Constructive Computer Architecture

Cache Coherence

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

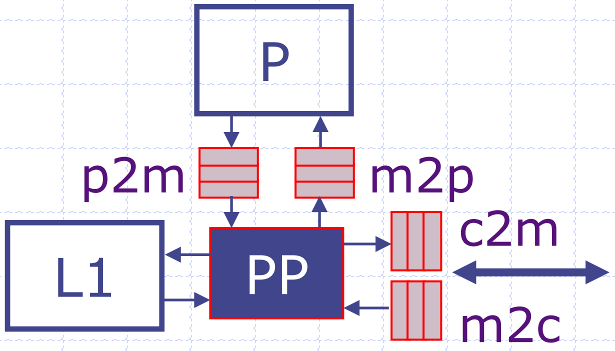
Plan

- The invalidation protocol
- Non-blocking L1

Processor Hit Rules

- Load-hit rule

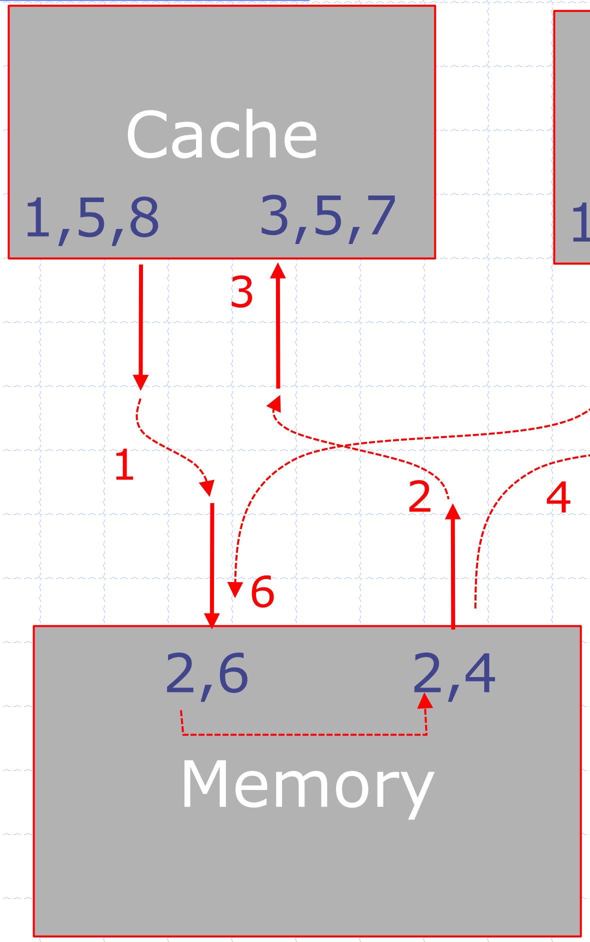
```
    p2m.msg=(Load a) &
    c.tag[cs(a)]=tag(a) &
    c.state[cs(a)]>I
→   p2m.deq;
    m2p.enq(c.data[cs(a)]);
```



- Store-hit rule

```
    p2m.msg=(Store a v) &
    c.tag[cs(a)]=tag(a) &
    c.state[cs(a)]=M
→   p2m.deq;
    m2p.enq(Ack);
    c.data[cs(a)]:=v;
```

Processing misses: Requests and Responses



1 Up-req send (cache)
2 Up-req proc, Up resp send (memory)
3 Up-resp proc (cache)
4 Dn-req send (memory)
5 Dn-req proc, Dn resp send (cache)
6 Dn-resp proc (memory)
7 Dn-req proc, drop (cache)
8 Voluntary Dn-resp (cache)

Invariants for a CC-protocol design

- Directory state is always a conservative estimate of a child’s state
  - E.g., if directory thinks that a child cache is in S state then the cache has to be in either I or S state
- For every request there is a corresponding response, though sometimes it is generated even before the request is processed
- Communication system has to ensure that
  - responses cannot be blocked by requests
  - a request cannot overtake a response for the same address
- At every merger point for requests, we will assume fair arbitration to avoid starvation

Child Requests

1. Child to Parent: Upgrade-to-y Request

```
c.tag[cs(a)]!=tag(a) & c.state[cs(a)]=I &
c.waitp[cs(a)]=No
→ c.tag[cs(a)]:= tag(a);
   c.waitp[cs(a)]:=Yes y;
   c2m.enq(<Req, c→m, a, y, - >);
```

A request is never sent unless the cache has a slot and the slot contains the tag

cs(a) is the cache slot address

```
c.tag[cs(a)]=tag(a) & (c.state[cs(a)]< y) &
c.waitp[cs(a)]=No
→ c.waitp[cs(a)]:=Yes y;
   c2m.enq(<Req, c→m, a, y, - >);
```

These rules are mutually exclusive and can be combined.
This rule would normally be triggered by a cache miss.

Parent Responds

2. Parent to Child: Upgrade-to-y response

```
(∀j, m.waitc[j][a]=No) & c2m.msg=<Req,c→m,a,y,-> &
(∀i≠c, IsCompatible(m.child[i][a],y))
→ m2c.enq(<Resp, m→c, a, y,
                (if (m.child[c][a]=I) then m.data[a] else -)>);
    m.child[c][a]:=y; c2m.deq;
```

Child receives Response

3. Child receiving upgrade-to-y response

```
m2c.msg=<Resp, m→c, a, y, data>
 → m2c.deq;
    if(c.state[cs(a)]=I) c.data[cs(a)]:=data;
    c.state[cs(a)]:=y;
    c.waitp[cs(a)]:=No;
// the child must be waiting for state y
```

Parent Requests

4. Parent to Child: Downgrade-to-y Request

```
c2m.msg=<Req,c→m,a,y,-> &
(m.child[i][a]>y) & (m.waitc[i][a]=No)
→ m.waitc[i][a]:=Yes y;
   m2c.enq(<Req, m→c, a, y, - >);
```

Child Responds

5. Child to Parent: Downgrade-to-y response

```
    (m2c.msg=<Req,m→c,a,y,->) &
    c.state[cs(a)]>y &
    c.tag[cs(a)]=tag(a)
    → c2m.enq(<Resp, c->m, a, y,
                 (if (c.state[cs(a)]=M) then c.data[a] else -)>);
      c.state[cs(a)]:=y; m2c.deq
```

Parent receives Response

6. Parent receiving downgrade-to-y response

```
c2m.msg=<Resp, c→m, a, y, data>
 → c2m.deq;
   if(m.child[c][a]=M) m.data[a]:=data;
   m.child[c][a]:=y;
   if(m.waitc[c][a]=(Yes x) & x≥y)
                         m.waitc[c][a]:=No;
```

Child receives served Request

7. Child receiving downgrade-to-y request

```
(m2c.msg=<Req, m→c, a, y, - >) &
((c.tag[cs(a)]=tag(a) & c.state[cs(a)]≤y)
|| c.tag[cs(a)]!=tag(a))
→ m2c.deq;
```

Child Voluntarily downgrades

8. Child to Parent: Downgrade-to-y response (vol)

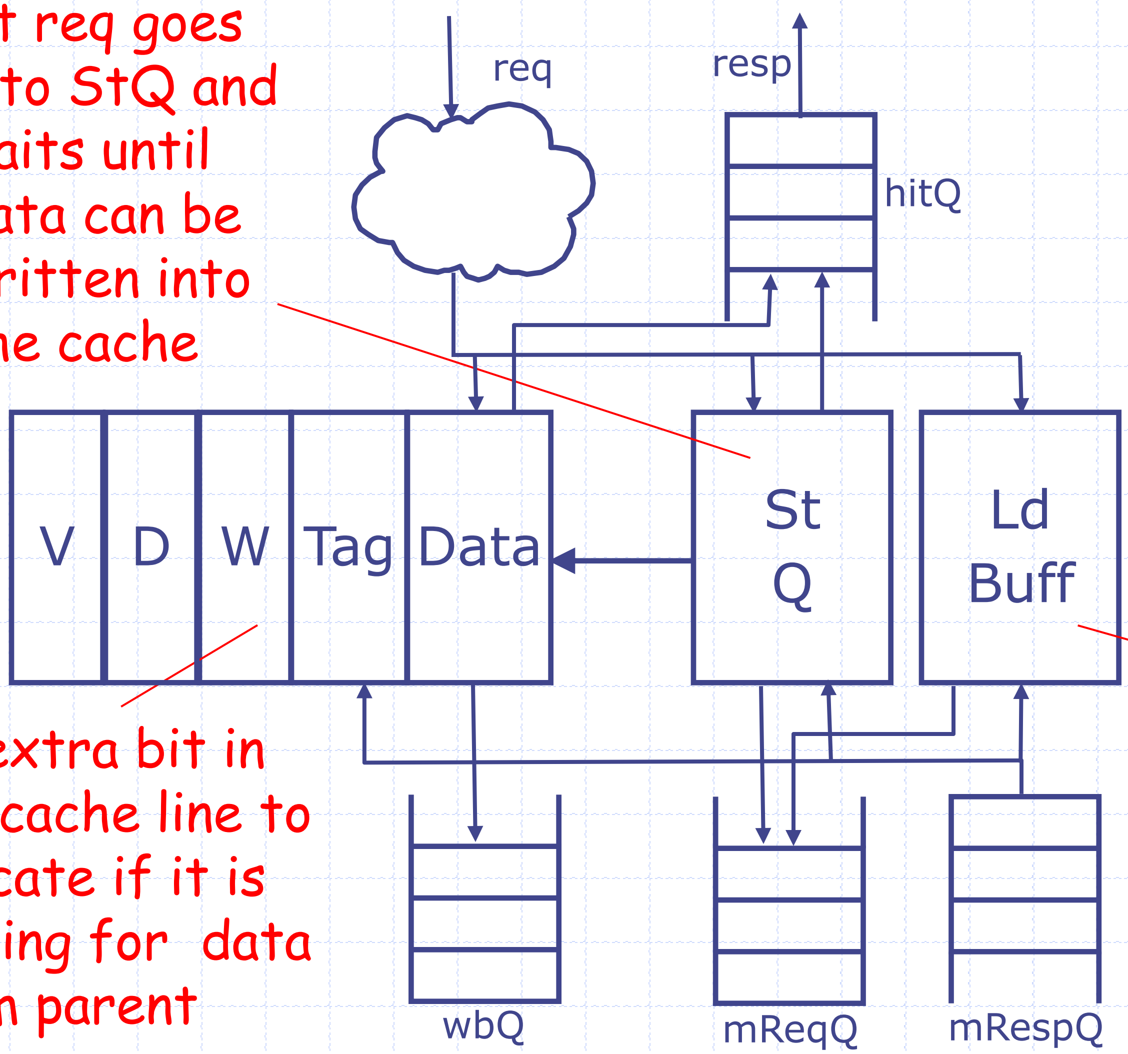
```
(c.waitp[cs(a)]=No) & (c.state[cs(a)]>y)
→ c2m.enq(<Resp, c->m, a, y,
               (if (c.state[cs(a)]=M) then c.data[a] else -)>);
   c.state[cs(a)]:=y;
```

Rules 1 to 8 are complete - cover all possibilities and cannot deadlock or violate cache invariants

Non-blocking Cache

single processor

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



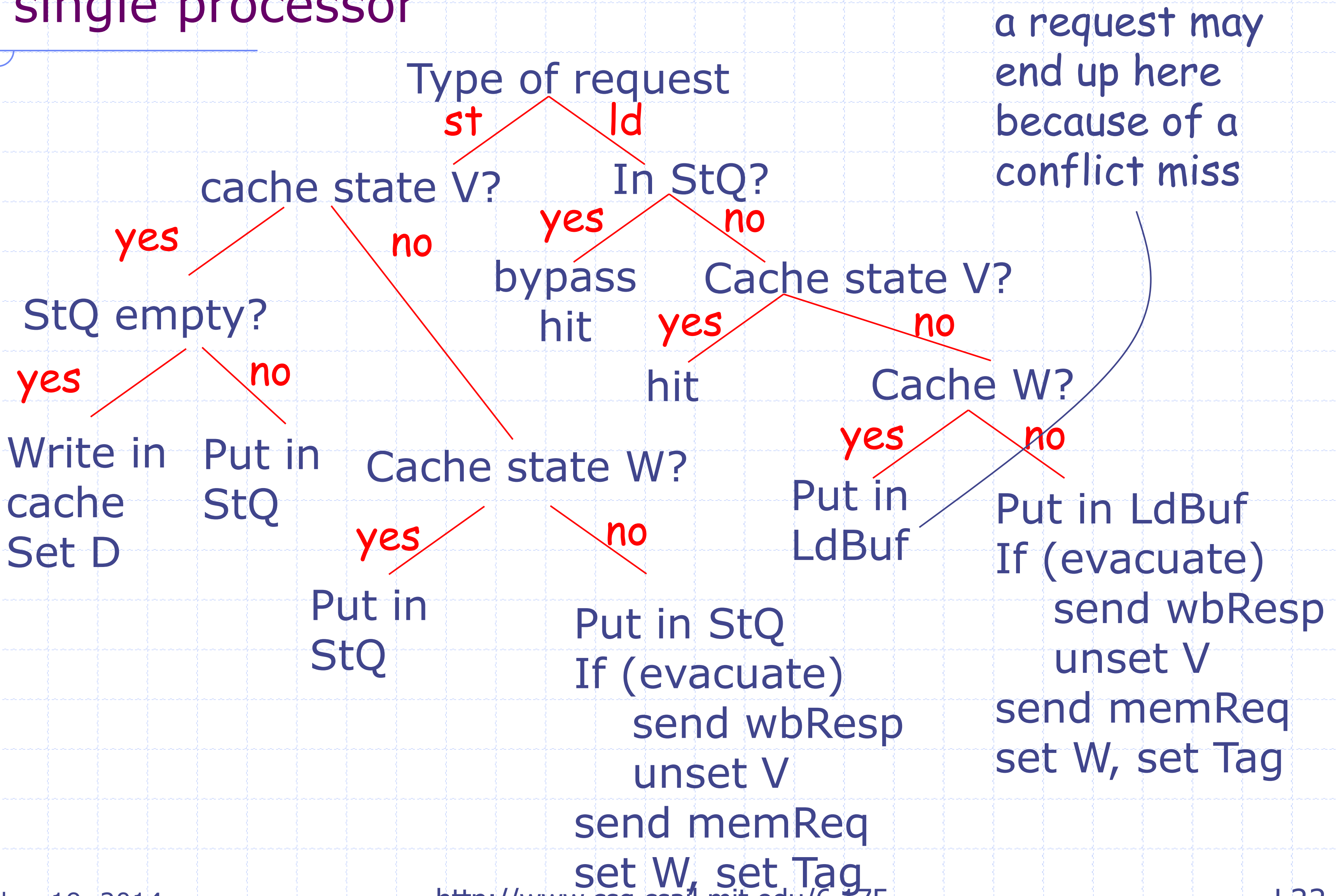
Behavior to be described by 2 concurrent FSMs to process input requests and memory responses, respectively

load reqs waiting for data

An extra bit in the cache line to indicate if it is waiting for data from parent

Incoming req

single processor



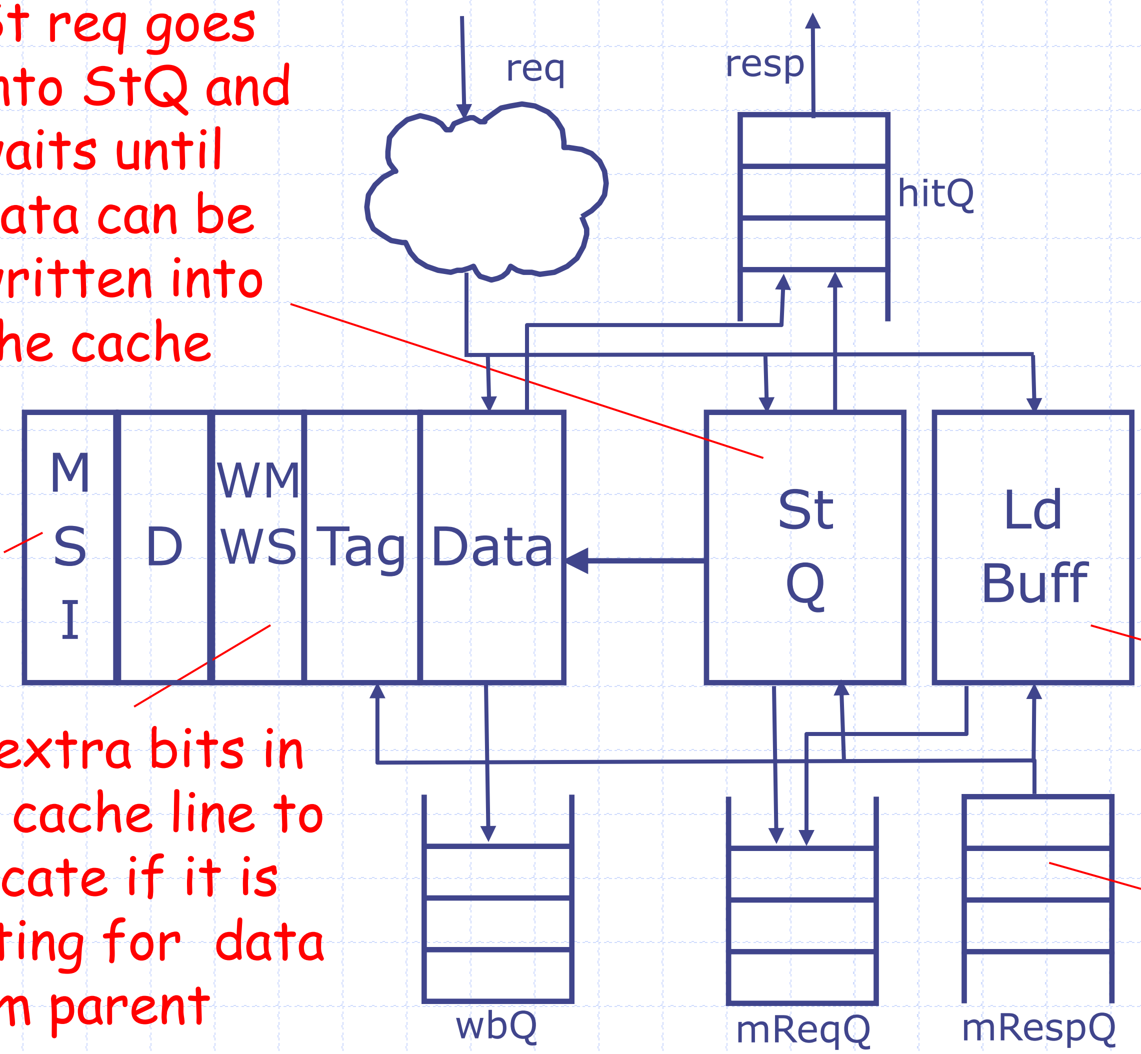
Mem Resp (for line cl)
single processor

```
1. Update cache line (set V, unset D, 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; set D
           remove the oldest entry;
           Loop back to L
        else if there is a ldBuff entry for cl // process conflict misses
               then if(evacuate) wbResp; unset V
                      memReq for the address in ldBuff;
                      set W, set Tag
        else if cachestate(oldest StQ entry address) = !W
               then if(evacuate) wbResp; unset V
                      memReq for this store entry;
                      set W, set Tag
```

Non-blocking Cache

multi-processor

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



state

An extra bits in the cache line to indicate if it is waiting for data from parent

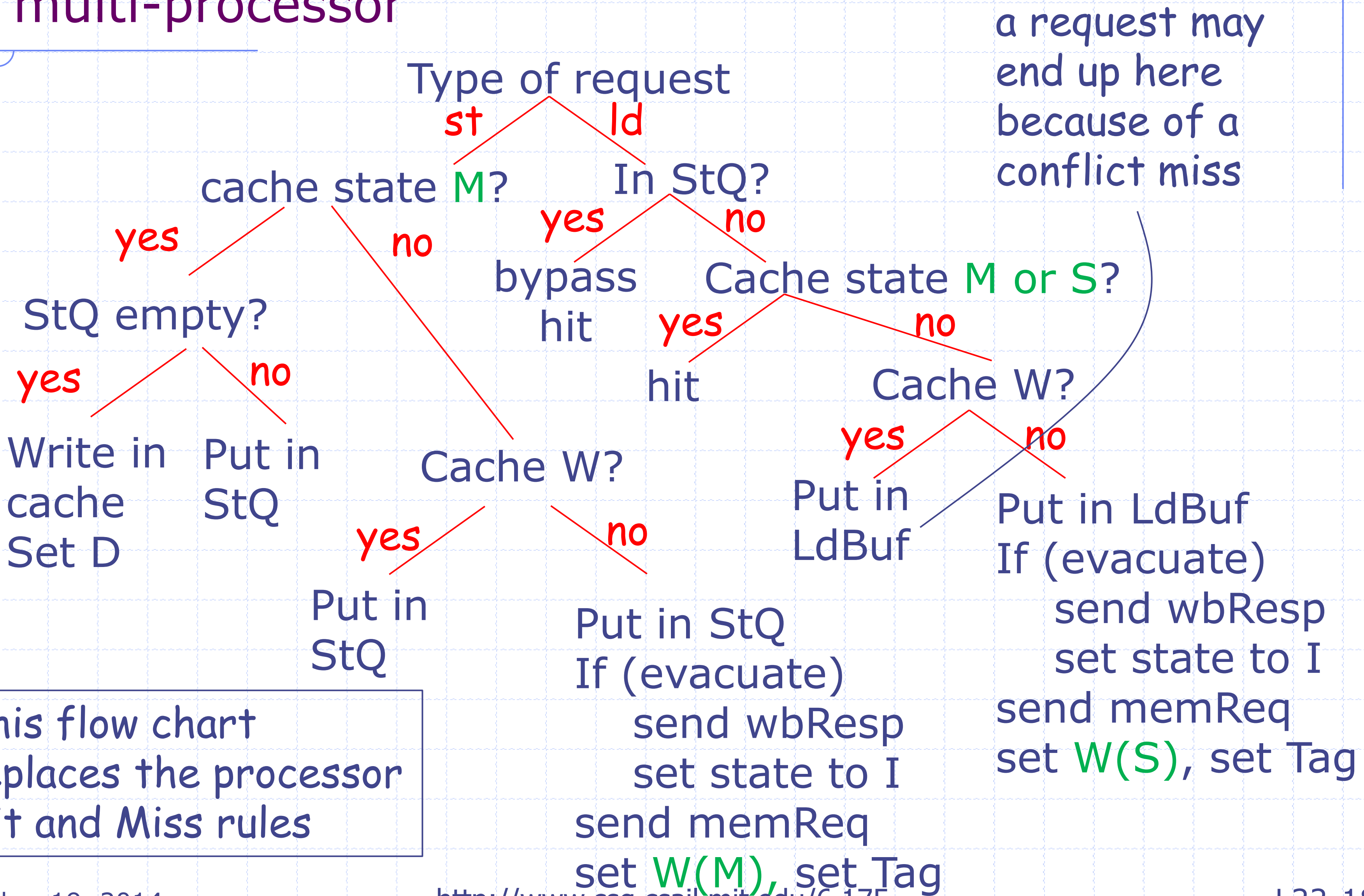
Behavior to be described by 2 concurrent FSMs to process input requests and memory responses, respectively

load reqs waiting for data

Includes invalidation messages

Incoming req

multi-processor



Type of request
st
ld
cache state M?
yes
no
StQ empty?
yes
no
Write in cache Set D
Put in StQ
Cache W?
yes
no
Put in StQ
Put in StQ If (evacuate) send wbResp set state to I send memReq set W(M), set Tag
In StQ?
yes
no
bypass hit
Cache state M or S?
yes
no
hit
Cache W?
yes
no
Put in LdBuf
Put in LdBuf If (evacuate) send wbResp set state to I send memReq set W(S), set Tag
a request may end up here because of a conflict miss

This flow chart replaces the processor Hit and Miss rules

Mem Resp (for line cl)

multi-processor

```
1. Update cache line (set state to M or S based on W,
                                          unset D,  unset W)
2. Process all matching ldBuff entries and send responses
3. L: If cachestate(oldest StQ entry address) = M
       then
          update the cache word with StQ entry; set D
          remove the oldest entry;
          Loop back to L
       else if there is a ldBuff entry for cl
             then if(evacuate) wbResp; set state to I
                   memReq for the address in ldBuff;
                   set W(S), set Tag
       else if cachestate(oldest StQ entry address) = !W
             then if(evacuate) wbResp; set state to I
                   memReq for this store entry;
                   set W(M), set Tag
```

This flow chart replaces rule 3 (for L1) *next - Network and buffer issues to avoid deadlocks*