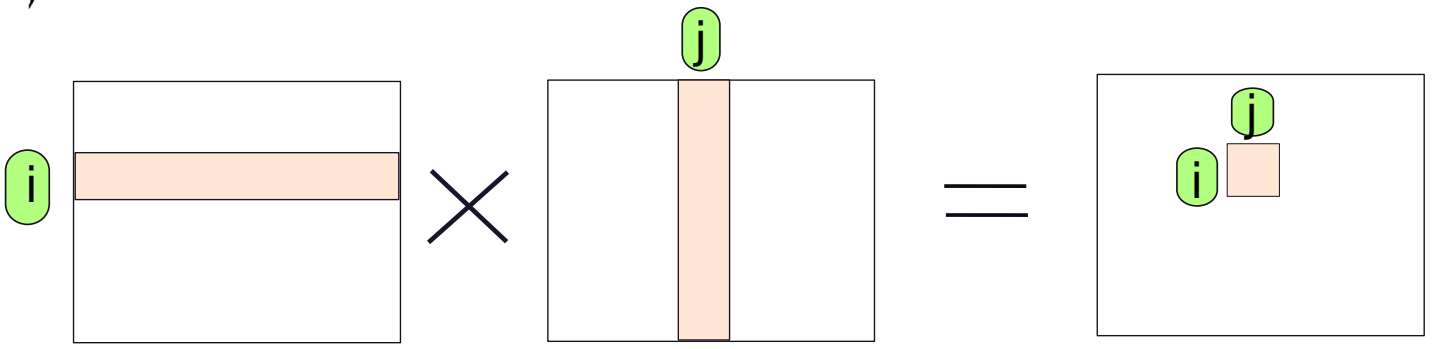
```
for (i=0; i<N; i++) {
        for (j=0; j<N; j++) {
                sum = 0;
                for (k=0; k<N; k++)
                        sum += A[i][k] * B[k][j];
                C[i][j] = sum;
        }
}
```

Not suitable when arrays are large, and caches are comparatively much smaller.

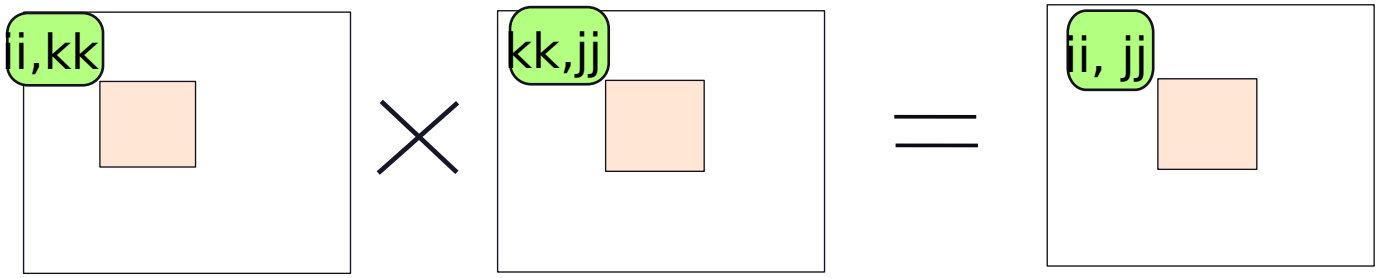
Tile the Multiplication



Operate on blocks of data



(a)

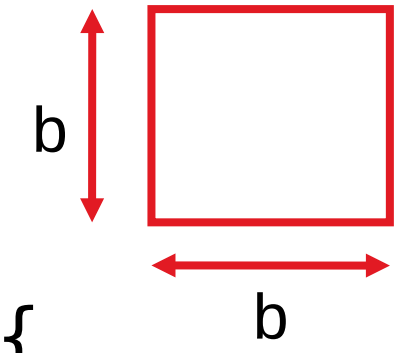


(b)

Tiled loops

```
/* Iterate through the tiles */
for (ii =0; ii <N; ii +=b) {
    for (jj =0; jj <N; jj +=b) {
        for (kk =0; kk <N; kk +=b) {

        /* iterate within a tile */
        for (i=ii; i < ( ii+b) ; i++) {
            for (j=jj; j < ( jj+b) ; j++) {
                for (k=kk; k < ( kk+b) ; k++) {
                    C[i][j] += A[i][k] * B[k][j];
```

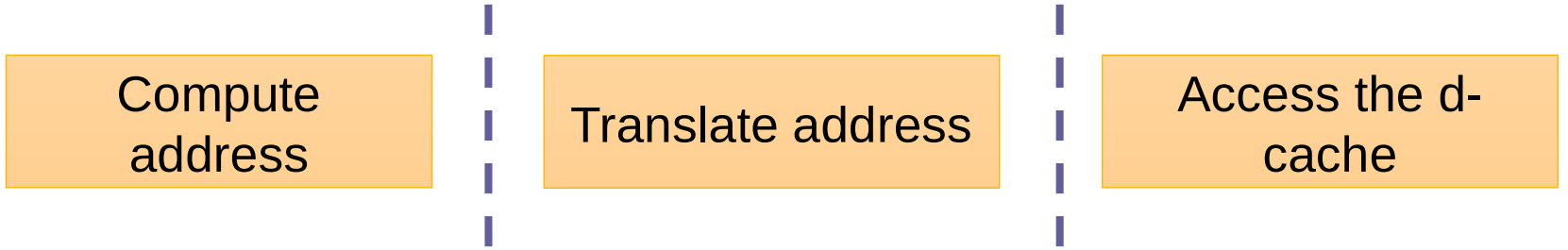


Fits within the cache

...

Prove that the algorithm is correct.

VIPT (Virtually Indexed Physically Tagged) Caches

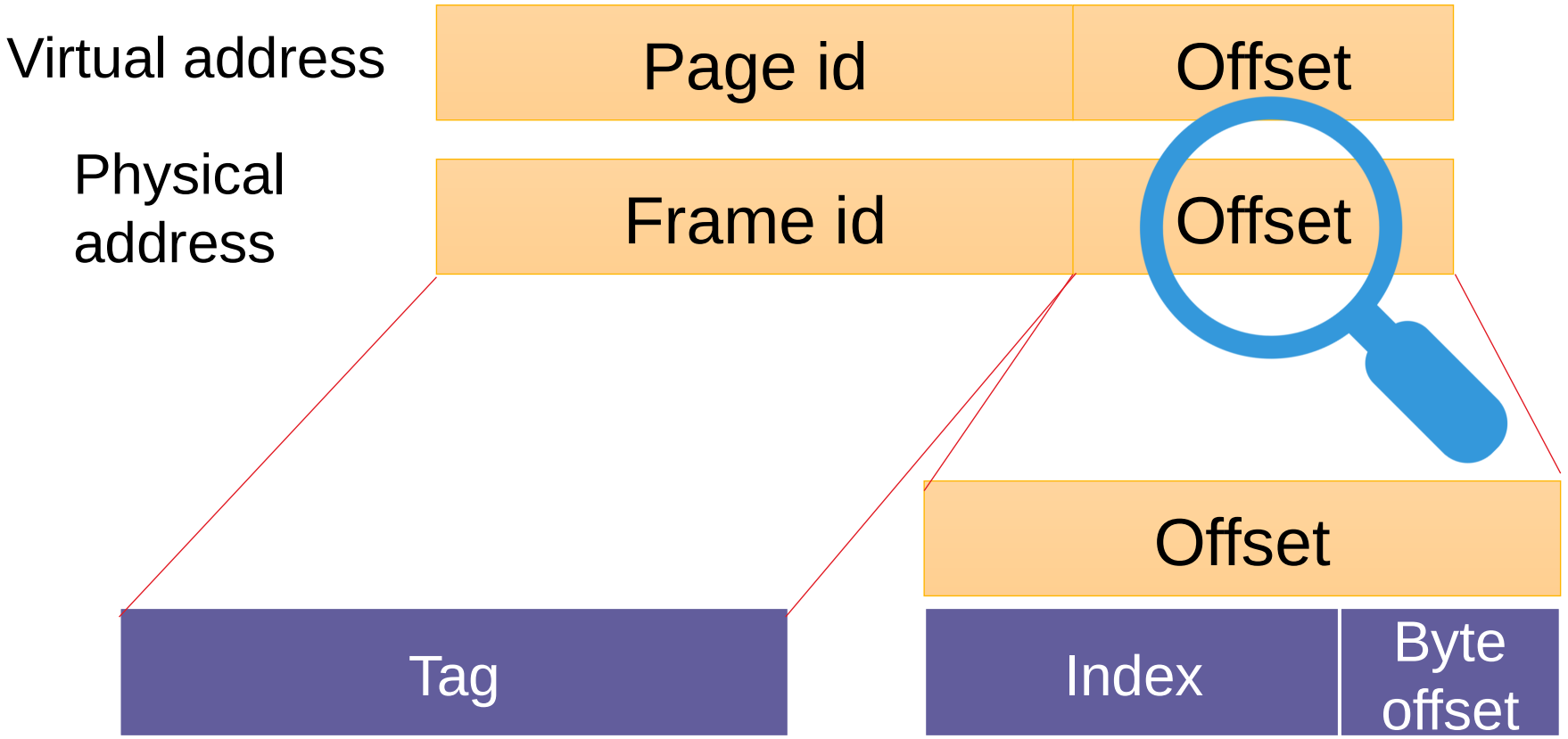


- We need to spend an additional cycle translating the address after we have computed it.

- What if we don't need any translation to access the correct set in the L1 cache?

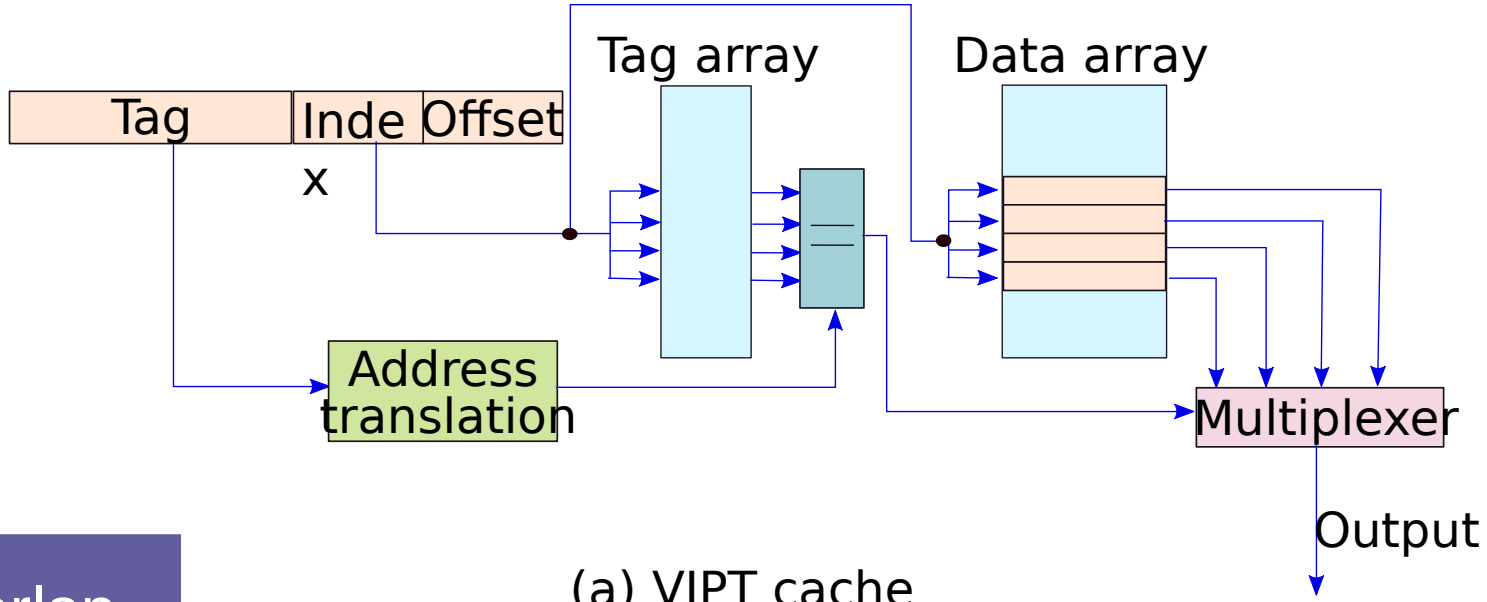
Possible

VIPT Caches

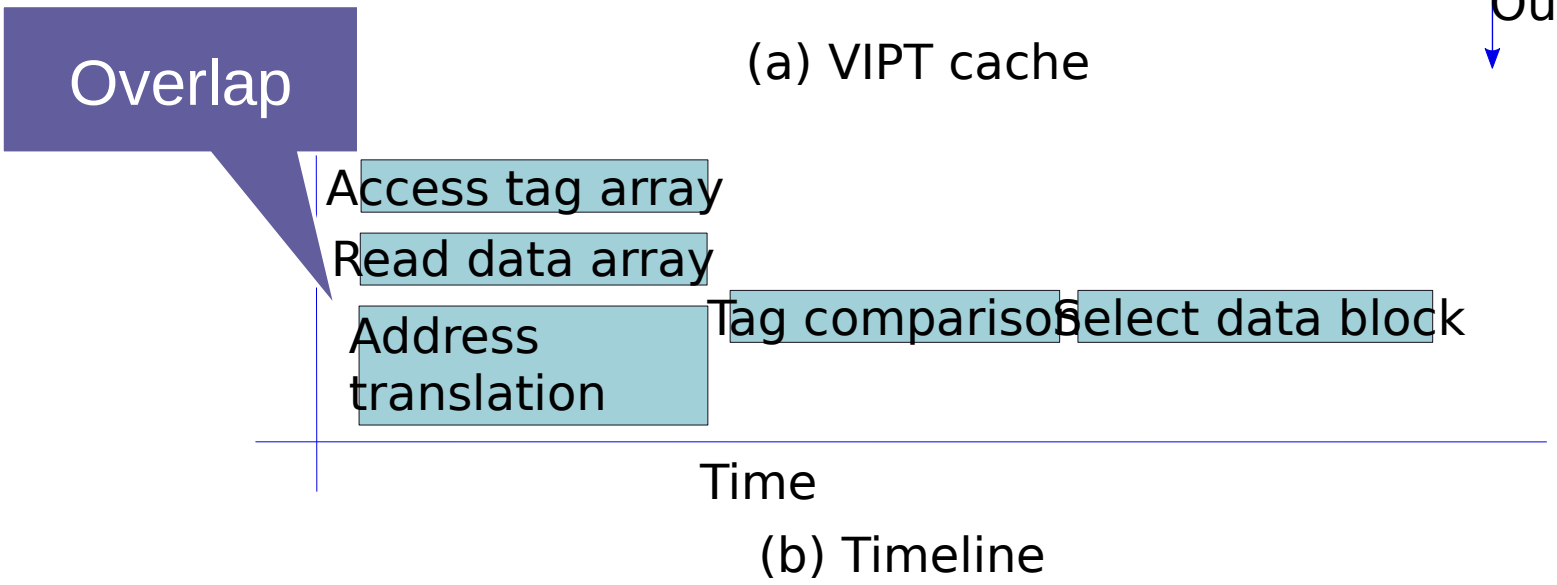


Access all the ways in the set while the address translation mechanism is working. It will compute the tag.

The Design of the VIPT Cache



(a) VIPT cache

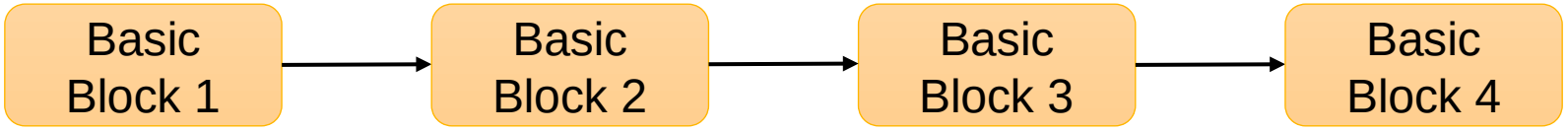


(b) Timeline

Contents

Trace Cache

Execution of a typical program



- Basic block ⎵ A set of instructions with a single point of entry and a single point of exit
- We fetch a sequence of basic blocks from the i-cache
- Such a sequence is called a trace
- Let us have a cache that stores such traces. We have the option of sequentially reading out a full trace
- This is called a trace cache (first major use: Pentium 4, circa 2000)
- If the trace is accurate, we need not predict branches or decode instructions

Early Approaches

Traditional approach

- A cache line contains contiguous cache blocks

Peleg and Weiser

- Store successive basic blocks per cache line
- Trace segments cannot span multiple cache lines

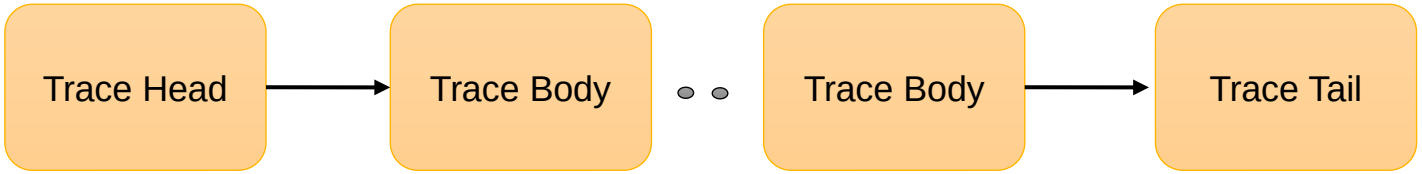
Melvin et al.

- CISC instruction sets decode instructions into micro-ops
- The idea is to store decoded micro-ops for each instruction in a cache line
- Saves some decoding effort

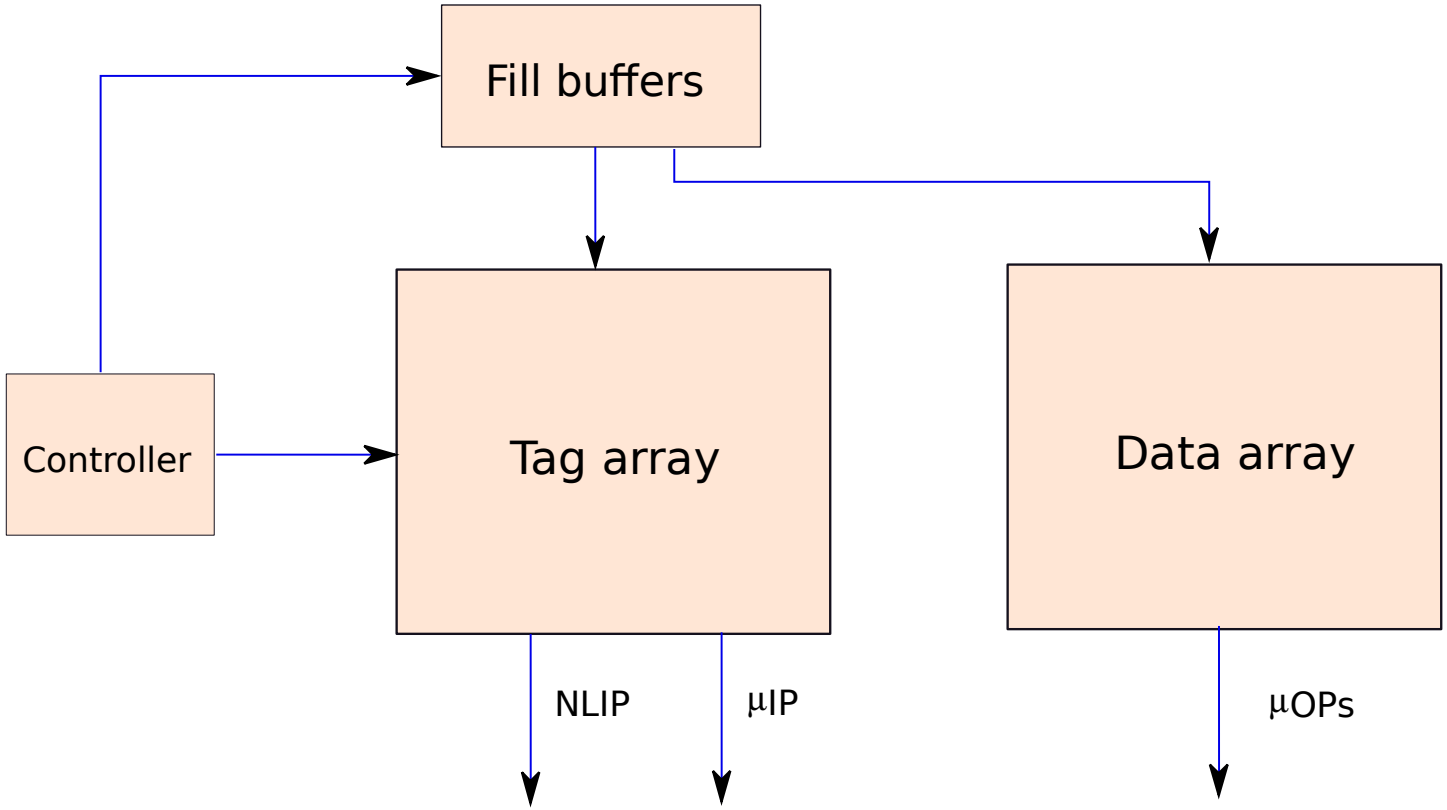
Can we combine and augment these solutions?

Structure of a Solution

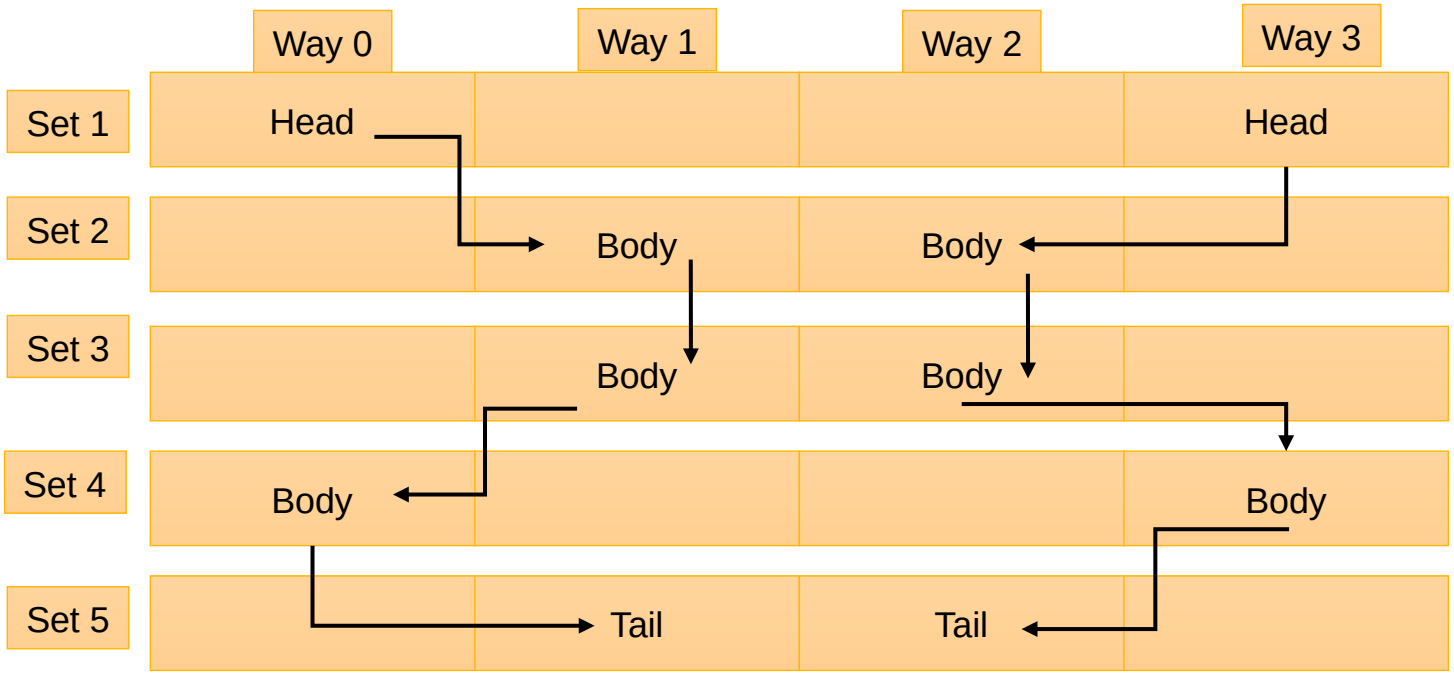
- A trace consists of multiple cache lines
- It is a linked list of cache lines (each line is a trace segment)
- We need a method to place a marker to terminate a trace and start a new trace.



Design of the Trace Cache



Storage of Traces



- Store trace segments in consecutive sets
- Each trace segment stores the number of the way in the next set
- Each data line can store up till 6 decoded micro-instructions

Basic Rules

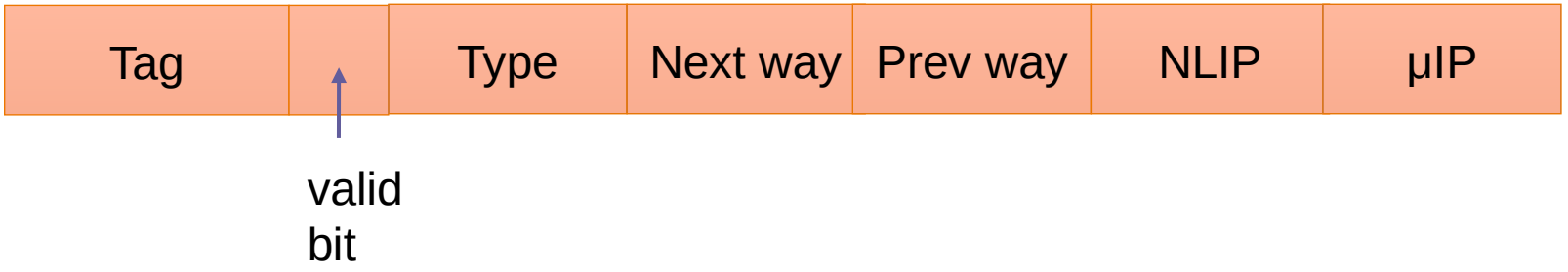
Rules for storing trace segments in a data line

- Never distribute micro-ops of a macro instruction across cache lines
- Terminate a data line if you encounter more branch micro-ops than a threshold

Termination of the trace creation process

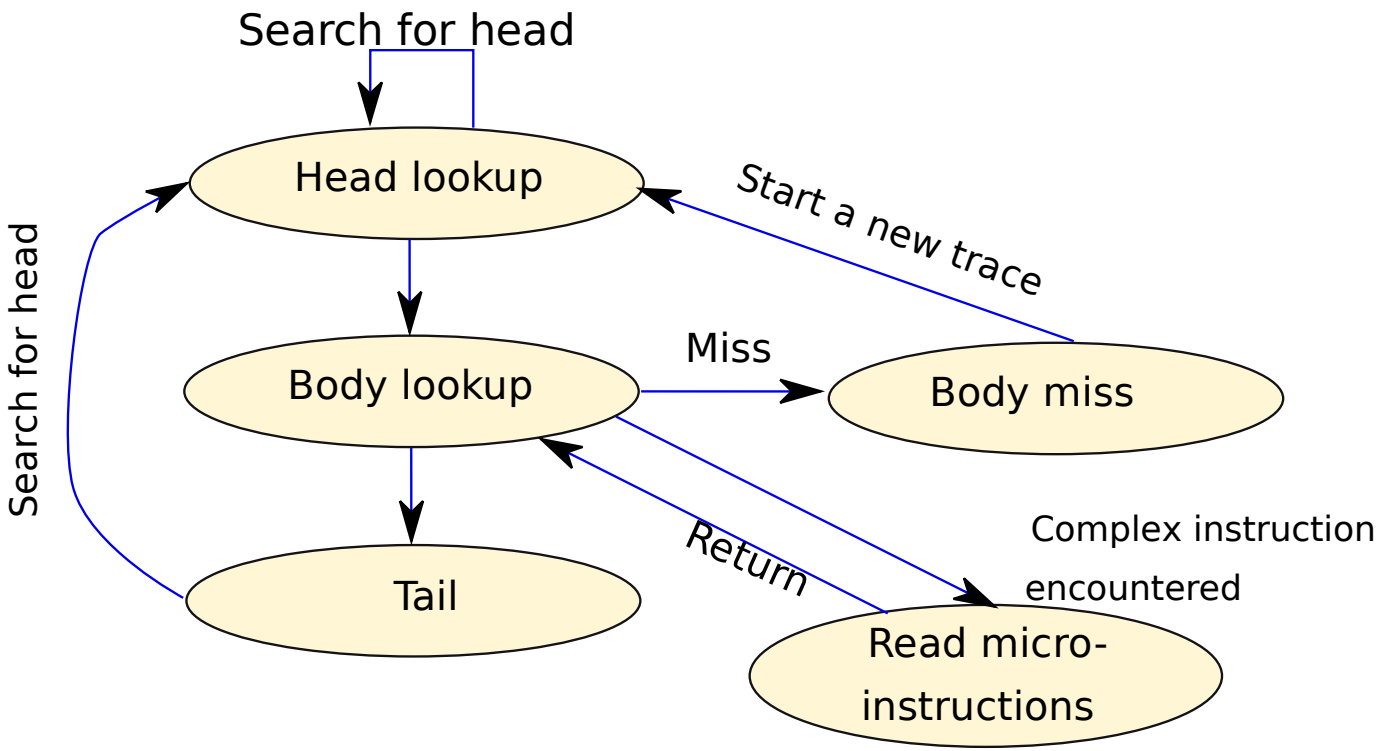
- Encounter an indirect branch
- Interrupt or branch misprediction notification
- Maximum length of trace (64 sets) reached

The Tag Array

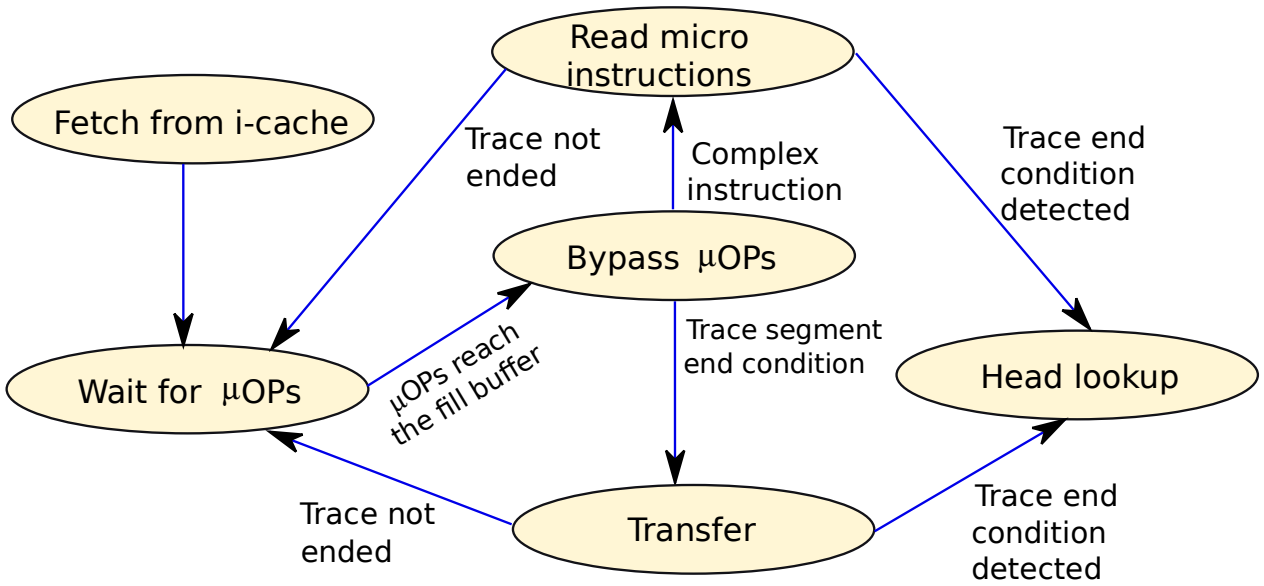


Field	Meaning
Type	Trace head, body, or tail
Next way	Index of the way for the next trace segment (next set)
Prev way	Index of the way of the previous trace segment (prev. set)
NLIP	Address of the next CISC instruction
µIP	Index into the table of micro-instructions

State Diagram (reading a trace)



State Diagram (creating a trace)



Contents

Misses in the Caches

We can incur a large performance penalty if there is a miss in the i-cache

- For the next 10-50 cycles, there will be no instructions to fetch, if there is an L2 hit
- If there is a miss in the L2, we have nothing to do for hundreds of cycles
- IPC will suffer

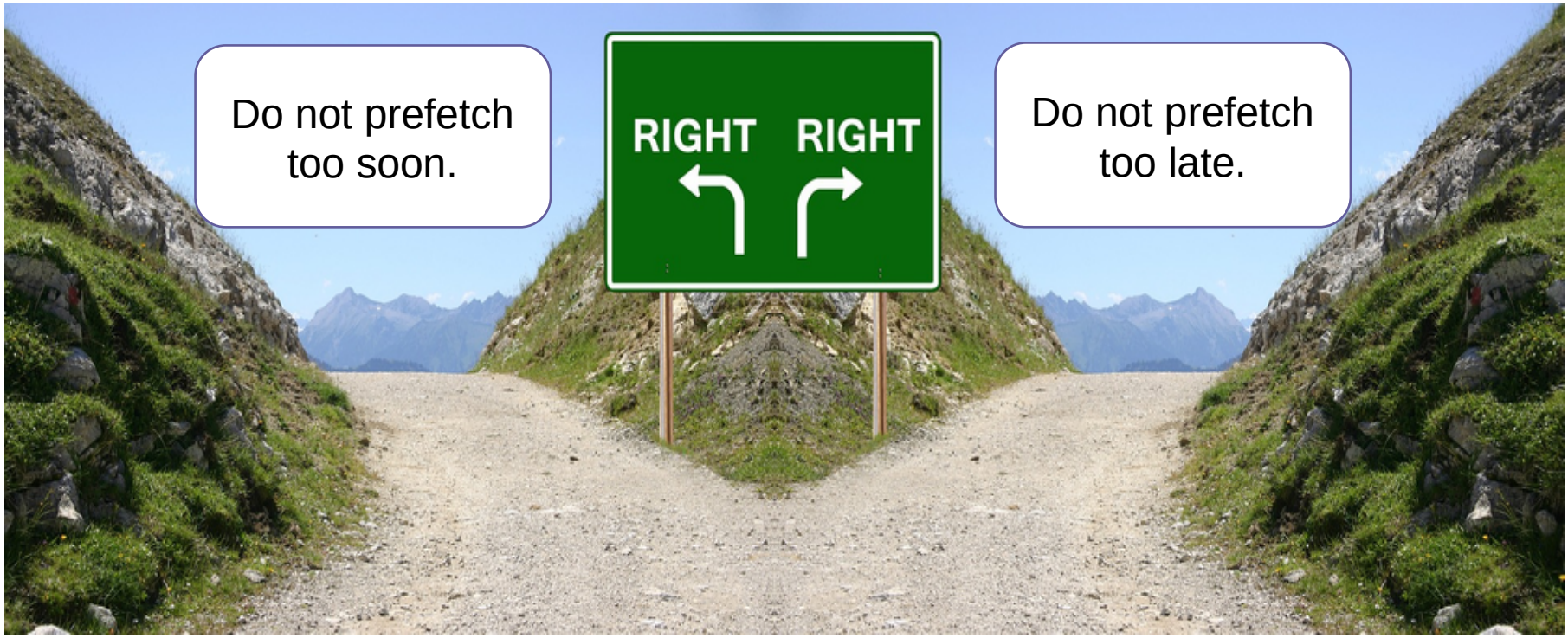
What is the solution?

- Prefetch memory addresses
- Meaning: Predict memory addresses that will be accessed in the future. Fetch them from the lower levels of the memory hierarchy before they are actually required.

Instruction Prefetching

1. Find patterns in the i-cache access sequence
2. Leverage this pattern for prefetching

Precautions



Next Line Prefetching

- Pattern: Spatial locality
- If cache line X is accessed, there is a high probability of accessing lines: *X+1* and *X+2*
- **Reason**: The code of most functions spans multiple i-cache lines and we have spatial locality
- Leverage this pattern: If a cache line X incurs an i-cache miss, read *X* and the next *k* lines from the L2 cache
- If *k* is too high, we might fill the cache with useless data.

Almost all prefetching algorithms operate on the miss sequence, not on the access sequence.

Markov prefetching

- Pattern: i-cache miss sequences have high repeatability
- The miss sequence of a core typically looks like this:

Miss sequence:

- High correlation between consecutive misses
- Leverage this pattern for prefetching

Markov prefetching

- Record the i-cache miss sequences in a table
- Given that cache line *X* incurs a miss, predict the line which will incur a miss next

Miss sequence:

Markov table:

Cache line	Option 1		Option 2	
	Address	Frequency	Address	Frequency
X	Y	3	Z	1

Higher probability

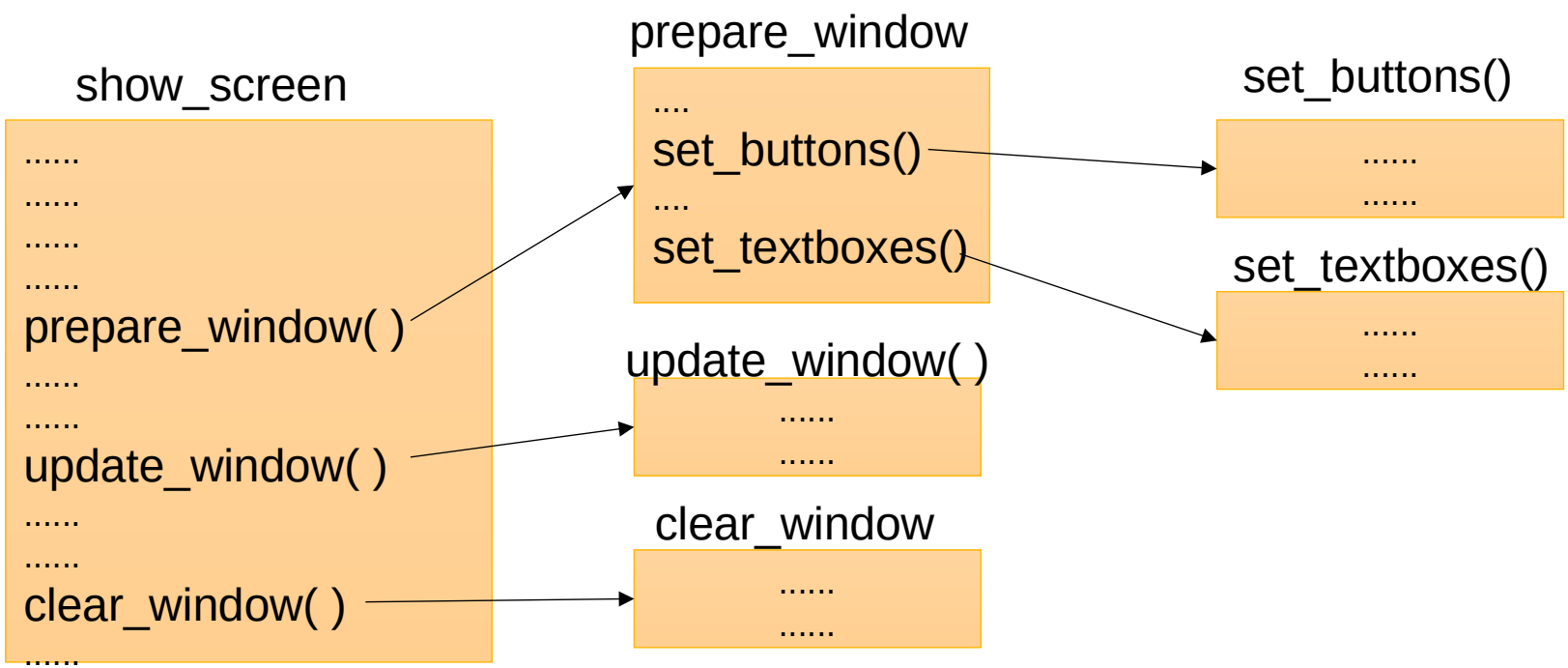
Markov Predictors - II

- We can instead have an *n-history* predictor that takes the last *n* misses into account
- Whenever there is a miss, access a prefetch table. Find the next few addresses to prefetch. Issue prefetch instructions.
- Also, update the frequencies of the entries for the last n-1 misses.
- All prefetch requests go to a prefetch queue. These requests are subsequently sent to the L2 cache (lower priority).

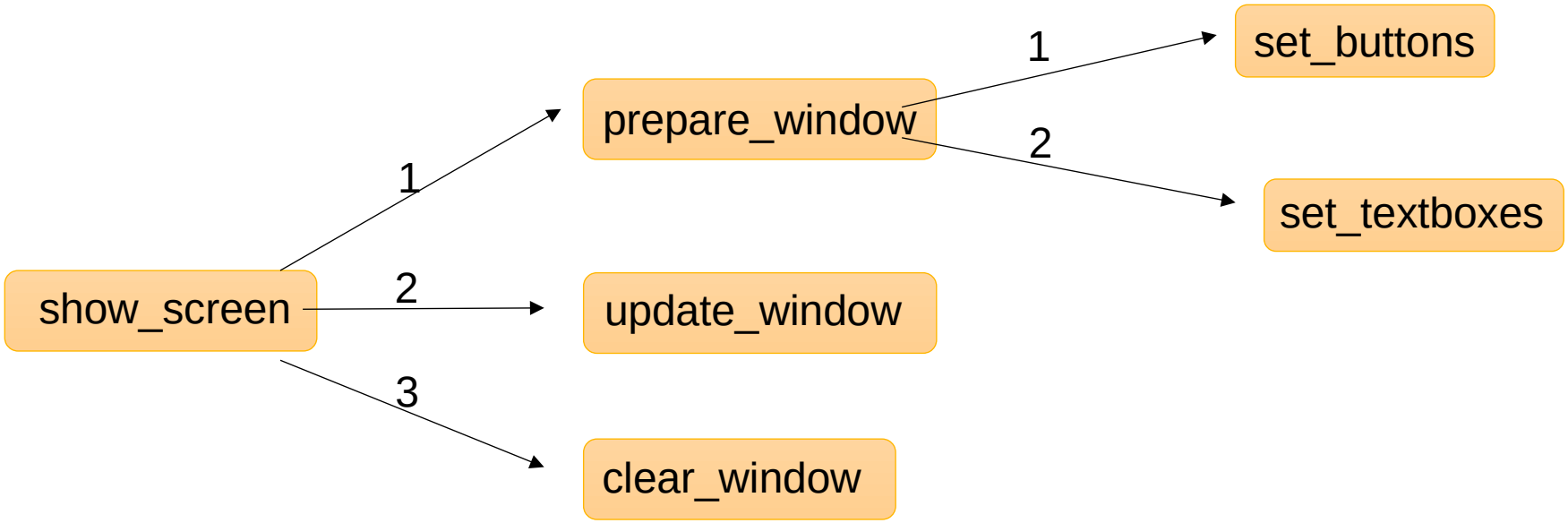
Call Graph Prefetching

- Pattern: The function call sequence is predictable
- Leverage this pattern: predict and prefetch the function that may be called next

Call Graphs


show_screen
.....
.....
.....
.....
prepare_window( )
.....
.....
update_window( )
.....
.....
clear_window( )
.....
prepare_window
....
set_buttons()
....
set_textboxes()
update_window( )
.....
.....
clear_window
.....
.....
set_buttons()
.....
.....
set_textboxes()
.....
.....

Call Graphs - II



Call Graph based Prefetching (in software)

show_screen

```
.....
prepare_window( )
.....
.....
update_window( )
.....
.....
clear_window( )
.....
.....
```

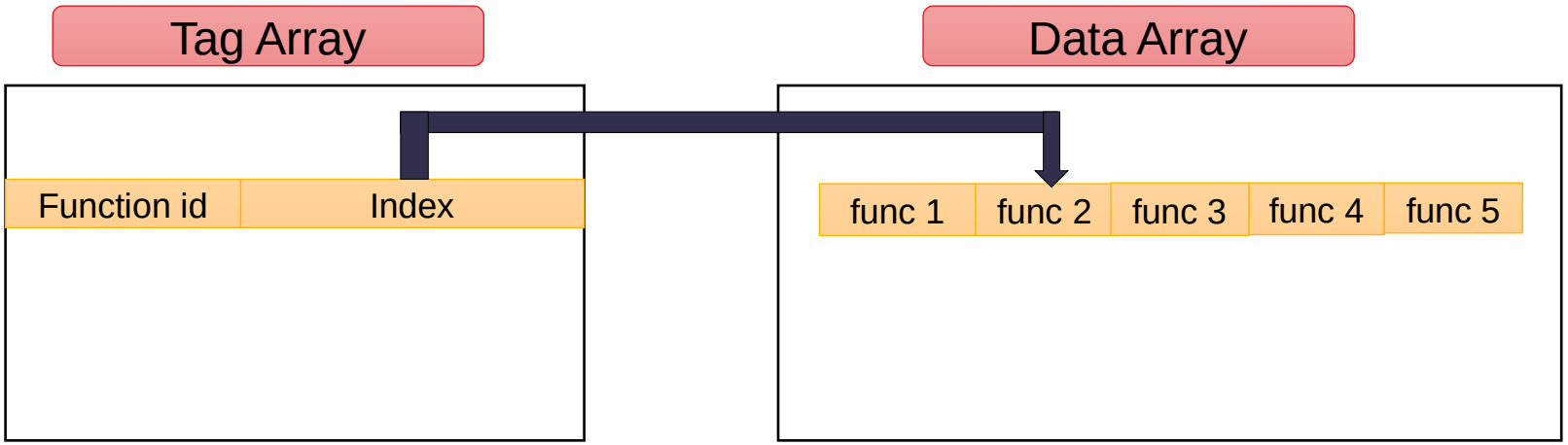
becomes

show_screen

```
prefetch prepare_window
.....
……
prepare_window()
prefetch_update_window
.....
.....
update_window( )
prefetch clear_window
.....
.....
clear_window( )
.....
```

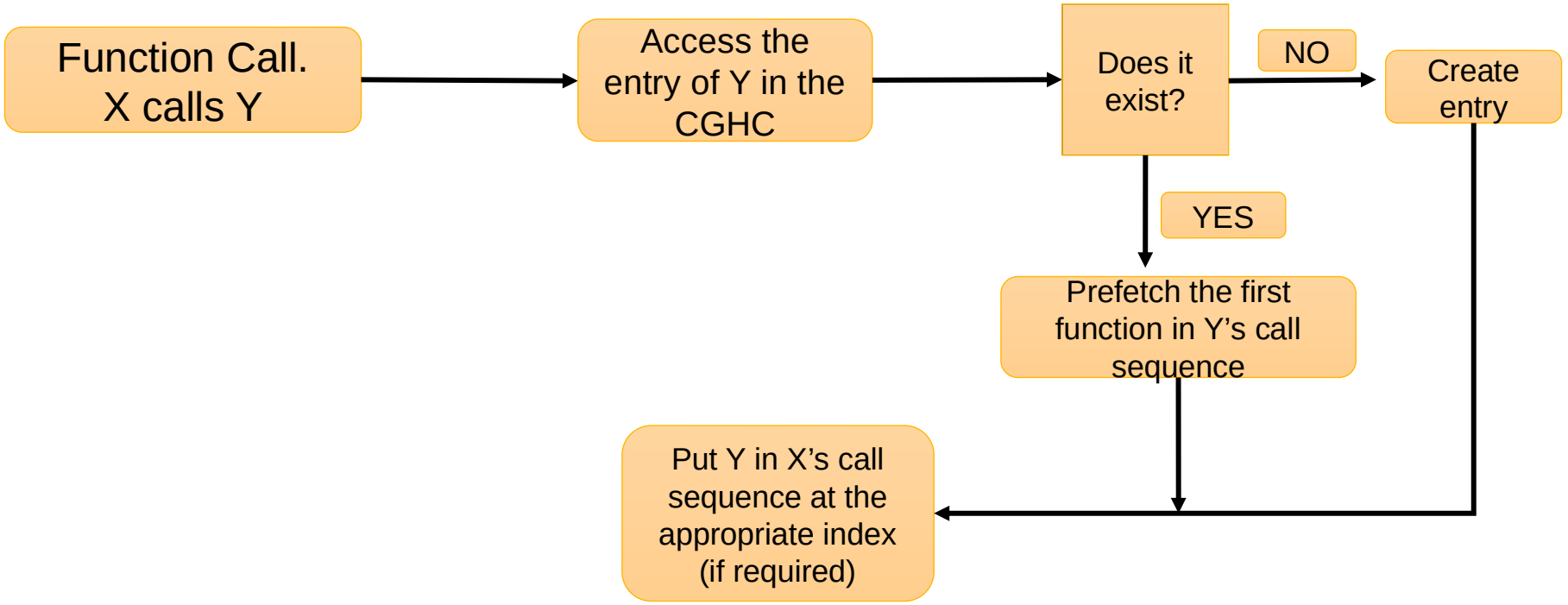
The compiler analyzes the code and inserts instructions to prefetch the code of functions.

Hardware Approach Call Graph History Cache (CGHC)

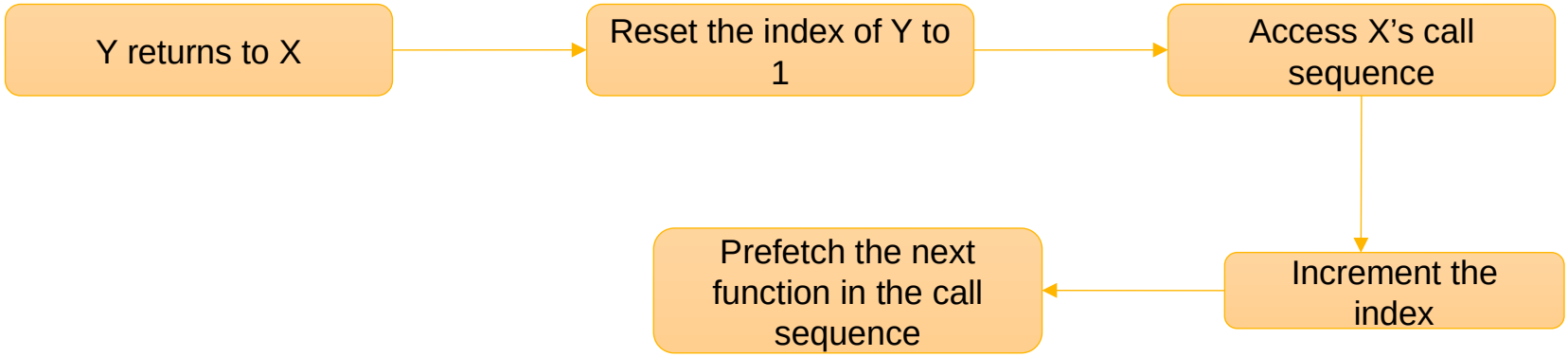


- Each entry in the data array contains a list of functions to be subsequently executed.
- The index maintains a pointer to a function in the data array entry.
- The index is initially 1. We prefetch the code of *func1.* After returning from *func1,* we set the index to 2, and prefetch the code for *func2*.

Operation (Function Call)



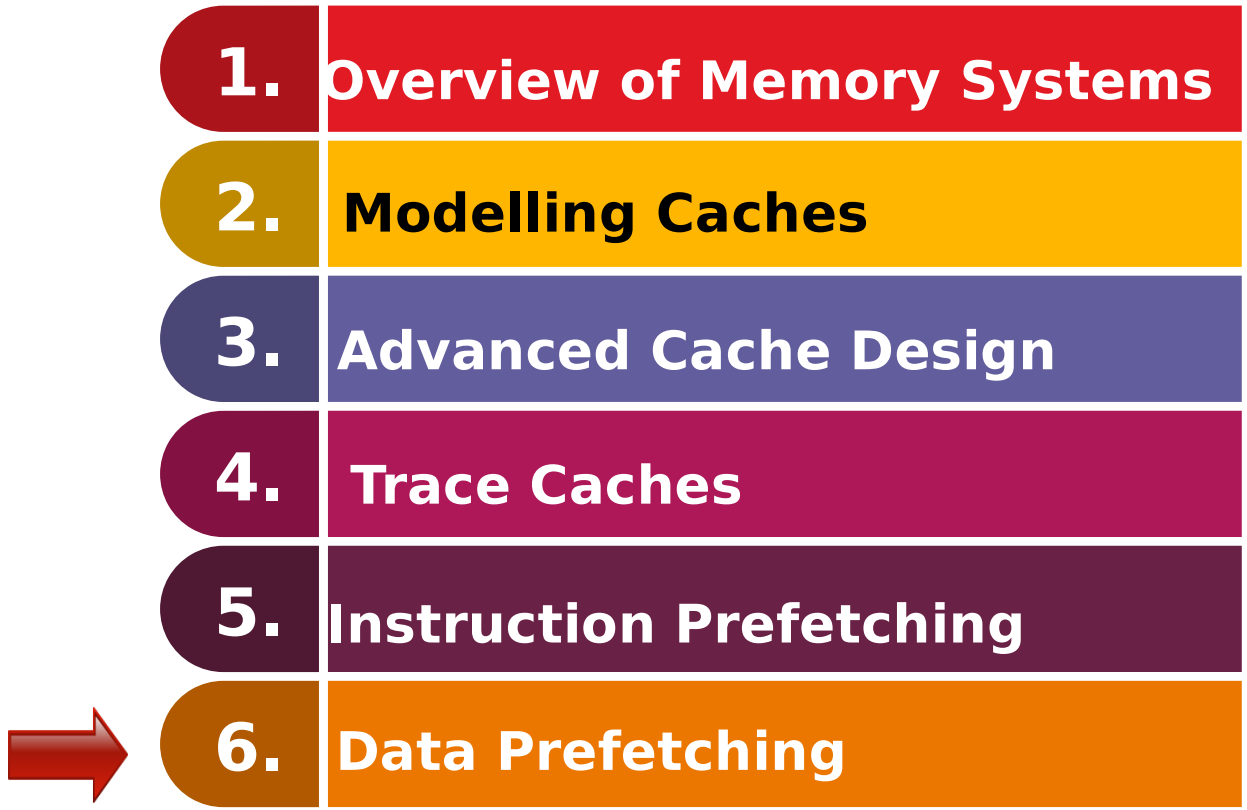
Operation (Function return)



Patterns

Technique	Pattern/Insight
Next line prefetching	Spatial locality
Markov prefetching	i-cache miss sequence has high repeatability
Call graph prefetching	Function access sequence has high repeatability

Contents


1. Overview of Memory Systems
2. Modelling Caches
3. Advanced Cache Design
4. Trace Caches
5. Instruction Prefetching
6. Data Prefetching

Data Prefetching

Instead of instructions, let us now prefetch data

Important distinction

- Instructions are fetched into the in-order part of the OOO pipeline
- Data is fetched into the OOO pipeline
  - As a result, the final IPC is slightly more resilient to data cache misses
  - Nevertheless, prefetching is very useful
  - i-cache hit rates are typically very high, whereas d-cache hit rates are much lower
  - Hence, the margin for improvement is significant.

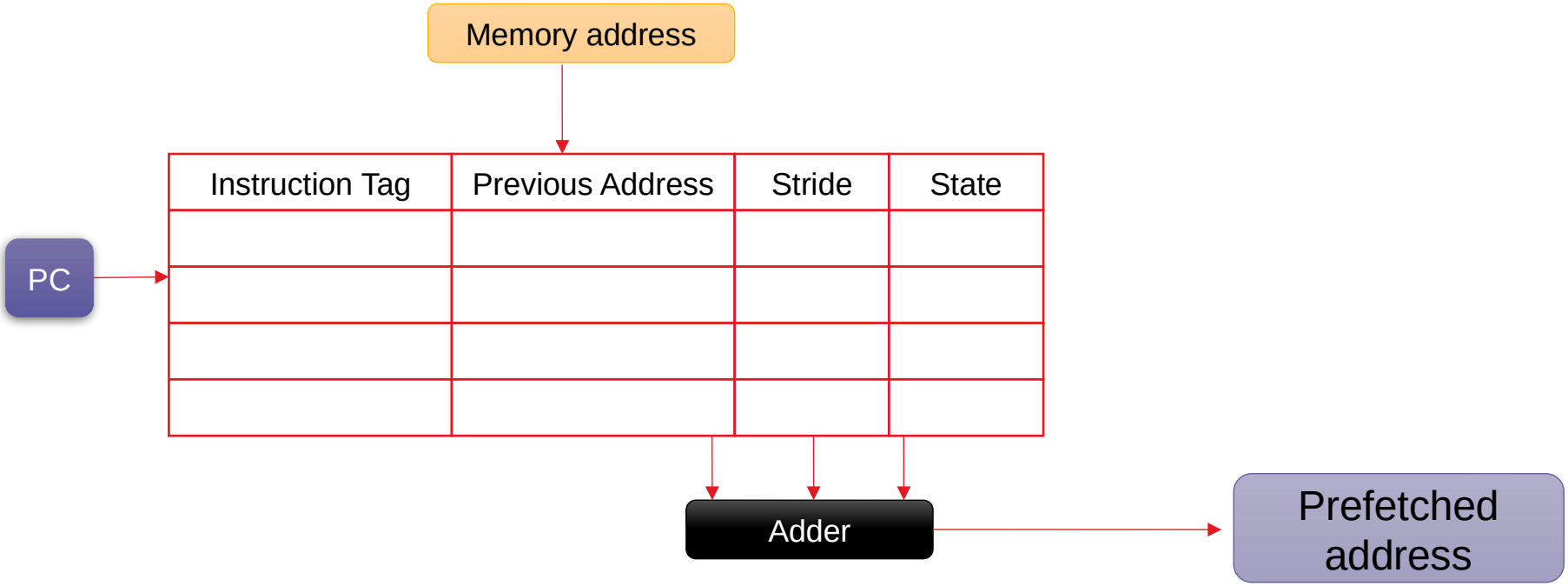
Stride based Prefetching

```
for (i = 0; i < N; i++ ) {
...
            A [i] = B[i] + C[2*i];
}
```

Consider the following piece of code

- For the arrays *A* and *B*, the addresses in each iteration differ by 4 bytes (assumed to be the size of an integer)
- For the array, *C,* consecutive accesses differ by 8 bytes
- This instruction will translate into multiple load/store RISC instructions
  - There will be only one memory access per instruction
- The hardware will observe that for the same PC
  - The memory address keeps getting incremented by a certain value (4 or 8 bytes in this case)
  - This fixed increment is called a stride

Reference Prediction Table (RPT)



For each instruction

- Keep track of the address and the stride
- We can decide to prefetch N iterations in advance
- **State**: initial ▭ transient (not sure about the stride) ▭ steady

Extensions

How many iterations do we prefetch in advance?

- This depends upon the rest of the instructions in the *for* loop
- This ideally should be dynamically adjusted
- For a subset of prefetches
  - Monitor the number of cycles between when a line is prefetched and it is actually used
  - Should not be negative
    - It is being prefetched too late. We will not get the desired benefits.
  - Should be a small positive number
    - The data arrives just before it is needed.

Pointer Chasing

```
...
/* traverse the linked list */
node * temp = start_node ;
while ( temp != NULL ) {
        prefetch (temp -> next ); /* prefetch the next node */
        process ( temp );
        temp = temp -> next ;
}
```

- Linked list accesses cannot be characterized by strides.
- We can insert code to prefetch subsequent linked list nodes.

Runahead Mode

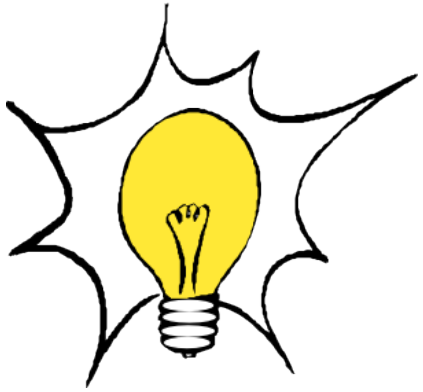
What happens when we do incorrect prefetching?

- We have misses because we do not prefetch the correct data and we displace useful data. L1 misses can more or less be taken care of by an OOO pipeline
- L2 misses are bigger problems

What happens on an L2 miss

- We go to main memory (200-400) cycles
- What does the OOO pipeline do?
  - The IW and ROB fill up
  - It pretty much stalls

Idea: Do some work during the stalled period



What can we do?

Enter runahead mode

- Return a *junk* value for the request that misses in the L2 cache
- Restart execution with the junk value
- We will produce junk values (in the forward slice)
- That is still okay. Why?
  - We will have a lot of instructions in the pipeline that can be executed correctly because they are not in the forward slice of the mispredicted load.
  - We will prefetch data into the caches, and train the predictors

Once when the L2 miss returns

- Flush the instructions in the forward slice of the L2 miss
- Re-execute them
- Very effective prefetching technique

Operation

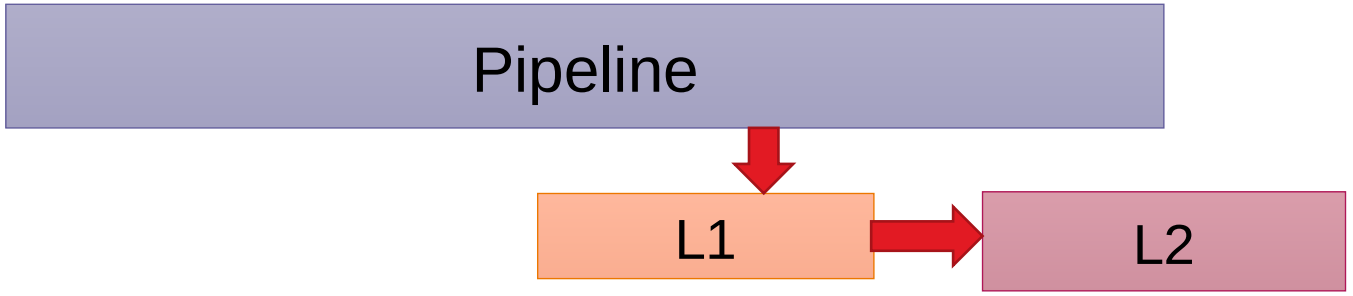
Before entering the runahead mode

- Take a checkpoint of the architectural register file + the branch history register + return address stack

Start the runahead mode

- Add an invalid (INV) bit to each register
- The L2 miss generates an INV value
- All the instructions in its foward slice have an INV value
- The INV value is same as the poison bit (learnt earlier)
- Question:
  - Do we restrict invalid values to the pipeline or let them propagate?

Value Propagation



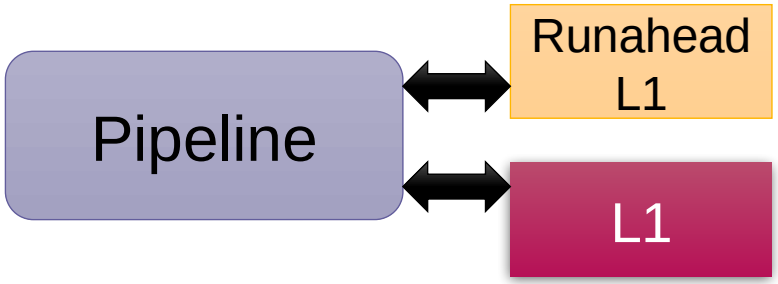
- If till the pipeline ␣ nullify all stores
- If till the L1 ␣ do not evict INV data from the L1. Invalidate it later.
- Preferably not till the L2 (massive amount of state management)

Let us take INV values till the L1

If we use the traditional L1 cache, there will be some problems

- We need to maintain both INV and non-INV data together
- What if they are on the same line (additional state required)

Solution: have an additional runahead cache that contains only INV data.



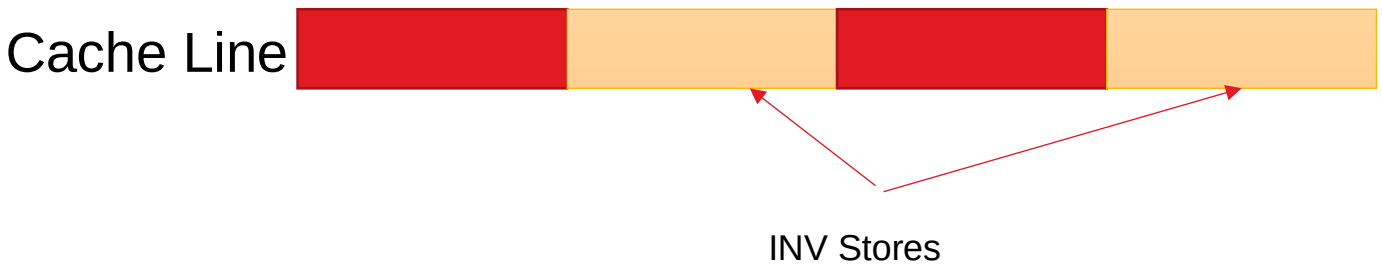
Runahead L1 Cache

A store might have the INV flag set for two reasons

- Its address is INV. Ignore.
- A stores data is INV. Access the runahead cache, and also prefetch.

Perform the store

- Let us keep some additional state per line. Let us mark those words written by an INV store as INV.



Loading a Value

First try forwarding in the LSQ

If not possible:

- Access the runahead cache and L1 cache in parallel
- The runahead cache gets preference
- Load the data. If the data is marked as INV, marked the load data as INV
- Otherwise, load data from the L1 cache
- If there is a miss, load data from the L2 (acts as prefetching)

Operation (Contd...)

- Keep fetching and retiring (in runahead mode) instructions
- All the instructions before the speculated load (in program order) will retire
- After that, all instructions are in runahead mode (will not retire)
- As we fetch and execute more and more instructions, we in effect do more prefetching

What about updating branch predictors?

- Best option: Treat runahead instructions as normal instructions

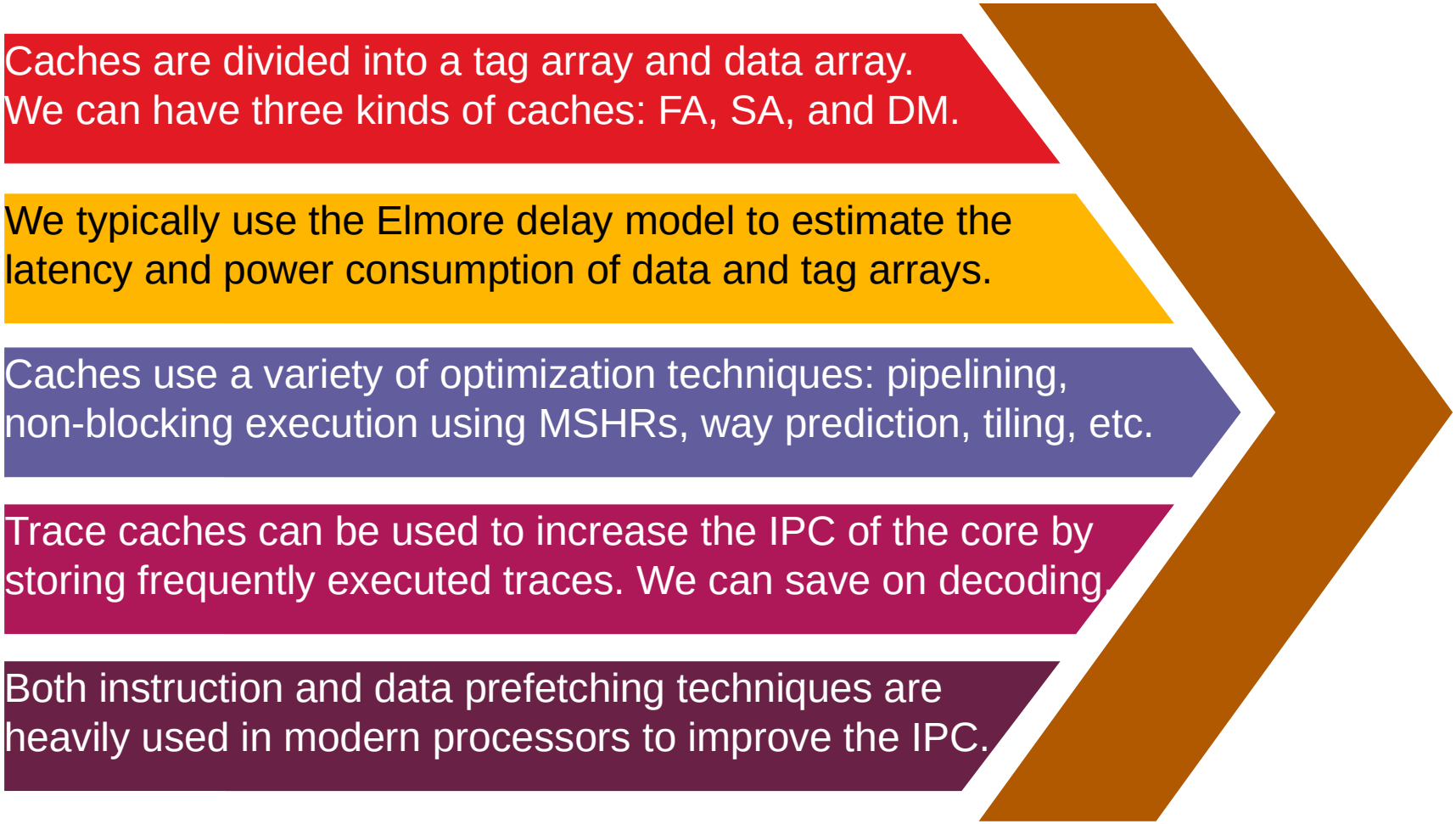
Return from runahead mode

- Similar to a branch misprediction, restore the checkpoint

Helper Threads

- In a multicore processor, we can spawn threads (lightweight processes) on other cores
- For some of the memory addresses, the threads can execute the backward slice of the load – instructions that determine the address of the load.
- The computed address can be prefetched.
- Requires some compiler support to identify the critical loads.

Conclusion


Caches are divided into a tag array and data array. We can have three kinds of caches: FA, SA, and DM.
We typically use the Elmore delay model to estimate the latency and power consumption of data and tag arrays.
Caches use a variety of optimization techniques: pipelining, non-blocking execution using MSHRs, way prediction, tiling, etc.
Trace caches can be used to increase the IPC of the core by storing frequently executed traces. We can save on decoding.
Both instruction and data prefetching techniques are heavily used in modern processors to improve the IPC.

The End