Constructive Computer Architecture

Caches-2

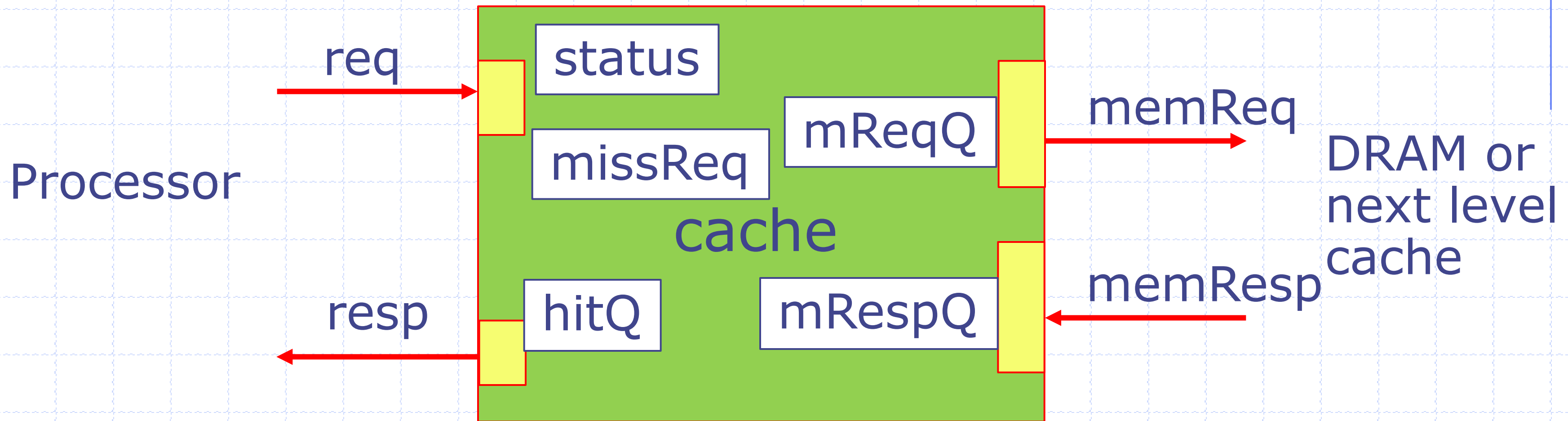
Arvind
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology

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
- Requests and responses follow the FIFO order
- Methods are guarded, e.g., the cache may not be ready to accept a request because it is processing a miss
- A status 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);
```

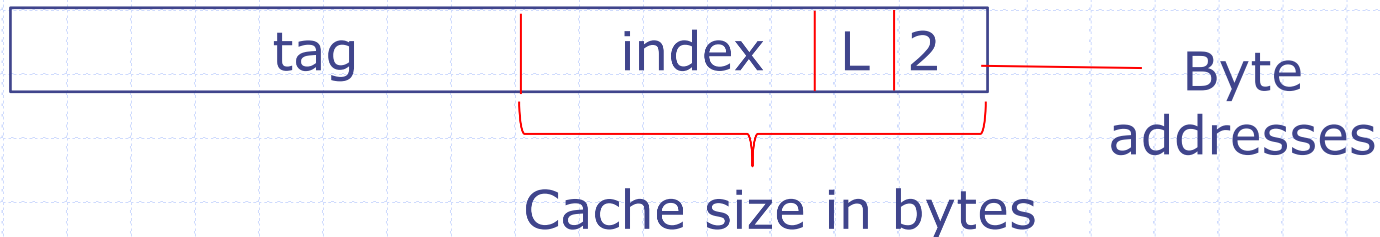
Blocking Cache code structure

```
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 getIdx, getTag, getOffset functions

```
function CacheIndex getIdx(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) status <- 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(status == 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; status <= StartMiss; end
endmethod
```

overwrite the appropriate word of the line

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

Start-miss and Send-fill rules

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

```
rule startMiss(status == 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
  status <= SendFillReq;
endrule
```

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

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

Wait-fill rule

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

```
rule waitFillResp(status == 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; status <= Ready;
endrule
```

Hit and miss performance

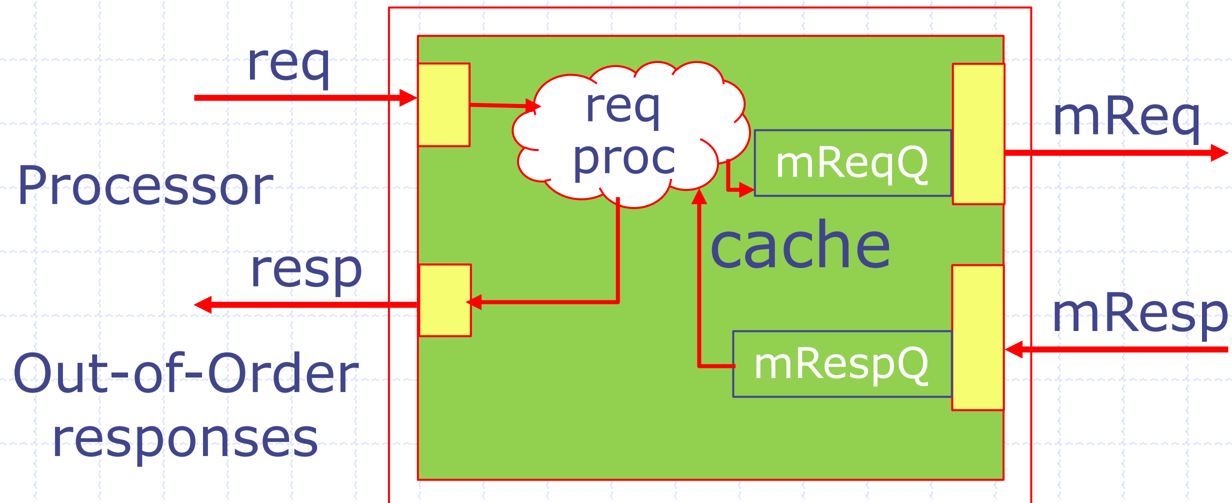
- Hit
  - Combinational read/write, i.e. 0-cycle response
  - Requires `req` and `resp` methods to be concurrently schedulable, which in turn requires

    `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

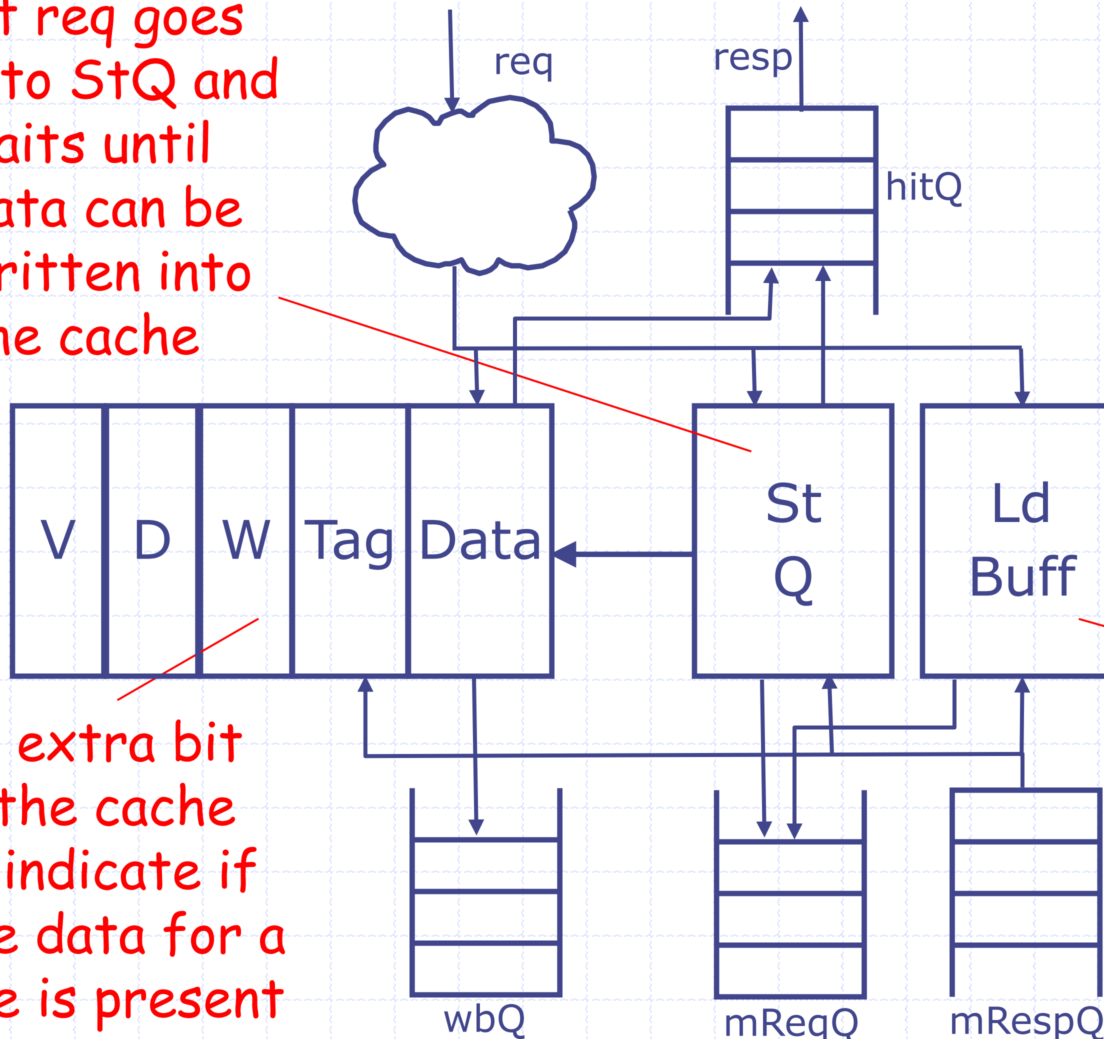
Non-blocking cache



- Requests have to be tagged because responses come out-of-order (OOO)
- We will assume that all tags are unique and the processor is responsible for reusing tags properly

Non-blocking Cache

St req goes into StQ and waits until data can be written into the cache

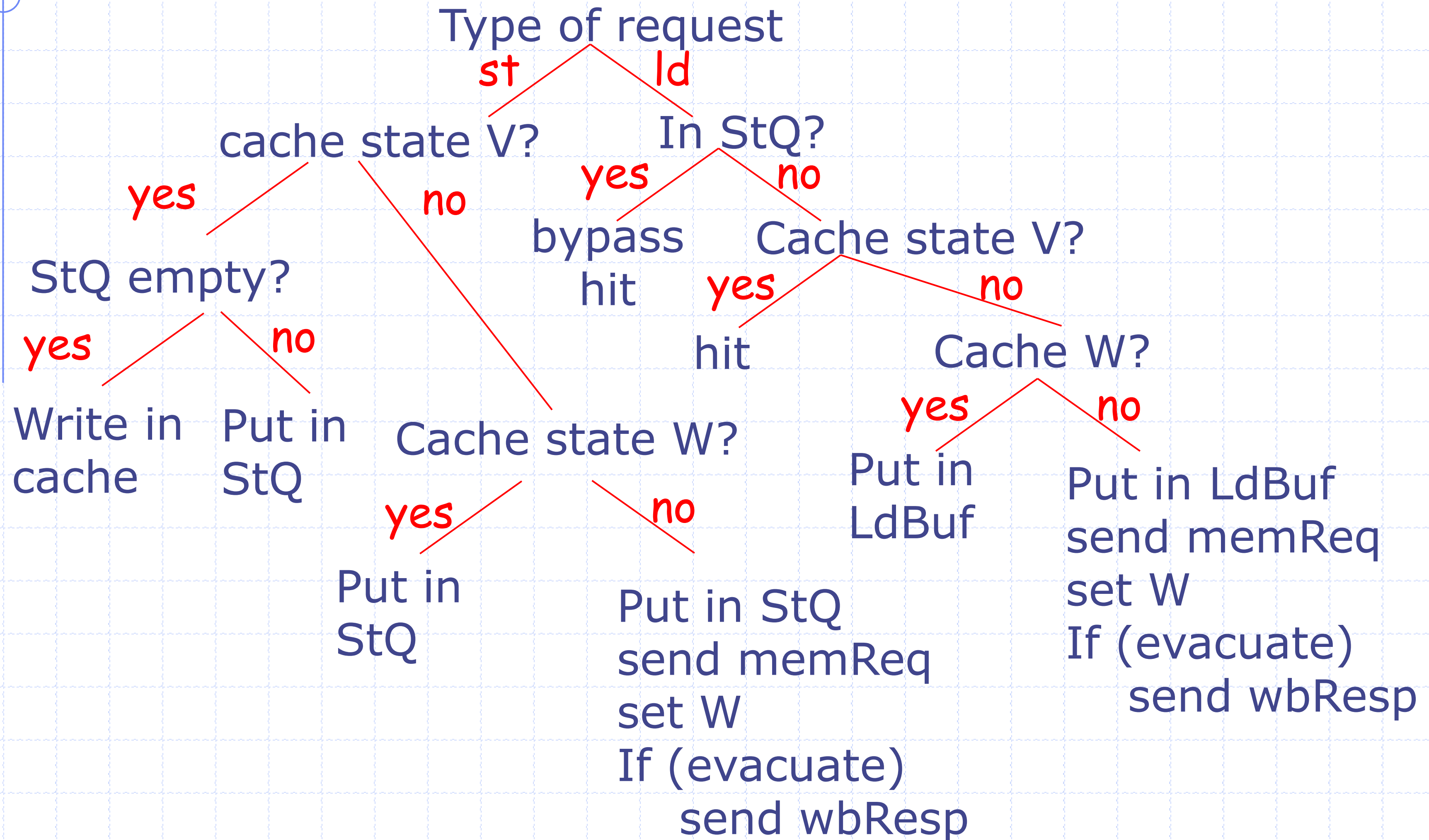


An extra bit in the cache to indicate if the data for a line is present

Behavior to be described by 2 concurrent FSMs to process input requests and memory responses, respectively

load reqs waiting for data

Incoming req



Mem Resp (line)

1. Update cache line (set V and unset W)
2. Process all matching ldBuff entries and send responses
3. L: If cachestate(oldest StQ entry address) = V

```
        then
           update the cache word with StQ entry;
           remove the oldest entry;
           Loop back to L
        else if cachestate(oldest StQ entry address) = !W
              then if(evacuate) wbResp;
                     memReq for this store entry;
                     set W
```