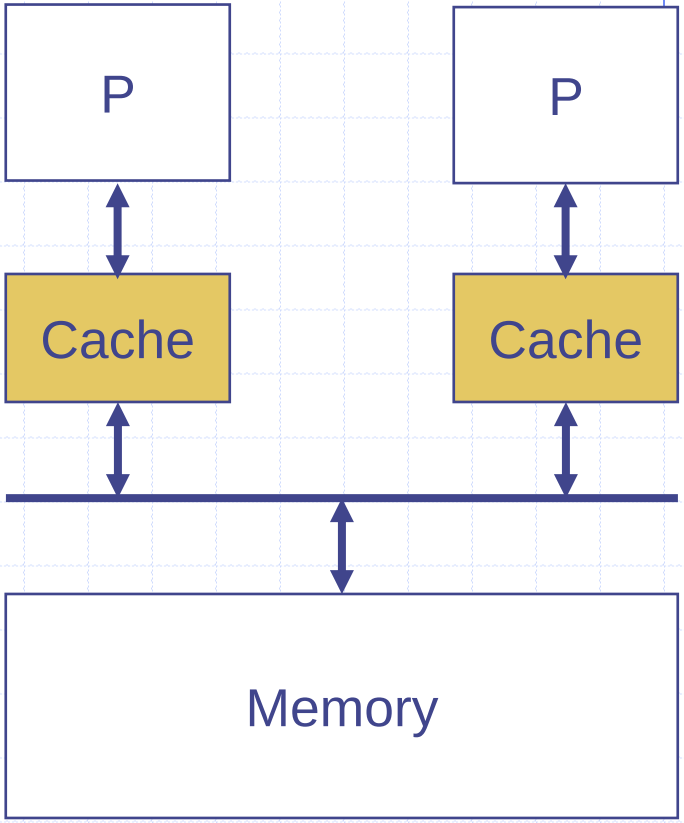
Constructive Computer Architecture

Cache Coherence

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

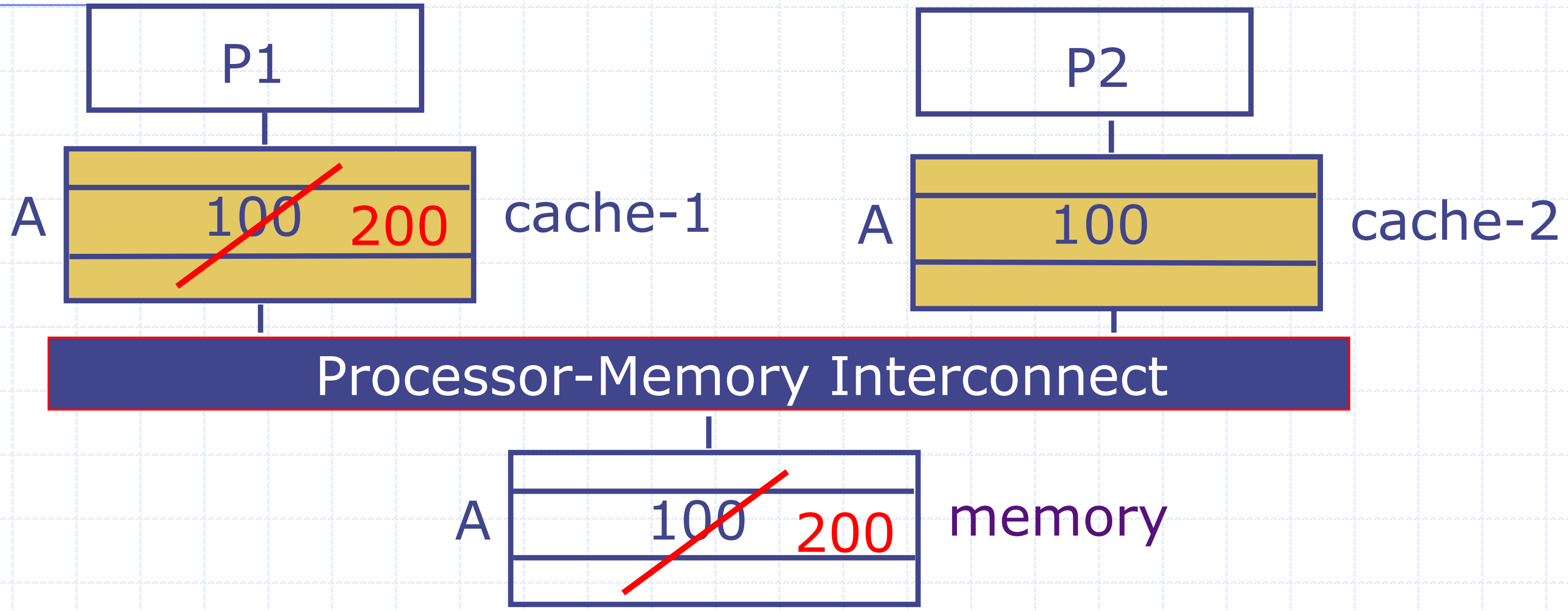
SC and caches

- Caches present a similar problem as store buffers – stores in one cache will not be visible to other caches automatically
- Cache problem is solved differently – *caches are kept coherent*



How to build coherent caches is the topic of this lecture

Cache-coherence problem

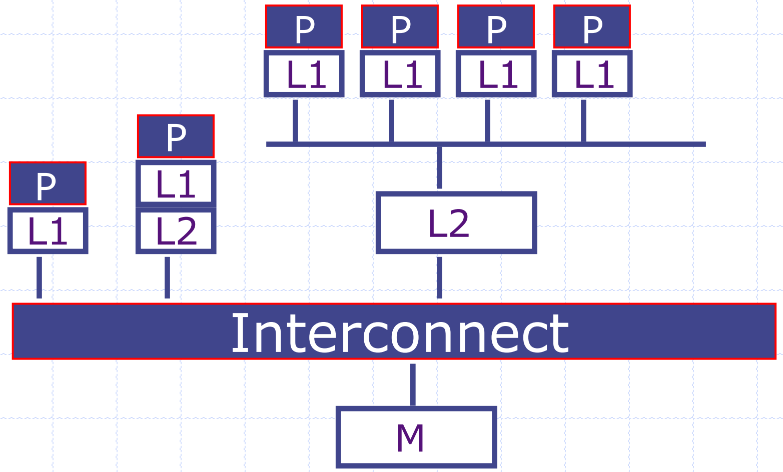


- Suppose P1 updates A to 200.
  - *write-back:* memory and P2 have stale values
  - *write-through:* P2 has a stale value

Do these stale values matter for programming?

Yes, if we want to implement SC or, in fact, any reasonable memory model

Shared Memory Systems

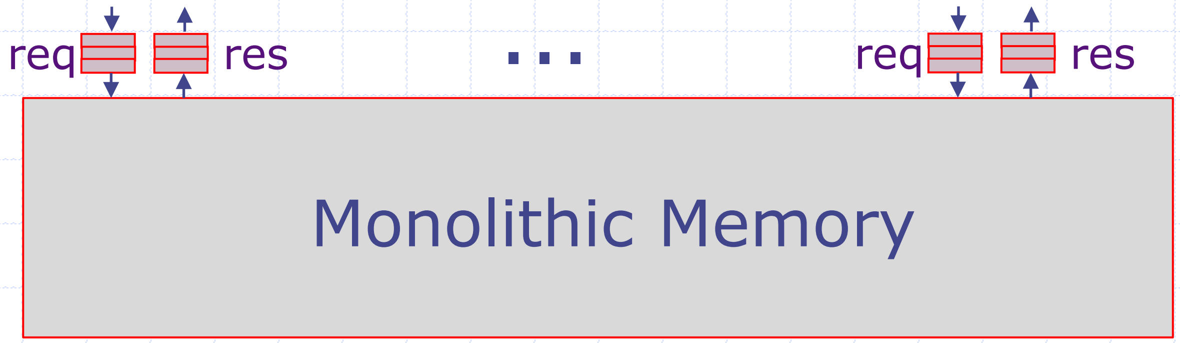


- Modern systems often have hierarchical caches
- Each cache has exactly one parent but can have zero or more children
- Logically only a parent and its children can communicate directly
- *Inclusion property* is maintained between a parent and its children, i.e.,

$$a \in L_i \Rightarrow a \in L_{i+1}$$

Because usually $L_{i+1} >> L_i$

Cache-Coherent Memory



- A monolithic memory processes one request at a time; it can be viewed as processing requests instantaneously
- A memory with hierarchy of caches is said to be *coherent*, if functionally it behaves like the monolithic memory

Maintaining Coherence

- In a *coherent memory* all loads and stores can be placed in a global order
  - multiple copies of an address in various caches can cause this property to be violated

- This property can be ensured if:
  - Only one cache at a time has the write permission for an address
  - No cache can have a stale copy of the data after a write to the address has been performed

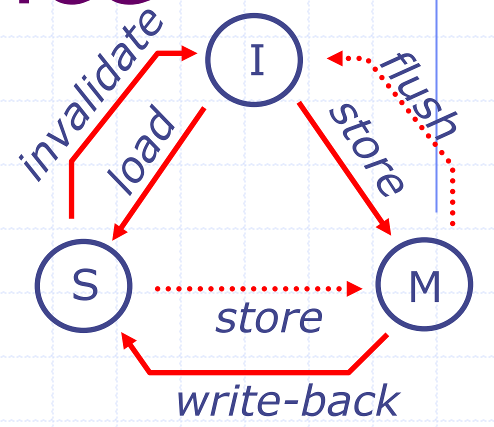
$\Rightarrow$ *cache coherence protocols are used to maintain coherence*

Cache Coherence Protocols

- Write request:
  - the address is *invalidated* in all other caches *before* the write is performed
- Read request:
  - if a dirty copy is found in some other cache then that value is written back to the memory and supplied to the reader. Alternatively the dirty value can be forwarded directly to the reader

Such protocols are called Invalidation-based

State and actions needed to maintain Cache Coherence



- Each line in each cache maintains MSI state:

  I - cache doesn't contain the address

  S- cache has the address but so may other caches; hence it can only be read

  M- only this cache has the address; hence it can be read and written
- Action on a read miss (i.e., Cache state is I):
  - If some other cache has the address in state M then write back the dirty data to Memory and set its state to S
  - Read the value from Memory and set the state to S
- Action on a write miss (i.e., Cache state is I or S):
  - *Invalidate* the address in other caches; in case some cache has the address in state M then write back the dirty data
  - Read the value from Memory if necessary and set the state to M

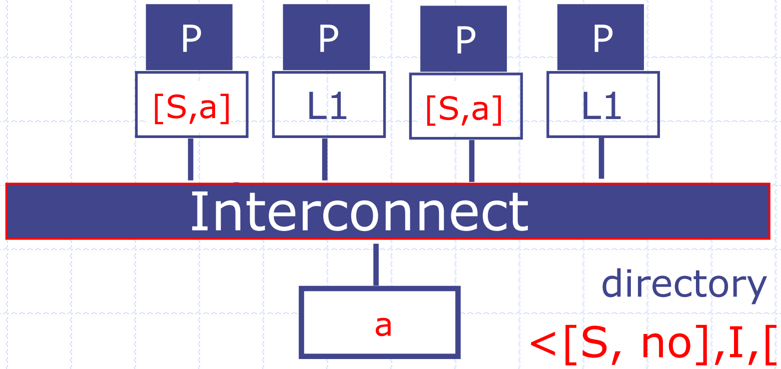
How do we know the state of other caches?

Protocols are distributed!

- Fundamental assumption
  - A processor or cache can only examine or set its own state
  - The state of other caches is inferred or set by sending request and response messages
- Each parent cache maintains information about each of its child cache in a directory
  - Directory information is *conservative*, e.g., if the directory say that the child cache c has a cache-line in state S, then cache c may have the address in either S or I state but not in M state
  - Sometimes the state of a cache line is transient because it has requested a change. Directory also contains information about outstanding messages

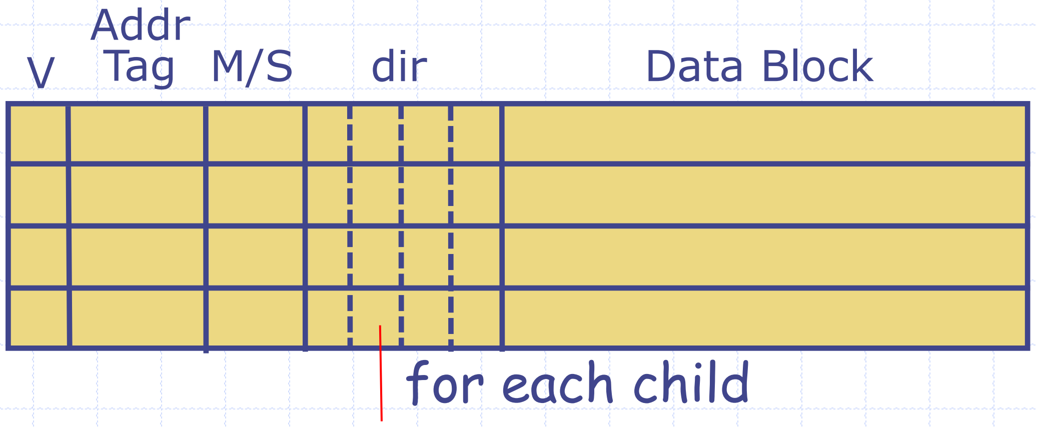
Directory State Encoding

Two-level (L1, M) system



directory for a

<[S, no],I,[S, no],I>



-L1 has no directory

-M has no need for MSI

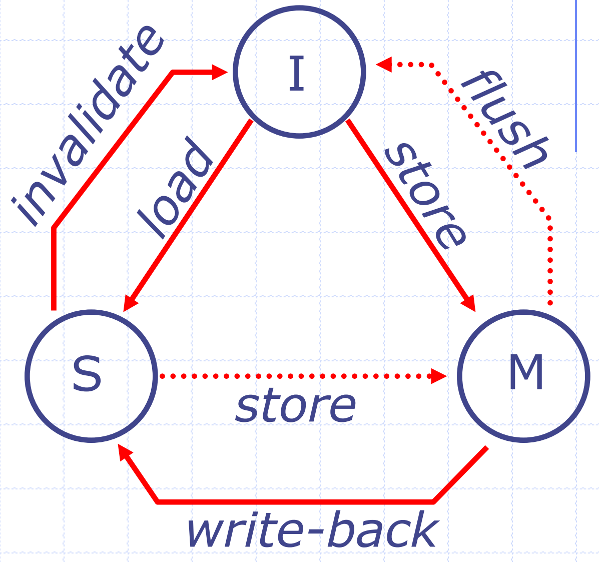
<[(M|S|I), (No | Yes)]>

Child's state

Waiting for downgrade response

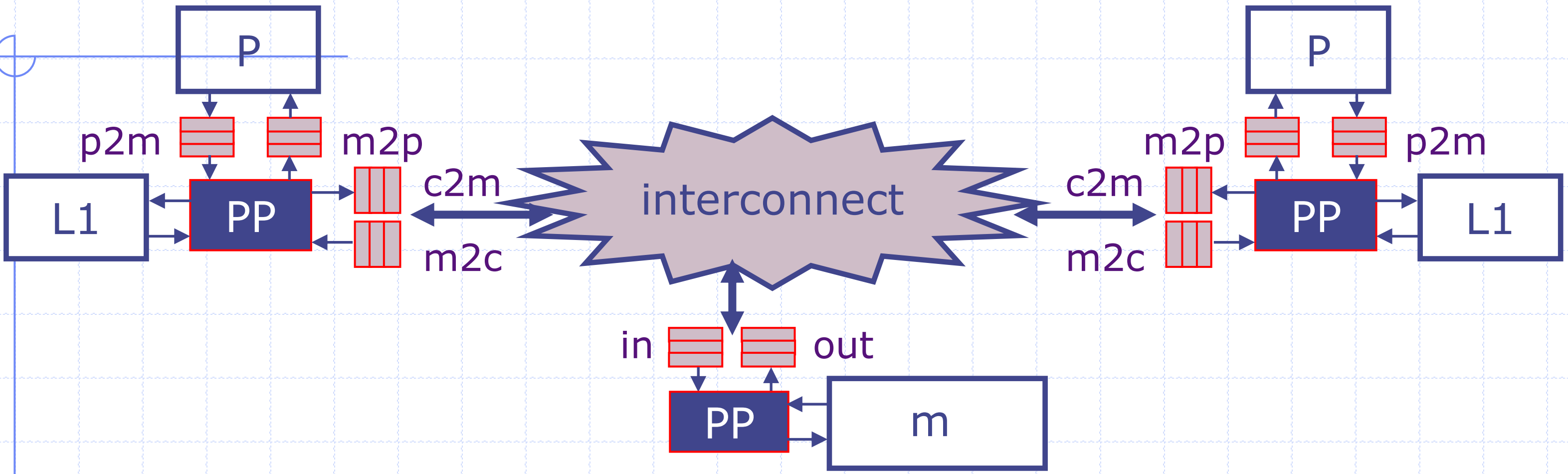
State ordering to develop protocols

- The states M, S, I can be thought of as an order M>S>I
  - *Upgrade:* A cache miss causes transition from a lower state to a higher state
  - *Downgrade:* A write-back or invalidation causes a transition from a higher state to a lower state



Message passing

an abstract view



- Each cache has 2 pairs of queues
  - (c2m, m2c) to communicate with the memory
  - (p2m, m2p) to communicate with the processor
- Message format: <cmd, src→dst, a, s, data>

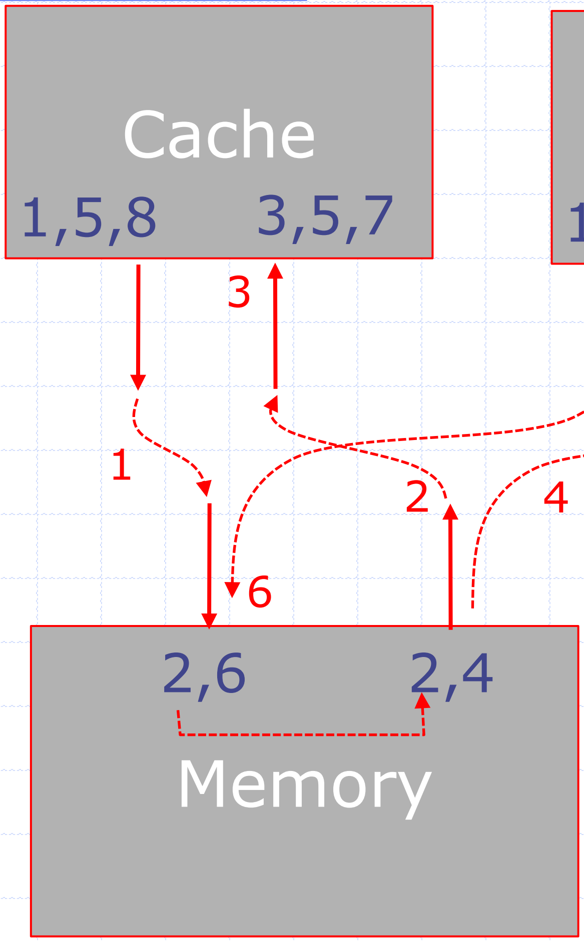
  Req/Resp address state
- FIFO message passing between each (src→dst) pair except a *Req cannot block a Resp*
- Messages in one src→dst path cannot block messages in another src→dst path

Consequences of distributed protocol

- In the blocking-cache protocol we presented in L15, a cache could go to sleep after it issued a request for a missing line
- A cache may receive an invalidation request at any time from other caches (via its parent); such requests cannot be ignored otherwise the system will deadlock
  - none of the requests may be able to complete

A difficult part of the protocol design is to determine which request can arrive in a given state

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)

CC protocol for blocking caches

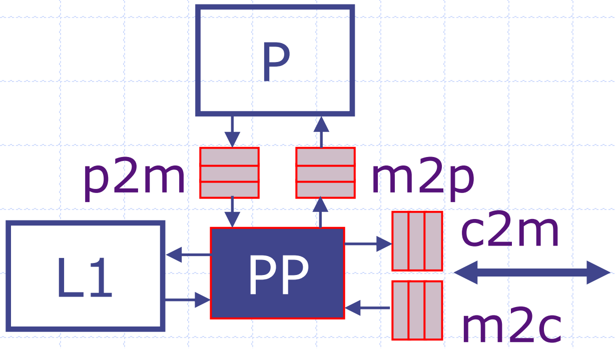
Extension to the cache rules for Blocking L1 design discussed in lecture L15

Code is somewhat simplified by assuming that cache-line size = one word

syntax is full of errors

Req method
hit processing

```
method Action req(MemReq r) if(mshr == Ready);
  let a = r.addr;
  let hit = contains(state, a);
  if(hit) begin
    let slot = getSlot(state, a);
    let x = dataArray[slot];
    if(r.op == Ld) hitQ.enq(x);
    else // it is store
         if (isStateM(state[slot])
              dataArray[slot] <= r.data;
         else begin missReq <= r; mshr <= SendFillReq;
                    missSlot <= slot; end
          end
  else begin missReq <= r; mshr <= StartMiss; end // (1)
endmethod
```



Start-miss and Send-fill rules

```
Rdy -> StrtMiss -> SndFillReq -> WaitFillResp -> Resp -> Rdy
```

```
rule startMiss(mshr == StartMiss);
  let slot = findVictimSlot(state, missReq.addr);
  if(!isStateI(state[slot]))
    begin // write-back (Evacuate)
      let a = getAddr(state[slot]);
      let d = (isStateM(state[slot])? dataArray[slot]: -);
      state[slot] <= (I, _);
      c2m.enq(<Resp, c->m, a, I, d>); end
  mshr <= SendFillReq; missSlot <= slot; endrule

 rule sendFillReq (mshr == SendFillReq);
   let upg = (missReq.op == Ld)? S : M;
   c2m.enq(<Req, c->m, missReq.addr, upg, - >);
   mshr <= WaitFillResp;  endrule  // (1)
```

Wait-fill rule and Proc Resp rule

```
Rdy -> StrtMiss -> SndFillReq -> WaitFillResp -> Resp -> Rdy
```

```
rule waitFillResp ((mshr == WaitFillResp) &&&
        (m2c.first matches <Resp, m->c, .a, .cs, .d>));
  let slot = missSlot;
  dataArray[slot] <=
       (missReq.op == Ld)? d : missReq.data;
  state[slot] <= (cs, a);
  m2c.deq;
  mshr <= Resp;
endrule // (3)

rule sendProc(mshr == Resp);
  if(missReq.op == Ld) begin
    c2p.enq(dataArray[slot]); end
  mshr <= Ready;
endrule
```

Parent Responds

```
rule parentResp
           (c2m.first matches <Req,.c->m,.a,.y,.*>);
   let slot = getSlot(state, a); // in a 2-level
       // system a has to be present in the memory
   let statea = state[slot];
   if((∀i≠c, isCompatible(statea.dir[i],y))
      && (statea.waitc[c]=No)) begin
     let d =(statea.dir[c]=I)? dataArray[slot]: -);
     m2c.enq(<Resp, m->c, a, y, d>);
     state[slot].dir[c]:=y;
     c2m.deq;
   end
endrule
```

IsCompatible(M, M) = False
IsCompatible(M, S) = False
IsCompatible(S, M) = False
All other cases = True

Parent (Downgrade) Requests

```
rule dwn (c2m.first matches <Req,c->m,.a,.y,.*>);
   let slot = getSlot(state, a);
   let statea = state[slot];
   if (findChild2Dwn(statea) matches (Valid .i))
   begin
      state[slot].waitc[i] <= Yes;
      m2c.enq(<Req, m->i, a, (y==M?I:S), ? >);
   end;
Endrule // (4)
```

This rule will execute as long some child cache is not compatible with the incoming request

Parent receives Response

```
rule dwnRsp (c2m.first matches <Resp, c->m, .a, .y,
                                              .data>);
  c2m.deq;
  let slot = getSlot(state, a);
  let statea = state[slot];
  if(statea.dir[c]=M) dataArray[slot]<=data;
  state[slot].dir[c]<=y;
  state[slot].waitc[c]<=No;
endrule // (6)
```

Child Responds

```
rule dng ((mshr != Resp)  &&&
                  m2c.first matches <Req,m->c,.a,.y,.*>);
   let slot = getSlot(state,a);
   if(getCacheState(state[slot])>y) begin
     let d = (isStateM(state[slot])? dataArray[slot]: -);
     c2m.enq(<Resp, c->m, a, y, d>);
     state[slot] <= (y,a);
   end
   // the address has already been downgraded
   m2c.deq;
endrule // (5) and (7)
```

Child Voluntarily downgrades

```
rule startMiss(mshr == Ready);
  let slot = findVictimSlot(state);
  if(!isStateI(state[slot]))
    begin // write-back (Evacuate)
      let a = getAddr(state[slot]);
      let d = (isStateM(state[slot])? dataArray[slot]: -);
      state[slot] <= (I, _);
      c2m.enq(<Resp, c->m, a, I, d>);
   end
endrule // (8)
```

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

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