Memory Systems: Design and Implementation

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

Memory Technologies

	Capacity	Latency	Cost/GB
Register	~1K bits	20 ps	$$$$
SRAM	~10 KB-10 MB	1-10 ns	~$1000
DRAM	~10 GB	80 ns	~$10
Flash*	~100 GB	100 us	~$1
Hard disk*	~1 TB	10 ms	~$0.1

Processor Datapath (Register)

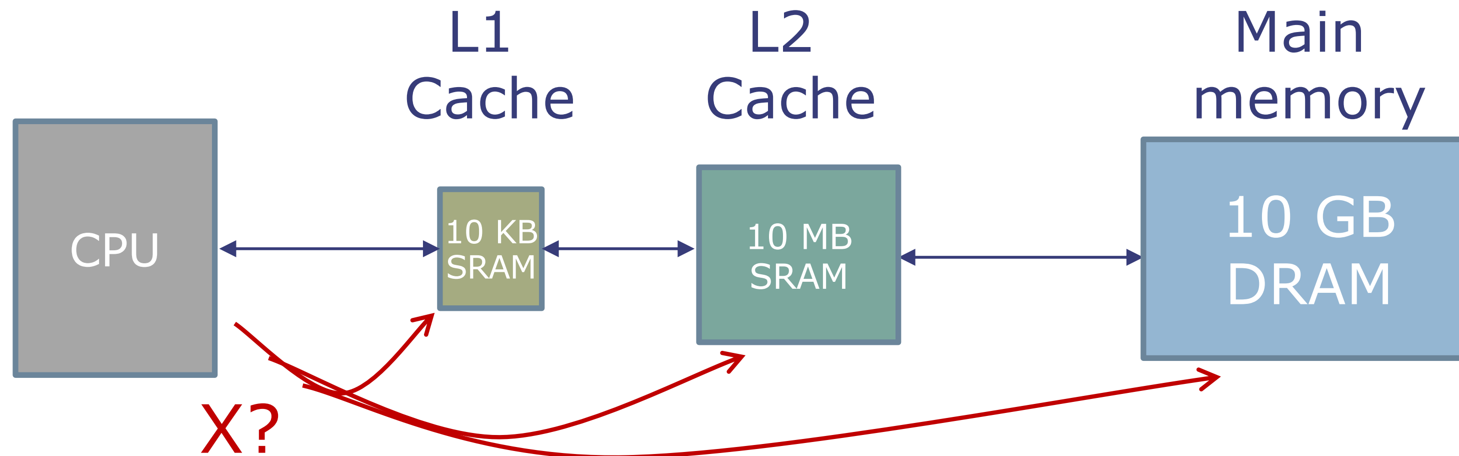
Memory Hierarchy (SRAM, DRAM)

I/O subsystem (Flash, Hard disk)

* non-volatile (retains contents when powered off)

- Different technologies have vastly different tradeoffs
- Size is a fundamental limit, even setting cost aside:
  - Small + low latency or
  - Large + high-latency
- Can we get best of both worlds? (large, fast, cheap)

Implicit Memory Hierarchy

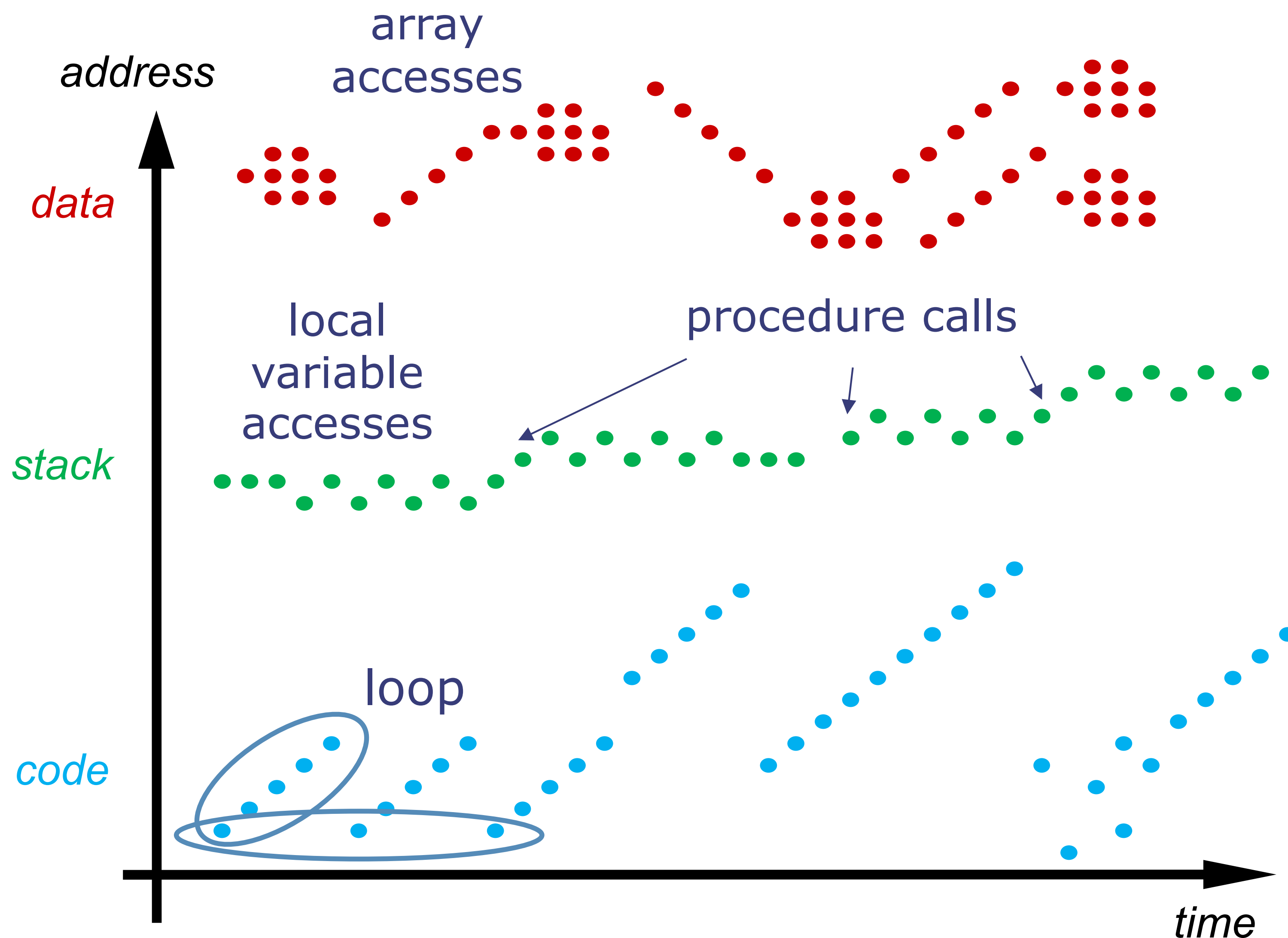


- Programming model: Single memory, single address space
- Machine transparently stores data in fast or slow memory, depending on usage patterns
- CPU effectively sees large, fast memory if values are found in cache most of the time.

Why Caches Work

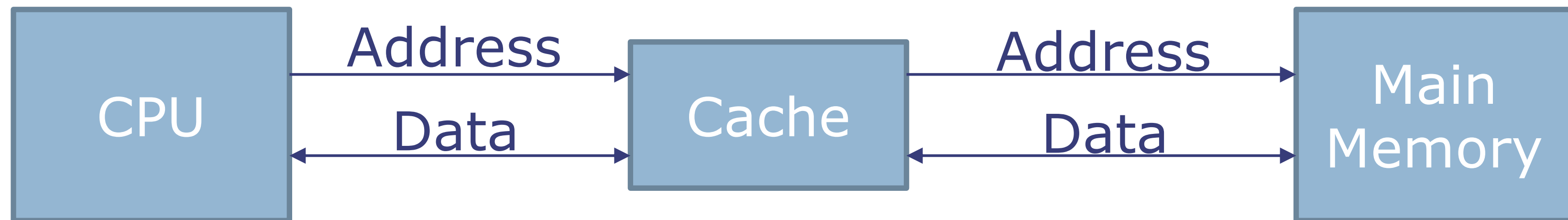
- Two predictable properties of memory accesses:
  - Temporal locality: If a location has been accessed recently, it is likely to be accessed (reused) in the near future
  - Spatial locality: If a location has been accessed recently, it is likely that nearby locations will be accessed in the near future

Typical Memory Access Patterns



Caches

- Cache: A small, interim storage component that transparently retains (caches) data from recently accessed locations



- Processor sends accesses to cache. Two options:
  - Cache hit: Data for this address in cache, returned quickly
  - Cache miss: Data not in cache
    - Fetch data from memory, send it back to processor
    - Retain this data in the cache (replacing some other data)

Processor must deal with variable access-time of memory

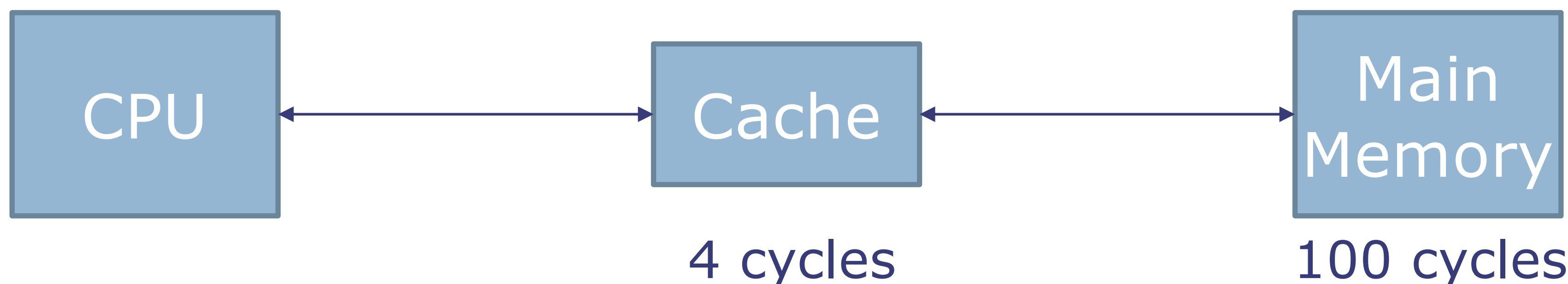
Cache Metrics

- Hit Ratio: $HR = \frac{hits}{hits + misses} = 1 - MR$
- Miss Ratio: $MR = \frac{misses}{hits + misses} = 1 - HR$
- Average Memory Access Time (AMAT):

AMAT = HitTime + MissRatio × MissPenalty

Cache design is all about reducing AMAT

How High of a Hit Ratio?



AMAT without a cache = 100 cycles

Latency with cache: Hit = 4 cycles; Miss = 104 cycles

What hit ratio do we need to break even?

$$100 = 4 + (1 - HR) \times 100 \Rightarrow HR = 4\%$$

should be easy to achieve

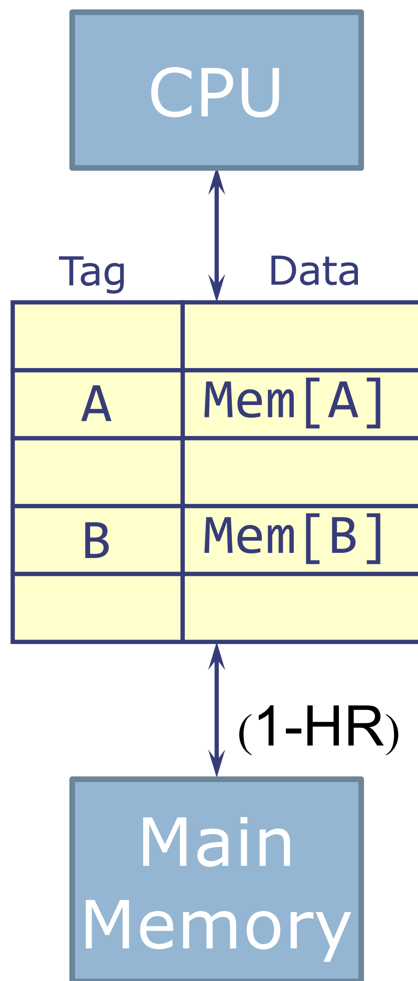
AMAT for different hit ratios:

$$HR=50\% \Rightarrow AMAT = 4 + (1 - .50) \times 100 = 54$$
$$HR=90\% \Rightarrow AMAT = 4 + (1 - .90) \times 100 = 14$$
$$HR=99\% \Rightarrow AMAT = 4 + (1 - .99) \times 100 = 5$$

Can we achieve such high HR?

With high HR caches can dramatically improve AMAT

Basic Cache Algorithm (Reads)



On reference to Mem[X], look for X among cache tags

HIT: X = Tag(i) for some cache line i

Return Data(i)

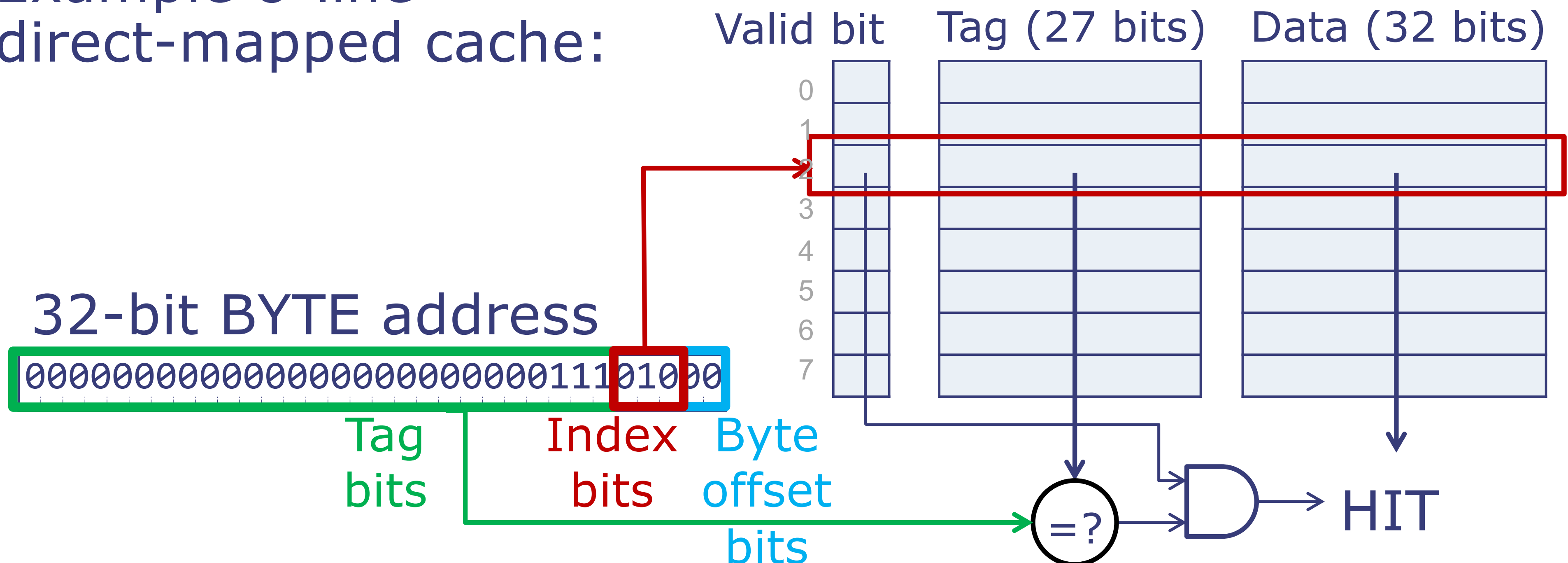
MISS: X not found in Tag of any cache line

1. Read Mem[X]
2. Return Mem[X]
3. Select a line k to hold Mem[X]
4. Write Tag(k)=X, Data(k) = Mem[X]

How do we "search" the cache?

Direct-Mapped Caches

- Each word in memory maps into a single cache line
- Access (for cache with 2^W lines):
  - Index into cache with W address bits (the index bits)
  - Read out valid bit, tag, and data
  - If valid bit == 1 and tag matches upper address bits, HIT
- Example 8-line direct-mapped cache:



Example: Direct-Mapped Caches

64-line direct-mapped cache → 64 indexes → 6 index bits

Read Mem[0x400C]

0100 0000 0000 1100

TAG: 0x40
INDEX: 0x3
OFFSET: 0x0

HIT, DATA 0x42424242

Would 0x4008 hit?

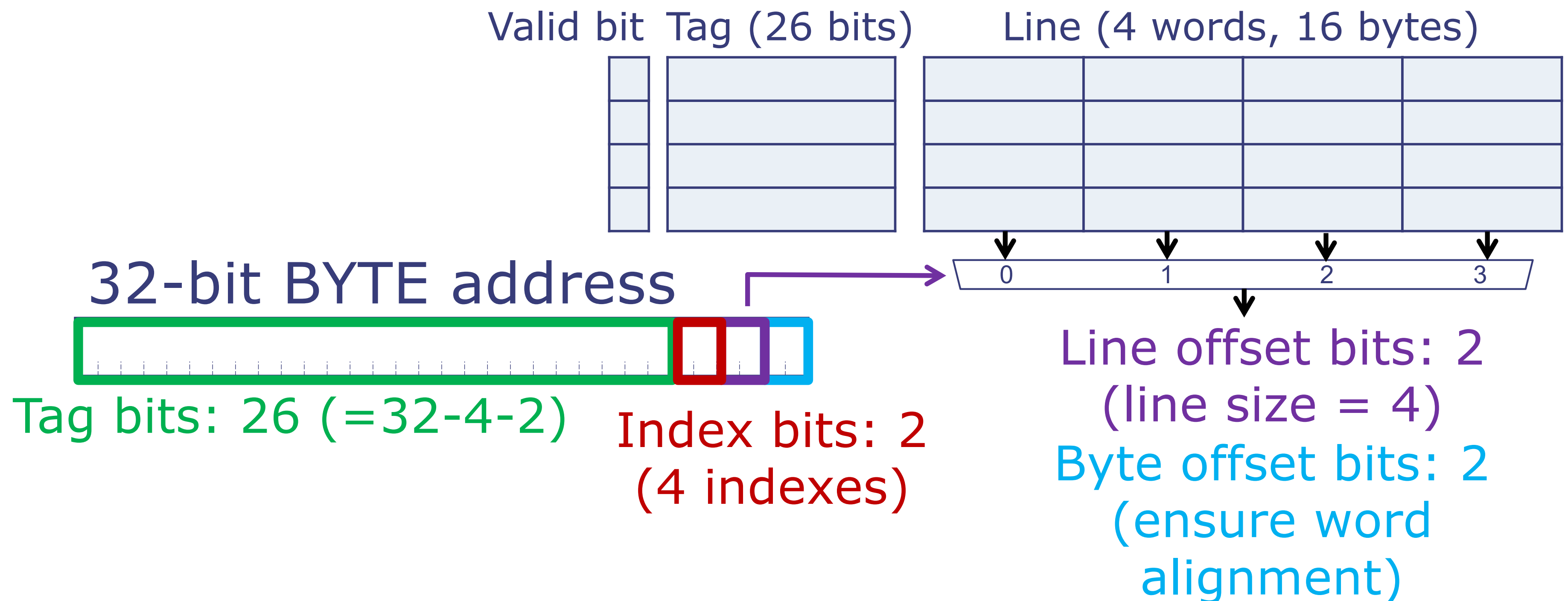
INDEX: 0x2 → tag mismatch
→ MISS

	Valid bit	Tag (24 bits)	Data (32 bits)
0	1	0x000058	0xDEADBEEF
1	1	0x000058	0x00000000
2	1	0x000058	0x00000007
3	1	0x000040	0x42424242
4	0	0x000007	0x6FBA2381
	⋮	⋮	⋮
63	1	0x000058	0xF7324A32

Part of the address (index bits) is encoded in the location
Tag + Index bits unambiguously identify the data's address

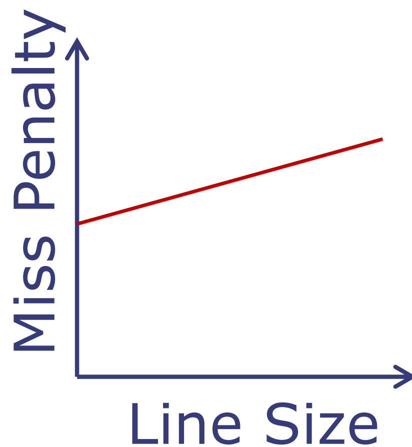
Exploiting spatial locality

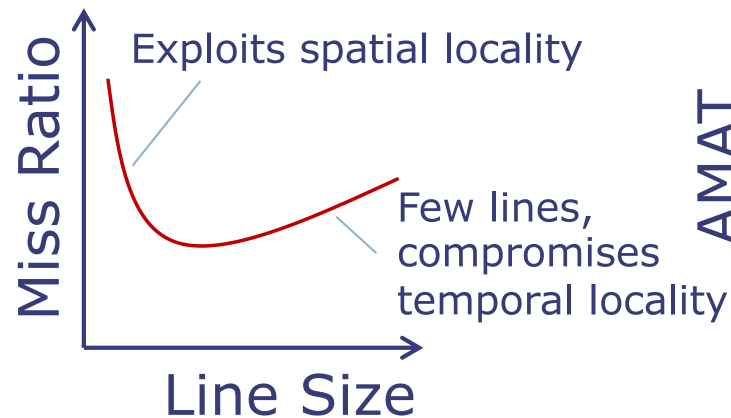
- Store multiple words per data line
  - Reduces size of tag memory!
  - Potential disadvantage: Fewer lines in the cache (more conflicts)
- Example: 4-word line, 16-word direct-mapped cache

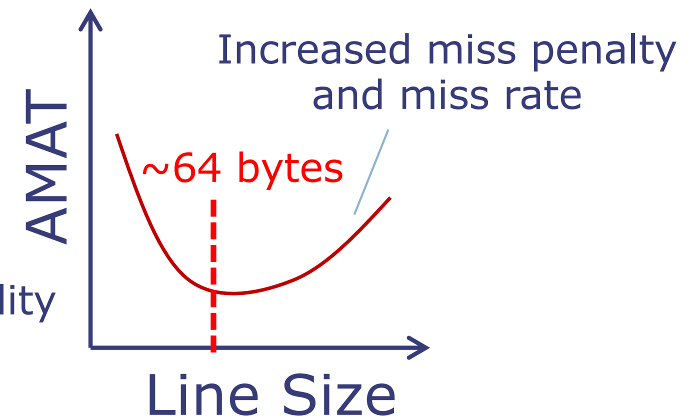


Line Size Tradeoffs

- Larger line sizes…
  - Take advantage of spatial locality
  - Incur larger miss penalty since it takes longer to transfer the line from memory
  - Can increase the average hit time and miss ratio
- AMAT = HitTime + MissPenalty*MissRatio







Write Policy

1. Write-through: CPU writes are cached, but also written to main memory immediately; Memory always holds current contents
2. Write-back: CPU writes are cached, but not written to main memory until we replace the line. Memory contents can be "stale"
   - Upon replacement, a modified cache line must first be written back to memory before loading the new cache line
   - To avoid unnecessary writebacks, a Dirty bit is added to each cache line to indicate if the value has been modified since it was loaded from memory
3. No cache write on a Write-miss: On a cache miss, write is sent directly to the memory without a cache write

Write-back is the most commonly used policy, because it saves cache-memory bandwidth

Direct-Mapped Cache Problem: Conflict Misses

	Word Address	Cache Line index	Hit/ Miss
Loop A: Code at 1024, data at 37	1024	0	HIT
	37	37	HIT
	1025	1	HIT
	38	38	HIT
	1026	2	HIT
	39	39	HIT
	1024	0	HIT
	37	37	HIT
	…		
Loop B: Code at 1024, data at 2048	1024	0	MISS
	2048	0	MISS
	1025	1	MISS
	2049	1	MISS
	1026	2	MISS
	2050	2	MISS
	1024	0	MISS
	2048	0	MISS
	...		

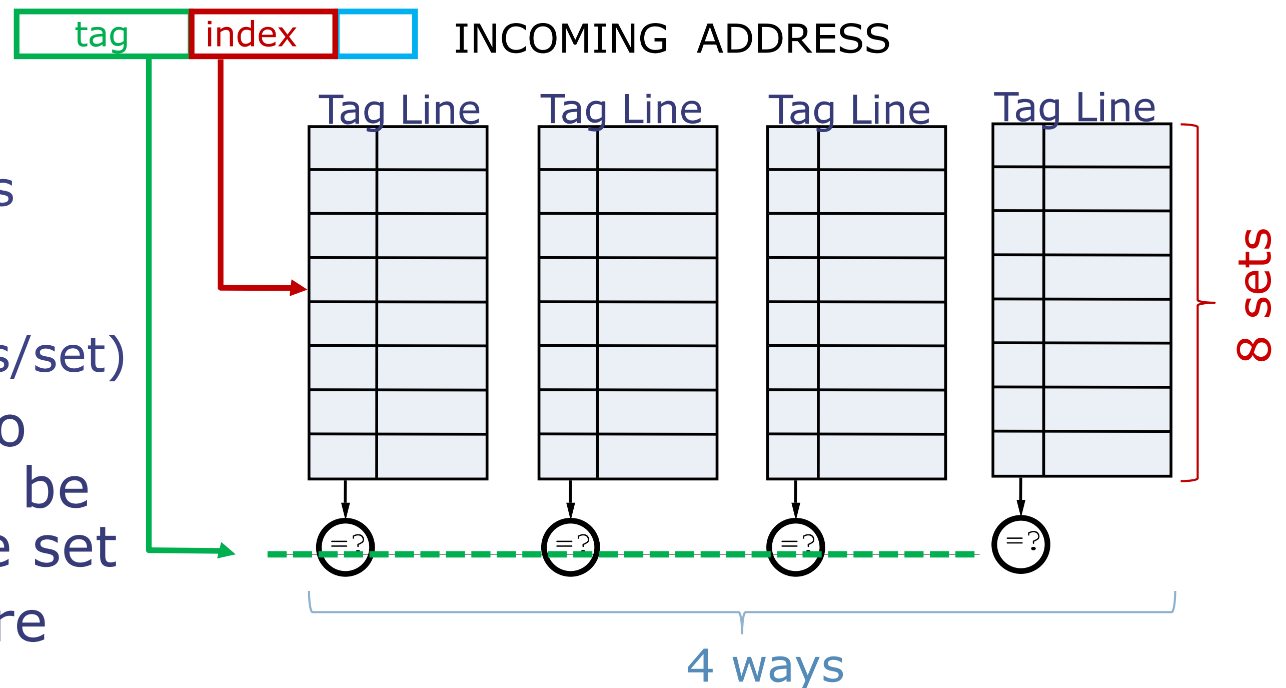
Assume:

- 1024-line DM cache
- line size = 1 word
- Consider looping code, in steady state
- Assume WORD, not BYTE, addressing

Inflexible mapping (each address can only be in one cache location) → Conflict misses!

N-way Set-Associative Cache

- Use multiple direct-mapped caches in parallel to reduce conflict misses
- Nomenclature:
  - # Rows = # Sets
  - # Columns = # Ways
  - Set size = #ways
    = "set associativity"
    (e.g. 4-way → 4 lines/set)
- Each address maps to only one set, but can be in any way within the set
- Tags from all ways are checked in parallel

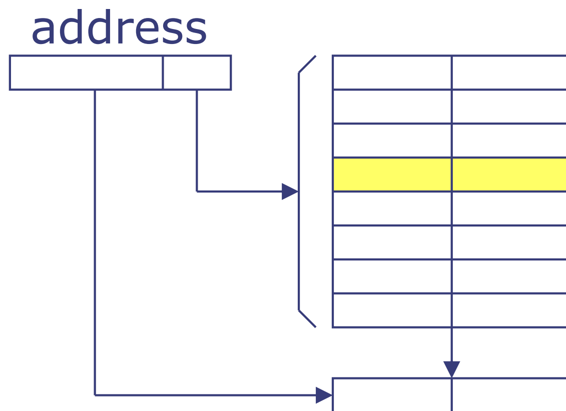


- Fully-associative cache: Number of ways = Number of lines
  - Any address can be in any line → No conflict misses, but expensive

Associativity Implies Choices

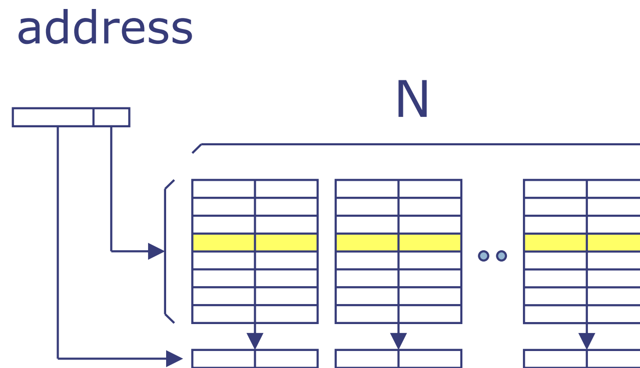
Issue: Replacement Policy

Direct-mapped



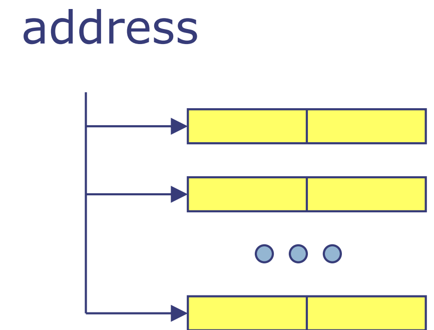
- Compare addr with only one tag
- Location A can be stored in exactly one cache line

N-way set-associative



- Compare addr with N tags simultaneously
- Location A can be stored in exactly one set, but in any of the N cache lines belonging to that set

Fully associative



- Compare addr with each tag simultaneously
- Location A can be stored in any cache line

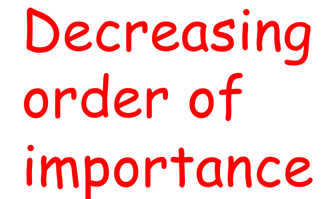
Replacement Policies

- Least Recently Used (LRU): Replace the line that was accessed furthest in the past
  - Works well in practice
  - Need to keep ordered list of N items $\rightarrow$ N! orderings $\rightarrow O(\log_2 N!) = O(N \log_2 N)$ "LRU bits" + complex logic
  - Caches often implement cheaper approximations of LRU

- Other policies:
  - First-In, First-Out (least recently replaced)
  - Random: Choose a candidate at random
    - Not very good, but does not have adversarial access patterns

Cache Design

- Cache designs have many parameters:
  - Cache size in bytes
  - Line size, i.e., the number of words in a line
  - Number of ways, the degree of associativity
  - Replacement policy



- A typical method of evaluating performance is by calculating the number of cache *hits* for a given set of *cache parameters* and a give set of *memory reference sequences*
  - Memory reference sequences are generated by simulating program execution
  - Number of hits, though fixed for a given memory reference pattern and cache design parameters, is extremely tedious to calculate (so it is done using a cache simulator)

Blocking vs. Non-Blocking cache

- Blocking cache
  - At most one outstanding miss
  - Cache must wait for memory to respond
  - Cache does not accept processor requests in the meantime
- Non-blocking cache
  - Continuous processing of cache hits
  - Blocking processing in case N outstanding misses

We will discuss the implementation of blocking caches

Now we will implement a cache

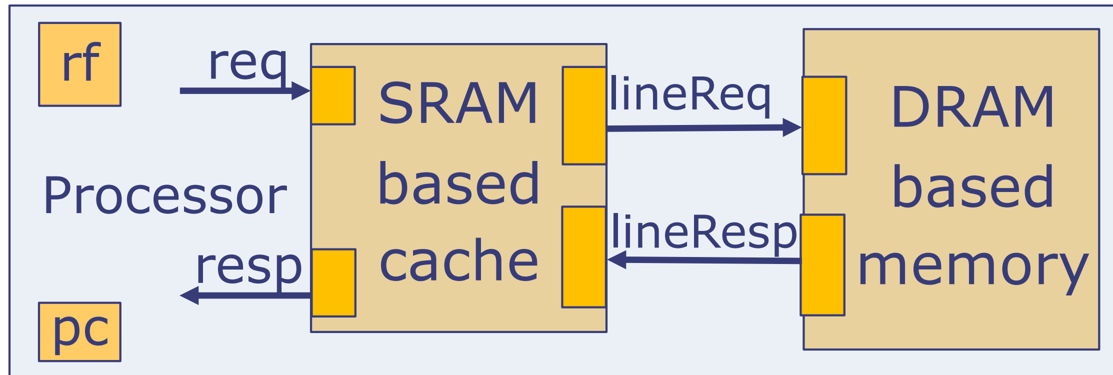
- One-way, Direct-mapped
- Write-back
- Write-miss allocate
- non-blocking cache but only one outstanding cache miss

Back-end memory (DRAM) is updated only when a line is evicted from the cache

Cache is updated on Store miss

Cache processes one request at a time

Cached Memory Systems



The memory has a small SRAM cache which is backed by much bigger DRAM memory

Processor accesses are for words while DRAM accesses are for lines

`mkDRAM` and `mkSRAM` primitives are given:

```
DRAM dram <- mkDRAM;
SRAM#(LogNumEntities, dataT) sram <- mkSRAM;
```

To avoid type clutter we assume that DRAM has 64Byte (16 word) lines and uses line addresses

Memory, SRAM and DRAM interfaces

Interfaces assume fixed sizes for memory, DRAM, line, and addresses

```
interface Memory;
   method Action req(MemReq req);
   method ActionValue#(Word) resp;
endinterface
```

no response for Stores

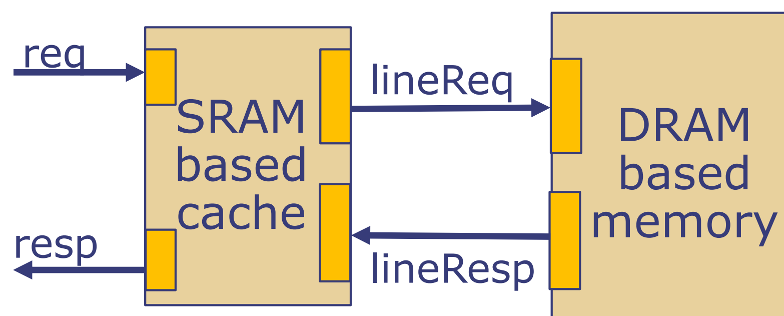
```
interface DRAM;
  method Action req(LReq req);
  method ActionValue#(Line) resp;
endinterface
```

```
interface SRAM#(numeric type indexSz, type dataT);
   method Action rdReq(Bit#(indexSz) index);
   method Action wrReq(Bit#(indexSz) index, dataT wrData);
method ActionValue#(dataT) resp;
endinterface
```

Size of SRAM = $2^{indexSz}$ data elements

```
typedef enum {Ld, St} MemOp deriving(Bits, Eq);
typedef struct {MemOp op; Word addr; Word data;} MemReq...;
typedef struct {MemOp op; LAddr laddr; Line line;} LReq...;
```

Cache Interface




```
interface Cache#(numeric type logNLines);
  method Action req(MemReq req);
  method ActionValue#(Word) resp();
  method ActionValue#(LReq) lineReq;
  method Action lineResp(Line r);
endinterface
```

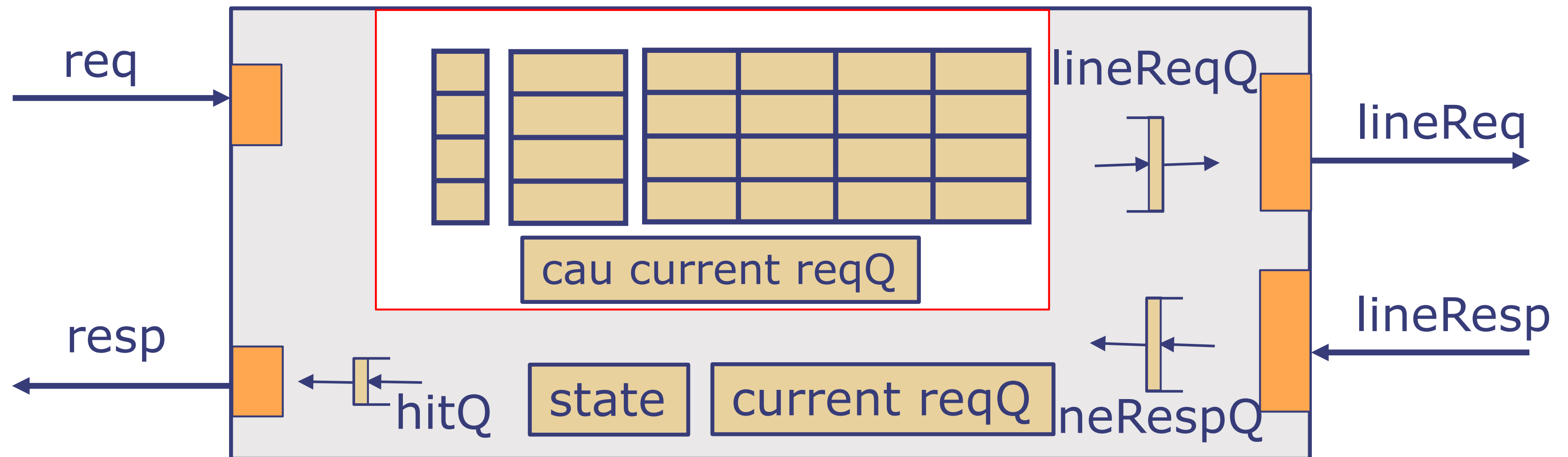
processor-side methods: fixed size words and (byte) addresses

back-side methods; fixed size cache lines and line-addresses

Notice, the cache size does not appear in any of its interface methods, i.e., users do not have to know the cache size

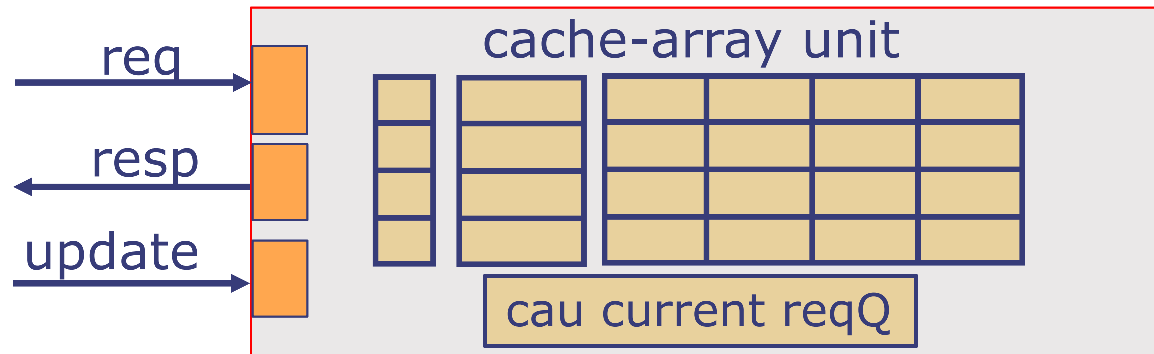
```
module mkMemory(Memory);
  DRAM dram <- mkDRAM;
  Cache#(LogNLines) cache <- mkNonBlockingCache;
...
```

Cache Organization



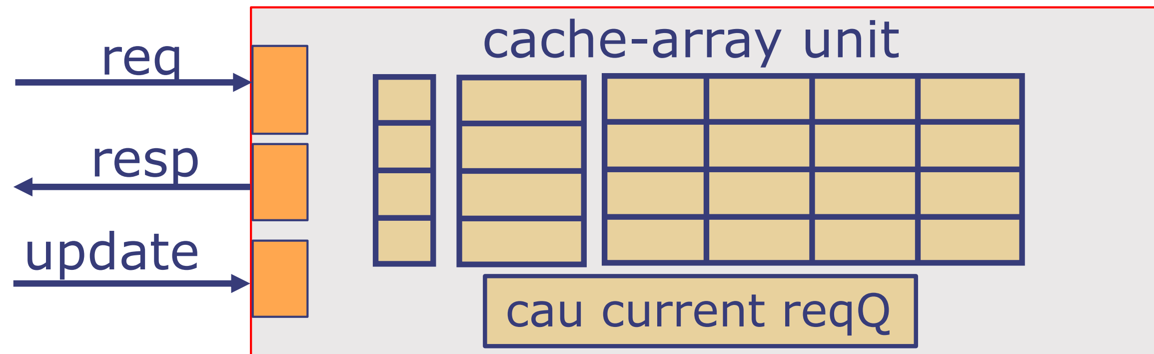
- cache-array unit encapsulates data, tag and status arrays, which are all made from SRAM
- Need queues to communicate with the back-end memory
- hitQ holds the responses until the processor picks them up
- state and current req registers hold the request and its status while the request is being processed

Cache-Array Unit Functionality



- Suppose a request gets a hit in the cache-array unit
  - Load hit returns a word
  - Store hits returns nothing (void)
- In case of a miss, the line slot must have been occupied; all the data in the missed slot is returned so that it can be written to the back-end memory if necessary
- When the correct data becomes available from the back end memory, the cache-array line is updated

Cache-Array Unit Interface




```
interface CAU#(numeric type logNLines);
  method Action req(MemReq r);
  method ActionValue#(CAUResp) resp();
  method Action update(CacheIndex index, TaggedLine newline);
endinterface
```

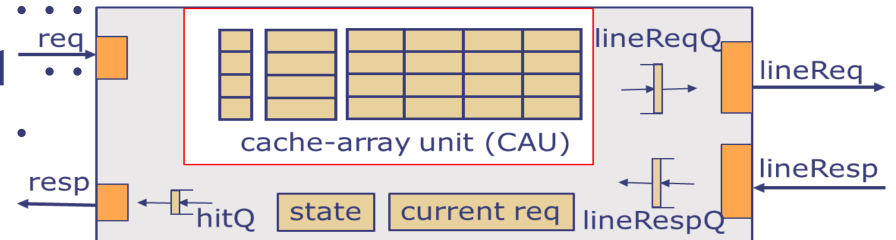
```
typedef struct{HitMissType hitMiss; Word ldValue;
               TaggedLine taggedLine;} CAUResp;
typedef enum{LdHit, StHit, Miss} hitMiss;
typedef struct{Line line; CacheStatus status; CacheTag tag;
   } TaggedLine;
```

Cache with non-blocking hits

```
module mkNonBlockingCache(Cache#(LogNLines));
   CAU#(LogNLines) cau <- mkCAU();

   FIFO#(Word)       hitQ <- mkBypassFIFO;
   FIFO#(MemReq)     currReqQ <- mkPipelineFIFO;
   Reg#(ReqStatus)   state <- mkReg(WaitCAUResp);
   FIFO#(LReq) lineReqQ <- mkFIFO;
   FIFO#(Line) lineRespQ <- mkFIFO;
```

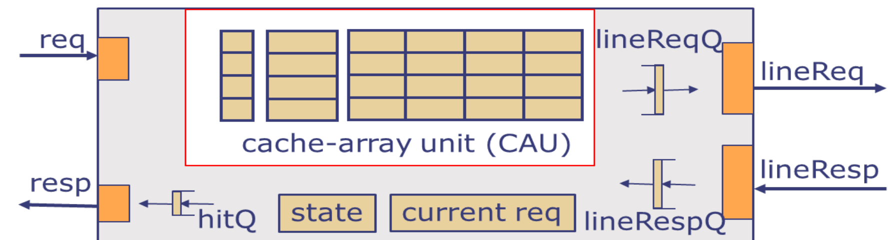
State Elements

```
    Rule to process CAU Response
    Rule to send a LineReq to DRAM
    Rule to process a LineResp from DRAM
    WaitCAUResp ->
      (if miss SendReq -> WaitDramResp) -> WaitCAUResp

  method Action req(MemReq req) ...
  method ActionValue#(Word) resp() ...
  method ActionValue#(LReq) lineReq ...
  method Action lineResp(Line r) ...

endmodule
```

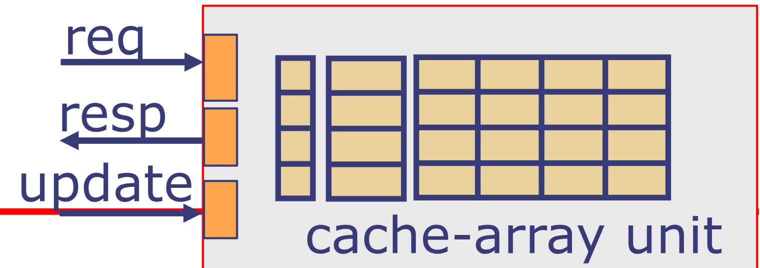


non-blocking hits methods

```
method Action req(MemReq r);
   cau.req(r);
   currReqQ.enq(r);
endmethod

method ActionValue#(Word) resp;
   hitQ.deq(); return hitQ.first;
endmethod

method ActionValue#(LReq) lineReq();
   lineReqQ.deq(); return lineReqQ.first();
endmethod

method Action lineResp (Line r);
      lineRespQ.enq(r);
endmethod
```



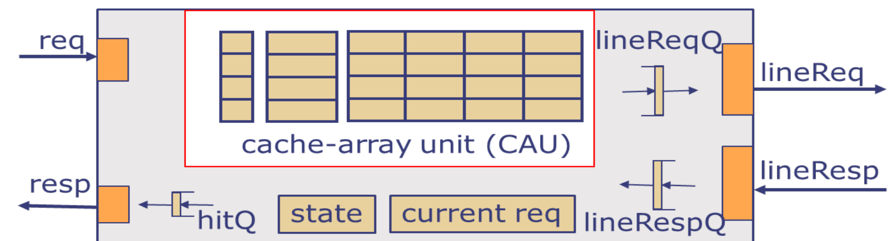
cache-array unit




```
module mkCAU(CAU#(LogNLines));
   Instantiate SRAMs for dataArray, tagArray, statusArray;
   Reg#(CAUStatus) status <- mkReg(Ready);
   Reg#(MemReq) currReq <- mkRegU; //shadow of outer currReq
   method Action req(MemReq r);
   initiate reads to tagArray, dataArray, and statusArray;
   store request r in currReq
   endmethod
   method ActionValue#(CAUResp) resp;
   Wait for responses for earlier requests
   Get currTag, idx, wOffset from currReq.addr and do tag match
   In case of a Ld hit, return the word; St hit, update the word;
   In case of a miss, return the data, tag and status;
   endmethod
   method Action update(CacheIndex index, TaggedLine newline);
   update the SRAM arrays at index
   endmethod
endmodule
```

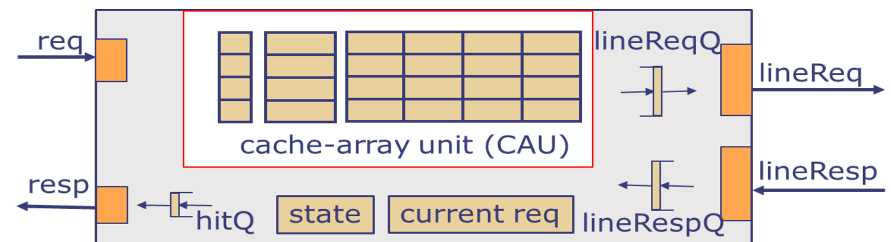
rule waitCAUResponse

```
rule waitCAUResponse (state == WaitCAUResp);
  let x <- cau.resp; let currReq = currReqQ.first;
  case (x.hitMiss)
      LdHit : begin Word v = x.ldValue;
              hitQ.enq(v); currReqQ.deq; end
      StHit : currReqQ.deq;
      Miss  : begin let oldTaggedLine = x.taggedLine;
        extract cstatus, evictLaddr, line from oldTaggedLine
        if (cstatus == Dirty) begin // writeback required
          lineReqQ.enq(LReq{op:St, laddr:evictLaddr, line:line});
          state<= SendReq;
        end else begin // no writeback required
          extract newLaddr from currReq
          lineReqQ.enq(LReq{op:Ld, laddr:newLaddr, line: ldv});
          state <= WaitDramResp;
        end
      end
  endcase
endrule
```



rule waitDramResponse

```
rule waitDramResponse(state == WaitDramResp);
  let line = lineRespQ.first(); lineRespQ.deq();
  let currReq = currReq.first;
  currReq.deq;
  get idx, tag, wOffset from currReq.addr;
  if (currReq.op == Ld) begin
     hitQ.enq(line[wOffset]);
     cau.update(idx,
       TaggedLine {line: line, status: Clean, tag: tag});
  end else begin // St
     line[wOffset] = currReq.data;
     cau.update(idx,
       TaggedLine {line: line, status: Dirty, tag: tag});
  end
  state <= WaitCAUResp;
endrule
```



Hit and miss performance

- Hit
  - Directly related to the latency of L1
  - 1-cycle latency with appropriate hitQ design
- Miss
  - No evacuation: DRAM load latency + 2 X SRAM latency
  - Evacuation: DRAM store latency + DRAM load latency + 2 X SRAM latency

Adding a few extra cycles in the miss case does not have a big impact on performance