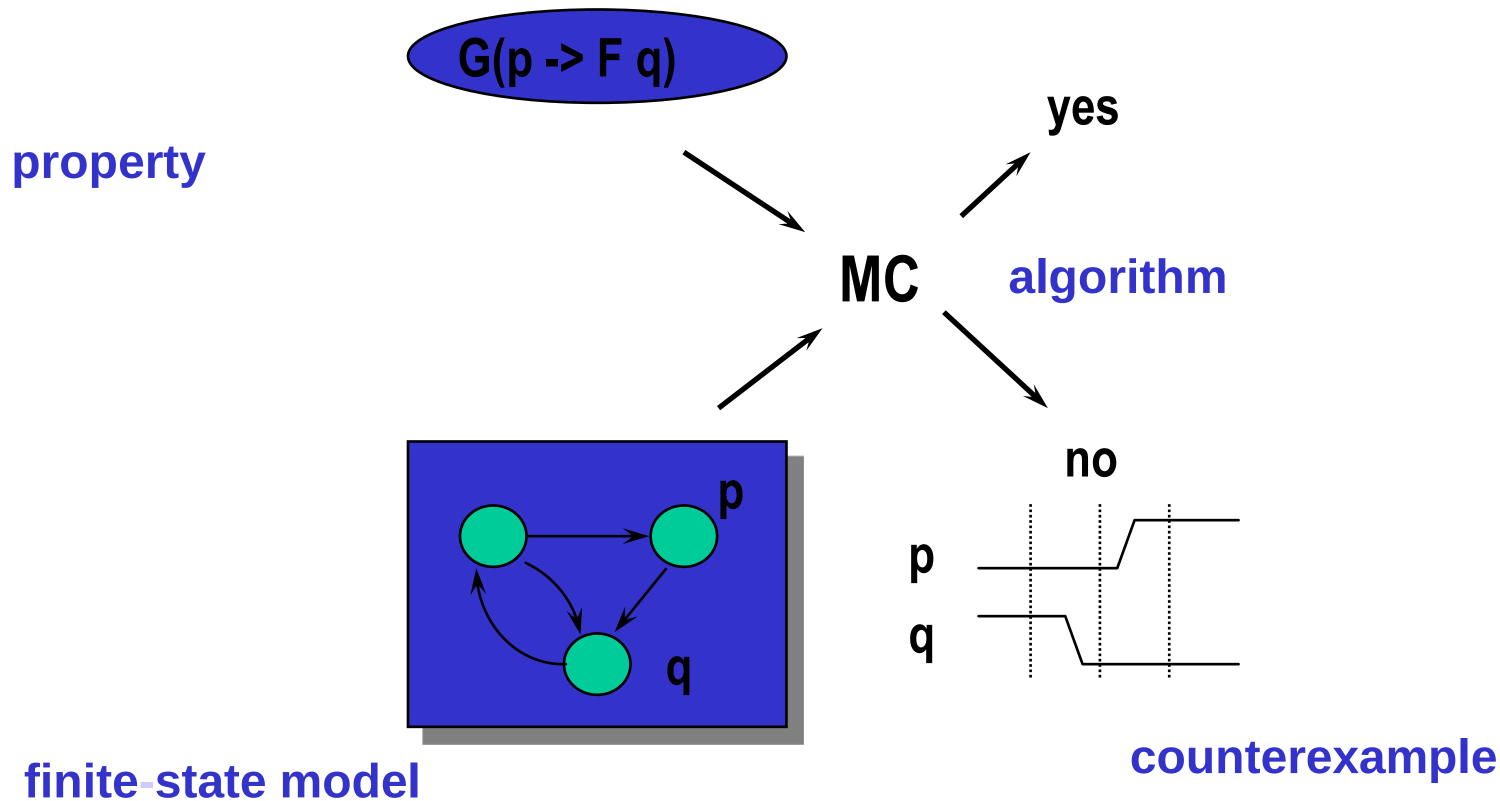
Specifying circuit properties in PSL

(Some of this material is due to Cindy Eisner and Dana Fisman, with thanks)

See also the Jasper PSL Quick Ref.

Background: Model Checking



(Ken McMillan)

Two main types of temporal logic

- Linear-time Temporal Logic (LTL)
  - must properties, safety and liveness
  - Pnueli, 1977
- Computation Tree Logic (CTL)
  - branching time, may properties, safety and liveness
  - Clarke and Emerson, Queille and Sifakis, 1981

Linear time conceptually simpler (words vs trees)
Branching time computationally more efficient
We will return to this in a later lecture

But

temporal logics hard to read and write!

Computation Tree Logic

A sequence beginning with the assertion of signal strt, and containing two not necessarily consecutive assertions of signal get, during which signal kill is not asserted, must be followed by a sequence containing two assertions of signal put before signal end can be asserted

AG~(strt & EX E[~get & ~kill U get & ~kill & EX E[~get & ~kill U get & ~kill & E[~put U end] or E[~put & ~end U (put & ~end & EX E[~put U end])]]])

PSL version

```
always({strrt; {get[=2]}&&{kill[=0]}}
                |=>  {{put[=2]}&&{end[=0]}})
```

Basis of PSL was Sugar (IBM, Haifa)

Grew out of CTL

Added lots of syntactic sugar

Engineer friendly, used in many projects

Used in the industrial strength MC RuleBase

Standardisation led to further changes

Assertion Based Verification (ABV) can be done in two ways

During simulation

– (dynamic, at runtime, called semi-formal verification, checks only those runs, restricted to a subset of the property language)

As a static check

– (formal verification, covers all possible runs, more comprehensive, harder to do)

(Note: this duality has been important for PSL's practical success, but it also complicates the semantics!)

Safety Properties

always (p)

”Nothing bad will ever happen”

Most common type of property checked in practice

Easy to check (more later)

Disproved by a finite run of the system

always (not (gr1 and gr2))

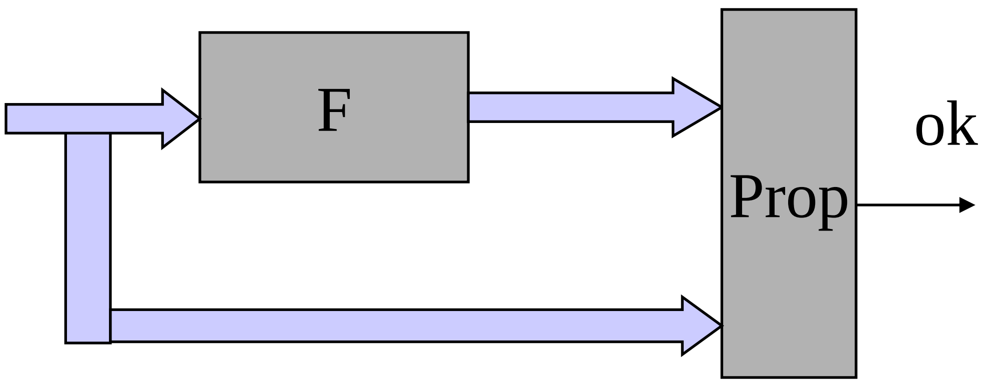
Observer
(alternative to property language)

Observer written in same language as circuit

Safety properties only

Used in verification of control programs such as Lustre programs that control safety critical features in the airbus

(and in Lava later)



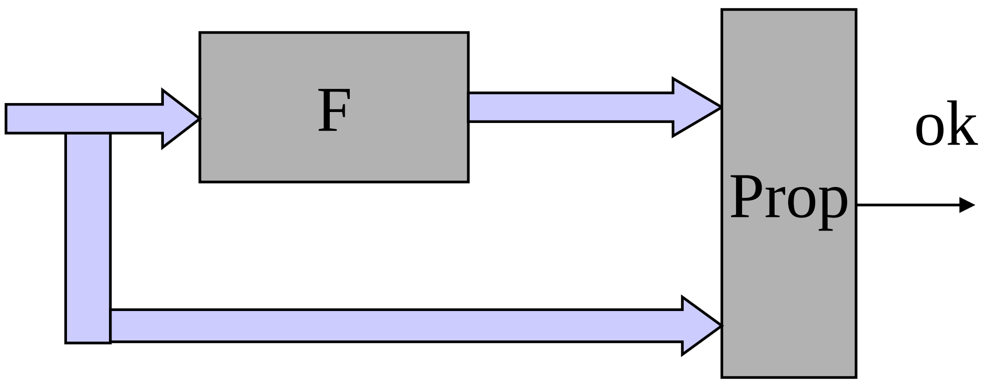
Observer
(alternative to property language)

Observer written in same language as circuit

Safety properties only

Used in verification of control programs such as Lustre programs that control safety critical features in the airbus

(and in Lava later)



Property language-based tools often use observers intenrally

Back to PSL

Layers

Boolean	(we use VHDL flavour and the simplest choice of what the clock in properties is)
Temporal	(temporal operators, SEREs)
Verification	(group properties, specify whether to verify or assume etc.)
Modelling	(subset of chosen HDL)

Concrete example

```
vunit counter_properties(counter(behavioral)) {
  default clock is (rising_edge(clk));

  property bounded is
   always (o < bound);

  property validtransitions is
     …                          ;

  assert bounded;
  assert validtransitions;
}
```

Concrete example

```
vunit counter_properties(counter(behavioral)) {
  default clock is (rising_edge(clk));

  property bound
   always

  property
     …

  assert b
  assert v
}
```

Defines a complete verification job
(verification layer)

Concrete example

```
vunit counter_properties(counter(behavioral)) {
  default clock is rising_edge(clk));

  property bounde
   always

  property
     …

  assert b
  assert v
}
```

Prop_name (entity (architecture))

Concrete example

```
vunit counter_properties(counter(behavioral)) {
  default clock is (rising_edge(clk));

  property bound
   always

  property
     …

  assert b
  assert v
}
```

Defines clock to be used by all properties and sequences in the vunit
(can also clock properties and sequences individually)

Concrete example

```
vunit counter_properties(counter(behavioral)) {
  default clock is (rising_edge(clk));

  property bounded is
   always (o < bound);

  property validtransitions is
     …                           ;

  assert bounded;
  assert validtransitions;
}
```

property declaration
(can also declare sequences)

Concrete example

```
vunit counter_properties(counter(behavioral)) {
  default clock is (rising_edge(clk));

  property bounded is
   always (o < bound);

  property validtransitions is
     …                          ;

  assert bounded;
  assert validtransitions;
}
```

property
with temporal operator always
(the most usual one)
Others include next and until

Concrete example

```
vunit counter_properties(counter(behavioral)) {
  default clock is (rising_edge(clk));

  property bounded is
   always (o < bound);

  property validtransitions i
     …                          ;

  assert bounded;
  assert validtransitions;
}
```

verification directive
assert <property>

(others include assume, restrict, cover)

Temporal layer

Our main concern

how to define properties and sequences

use both temporal operators (related to LTL) and sequences to build properties

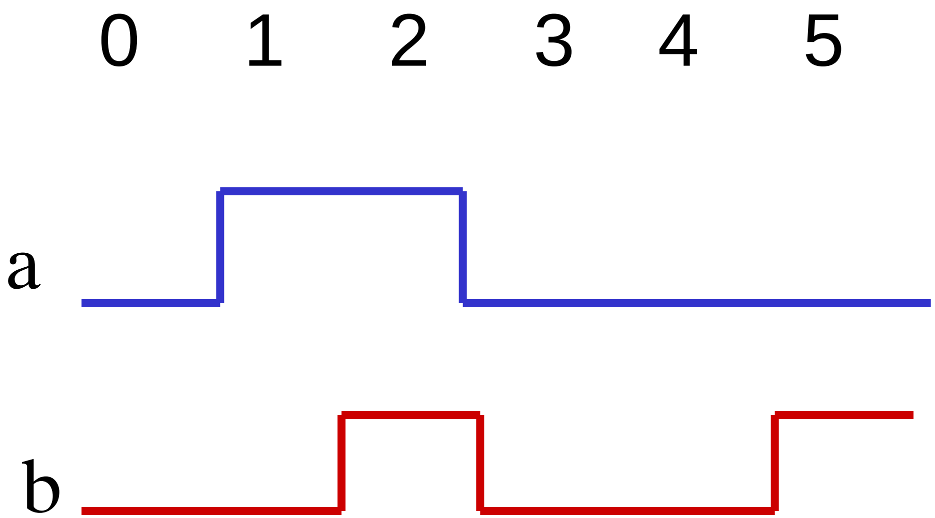
We are in the so-called Foundation Language (FL), used
for both simulation and FV (in sim. prop. can be checked in a single run)

There is an optional branching extension (OBE, related to CTL), used only for FV

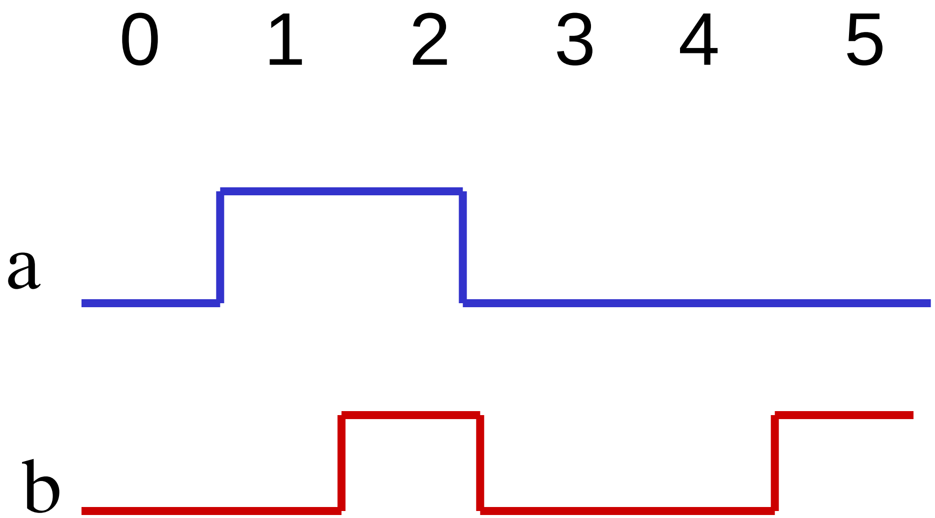
Temporal operators

always (= never not ...)

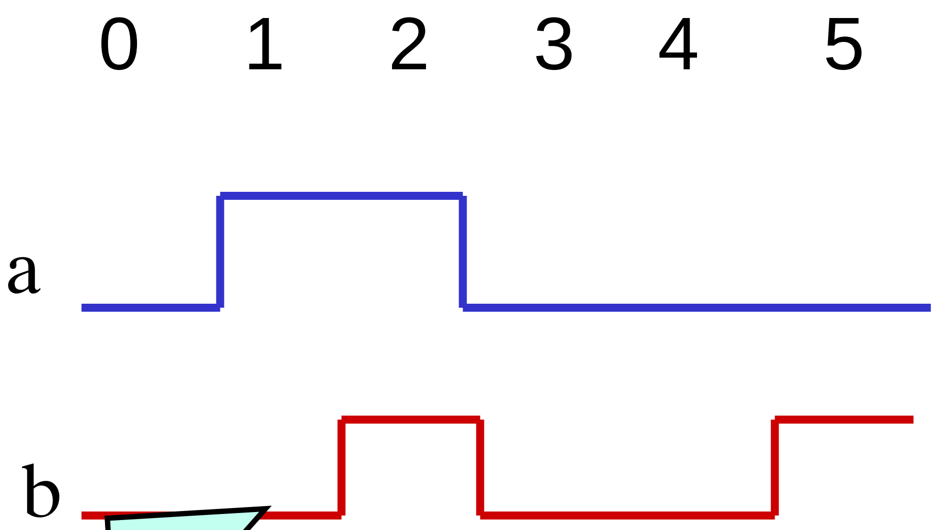
Most PSL properties start with this!



assert not (a and b) ?

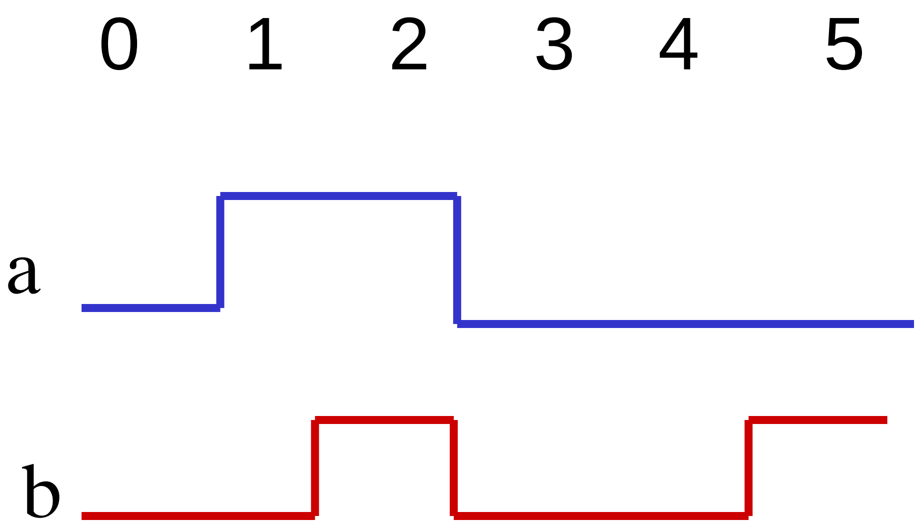


assert not (a and b) holds

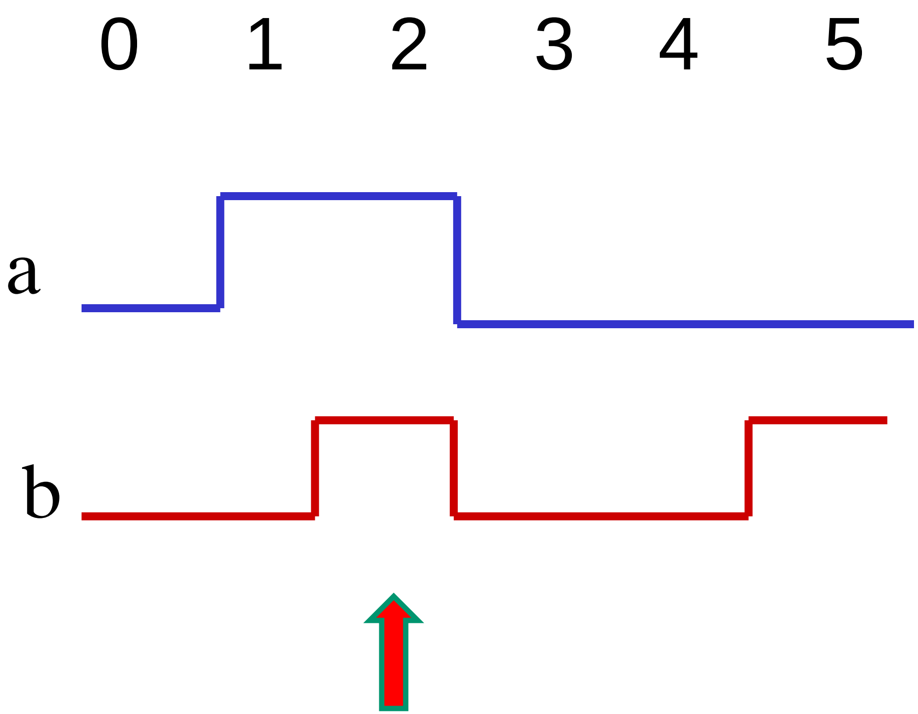


Pure boolean assertion refers to FIRST cycle ONLY

assert not (a and b) holds



assert always not (a and b) ?



assert always not (a and b) does not hold

assert never (a and b) is same

Temporal operators

next

next p holds in a cycle if p holds at the next cycle

Example

Whenever signal a is asserted then in the next cycle signal b must be asserted

Logical implication

Boolean

But often used inside temporal ops

p1 -> p2 is (not p1) or p2

if p1 then p2 else true

Logical implication

Boolean

But often used inside temporal ops

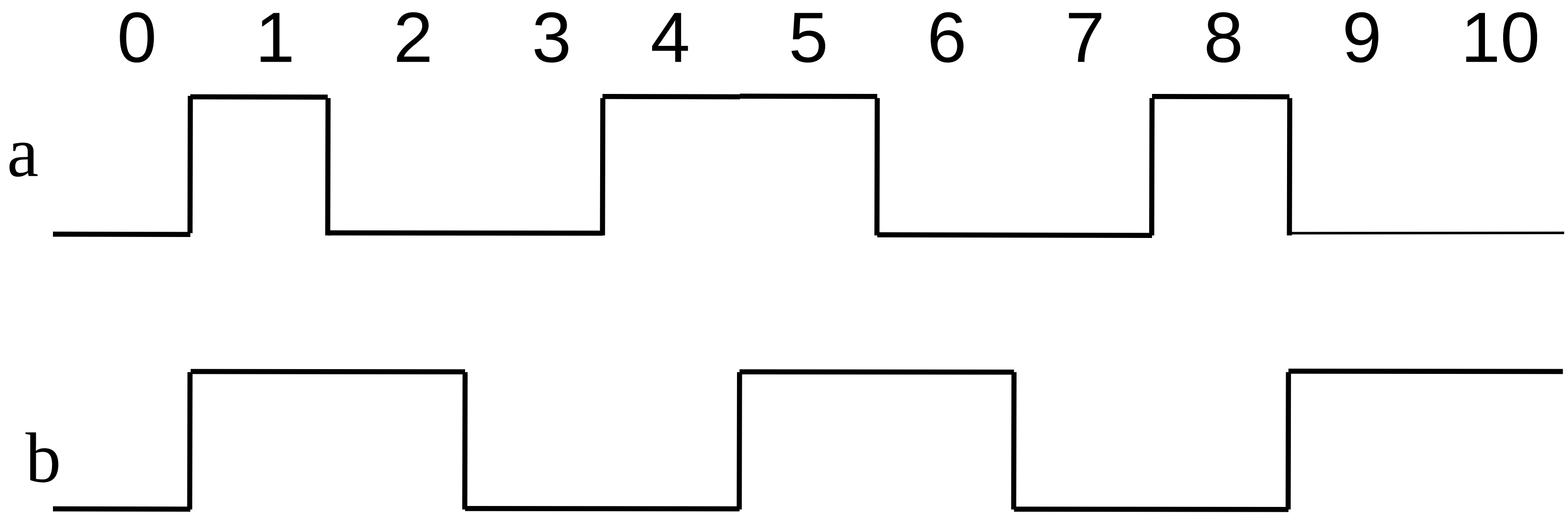
p1 -> p2 is (not p1) or p2

e true

is TRUE if p1 is false

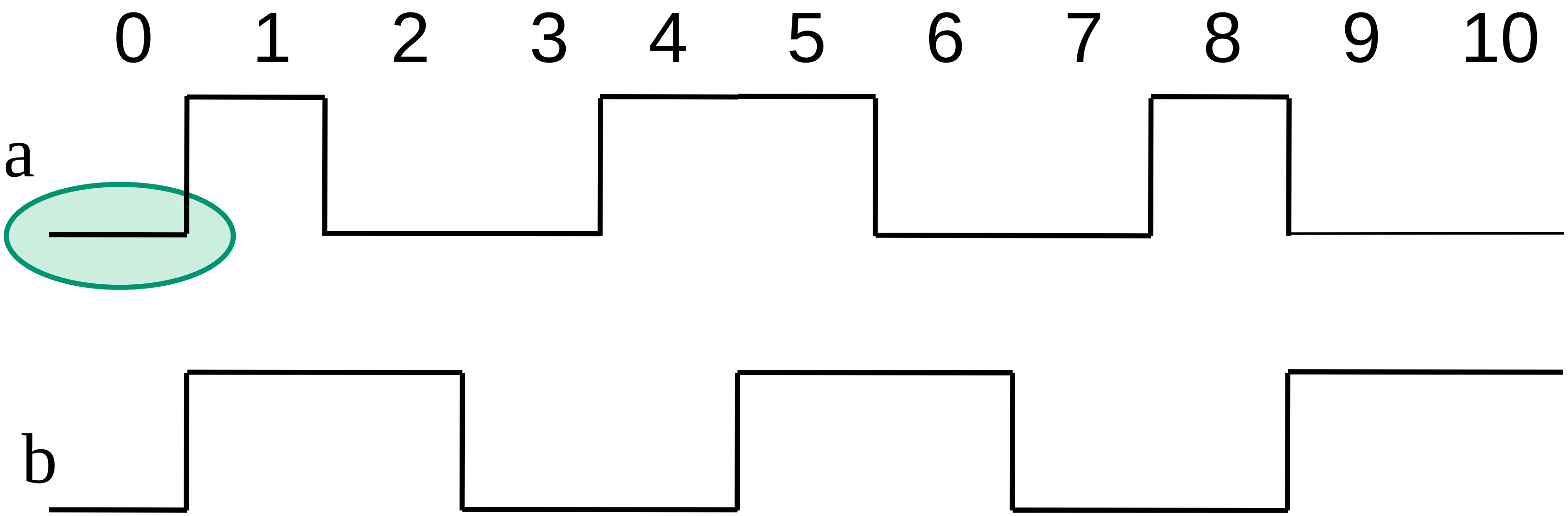
Beware!

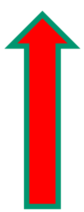
next and implication



assert always (a -> next b)

next and implication

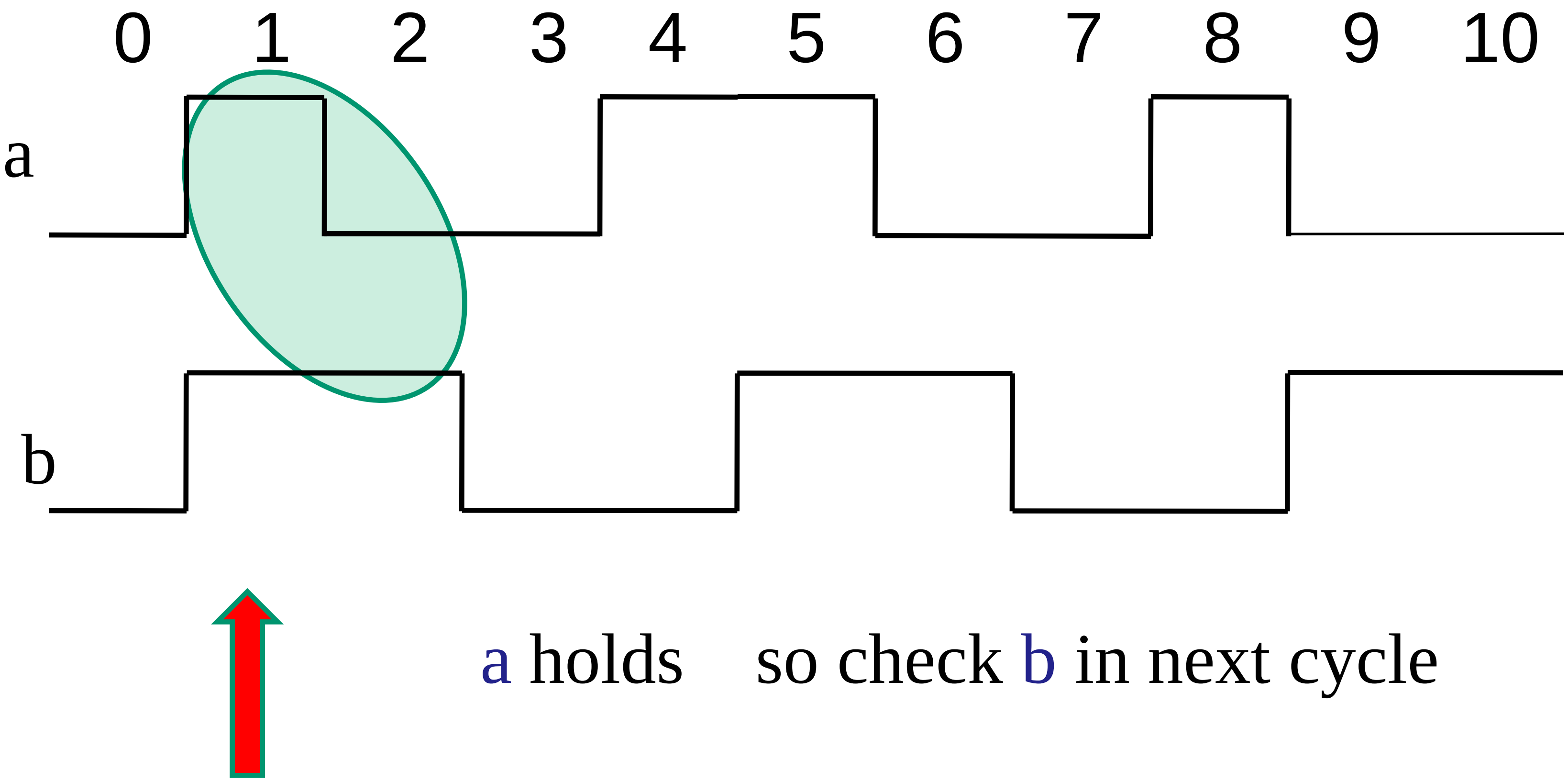




a does not hold ✓

assert always (a -> next b)

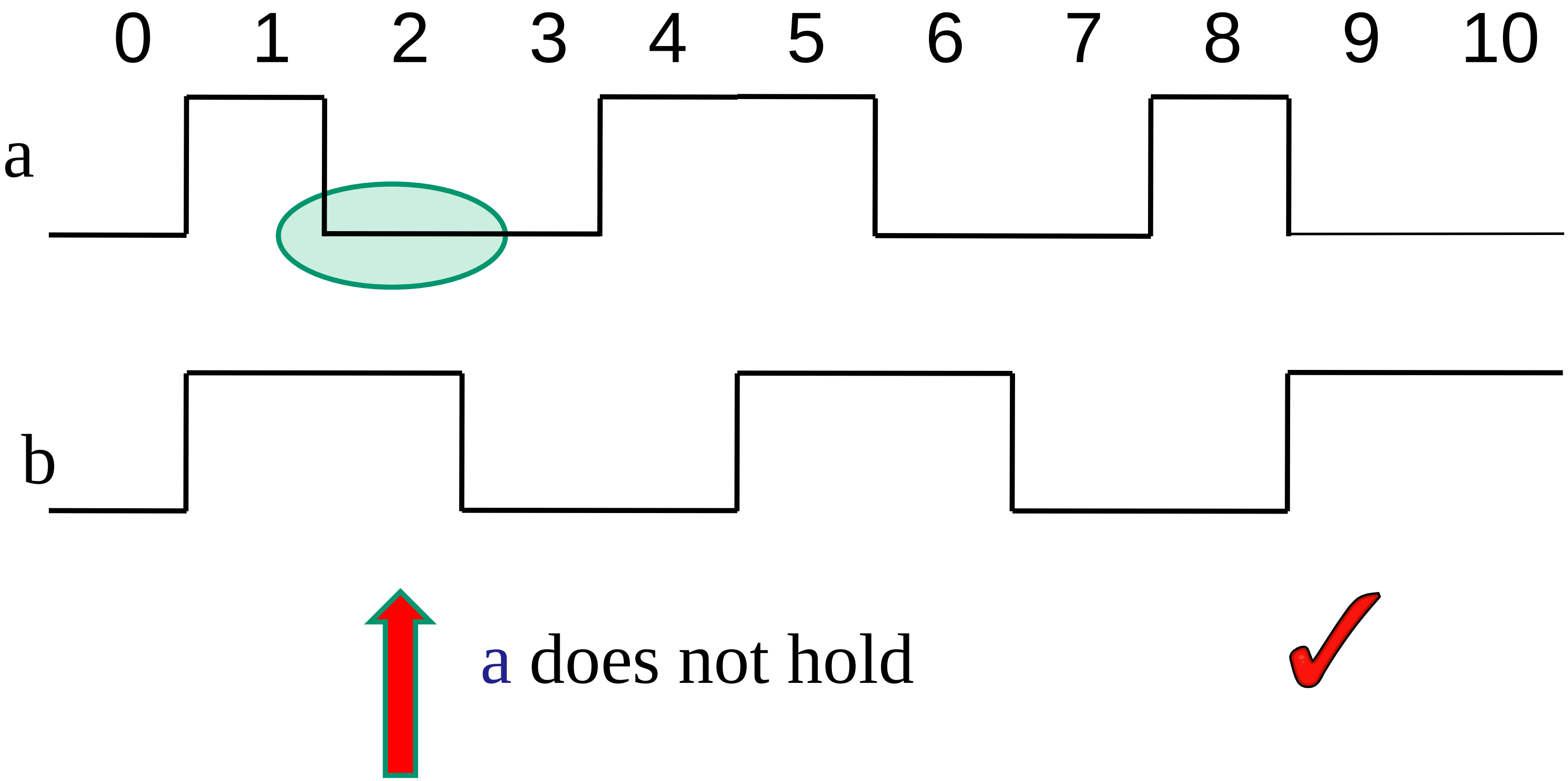
next and implication



a holds so check b in next cycle ✓

assert always (a -> next b)

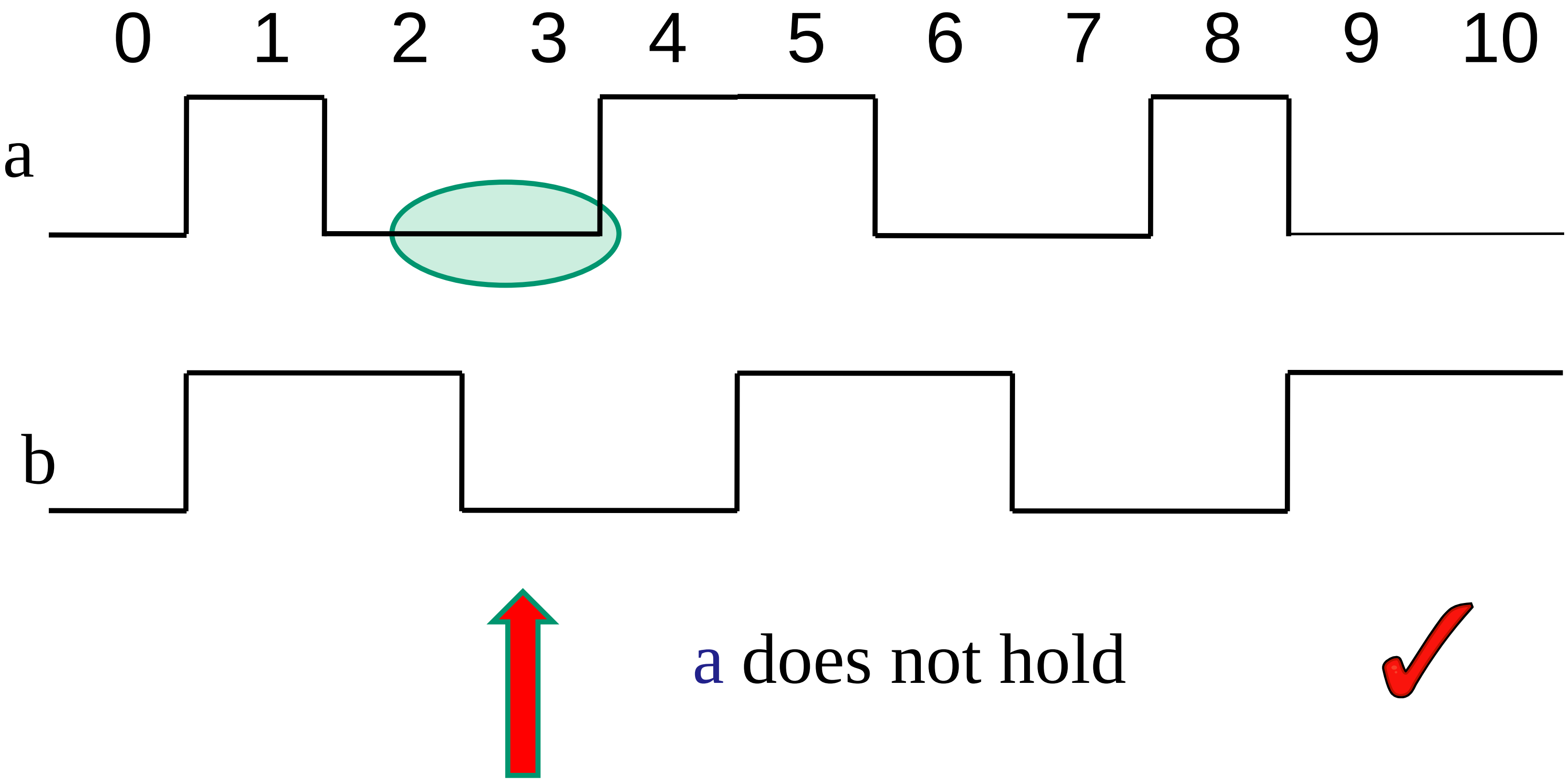
next and implication



assert always (a -> next b)

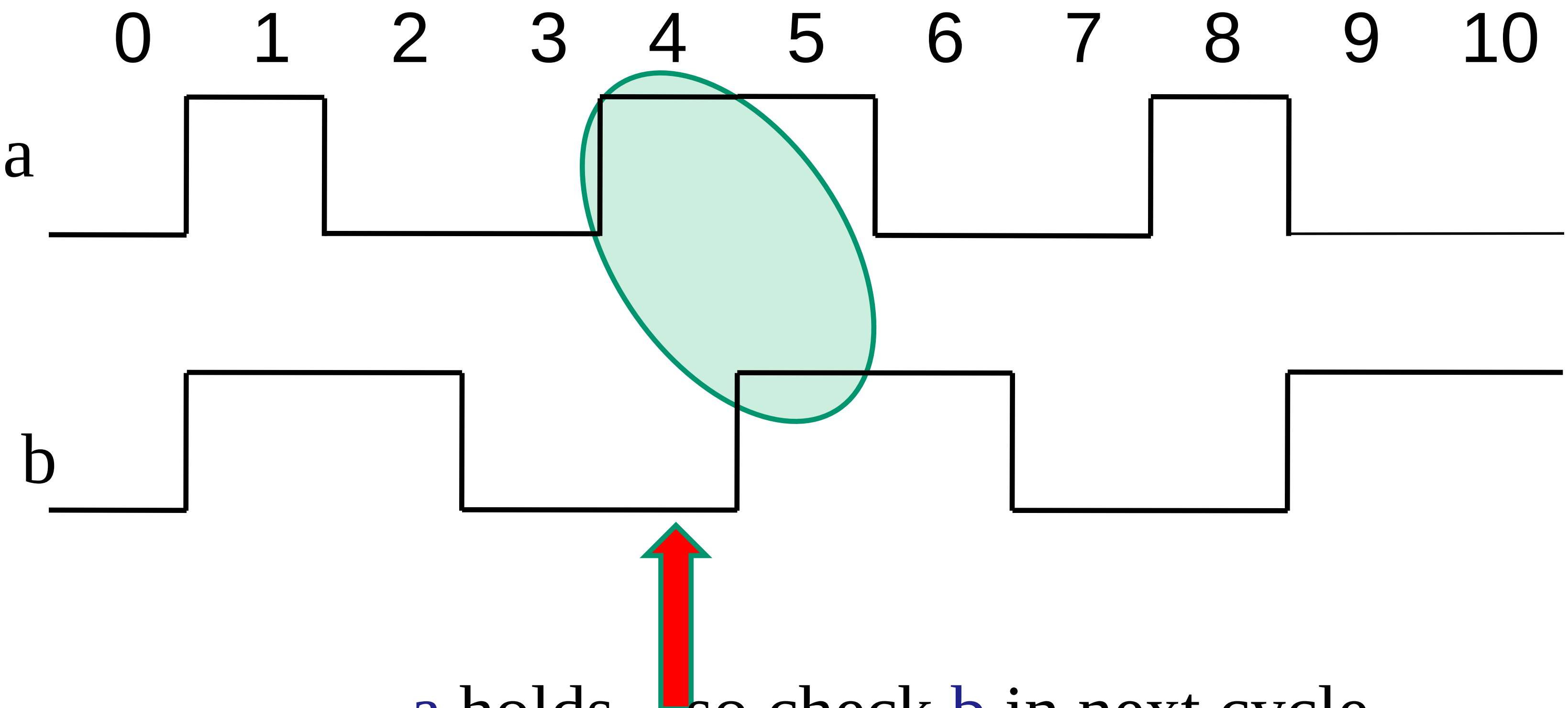
Note overlap with previous pair

next and implication



assert always (a -> next b)

next and implication

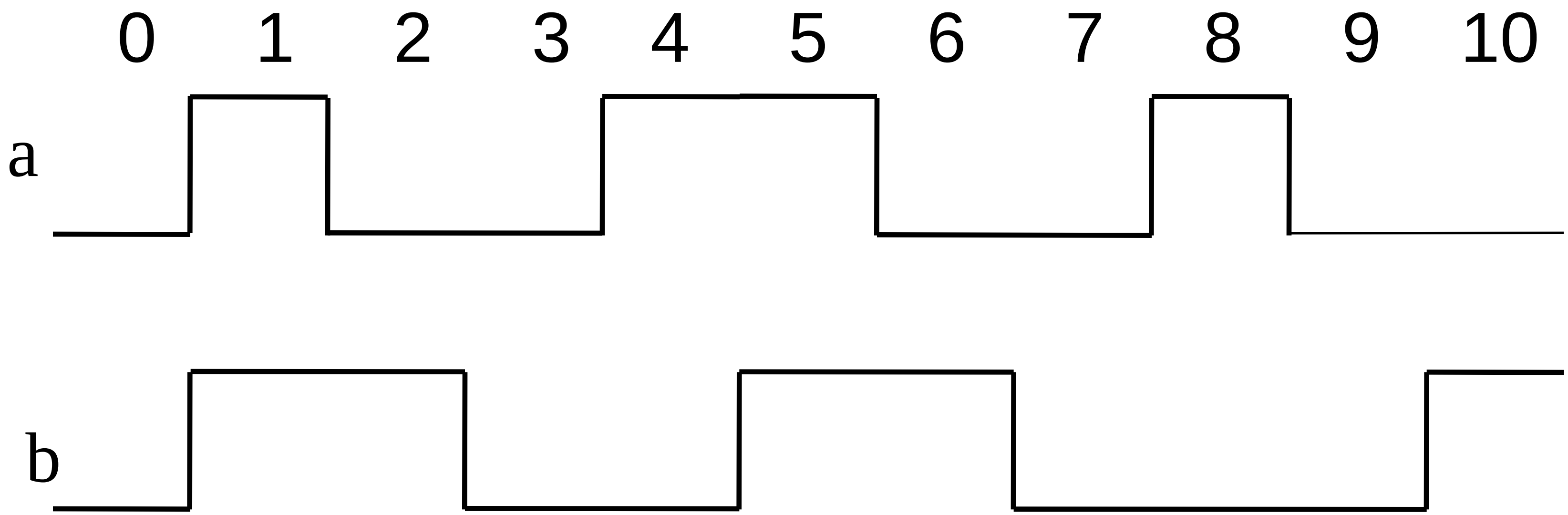


a holds so check b in next cycle ✓

assert always (a -> next b)

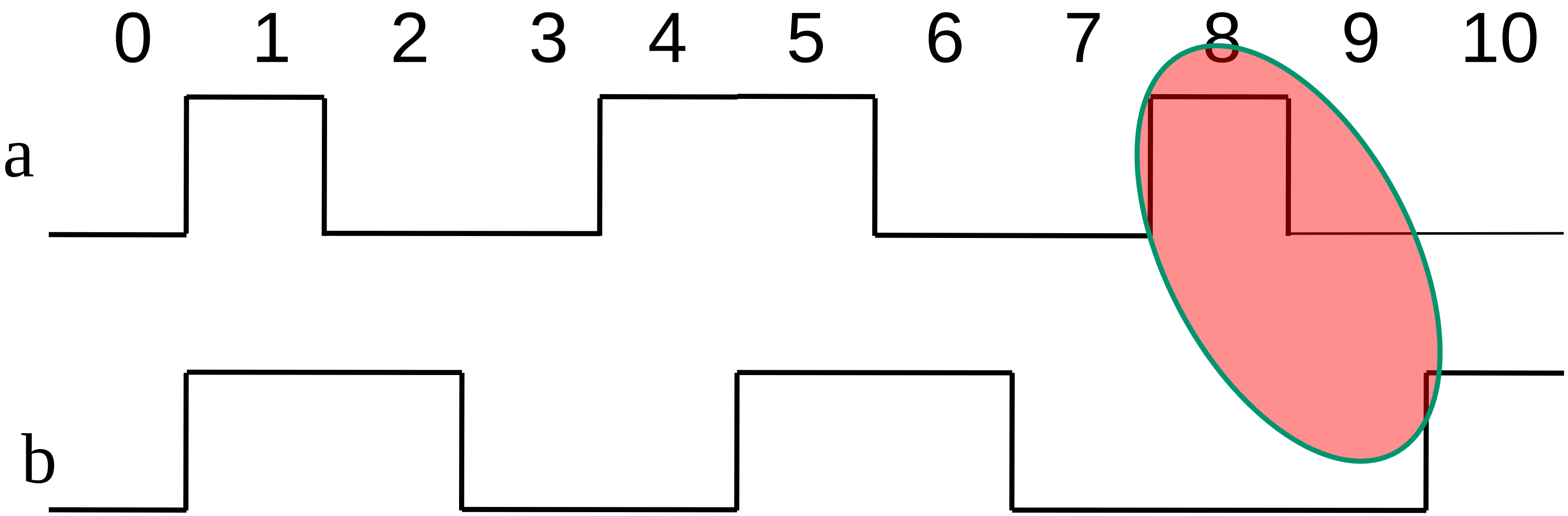
and so on…. ✓

Slightly different trace



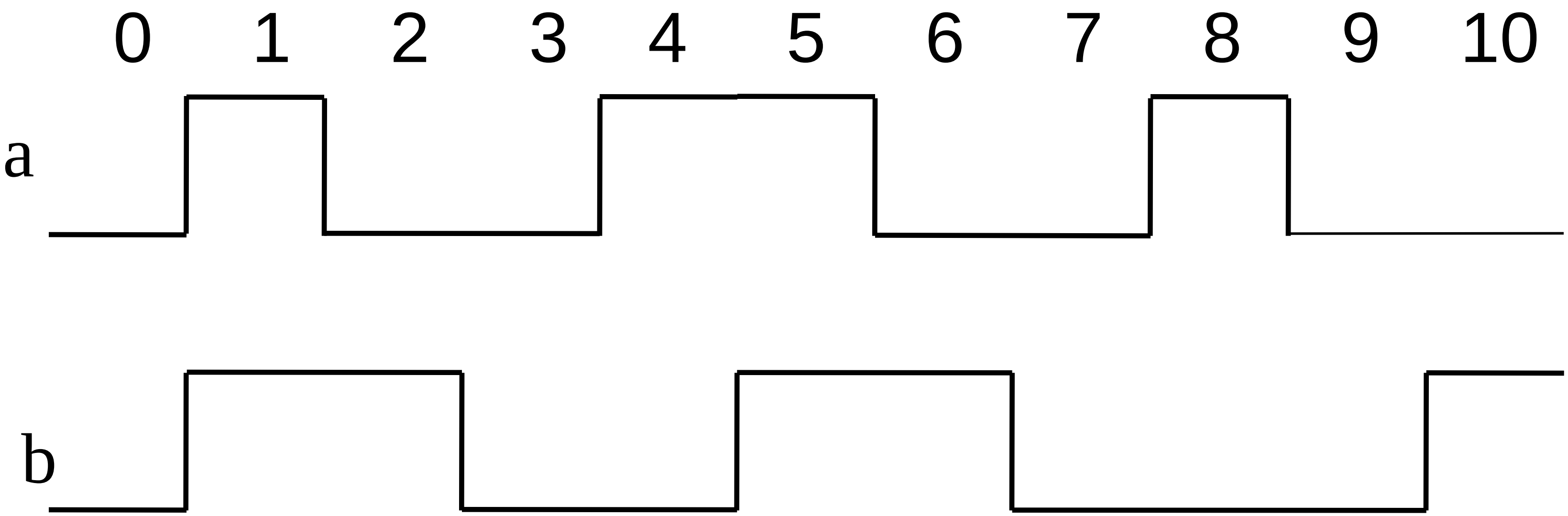
assert always (a -> next b) ?

Slightly different trace



assert always (a -> next b)

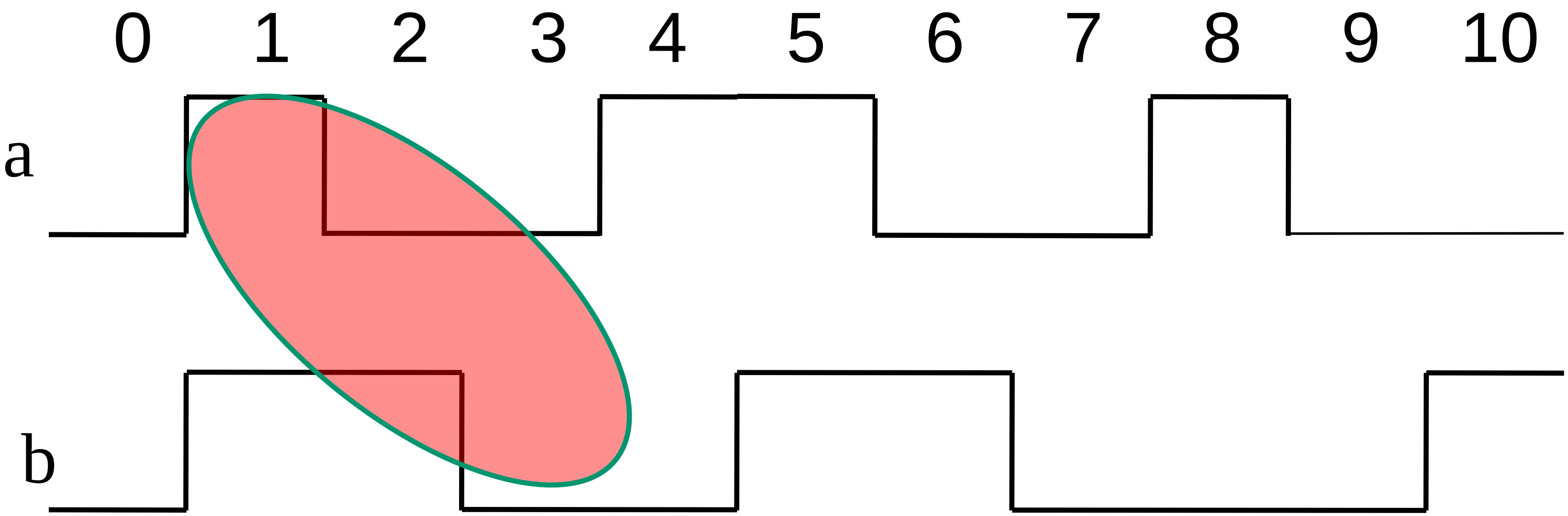
next[n] p   holds if p holds in nth cycle in future



next is next[1]

assert always (a -> next[2] b)   ?

next[n] p   holds if p holds in nth cycle in future



next is next[1]

assert always (a -> next[2] b)

More variants

Ranges

next_a[3 to 7] — all in range

next_e [3 to 5] — exists (= some) in range

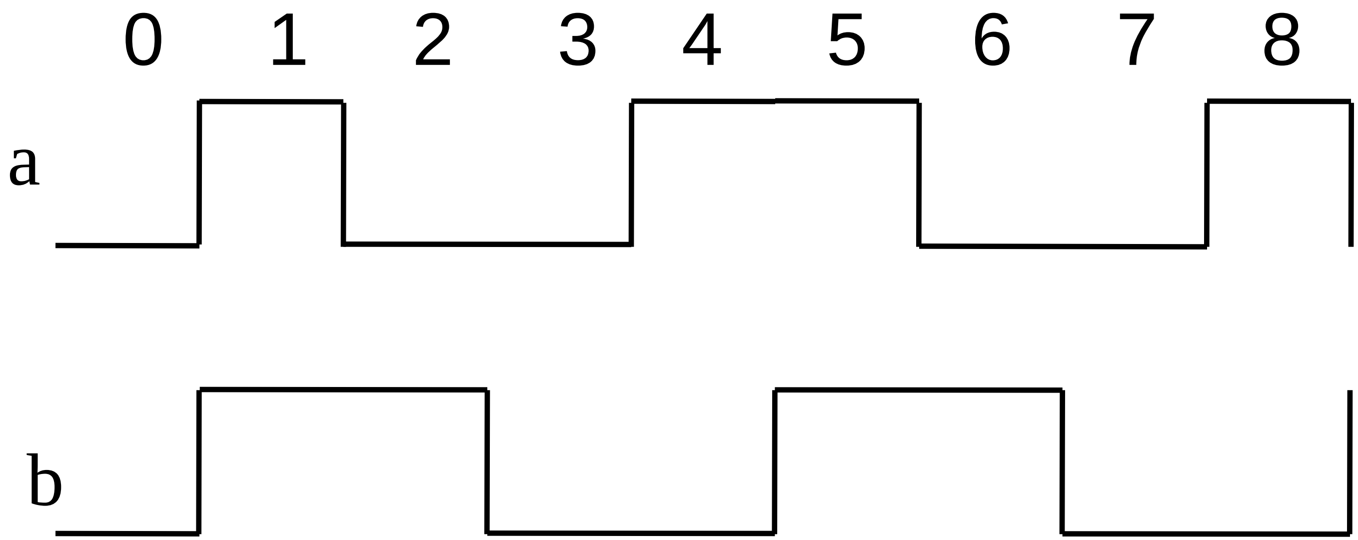
next_event(b) p — p should hold at next cycle at which Boolean b holds (could be this cycle)

Also comes in a and e versions for ranges

And yet more!   weak vs strong

Strong operator demands that the trace “not end too soon” indicated by !

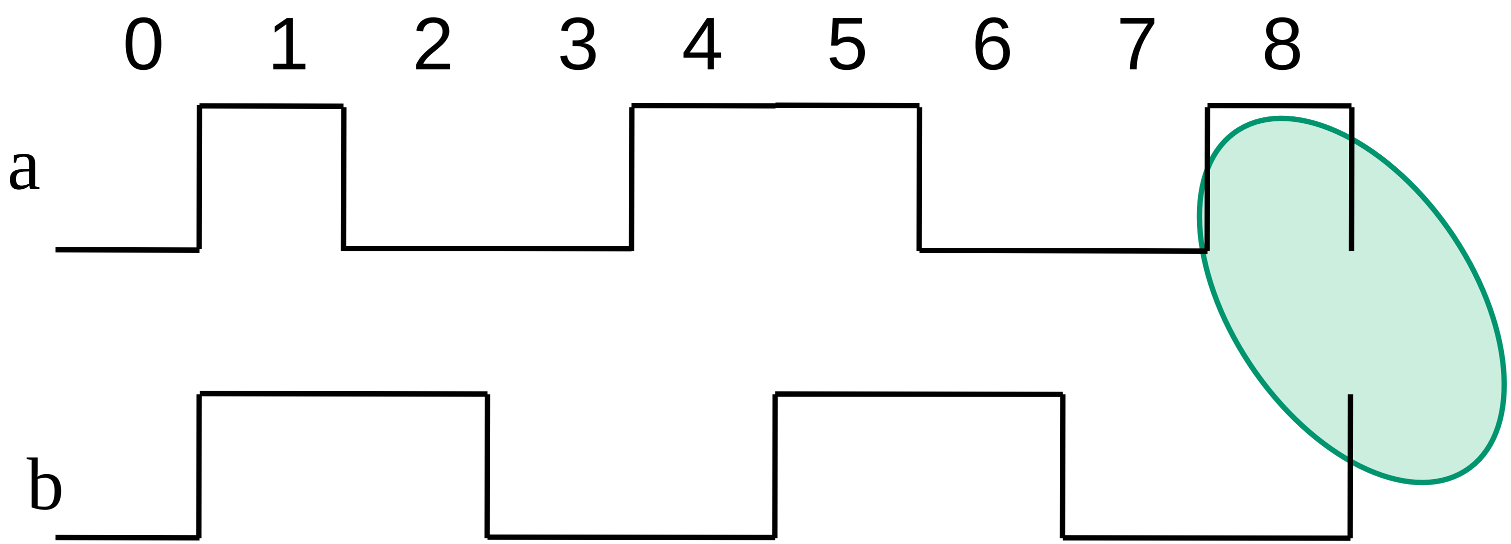
example



weak operator is lenient

assert always (a -> next b)

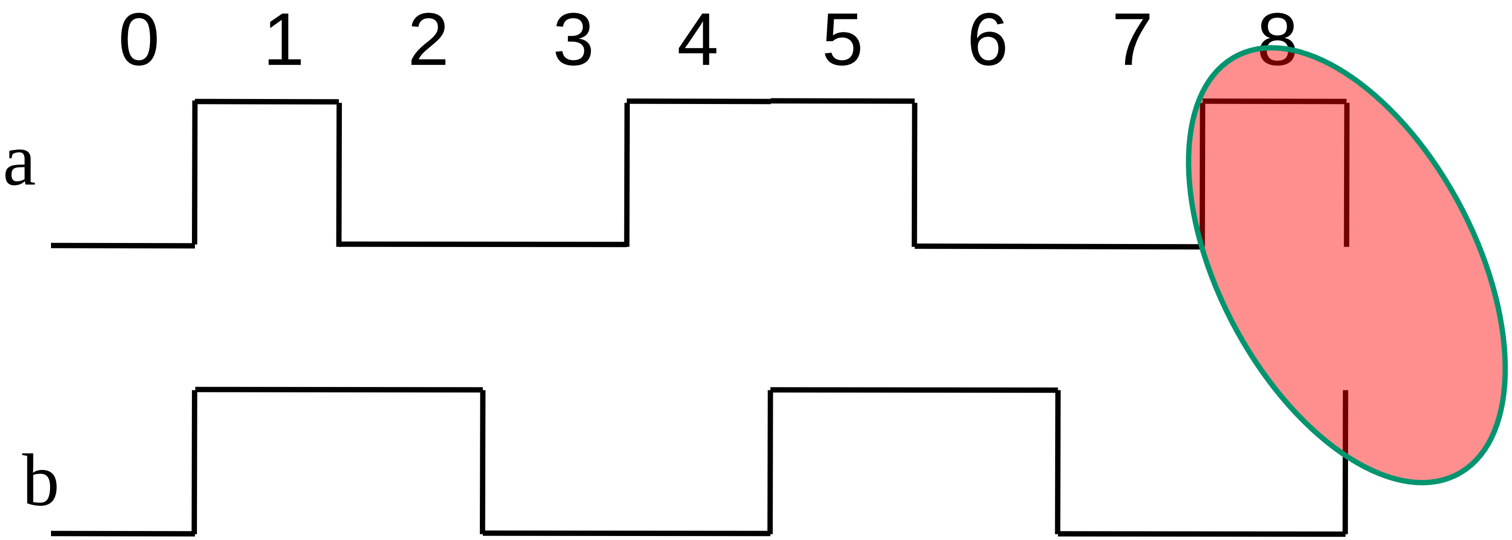
example



weak operator is lenient

assert always (a -> next b)

example



strong operator is strict

assert always (a -> next! b)

Temporal operators

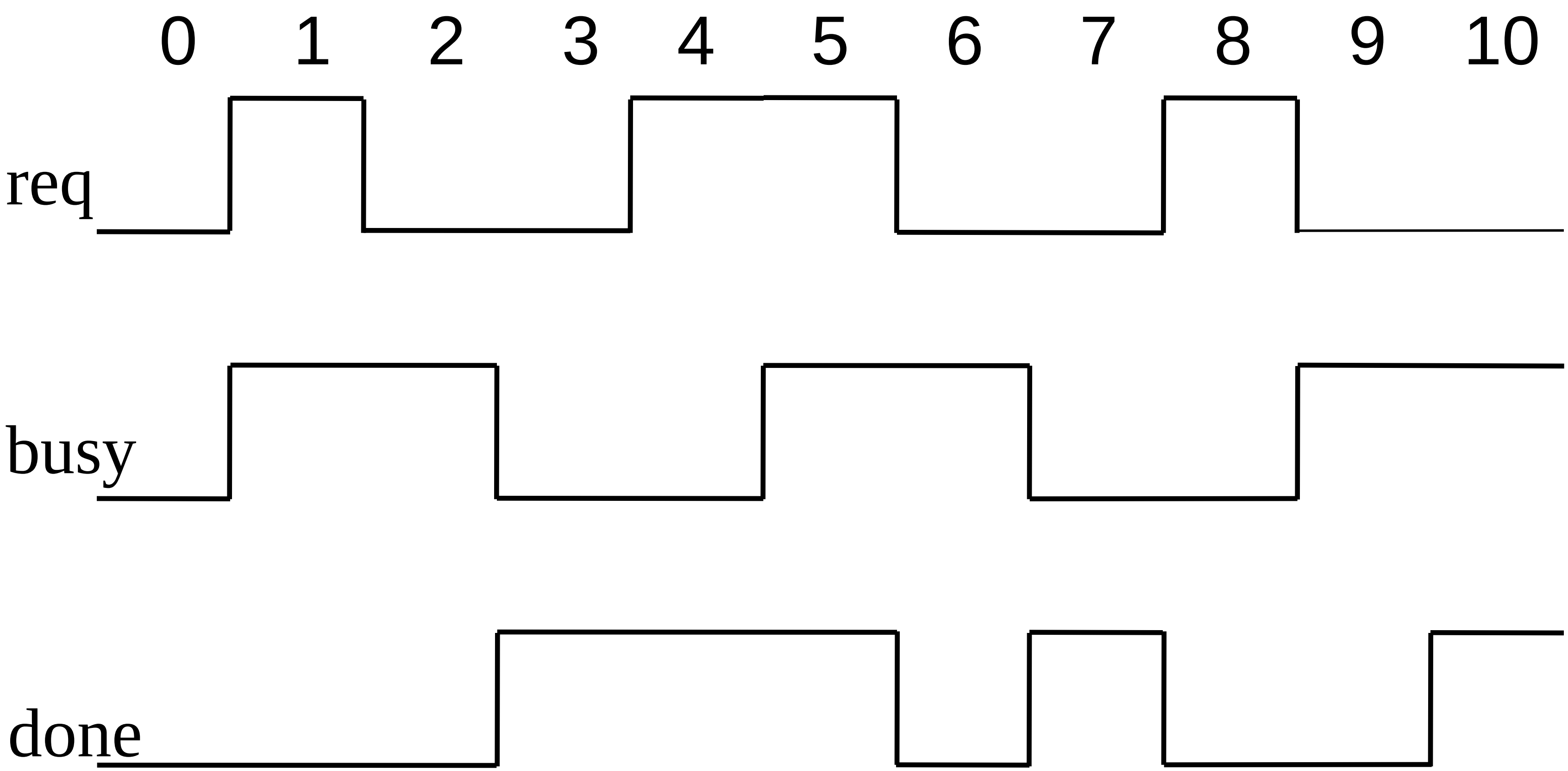
until

p until q	p holds in each cycle until (the one before) q holds
p until_ q	p holds in each cycle until (and including the one where) q holds

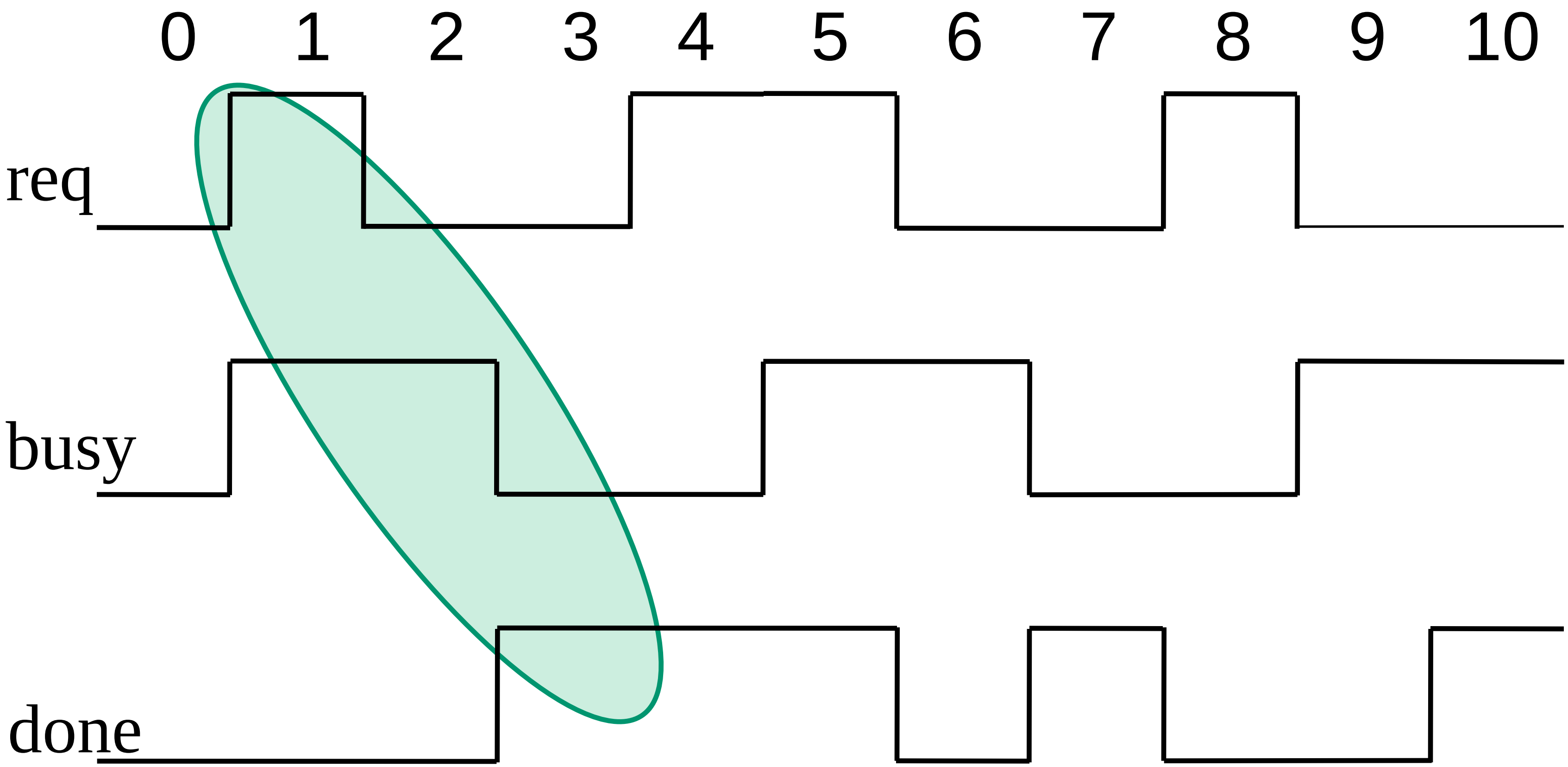
Example

Whenever signal req is asserted then, starting from the next cycle, signal busy must be asserted until signal done is asserted.

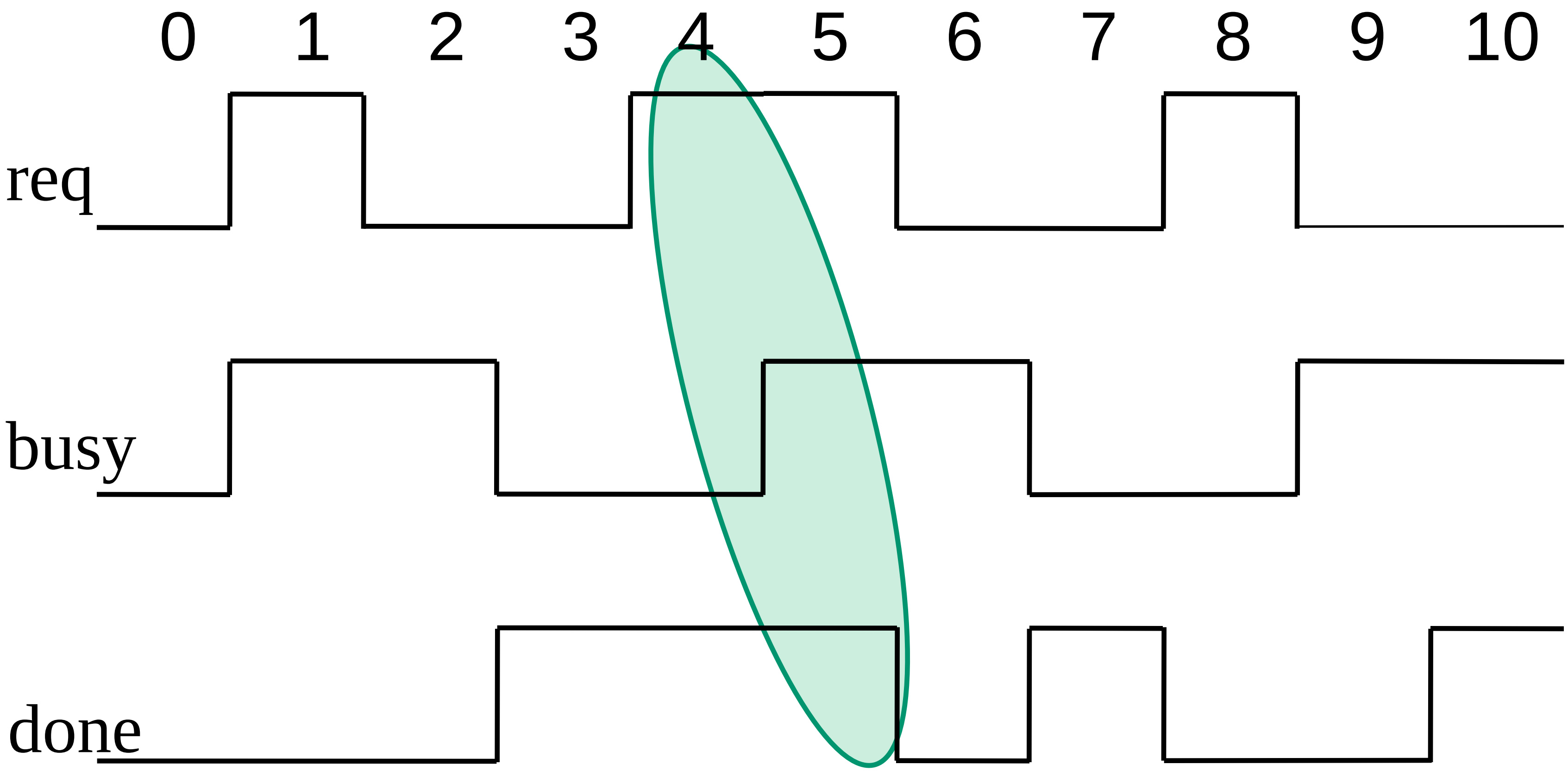
assert always (req -> next (busy until done))



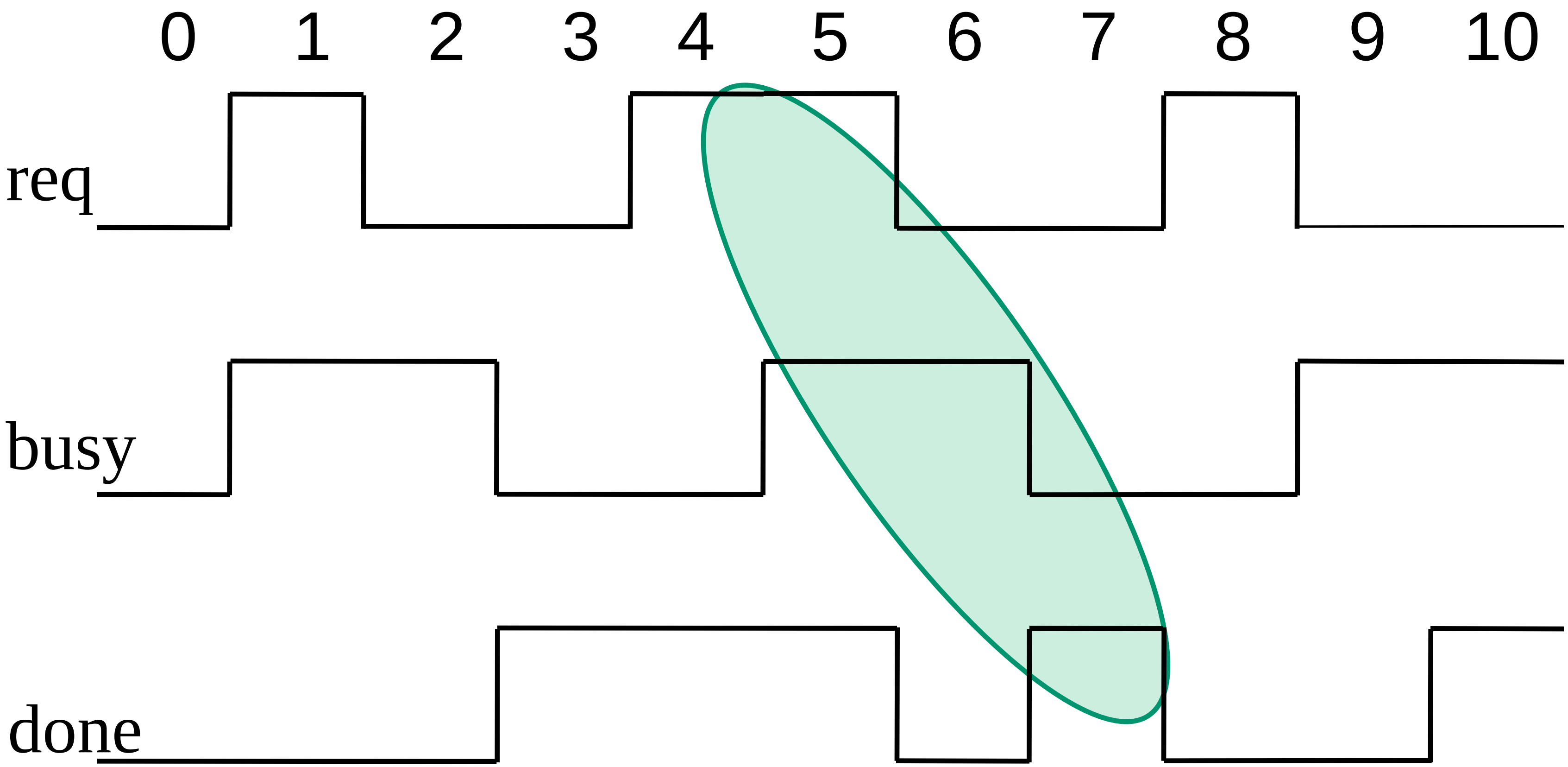
assert always (req -> next (busy until done))



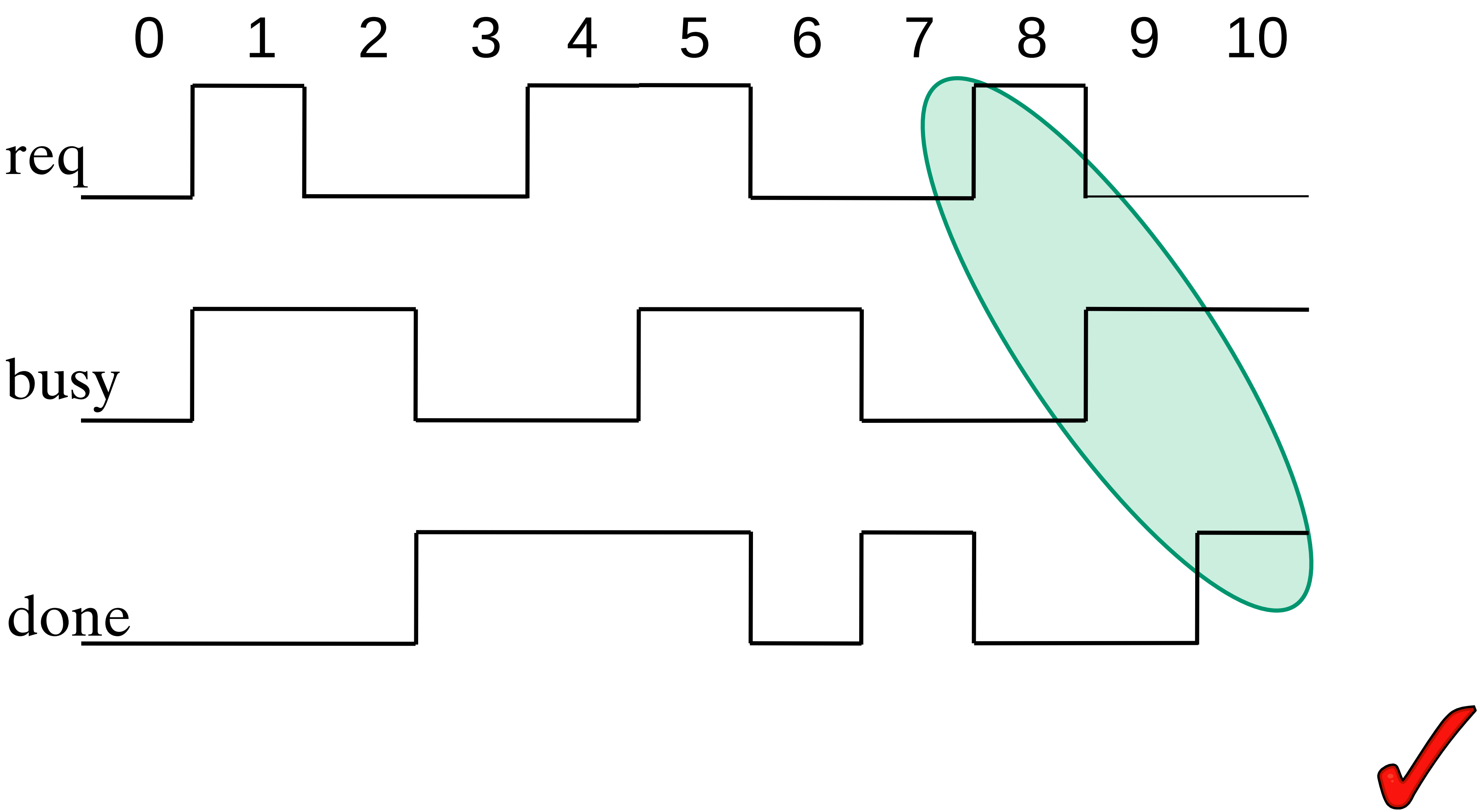
assert always (req -> next (busy until done))



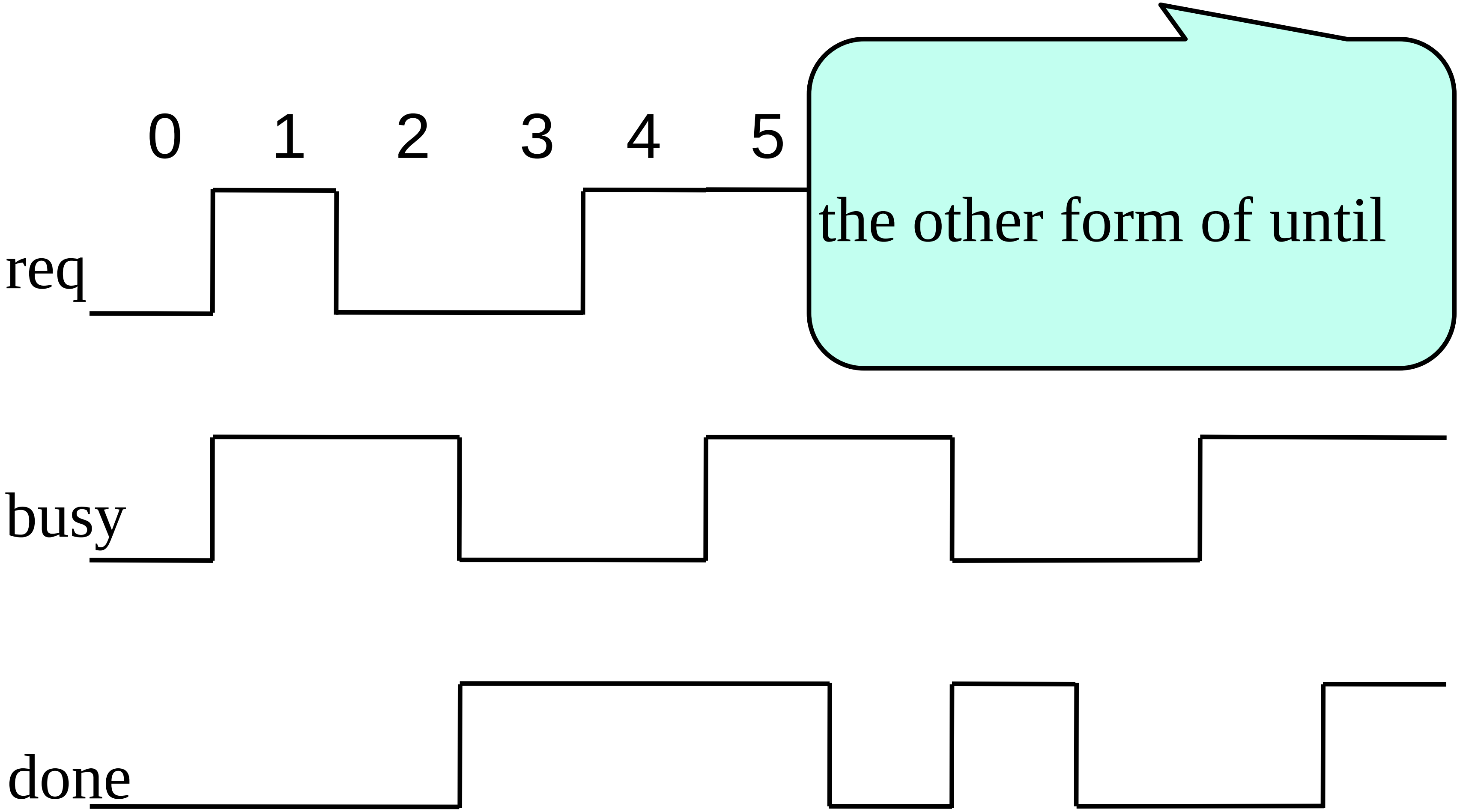
assert always (req -> next (busy until done))



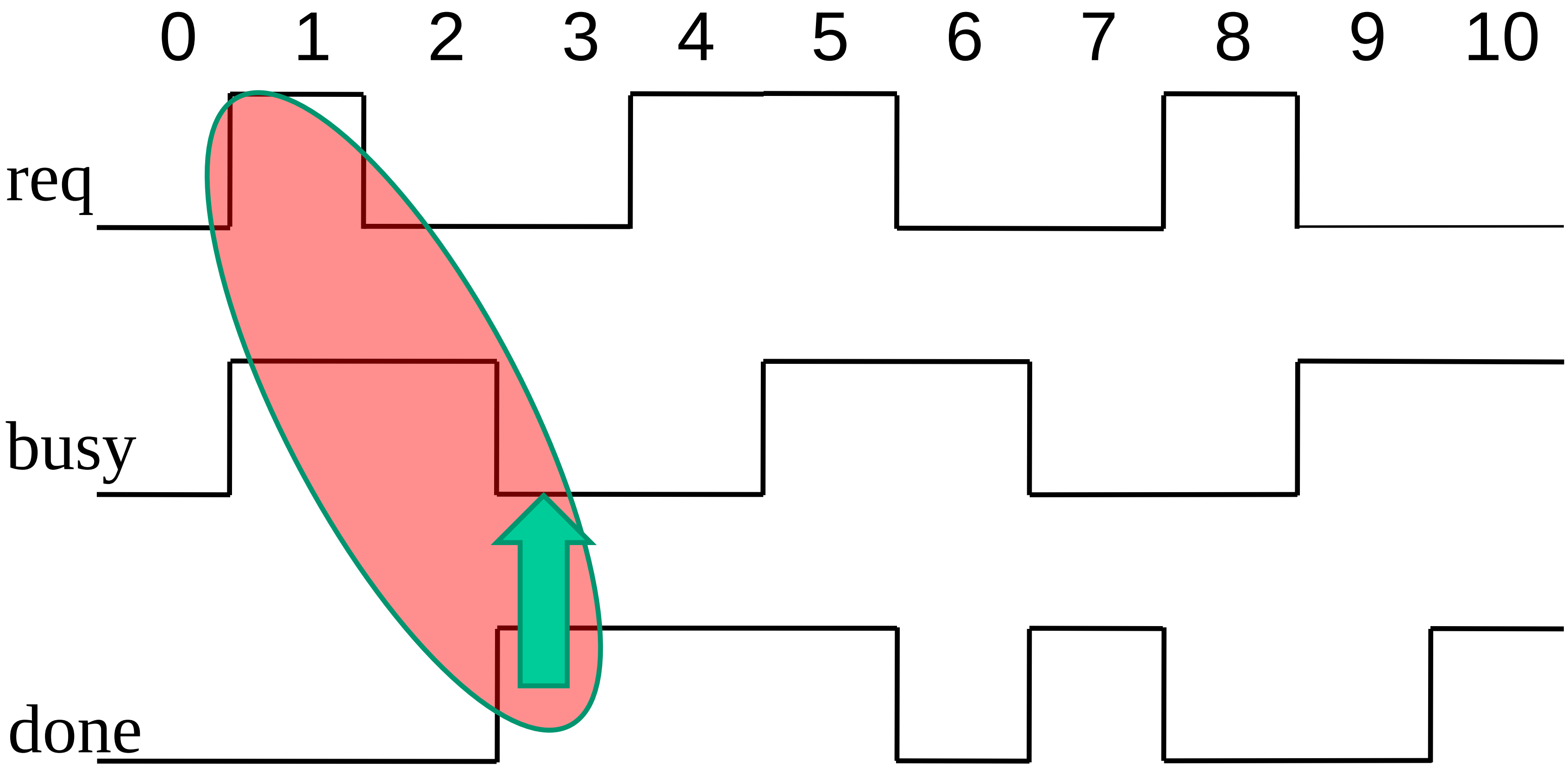
assert always (req -> next (busy until done))



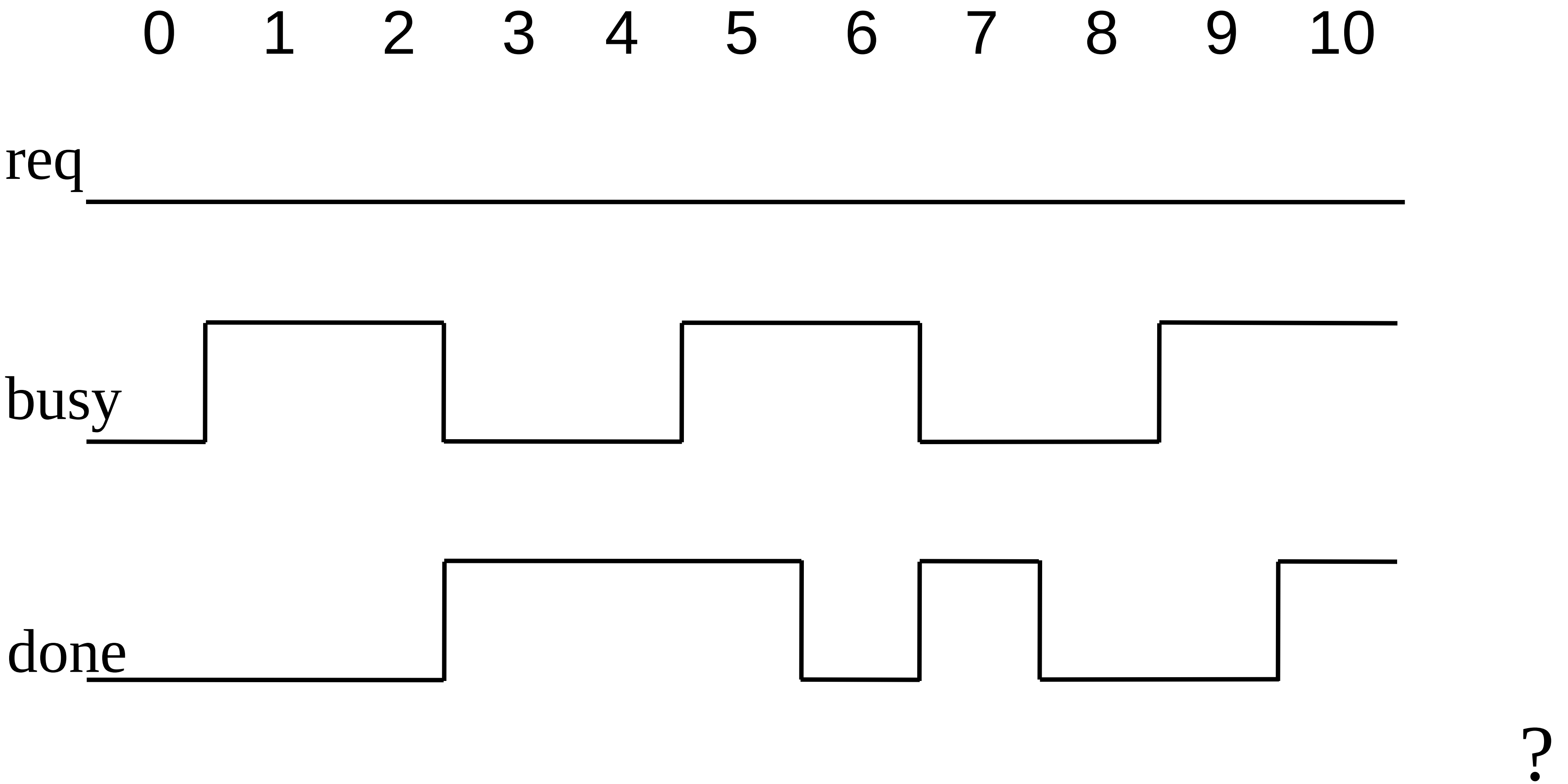
assert always (req -> next (busy until_ done))



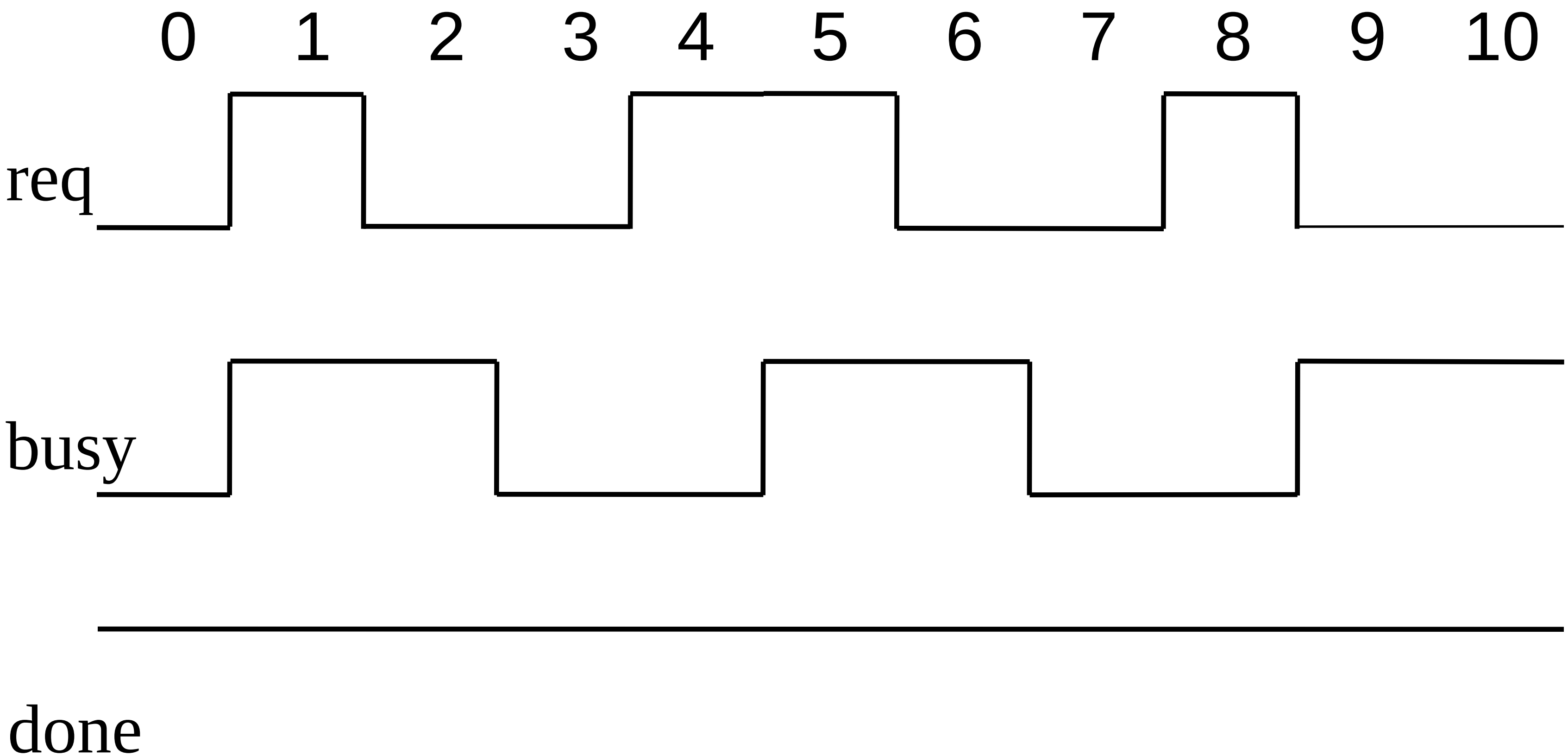
assert always (req -> next (busy until_ done))



Ex: assert always (req -> next (busy until_ done))



Ex: assert always (req -> next (busy until done))



?

Temporal operators

before

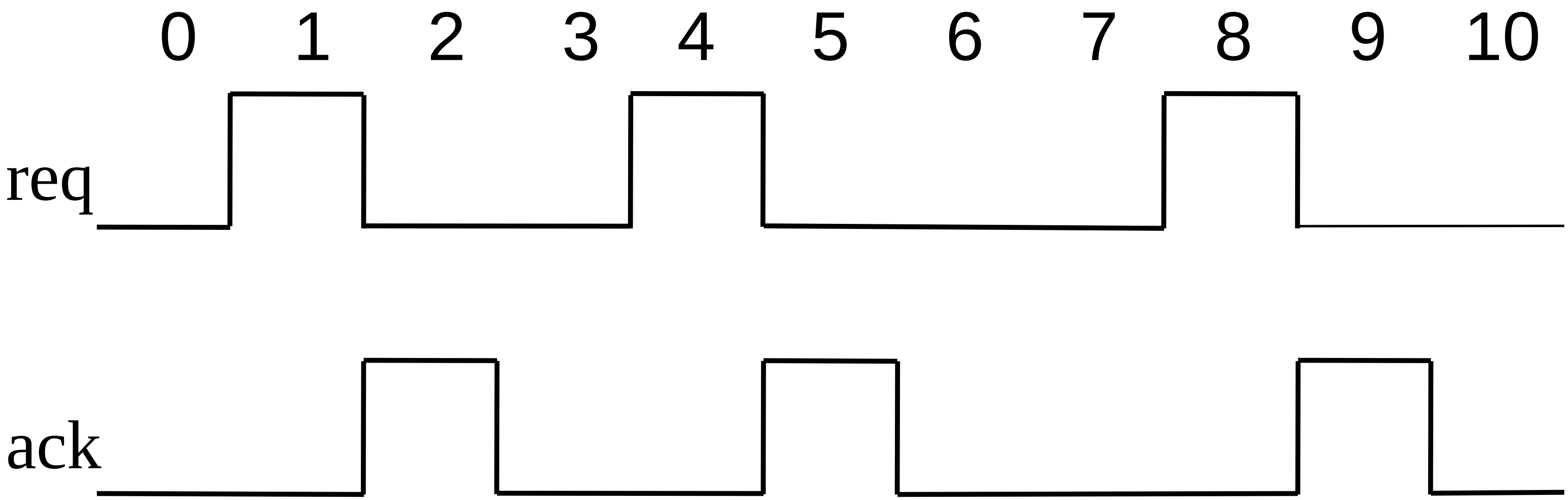
p before q	p must hold at least once strictly before q holds
p before_ q	p holds at least once before or at the same cycle as q holds

Example

Pulsed request signal req

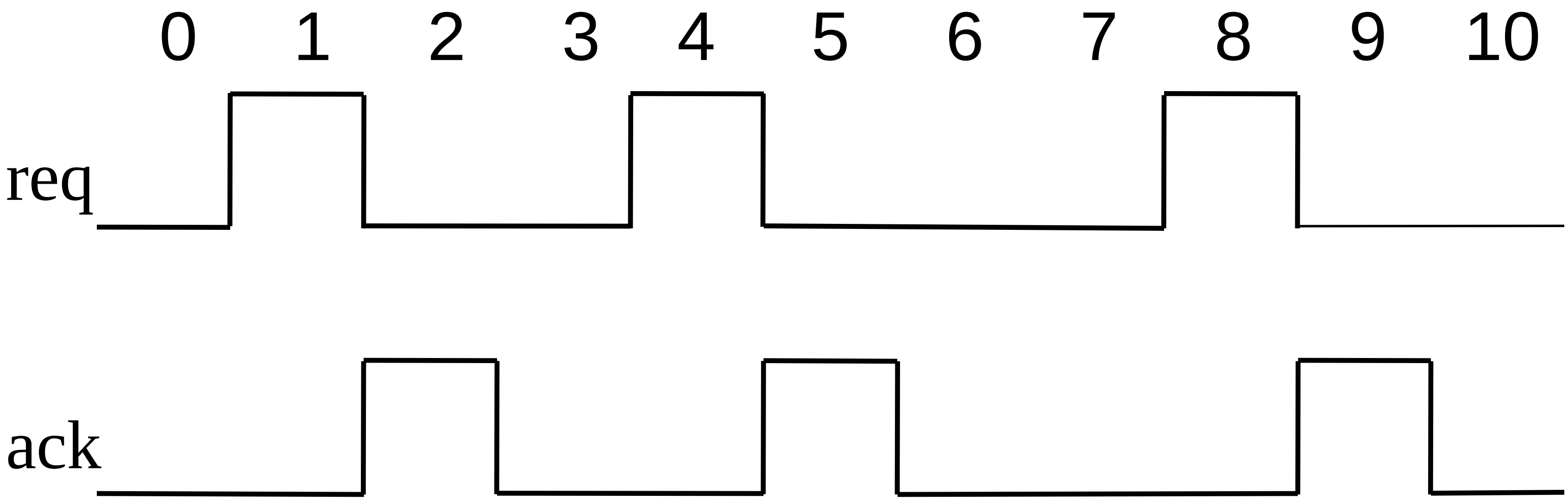
Requirement:

Before we can make a second request, the first must be acknowledged assert always (req -> next (ack before req))

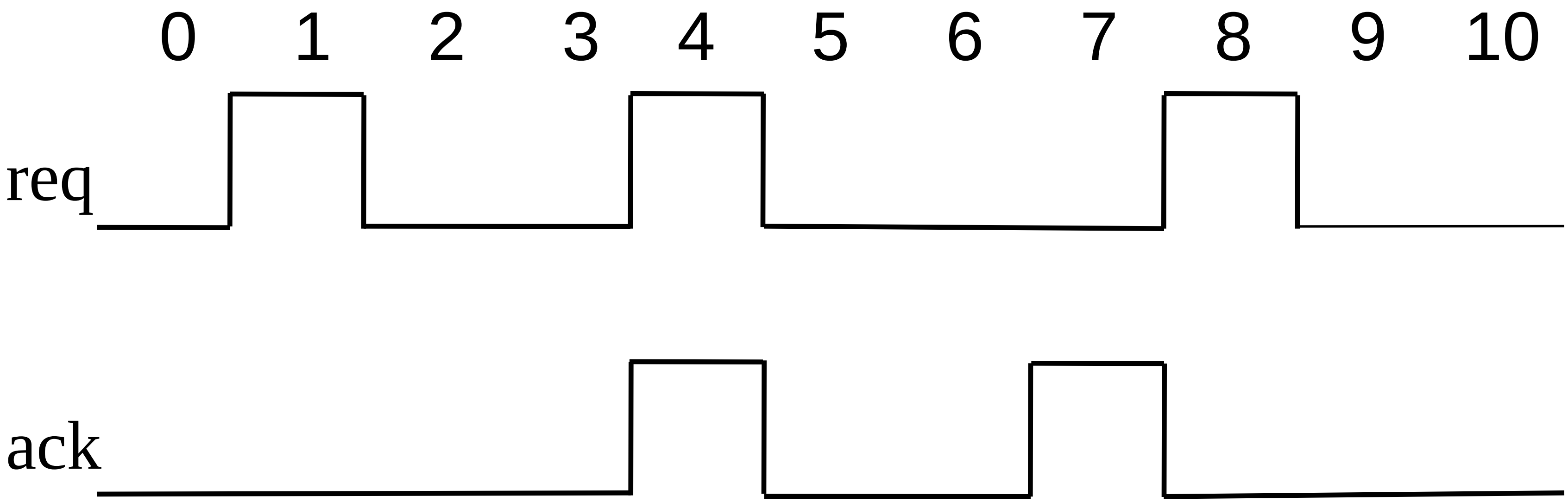


?

assert always (req -> next (ack before req))

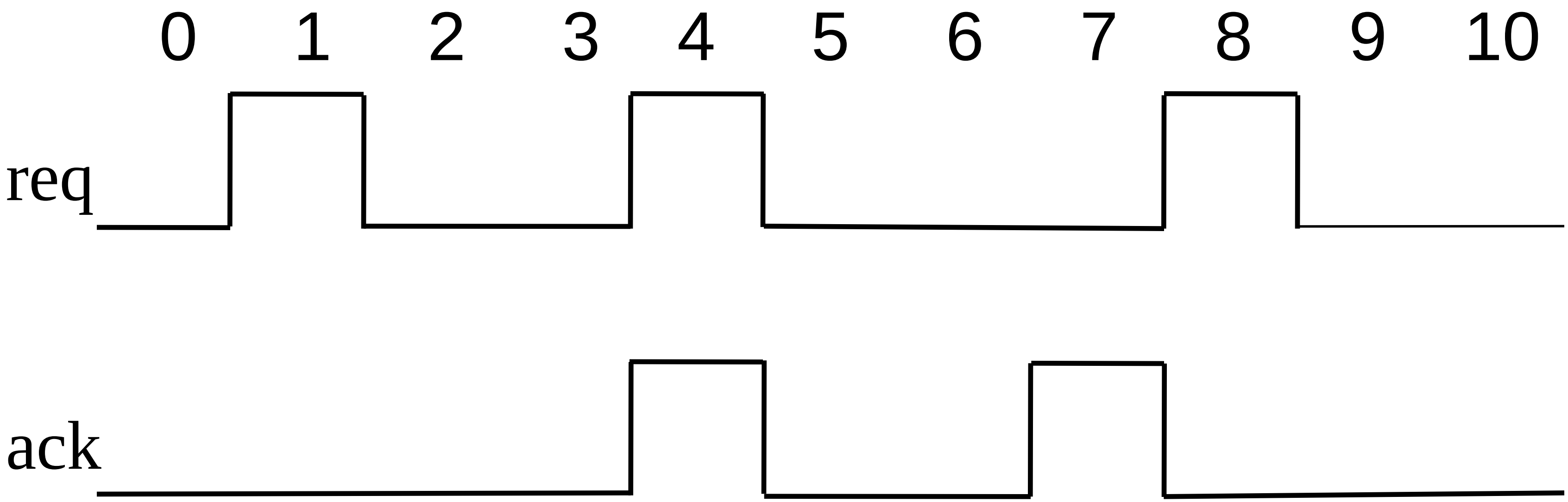


Allow ack simultaneously with next req



assert always (req -> next (ack before_ req))

Questions



1) assert always (req -> next (ack before req)) ?

Questions

2) Would

assert always (req -> (ack before req))

match the original English requirement?

3) What if we want to allow the ack to come not together with the next req but with the req that it is acknowledging?? Write a new property for this.

Next topic

Sequential Extended Regular Expressions

SEREs