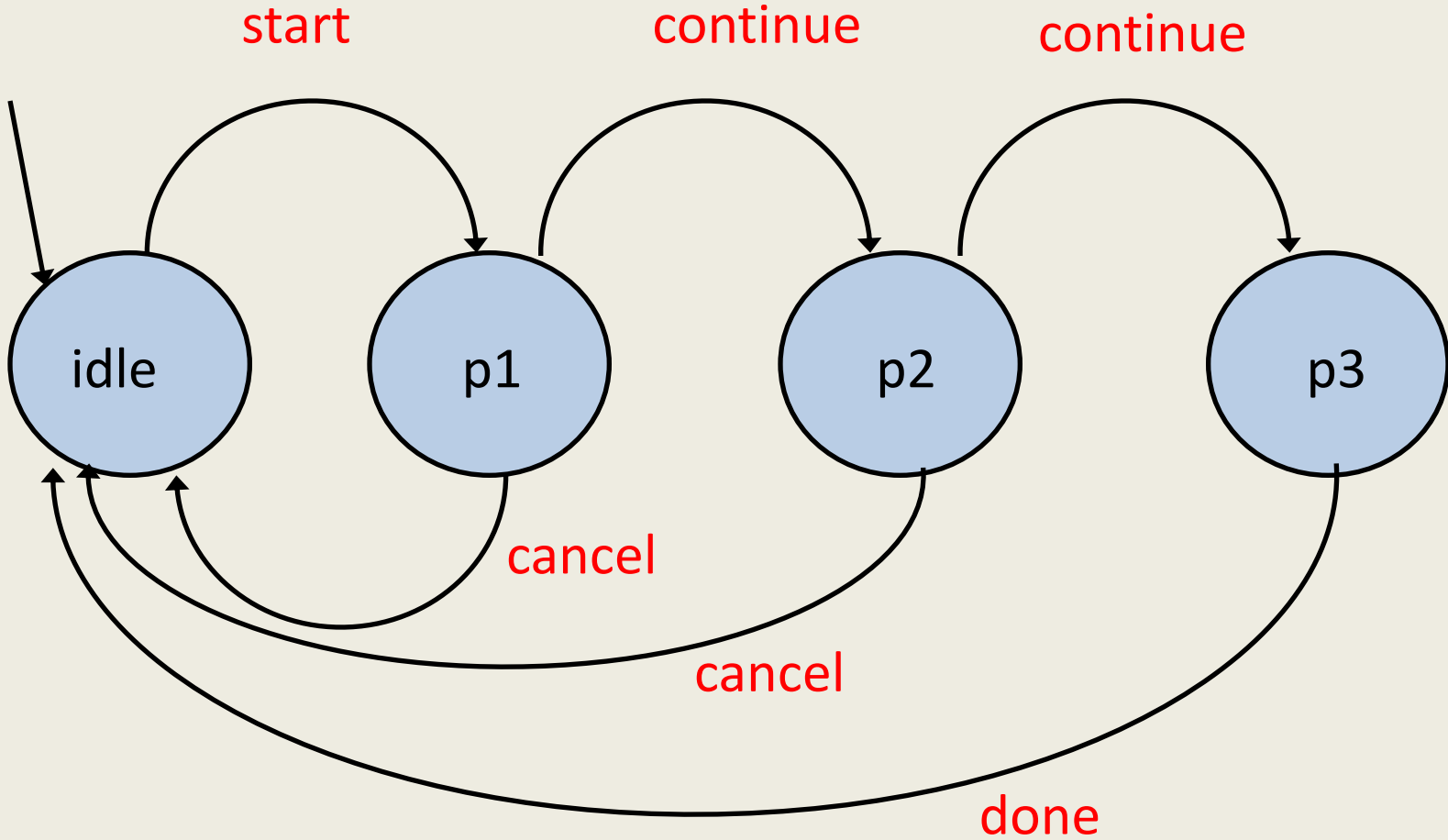


More on PSL

some examples,
some pitfalls

FSM



Low level assertions

```
assert always ((state = idle and start) ->  
  next (state = p1));
```

```
assert always ((state = idle and not start) ->  
  next (state = idle));
```

```
assert always ((state = p1 and continue) ->  
  next (state = p2));
```

and so on... one for each transition
good, but very localised

Low level assertions

```
assert always ((state = idle and start) ->  
  next (state = p1));
```

```
assert always ((state = p1 and start) ->  
  next (state = idle));
```



Bit-vector

```
assert always ((state = p1 and continue) ->  
  next (state = p2));
```

and so on... one for each transition
good, but very localised

Low level assertions

```
assert always ((state = idle and start) ->  
  next (state = p1));
```



constant

```
assert always ((state = idle a  
  next (state = idle));
```

```
assert always ((state = p1 and continue) ->  
  next (state = p2));
```

and so on... one for each transition
good, but very localised

Low level assertions

```
assert always ((state = idle and start) ->  
  next (state = p1));
```

```
assert always ((state = idle and not start) ->  
  next (state = idle));
```

```
assert always ((state = p1 and co  
  next (state = p2));
```



Implicit self-loop

and so on... one for each transition
good, but very localised

Higher level assertion

assert always (not (state = idle) ->
eventually! (state = idle))

Note: not a safety property!

Will also likely need to link the state machine to the system that it is controlling and check that the desired functionality is achieved

Message: try to raise level of abstraction of properties (while keeping them short and simple)

Example: simple bus interface spec (1)

1. 2 commands, read and write (with corresponding signals)
2. Command can be issued only after requesting the bus, indicated by a pulsed assertion of signal `bus_req`, and receiving a grant, indicated by the assertion of signal `gnt` one cycle after the assertion of `bus_req`
3. If the bus was not requested, it shouldn't be granted
4. Command is issued the cycle following receipt of grant
5. Either a read or a write can be issued, not both simultaneously

Example: simple bus interface spec (2)

6. Reads and writes come with an address,
on `addr[7 downto 0]`, that should be valid in the
following cycle
7. Address validity is indicated by signal `addr_valid`
8. If a read is issued, then one pulse of data on
`data_in[63 downto 0]` is expected the following cycle
9. If a write is issued, then one pulse of data on
`data_out[63 downto 0]` is expected the following cycle
10. Valid read data is indicated by `data_in_valid` and valid
write data by `data_out_valid`

Example: simple bus interface

low level checks

2, 4. assert always ((read or write) ->
ended(bus_req; gnt; true))

Built in function

Returns true when the SERE has just ended

Example: simple bus interface

low level checks

3. assert always (not bus_req -> next (not gnt))

Example: simple bus interface

low level checks

5. assert never (read and write)

Example: simple bus interface

low level checks

part of 6,7.

assert always ((read or write) -> next addr_valid)

assert always (not (read or write)
-> next (not addr_valid))

Example: simple bus interface

low level checks

10.

assert always (read -> next data_in_valid)

assert always (not read -> next (not data_in_valid))

assert always (write -> next data_out_valid)

assert always (not write -> next (not data_out_valid))

Example: simple bus interface

low level checks

Have checked the protocol

but not mentioned the addr, data_in or
data_out buses

Need to think about overall functionality as
well as low level details

Example: simple bus interface

high level checks

Let's assume two input signals `get_data` and `put_data` indicating that a read or write is needed

Assume also we have a way to recognise, at the assertion of `get_data` or `put_data`, the data that needs to be read or written

(from address `get_addr[7 downto 0]` to `read_buffer[63 downto 0]` or

from `write_buffer[63 downto 0]` to address `put_addr[7 downto 0]`)

Assume also a suitable memory

Example: simple bus interface

high level checks

assert forall ADR[7 downto 0] in boolean:

always ((get_data and

get_adr[7 downto 0] = ADR[7 downto 0])

->

eventually!

(read_buffer[63 downto 0] = mem[ADR[7 downto 0]]))

Notes:

have made some assumptions e.g. about memory not changing after read

included to show some of the fancier PSL constructs and use of bus structures

Main message

Write both low level and high level checks

Low level checks will be easier to write – often transcribed from spec.

High level specs consider desired functionality, which may be implicit in the spec. Hard to write but high pay-off

For one approach to a methodology for use of PSL, see the Prosyd Eu project web page (www.prosyd.org)

Contains many interesting examples both small and large (including the following example)

Common PSL errors

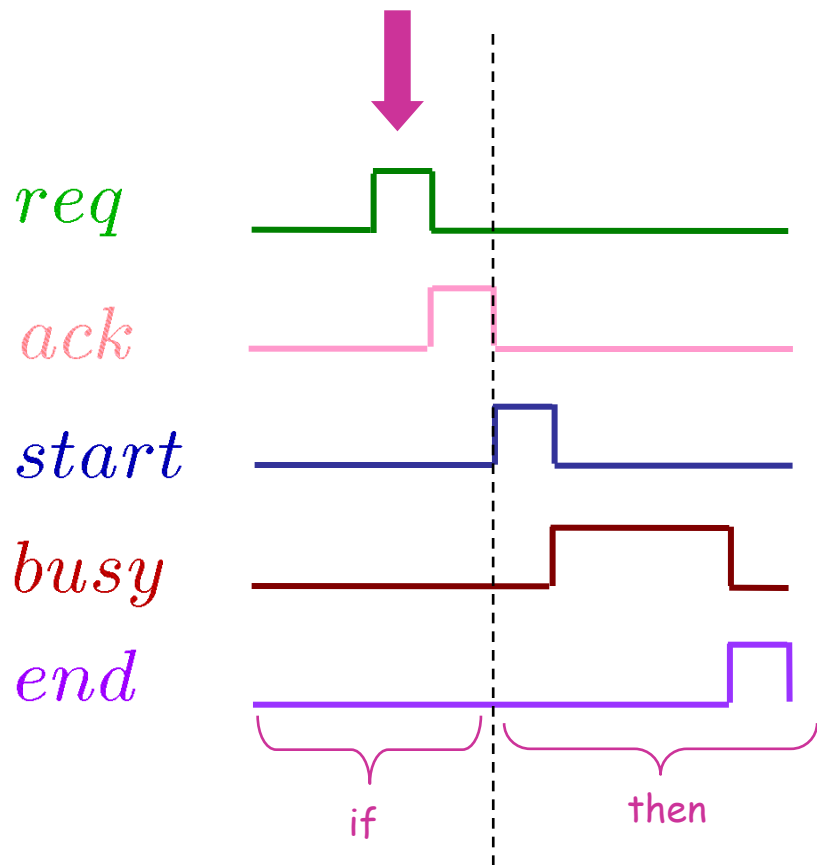
Mixing up logical implication and suffix implication

`assert always {req; ack} -> {start; busy[*]; end}`

Probably didn't mean start to coincide with req

Probably meant

assert always {req; ack} \Rightarrow {start; busy[*]; end}



Confusing and with implication

Every high priority request (req and high_pri) should be followed immediately by an ack and then by a gnt

assert always (req and high_pri) -> next (ack -> next gnt)

or

assert always (req and high_pri) -> next (ack and next gnt)

or

assert always (req and high_pri) | => {ack; gnt}

Which?

Why?

Confusing concatenation with implication

Are these equivalent?

assert always {a; b; c}

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

Confusing concatenation with suffix implication

Are these equivalent?

assert always {a; b[+]; c} \Rightarrow {d}

assert always {a; b[+]; c; d}

Exercise

Figure out from the standard what

$\{\text{SERE}\}(\text{FL_property})$

means

Using never with implication

assert always (req -> next ack)

req is always followed by ack

Two consecutive reqs are not allowed

assert never (req -> next req)

?

Using never with implication

assert always (req -> next ack)

req is always followed by ack

Two consecutive reqs are not allowed

assert never (req -> next req)

??

or

assert always (req -> next (not req))

or

assert never {req; req}

Which? Why?

(And similarly for suffix implication)

Negating implications

assert always ((high_pri and req) -> ack)

High priority req gives an immediate ack

Low priority request does not give an immediate ack

assert always not ((low_pri and req) -> ack) ??

Ex: What should it be?

Check all three assertions on the following traces

(And similarly for suffix implication)

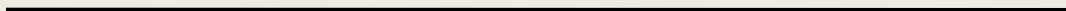
req



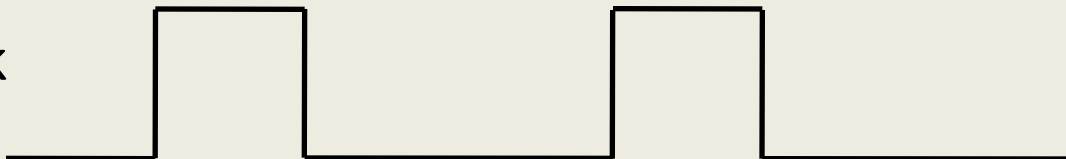
high_pri



low_pri



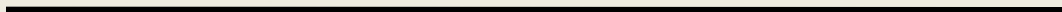
ack



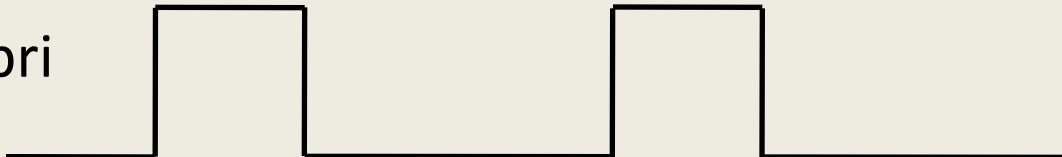
req



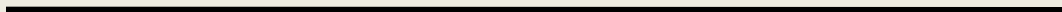
high_pri



low_pri



ack



Incorrect nesting of implications (1)

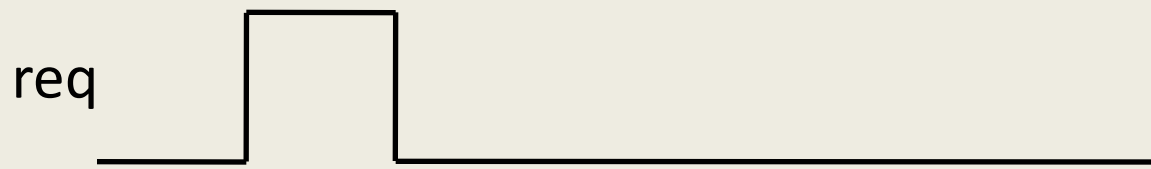
If a request (assertion of req) is acknowledged (assertion of ack the following cycle), then it must receive a grant (assertion of gnt) the cycle following ack

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

Faults?

What should it be? (Write in both LTL and SERE style)

Check on following trace



Incorrect nesting of implications (2)

If there is a granted read request (assertion of req followed by ack followed by gnt), then if there follows a complete data transfer {start; data[*], end}, then the whole thing should be followed by an assertion of signal read_complete.

assert always ({req; gnt; ack} \Rightarrow {start; data[*]; end}) $\mid \Rightarrow$ {read_complete} ??

Fault?

What should it be? (Write with two suffix implications and with one)
In the latter case, think about how moving the position of the suffix implication changes the meaning of the property

Thinking you are missing a “first match” operator


On the cycle after the first ack following a request, a data transfer should begin

assert always ({req; [*]; ack} \Rightarrow {start; data[*]; end}) ??

Wrong: Demands a transfer after every assertion of ack after the req.

Thinking you are missing a "first match" operator

On the cycle after the first ack following a request, a data transfer should begin

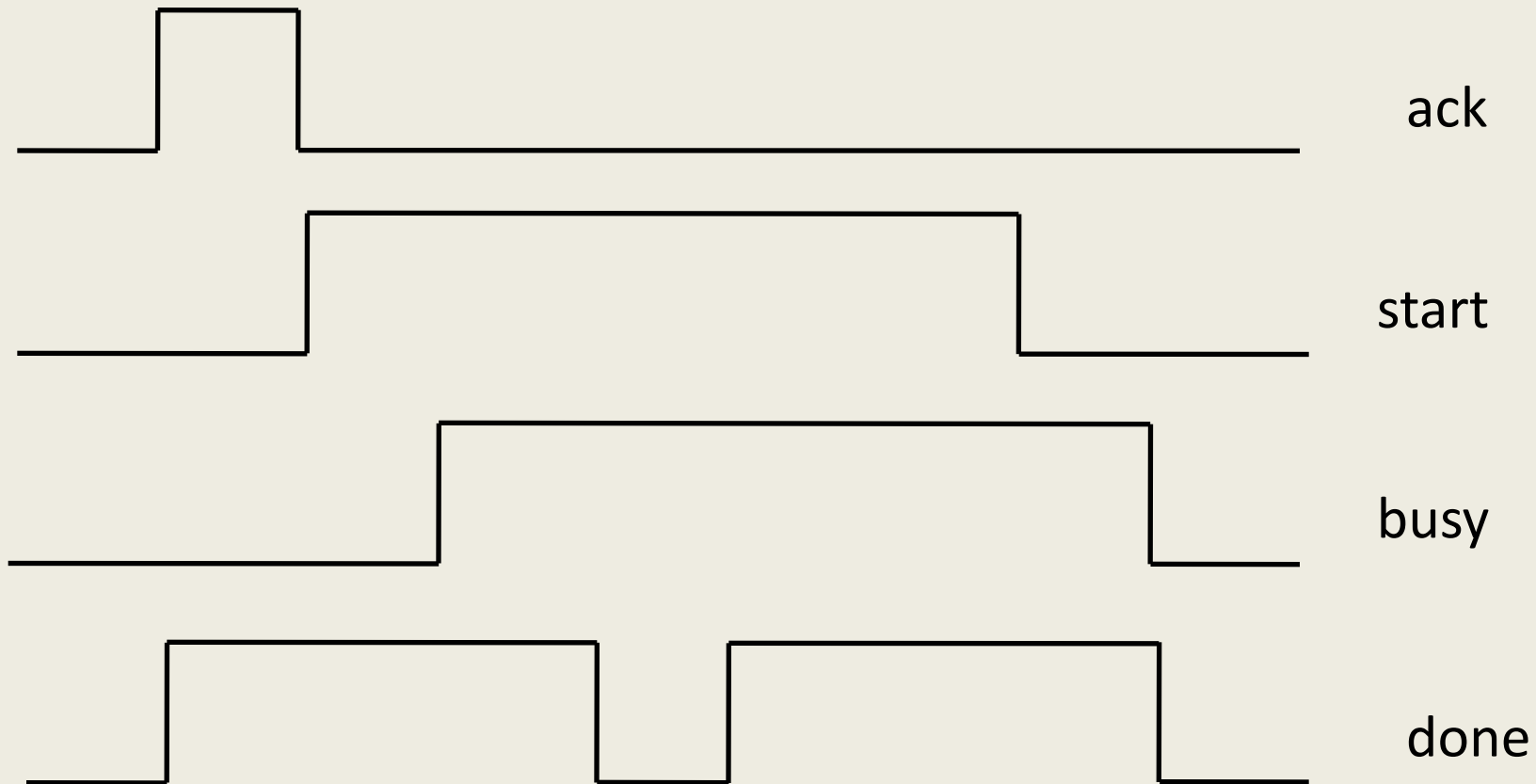
assert always ({req; } | => {start; data[*]; end}) ??

Wrong: Demands a transfer after every assertion of ack after the req.

Answer: use `ack[->]`

“Extraneous” assertions of signals

`assert always {ack} ==> {start ; busy[*] ; done}`



PSL

Seems to be reasonably simple, elegant and concise!

Jasper's Göteborg based team have helped
to define and simplify the formal semantics.

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

In larger examples, one uses the modelling layer (in VHDL) to augment
specs, ending up with small and readable PSL properties