Example

Pulsed request signal req

Requirement:

Before can make a second request, the first must be acknowledged

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.

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

Sequential Extended Regular Expressions (SEREs)

(based on source : Dana Fisman and Cindy Eisner, with thanks)

SEREs

Alternative to the temporal operators

(always, next, until, before)

Related to good old regular expressions

A SERE is a property but a property is not necessarily a SERE

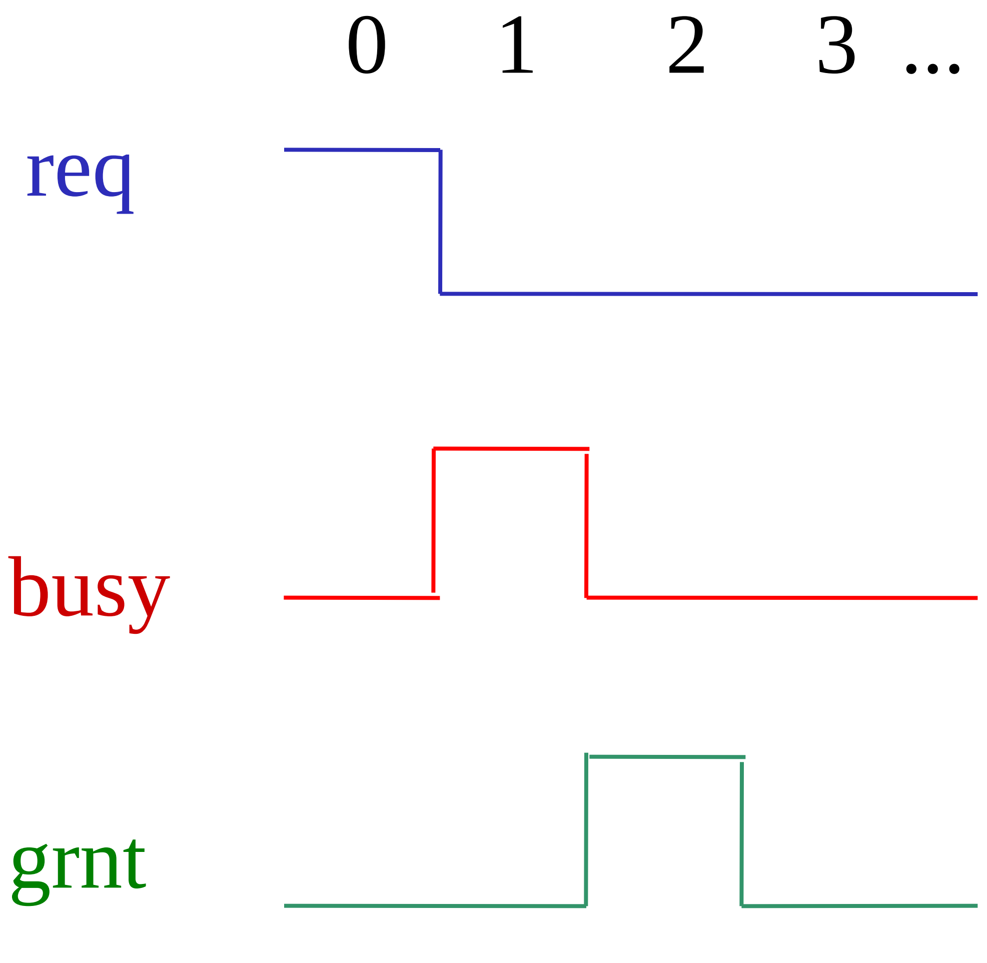
SERE examples

```
assert {req; busy; grnt}
```

req is high on the first cycle, busy on the second, and grnt on the third.

SERE examples

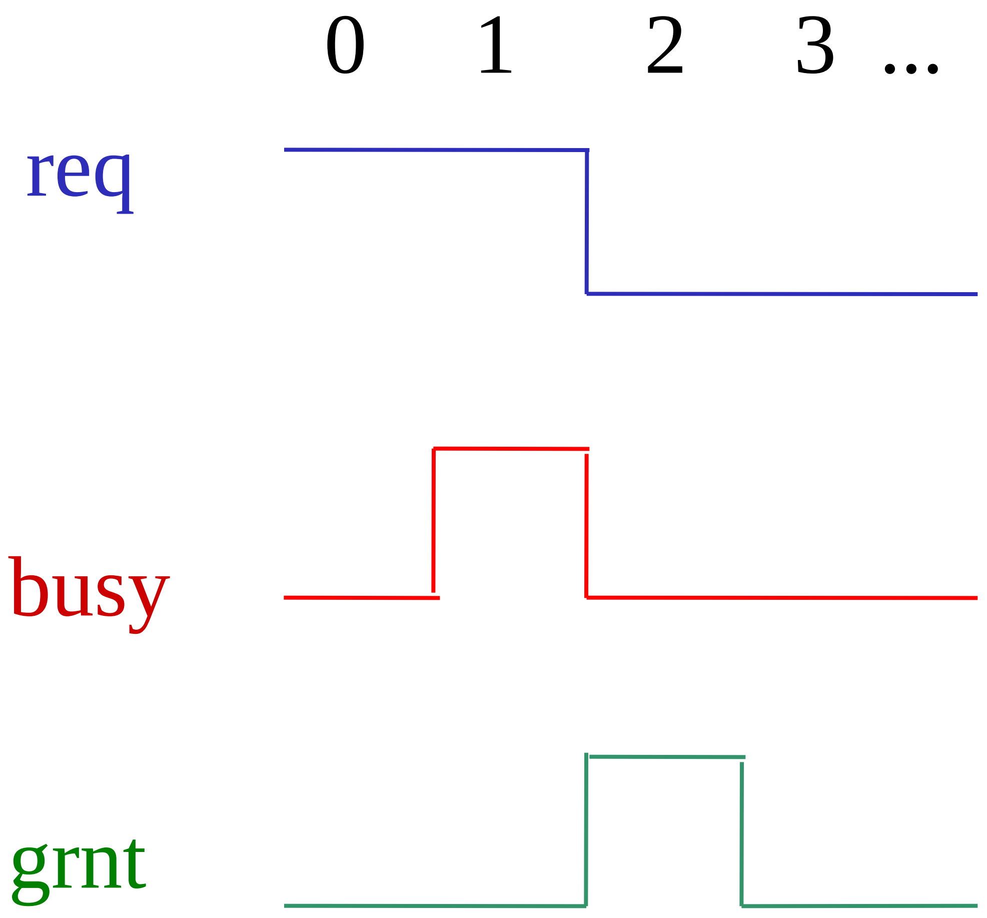
assert {req; busy; grnt}



✓

SERE examples

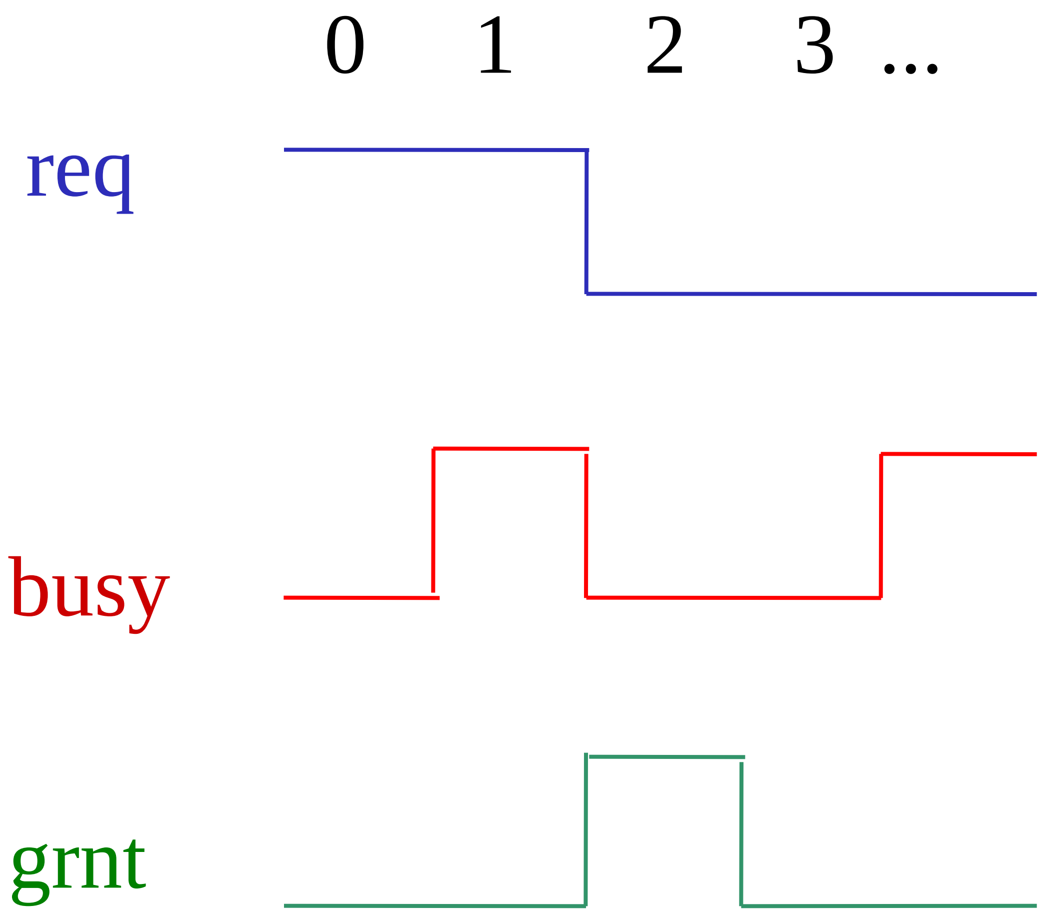
assert {req; busy; grnt}



this too ✓

SERE examples

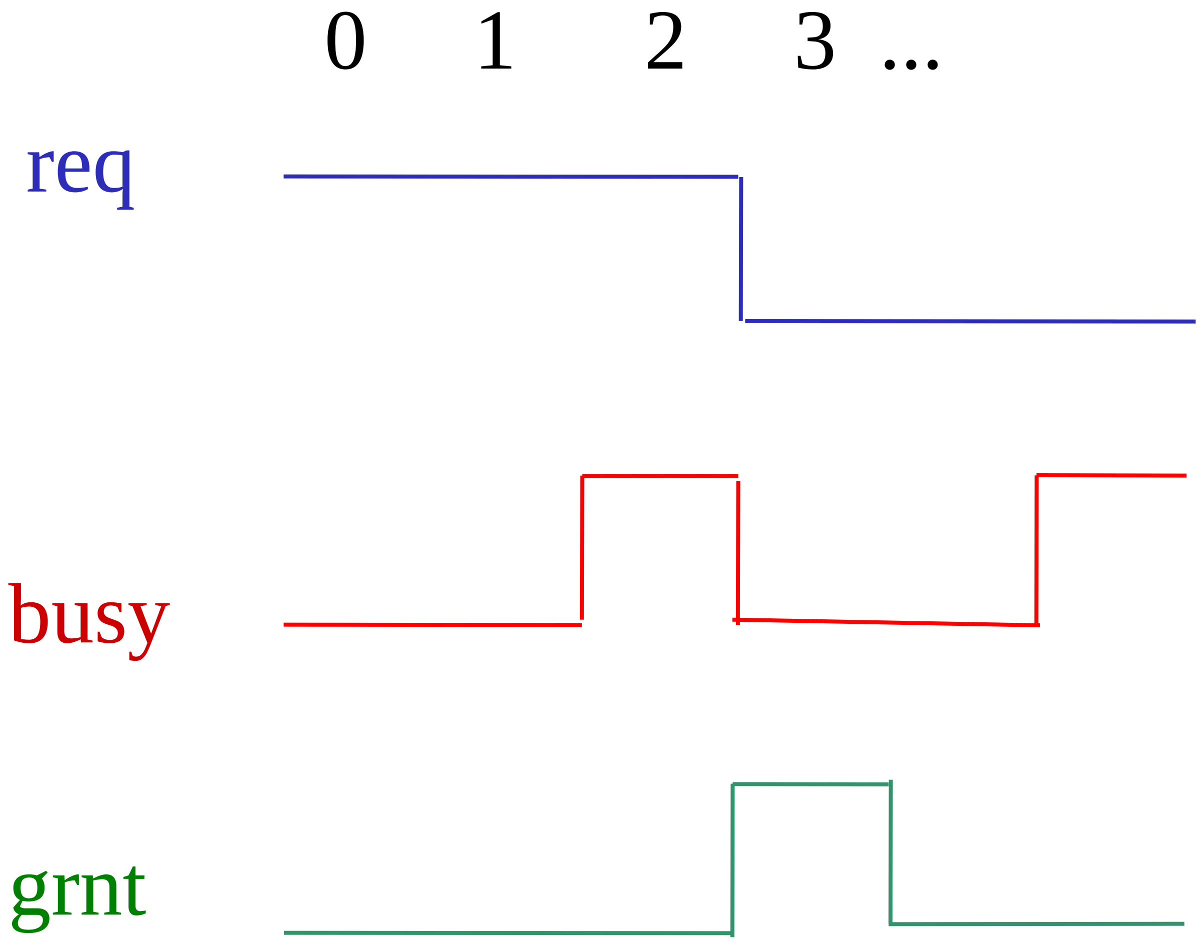
assert {req; busy; grnt}



and this ✓

SERE examples

assert {req; busy; grnt}

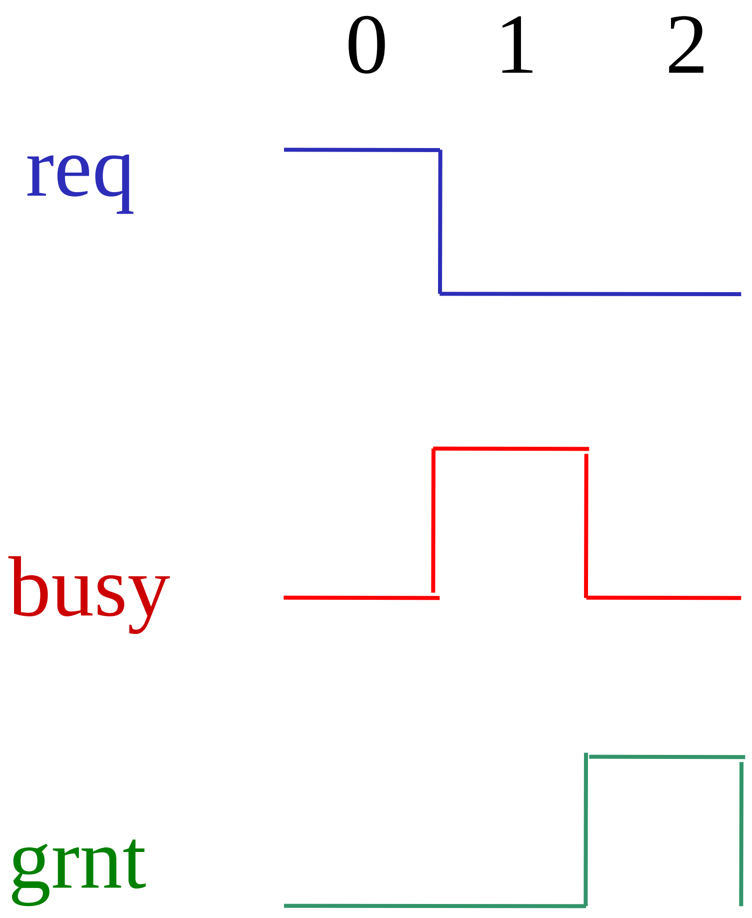


but not this

Why?

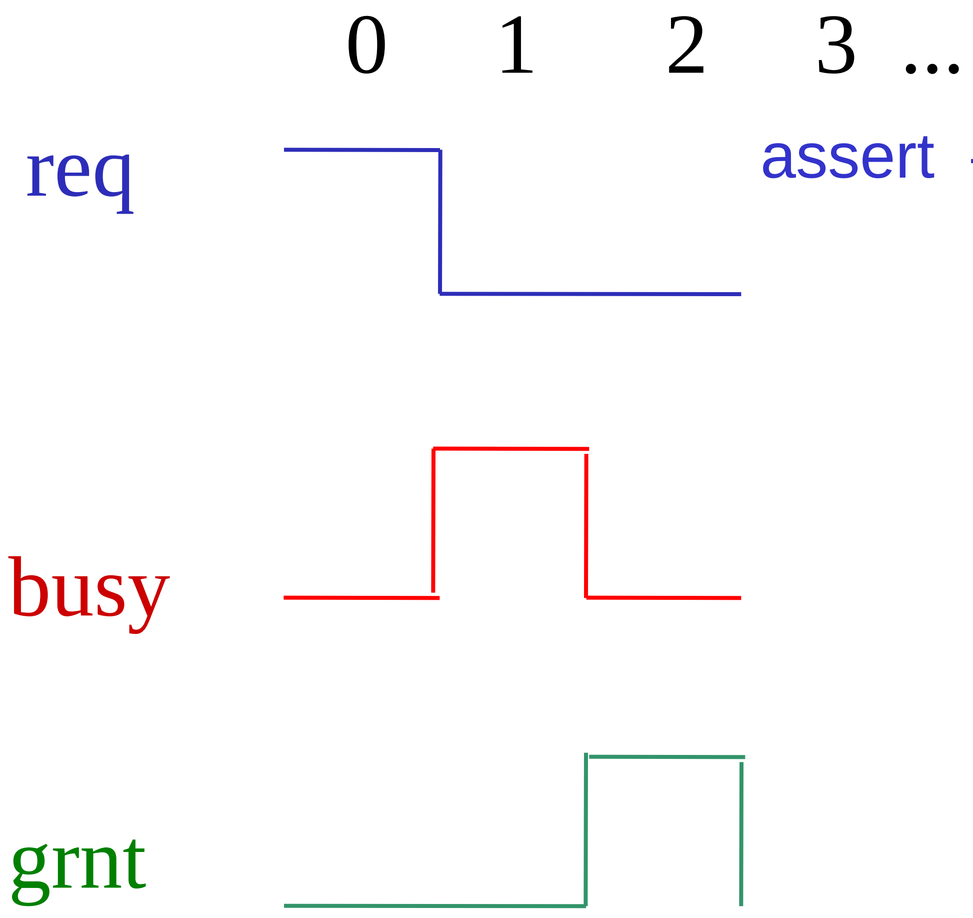
SERE examples

Specify only traces that start like this?



SERE examples

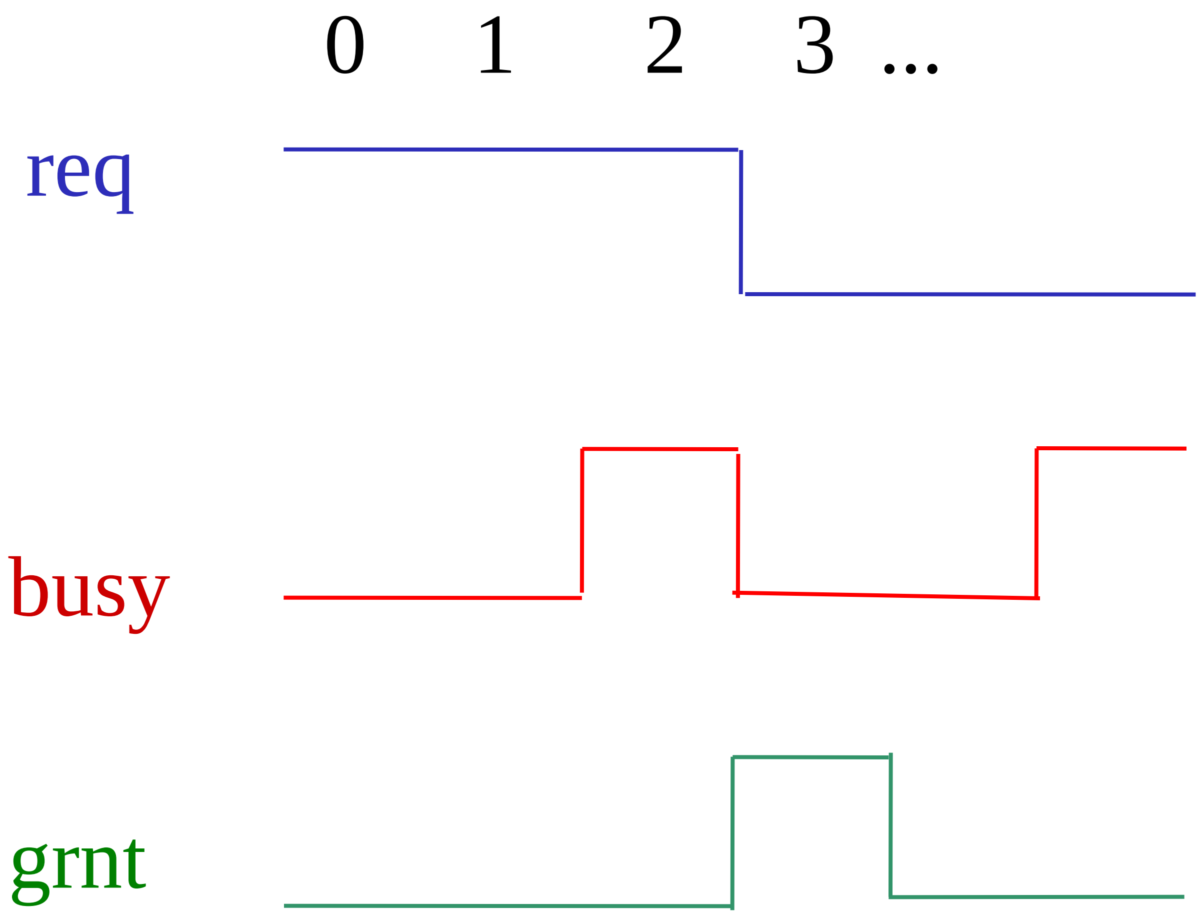
Specify only traces that start like this?



assert {req and not busy and not grnt;
not req and busy and not grnt;
not req and not busy and grnt}

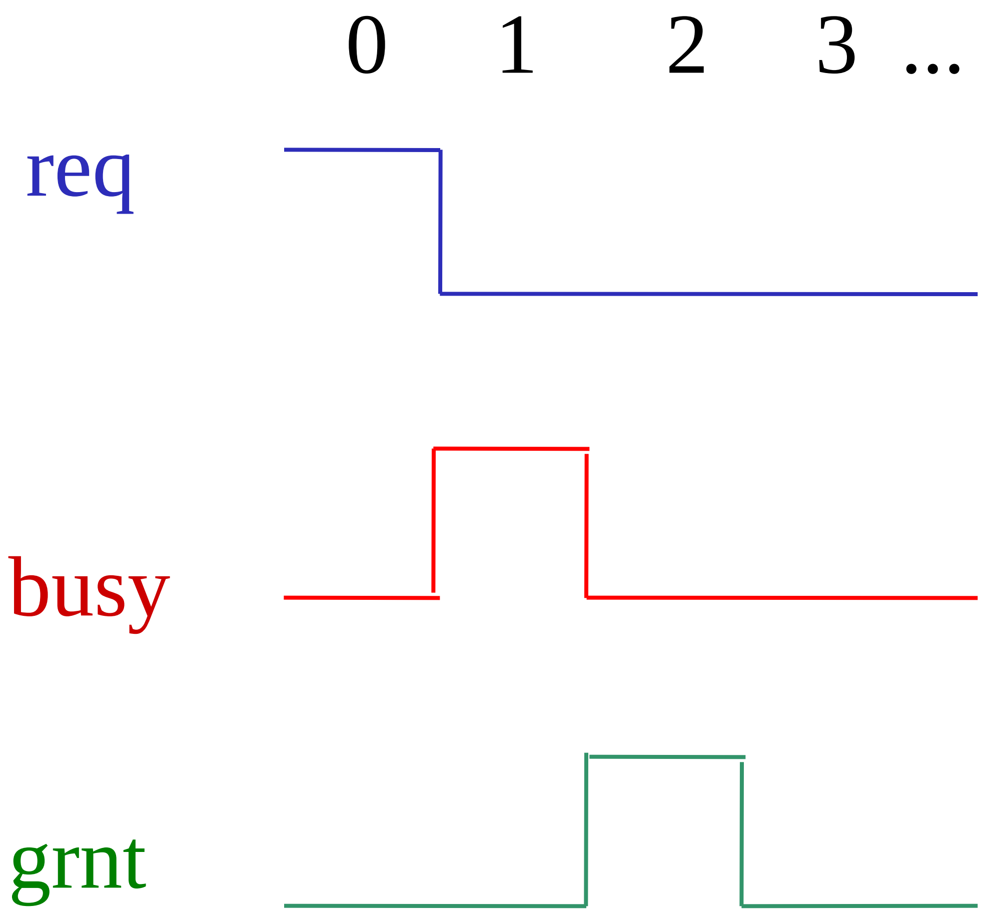
assert {[*]; req; busy; grnt}

[*] means skip zero or more cycles



✓

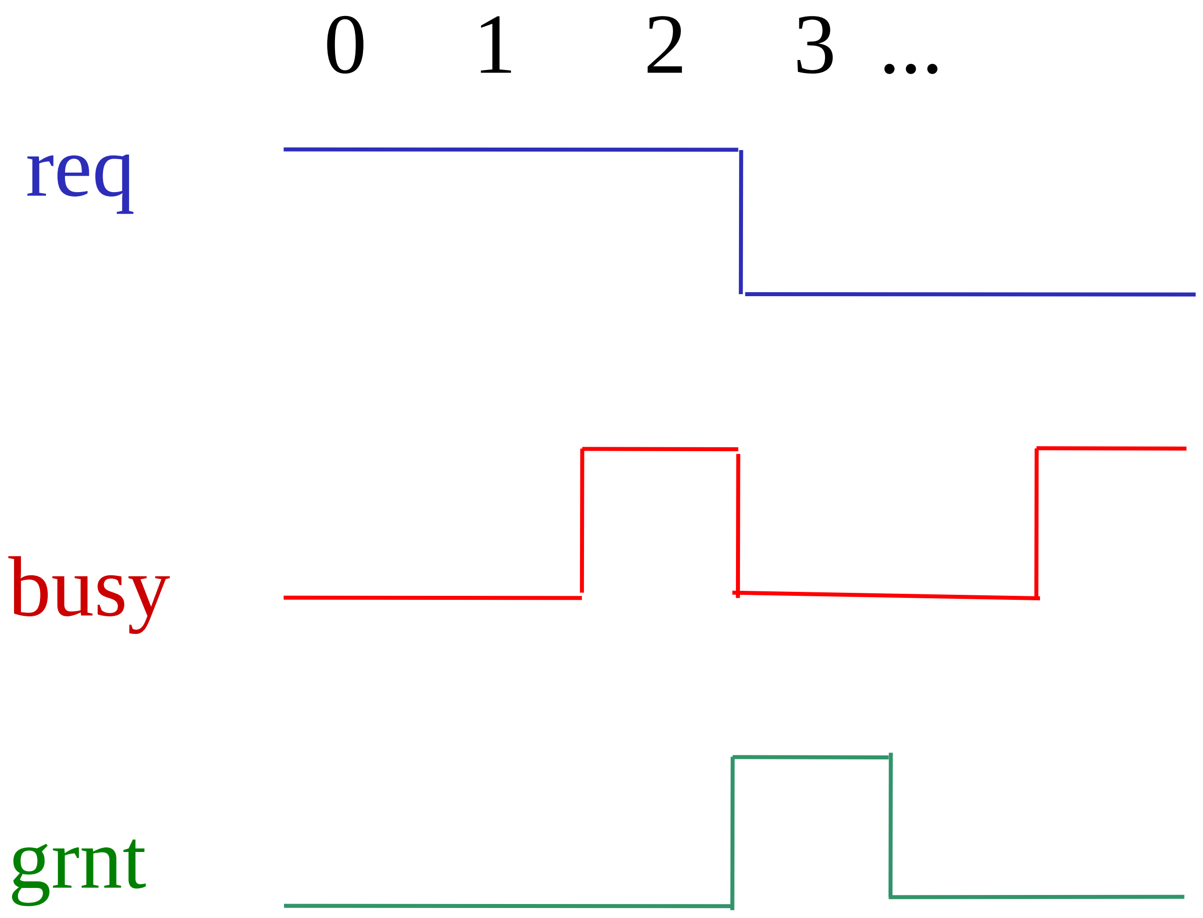
assert {[*]; req; busy; grnt}



so our original trace is still in the set described

✓

assert {true; req; busy; grnt}



says
req;busy;grnt
should start after
one cycle

constrains only
cycles 1,2,3

✓

{true[*4]; req; busy; grnt}   4 repetitions

{true[+]; req; busy; grnt}   true[+] = [+]

one or more trues

true[*] = [*]

```
{[*]; req; busy[*3 to 5]; grnt}
```

at least 3 and at most 5 busys

```
{[*]; req; {b1;b2}[*]; grnt}
```

```
{[*]; req; {b1;b2;b3}[*7]; grnt}
```

subsequences can also be repeated

Yet more SERE repetition operators

{[*]; req; busy[=3]; grnt}

3 busys, not necessarily in consecutive cycles, between req and grnt

(and with possible ”padding” before and after

occurrences of busy)

Applies only to Boolean expressions

(example trace later)

Yet more SERE repetition operators

{[*]; req; busy[->3]; grnt}

3 busys, not necessarily in consecutive cycles, between req and grnt

(and with possible ”padding” before and after busys, but NOT after last occurrence of busy)

Applies only to Boolean expressions

Called the ”goto repetition operator”

[->1] can be written [->]

&&

Simultaneous subsequences

Same length, start and end together

```
{start; a[*]; end} && {not abort[*]}
```

”length matching and”

&

Both sequences should be matched, starting at the same clock cycle

But they don’t need to be the same length

```
{p1_done[->]  &  p2_done[->] &  p3_done[->]}
```

Describes an interval in which p1, p2 and p3 all get done, but not necessarily simultaneously

|

One of the subsequences should be matched (or)

Don’t need to be the same length

```
{request;
{rd; not c_r; not dne[*]} | {wr; not c_w; not dne[*]};
dne}
```

Fancier properties at last!

SEREs are themselves properties
Properties are also built from subproperties

{SERE1} |=> {SERE2} is a property

If a trace matches SERE1, then its continuation should match SERE2

Fancier properties at last!

SEREs are themselves properties

Properties are also built from subproperties

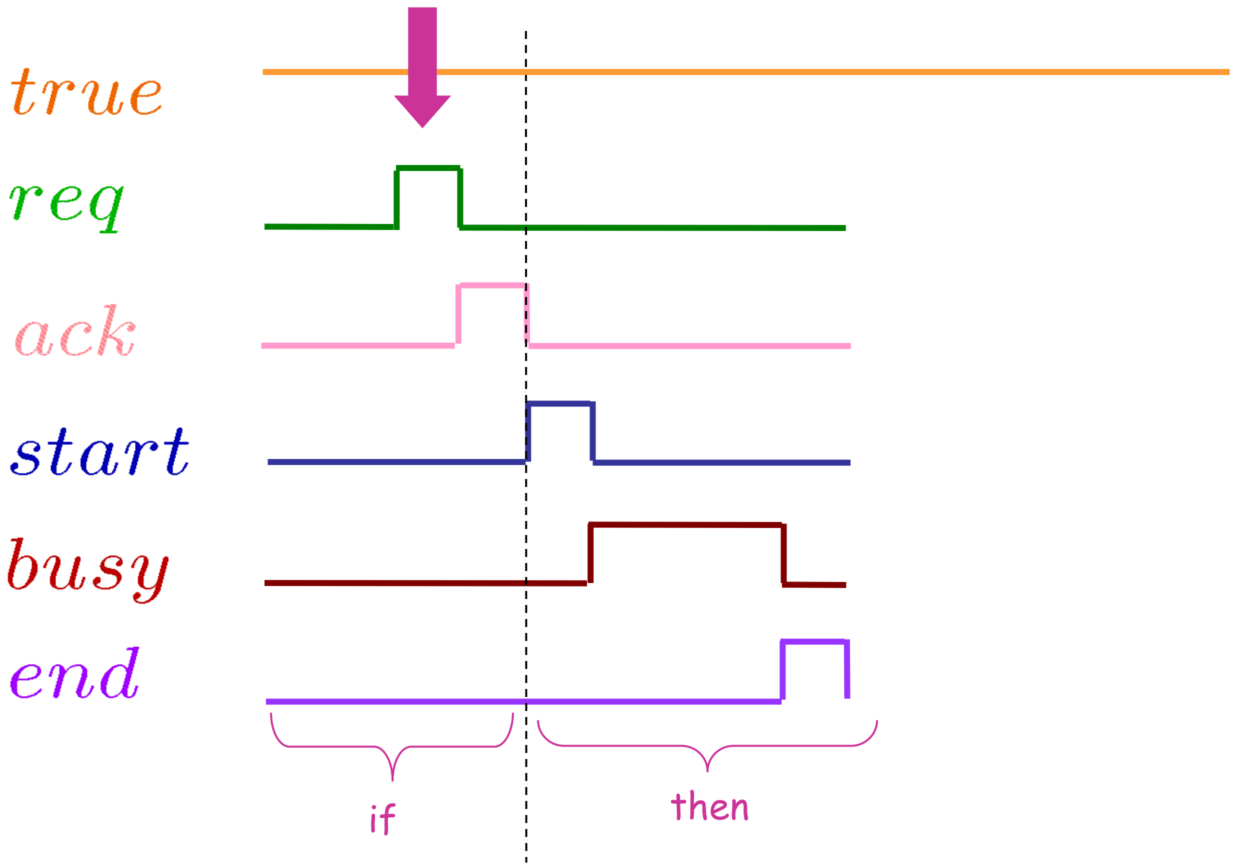
{SERE1} |=> {SERE2} is a property

If a trace matches

continuation should

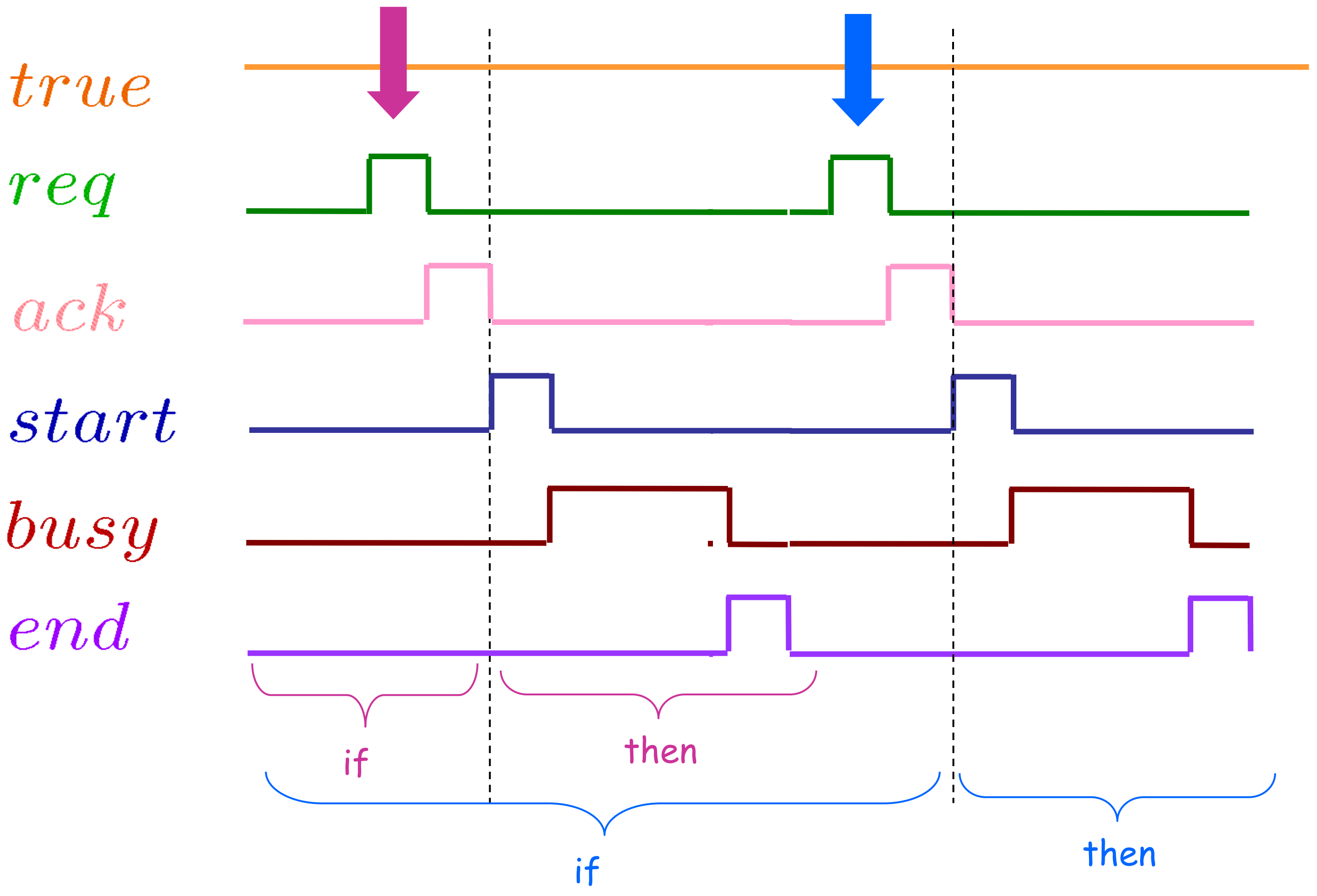
Non-overlapping suffix implication

{true[*]; req; ack} |=> {start; busy[*]; end}



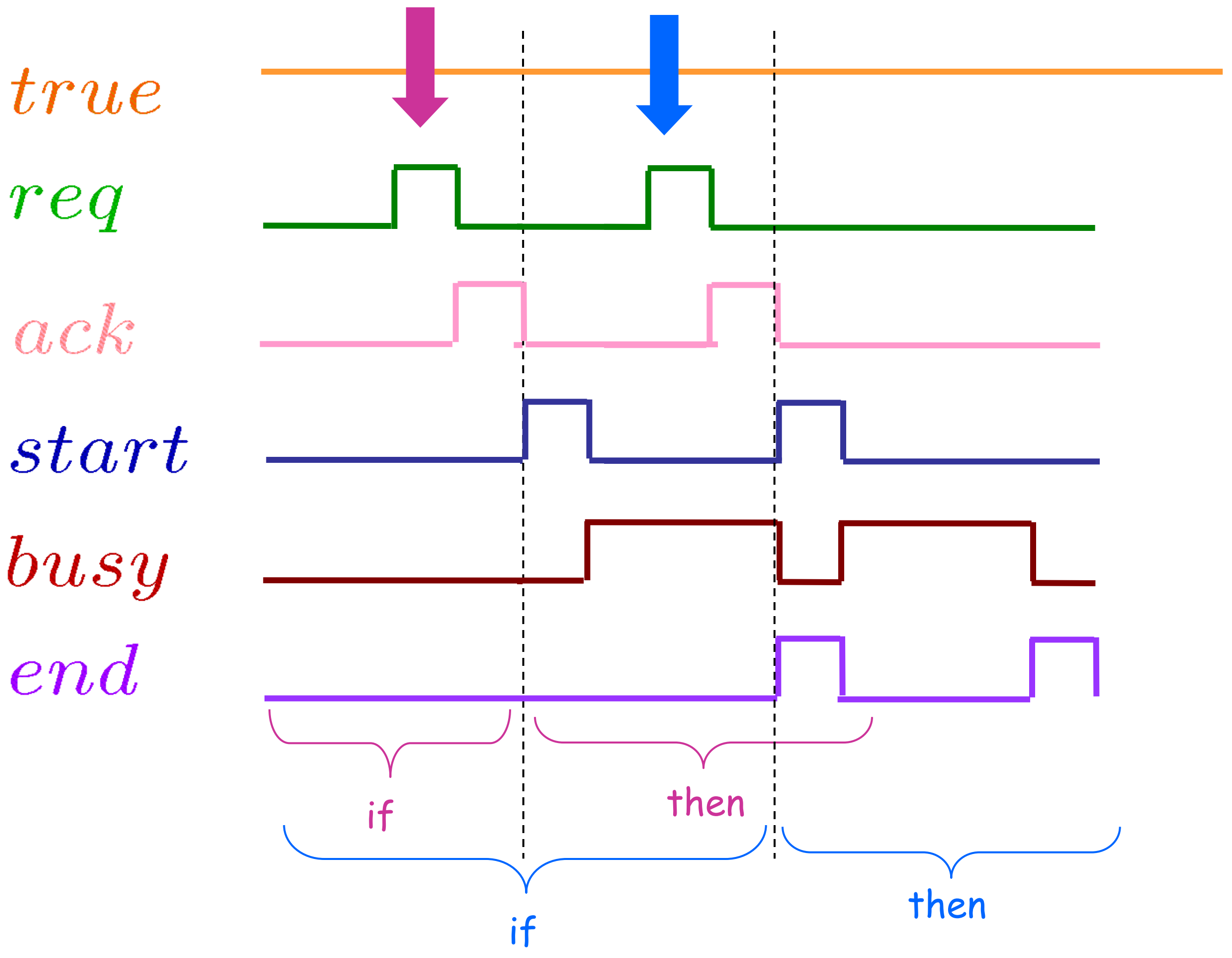
Not just the first req; ack

{true[*]; req; ack} |=> {start; busy[*]; end}

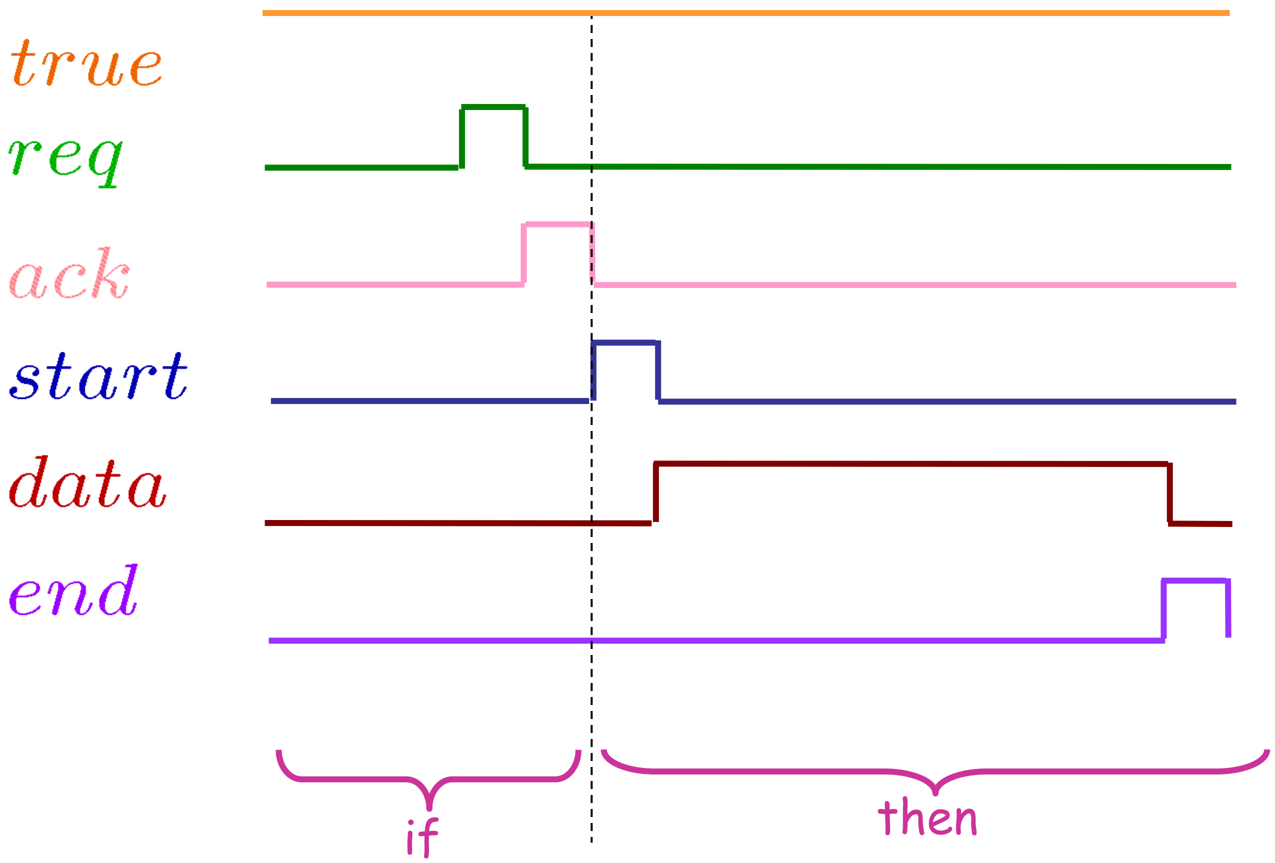


Overlap also possible!

{true[*]; req; ack} |=> {start; busy[*]; end}

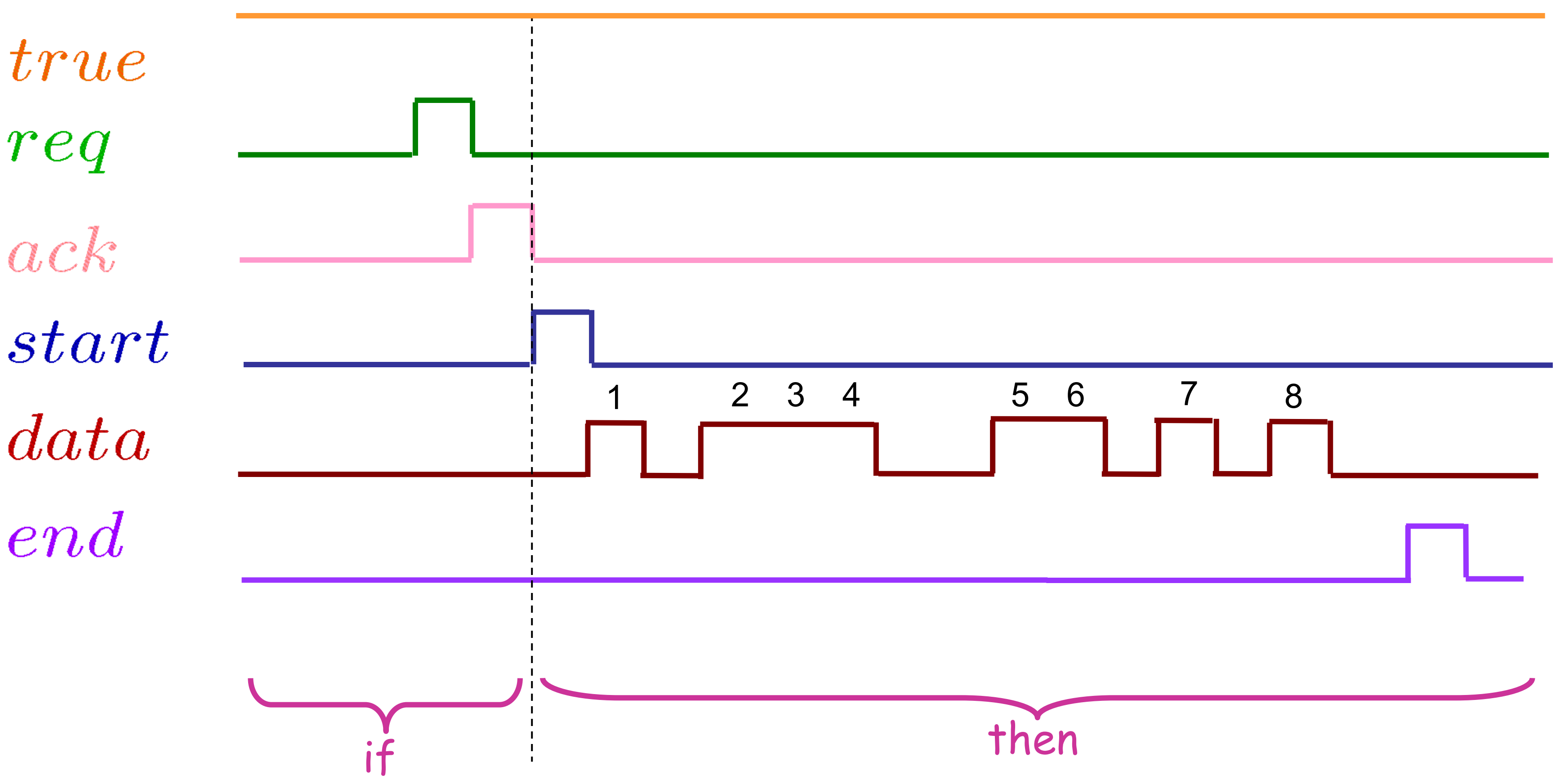


{true[*]; req; ack} |=> {start; data[*]; end}



Can check for data in non-consecutive cycles

{true[*]; req; ack} |=> {start; data[=8]; end}



A form of implication

{SERE1} |=> {SERE2}

If a trace matches SERE1, then its continuation should match SERE2

A form of implication

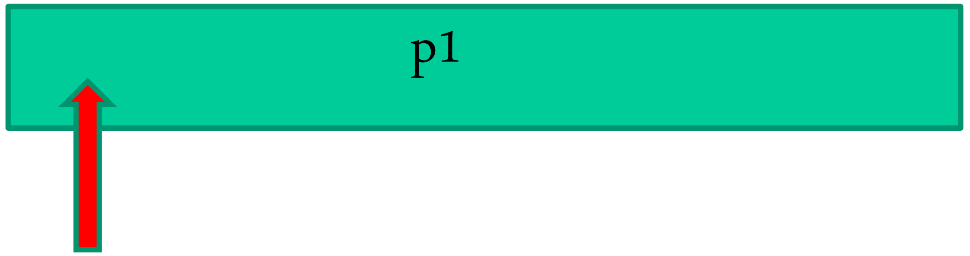
v. similar to logical implication

(with same ”false implies everything” trap)

Difference is timing relationship between if and then parts

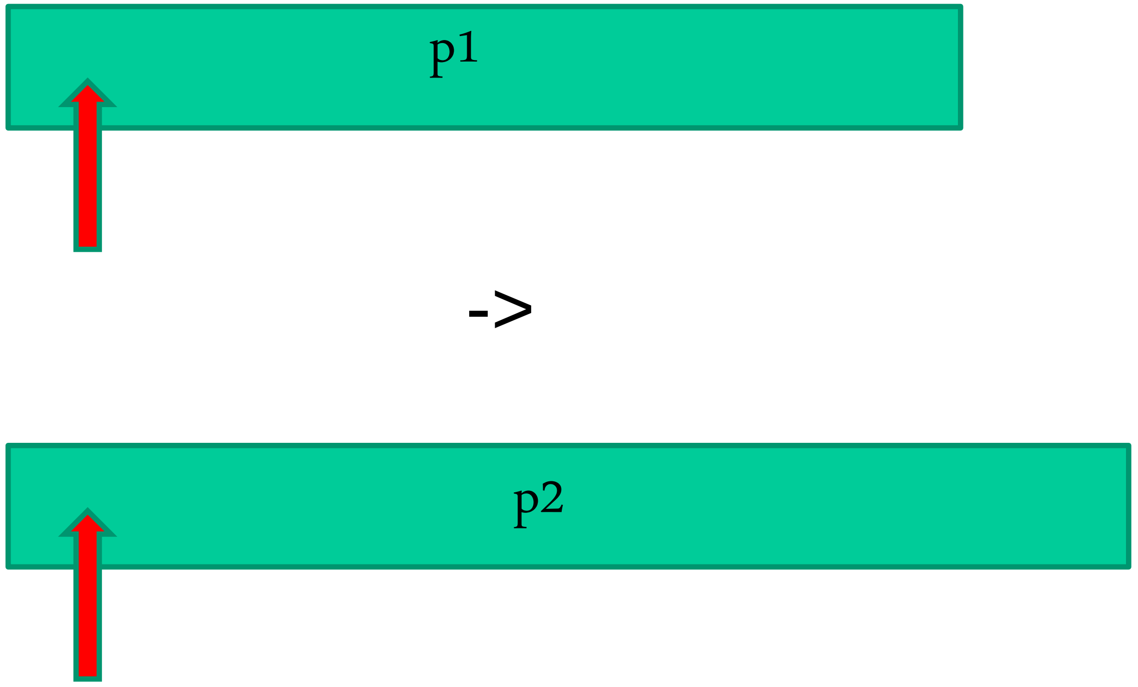
logical implication

p1 -> p2



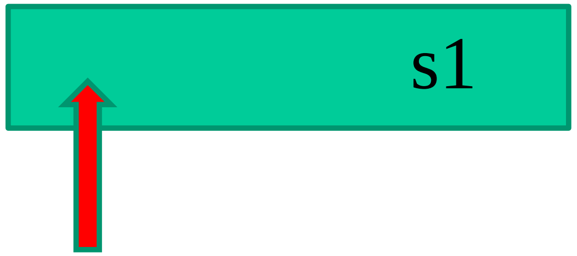
logical implication

p1 -> p2



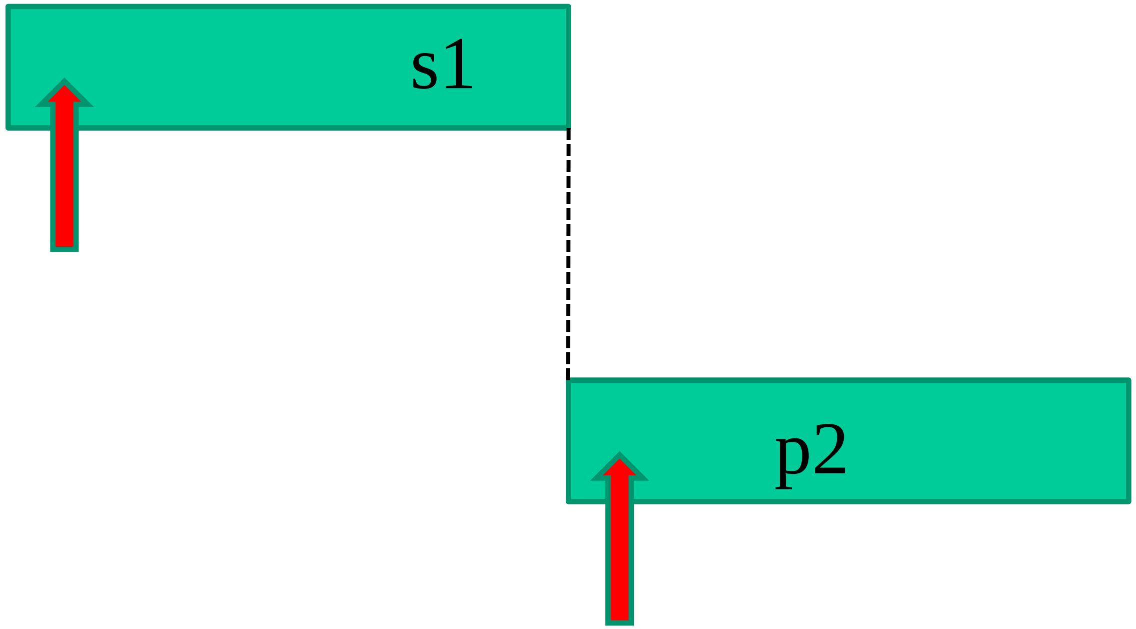
suffix implication

s1 |=> p2



suffix implication

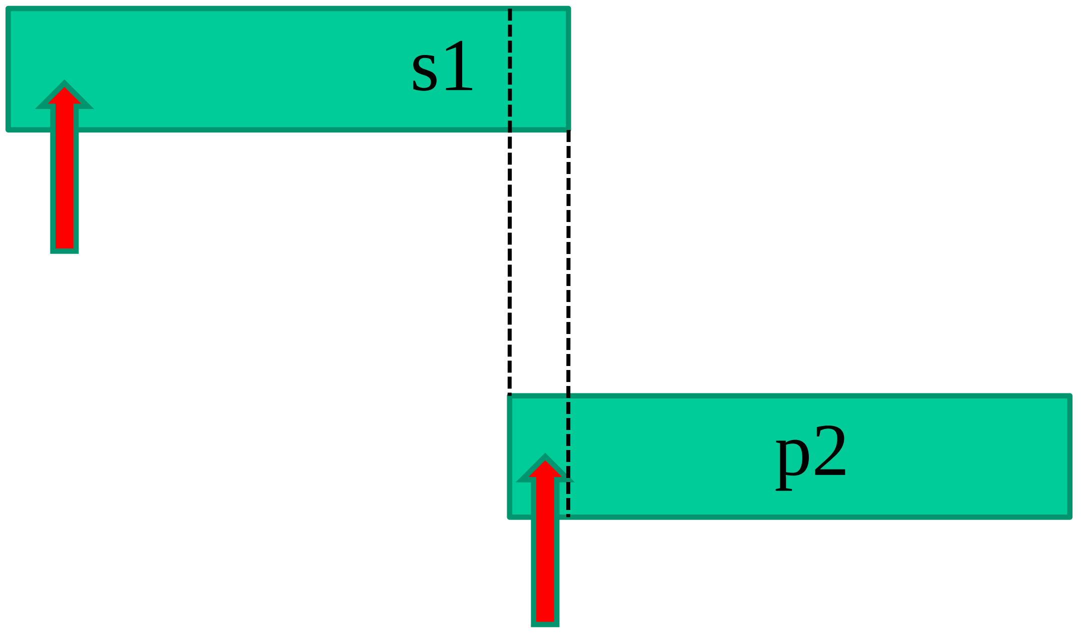
s1 |=> p2



|=>

overlapping suffix implication

s1 |-> p2



|->

Another form of implication

{SERE1} |-> {SERE2}

If a trace matches SERE1, then SERE2 should be matched, starting from the last element of the trace matching SERE1

So there is one cycle of overlap in the middle

Example

```
{[*]; start; busy[*]; end} |-> {success; done}
```

If signal start is asserted, signal end is asserted at the next cycle or later, and in the meantime signal busy holds, then success is asserted at the same time as end is, and in the next cycle done is asserted

Example

{[*]; {{start; c[*]; end}&&{not abort[*]}}} |-> {success}

If there is no abort during {start;c[*];end}, success will be asserted with end

Question

Can you express one of the suffix implications in terms of the other?

Don’t forget always!

{[*];s} |-> p (SERE style)

equivalent to

always {s} |-> p (LTL style)

(and sim. for |=>)

PSL has a small core and the rest is syntactic sugar, for example

b[+] can be defined in terms of b[*]

How?

b[=i] can be defined in terms of [*] and [*i]

How?

```
b[=i]        =  {not b[*]; b}[*i]  ;  not b[*]
```

Q: define b[->k] in similar style

See formal semantics in LRM

PSL

Regular expressions (plus some operators)

+

Linear temporal logic (LTL)

+

Lots of syntactic sugar

+ (optional)

Computation tree logic (CTL)

Example revisited

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])]]])

In PSL (with 8 for 2)

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

```
always({strt; {get[=8]}&&{kill[=0]}}
                    |=>   {{put[=8]}&&{end[=0]}})
```

PSL

Seems to be reasonably simple, elegant and concise!

Jasper’s Göteborg based team (with Koen Claessen) have helped to define and simplify the formal semantics.

See the LRM (or IEEE standard) and also the paper in FMCAD 2004   (links page)