

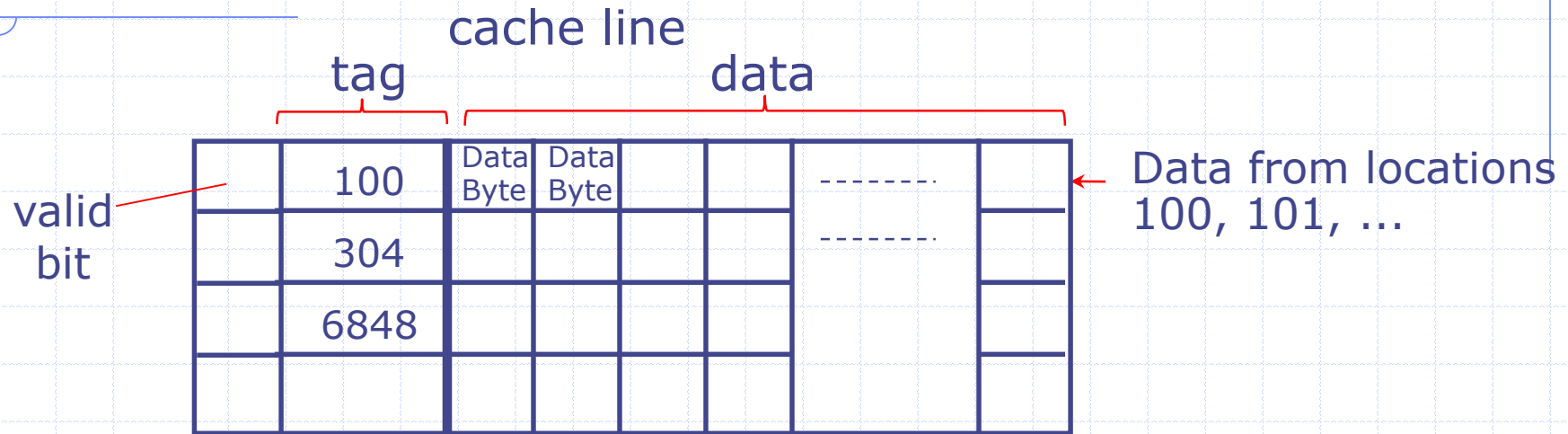
Constructive Computer Architecture

Caches and store buffers

Arvind

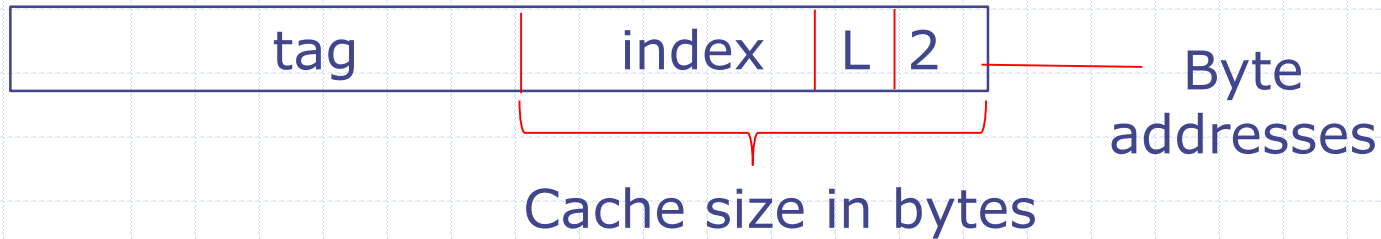
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology

Inside a Cache



- ◆ A cache line usually holds more than one word to
 - exploit spatial locality
 - transport large data sets more efficiently
 - reduce the number of tag bits needed to identify a cache line

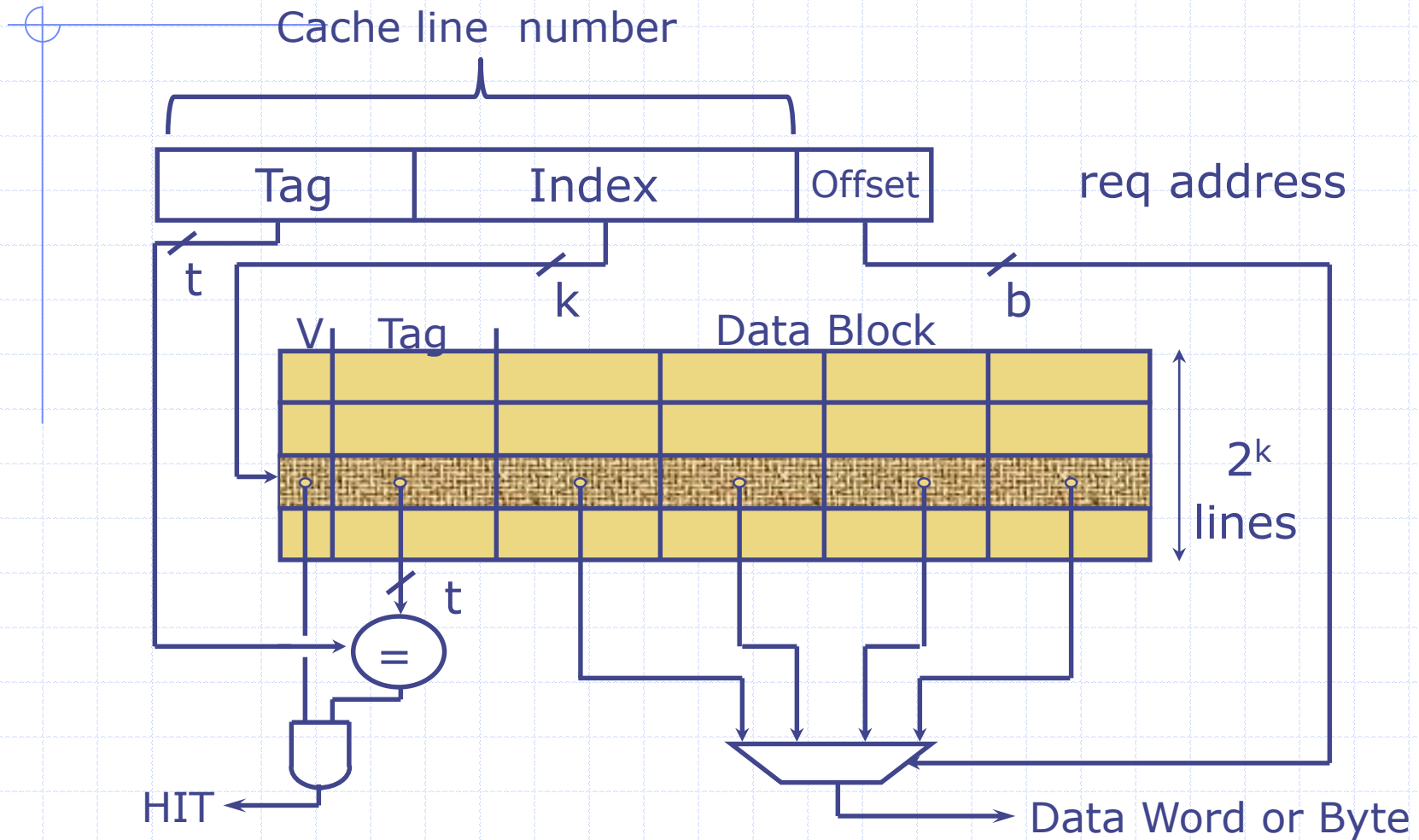
Extracting address tags & index



- ◆ Processor requests are for a single word but cache line size is 2^L words (typically $L=2$)
- ◆ Processor uses *word-aligned byte addresses*, i.e. the two least significant bits of the address are 00
- ◆ Need `getIndex`, `getTag`, `getOffset` functions

Direct-Mapped Cache

The simplest implementation



What is a bad reference pattern?

Strided and
stride = size of cache

Loads

Search the cache for the processor generated address

Found in cache
a.k.a. **hit**

Not in cache
a.k.a. **miss**

Return a copy of the word from the cache-line

Bring the missing cache-line from Main Memory

May require writing back a cache-line to create space

...

Update cache and return word to processor

Stores

- ◆ On a write hit: write only to cache and update the next level memory when line is evacuated
- ◆ On a write miss: allocate – because of multi-word lines we first fetch the line, and then update a word in it

Blocking vs. Non-Blocking cache

◆ Blocking cache:

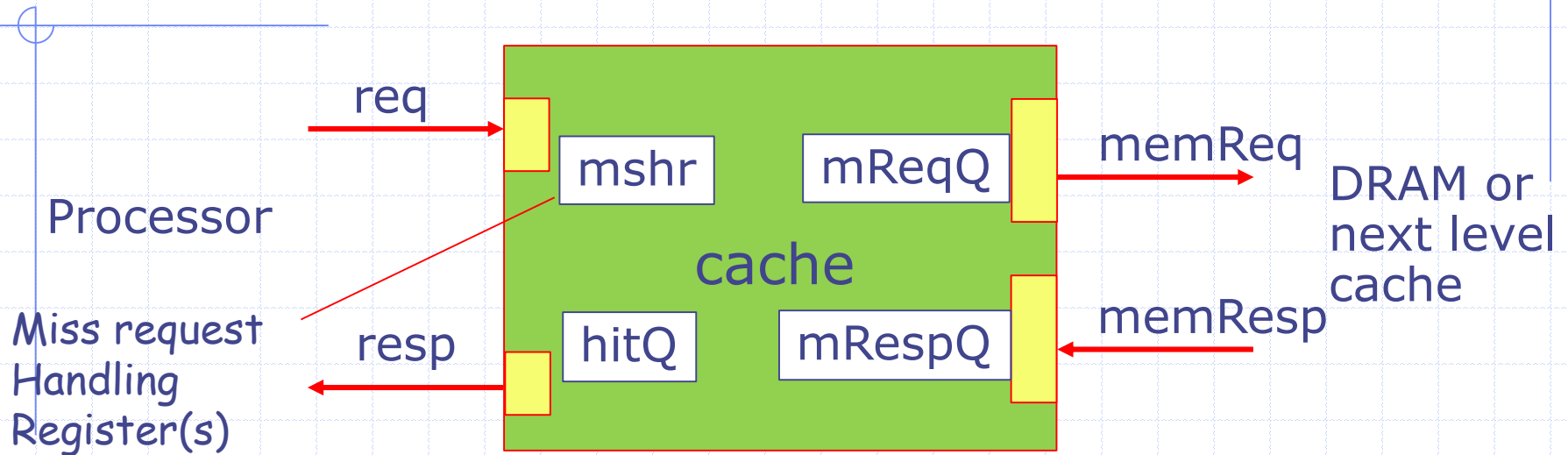
- At most one outstanding miss
- Cache must wait for memory to respond
- Cache does not accept requests in the meantime

◆ Non-blocking cache:

- Multiple outstanding misses
- Cache can continue to process requests while waiting for memory to respond to misses

We will design a write-back, Write-miss allocate, Direct-mapped, blocking cache

Cache Interface



```
interface Cache;
```

```
  method Action req(MemReq r);
```

```
    // (op: Ld/St, addr: ..., data: ...)
```

```
  method ActionValue#(Data) resp;
```

```
    // no response for St
```

```
  method ActionValue#(MemReq) memReq; methods
```

```
  method Action memResp(Line r);
```

```
endinterface
```

Requests are tagged for non-blocking caches

} Can be replaced by first and deq

Interface dynamics

- ◆ The cache either gets a hit and responds immediately, or it gets a miss, in which case it takes several steps to process the miss
- ◆ Reading the response dequeues it
 - Resp can be replaced by first and deq methods
- ◆ Methods are guarded, e.g., cache may not be ready to accept a request because it is processing a miss
- ◆ A mshr register keeps track of the state of the request while processing it

```
typedef enum {Ready, StartMiss, SendFillReq,  
             WaitFillResp} ReqStatus deriving (Bits, Eq);
```

Blocking cache state elements

```
RegFile#(CacheIndex, Line) dataArray <- mkRegFileFull;
RegFile#(CacheIndex, Maybe#(CacheTag))
    tagArray <- mkRegFileFull;
RegFile#(CacheIndex, Bool) dirtyArray <- mkRegFileFull;

Fifo#(1, Data) hitQ <- mkBypassFifo;
Reg#(MemReq) missReq <- mkRegU;
Reg#(ReqStatus) mshr <- mkReg(Ready);

Fifo#(2, MemReq) memReqQ <- mkCFFifo;
Fifo#(2, Line) memRespQ <- mkCFFifo;
```


Tag and valid bits are kept together as a Maybe type

mshr and missReq go together

CF Fifos are preferable because they provide better decoupling. An extra cycle here may not affect the performance by much

Req method

Blocking cache

```
method Action req(MemReq r) if(mshr == Ready);  
    let idx = getIdx(r.addr); let tag = getTag(r.addr);  
    let wOffset = getOffset(r.addr);  
    let currTag = tagArray.sub(idx);  
    let hit = isValid(currTag)?  
        fromMaybe(?, currTag) == tag : False;  
    if(hit) begin  
        let x = dataArray.sub(idx);  
        if(r.op == Ld) hitQ.enq(x[wOffset]);  
        else begin x[wOffset]=r.data;    
            dataArray.upd(idx, x);  
            dirtyArray.upd(idx, True); end  
    else begin missReq <= r; mshr <= StartMiss; end  
endmethod
```

overwrite the
appropriate word
of the line

Miss processing

Ready -> StartMiss -> SendFillReq -> WaitFillResp -> Ready

◆ $\text{mshr} = \text{StartMiss} \Rightarrow$

- if the slot is occupied by dirty data, initiate a write back of data
- $\text{mshr} \leq \text{SendFillReq}$

◆ $\text{mshr} = \text{SendFillReq} \Rightarrow$

- send request to the memory
- $\text{mshr} \leq \text{WaitFillReq}$

◆ $\text{mshr} = \text{WaitFillReq} \Rightarrow$

- Fill the slot when the data is returned from the memory and put the load response in hitQ
- $\text{mshr} \leq \text{Ready}$

Start-miss and Send-fill rules

Ready -> StartMiss -> SendFillReq -> WaitFillResp -> Ready

```
rule startMiss(mshr == StartMiss);
  let idx = getIdx(missReq.addr);
  let tag=tagArray.sub(idx); let dirty=dirtyArray.sub(idx);
  if(isValid(tag) && dirty) begin // write-back
    let addr = {fromMaybe(?,tag), idx, 4'b0};
    let data = dataArray.sub(idx);
    memReqQ.enq(MemReq{op: St, addr: addr, data: data});
  end
```

```
mshr <= SendFillReq;
```

endrule

Ready -> StartMiss -> SendFillReq -> WaitFillResp -> Ready

```
rule sendFillReq (mshr == SendFillReq);
  memReqQ.enq(missReq);  mshr <= WaitFillResp;
```

endrule

Wait-fill rule

Ready -> StartMiss -> SendFillReq -> WaitFillResp -> Ready

```
rule waitFillResp(mshr == WaitFillResp);
  let idx = getIdx(missReq.addr);
  let tag = getTag(missReq.addr);
  let data = memRespQ.first;
  tagArray.upd(idx, Valid (tag));
  if(missReq.op == Ld) begin
    dirtyArray.upd(idx, False); dataArray.upd(idx, data);
    hitQ.enq(data[wOffset]); end
  else begin data[wOffset] = missReq.data;
    dirtyArray.upd(idx, True); dataArray.upd(idx, data);
  end
  memRespQ.deq; mshr <= Ready;
endrule
```

Rest of the methods

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

```
method ActionValue#(MemReq) memReq;
  memReqQ.deq;
  return memReqQ.first;
endmethod
```

```
method Action memResp(Line r);
  memRespQ.enq(r);
endmethod
```

Memory side
methods

Hit and miss performance

◆ Hit

- Directly related to the latency of L1
- 0-cycle latency if hitQ is a bypass FIFO

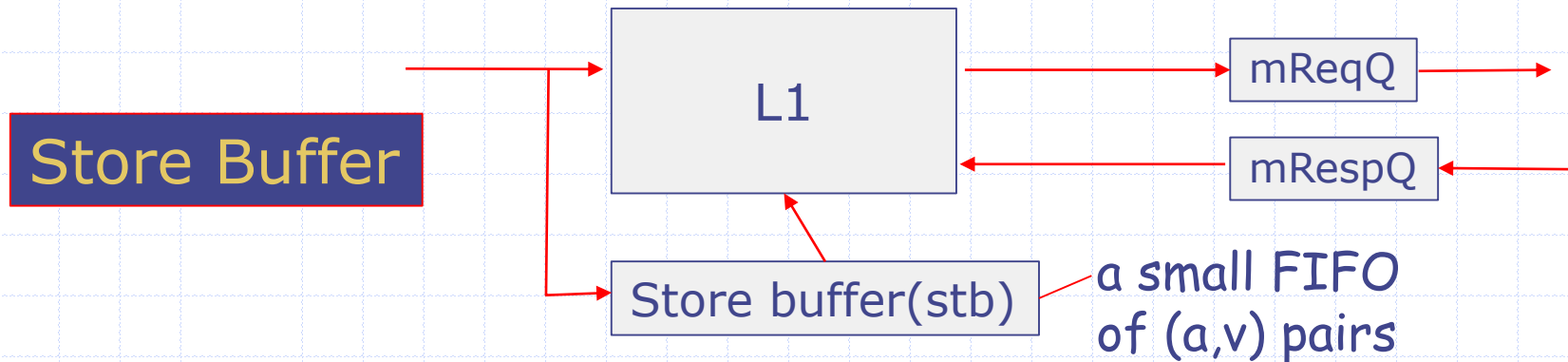
◆ Miss

- No evacuation: memory load latency plus combinational read/write
- Evacuation: memory store followed by memory load latency plus combinational read/write

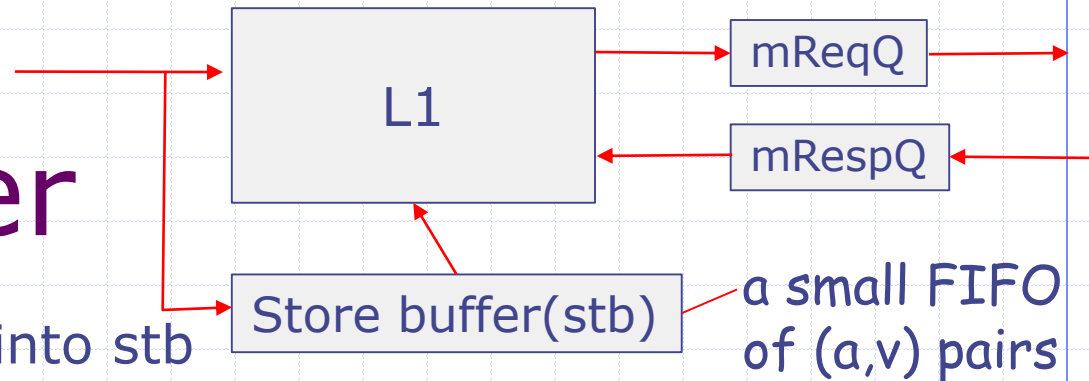
Adding a few extra cycles in the miss case does not have a big impact on performance

Speeding up Store Misses

- ◆ Unlike a load, a store does not require memory system to return any data to the processor; it only requires the cache to be updated for future load accesses
- ◆ Instead of delaying the pipeline, a store can be performed in the background; In case of a miss the data does not have to be brought into L1 at all (Write-miss no allocate policy)



Store Buffer



- ◆ A St req is enqueued into stb
 - input reqs are blocked if there is no space in stb
- ◆ A Ld req simultaneously searches L1 and stb;
 - If Ld gets a hit in stb – it selects the most recent matching entry; L1 search result is discarded
 - If Ld gets a miss in stb but a hit in L1, the L1 result is returned
 - If no match in either stb and L1, miss-processing commences
- ◆ In background, oldest store in stb is dequeued and processed
 - If St address hits in L1: update L1; if write-through then also send to it to memory
 - If it misses:
 - ◆ Write-back write-miss-allocate: fetch the cache line from memory (miss processing) and then process the store
 - ◆ Write-back/Write-through write-miss-no-allocate: pass the store to memory

L1+Store Buffer (write-back, write-miss-allocate): Req method

```
method Action req(MemReq r) if(mshr == Ready);
  ... get idx, tag and wOffset
  if(r.op == Ld) begin // search stb
    let x = stb.search(r.addr);
    if (isValid(x)) hitQ.enq(fromMaybe(?, x));
    else begin // search L1
      let currTag = tagArray.sub(idx);
      let hit = isValid(currTag) ?
        fromMaybe(?, currTag) == tag : False;
      if(hit) begin
        let x = dataArray.sub(idx); hitQ.enq(x[wOffset]); end
      else begin missReq <= r; mshr <= StartMiss; end
    end
  end
  else stb.enq(r.addr, r.data); // r.op == St
endmethod
```

No change in miss
handling rules

Entry into store buffer

L1+Store Buffer (write-back, write-miss-allocate): Exit from Store Buff

```
rule mvStbToL1 (mshr == Ready);
  stb.deq; match {.addr, .data} = stb.first;
                                // move the oldest entry of stb into L1
                                // may start allocation/evacuation
  ... get idx, tag and wOffset
  let currTag = tagArray.sub(idx);
  let hit = isValid(currTag) ?
            fromMaybe(?, currTag) == tag : False;
  if(hit) begin
    let x = dataArray.sub(idx); x[wOffset] = data;
    dataArray.upd(idx, x); dirtyArray.upd(idx, True); end
  else begin missReq <= r; mshr <= StartMiss; end
endrule
```

Give priority to req method in accessing L1

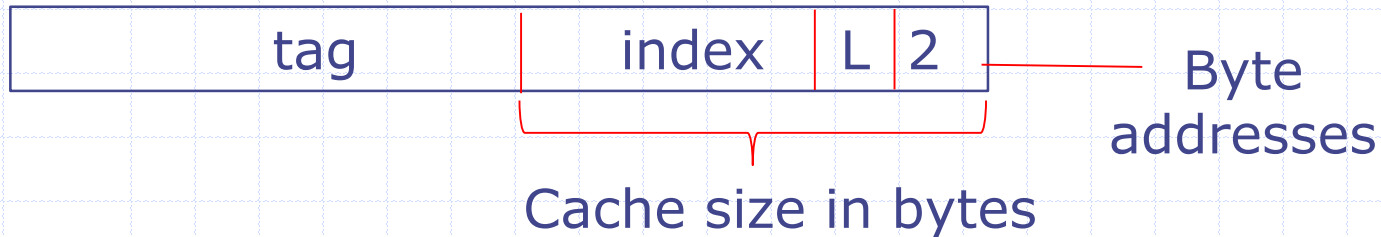
```
method Action req(MemReq r) if(mshr == Ready);  
    ... get idx, tag and wOffset  
if(r.op == Ld) begin // search stb  
    let x = stb.search(r.addr);  
    if (isValid(x)) hitQ.enq(fromMaybe(?, x));  
    else begin // search L1  
        ...  
    else stb.enq(r.addr, r.data) // r.op == St  
endmethod  
  
rule mvStbToL1 (mshr == Ready);  
    stb.deq; match {.addr, .data} = stb.first;  
    ... get idx, tag and wOffset  
endrule  
  
rule clearL1Lock; lockL1[1] <= False; endrule
```

lockL1[0] <= True;

Lock L1 while processing processor requests

&& !lockL1[1]

Functions to extract cache tag, index, word offset



```
function CacheIndex getIndex(Addr addr) = truncate(addr>>4);  
function Bit#(2) getOffset(Addr addr) = truncate(addr >> 2);  
function CacheTag getTag(Addr addr) = truncateLSB(addr);
```

truncate = truncateMSB