BSV and BH,

High-Level Hardware Design Languages (HLHDLs)

Rishiyur S. Nikhil
Co-founder and CTO, Bluespec, Inc.

Presentation at: OSDA (Open Source Design Automation) Workshop,
Antwerp, Belgium, April 17, 2023

Doc. rev. 2023-04-13.1

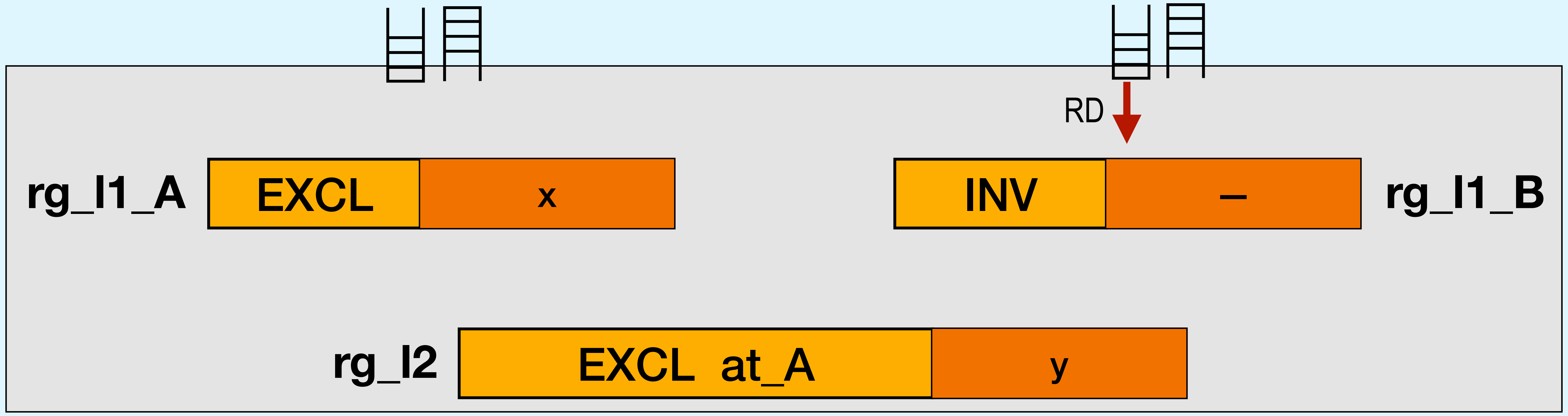
BSV and BH

Key points today

1. *“Condition-Action Rules” for formal and synthesizable spec of hardware systems*
    - *a.k.a. “Guarded Atomic Actions”*
    - *From abstract specs to implementation-level specs*
2. BSV and BH: High-Level Hardware Development Languages (HLHDLs) based on this idea

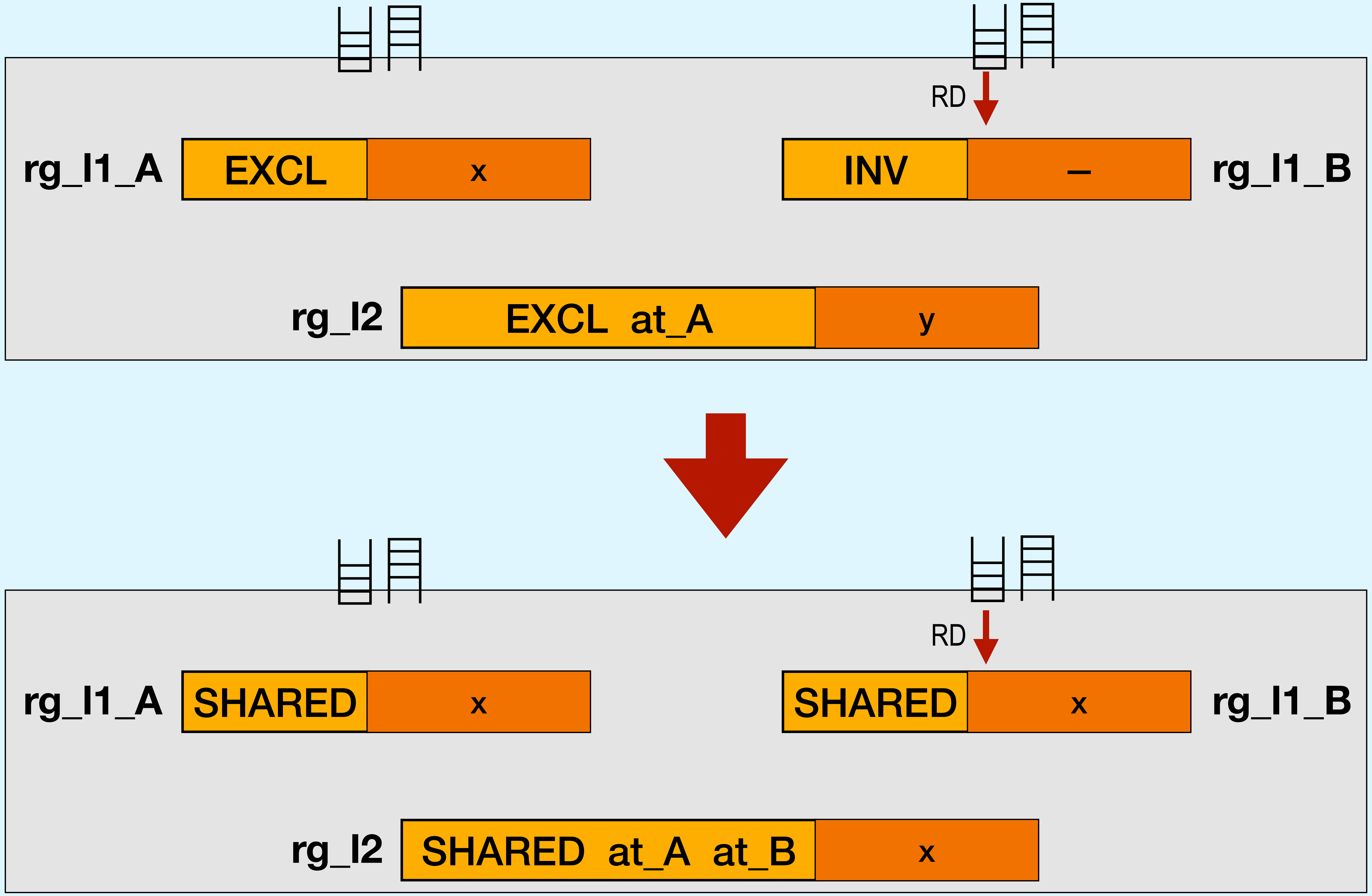
Specifying a (small) cache-coherence system with *rules*

Two “level 1” caches rg_l1_A and rg_l1_B, shared “level 2” cache rg_l2



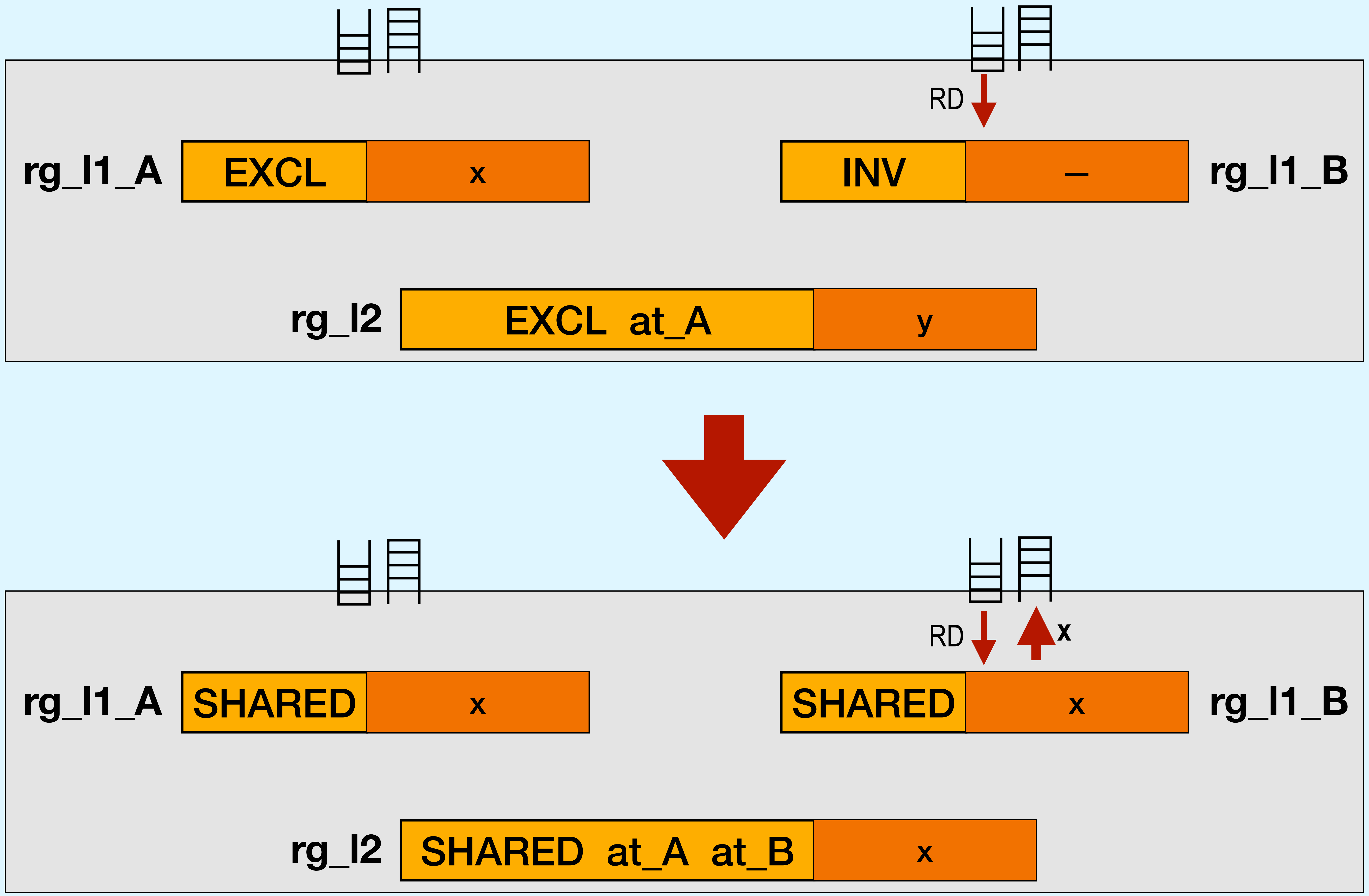
Specifying a (small) cache-coherence system with *rules*

Two “level 1” caches rg_l1_A and rg_l1_B, shared “level 2” cache rg_l2



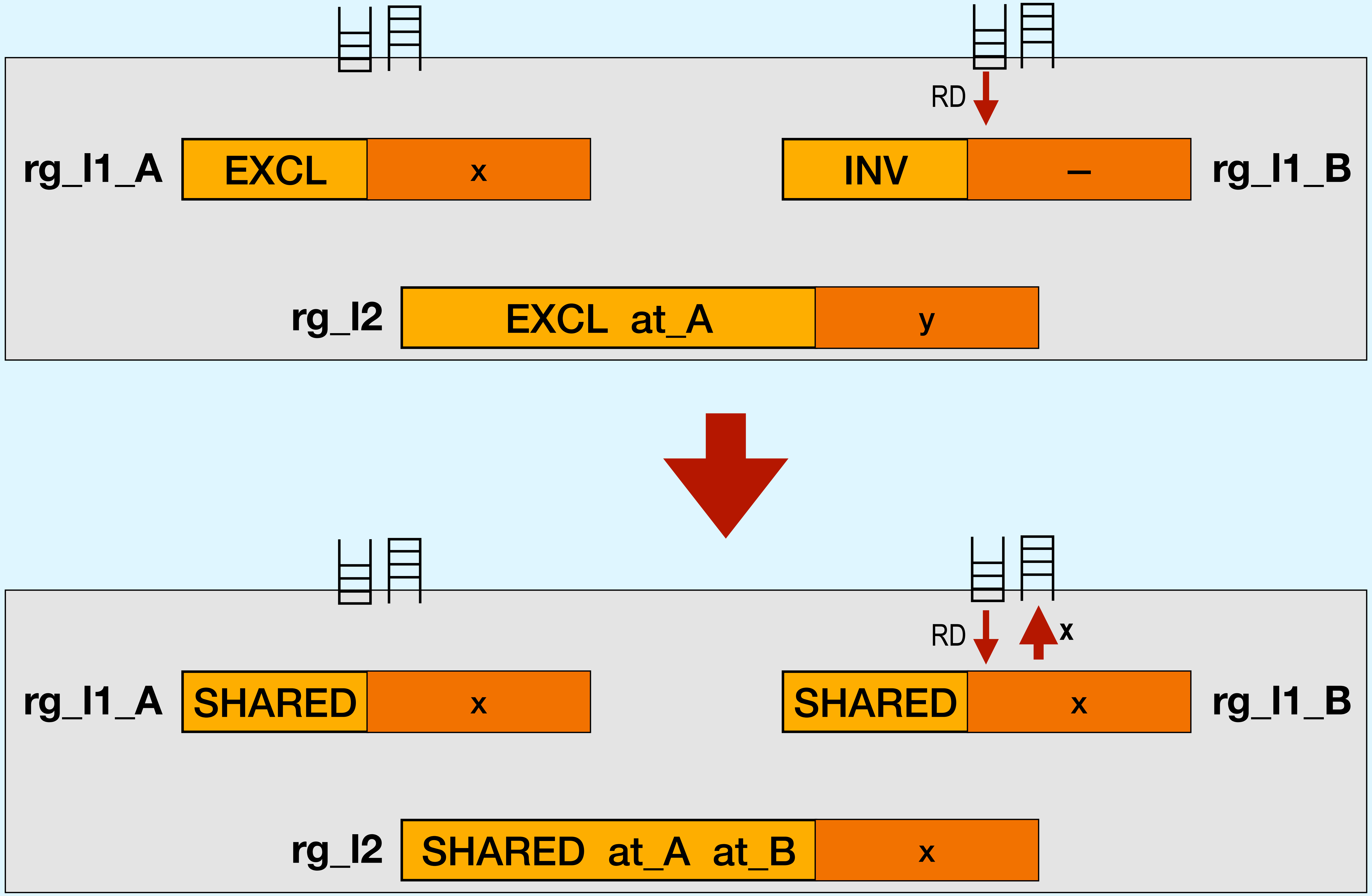
Specifying a (small) cache-coherence system with *rules*

Two “level 1” caches rg_l1_A and rg_l1_B, shared “level 2” cache rg_l2



Specifying a (small) cache-coherence system with *rules*

Two “level 1” caches rg_l1_A and rg_l1_B, shared “level 2” cache rg_l2



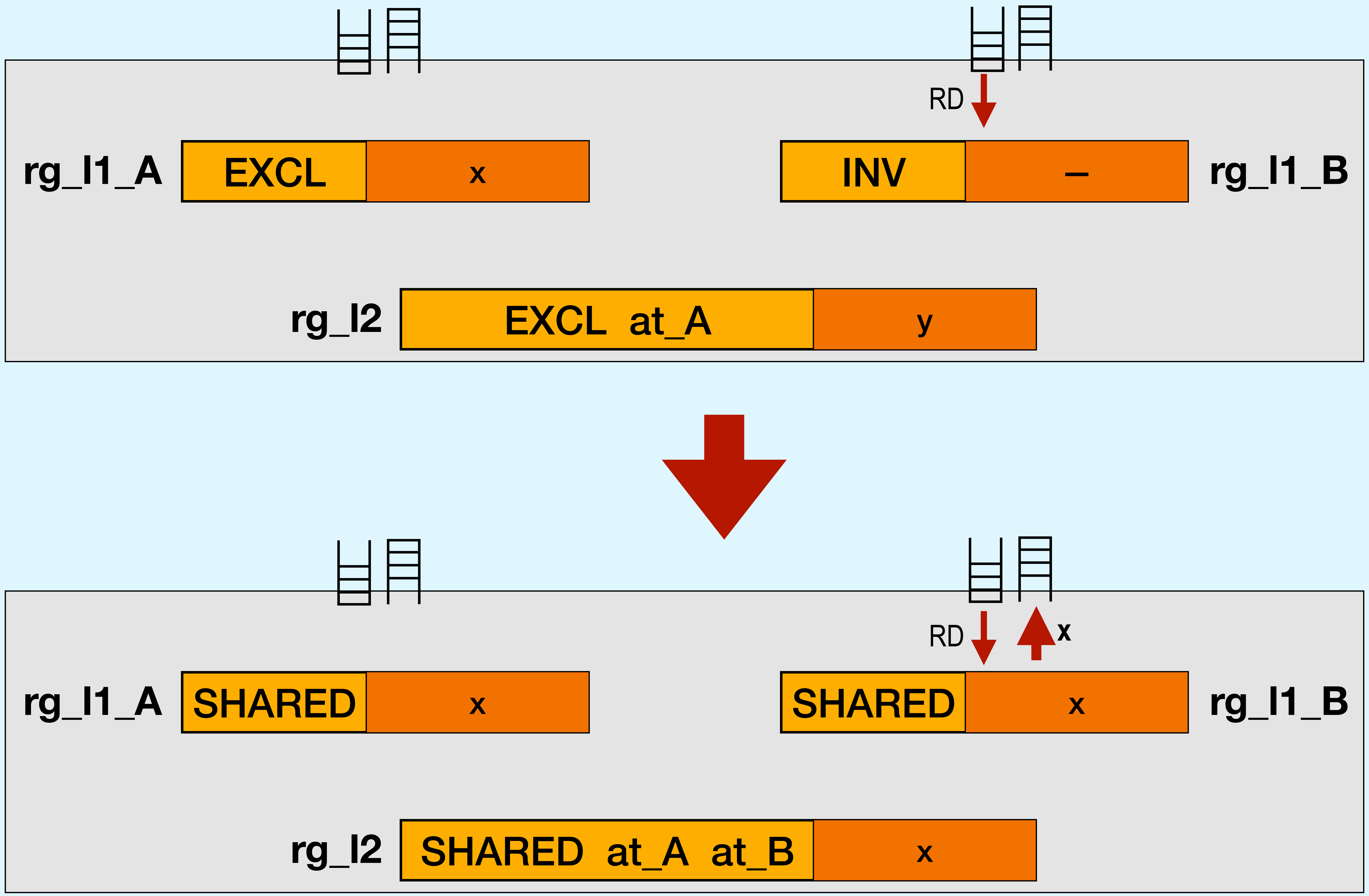
Condition

```
rule r1 (      (f_reqs_B.first.req_type == RD)
           &&  (rg_l1_B.cstate == INVALID)
           &&  (rg_l2.cstate == EXCLUSIVE));

  let x = rg_l1_A.data;
  rg_l1_A <= L1_Cell {cstate: SHARED, data: x};
  rg_l1_B <= L1_Cell {cstate: SHARED, data: x};
  rg_l2     <= L2_Cell {cstate: SHARED, at_A: True, at_B: True, data: x};
  f_rsps_B.enq (CPU_Rsp{data:x});
endrule
```

Action

Specifying a (small) cache-coherence system with *rules*

Two "level 1" caches rg_l1_A and rg_l1_B, shared "level 2" cache rg_l2



Condition

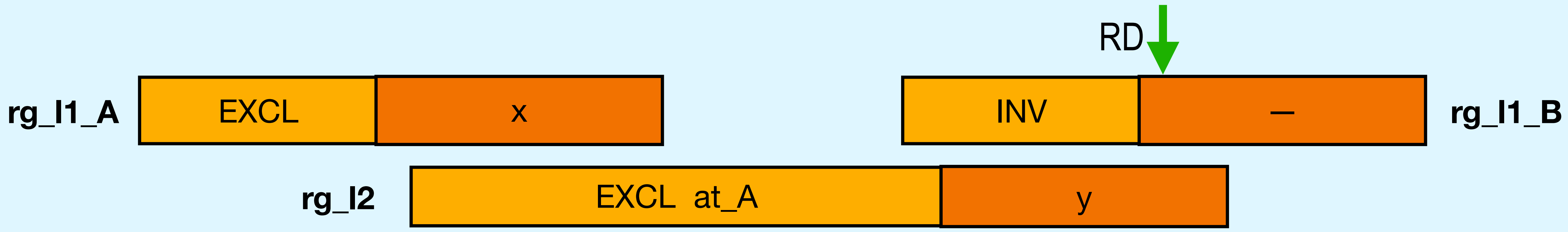
```
rule r1 (      (f_reqs_B.first.req_type == RD)
          &&  (rg_l1_B.cstate == INVALID)
          &&  (rg_l2.cstate == EXCLUSIVE));

   let x = rg_l1_A.data;
   rg_l1_A <= L1_Cell {cstate: SHARED, data: x};
   rg_l1_B <= L1_Cell {cstate: SHARED, data: x};
   rg_l2     <= L2_Cell {cstate: SHARED, at_A: True, at_B: True, data: x};
   f_rsps_B.enq (CPU_Rsp{data:x});
endrule
```

Action

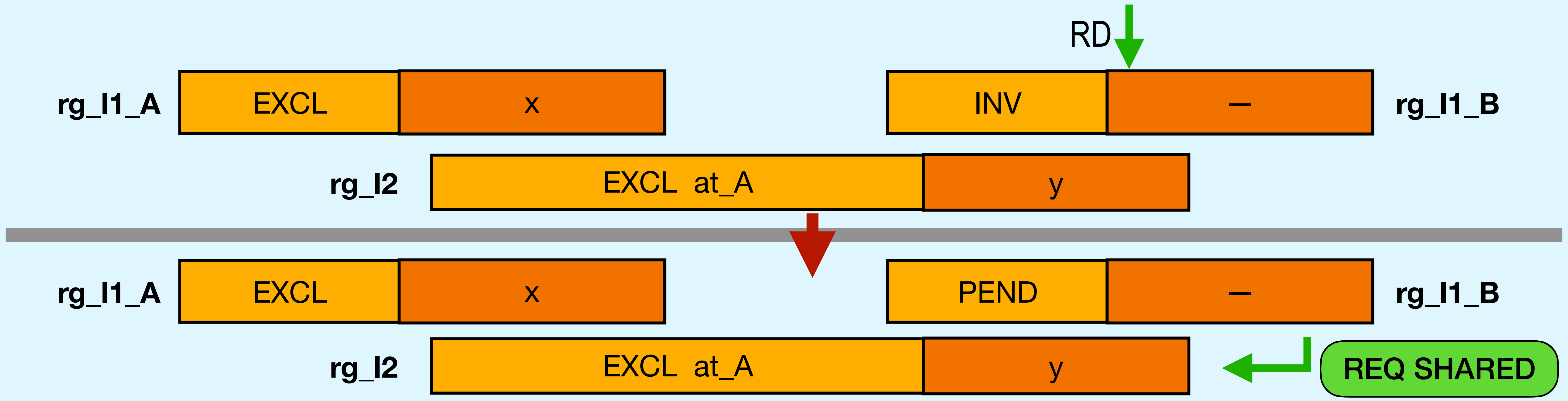
The entire behavior of the cache system can be expressed using such rules.

- *Rules for RD/WR at L1_A and L1_B*
- *Rules express **concurrency***
  - *E.g., simultaneous requests at L1A, L1_B*
  - *Rule choice is non-deterministic*
- *Rules are **atomic***

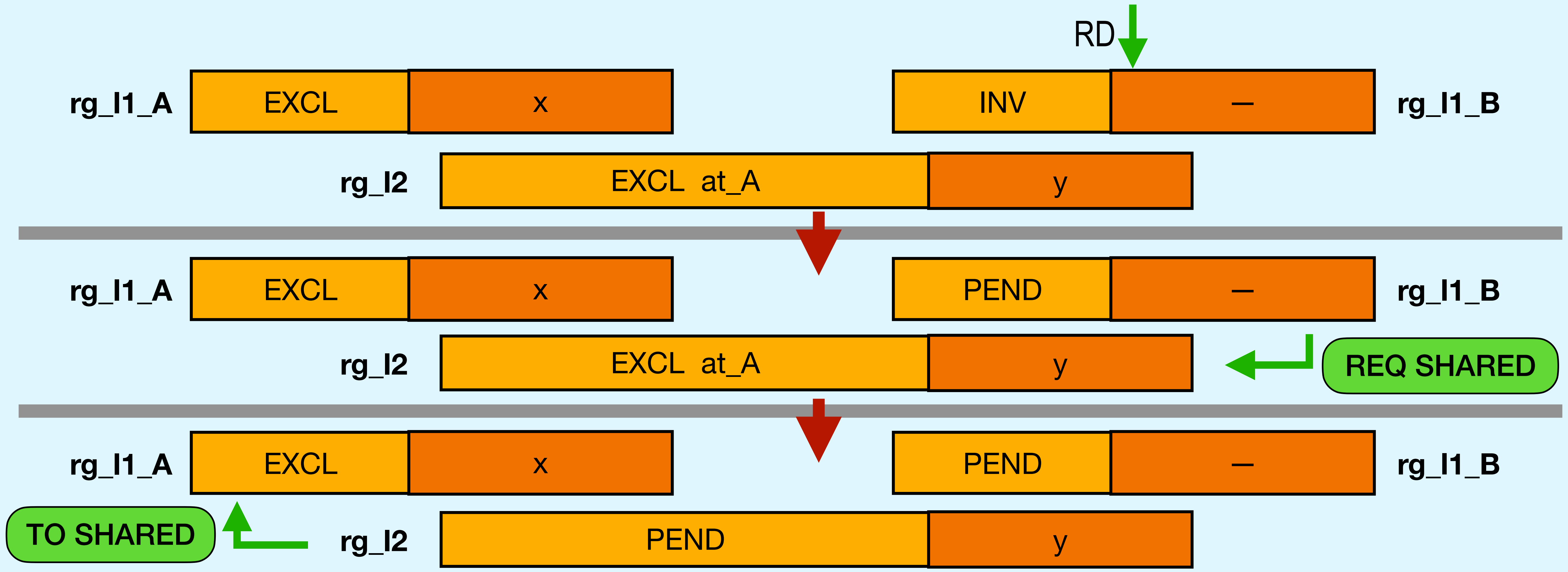
A more implementable spec ("local" rules)



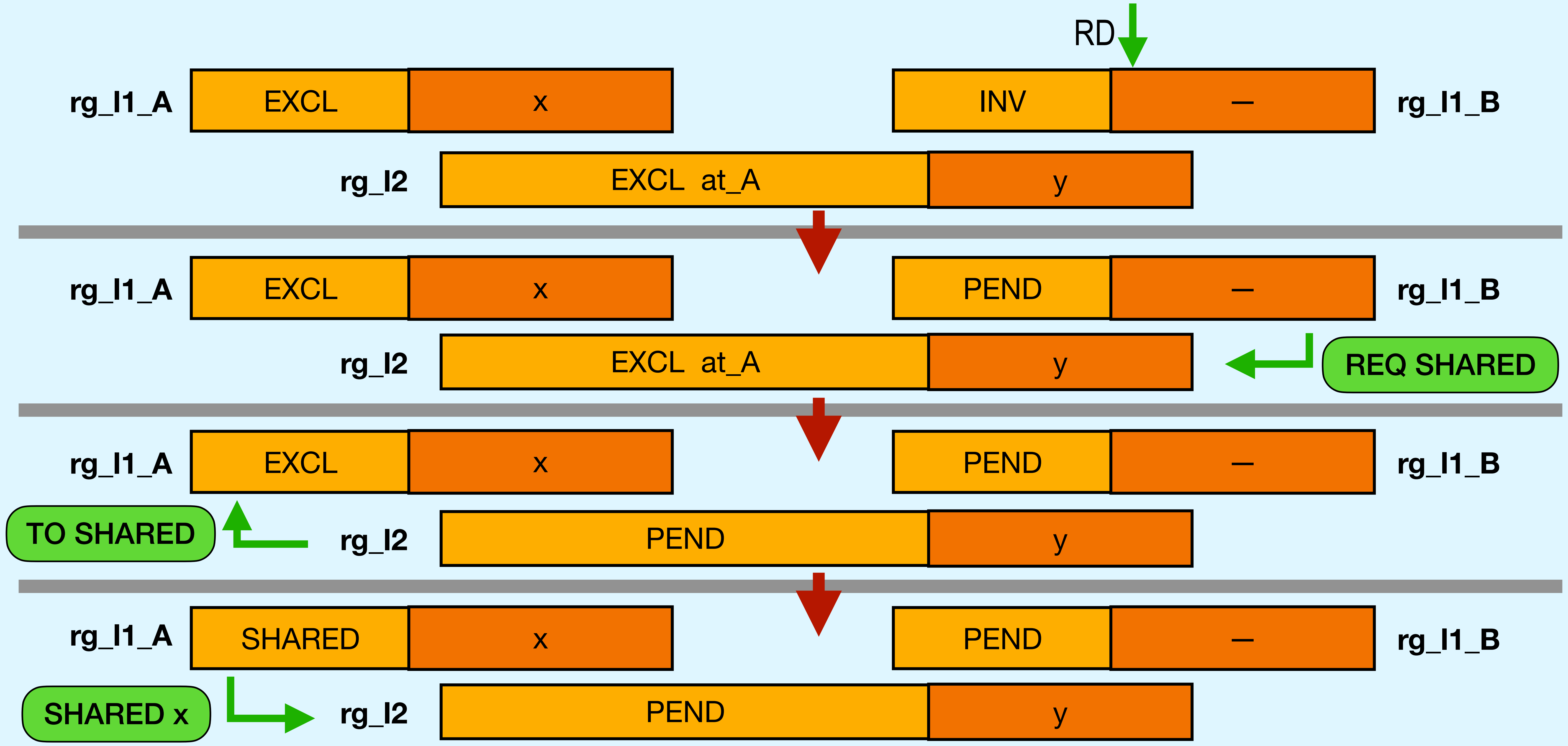
A more implementable spec (“local” rules)



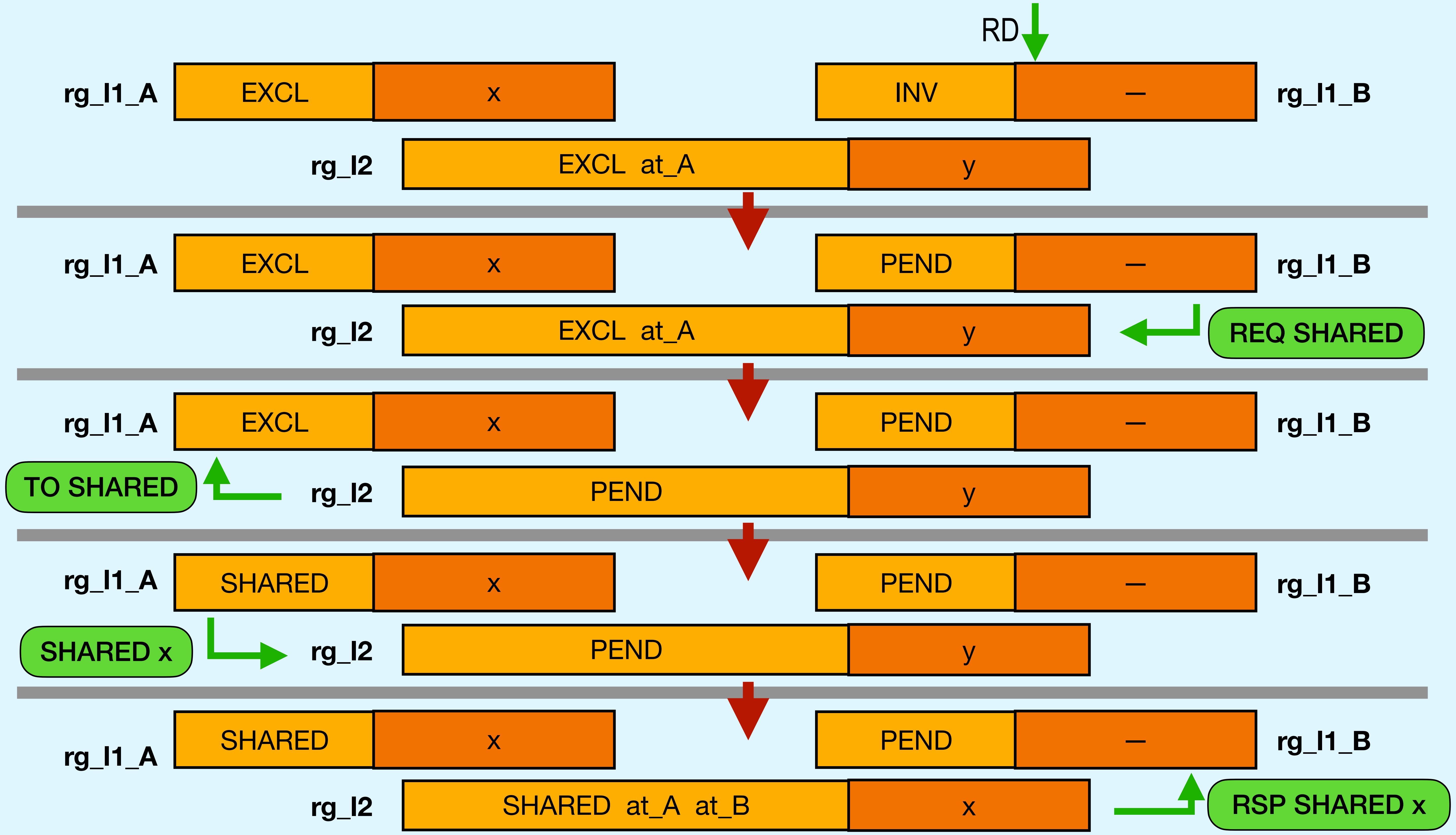
A more implementable spec (“local” rules)



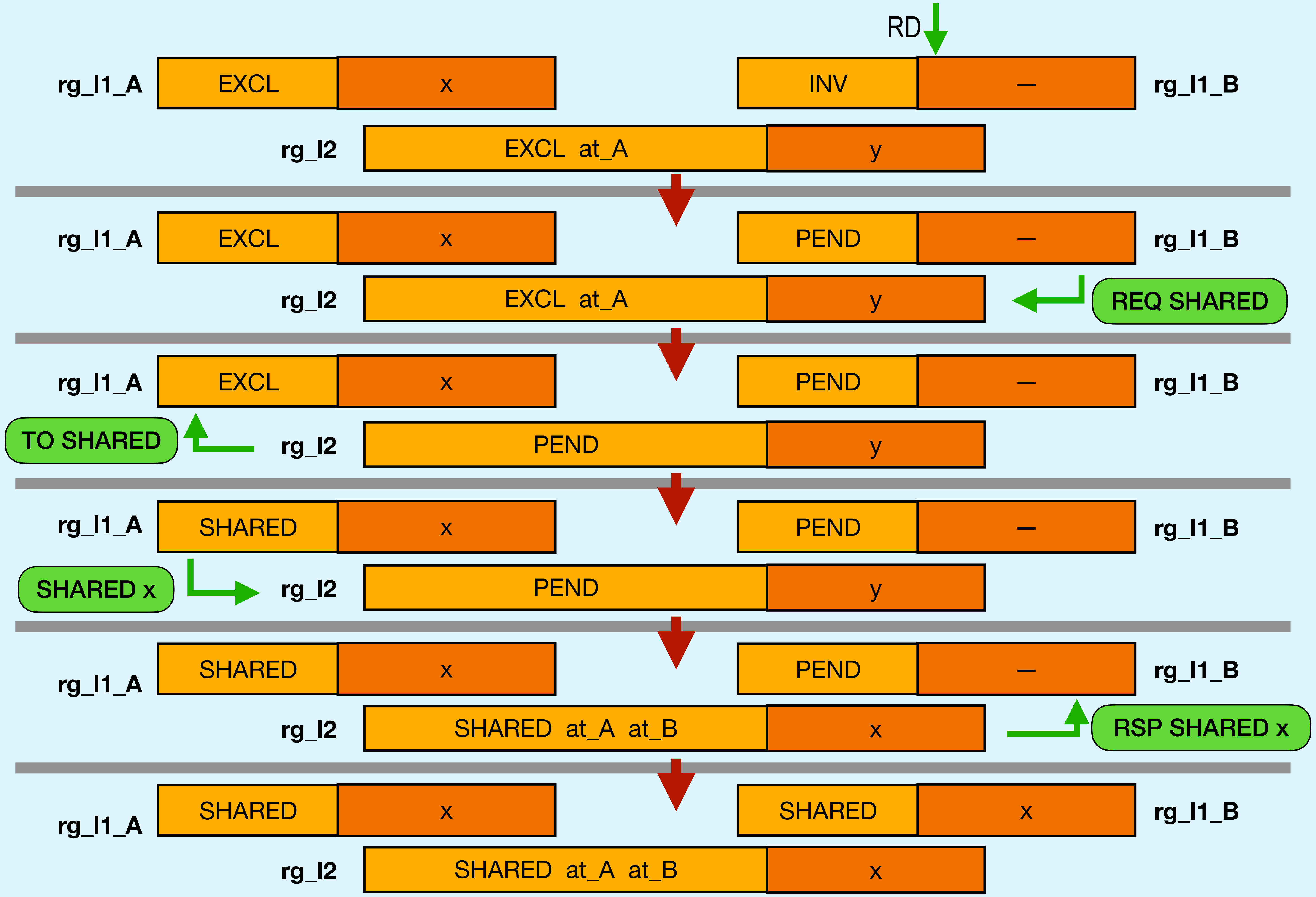
A more implementable spec (“local” rules)



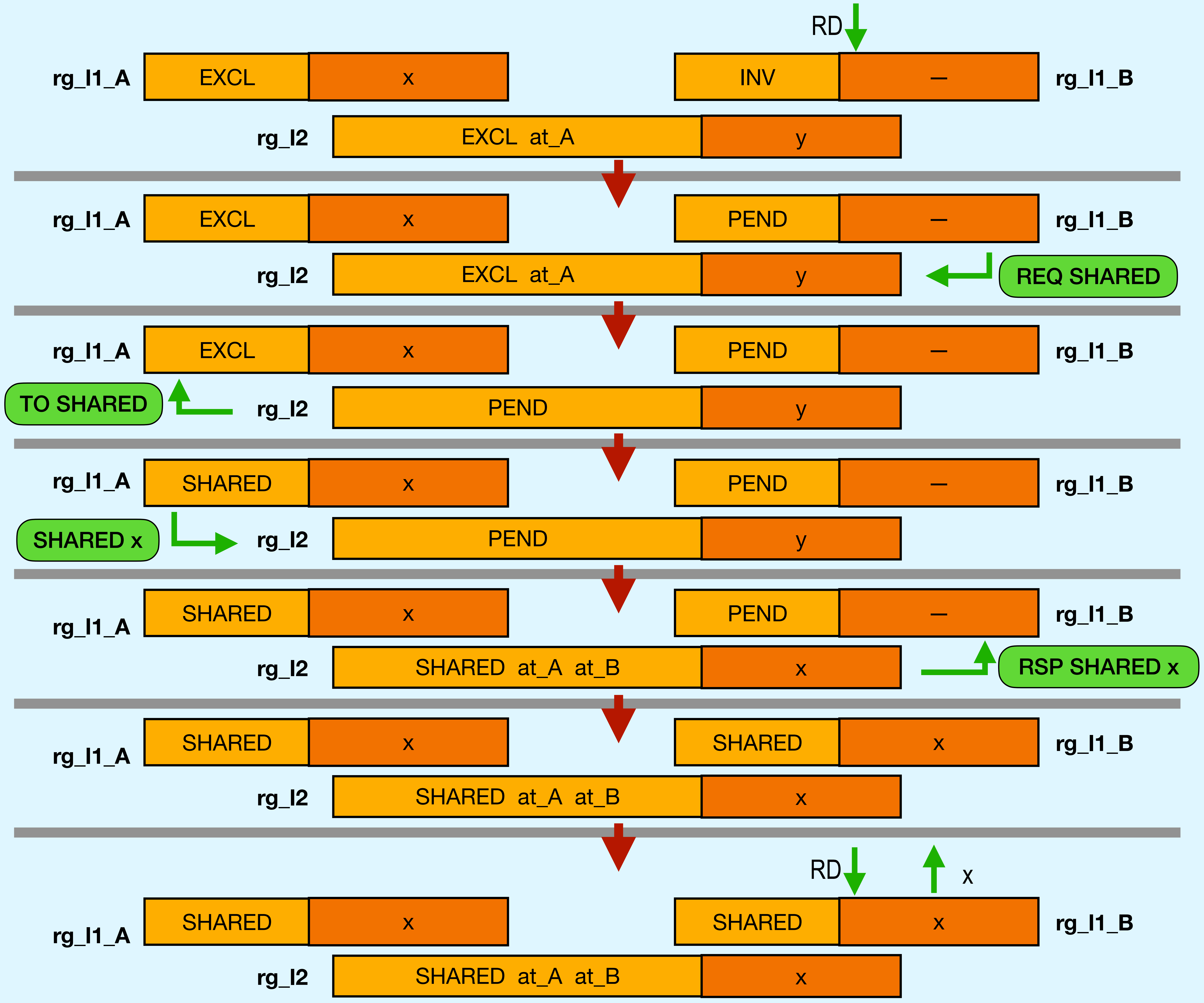
A more implementable spec ("local" rules)



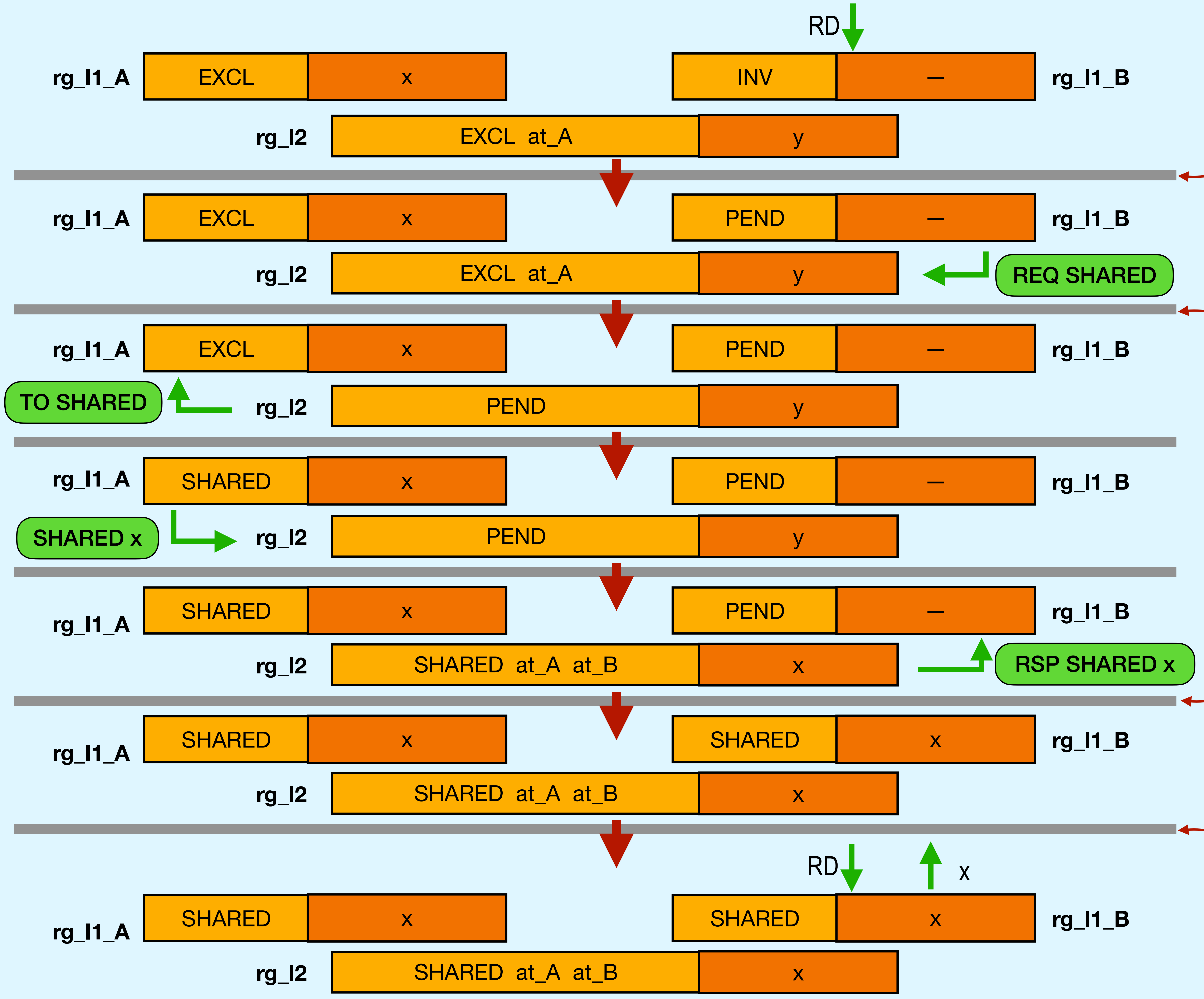
A more implementable spec ("local" rules)



A more implementable spec (“local” rules)



A more implementable spec ("local" rules)




```
rule rl_RD_miss (      (f_cpu_reqs.first.req_type == RD)
                    && (rg_cell.cstate == INV));
   f_l1_to_l2_msgs.enq (L1_to_L2_Msg {msg_type: REQ, to_state: SHARED});
   rg_cell <= L1_Cell {l1_state: PENDING};
endrule
```

```
rule rl_l1_to_l2_msg_S_E (      (f_l1_B_to_l2_msgs.first.to_state == SHARED)
                             && (rg_cell.l2_state == L2_EXCL));
   f_l1_B_to_l2_msgs.deq;
   rg_cell <= L2_Cell {l2_state: L2_PENDING};
   f_l2_to_l1_A_msgs.enq (L2_to_L1_Msg {to_state: SHARED};
endrule
```

```
rule rl_PENDING (      (rg_cell.l1_state == PENDING)
                    && (f_l2_to_l1_msgs.first.msg_type == RSP));
   let l2_to_l1_msg <- pop (f_l2_to_l1_msgs);
   rg_cell <= L1_Cell {cstate: SHARED, data: l2_to_l1_msg.data};
endrule
```

```
rule rl_RD_hit (      (f_cpu_reqs.first.req_type == RD)
                   && (   (rg_cell.cstate == SHARED)
                       || (rg_cell.cstate == EXCL)));
   let cpu_req <- pop (f_cpu_reqs);
   f_cpu_rsps.enq (CPU_Rsp {rdata: rg_cell.data});
endrule
```

Condition-Action Rules for HW specs

a.k.a. Guarded Atomic Actions

- Rules are popular as a spec for concurrent behavior (concurrent state machines)
  - See: ***TLA+, Event-B, UNITY, Term Rewriting Systems, …***
  - ***Atomicity*** of rules enables formal reasoning about invariant-preservation
  - ***Concurrency*** and ***non-deterministic choice*** in rule execution allows *families* of behaviors

- The same language of rules can be used at many levels of abstraction, from abstract specs to implementation-level specs
  - Can *refine* from abstract spec to implementation spec
    - Refine a submodule through several levels (“vertical refinement”)
    - Refine separate submodules independently (“horizontal refinement”)
  - Atomicity is good for proofs that an Implementation Spec is correct w.r.t. an Abstract Spec
  - Refinements and proofs can be manual or automated (“correct by construction”)

BSV and BH

Key points today

1. “Condition-Action Rules” for formal and synthesizable spec of hardware systems
    - a.k.a. “Guarded Atomic Actions”
    - From abstract specs to implementation-level specs
2. *BSV and BH: High-Level Hardware Development Languages (HLHDLs) based on this idea*

BSV/BH Rules are synthesizable to Verilog

- The code shown on previous slides is *actual code* in BSV
  - *All* behavior in BSV/BH is expressed using Rules (no RTL-like “processes”)
  - The ***bsc*** compiler compiles rules to synthesizable, clocked Verilog RTL (preserving atomicity)
  - That RTL can go into ASIC/FPGA flows and can be mixed with hand-written RTL

- Two syntaxes (choose according to your preference, or mix and match):
  - BSV: “SystemVerilog-ish” syntax
  - BH: “Haskell-ish” syntax

- Very expressive, stable, robust, mature, scalable
  - 20+ years in the making … “Batteries Included”—extensive IP libs, open-source resources
  - Has been used for many ASIC and FPGA designs
    - Superscalar, out-of-order, speculative RISC-V CPUs; non-blocking coherent caches; security enhancements for RISC-V CPUs; RISC-V SoCs; TPUs; Video/Image processing IPs; NICs; …

- ***bsc*** compiler has been FOSS since January 2020
  - https://github.com/B-Lang-org/bsc

EXTRAS

Structural spec in BSV/BH has full power of Haskell

E.g., a "Butterfly" switch coded in BSV

```
module mkXBar #(Integer n,
                function UInt #(32) destinationOf (t x),
                Module #(Merge2x1 #(t)) mkMerge2x1)        ( XBar #(t) );
   List#(Put#(t)) iports; List#(Get#(t)) oports;
   if (n == 1) begin
      FIFO#(t) f <- mkFIFO;
      iports = cons(toPut(f),Nil); oports = cons(toGet(f),Nil);
   end
   else begin
      XBar#(t) upper <- mkXBar(n/2, destinationOf, mkMerge2x1);
      XBar#(t) lower <- mkXBar(n/2, destinationOf, mkMerge2x1);

      List#(Merge2x1) merges <- replicateM (n, mkMerge2x1);

      for (Integer j = 0; j < n; j = j + 1)
         rule route;
            let x <- append (upper.output_ports, lower.output_ports) [j].get;
            let jDest = computeRoute (destinationOf (x), j, n);
            if (jDest == j)
               merges [j]    .iport0.put (x);
            else
               merges [jDest].iport1.put (x);
         endrule
      iports = append (upper.input_ports, lower.input_ports);
      oports = map (oport_of, merges);
   end
   interface input_ports = iports; interface output_ports = oports;
endmodule
```

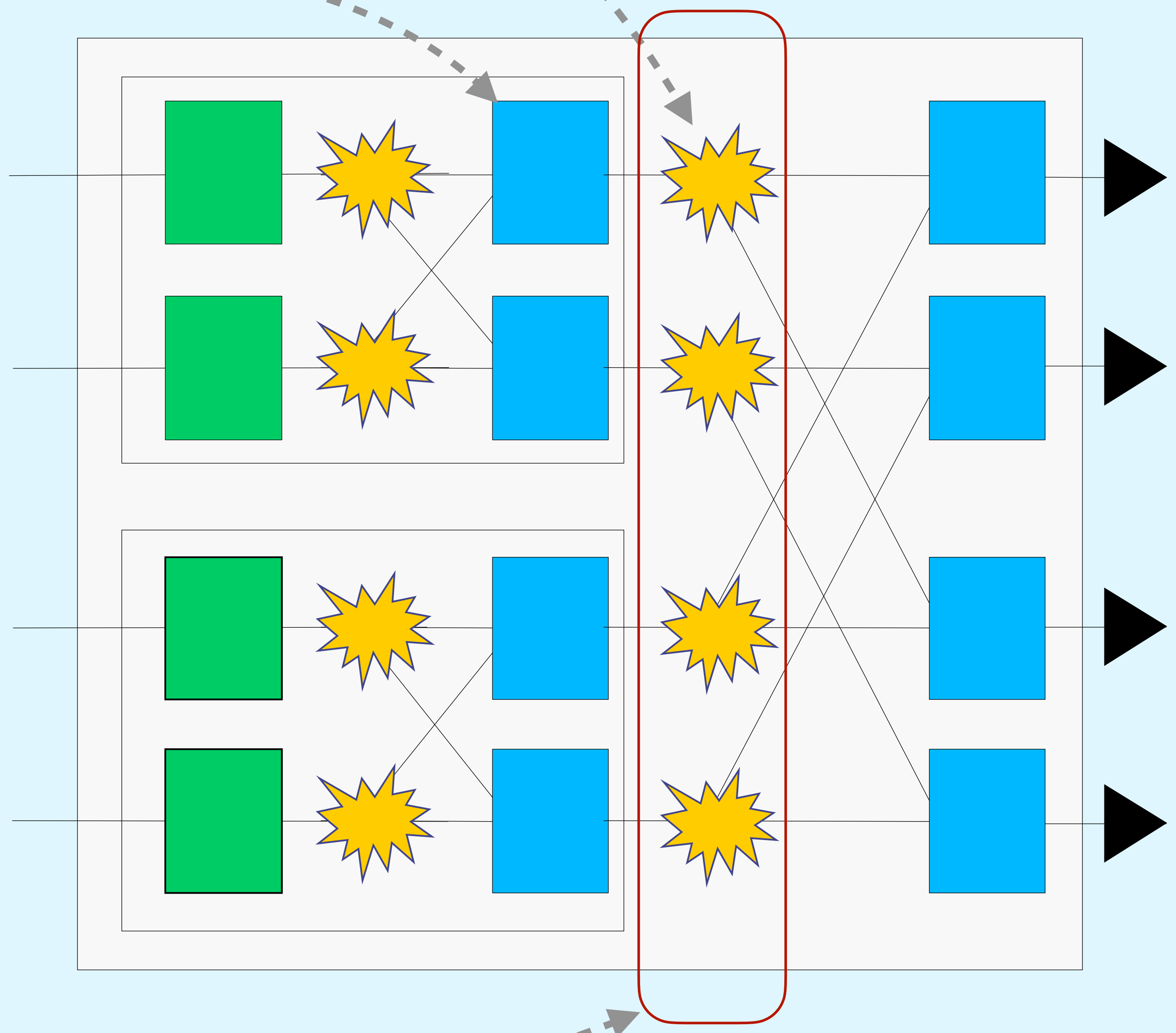
"Routing function" module parameter

"Merge module" module parameter

Interface type; polymorphic type

Recursive module instantiation

Loop to generate rules



- Full Haskell type system
  (polymorphism, typeclasses)
- Monadic types to isolate "pure" expressions
  (e.g., rule and interface method conditions)
- Functions, module, rules are all first-class types