Introduction to PSL for VHDL 200x

Erich Marschner
Senior Architect, Advanced Verification

July 2003

Boolean Logic Review

- If P, P_1, P_2 are predicates, then the following are also:
  - (P)
  - not P
  - P_1 and P_2

  Basic Definitions

  - P_1 or P_2 = (not ((not P_1) and (not P_2)))
  - P_1 -> P_2 = ((not P_1) or P_2))
  - P_1 <-> P_2 = ((P_1 -> P_2) and (P_2 -> P_1))
  - true = (P or (not P)) for any P
  - false = (P and (not P)) for any P

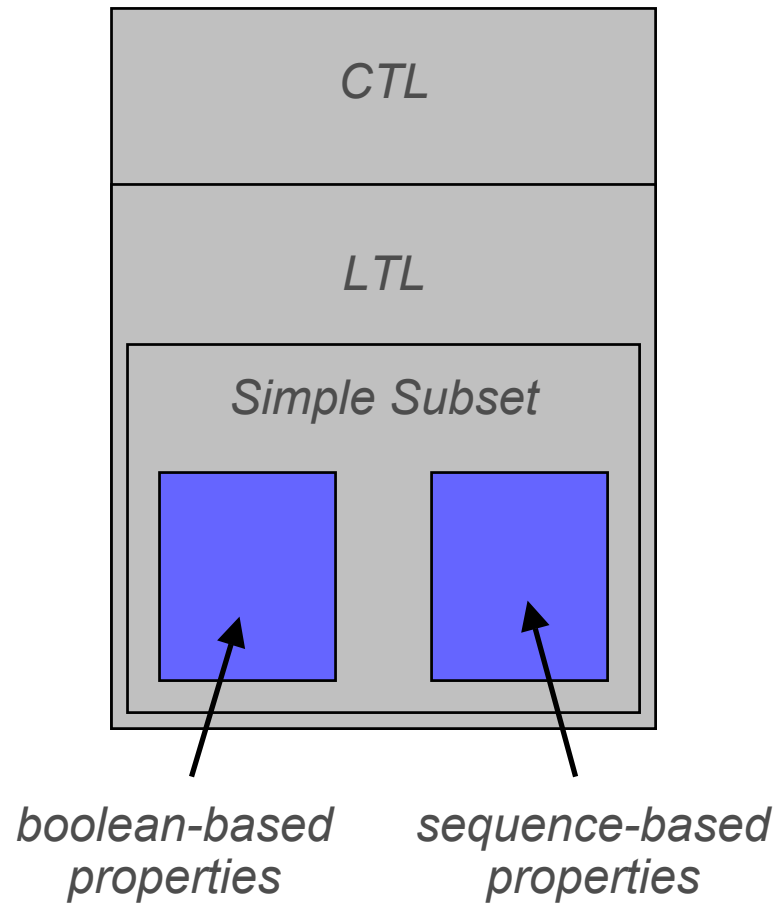
  "Syntactic Sugar"

Temporal Logics

- Computation Tree Logic (CTL)
  - Expresses properties of states
    - e.g., “for all states reachable in one step from this state, ...”
    - leads to branching-time statements such as AGEFp:
      - “for all states globally, there exists a future state in which p holds”
- Linear-Time Temporal Logic (LTL)
  - Expresses properties of paths, or sequences of states
    - e.g., “at every state along the path, ...”
    - leads to linear-time statements such as Ap->Xq:
      - “whenever p holds along a given path, q holds in the next state of the path.”

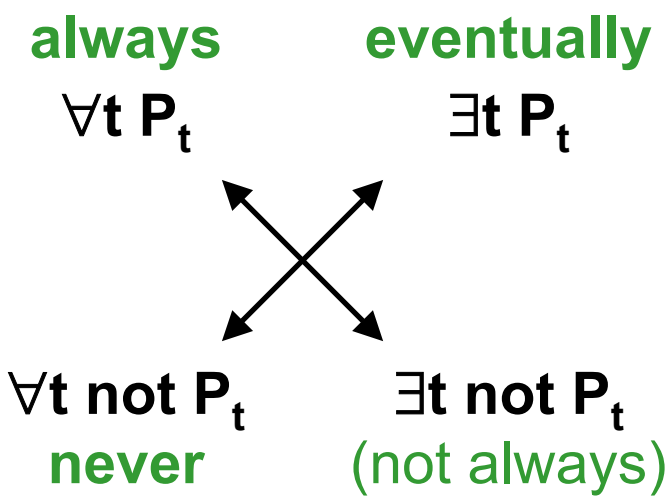
Structure of PSL

- **Linear-Time (LTL) part**
  - for formal verification
  - for simulation
  - Simple Subset
    - LHS of certain binary operators must be boolean
    - easy to implement in simulation
- **Branching-Time (CTL) part**
  - for formal verification



Basic LTL Operators

- P_t = true iff predicate P is true at time t
- **next** P = true iff, at time t, P_{t+1} = true



- **always** P = (P and **next** (P and **next** (P and ...)))
- **eventually** P = (P or **next** (P or **next** (P or ...)))
- **never** P = (not P and **next** (not P and **next** (not P and ...)))

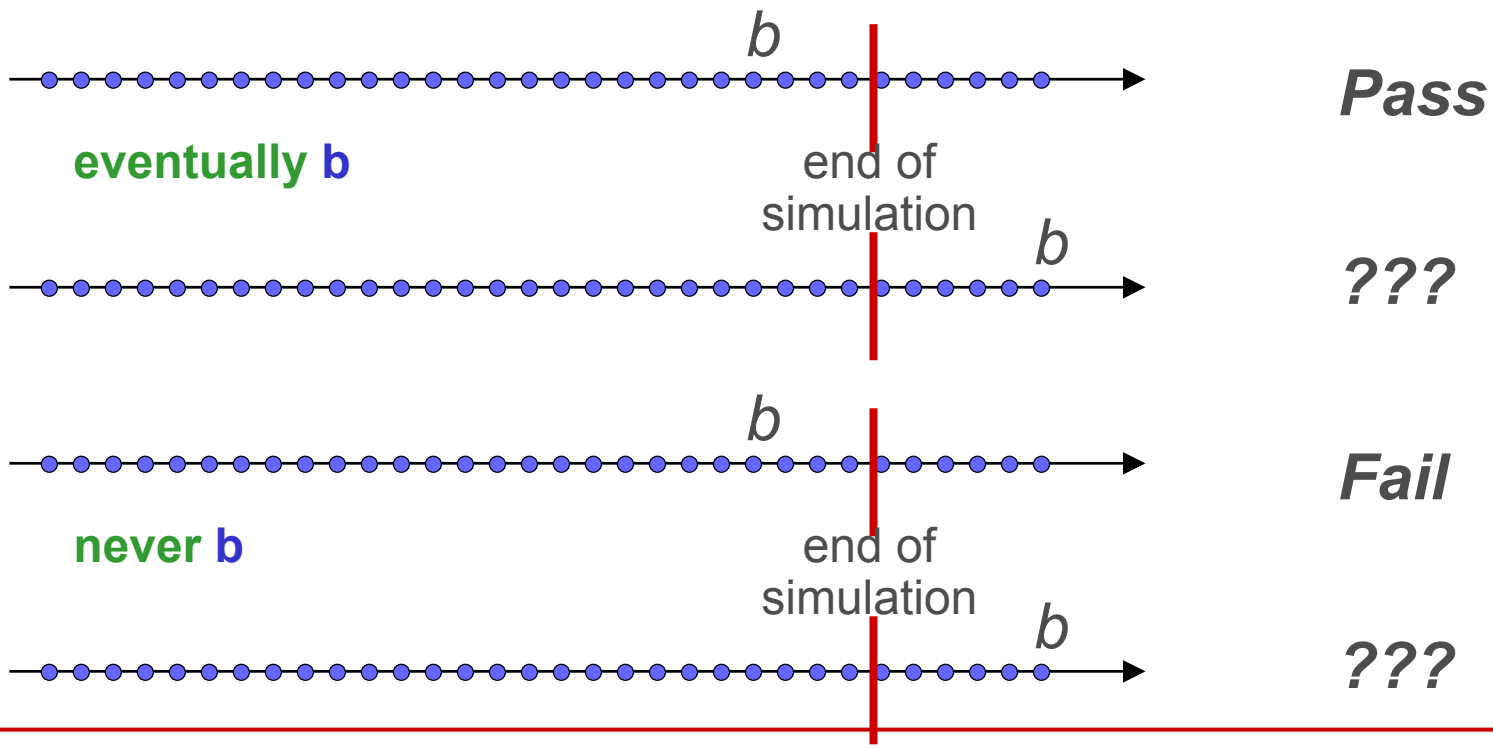
Sequences, SEREs, and Suffix Implication

- **A "Sequence" is a brace-enclosed series of Boolean expressions that are considered in successive timesteps**
  - **{B1; B2; B3; ... }**
- **A "Sequential Regular Expression" (SERE) is an element of a Sequence:**
  - **B B[*] B[*n] B[*n:m] B[+]**
  - **B[=0] B[=n] B[=n:m] B[->] B[->n] B[->n:m]**
  - **r1 ; r2 {r1} : {r2} {r1} | {r2} {r1} & {r2} {r1} && {r2}**

- General use of 'next'
  - **B1 → next (B2 → next (B3 → ... → (Bn-2 and next (Bn-1 and next (Bn)))..)))**
- **Equivalent to suffix implication:**
  - **{B1; B2; B3; ...} |=> {...; Bn-2; Bn-1; Bn}**

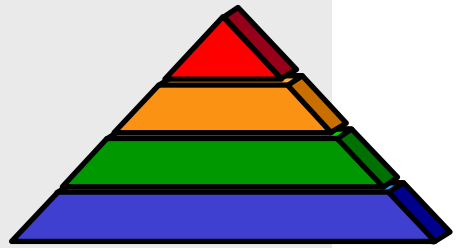
Reasoning over Finite Traces

- LTL is usually applied to infinite traces.
- Simulation deals with finite traces.
- How should we interpret temporal operators in simulation?

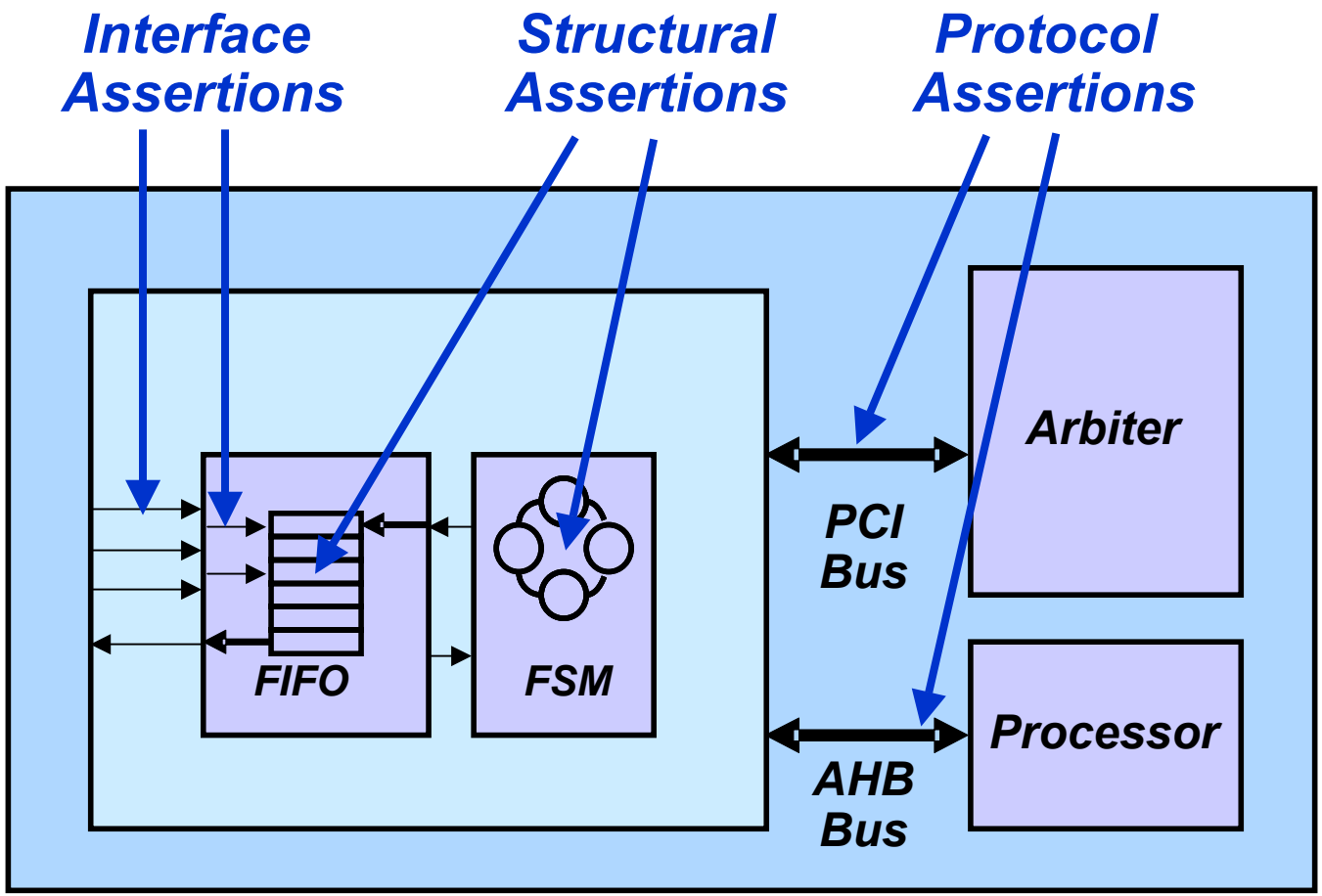


Overview of PSL

- Boolean Expressions
  - **HDL expressions**
  - PSL/Sugar functions **rose()**, **fell()**, **prev()**, ...
- Temporal Operators
  - **always**, **never**, **next**, **eventually**, **until**, **before**, **abort**, ...
  - **@ -> <-> ; { } [*] [=] [->] && & | :**
- Verification Directives
  - **assert**, **cover**, ...
- Modeling Constructs
  - **HDL statements** used to model the environment

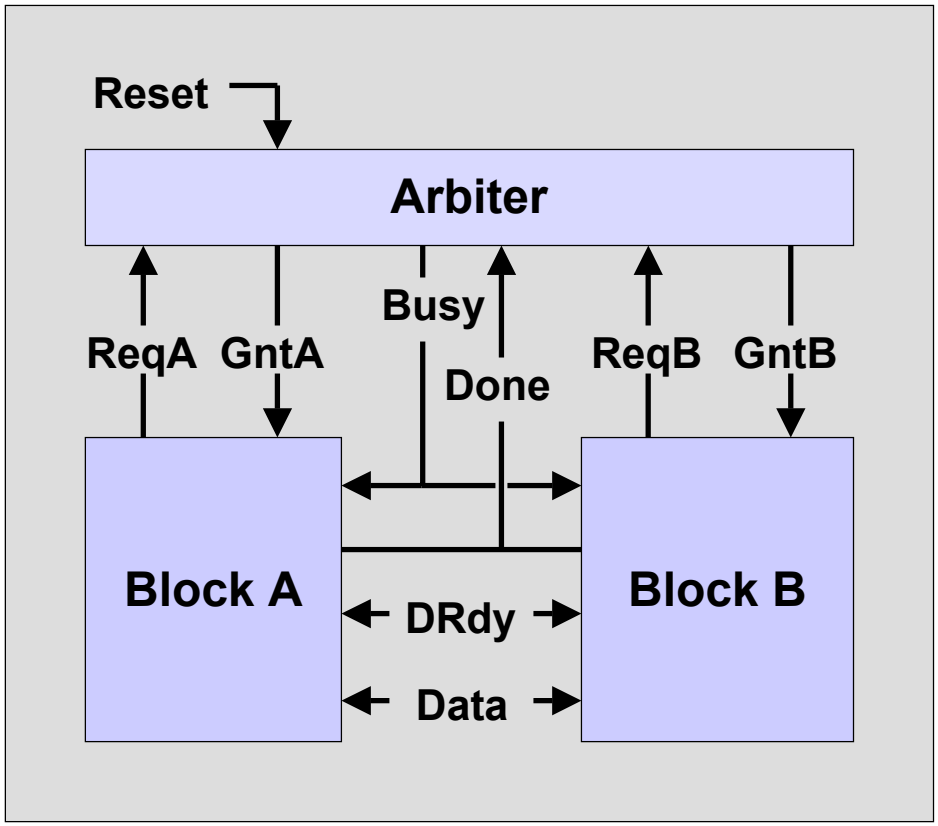


Kinds of Assertions



A Simple Example

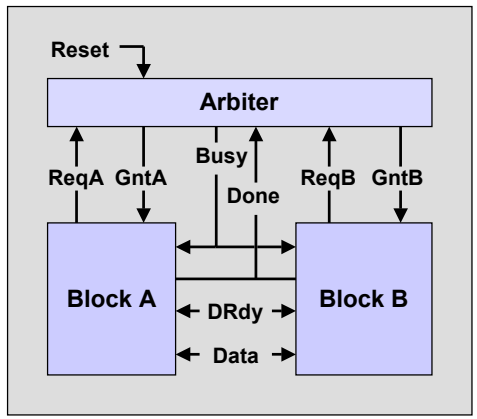
- Two blocks A,B exchange data via a common bus.



- A and/or B sends ‘Req’ to the Arbiter.
- Arbiter does round-robin scheduling between A,B.
- Arbiter sends ‘Gnt’ back to A or B, making it Master.
- Arbiter sets ‘Busy’ while A or B is Master.
- Master sets ‘DRdy’ when Data is on the bus.
- Master sets ‘Done’ in the last cycle of a grant.
- ‘Reset’ resets the bus.

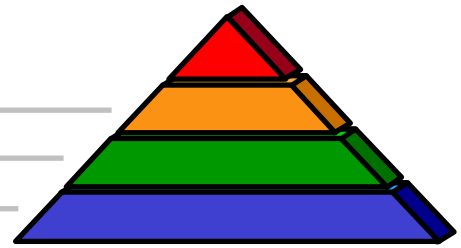
Some Assertions to Check

- A Grant never occurs without a Request.
  - assert never GntA and not ReqA
- If A (B) receives a Grant, then B (A) does not.
  - assert always GntA -> not GntB
- A (B) never receives a Grant in two successive cycles.
  - assert never GntA and next GntA
- A Grant is always followed by Busy.
  - assert always GntA or GntB -> next Busy
- A Request is eventually followed by a Grant.
  - assert always ReqA -> eventually GntA



Invariants

A Grant never occurs without a Request.

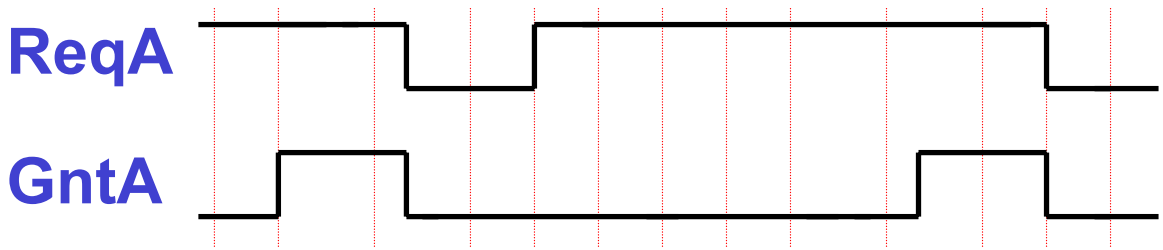


Verification Directive — **assert**

Temporal Operator — **never**

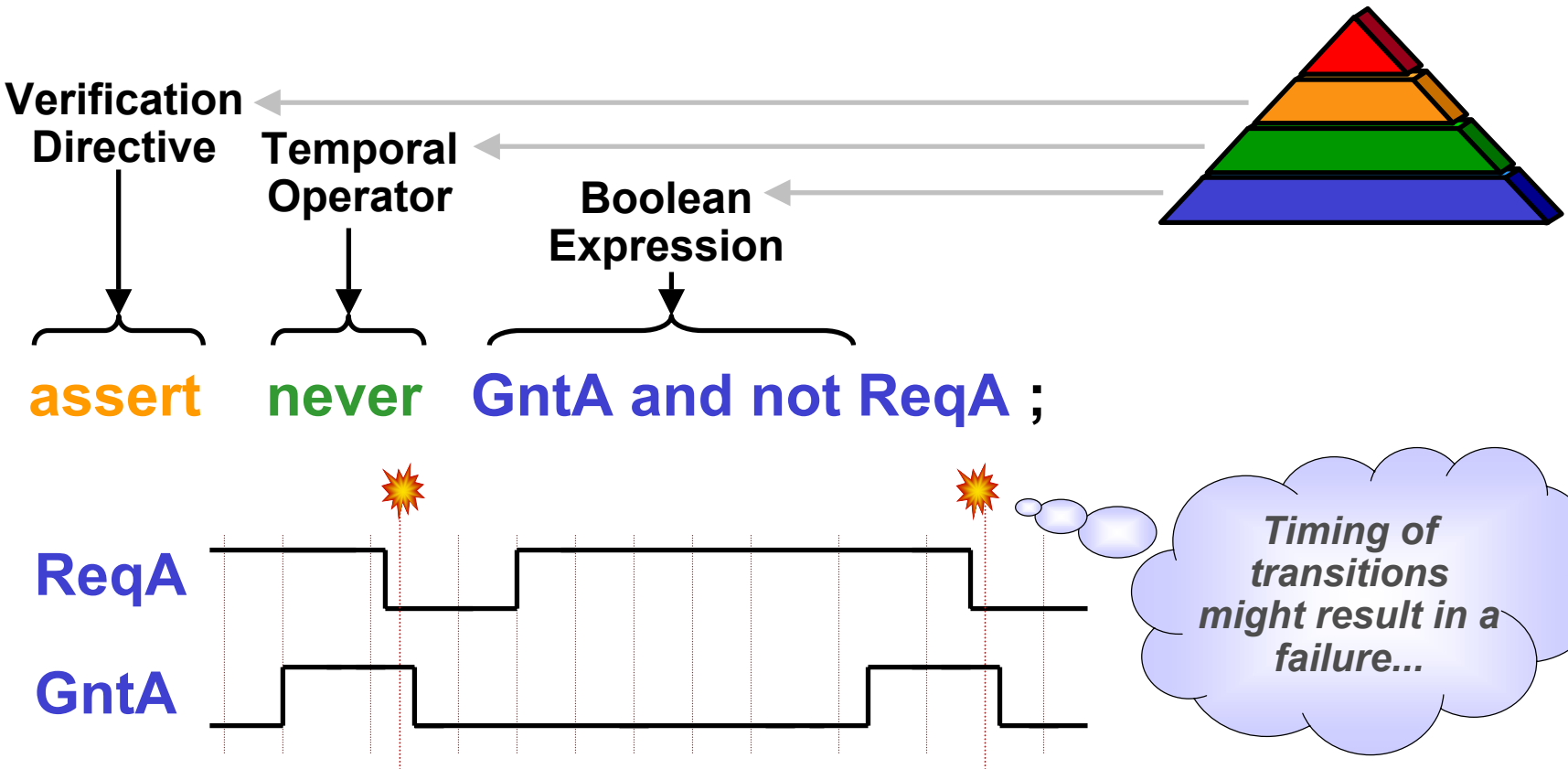
Boolean Expression — **GntA and not ReqA ;**

This assertion should hold at every time step



Unclocked Invariants

A Grant never occurs without a Request.



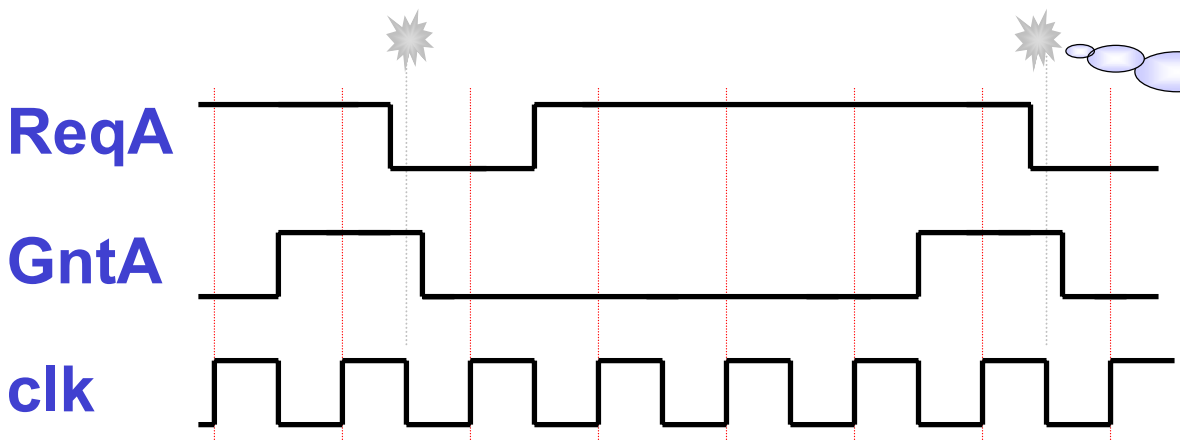
Clocked Invariants

A Grant never occurs without a Request.

... but assertions can be clocked ...

Verification Directive | Temporal Operator | Boolean Expression | Clock Expression

```
assert never GntA and not ReqA @rose(clk) ;
```



... which causes them to ignore glitches

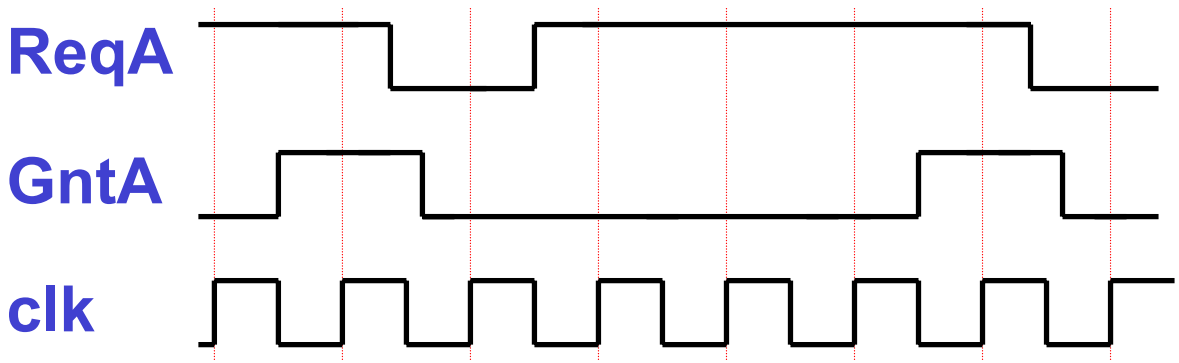
Clocked Invariants

A Grant never occurs without a Request.

Clock can be level-sensitive or edge-sensitive

Verification Directive — **Temporal Operator** — **Boolean Expression** — **Clock Expression**

```
assert never GntA and not ReqA @rose(clk) ;
```



@clk
@(not clk)

@rose(clk)
@fell(clk)

@rising(clk)
@falling(clk)

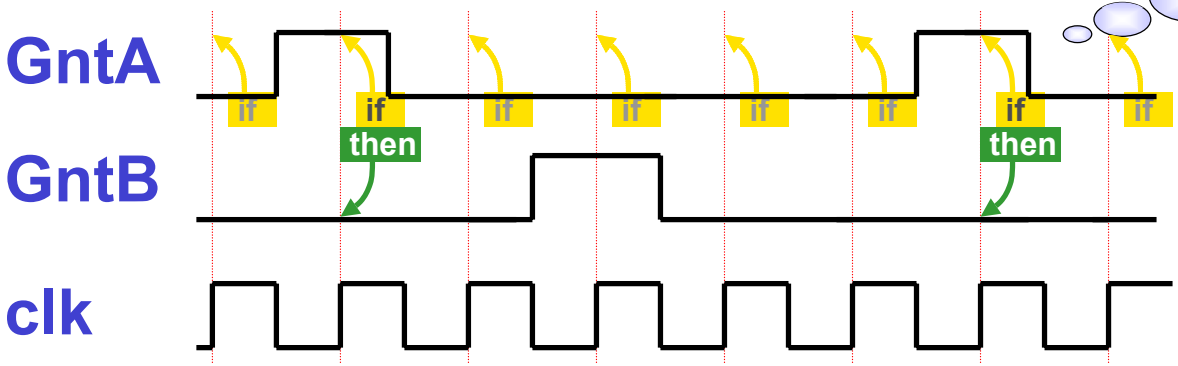
Conditional Behavior

If A receives a Grant, then B does not.

```
assert always (GntA -> not GntB) @rose(clk) ;
```

Implication (->) expresses "if...then"

At the rising clk, if GntA is high, then GntB must be low



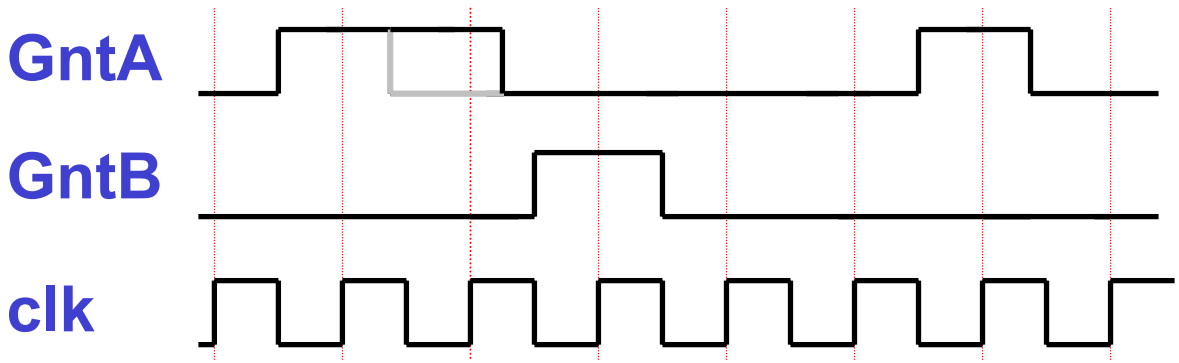
Multi-Cycle Behavior

A (B) never receives a Grant in two successive cycles.

```
assert never GntA and next GntA @rose(clk) ;
```

If GntA is high for two cycles, the assertion fails

'next' refers to the next time step (or clock edge)



Multi-Cycle Behavior using Sequences

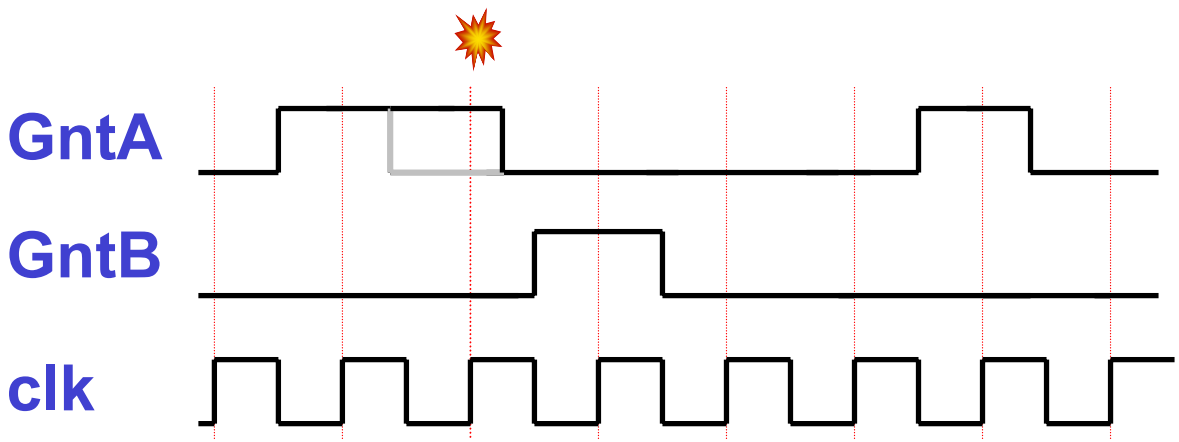
A (B) never receives a Grant in two successive cycles.

```
assert never GntA and next GntA @rose(clk) ;

assert never {GntA ; GntA} @rose(clk) ;
```

Sequence

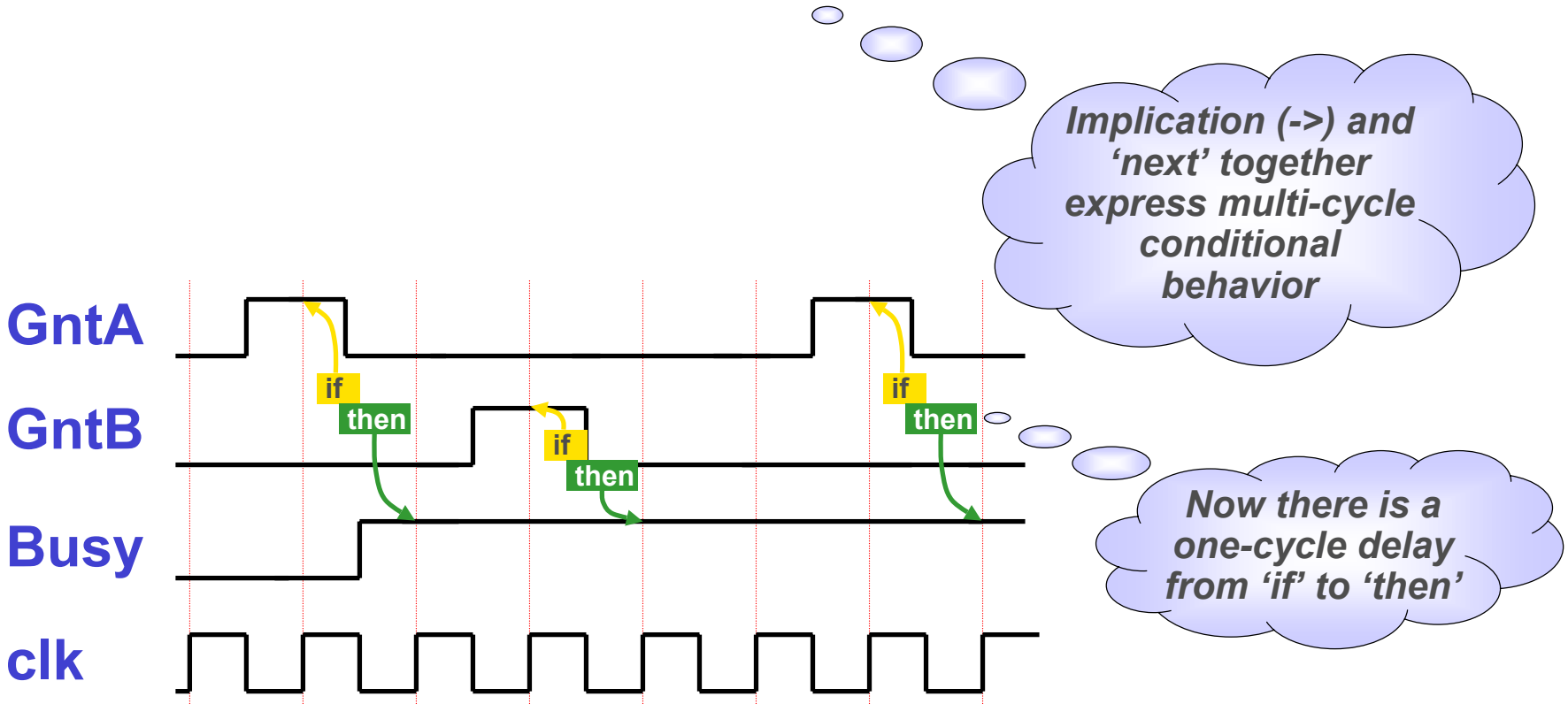
A sequence is a shorthand for a series of 'next's



Multi-Cycle Conditional Behavior

A Grant is always followed by Busy.

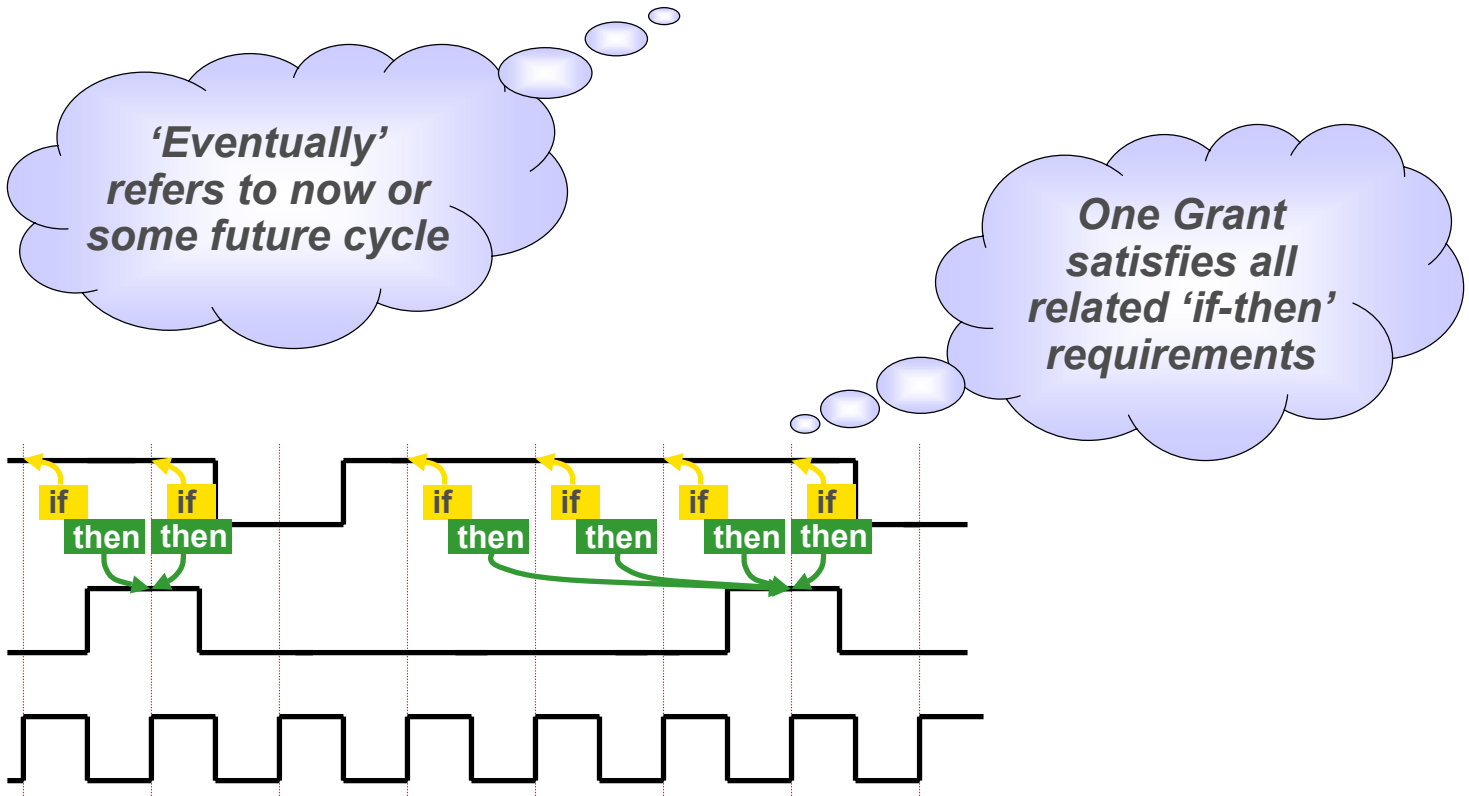
```
assert always GntA or GntB -> next Busy @rose(clk) ;
```



Multi-Cycle Conditional Behavior

- *A Request is eventually followed by a Grant.*

```
assert always ReqA -> eventually! GntA @rose(clk) ;
```



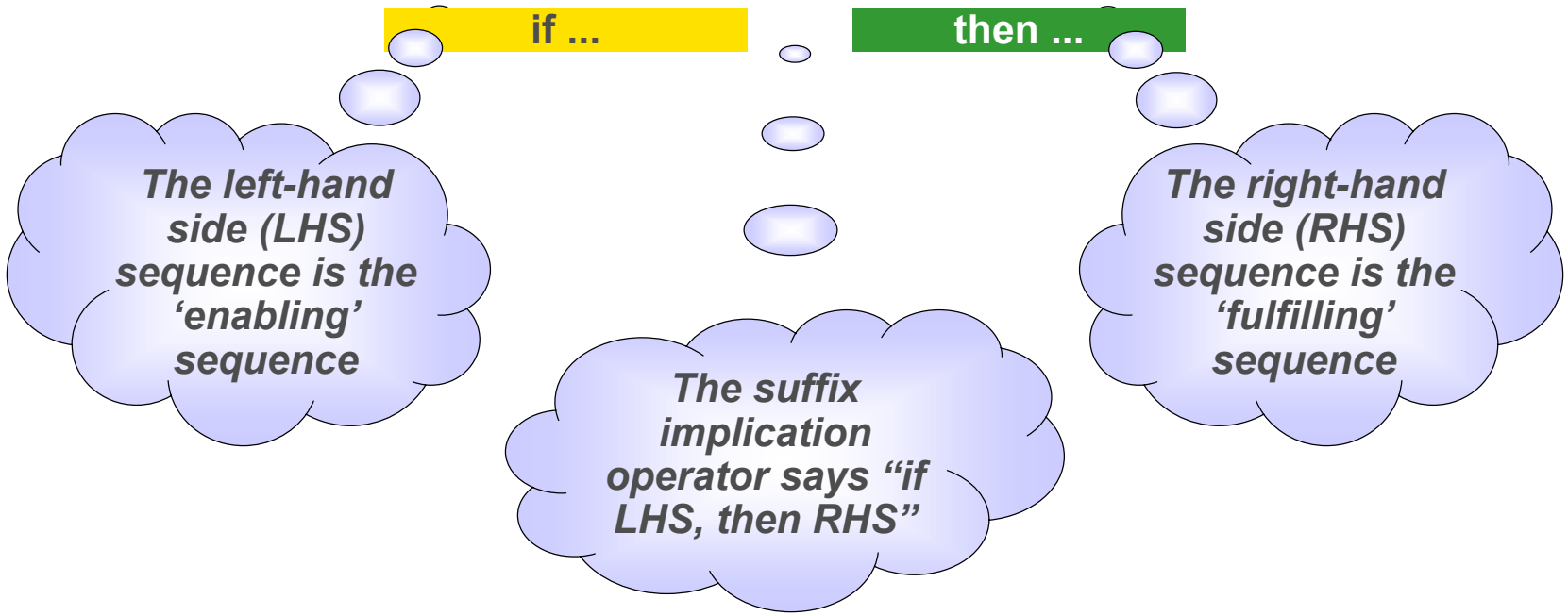
More Assertions to Check

- If Request is followed by Grant, then next is Busy, and next is Done.
  - assert always (ReqA -> next (GntA -> next (Busy and next Done)))
  - assert always {ReqA; GntA} |=> {Busy; Done}
- If Request is followed by Grant, then next Busy is high until Done.
  - assert always (ReqB -> next (GntB -> next (Busy until Done)))
  - assert always {ReqB; GntB} |=> {Busy[*]; Done}
- A Grant is always followed by Busy until, and overlapping with, Done.
  - assert always (GntA or GntB) -> next (Busy until_ Done)
  - assert always {GntA or GntB} |=> {Busy[*]; Busy and Done}
- If A has a Request outstanding when B receives a Grant, then A will receive a Grant before B receives another Grant.
  - assert always (ReqA and GntB) -> next (GntA before GntB)
  - assert always {ReqA and GntB} |=> {[*]; GntA} && {GntB[=0]}

Sequences and Suffix Implication

```
assert always
  (ReqA -> next (GntA -> next (Busy and next Done)))@rose(clk);

assert always ({ReqA; GntA} |=> {Busy; Done})@rose(clk);
```

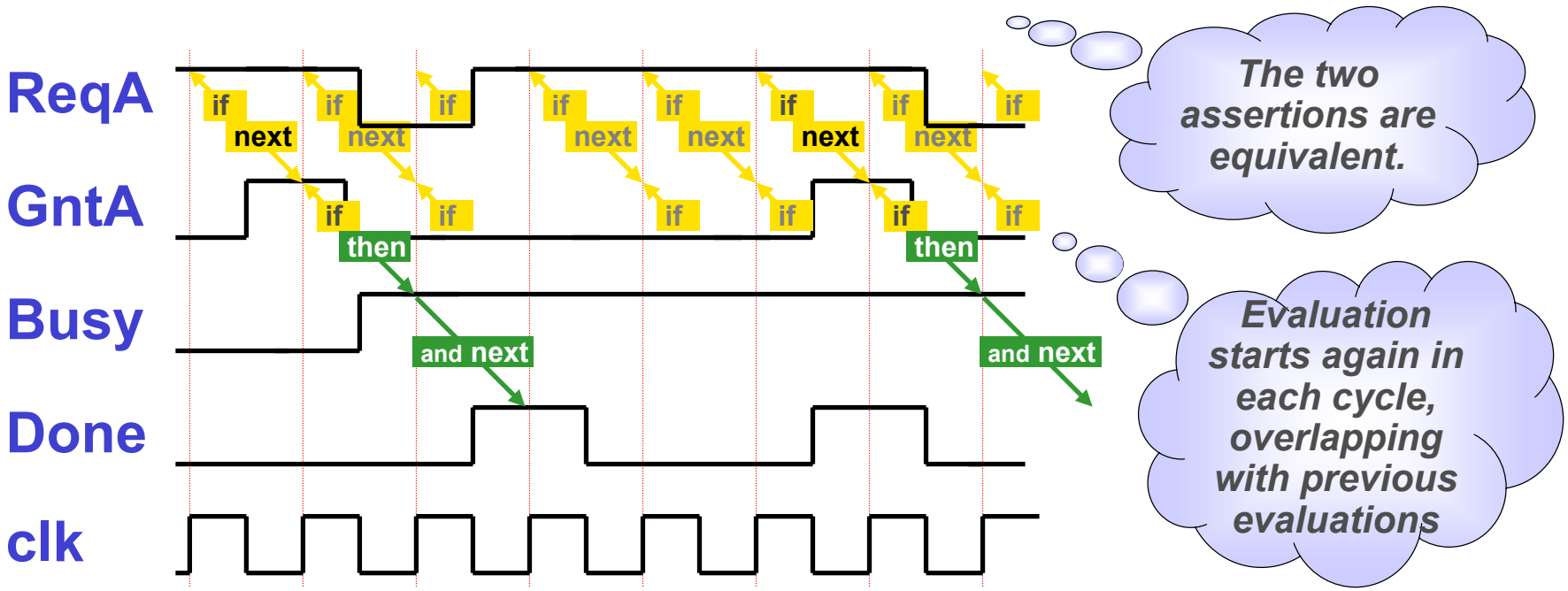


Compound Assertions

If Request is followed by Grant, then next is Busy, and next is Done.

```
assert always
    (ReqA -> next (GntA -> next (Busy and next Done))) @rose(clk) ;

assert always {ReqA; GntA} |=> {Busy; Done} @rose(clk) ;
```

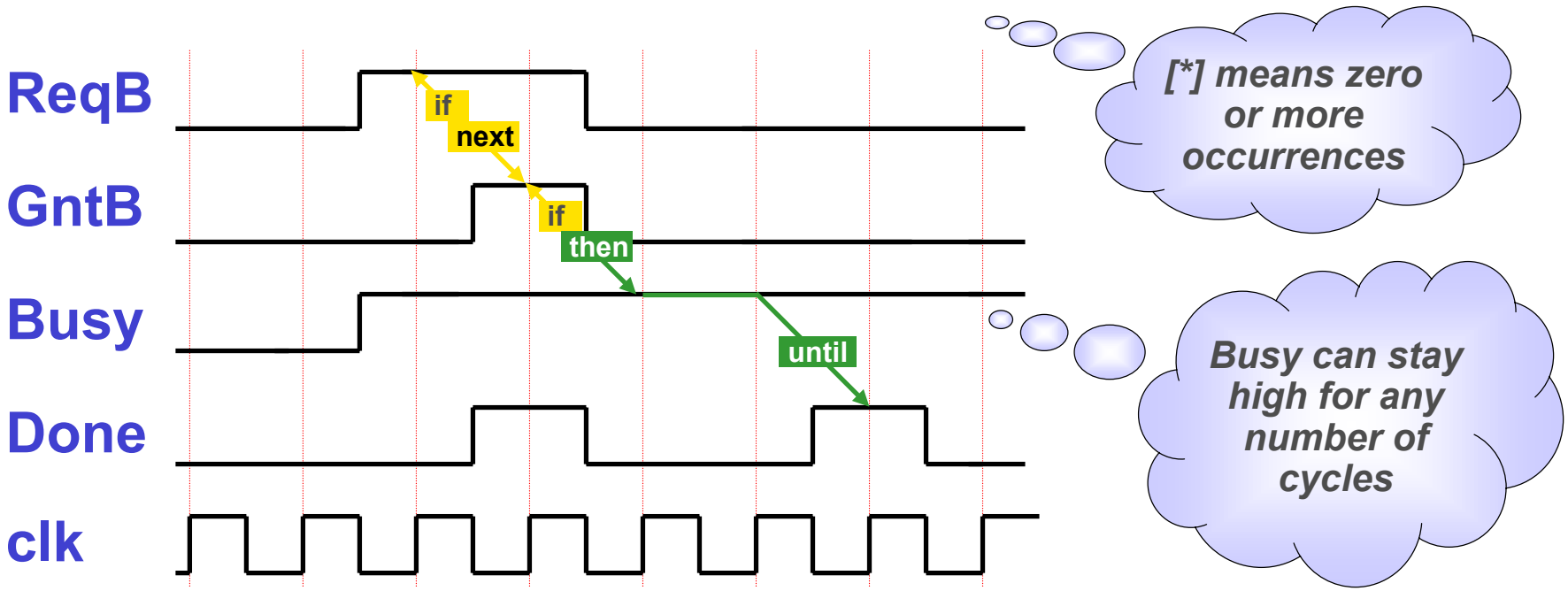


More Precise Specification

If Request is followed by Grant, then next Busy is high until Done.

```
assert always (ReqB -> next (GntB -> next (Busy until Done))) @rose(clk) ;

assert always {ReqB; GntB} |=> {Busy[*]; Done} @rose(clk) ;
```

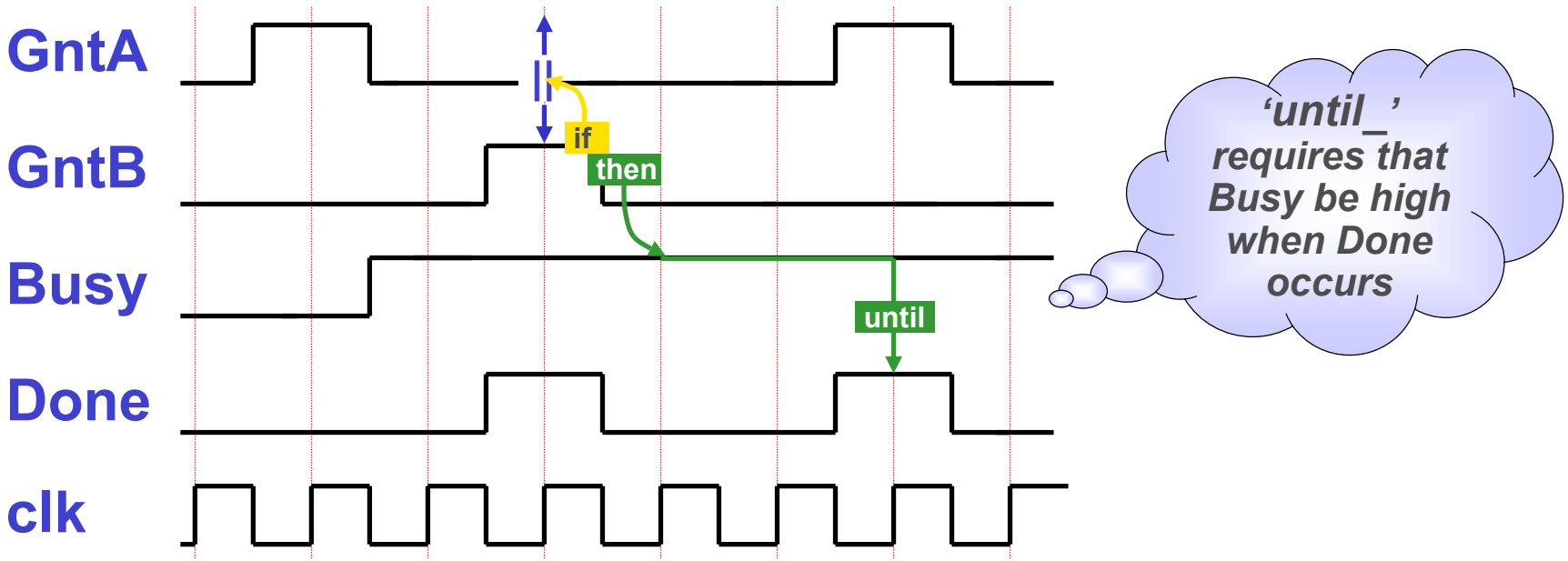


Even More Precise Specification

A Grant is always followed by Busy until, and overlapping with, Done.

```
assert always ((GntA or GntB) -> next (Busy until_ Done)) @rose(clk) ;

assert always {GntA or GntB} |=> {Busy[*]; Busy and Done} @rose(clk) ;
```

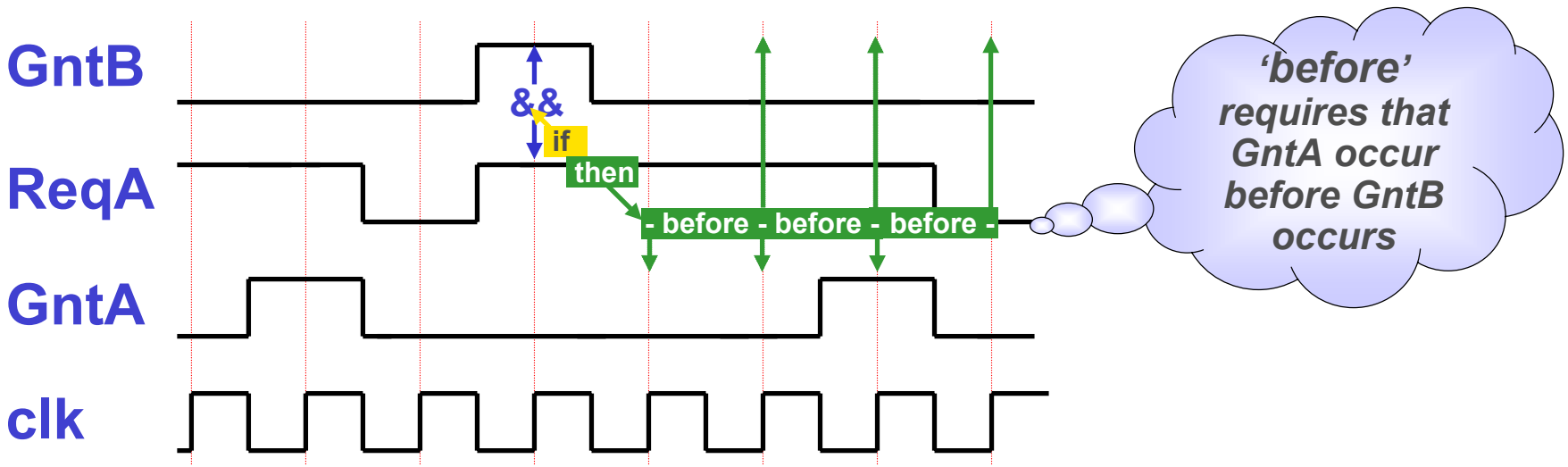


Still More Assertions

If A has a Request outstanding when B receives a Grant, then A will receive a Grant before B receives another Grant.

```
assert always ((ReqA and GntB) -> next (GntA before GntB)) @rose(clk) ;

assert always {ReqA and GntB} |=> {[*]; GntA} && {GntB[=0]} @rose(clk) ;
```



Assertions and Coverage

- **Assertions only catch bugs that occur during verification.**
  - Conditional assertions must be enabled before they can fail.
- **Coverage Monitors ensure that verification is thorough.**
  - Monitors check that all interesting behavior is exercised.
  - Functional coverage is more effective than code coverage.
- **PSL includes both**
  - Assert Directives (assertions)
  - Cover Directives (coverage monitors)

Some Scenarios to Cover

- Test the case in which a transfer includes from 1 to 3 successive data ready cycles.
  - **cover {(GntA or GntB); {Busy[*] && DRdy[*1:3]} : {Done}}**
- Test the case in which a transfer includes exactly 4 data ready cycles (but not necessarily in succession).
  - **cover {(GntA or GntB); {Busy[*] && DRdy[=4]} : {Done}}**
- Test the case in which a transfer completes without having the bus reset.
  - **cover {(GntA or GntB); {Busy[*] && Reset[=0]} : {Done}}**
- Test the case in which a transfer is interrupted by a bus reset.
  - **cover {(GntA or GntB); {Busy[*] && Reset[->1]}}**


cadence