Lecture 5: Memory and Concurrent Access

Based on material from Chapter 5, Specifying Systems by Leslie Lamport

Today’s plan

- Review TLA+
- Specify a memory interface
- Specify linearizable memory
- Implement a linearizable cache on top of linearizable memory
- Review refinement

TLA+ review

Definition: *State*

- A *state* is an assignment of values to (*all*) variables
- TLA+ notation: $[var_1 = value_1, var_2 = value_2, \cdots]$

Definition: *Behavior*

- A *behavior* is a sequence of states
- Notation: $state_1 \rightarrow state_2 \rightarrow state_3 \rightarrow \cdots$
- Example: $[hr = 11] \rightarrow [hr = 12] \rightarrow [hr = 1]$

Definition: *Step*

- A *step* consists of two consecutive states in a behavior
- aka *transition*
- Notation: $state_1 \rightarrow state_2$
- Example: $[hr = 3] \rightarrow [hr = 4]$

Definition: *Specification*

- A *specification* is a set of all possible behaviors
- Consists of at least two parts
    1. Set of all possible *initial states*
    2. A "*next-state*" relation that describes the ways a state may change in a step
        - i.e., the set of all possible pairs of states
- May also contain a liveness condition and some theorems

Set of Initial States

- Example: HCini $\triangleq hr \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$
- A set of states can often be succinctly described by a predicate
  - Example: HCini $\triangleq hr \in \mathbb{N} \land 1 \leq hr \land hr \leq 12$

Definition: *Action*

- An *action* is a predicate over a pair of states in a step
- Example: HCnxt $\triangleq hr' = hr \ \% \ 12 + 1$
- hr' is the value of hr in the new state; hr is the value in the old state

Definition: *Stuttering steps*

- A stuttering step keeps (certain) state variable unchanged
- Example:

$$[hr' = hr \% 12 + 1]_{hr} \triangleq (hr' = hr \% 12 + 1) \lor (hr' = hr)$$

Definition: *State Function/Predicate*

- A *state function* is a first-order logic expression
- A *state predicate* is a Boolean state function

Definition: *Temporal Formula*

- A *temporal formula* F assigns a Boolean value to a behavior σ
- $\sigma \vDash F$ means that F holds over σ
- If P is a state predicate, then $\sigma \vDash P$ means that P holds over the first state in σ
- If A is an action, then $\sigma \vDash A$ means that A holds over the first two states in σ
- If A is an action, then $\sigma \vDash [A]_v$ means that the first step in σ is an A step or a stuttering step with respect to v

$\Box$Always

- $\sigma \vDash \Box F$ means that F holds over every suffix of σ
- More formally
  - Let σ^{+n} be σ with the first n states removed
  - Then $\sigma \vDash \Box F \triangleq \forall n \in \mathbb{N}: \sigma^{+n} \vDash F$

Example specification: hardware clock

Module HourClock

- VARIABLE hr
- HCini $\triangleq hr \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$
- HCnxt $\triangleq hr' = hr \bmod 12 + 1$
- HC $\triangleq$ HCini $\wedge \square[\text{HCnxt}]_{hr}$

Definition: *Theorem*

- A *theorem* is a temporal formula that holds over every behavior of the specification
- Example: HC $\Rightarrow \Box\ hr \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$
  - That is, HC $\Rightarrow \Box$ HCini

Definition: *Invariant*

- If S is a specification and I is a predicate and $S \Rightarrow \Box I$ is a theorem then we call I an *invariant* of S.

Modeling Shared Memory

(naïve) attempt: function [Address ↦ Value]

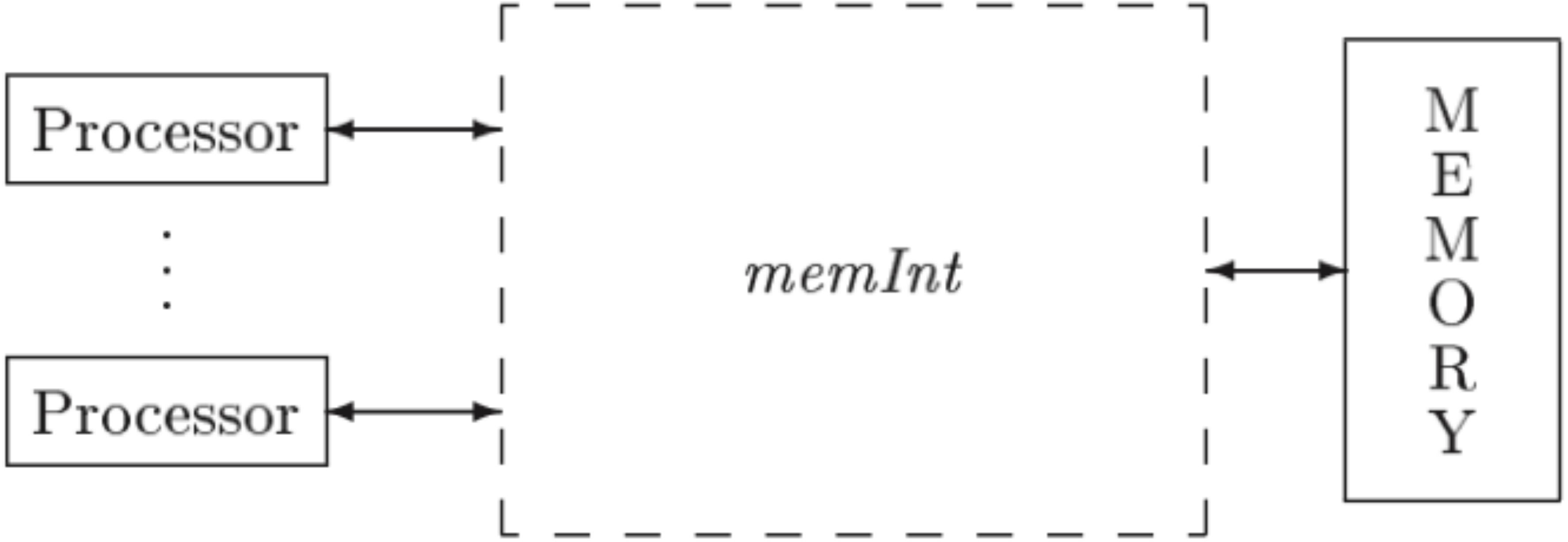
- CONSTANTS Adr, Val
- VARIABLE mem
- TypeInvariant $\triangleq mem \in [Adr \rightarrow Val]$
- Read$(a) \triangleq mem[a]$
- Write$(a, v) \triangleq mem' = [mem \text{ EXCEPT } ! [a] = v]$

(naïve) attempt 1: function [Address $\mapsto$ Value]

- CONSTANTS Adr, Val
- VARIABLE mem
- TypeInvariant $\triangleq mem \in [Adr \rightarrow Val]$
- Read$(a) \triangleq mem[a]$
- Write$(a, v) \triangleq mem' = [mem \text{ EXCEPT } ! [a] = v]$

Ignores how processes interact with memory
Ignores memory coherence properties

A Memory System



An abstract memory interface

MODULE *MemoryInterface*

VARIABLE $memInt$

CONSTANTS $Send(\_, \_, \_, \_)$, A $Send(p, d, memInt, memInt')$ step represents processor p sending value d to the memory.

$Reply(\_, \_, \_, \_)$, A $Reply(p, d, memInt, memInt')$ step represents the memory sending value d to processor p.

$InitMemInt$, The set of possible initial values of $memInt$.

$Proc$, The set of processor identifiers.

Adr, The set of memory addresses.

Val The set of memory values.

ASSUME $\forall\, p, d, miOld, miNew : \land\ Send(p, d, miOld, miNew) \in \text{BOOLEAN}$

$\qquad\qquad\qquad\qquad\qquad\qquad\ \land\ Reply(p, d, miOld, miNew) \in \text{BOOLEAN}$

MODULE *MemoryInterface*

VARIABLE $memInt$

CONSTANTS $Send(\_, \_, \_, \_)$, A $Send(p, d, memInt, memInt')$ step represents processor p sending value d to the memory.

$Reply(\_, \_, \_, \_)$, A $Reply(p, d, memInt, memInt')$ step represents the memory sending value d to processor p.

$InitMemInt$,	The set of possible initial values of $memInt$.
$Proc$,	The set of processor identifiers.
Adr,	The set of memory addresses.
Val	The set of memory values.

parameters

ASSUME $\forall\, p, d, miOld, miNew : \land\ Send(p, d, miOld, miNew) \in \text{BOOLEAN}$

$\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\ Reply(p, d, miOld, miNew) \in \text{BOOLEAN}$

MODULE *MemoryInterface*

VARIABLE *memInt*

CONSTANTS

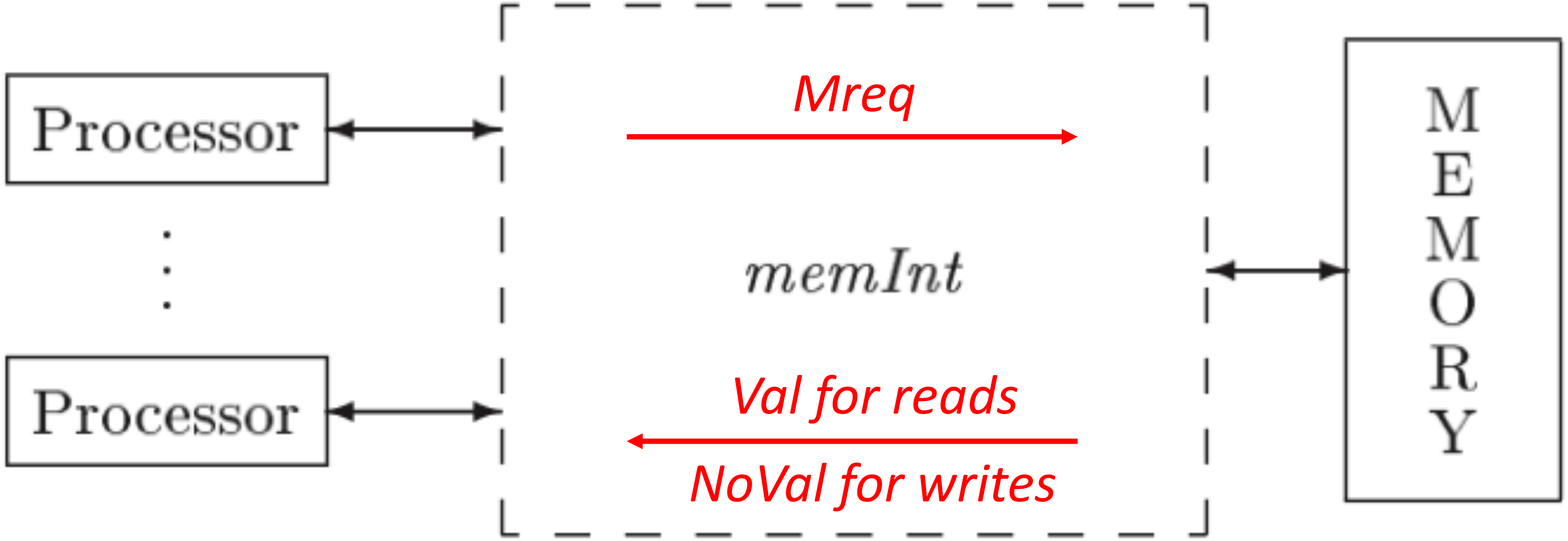
$Send(\_, \_, \_, \_)$,	A $Send(p, d, memInt, memInt')$ step represents processor p sending value d to the memory.
$Reply(\_, \_, \_, \_)$,	A $Reply(p, d, memInt, memInt')$ step represents the memory sending value d to processor p.
$InitMemInt$,	The set of possible initial values of $memInt$.
$Proc$,	The set of processor identifiers.
Adr,	The set of memory addresses.
Val	The set of memory values.

ASSUME $\forall\, p, d, miOld, miNew : \land\ Send(p, d, miOld, miNew) \in \text{BOOLEAN}$
$\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\ Reply(p, d, miOld, miNew) \in \text{BOOLEAN}$

$MReq \triangleq [op : \{\text{“Rd”}\},\ adr : Adr] \ \cup\ [op : \{\text{“Wr”}\},\ adr : Adr,\ val : Val]$

The set of all requests; a read specifies an address, a write specifies an address and a value.

A Memory System



MODULE *MemoryInterface*

VARIABLE $memInt$

CONSTANTS $Send(\_, \_, \_, \_)$, A $Send(p, d, memInt, memInt')$ step represents processor p sending value d to the memory.

$Reply(\_, \_, \_, \_)$, A $Reply(p, d, memInt, memInt')$ step represents the memory sending value d to processor p.

$InitMemInt$, The set of possible initial values of $memInt$.

$Proc$, The set of processor identifiers.

Adr, The set of memory addresses.

Val The set of memory values.

ASSUME $\forall\, p, d, miOld, miNew : \land\ Send(p, d, miOld, miNew) \in \text{BOOLEAN}$

$\qquad\qquad\qquad\qquad\qquad\qquad\ \ \land\ Reply(p, d, miOld, miNew) \in \text{BOOLEAN}$

$MReq \triangleq [op : \{\text{“Rd”}\},\ adr : Adr] \ \cup\ [op : \{\text{“Wr”}\},\ adr : Adr,\ val : Val]$

The set of all requests; a read specifies an address, a write specifies an address and a value.

$NoVal \triangleq \text{CHOOSE } v : v \notin Val$ An arbitrary value not in Val.

A Linearizable Memory System

Linearizability [Herlihy & Wing 1990]

$$[\quad] \xrightarrow{Req(p)} [\quad] \xrightarrow{Do(p)} [\quad] \xrightarrow{Rsp(p)} [\quad]$$

Linearization Step

MODULE *InternalMemory*

EXTENDS *MemoryInterface*

VARIABLES mem, ctl, buf

$IInit \triangleq$ The initial predicate

$$\begin{array}{l} \land\ mem \in [Adr \rightarrow Val] \\ \land\ ctl = [p \in Proc \mapsto \text{“rdy”}] \\ \land\ buf = [p \in Proc \mapsto NoVal] \\ \land\ memInt \in InitMemInt \end{array}$$

Initially, memory locations have any values in Val, each processor is ready to issue requests, each $buf[p]$ is arbitrarily initialized to $NoVal$, and $memInt$ is any element of $InitMemInt$.

$TypeInvariant \triangleq$ The type-correctness invariant.

$$\begin{array}{ll} \land\ mem \in [Adr \rightarrow Val] & \text{mem is a function from Adr to Val.} \\ \land\ ctl \in [Proc \rightarrow \{\text{“rdy”}, \text{“busy”}, \text{“done”}\}] & \text{$ctl[p]$ equals “rdy”, “busy”, or “done”.} \\ \land\ buf \in [Proc \rightarrow MReq \cup Val \cup \{NoVal\}] & \text{$buf[p]$ is a request or a response.} \end{array}$$

Behaviors

$$req \equiv [op \mapsto \text{"}Wr\text{"}, adr \mapsto a, val \mapsto v]$$

$$\begin{bmatrix} ctl[p] & = & \text{"}rdy\text{"} \\ buf[p] & = & \ldots \\ mem[a] & = & \ldots \end{bmatrix} \xrightarrow{Req(p)} \begin{bmatrix} ctl[p] & = & \text{"}busy\text{"} \\ buf[p] & = & req \\ mem[a] & = & \ldots \end{bmatrix} \xrightarrow{Do(p)} \begin{bmatrix} ctl[p] & = & \text{"}done\text{"} \\ buf[p] & = & NoVal \\ mem[a] & = & v \end{bmatrix} \xrightarrow{Rsp(p)} \begin{bmatrix} ctl[p] & = & \text{"}rdy\text{"} \\ buf[p] & = & NoVal \\ mem[a] & = & v \end{bmatrix}$$

$$req \equiv [op \mapsto \text{"}Rd\text{"}, adr \mapsto a]$$

$$\begin{bmatrix} ctl[p] & = & \text{"}rdy\text{"} \\ buf[p] & = & \ldots \\ mem[a] & = & \ldots \end{bmatrix} \xrightarrow{Req(p)} \begin{bmatrix} ctl[p] & = & \text{"}busy\text{"} \\ buf[p] & = & req \\ mem[a] & = & v \end{bmatrix} \xrightarrow{Do(p)} \begin{bmatrix} ctl[p] & = & \text{"}done\text{"} \\ buf[p] & = & v \\ mem[a] & = & \ldots \end{bmatrix} \xrightarrow{Rsp(p)} \begin{bmatrix} ctl[p] & = & \text{"}rdy\text{"} \\ buf[p] & = & v \\ mem[a] & = & \ldots \end{bmatrix}$$

$$
\begin{array}{ll}
Req(p) \triangleq & \text{Processor } p \text{ issues a request.} \\
\quad \land\ ctl[p] = \text{“rdy”} & \text{Enabled iff } p \text{ is ready to issue a request.} \\
\quad \land\ \exists\, req \in MReq : & \text{For some request } req\text{:} \\
\qquad\quad \land\ Send(p, req, memInt, memInt') & \text{Send } req \text{ on the interface.} \\
\qquad\quad \land\ buf' = [buf \text{ EXCEPT } ![p] = req] & \text{Set } buf[p] \text{ to the request.} \\
\qquad\quad \land\ ctl' = [ctl \text{ EXCEPT } ![p] = \text{“busy”}] & \text{Set } ctl[p] \text{ to “busy”.} \\
\quad \land\ \text{UNCHANGED } mem &
\end{array}
$$

$$
\begin{bmatrix} ctl[p] & = & \text{"}rdy\text{"} \\ buf[p] & = & \dots \\ mem[a] & = & \dots \end{bmatrix}
\xrightarrow{Req(p)}
\begin{bmatrix} ctl[p] & = & \text{"}busy\text{"} \\ buf[p] & = & req \\ mem[a] & = & \dots \end{bmatrix}
\xrightarrow{Do(p)}
$$

```
$Do(p) \triangleq$                                          Perform $p$'s request to memory.
    $\land\ ctl[p] = \text{"busy"}$                         Enabled iff $p$'s request is pending.
    $\land\ mem' = \text{IF}\ \ buf[p].op = \text{"Wr"}$
                 $\text{THEN}\ \ [mem\ \text{EXCEPT}$                 Write to memory on a
                            $![buf[p].adr] = buf[p].val]$        "Wr" request.
                 $\text{ELSE}\ \ mem$                             Leave $mem$ unchanged on a "Rd" request.
    $\land\ buf' = [buf\ \text{EXCEPT}$
                 $![p] = \text{IF}\ \ buf[p].op = \text{"Wr"}$            Set $buf[p]$ to the response:
                          $\text{THEN}\ \ NoVal$                           $NoVal$ for a write;
                          $\text{ELSE}\ \ mem[buf[p].adr]]$                the memory value for a read.
    $\land\ ctl' = [ctl\ \text{EXCEPT}\ ![p] = \text{"done"}]$       Set $ctl[p]$ to "done".
    $\land\ \text{UNCHANGED}\ memInt$
```

$$\begin{bmatrix} ctl[p] & = & \text{"busy"} \\ buf[p] & = & req \\ mem[a] & = & \ldots \end{bmatrix} \xrightarrow{Do(p)} \begin{bmatrix} ctl[p] & = & \text{"done"} \\ buf[p] & = & NoVal \\ mem[a] & = & v \end{bmatrix} \xrightarrow{Rsp(p)}$$

$Rsp(p) \triangleq$ Return the response to p's request.

$\land\ ctl[p] =$ "done" — Enabled iff req. is done but resp. not sent.

$\land\ Reply(p,\ buf[p],\ memInt,\ memInt')$ — Send the response on the interface.

$\land\ ctl' = [ctl$ EXCEPT $![p] =$ "rdy"$]$ — Set $ctl[p]$ to "rdy".

$\land$ UNCHANGED $\langle mem, buf \rangle$

$$\begin{bmatrix} ctl[p] & = & \text{"done"} \\ buf[p] & = & NoVal \\ mem[a] & = & v \end{bmatrix} \xrightarrow{Rsp(p)} \begin{bmatrix} ctl[p] & = & \text{"rdy"} \\ buf[p] & = & NoVal \\ mem[a] & = & v \end{bmatrix}$$

MODULE *InternalMemory*

EXTENDS *MemoryInterface*

VARIABLES mem, ctl, buf

$IInit \triangleq$ The initial predicate

$\land\ mem \in [Adr \rightarrow Val]$
$\land\ ctl = [p \in Proc \mapsto \text{“rdy”}]$
$\land\ buf = [p \in Proc \mapsto NoVal]$
$\land\ memInt \in InitMemInt$

Initially, memory locations have any values in Val, each processor is ready to issue requests, each $buf[p]$ is arbitrarily initialized to $NoVal$, and $memInt$ is any element of $InitMemInt$.

[redacted]

$INext \triangleq \exists\, p \in Proc : Req(p) \lor Do(p) \lor Rsp(p)$ The next-state action.

$ISpec \triangleq IInit \land \Box[INext]_{\langle memInt,\, mem,\, ctl,\, buf \rangle}$ The specification.

THEOREM $ISpec \Rightarrow \Box\, TypeInvariant$

MODULE *Memory*

EXTENDS *MemoryInterface*

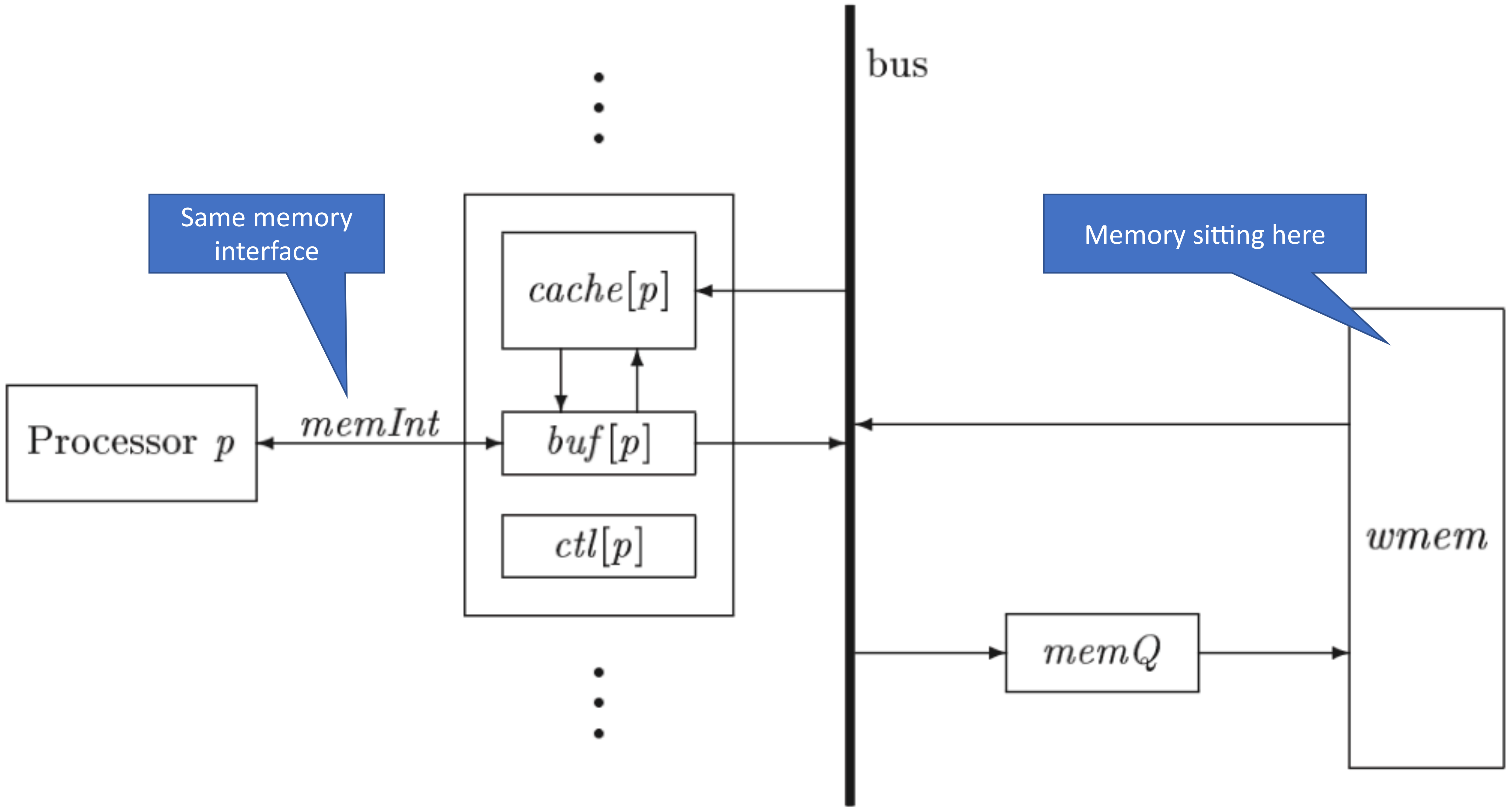
$Inner(mem, ctl, buf) \triangleq$ INSTANCE *InternalMemory*

$Spec \triangleq \exists\, mem, ctl, buf : Inner(mem, ctl, buf)!ISpec$

Note, this is a “specification” describing behaviors of linearizable memory more than an “implementation” (mapping onto physical hardware). Of course, both can be described by TLA+ formulas.

Implementing a Write-Through Cache

Implements the linearizable memory interface (on top of another)


Same memory interface
Memory sitting here
bus
Processor p
$memInt$
$cache[p]$
$buf[p]$
$ctl[p]$
$memQ$
$wmem$

MODULE *WriteThroughCache*

EXTENDS *Naturals*, *Sequences*, *MemoryInterface*

VARIABLES *wmem*, *ctl*, *buf*, *cache*, *memQ*

CONSTANT *QLen*

ASSUME $(QLen \in Nat) \land (QLen > 0)$

$M \triangleq$ INSTANCE *InternalMemory* WITH $mem \leftarrow wmem$

mem renamed for clarity

$Init \triangleq$ The initial predicate

$\land\ M!IInit$ *wmem*, *buf*, and *ctl* are initialized as in the internal memory spec.

$\land\ cache =$ All caches are initially empty ($cache[p][a] = NoVal$ for all p, a).

$[p \in Proc \mapsto [a \in Adr \mapsto NoVal]]$

$\land\ memQ = \langle \rangle$ The queue *memQ* is initially empty.

$TypeInvariant \triangleq$ The type invariant.

$\land\ wmem \in [Adr \rightarrow Val]$

$\land\ ctl \in [Proc \rightarrow \{\text{“rdy”}, \text{“busy”}, \text{“waiting”}, \text{“done”}\}]$

$\land\ buf \in [Proc \rightarrow MReq \cup Val \cup \{NoVal\}]$

$\land\ cache \in [Proc \rightarrow [Adr \rightarrow Val \cup \{NoVal\}]]$

$\land\ memQ \in Seq(Proc \times MReq)$ *memQ* is a sequence of ⟨proc., request⟩ pairs.

New *ctl* state ”waiting” added

Note

In TLA+

- Sequences and tuples are functions [[1 …] → values]
- Recall: records are functions [String → values]

MODULE *WriteThroughCache*

EXTENDS *Naturals*, *Sequences*, *MemoryInterface*

VARIABLES *wmem*, *ctl*, *buf*, *cache*, *memQ*

CONSTANT *QLen*

ASSUME $(QLen \in Nat) \land (QLen > 0)$

$M \triangleq$ INSTANCE *InternalMemory* WITH $mem \leftarrow wmem$

mem renamed for clarity

$Init \triangleq$ The initial predicate

$\land\ M!IInit$ — *wmem*, *buf*, and *ctl* are initialized as in the internal memory spec.

$\land\ cache =$ — All caches are initially empty ($cache[p][a] = NoVal$ for all p, a).

$[p \in Proc \mapsto [a \in Adr \mapsto NoVal]]$

$\land\ memQ = \langle\rangle$ — The queue *memQ* is initially empty.

$TypeInvariant \triangleq$ The type invariant.

$\land\ wmem \in [Adr \rightarrow Val]$

$\land\ ctl \in [Proc \rightarrow \{$"rdy", "busy", "waiting", "done"$\}]$

$\land\ buf \in [Proc \rightarrow MReq \cup Val \cup \{NoVal\}]$

$\land\ cache \in [Proc \rightarrow [Adr \rightarrow Val \cup \{NoVal\}]]$

$\land\ memQ \in Seq(Proc \times MReq)$ — *memQ* is a sequence of ⟨proc., request⟩ pairs.

New *ctl* state "waiting" added

Cache Coherence

$$Coherence \triangleq$$

Asserts that if two processors' caches both have copies of an address, then those copies have equal values.

$$\forall\, p, q \in Proc,\ a \in Adr :$$
$$(NoVal \notin \{cache[p][a],\ cache[q][a]\}) \;\Rightarrow\; (cache[p][a] = cache[q][a])$$

External Interface is the same

$Req(p) \triangleq$ Processor p issues a request.

$\quad M!Req(p) \land \text{UNCHANGED } \langle cache, memQ \rangle$

$Rsp(p) \triangleq$ The system issues a response to processor p.

$\quad M!Rsp(p) \land \text{UNCHANGED } \langle cache, memQ \rangle$

$$\begin{bmatrix} ctl[p] & = & \text{"rdy"} \\ buf[p] & = & \dots \\ & \dots & \end{bmatrix} \xrightarrow{Req(p)} ???????? \xrightarrow{Rsp(p)} \begin{bmatrix} ctl[p] & = & \text{"rdy"} \\ buf[p] & = & \dots \\ & \dots & \end{bmatrix}$$

```
$DoWr(p) \triangleq$   Write to $p$'s cache, update other caches, and enqueue memory update.
  LET $r \triangleq buf[p]$   Processor $p$'s request.
  IN  $\land (ctl[p] = \text{"busy"}) \land (r.op = \text{"Wr"})$   Enabled if write request pending
      $\land Len(memQ) < QLen$                                           and $memQ$ is not full.
      $\land cache' =$   Update $p$'s cache and any other cache that has a copy.
          $[q \in Proc \mapsto \text{IF } (p = q) \lor (cache[q][r.adr] \neq NoVal)$
                               $\text{THEN } [cache[q] \text{ EXCEPT } ![r.adr] = r.val]$
                               $\text{ELSE } cache[q]]$
      $\land memQ' = Append(memQ, \langle p, r \rangle)$   Enqueue write at tail of $memQ$.
      $\land buf' = [buf \text{ EXCEPT } ![p] = NoVal]$   Generate response.
      $\land ctl' = [ctl \text{ EXCEPT } ![p] = \text{"done"}]$   Set $ctl$ to indicate request is done.
      $\land \text{UNCHANGED } \langle memInt, wmem \rangle$
```

$MemQWr \ \triangleq$ Perform write at head of $memQ$ to memory.

$\quad \text{LET } r \ \triangleq \ Head(memQ)[2]$ The request at the head of $memQ$.

$$\begin{array}{ll}
\text{IN} & \land\ (memQ \neq \langle\, \rangle) \land (r.op = \text{“Wr”}) \\
 & \land\ wmem' = \\
 & \qquad [wmem \text{ EXCEPT } ![r.adr] = r.val] \\
 & \land\ memQ' = Tail(memQ) \\
 & \land\ \text{UNCHANGED } \langle memInt,\ buf,\ ctl,\ cache \rangle
\end{array}$$

Enabled if $Head(memQ)$ a write.
Perform the write to memory.
Remove the write from $memQ$.

$RdMiss(p) \triangleq$ Enqueue a request to write value from memory to p's cache.

$$\begin{aligned}
&\land (ctl[p] = \text{“busy”}) \land (buf[p].op = \text{“Rd”}) \\
&\land cache[p][buf[p].adr] = NoVal \\
&\land Len(memQ) < QLen \\
&\land memQ' = Append(memQ, \langle p, buf[p] \rangle) \\
&\land ctl' = [ctl \text{ EXCEPT } ![p] = \text{“waiting”}] \\
&\land \text{UNCHANGED } \langle memInt,\ wmem,\ buf,\ cache \rangle
\end{aligned}$$

Enabled on a read request when the address is not in p's cache and $memQ$ is not full.
Append $\langle p, \text{request} \rangle$ to $memQ$.
Set $ctl[p]$ to “waiting”.

$DoRd(p) \triangleq$ Perform a read by p of a value in its cache.

$$\begin{aligned}
&\land ctl[p] \in \{\text{“busy”}, \text{“waiting”}\} \\
&\land buf[p].op = \text{“Rd”} \\
&\land cache[p][buf[p].adr] \neq NoVal \\
&\land buf' = [buf \text{ EXCEPT } ![p] = cache[p][buf[p].adr]] \\
&\land ctl' = [ctl \text{ EXCEPT } ![p] = \text{“done”}] \\
&\land \text{UNCHANGED } \langle memInt,\ wmem,\ cache,\ memQ \rangle
\end{aligned}$$

Enabled if a read request is pending and address is in cache.
Get result from cache.
Set $ctl[p]$ to “done”.

```
$vmem \triangleq$   The value wmem will have after all the writes in memQ are performed.
  LET $f[i \in 0 \,..\, Len(memQ)] \triangleq$   The value wmem will have after the first
                                                   i writes in memQ are performed.
        IF $i = 0$ THEN $wmem$
               ELSE IF $memQ[i][2].op = \text{“Rd”}$
                      THEN $f[i-1]$
                      ELSE $[f[i-1] \text{ EXCEPT } ![memQ[i][2].adr] =$
                                                $memQ[i][2].val]$
  IN  $f[Len(memQ)]$

$MemQRd \triangleq$   Perform an enqueued read to memory.
  LET $p \triangleq Head(memQ)[1]$   The requesting processor.
      $r \triangleq Head(memQ)[2]$   The request at the head of memQ.
  IN  $\land (memQ \neq \langle\rangle) \land (r.op = \text{“Rd”})$   Enabled if Head(memQ) is a read.
      $\land memQ' = Tail(memQ)$                                  Remove the head of memQ.
      $\land cache' =$   Put value from memory or memQ in p’s cache.
            $[cache \text{ EXCEPT } ![p][r.adr] = vmem[r.adr]]$
      $\land \text{UNCHANGED } \langle memInt, wmem, buf, ctl \rangle$
```

Completing the spec

$Evict(p, a) \triangleq$ Remove address a from p's cache.

$$\begin{aligned}
&\land (ctl[p] = \text{“waiting”}) \Rightarrow (buf[p].adr \neq a)\\
&\land cache' = [cache \text{ EXCEPT } ![p][a] = NoVal]\\
&\land \text{UNCHANGED } \langle memInt, wmem, buf, ctl, memQ \rangle
\end{aligned}$$

Can't evict a if it was just read into cache from memory.

$$\begin{aligned}
Next \triangleq\ &\lor \exists p \in Proc : \lor Req(p) \lor Rsp(p)\\
&\qquad\qquad\qquad \lor RdMiss(p) \lor DoRd(p) \lor DoWr(p)\\
&\qquad\qquad\qquad \lor \exists a \in Adr : Evict(p, a)\\
&\lor MemQWr \lor MemQRd
\end{aligned}$$

$$Spec \triangleq Init \land \Box[Next]_{\langle memInt, wmem, buf, ctl, cache, memQ \rangle}$$

Theorems

$$\begin{array}{l}\text{THEOREM } \mathit{Spec} \Rightarrow \Box \mathit{TypeInvariant} \\ \text{THEOREM } \mathit{Spec} \Rightarrow \Box \mathit{Coherence}\end{array}\Bigg\}$$ More like lemmas

$$\boxed{\begin{array}{l}\mathit{LM} \triangleq \text{INSTANCE } \mathit{Memory} \\ \text{THEOREM } \mathit{Spec} \Rightarrow \mathit{LM}!\mathit{Spec}\end{array}}$$

Inductive Invariants

$$\text{THEOREM } \mathit{TypeInvariant} \land \mathit{Next} \Rightarrow \mathit{TypeInvariant}'$$

TypeInvariant is an invariant of the next-state action

Thus, if *TypeInvariant* holds over initial states, by induction it holds over all states

Coherence is not an inductive invariant

- Consider a state in which:
  - $cache[p1][a] = 1$
  - $\forall \langle q, b \rangle : cache[q][b] = NoVal$
  - $wmem[a] = 2$
  - $memQ = \langle \langle p2, [op \mapsto "Rd", adr \mapsto a] \rangle \rangle$

} Coherence satisfied

- Now take the *MemQRd* step:
  - $cache[p1][a] = 1$
  - $cache[p2][a] = 2$

} Coherence violated

Need to prove an inductive invariant that implies Coherence

Suggestions?

A proposed stronger invariant

- Recall that function $vmem$ represents current state of memory
- Inductive Invariant*:*
  $\forall p \in Proc, a \in Adr: (cache[p][a] = NoVal) \vee (cache[p][a] = vmem[a])$
- Implies *Coherence*

Proving $Spec \Rightarrow LM!Spec$

By definition of $LM!Spec$, we need to prove

$$\text{THEOREM } Spec \Rightarrow \exists\, mem, ctl, buf : LM!Inner(mem, ctl, buf)!ISpec$$

Which means we have to find "*witnesses*" for mem, ctl and buf: this is called a *refinement mapping*

Any guesses?

Proving $Spec \Rightarrow LM!Spec$

By definition of $LM!Spec$, we need to prove

$$\text{THEOREM } Spec \Rightarrow \exists\, mem, ctl, buf : LM!Inner(mem, ctl, buf)!ISpec$$

Which means we have to find "*witnesses*" for mem, ctl and buf: this is called a *refinement mapping*:

$$\begin{aligned}
omem &\triangleq vmem \\
octl &\triangleq [p \in Proc \mapsto \text{IF } ctl[p] = \text{"waiting" THEN "busy" ELSE } ctl[p]] \\
obuf &\triangleq buf
\end{aligned}$$

MODULE *InternalMemory*

EXTENDS *MemoryInterface*
VARIABLES *mem*, *ctl*, *buf*

$IInit \triangleq$ The initial predicate

$\wedge\ mem \in [Adr \rightarrow Val]$
$\wedge\ ctl = [p \in Proc \mapsto \text{“rdy”}]$
$\wedge\ buf = [p \in Proc \mapsto NoVal]$
$\wedge\ memInt \in InitMemInt$

Initially, memory locations have any values in Val, each processor is ready to issue requests, each $buf[p]$ is arbitrarily initialized to $NoVal$, and $memInt$ is any element of $InitMemInt$.

[redacted]

$INext \triangleq \exists\, p \in Proc : Req(p) \vee Do(p) \vee Rsp(p)$ The next-state action.

$ISpec \triangleq IInit \wedge \Box[INext]_{\langle memInt,\, mem,\, ctl,\, buf \rangle}$ The specification.

THEOREM $ISpec \Rightarrow \Box\, TypeInvariant$

Proving refinement

- If F is a formula of module *InternalMemory* (the high-level spec), let
  - $\overline{F} \equiv LM!Inner(omem, octl, obuf)$
  - That is: F with $omem$, $octl$, and $obuf$ substituted for mem, ctl, and buf
- Then we need to prove that $Spec \Rightarrow \overline{ISpec}$
- Replacing definitions, we need to prove:

$$Init \land \Box[Next]_{\langle memInt, wmem, buf, ctl, cache, memQ \rangle}$$

$$\Rightarrow \quad \overline{IInit} \land \Box[\overline{INext}]_{\langle memInt, \overline{mem}, \overline{ctl}, \overline{buf} \rangle}$$

- Find an invariant *Inv*: $\land\ Init \Rightarrow \overline{IInit}$

step simulation $\left[\begin{array}{l} \land\ Inv \land Next \ \Rightarrow \ \lor\ \overline{INext} \\ \qquad\qquad\qquad\qquad\quad \lor\ \text{UNCHANGED}\ \langle memInt, \overline{mem}, \overline{ctl}, \overline{buf} \rangle \end{array}\right.$

Show every step of *WriteThroughCache* is a step of *InternalMemory* or a stuttering step of *InternalMemory*

About memory

- Real memory is not linearizable
  - Linearizability is not strong enough for modern processors that submit multiple requests to memory
    - If a processor submits a write and, before completion, a read to the same address, linearizability would allow the second operation to be ordered before the first
  - Linearizability is too strong for concurrent processing
    - If p1 submits operation o1 and p2 submits operation o2 and o1 completes before o2, we do not need to require that o1 is ordered before o2 (use locks if you need that)
- Sequential Consistency is more realistic and easier to implement
  - Serializability: result of execution same as some total order of operations
  - Local ordering: operations of a process ordered in submission order
- See Figure 11.7 in *Specifying Systems*

Final words

- We use TLA+ to *model* a system. You get to choose a level of abstraction. Choose it too high and you won't reveal problems. Choose it too low and you get stuck in the weeds.
- Choosing the level of abstraction involves choosing what constitutes (atomic) steps: *grain of atomicity*
- Also involves how accurately to model the state (data structures). Consider where you are trying to reveal problems.