

PSL OVERVIEW

Formal Specifications

- Properties and specifications defined in **natural languages** are **imprecise, long and hard to understand, ...**
 - What does this mean: “every request must remain asserted until a grant is received unless the request is canceled first?”
 - Which signals are meant? What does “until” mean? ...
- **Property Specification Language (PSL)** provides a way to formally specify assertions in a compact and precise way

PSL – The language for Assertions

- Property Specification Language (PSL) is based on the Sugar language from IBM
- PSL has been selected by Accellera as the basis for its standard property specification language
- PSL endorsement and adoption is growing rapidly...

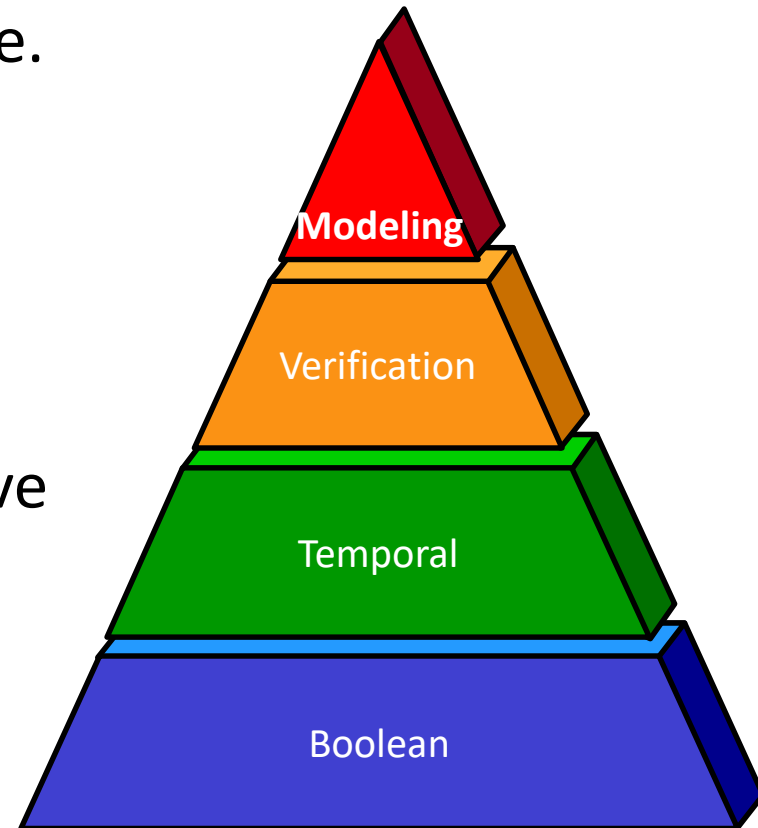


"A Vendor-neutral standard that's open to anyone to use and/or build tools around so customers are not locked to a single vendor"



PSL is a layered language

- Boolean layer
 - Boolean expressions involving HDL, logic operators and PSL functions. It is evaluated immediately, at an instant in time.
 - a) `(ack) (namely, ack == 1)`
 - b) `(out == prev(in, 1))`
- Temporal layer
 - Sequential expressions and properties, both of which describe behavior of the design over time through temporal operators.
 - a) `{s1;s2;s3}`
 - b) `always (req -> next[1](!ack))`



PSL is a layered language

- Verification layer

- Directives for the verification tool that specify what to do with the defined sequences and properties

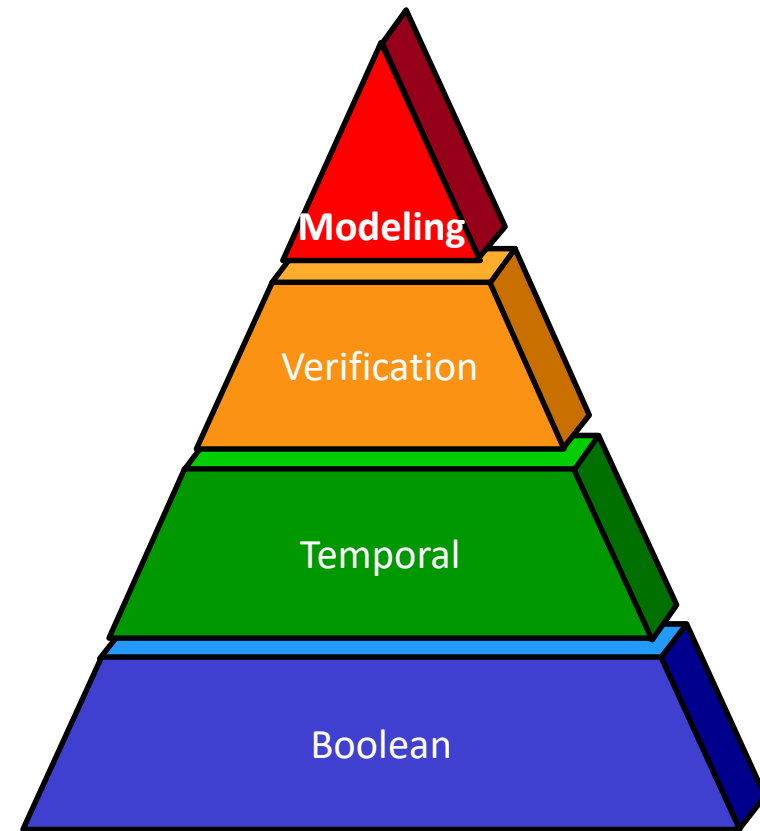
- a) **assert** (p)

- b) **cover** (s)

- Modelling layer

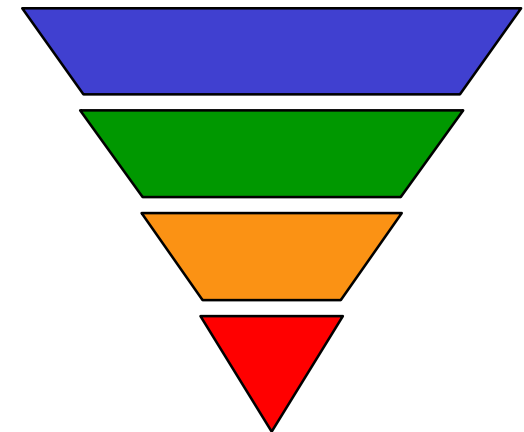
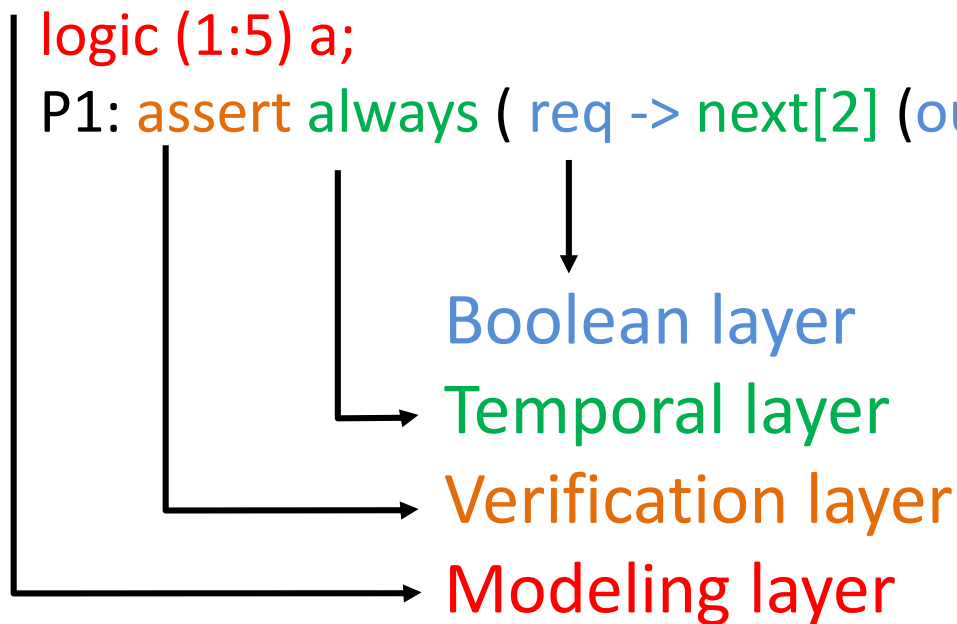
- Directives to declare and give a behavior to design's inputs, auxiliar symbols/variables

(useful to formal verification tools in which the behavior of such elements isn't otherwise specified)



Example

A clock expression determining
when the expression is evaluated



BOOLEAN LAYER

(Boolean_)Expression

- Expression:
 - var_name (the sampled value of the variable named var_name)
 - HDL expression
- Boolean_Expression
 - 1 is **True**
 - 0,x,z is **False**
 - e (!e) (e is an Expression having value 1, 0, x, z)
 - conjunction: e1 && e2 (e1 and e2 Boolean_Expression)
 - disjunction: e1 || e2 (e1 and e2 Boolean_Expression)
 - implication: e1 -> e2 (maps to (!e1 || e2))
 - equality: e1 == e2 (maps to (e1 && e2) || (!e1 && !e2))
 - ...

Built-in functions

- `prev(expr, [i])`: gives the value of `expr` in the i^{th} previous cycle with respect to the clock of its context. The value of i is 1 when not specified.

Examples:

- `prev(a)` gives 1 at times 5 and 7 (clk context)
- `prev(a,2)` gives 1 at time 7 (clk context)

time	0	1	2	3	4	5	6	7
clk	0	1	0	1	0	1	0	1
a	0	0	1	1	0	1	0	0

Built-in functions

- `next(expr)`: gives the value of `expr` at the next cycle with respect to the finest granularity of time.

Example:

- `next(a)` gives 1 at times 1, 2 and 4

time	0	1	2	3	4	5	6	7
clk	0	1	0	1	0	1	0	1
a	0	0	1	1	0	1	0	0

Built-in functions

- `rose(expr)`: takes a Bit expression as argument, and it returns True if the `expr`'s value is 1 at the current cycle and 0 at the previous cycle, with respect to the clock of its context.

Example:

— `rose(a)` gives True at time 3 (clk context)

time	0	1	2	3	4	5	6	7
clk	0	1	0	1	0	1	0	1
a	0	0	1	1	0	1	0	0

Built-in functions

- `fell(expr)`: takes a Bit expression as argument, and it returns True if the `expr`'s value is 0 at the current cycle and 1 at the previous cycle, with respect to the clock of its context.

Example:

– `fell(a)` gives True at time 7 (clk context)

time	0	1	2	3	4	5	6	7
clk	0	1	0	1	0	1	0	1
a	0	0	1	1	0	1	0	0

TEMPORAL LAYER

Property and Sequence (part 1)

- Property = Boolean_Expression
 - | Property **@ clock_expression**
 - | **always** Property
 - | **never** Property
 - | **eventually!** Property
 - | **next[i]** Property
 - | Property **until** Property
 - | Property -> Property
- Sequence = ... (next slides)

clock_expression

- A *clock expression* is a Boolean Expression which defines a clock context. A clock context determines the path on which a Property, Sequence, or Expression is evaluated.

Example:

Evaluating a formula only at the positive edge of clk.

(always (req -> next ack)) @ (**posedge clk**);

- The directive *default clock* defines the default clock context.

Example:

default clock = (posedge clk);

(always (req -> next ack));

Temporal operator

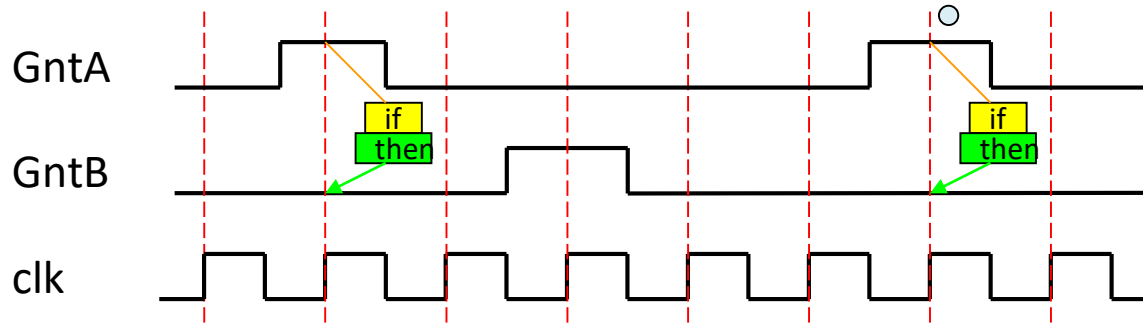
- **always:** the always operator specifies that a *Property* or a *Sequence* holds at all times, starting from the present.
- **never:** the never operator specifies that a *Property* or a *Sequence* never holds.
- **eventually!:** The eventually! operator specifies that a *Property* holds at the current cycle or at some future cycle.

Example

- If GntA is high, then GntB is low:
`always (GntA -> !(GntB)) @ (posedge clk);`

Implication (->) expresses “if... then”

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

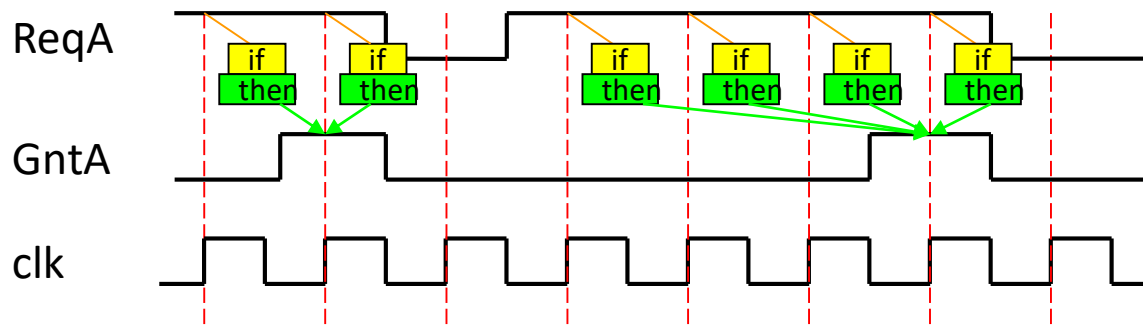


Example

- ReqA is eventually followed by GntA:
`always (ReqA -> eventually! GntA) @ (posedge clk);`

'eventually' refers to now or some future cycle

One Grant satisfies all related 'if-then' requirements



Temporal operator

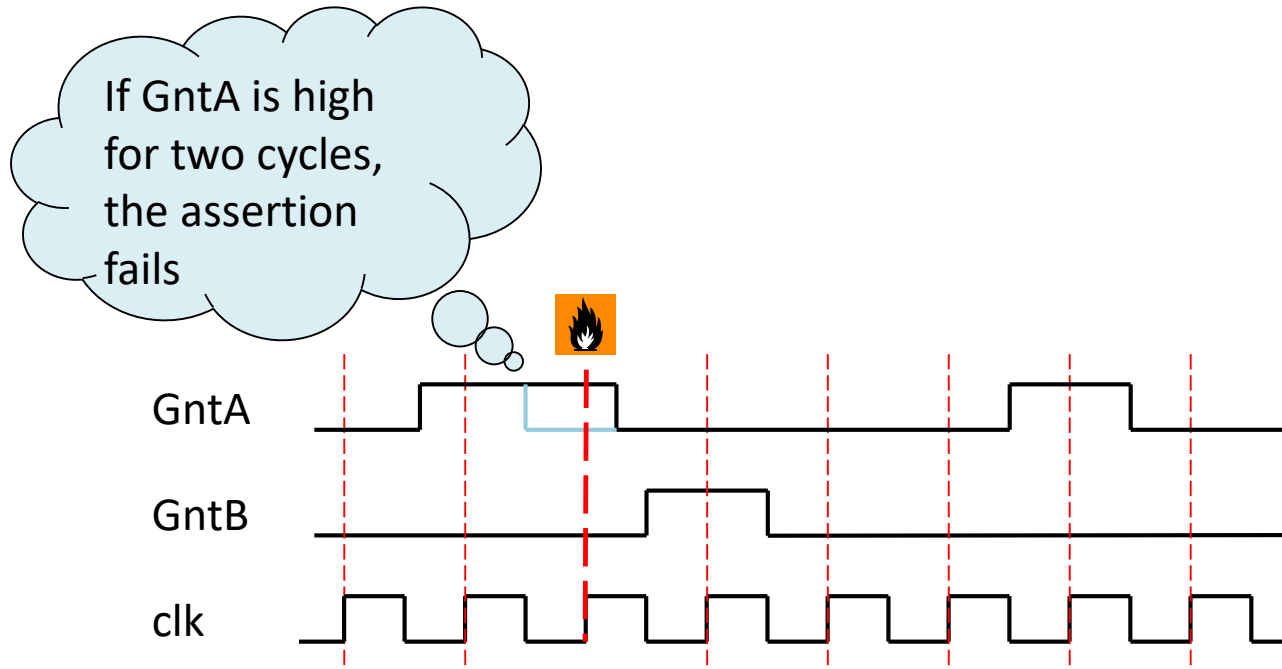
- **next:** the *next* family specify that a *Property* holds at some next cycle.

Property = next! (*Property*)
 | next (*Property*)
 | next![i] (*Property*)
 | next[i] (*Property*)

- If present, the number *i* indicates at which next cycle the property holds.
- next! and next![i], specifies that the i^{th} next cycle must exist!

Example

- GntA is never high for two successive clock cycles
`never (GntA && next[1](GntA)) @ (posedge clk);`

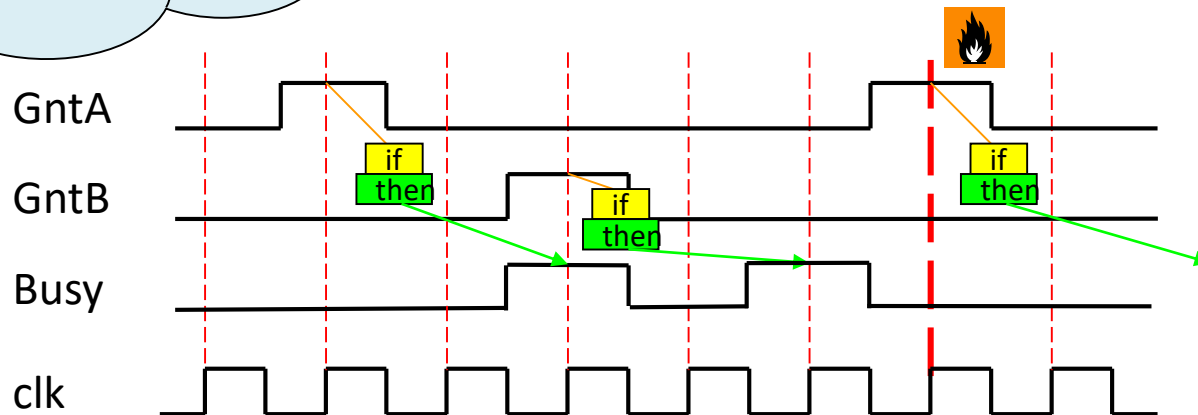


Example

- If GrantA, or GrantB, is high, then Busy is high after two clocks cycles:

```
always ((GntA || GntB) -> next![2](Busy)) @(posedge clk);
```

Implication (->) and
'next' together express
multi-cycle conditional
behavior



Temporal operator

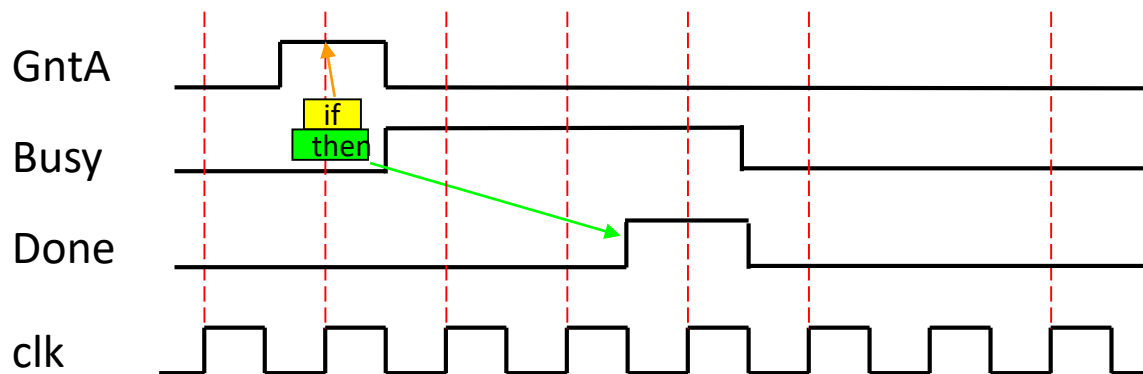
- **until**: the *until* family specify that a one *Property* holds until a second *Property* holds (*terminating property*).

Property = *Property* **until!** (*Property*)
 | *Property* **until!_** (*Property*)
 | *Property* **until** (*Property*)
 | *Property* **until_** (*Property*)

- **until!** and **until!_** operators specify that the *terminating property* eventually holds (otherwise the left operand holds forever).
- **until_** and **until!_** operators specify that that the left operand holds up to and including the cycle in which the right operand holds.

Example

- If GntA is high, then Busy is high after one clocks cycle, and it remains high until (inclusive) Done is high
`always ((GntA) -> next[1](Busy until_ Done)) @(posedge clk);`



Property and Sequence (part 2)

- Property = ... (see previous slides)
 - | Sequence
 - | Sequence \rightarrow Property
 - | Sequence \Rightarrow Property
- SERE = Boolean_Expression
 - | SERE ; SERE
 - | SERE : SERE
 - | {SERE} [**@ clock_expression**]
 - | repeated_SERE
- A Sequence is a SERE!

Sequential Extended Regular Expression (SERE)

- SEREs describe single- or multi-cycle behavior built from a series of Boolean expressions.
- *The concatenation* operator (;) constructs a SERE that is the concatenation of two other SEREs.

Informal semantic:

$A;B$ holds on a path iff there is a future cycle n , such that A holds up to and including the n^{th} cycle, and B holds on the path starting at the $(n+1)^{\text{th}}$ cycle.

Sequential Extended Regular Expression (SERE)

- The *fusion* operator ($:$) constructs a SERE in which two SEREs overlap by one cycle. Namely, the second starts at the cycle in which the first completes.

Informal semantic:

$A:B$ holds on a path iff there is a future cycle n , such that A holds up to and including the n^{th} cycle, and B holds on the path starting at the n^{th} cycle.

Clocked SERE

- The *clock operator* @ defines the clock context in a SERE.

Example:

The SERE {a;b} holds from time 2 to time 3


The clocked SERE {a;b}@clk holds from time 1 to time 3

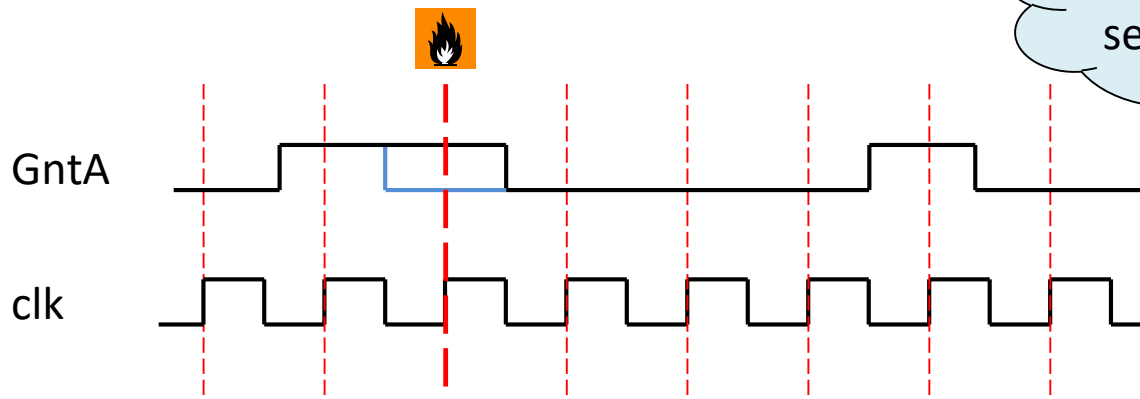
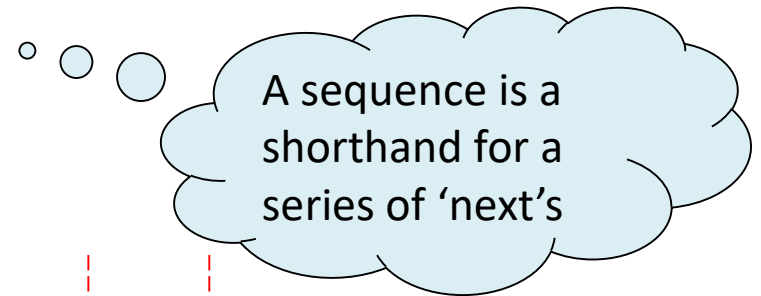
time	0	1	2	3	4
clk	0	1	0	1	0
a	0	1	1	0	0
b	0	0	0	1	0

Sequence is a Property

(Property = Sequence)

- GntA is never high for two successive clock cycles
`never (GntA and next[1] GntA) @ (posedge clk);`


`never (GntA; GntA) @ (posedge clk);`
└──────────┘
Sequence/SERE



Sequence is a Property

(Property = Sequence $\mid \rightarrow$ Sequence
| Sequence $\mid \Rightarrow$ Sequence)

Suffix implication operators:

- SERE_A $\mid \rightarrow$ SERE_B
If SERE_A completes, SERE_B has to begin in the same cycle
- SERE_A $\mid \Rightarrow$ SERE_B
If SERE_A completes, SERE_B has to begin in the next cycle

Example

```
always ((ReqA and next[1] GntA) -> next (Busy and next[1] Done)) @ (posedge clk);
```

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

if

then

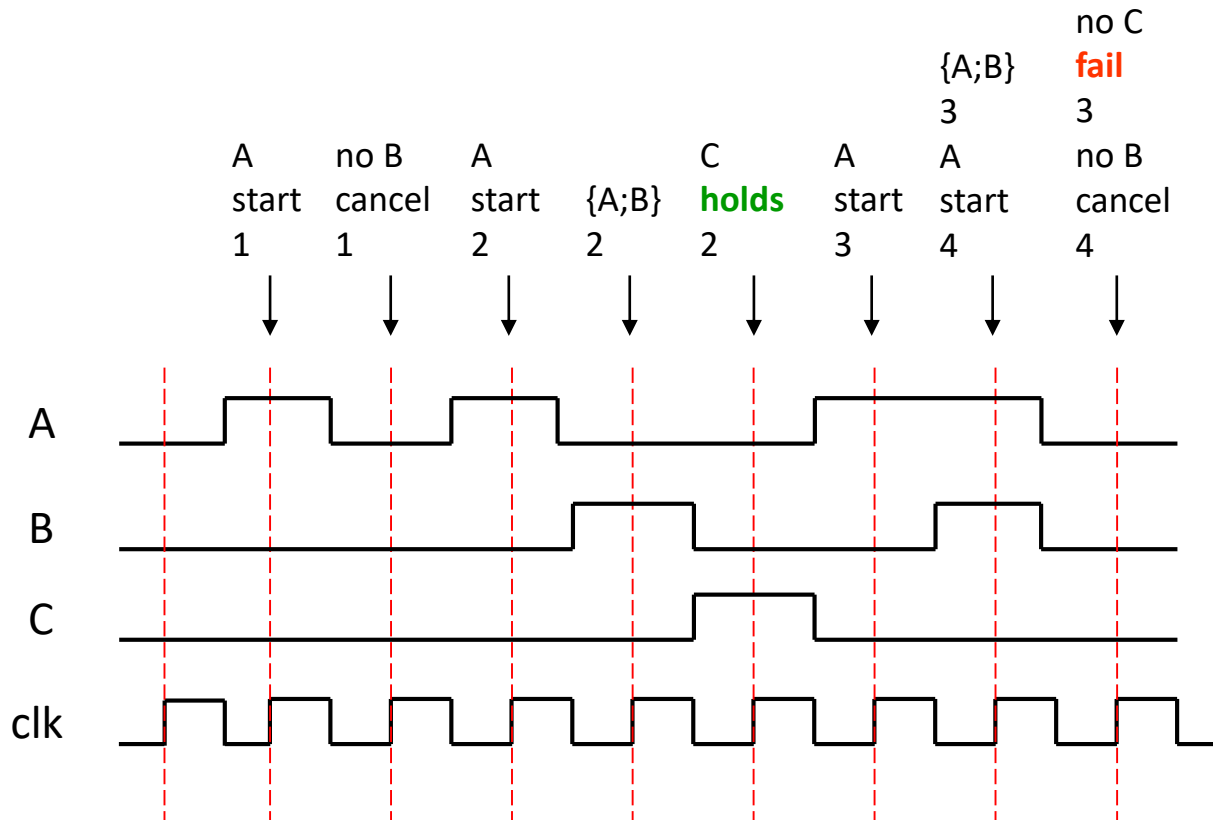
The left-hand side (LHS) sequence is the 'enabling' sequence

The suffix implication operator says "if LHS, then RHS"

The right-hand side (RHS) sequence is the 'fulfilling sequence'

Example

`always ({ A ; B } | => { C }) (posedge clk);`



Consecutive repetition of SERE

The SERE consecutive repetition operator ($[*]$) constructs repeated consecutive concatenation of a given SERE.

- $\text{repeated_SERE} = \text{SERE } [*[\text{Count}]]$
| $\text{SERE } [+]$
- $\text{Count} = \text{number} \mid \text{lowerBound} : \text{upperBound}$

Example:

$s[*i]$ i consecutive repetitions of s

$s[*i:j]$ between i to j consecutive repetitions of s

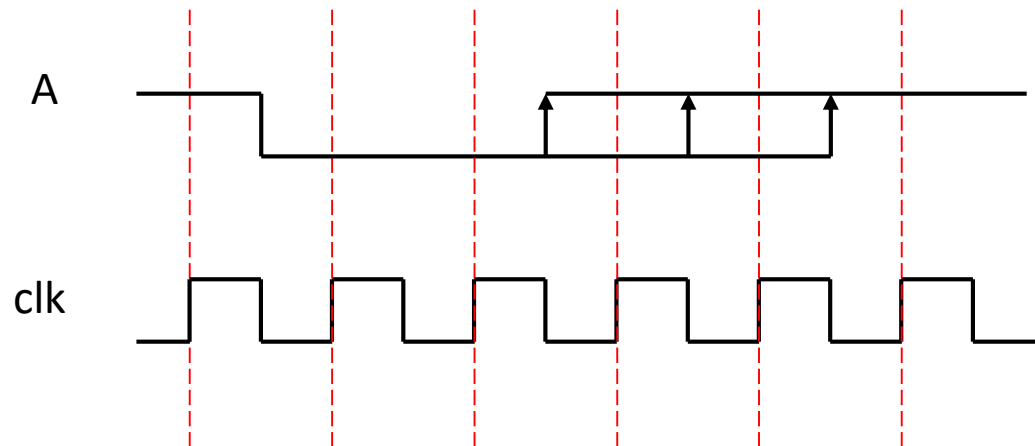
$s[*]$ 0 or more consecutive repetitions of s

$s[+]$ 1 or more consecutive repetitions of s

Example

- If A fell (1->0), then it remains low for 2 up to 4 clock cycles

`always ({ A ; !A } | => { !A[*1:3] ; A }) @ (posedge clk);`



VERIFICATION LAYER

Verification layer

The verification layer provides *directives* that tell a verification tool what to do with the specified sequences and properties.

PSL_directive = [Str. identifier:] Verification_directive

Verification_directive = **assert** Property [report Str]
 | **assume** Property
 | **cover** Sequence [report Str]
 ...

Verification directive

- **assert:** the assert directive instructs the verification tool to verify that a property holds. It may optionally include a character string containing a message to report when the property fails to hold.

Example:

```
assert always (ack -> next (!ack until req))  
  report "A second ack occurred before the next req";
```

If `always (ack -> next (!ack until req))` does not hold, then the message "A second ack occurred before the next req" should be displayed.

Verification directive

- **assume:** the assume directive instructs the verification tool to constrain the verification (e.g., the behavior of the input signals) so that the given property holds.

Example:

```
assume always (ack -> next (ack));
```

Instructs the verification tool to constrain the verification (e.g., the behavior of the input signals) so that the property always (ack -> next !ack) holds in the design.

Verification directive

- **cover:** the cover directive directs the verification tool to check if a certain path was covered by the verification space based on a simulation test suite or a set of given constraints. It may optionally include a character string containing a message to report when the specified sequence occurs.

Example:

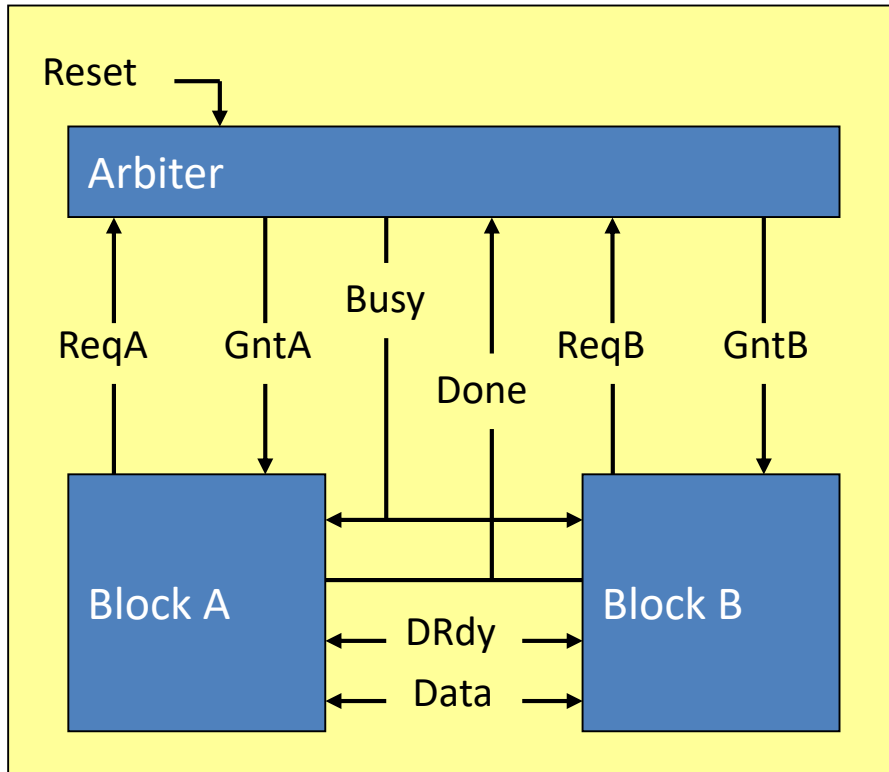
```
cover {a; b; c}
```

```
report "The sequence a, b and c was covered";
```

If {a;b;c} is covered at least one time, then the message “The sequence a, b and c was covered” should be displayed.

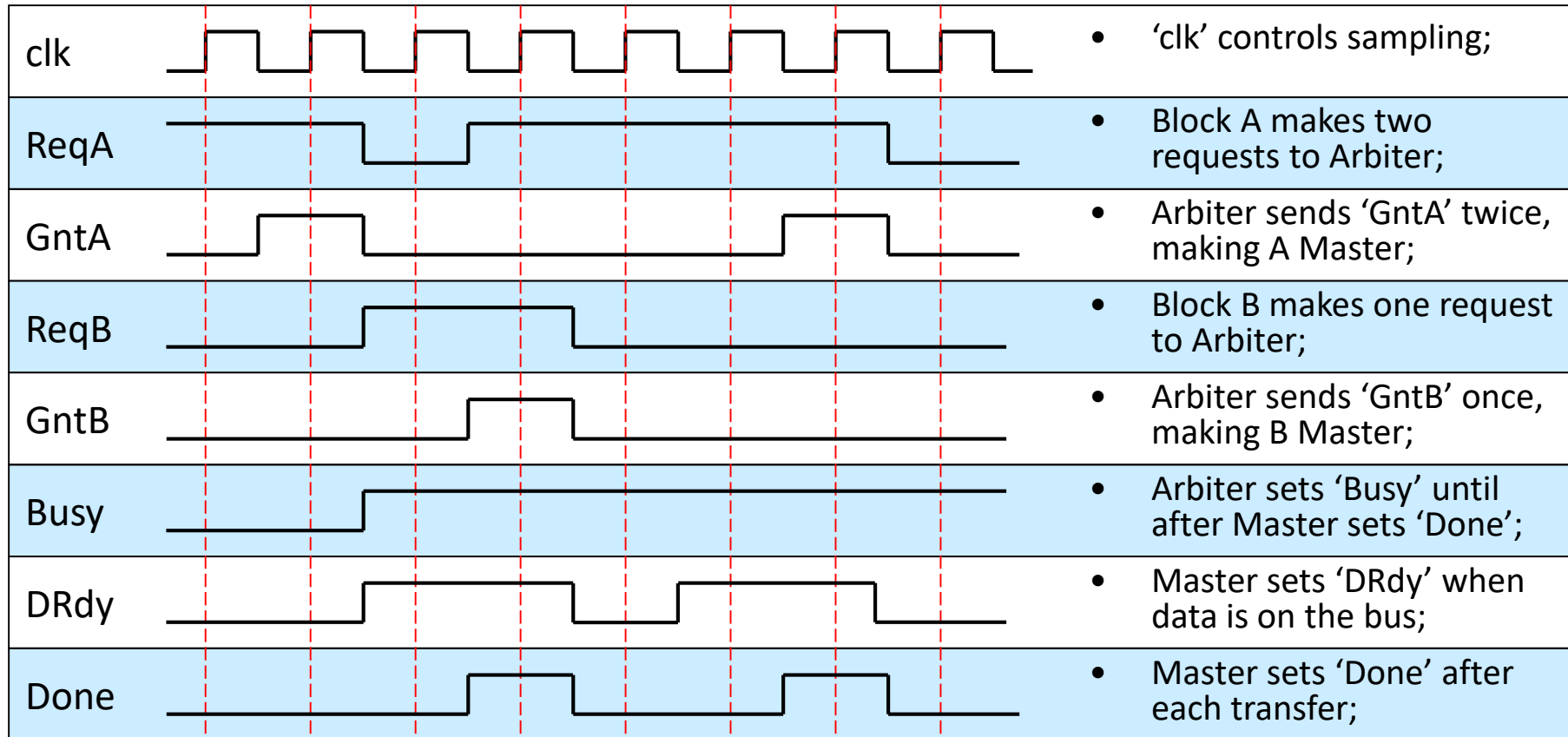
A simple example

- 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 Arbiter.

Simple example bus protocol



Some assertions to check...

default clock = posedge clk;

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

More assertions to check...

- If ReqB is followed by GntB, then next is Busy, and next is Done.
 - assert always (ReqB -> next (GntB -> next (Busy -> next (Done))));
 - assert always ({ReqB ; GntB } ==> (Busy ; Done));
- If ReqB is followed by GntB, then next is Busy 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 || GntB) -> next (Busy until_ Done));
 - assert always ({GntA || GntB} ==> {Busy[*]:(Busy && Done)});

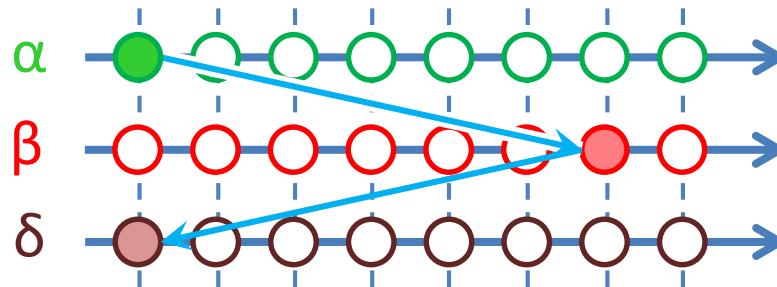
PSL SIMPLE SUBSET

PSL simple subset

- PSL can express properties that involve jumping back and forth the future
 - In static verification jumps are always possible
 - In dynamic verification time advances monotonically
- The *Simple Subset* of PSL conforms to the notion of monotonic advancement of time
 - Left to right through the property

An illustrative example

- $(\alpha \ \&\& \text{eventually!} \ (\beta)) \rightarrow (\delta)$
is NOT part of the Simple Subset



Simple Subset Restrictions

PSL Operator	Restrictions
Not	Operand must be Boolean
Never	Operand must be Boolean or sequence
eventually!	Operand must be Boolean or sequence
or	At most one operand may be non-Boolean
->	LHS must be Boolean or sequence
<->	Both operands must be Boolean
until until!	RHS must be Boolean
until_ until!_	Both operands must be Boolean