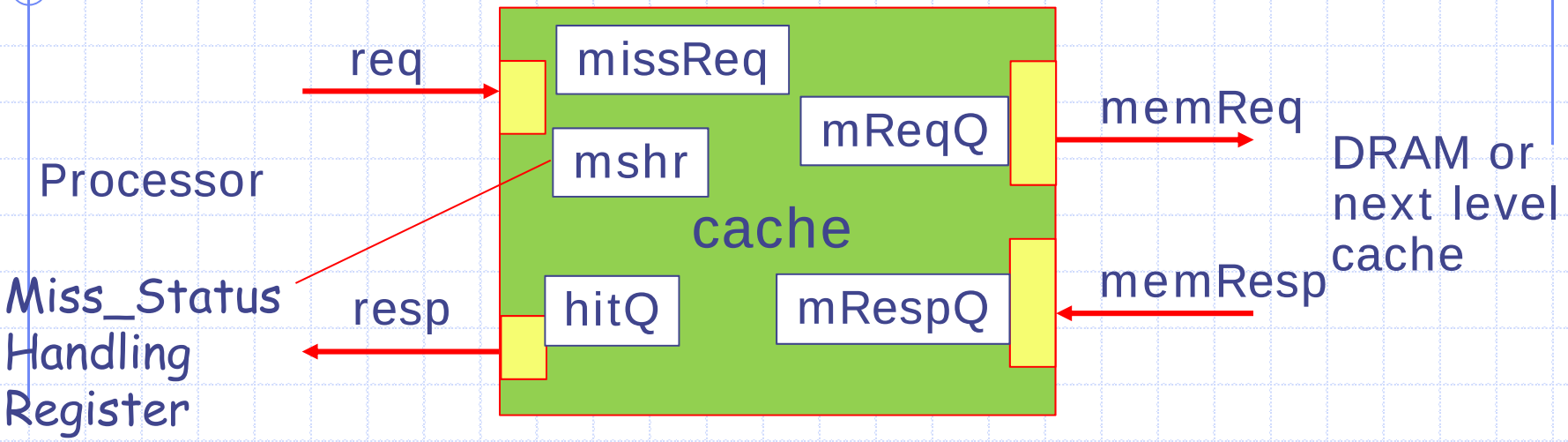
Constructive Computer Architecture

Caches-2

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

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

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);
```

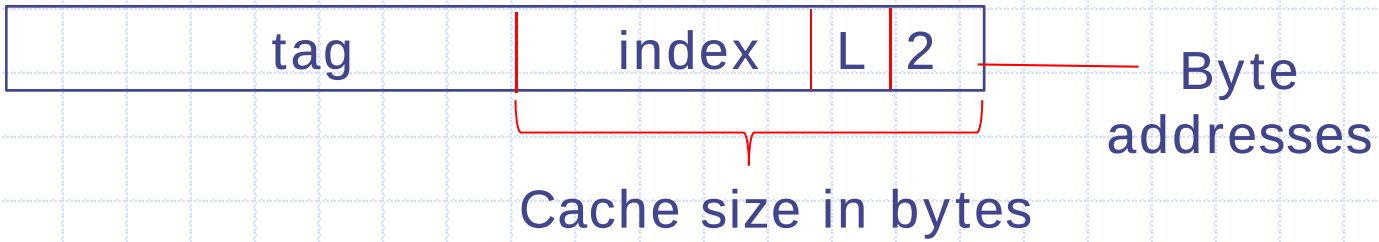
Blocking Cache code structure

```
module mkCache(Cache);
    RegFile#(CacheIndex, Line) dataArray <-
                                    mkRegFileFull; …
    rule startMiss … endrule;

    method Action req(MemReq r) …          endmethod;
    method ActionValue#(Data) resp …       endmethod;

    method ActionValue#(MemReq) memReq … endmethod;
    method Action memResp(Line r) …        endmethod;
endmodule
```

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

```
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

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);
  Bit#(2) wOffset = truncate(r.addr >> 2);
  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
```

- 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.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

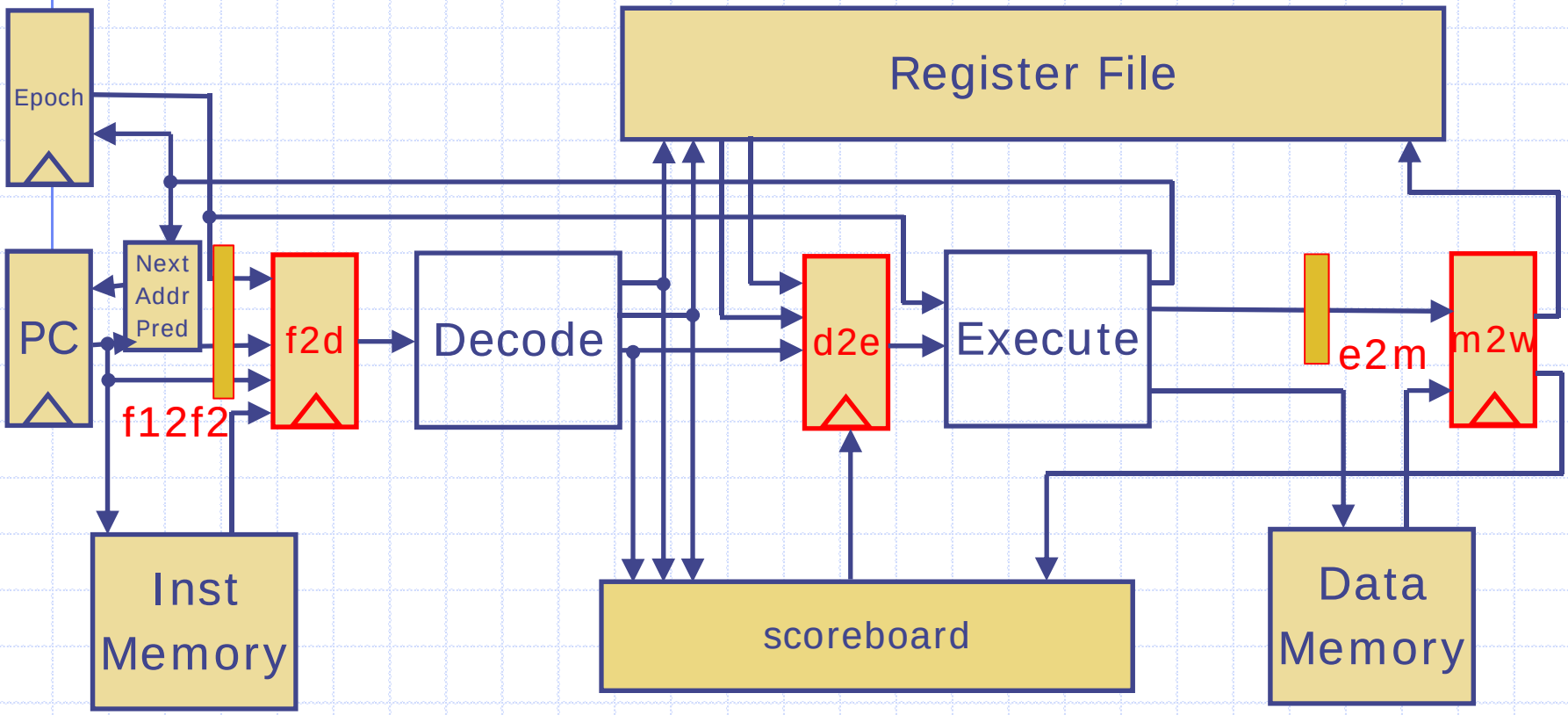
```
method ActionValue#(Data) resp;
  hitQ.deq;
  return hitQ.first;
endmethod

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

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

Memory side methods

Caches: Variable number of cycles in memory access pipeline stages



insert bypass FIFO's to deal with (0,n) cycle memory response

Hit and miss performance

- Hit
  - For combinational reads, `req` and `resp` methods have to be concurrently schedulable,

    `hitQ.enq < {hitQ.deq, hitQ.first}`

    i.e., `hitQ` should be 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

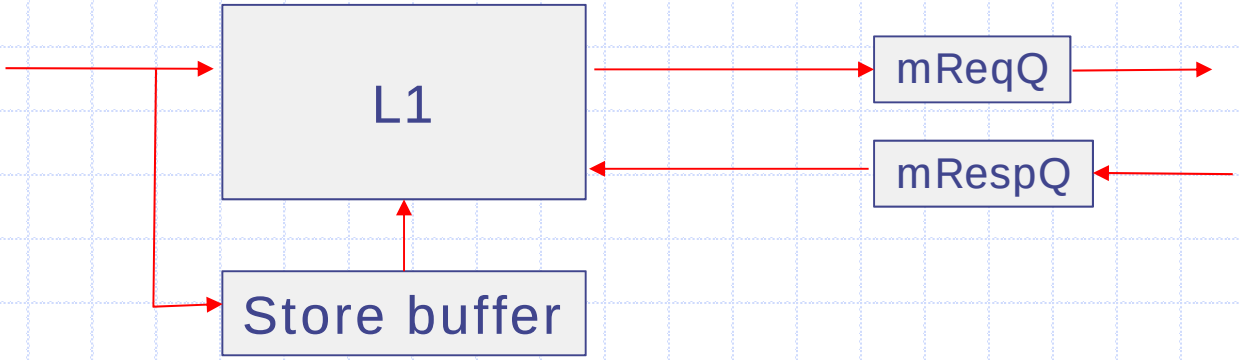
Adding an extra cycle here and there in the miss case should not have a big negative performance impact

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
- 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)

We will first study Write-miss allocate and then Write-miss no allocate

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

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.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
```

Store Buff to mReqQ

```
rule mvStbToL1 (mshr == Ready);
  stb.deq; match {.addr, .data} = stb.first;
  ... 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) end
  else begin missReq <= r; mshr <= StartMiss; end
endrule
```

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]
```

Store Buff: Req method
hit processing (write-miss no allocate)

```
method Action req(MemReq r) if(mshr == Ready);
  ... get idx, tag and wOffset
 if(r.op == Ld) begin // search stb
    lockL1[0] <= True;
    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

Store Buff to mReqQ

(write-miss no allocate)

```
rule mvStbToL1 (mshr == Ready && !lockL1[1]);
  stb.deq; match {.addr, .data} = stb.first;
  ... 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) end
  else begin missReq <= r; mshr <= StartMiss; end
endrule
```

```
                missReq = MemReq'{op: St, addr: addr,
                                  lineMask: 1<<wOffset,
                                  data: replicate(data)};
                memReqQ.enq(missReq);
```

Types are quite correct