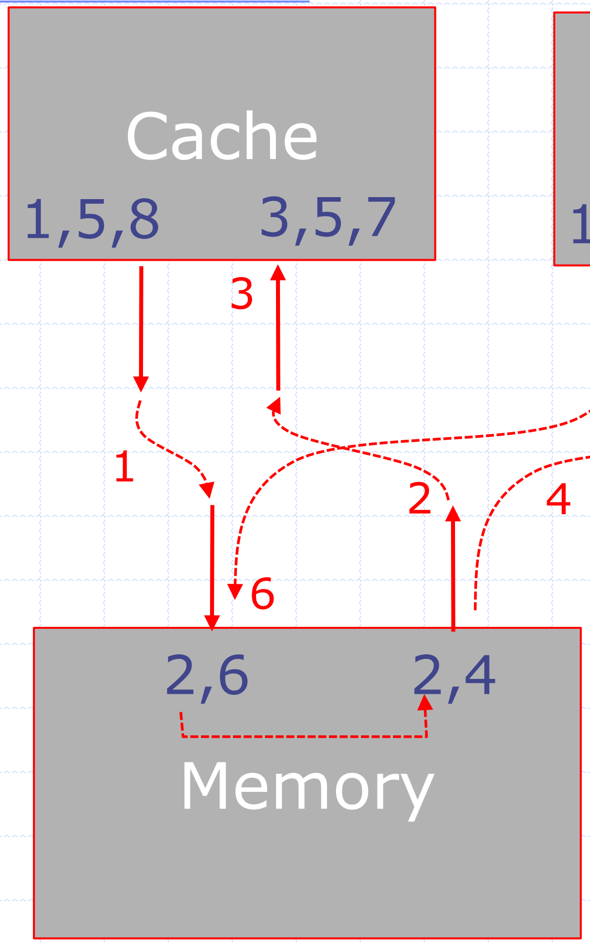
CC protocol for blocking caches

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

Some pseudo syntax is used

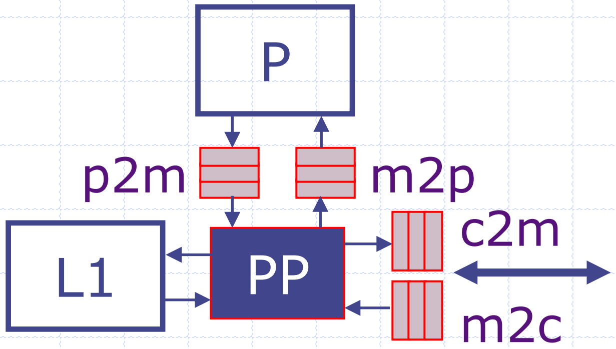
Processing misses: Requests and Responses



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)

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);
      // must have a reserved slot to receive reply
  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.firstResp matches <Resp, m->c, .a, .cs, .d>));
  let slot = missSlot;
  dataArray[slot] <=
       (missReq.op == Ld)? d : missReq.data;
  state[slot] <= (cs, a);
  m2c.deqResp;
  mshr <= Resp;
endrule // (3)
rule sendProc(mshr == Resp);
  if(missReq.op == Ld) begin let slot = missSlot;
    c2p.enq(dataArray[slot]); end
  mshr <= Ready;
endrule
```

Parent Responds

```
rule parentResp
        (c2m.firstReq 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.firstReq 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.firstResp matches <Resp, c->m, .a, .y, .data>);
  c2m.deqResp;
  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

Incoming downgrade requests in L1

```
rule dng ((mshr != Resp) &&&
                 (m2c.firstReq 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.deqReq;
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

MSI protocol: some issues

- It never makes sense to have two outstanding requests for the same address from the same processor/cache
- It is possible to have multiple requests for the same address from different processors. Hence there is a need to arbitrate requests
- A cache needs to be able to evict an address in order to make room for a different address
  - Voluntary downgrade
- Memory system (higher-level cache) should be able to force a lower-level cache to downgrade
  - caches need to keep track of the state of their children’s caches

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

Nonblocking Synchronization

Load-reserve & Store-conditional

Special register(s) to hold reservation flag and address, and the outcome of store-conditional

```
Load-reserve R, m:
    <flag, adr> ← <1, m>;
    R ← M[m];
```

```
Store-conditional m, R:
    if <flag, adr> == <1, m>
    then  cancel other procs'
          reservation on m;
          M[m] ← R;
          status ← succeed;
    else  status ← fail;
```

```
try:    Load-reserve $R_{head}$, head
spin:   Load $R_{tail}$, tail
        if $R_{head}$==$R_{tail}$ goto spin
        Load R, ($R_{head}$)
        $R_{head}$ = $R_{head}$ + 1
        Store-conditional head, $R_{head}$
        if (status==fail) goto try
        process(R)
```

The corresponding instructions in RISC V are called lr and sc, respectively

Nonblocking Synchronization

```
Load-reserve R, (m):
    <flag, adr> ← <1, m>;
    R ← M[m];
```

```
Store-conditional (m), R:
    if <flag, adr> == <1, m>
    then  cancel other procs'
          reservation on m;
          M[m] ← R;
          status ← succeed;
    else  status ← fail;
```

- The flag is cleared in other processors on a Store using the CC protocol's invalidation mechanism
- Usually address m is not remembered by Load-reserve; the flag is cleared on *any* invalidation
  - works as long as the Load-reserve instructions are not used in a nested manner
- These instructions won't work properly if Loads and Stores can be reordered dynamically