Directory-Based Cache Coherence Protocols

Joel Emer
Computer Science and Artificial Intelligence Lab
M.I.T.

http://www.csg.csail.mit.edu/6.823

Maintaining Cache Coherence

It is sufficient to have hardware such that

- only one processor at a time has write permission for a location
- no processor can load a stale copy of the location after a write

$\Rightarrow$ A correct approach could be:

write request:

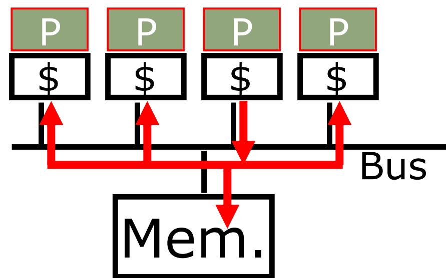
The address is *invalidated* in all other caches *before* the write is performed

read request:

If a dirty copy is found in some cache, a write-back is performed before the memory is read

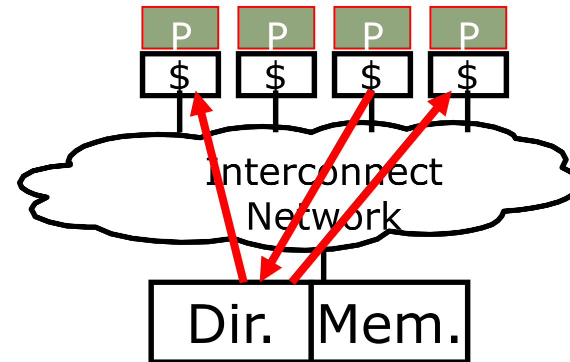
Directory-Based Coherence
(Censier and Feautrier, 1978)

Snoopy Protocols



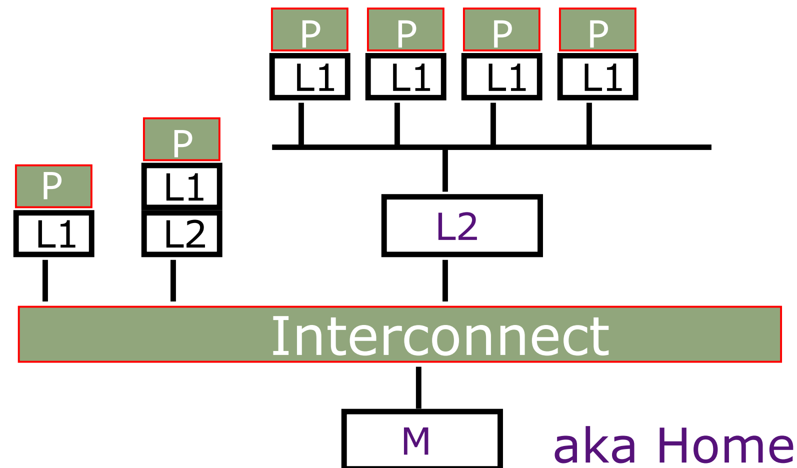
- Snoopy schemes broadcast requests over memory bus
- Difficult to scale to large numbers of processors
- Requires additional bandwidth to cache tags for snoop requests

Directory Protocols



- Directory schemes send messages to only those caches that might have the line
- Can scale to large numbers of processors
- Requires extra directory storage to track possible sharers

A System with Multiple Caches



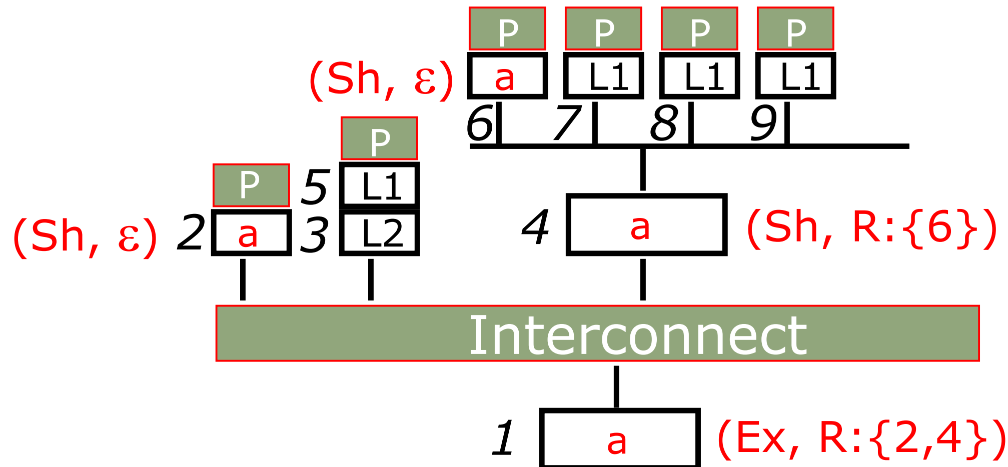
aka Home

Assumptions: Caches are organized in a hierarchical manner

- Each cache has exactly one parent but can have zero or more children
- Only a parent and its children can communicate directly
- *Inclusion property* is maintained between a parent and its children, i.e.,

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

Directory State Encoding



Each address in a cache keeps two types of state info

- *sibling info:* do my siblings have a copy of address *a*
  - Ex (means no), Sh (means maybe)
- *children info:* has address *a* been passed on to any of my children
  - W:{id} means child id has a writable version
  - R:dir means only children named in the *directory* dir have copies

Cache State Invariants

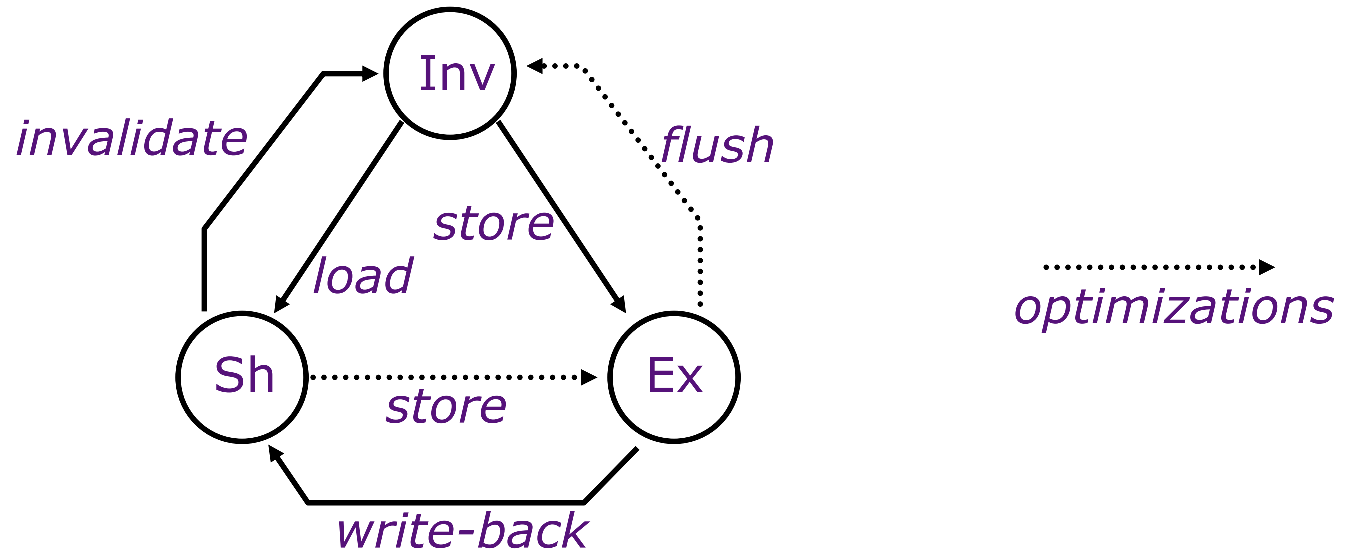
Sh: Cache's siblings and descendents can only have Sh copies

Ex: Each ancestor of the cache must be in Ex

⇒ *either* all children can have Sh copies
or one child can have an Ex copy

- Once a parent gives an Ex copy to a child, the parent's data is considered stale
- A processor cannot overwrite L1 data in Sh state
- By definition all addresses in the home memory are in the Ex state

Cache State Transitions

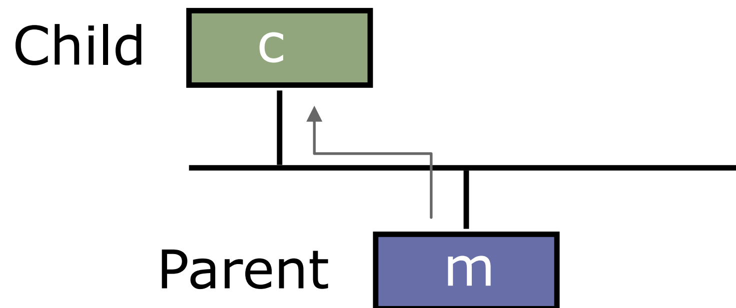


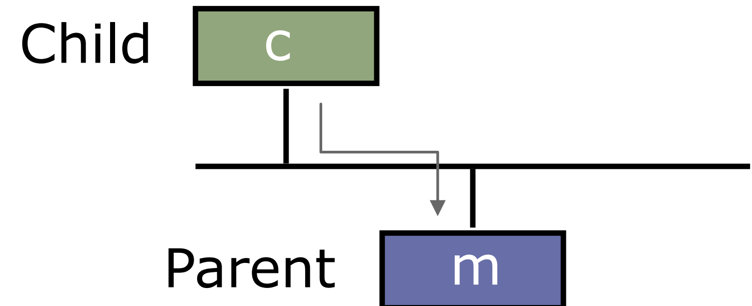
This state diagram is helpful as long as one remembers that each transition involves cooperation of other caches and the main memory.

Guarded Atomic Actions

- Rules specified using guarded atomic actions:
  <guard predicate>
  $\rightarrow$ {set of state updates that must occur
  atomically with respect to other rules}
- Example
  m.state(a) is R:dir & $\exists$id s.t. id$\notin$dir
  $\rightarrow$ m.setState(a, R:(dir+{id}));
  c_{id}.setState(a, Sh); c_{id}.setData(a, m.data(a));

Data Propagation Between Caches





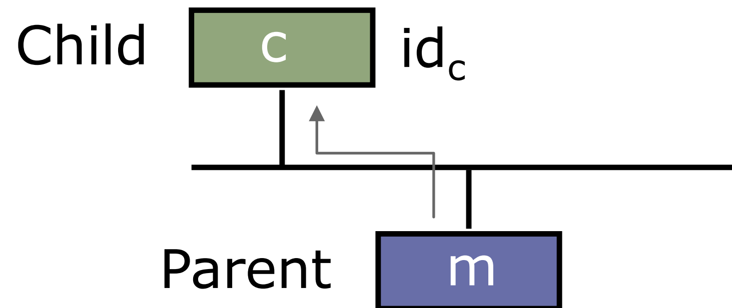
Caching rules

- *Read caching rule*
- *Write caching rule*

De-caching rules

- *Write-back rule*
- *Invalidate rule*

Caching Rules: *Parent to Child*



- Read caching rule

  m.state(a) is R:dir & id∉dir

  → m.setState(a, R:(dir+{id}))

  c_{id}.setState(a, Sh); c_{id}.setData(a, m.data(a));

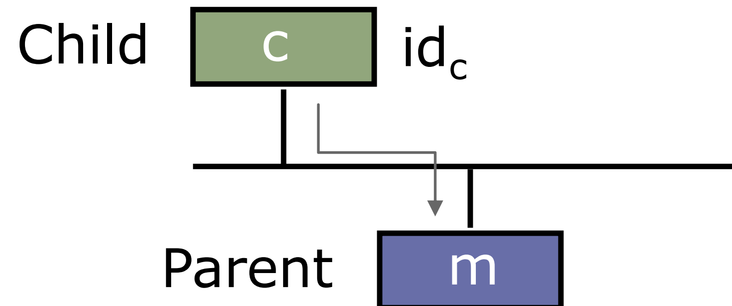
- Write caching rule

  m.state(a) is R:{} (no cache has it)

  → m.setState(a, W:{id})

  c_{id}.setState(a, Ex); c_{id}.setData(a, m.data(a));

De-caching Rules: *Child to Parent*



- Writeback rule

  m.state(a) is W:{id} & c.state(a) is Ex

  → m.setState(a, R:{id})

  m.setData(a, c.data(a));

  c_{id}.setState(a, Sh);

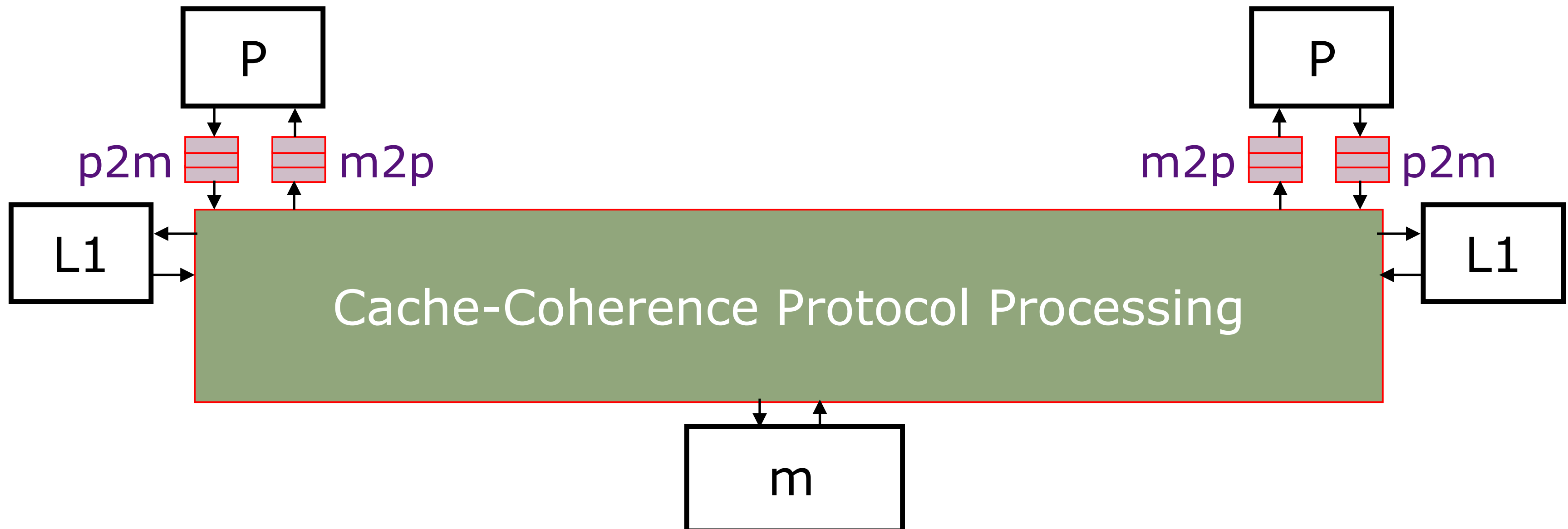
- Invalidate rule

  m.state(a) is R:dir & id∈dir & c.state(a) is Sh

  → m.setState(a, R:(dir-{id}))

  c_{id}.invalidate(a);

A simple CC Protocol: 6823s



- Assume only one processor can talk to the memory about an address at a time
  - a global lock (*serving*) per address. It remembers the processor and the type of request being serviced (id, (ShReq | ExReq))
- We can simultaneously examine the current cache state of one processor and the home directory, and atomically
  - begin a memory activity, and
  - set the directory state, and
  - set the cache state

6823s: State and Functions

Cache states: Sh, Ex, Pending, Nothing
Memory states: R:dir, W:{id}
Memory locks: False, (id, req-type)

Operations on cache:

- c.state(a) – returns state s
- c.data(a) - returns data v
- c.setState(a,s); c.setData(a,v); c.invalidate(a)

Operations on memory:

- m.data(a) - returns data v
- m.setData(a,v);

Operations on Directory:

- m.serving(a) – returns either (id, req-type) or False
- m.setServing(a, id, (ShReq | ExReq)) or m.setServing(False)
- m.state(a) - returns state either R:dir or W:{id}
- m.setState(a,s);

6823s CC protocol

Load Rules (at cache)

inst = p2m.first()

- Load-hit rule

inst is (Load a)
& c_{id}.state(a) is Sh or Ex
→ p2m.deq;
m2p.enq(c_{id}.data(a))

- Load-miss rule

inst is (Load a)
& c_{id}.state(a) is Nothing
& m.serving(a) is False
→ c_{id}.setState(a, Pen); m.setServing(a, id, ShReq)

6823s CC protocol

Store Rules (at cache)

- Store-hit rule

```
      inst is (Store a v)
   &  c_id.state(a) is Ex
→     p2m.deq;
      m2p.enq(Ack);
      c_id.setData(a, v)
```

- Store-miss rules

```
      inst is (Store a v)
   &  c_id.state(a)  is Nothing or Sh
   &  m.serving(a) is False
→     c_id.setState(a, Pen);
      m.setServing(a, id, ExReq)
```

6823s CC protocol

Voluntary Rules (at cache)

- Purge rule

"no space in cache"
& c_{id}.state(a) is Sh & m.state(a) must be R:dir
→ c_{id}.invalidate(a); m.setState(a, R:(dir-{id}));

- Writeback rule

c_{id}.state(a) is Ex & m.state(a) must be W:{id}
→ c.setState(a,Sh); m.setState(a, R:{id});
m.setData(a, c_{id}.data(a))

6823s CC protocol

Memory-side ShReq rules

- Serving Loads – Only Sh copies are out

  m.serving(a) is (id, ShReq) & m.state(a) is R:dir

  $\rightarrow$ m.setState(a, R:(dir+{id})); m.setServing(a, False)

  c_{id}.setState(a, Sh); c_{id}.setData(a, m.data(a));

- Serving Loads – An Ex copy is out

  m.serving(a) is (id, ShReq) & m.state(a) is W:{id'}

  $\rightarrow$ m.setState(a, R:{id'}); m.setData(a, $c_{id'}$.data(a));

  $c_{id'}$.setState(a, Sh);

6823s CC protocol

Memory-side ExReq rules

- Serving Stores – No copies out

  m.serving(a) is (id, ExReq) & m.state(a) is R:{}

  → m.setState(a, W:{id}); m.setServing(a, False)

  c_{id}.setState(a, Ex); c_{id}.setData(a, m.data(a));
- Serving Stores – Only the requesting cache has a copy

  m.serving(a) is (id, ExReq) & m.state(a) is R:{id}

  → m.setState(a, W:{id}); m.setServing(a, False)

  c_{id}.setState(a, Ex); c_{id}.setData(a, m.data(a));
- Serving Stores – Sh copies are out

  m.serving(a) is (id, ExReq) & m.state(a) is R:dir
  & ∃ id′ s.t. ((id′∈dir) & (id′≠ id))

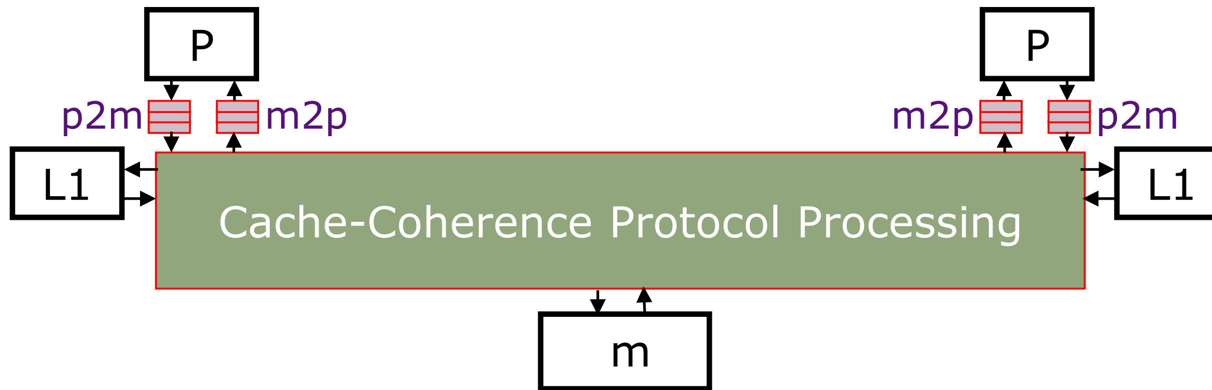
  → m.setState(a, R:(dir-{id′})); $c_{id'}$.invalidate(a);
- Serving Stores – A Ex copy is out

  m.serving(a) is (id, ExReq) & m.state(a) is W:{id′}

  → m.setState(a, R:{}); m.setData(a, $c_{id'}$.data(a));

  $c_{id'}$.invalidate(a);

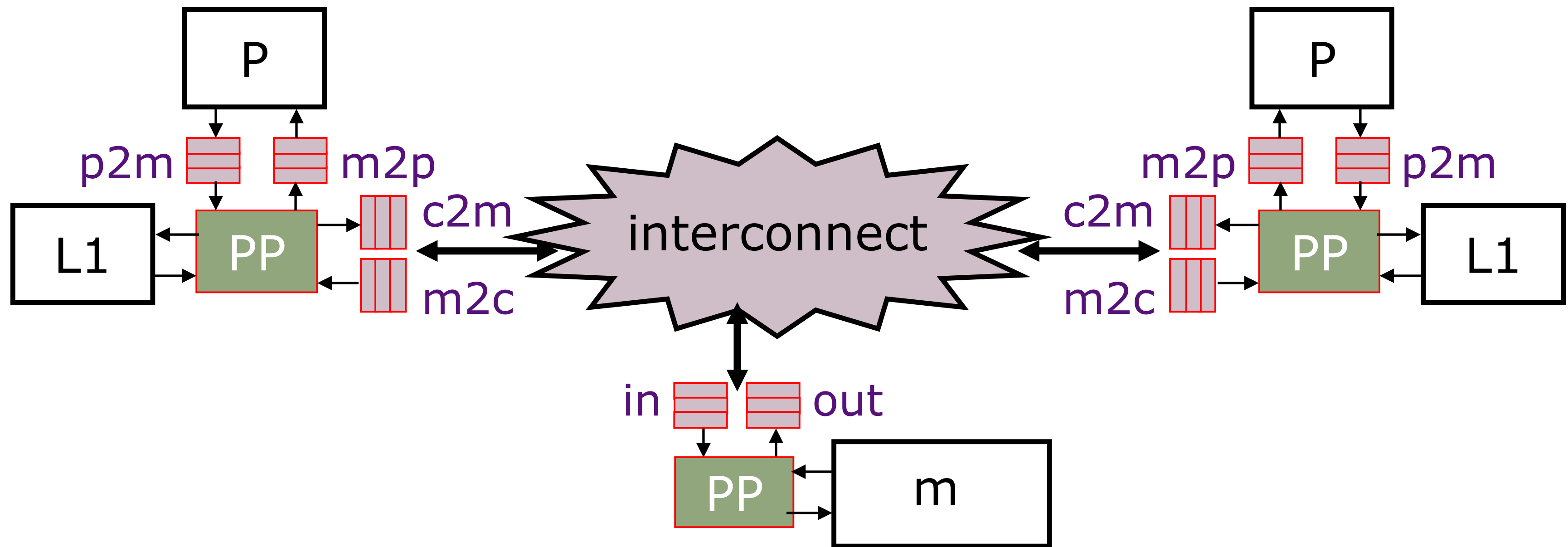
Making 6823s more realistic



- Rules require observing and changing the state of cache and memory simultaneously (atomically).
  - very difficult to implement, especially if caches are separated by a network

Split rules into multiple rules – "request for an action" followed by "an action and an ack".
- ultimately all actions are triggered by some processor

The 6823 CC Protocol *an abstract view*

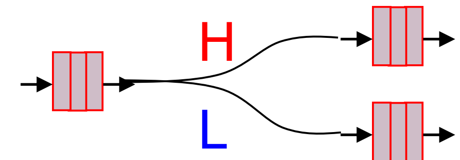


- Each cache has 2 pairs of queues
  - one pair (c2m, m2c) to communicate with the memory
  - one pair (p2m, m2p) to communicate with the processor
- Message format:
  Msg(idsrc,iddest,cmd-priority,a,v)
- FIFO message passing between each (src,dest) pair except a Low priority (L) msg cannot block a high priority (H) msg

H and L Priority Messages

- At the memory, unprocessed request messages cannot block reply messages. Hence all messages are classified as H or L priority.
  - all messages carrying replies are classified as high priority

- Accomplished by having separate paths for H and L priority
  - In Theory: separate networks
  - In Practice:
    - Separate Queues
    - Shared physical wires for both networks



6823: States and Functions

Cache states: Sh, Ex, Pending, Nothing

Memory states: R:dir, W:{id}, T_R:dir, T_W:{id}

If dir is empty then R:dir and T_R:dir represent the same state

Messages:

Cache to Memory requests: (ShReq a); (ExReq a)

Memory to Cache requests: (WbReq a); (InvReq a); (FlushReq a)

Cache to Memory replies: (WbRep a v); (InvRep a); (FlushRep a v)

Memory to Cache replies: (ShRep a v); (ExRep a v)

Operations on cache:

cache.state(a) – returns state s

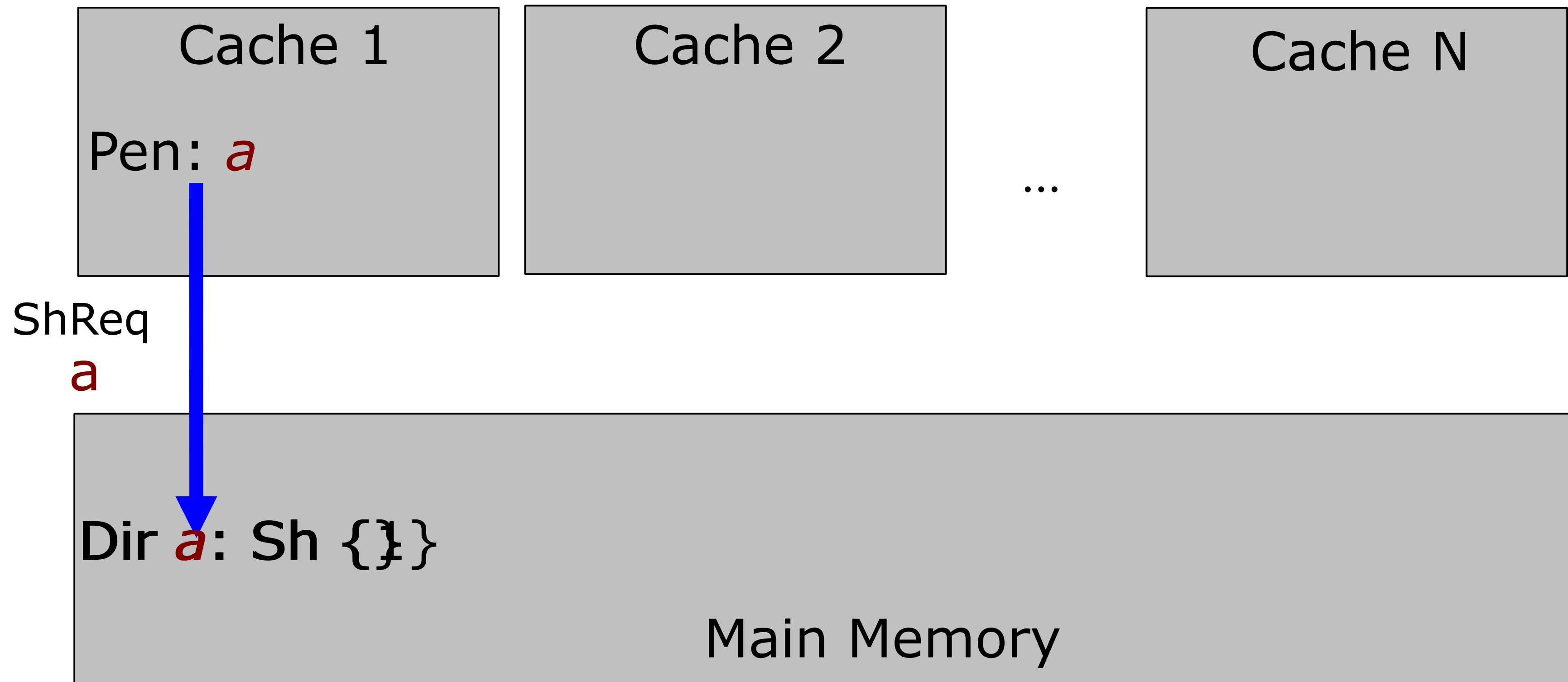
cache.data(a) - returns data v

cache.setState(a,s); cache.setData(a,v); cache.invalidate(a)

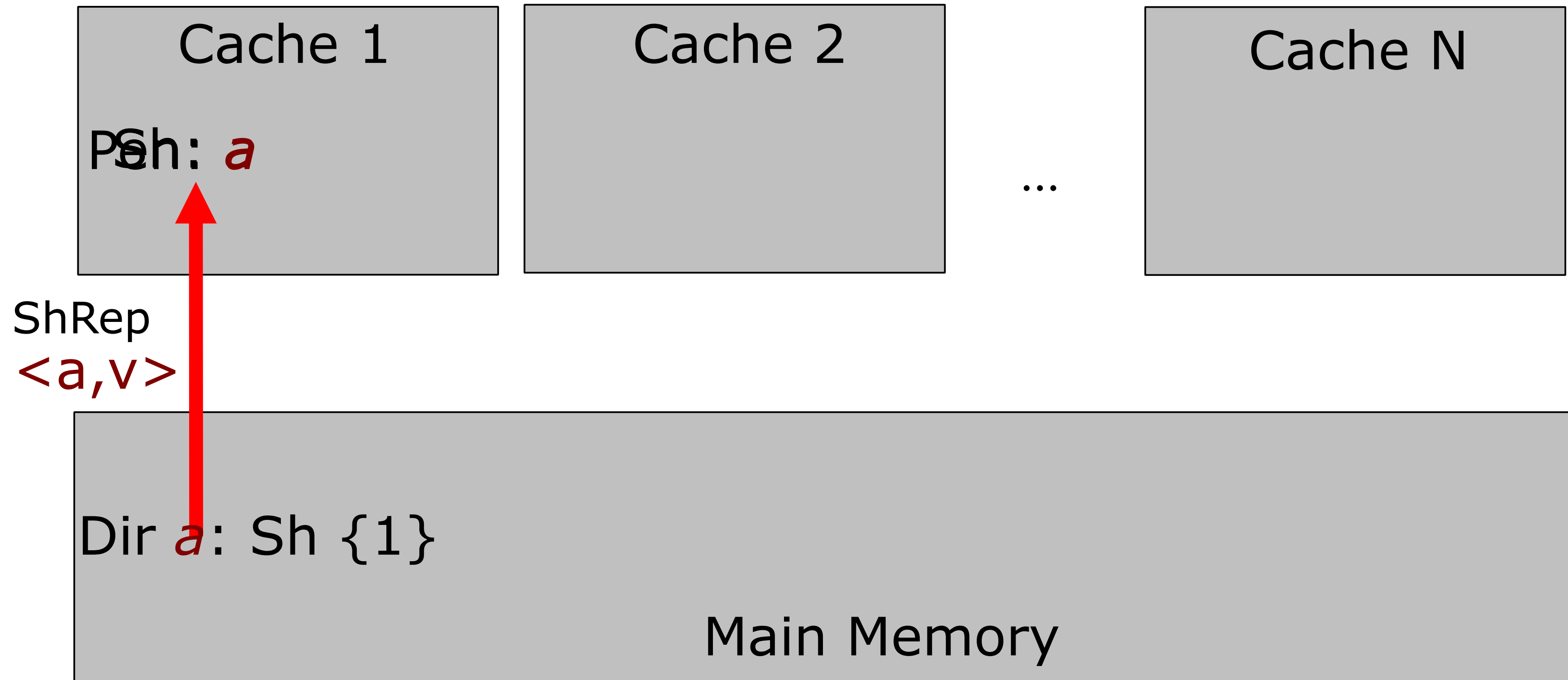
inst = first(p2m); msg= first(m2c); mmsg = first(in)

6823 Protocol Animation

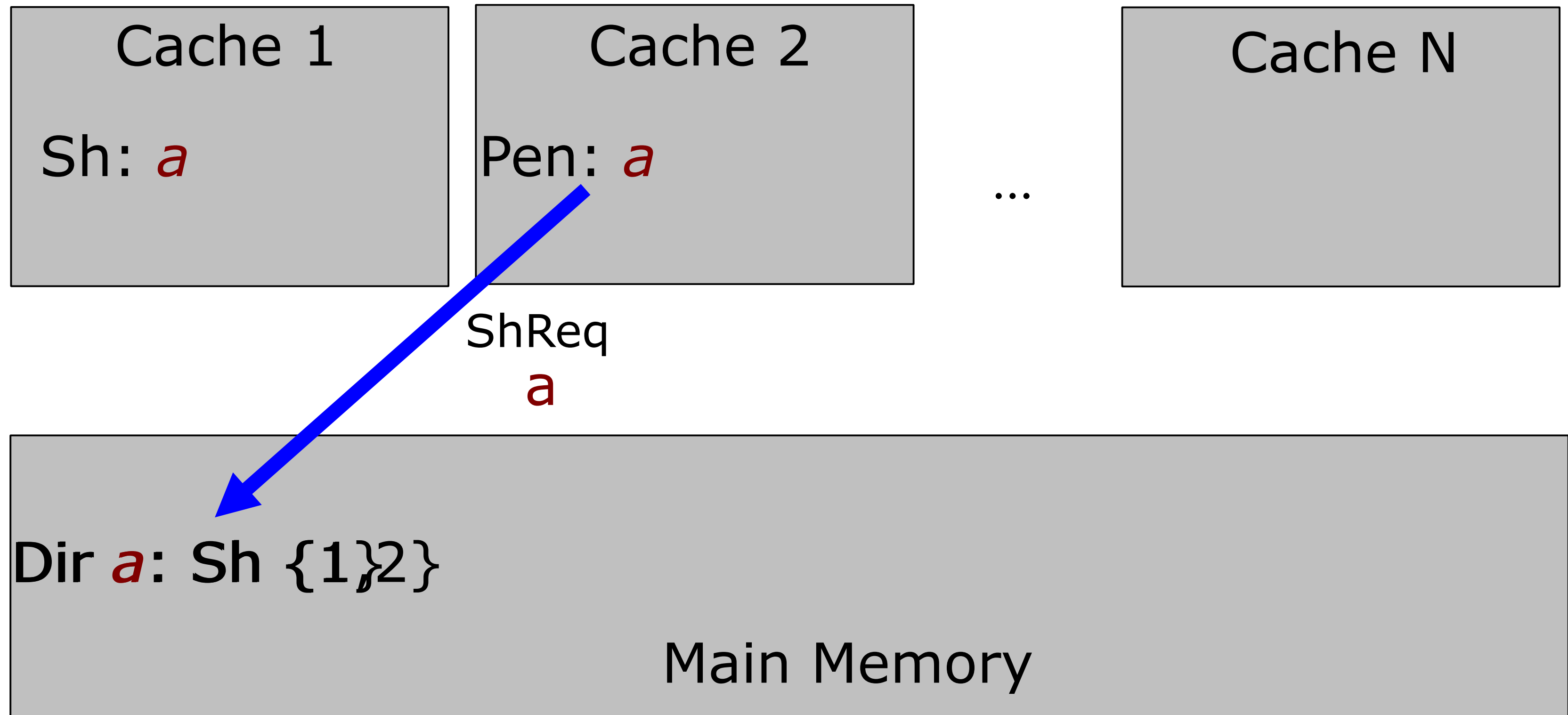
Protocol Diagram



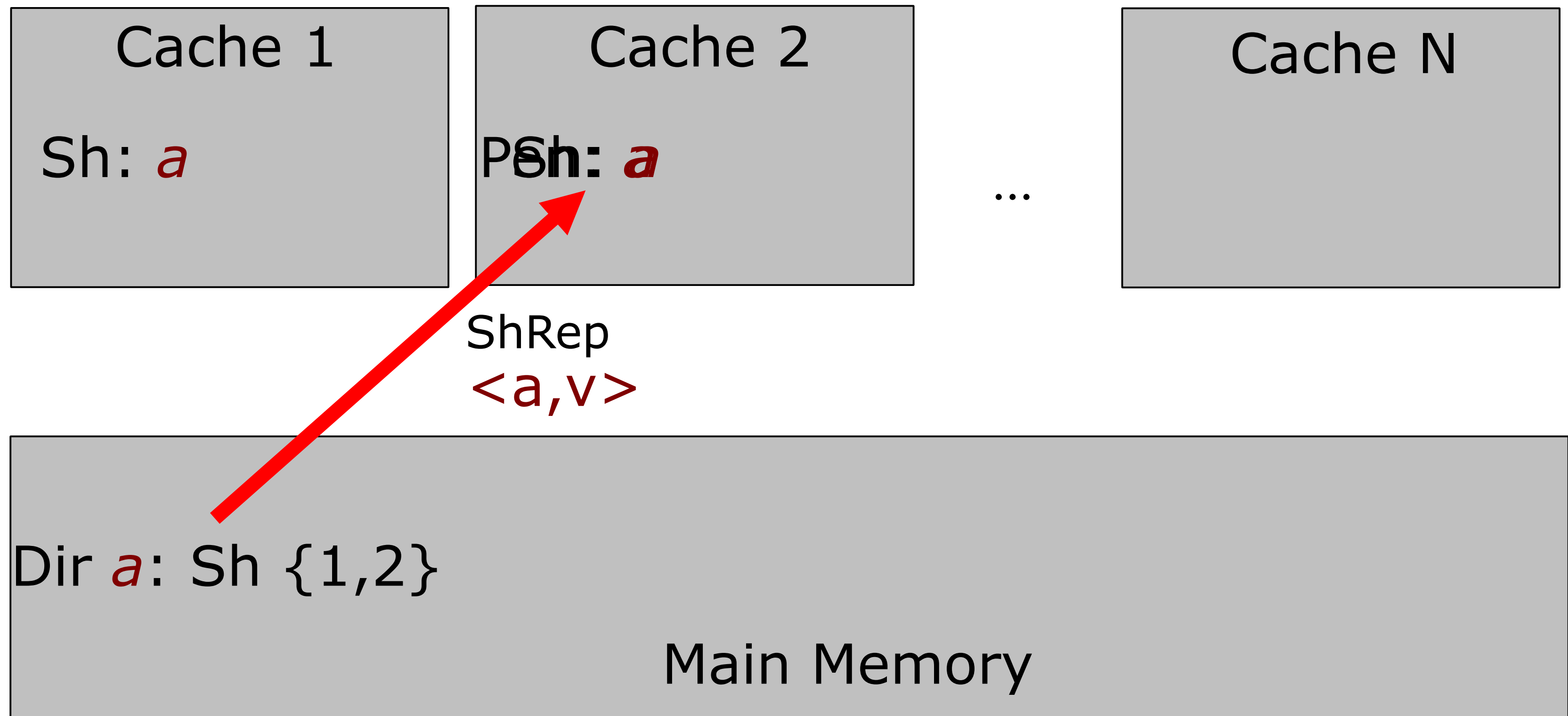
Protocol Diagram



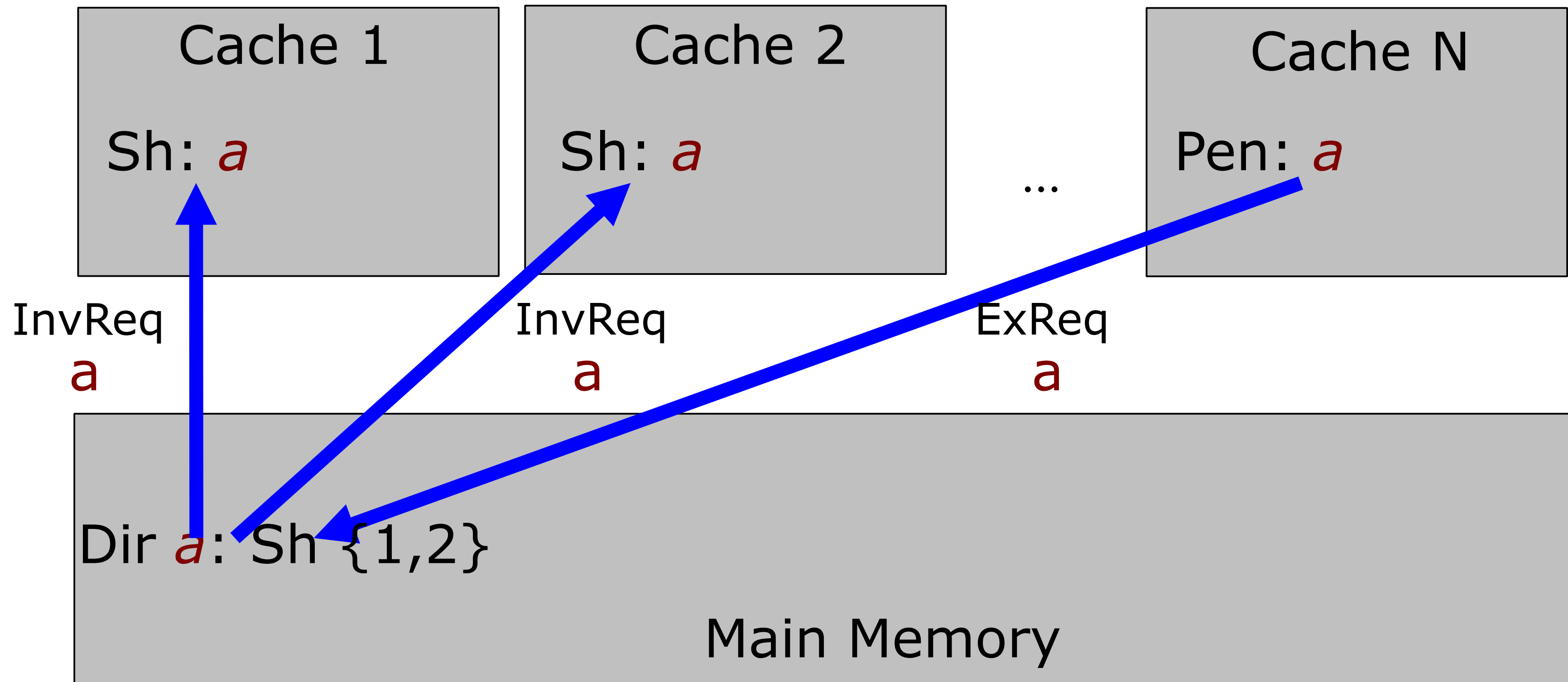
Protocol Diagram



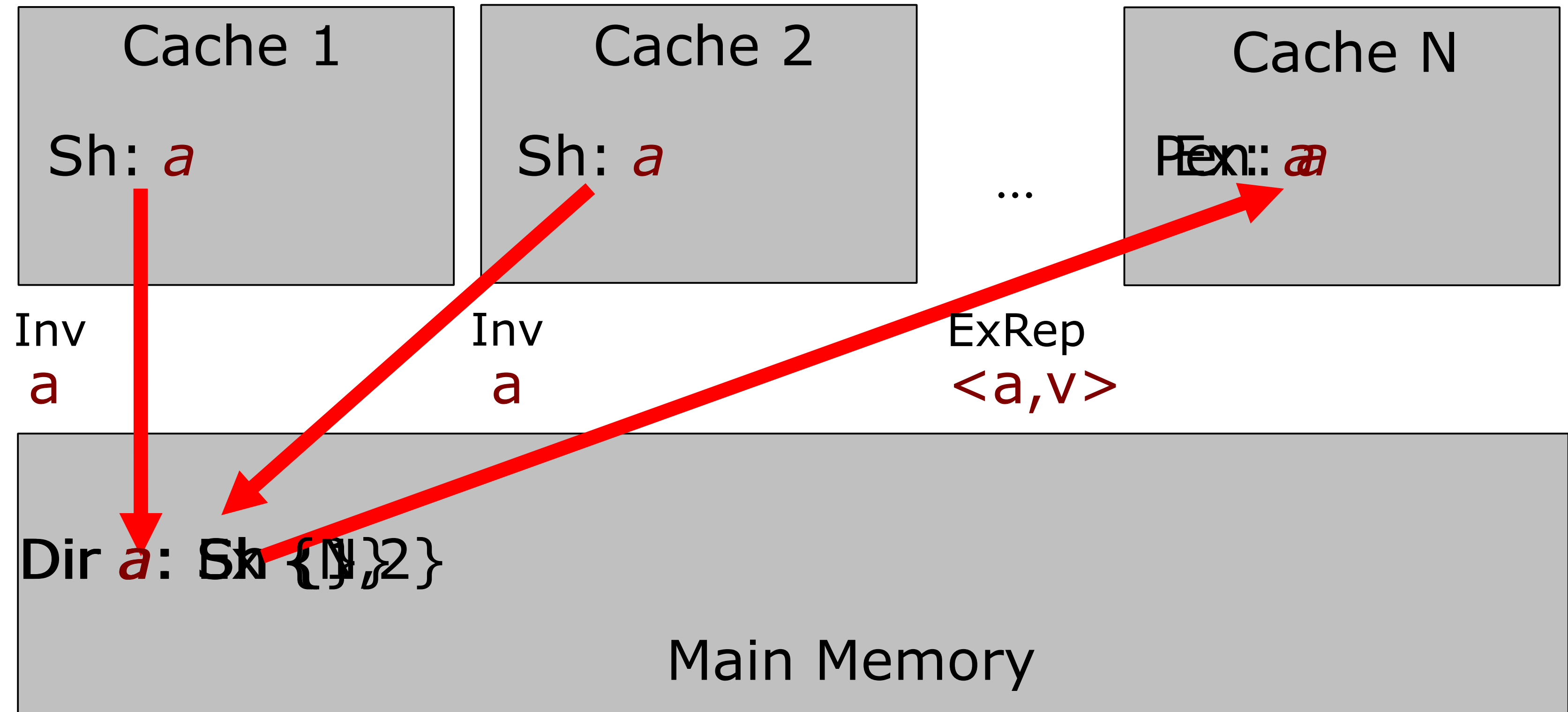
Protocol Diagram



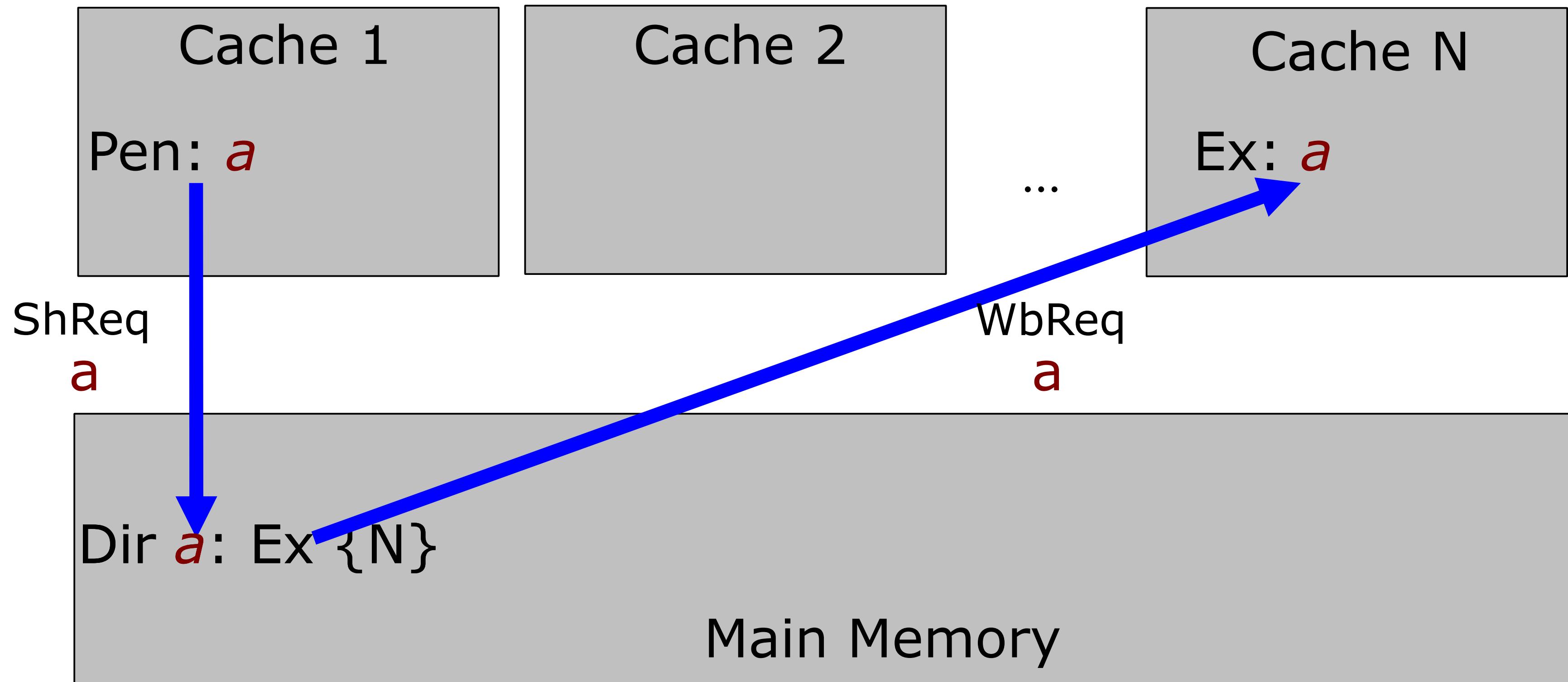
Protocol Diagram



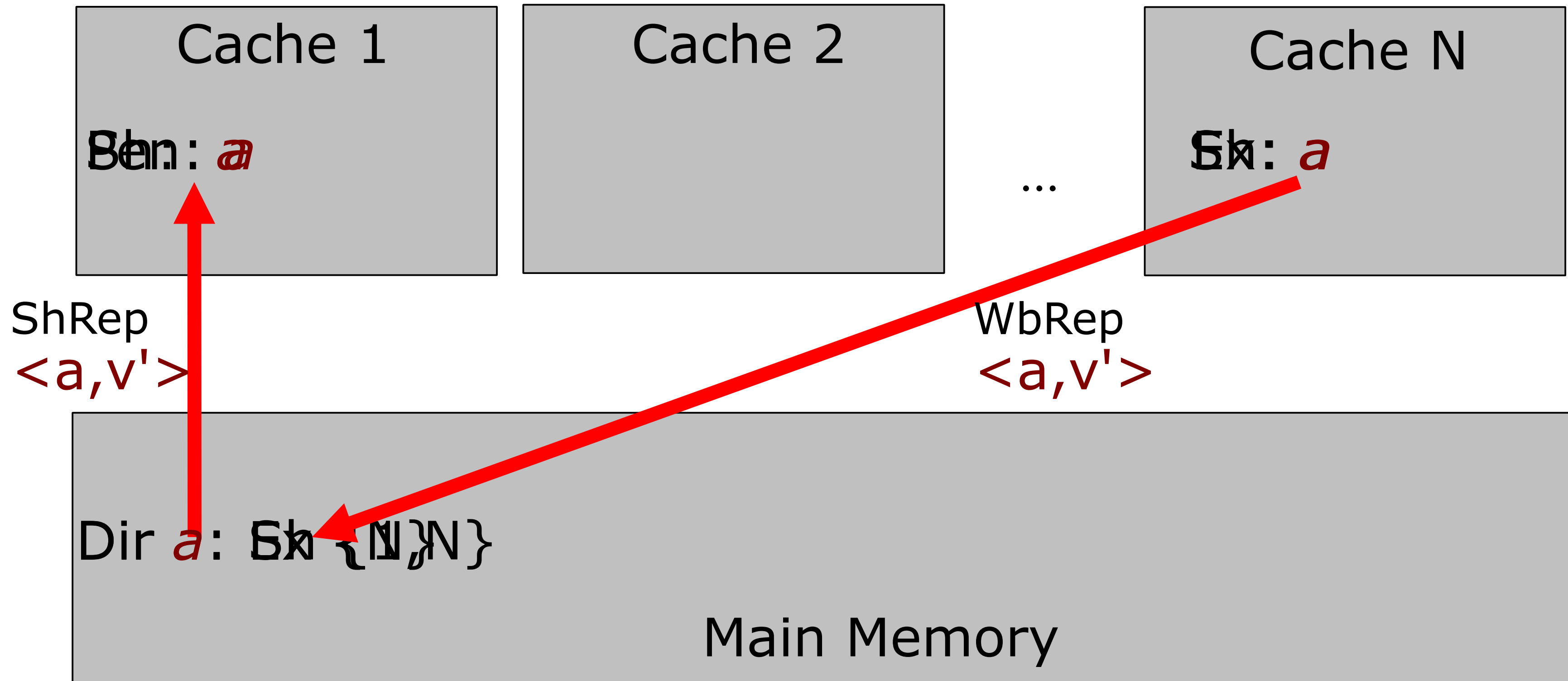
Protocol Diagram



Protocol Diagram



Protocol Diagram



Thank you