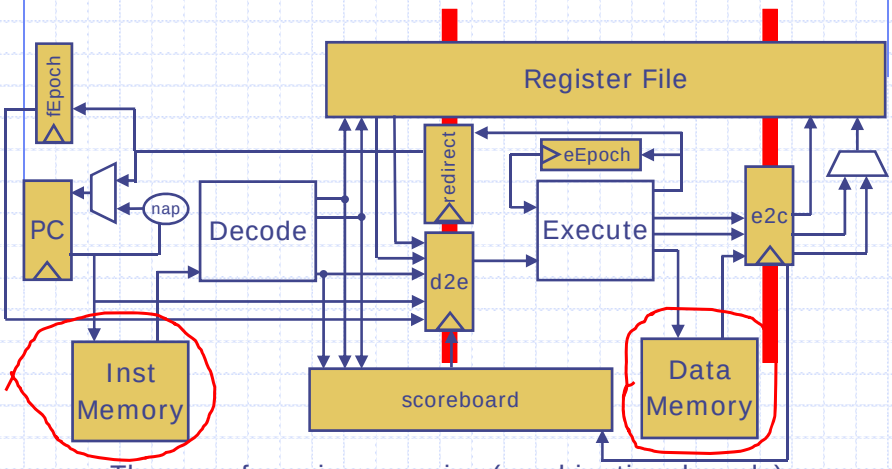
Realistic Memories and Caches

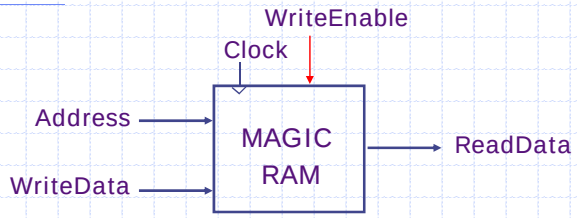
Arvind
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology

Multistage Pipeline



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

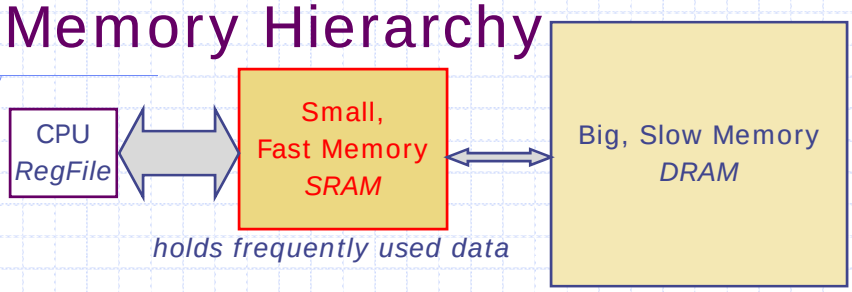
Magic Memory Model



- 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		

why?

On a data access:

hit (data ∈ fast memory) ⇒ low latency access

miss (data ∉ fast memory) ⇒ long latency access *(DRAM)*

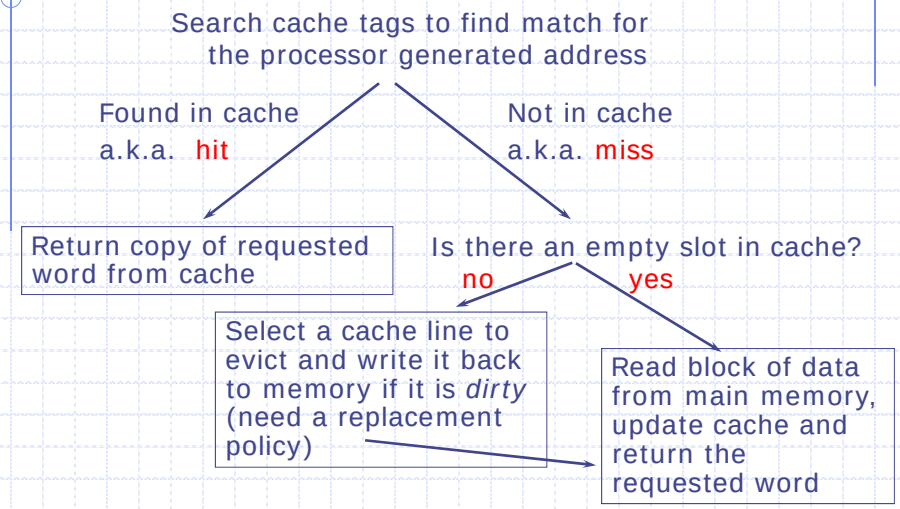
Cache organization

- *Temporal locality:* A recently accessed address has a much higher probability of being accessed in the near future than other addresses
- *Spatial locality:* If address a is accessed then locations in the neighborhood of a, e.g., a-1, a , a+1, a+2, are also accessed with high probability
  - Therefore processor caches are almost always organized in terms of cache lines which are typically 4 to 8 words
  - It is also more efficient to transfer cache lines as opposed to words to the main memory

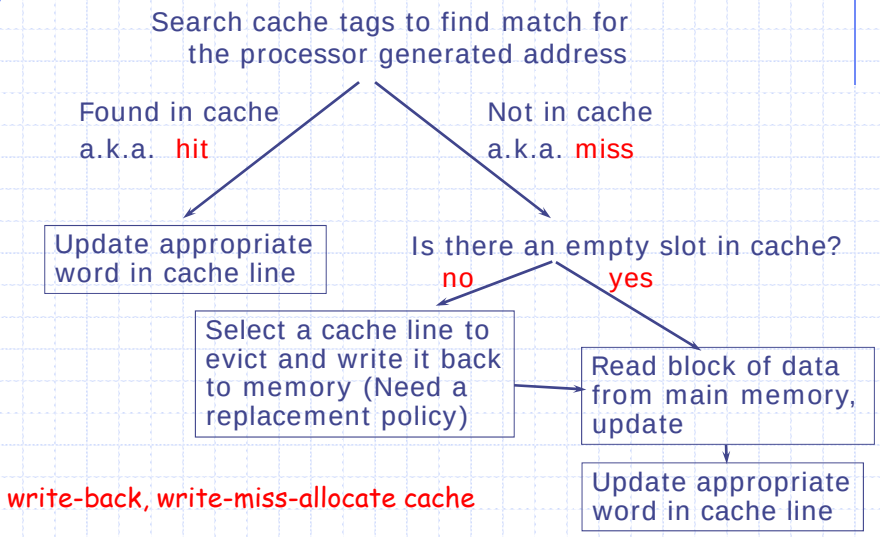
cache line = <Add tag, Data blk>

If the line size is 4 words then the address tag is 4 bits shorter than the byte address and the data block size is 4 words

Memory Read behavior



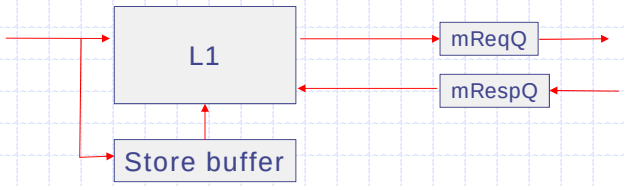
Memory Write behavior



Store Buffer: Speeding up Store Misses

- A 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
- A store can be performed in the background; In case of a store miss, the miss can be processed even after the store instruction has retired from the processor pipeline

Store Buffer



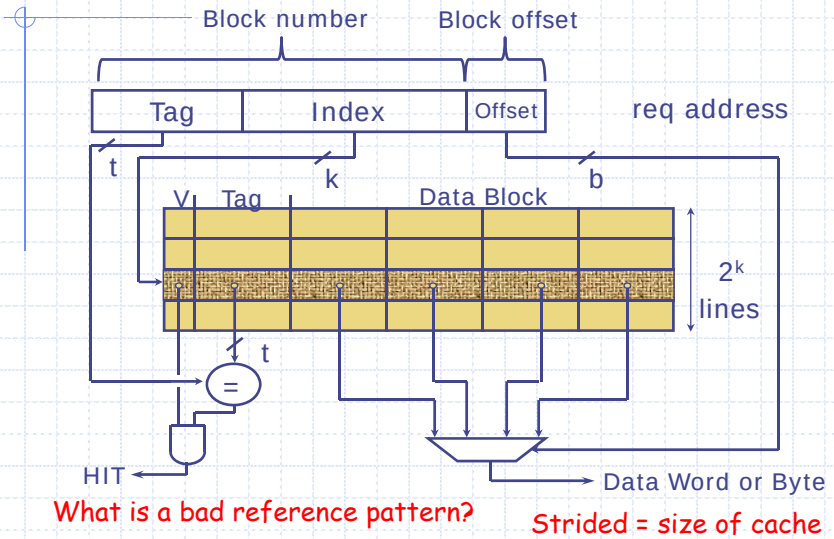
- Store Buffer (stb) is a small FIFO of (a,v) pairs
- A St req is enqueued into stb
  - if there is no space in stb further input reqs are blocked
- Later a St in stb is stored into L1
- A Ld req simultaneously searches L1 and stb; in case of a miss the request is processed as before
  - Can get a hit in at both places; stb has priority
  - A Ld can get multiple hits in stb – it must select the most recent matching entry

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 the cache. The cache location is typically determined by the lowest order address bits
  - *n-way Set associative:* An address can reside in any of the a set of n locations in the cache. The set is typically determine by the lowest order address bits

Direct-Mapped Cache

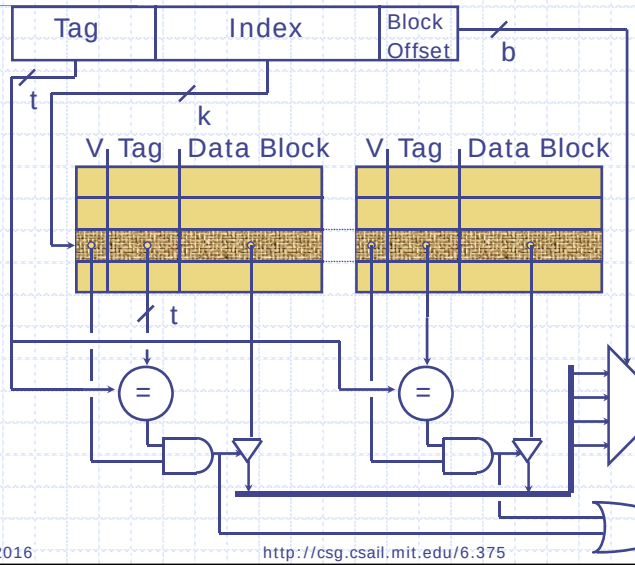
The simplest implementation



What is a bad reference pattern?

Strided = size of cache

2-Way Set-Associative Cache



Reduces conflict misses by allowing a cache line to go to several different slots in a cache

Replacement Policy

- In order to bring in a new cache line, another cache line may have to be thrown out. Which one?
  - No choice in replacement in direct-mapped caches
  - For set-associative caches, select a set from the index
    - Select the least recently used, or most recently used, random …
    - Select a not dirty set

How much is performance affected by the choice?

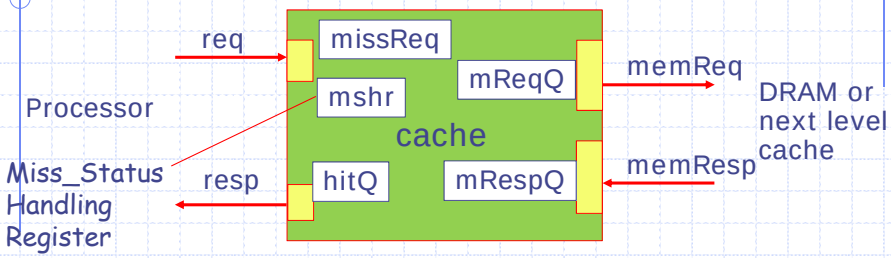
Difficult to know without benchmarks and quantitative measurements

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
```

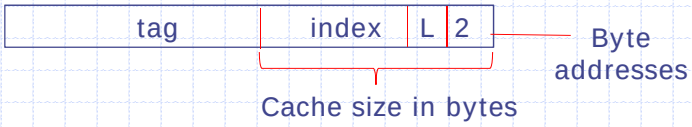
We will design a write-back, Write-miss allocate, Direct-mapped, blocking cache, first without and then with store buffer

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., the 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);
```

Extracting cache tags & index



- Processor requests are for a single word but internal communications are in line sizes (2^L words, typically L=2)
- AddrSz = CacheTagSz + CacheIndexSz + LS + 2
- Need getIndex, getTag, getOffset functions

```
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
```


Blocking cache
state elements

```
Vector#(CacheSize, Reg#(Line)) dataArray <-
                                    replicateM(mkRegU);
Vector#(CacheSize, Reg#(Maybe#(CacheTag))) tagArray <-
                        replicateM(mkReg(tagged Invalid));
Vector#(CacheSize, Reg#(Bool)) dirtyArray <-
                                  replicateM(mkReg(False));

Fifo#(2, Data)      hitQ <- mkCFFifo;
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

It is straightforward to extend the cache interface to include a cacheline flush command

```
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[idx];
  let hit = isValid(currTag)?
            fromMaybe(?,currTag)==tag : False;
  if(hit) begin
    let x = dataArray[idx];
    if(r.op == Ld) hitQ.enq(x[wOffset]);
    else begin x[wOffset]=r.data;
               dataArray[idx] <= x;
               dirtyArray[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

- 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

Start-miss and Send-fill rules

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

```
rule startMiss(mshr == StartMiss);
  let idx = getIdx(missReq.addr);
  let tag=tagArray[idx]; let dirty=dirtyArray[idx];
  if(isValid(tag) && dirty) begin // write-back
    let addr = {fromMaybe(?,tag), idx, 4'b0};
    let data = dataArray[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[idx] <= Valid(tag);
  if(missReq.op == Ld) begin
    dirtyArray[idx] <= False; dataArray[idx] <= data;
    hitQ.enq(data[wOffset]); end
  else begin data[wOffset] = missReq.data;
    dirtyArray[idx] <= True; dataArray[idx] <= data;
        end
  memRespQ.deq; mshr <= Ready;
endrule
```

Rest of the methods

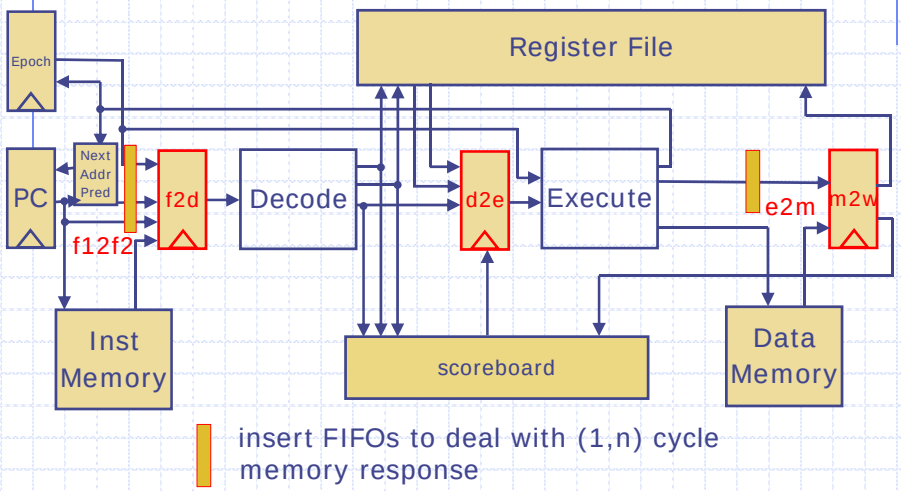
```
method ActionValue#(Data) resp;
  hitQ.deq;
  return hitQ.first;
endmethod

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

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

Memory side methods

Caches: Variable number of cycles in memory access pipeline stages



Store Buff: Req method
hit processing

```
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[idx];
     let hit = isValid(currTag) ?
               fromMaybe(?,currTag)==tag : False;
     if(hit) begin
       let x = dataArray[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
```


Store Buff to mReqQ

```
rule mvStbToL1 (mshr == Ready);
  stb.deq; match {.addr, .data} = stb.first;
  ... get idx, tag and wOffset
  let currTag = tagArray[idx];
  let hit = isValid(currTag) ?
            fromMaybe(?,currTag)==tag : False;
  if(hit) begin
     let x = dataArray[idx];
     x[wOffset] = data; dataArray[idx] <= x; end
  else begin missReq <= r; mshr <= StartMiss; end
endrule
```

may cause a simultaneous access to L1 cache arrays, because of load requests

Preventing simultaneous accesses to 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;
```

L1 needs to be locked even if the hit is in stb

```
&& !lockL1[1]
```


Memory System

- All processors use store buffers in conjunction with caches
- Most systems today use non-blocking caches, which are more complicated than the blocking cache described here
- The organization we have described is similar to the one used by Intel
- IBM and ARM use a different caching policy known as *write-through*, which simultaneously updates L1 and sends a message to update the next level cache