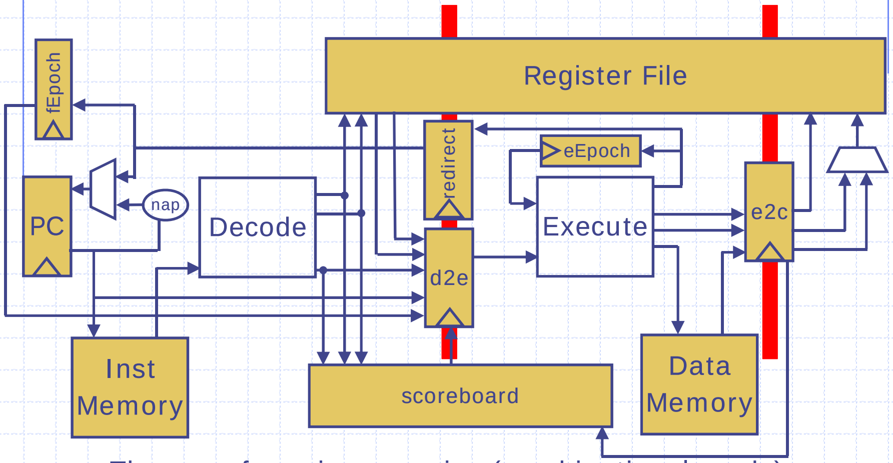
Constructive Computer Architecture

Realistic Memories and Caches

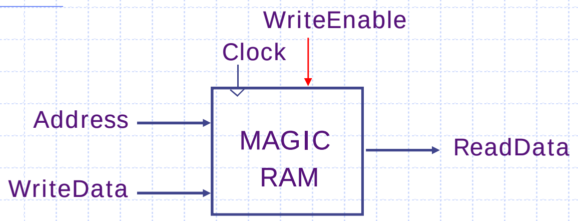
Arvind
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology

Multistage Pipeline



The use of magic memories (combinational reads) makes such design unrealistic

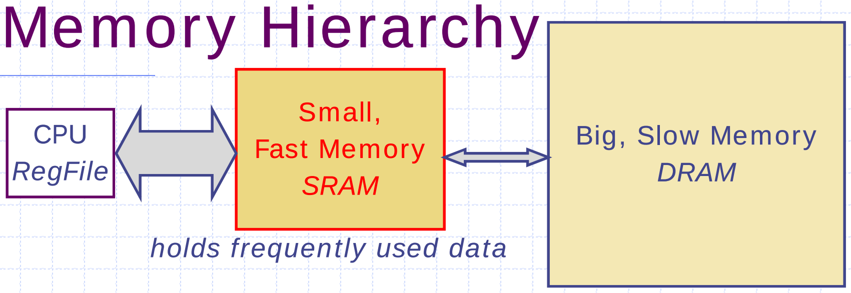
Magic Memory Model



- Reads and writes are always completed in one cycle
  - a Read can be done any time (i.e. combinational)
  - If enabled, a Write is performed at the rising clock edge
    (the write address and data must be stable at the clock edge)

In a real DRAM the data will be available several cycles after the address is supplied

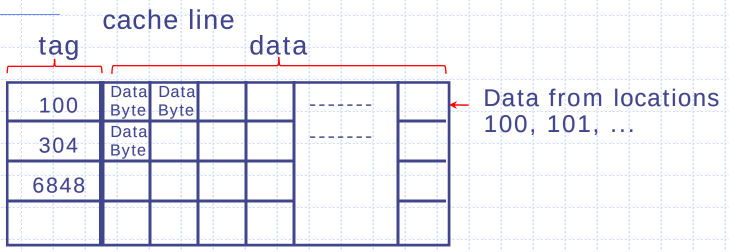
Memory Hierarchy



size:	RegFile	<<	SRAM	<<	DRAM
latency:	RegFile	<<	SRAM	<<	DRAM
bandwidth:	on-chip	>>	off-chip		

On a data access:
hit (data $\in$ fast memory) $\Rightarrow$ low latency access
miss (data $\notin$ fast memory) $\Rightarrow$ long latency access *(DRAM)*

Inside a Cache

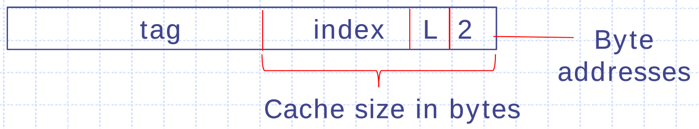


- A cache line usually holds more than one word
  - Spatial locality: if address x is referenced then addresses x+1, x+2 etc. are very likely to be referenced in the near future
    - consider instruction streams, array and record accesses
  - Larger data sets can be transported more efficiently
  - Reduces the number of tag bits needed to identify a cache line

Internal Cache Organization

- Cache designs restrict where in cache a particular address can reside
  - *Direct mapped:* An address can reside in exactly one location in cache. The cache location is typically determined by the lowest order address bits
  - *n-way Set associative:* An address can reside in any of a set of n locations in the cache. The set is typically determine by the low order address bits

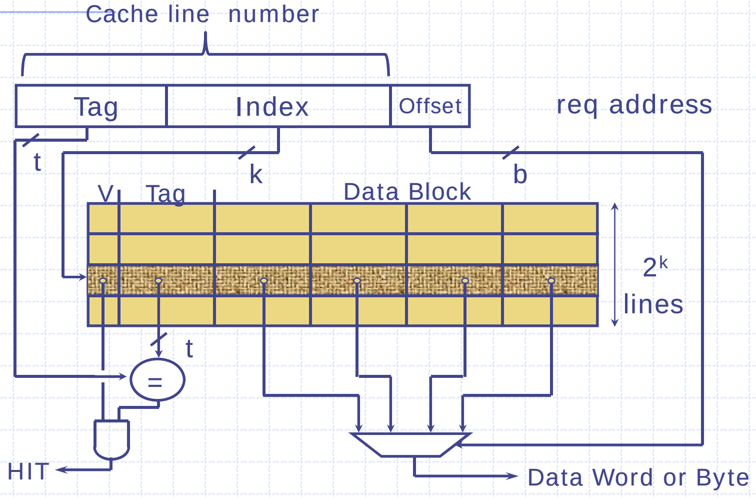
Extracting cache tags & index



- 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
- Suppose cache size $=2^{(K+L+2)}$ bytes
  - Direct mapped cache: Index=K; tag=32-(K+L+2)
  - 2-Way set-associative cache: Index=K-1; tag = 32-(K-1+L+2)
- Need getIndex, getTag, getOffset functions

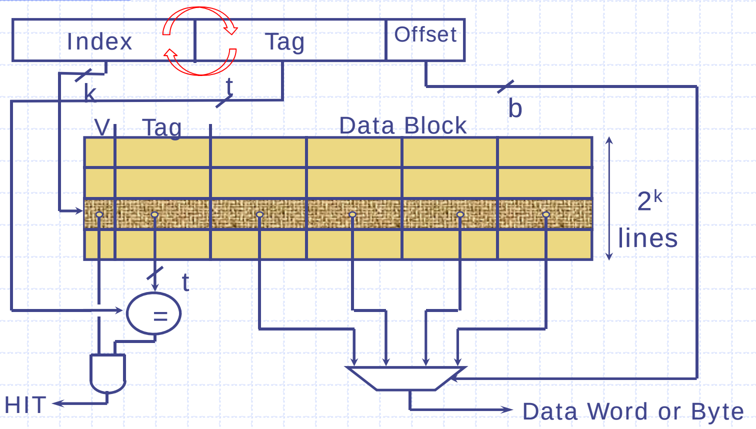
Direct-Mapped Cache

The simplest implementation



Direct Map Address Selection

higher-order vs. lower-order address bits

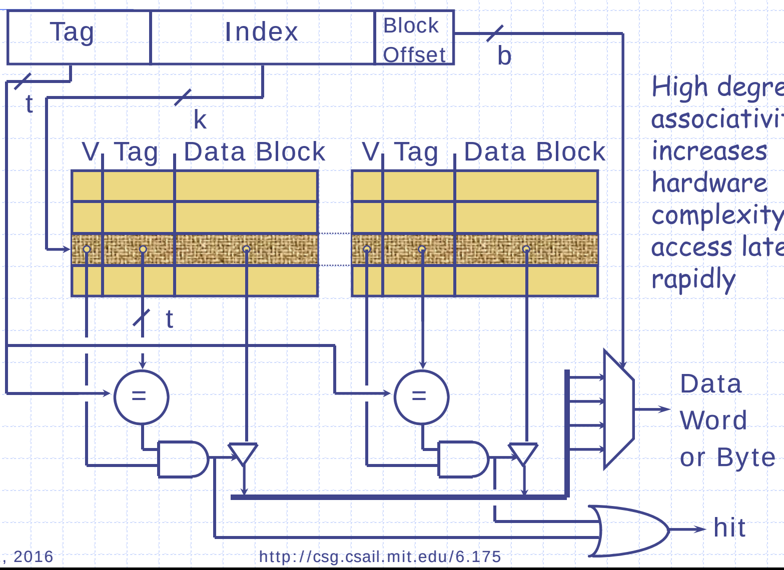


Why higher-order bits as tag may be undesirable?

Conflict Misses

- In a direct map cache, a cache-line can be stored only if the cache slot corresponding to its address is not occupied (even if there are other unoccupied slots)
- In a n-way set associate cache, a cache line can be stored in n different slots - reduces misses due to address conflicts

2-Way Set-Associative Cache



High degree of associativity increases hardware complexity and access latency rapidly

Loads

Search the cache for the processor generated address

Found in cache
a.k.a. hit

Return a copy of the word from the cache-line

Not in cache
a.k.a. miss

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-back policy: write only to cache and update the next level memory when line is evacuated
  - Write-through policy: write to both the cache and the next level memory
- On a write miss
  - Allocate – because of multi-word lines we first fetch the line, and then update a word in it
  - No allocate – cache is not affected, the Store is forwarded to memory

Typically one uses either *Write-back-Allocate* policy or *Write-through-No-allocate* policy

Replacement Policy

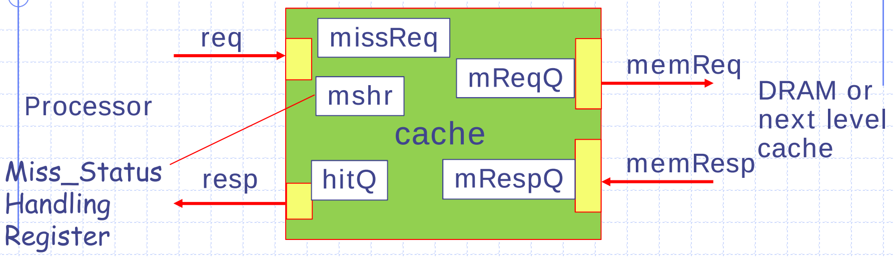
- To bring in a new cache line, usually another cache line has to be thrown out. Which one?
  - Direct mapped cache: No choice
  - N-way set associative cache: Choice of policy
    - One that is not dirty, i.e., has not been modified. In I-cache all lines are clean;
    - If there is still a choice of more than one then Least Recently Used (LRU)? Most Recently Used? Random?

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

Blocking Cache Interface



```
interface Cache;
  method Action req(MemReq r);
  method ActionValue#(Data) resp;

  method ActionValue#(MemReq) memReq;
  method Action memResp(Line r);
endinterface
```

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
- 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 cache while it is processing a miss

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

Blocking cache
state elements

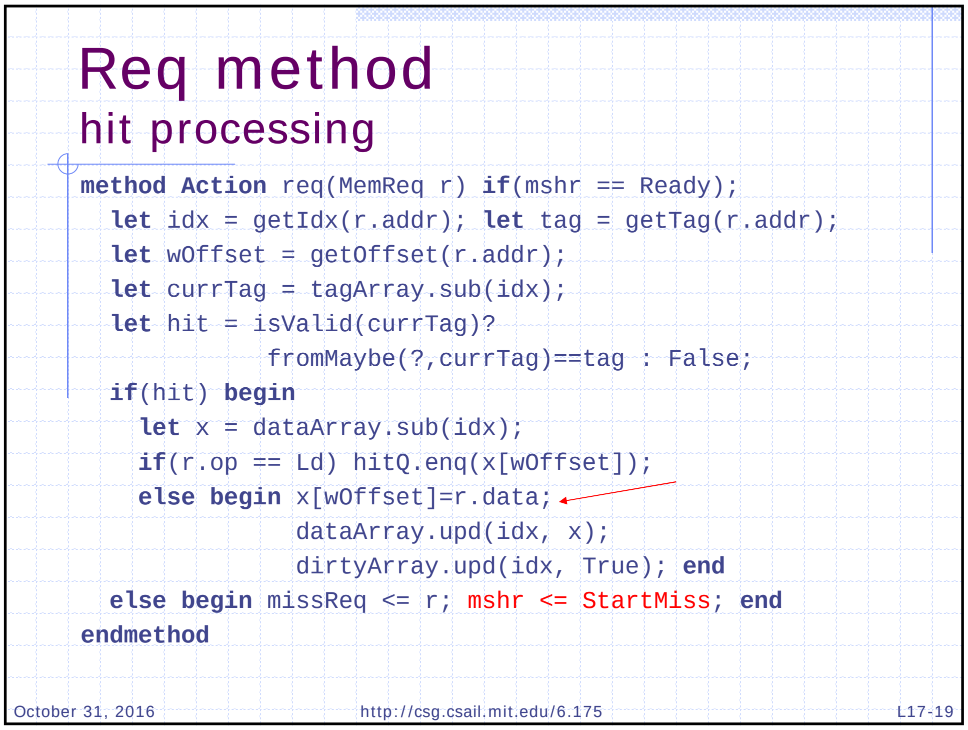
```
RegFile#(CacheIndex, Line) dataArray <- mkRegFileFull;
RegFile#(CacheIndex, Maybe#(CacheTag))
                                tagArray <- mkRegFileFull;
RegFile#(CacheIndex, Bool) dirtyArray <- mkRegFileFull;

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

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

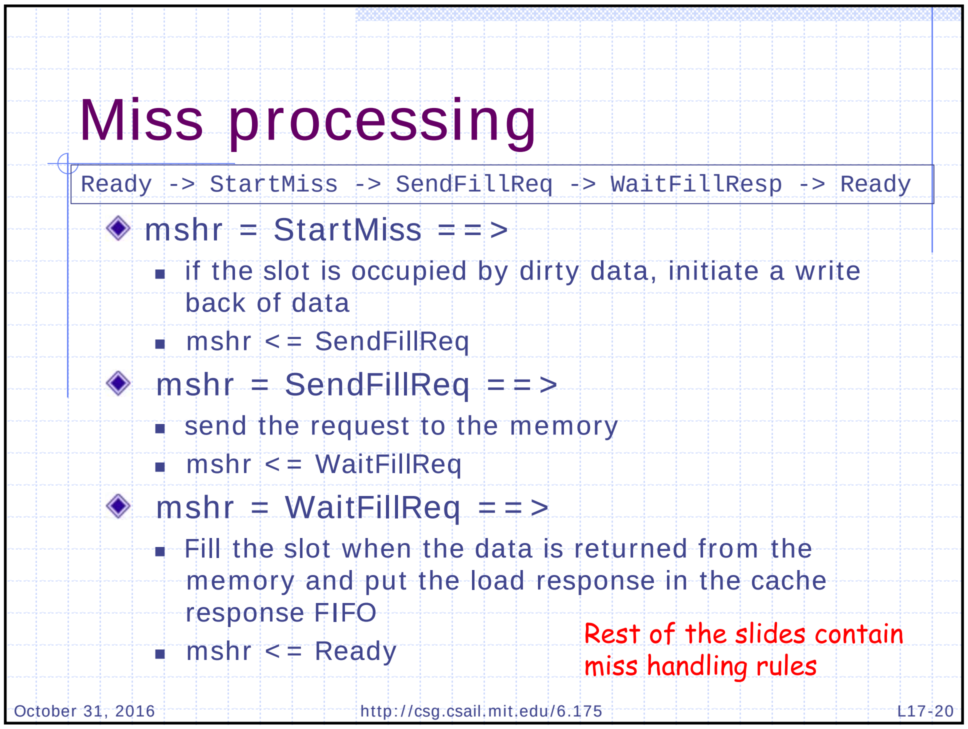
Tag and valid bits are kept together as a Maybe type

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



Req method
hit processing

```
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
```



Miss processing

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

- mshr = StartMiss ==>
  - if the slot is occupied by dirty data, initiate a write back of data
  - mshr <= SendFillReq
- mshr = SendFillReq ==>
  - send the request to the memory
  - mshr <= WaitFillReq
- mshr = WaitFillReq ==>
  - Fill the slot when the data is returned from the memory and put the load response in the cache response FIFO
  - mshr <= Ready

Rest of the slides contain miss handling rules

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

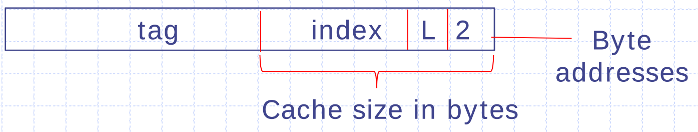
```
method ActionValue#(Data) resp;
  hitQ.deq;
  return hitQ.first;
endmethod

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

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

Memory side methods

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
```