<table>
<thead>
<tr>
<th>Previous notation</th>
<th>PSL ASCII notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P \land Q$</td>
<td>$P &amp; Q$</td>
</tr>
<tr>
<td>$P \Rightarrow Q$</td>
<td>$P \rightarrow Q$</td>
</tr>
<tr>
<td>$\neg P$</td>
<td>${\neg P}$</td>
</tr>
<tr>
<td>$XP$</td>
<td>next $P$</td>
</tr>
<tr>
<td>$FP$</td>
<td>eventually $P$</td>
</tr>
<tr>
<td>$GP$</td>
<td>always $P$</td>
</tr>
<tr>
<td>$[P \cup Q]$</td>
<td>$P$ until $Q$</td>
</tr>
<tr>
<td>$[P \cap Q]$</td>
<td>$P$ until $Q$</td>
</tr>
<tr>
<td>$R^*$</td>
<td>true</td>
</tr>
<tr>
<td>$R_1; R_2$</td>
<td>$R_1 : R_2$</td>
</tr>
<tr>
<td>$R_1; \text{skip}; R_2$</td>
<td>$R_1 ; R_2$</td>
</tr>
</tbody>
</table>

### Semantics:
- $r := \text{Atom}(p)$ (Atomic formula)
- $f := \neg f$ (Negation)
- $f \lor f_2$ (Disjunction)
- $\text{next } f$ (successor)
- $(r)|-> (r_2)$ (Suffix implication)
- $[f]$ (Unit)

### SEREs in HOL

- **Syntax:**
  - $r := \text{Atom}(p)$ (Atomic formula)
  - $f := \neg f$ (Negation)
  - $f \lor f_2$ (Disjunction)
  - $\text{next } f$ (successor)
  - $(r)|-> (r_2)$ (Suffix implication)
  - $[f]$ (Unit)

- **Semantics:**
  - $\text{Atom}(p) = \lambda \sigma. p(\sigma(0))$
  - $\neg f = \lambda \sigma. \neg f(\sigma)$
  - $f_1 \lor f_2 = \lambda \sigma. f_1(\sigma) \lor f_2(\sigma)$
  - $\text{next } f = \lambda \sigma. f(\text{next } 1(\sigma))$
Combining SEREs with LTL formulas

- Formula $[r]$ means LTL formula $f$ true after SERE $r$
- Example
  After a sequence in which $req$ is asserted, followed four cycles later by an assertion of $grant$, followed by a cycle in which $abortin$ is not asserted, we expect to see an assertion of $ack$ some time in the future.
  - Can represent by
    \[
    \text{always } (\text{req};[\ast 3];\text{grant};\text{!abortin})(\text{eventually!} \text{ } \text{ack})
    \]
    where $\text{eventually!}$ is LTL future operator $F$, so:
    $\text{eventually! } f = [T \cup f] = [\text{true until! } f]$  
  - N.B. suffix $\text{!}$ denotes “strong”
    - strong/weak distinction not covered here – important for dynamic checking
    - gives semantics when simulator halts before an expected event occurs

Examples of defined notations: consecutive repetition

- Define
  \[
  \begin{align*}
  r[\ast] &= \{r; r[\ast]\} \\
  r[\ast i] &= \{f; r[r]; \ldots ; r\} \quad \text{otherwise (i repetitions of r)} \\
  r[\ast i..j] &= \{r[\ast i]\} \cup \{r[\ast (i+1)]\} \cup \ldots \cup \{r[\ast j]\} \\
  [\ast] &= \text{true}[\ast] \\
  [\ast i] &= \text{true}[\ast]
  \end{align*}
  \]
- Example
  Whenever we have a sequence of $req$ followed by $ack$, we should see a full transaction starting the following cycle. A full transaction starts with an assertion of the signal $\text{start} \text{ } \text{trans}$, followed by one to eight consecutive data transfers, followed by the assertion of signal $\text{end} \text{ } \text{trans}$. A data transfer is indicated by the assertion of signal $\text{data}$
  \[
  \text{always } (\text{req};\text{ack}) \Rightarrow \{\text{start} \text{ } \text{trans};\{\{\text{!data}[\ast];\text{data}[\ast]\}\times 8;\{\text{data}[\ast]\};\text{end} \text{ } \text{trans}\}
  \]

Fixed number of non-consecutive repetitions

- Example
  Whenever we have a sequence of $req$ followed by $ack$, we should see a full transaction starting the following cycle. A full transaction starts with an assertion of the signal $\text{start} \text{ } \text{trans}$, followed by eight not necessarily consecutive data transfers, followed by the assertion of signal $\text{end} \text{ } \text{trans}$. A data transfer is indicated by the assertion of signal $\text{data}$
  - Can represent by
    \[
    \text{always } (\text{req};\text{ack}) \Rightarrow \{\text{start} \text{ } \text{trans};\{\{\text{data}[\ast];\text{data}[\ast]\}\times 8;\{\text{data}[\ast]\};\text{end} \text{ } \text{trans}\}
    \]
    - Define
      \[
      b[= i] = \{\text{!b}[\ast];b[\ast]\times i;\text{!b}[\ast]
      \]
    - Then have a nicer representation
      \[
      \text{always } (\text{req};\text{ack}) \Rightarrow \{\text{start} \text{ } \text{trans};\{\text{data}[= 8];\text{end} \text{ } \text{trans}\}
      \]
Variable number of non-consecutive repetitions

- Example
  Whenever we have a sequence of req followed by ack, we should see a full transaction starting the following cycle. A full transaction starts with an assertion of the signal start\_trans, followed by one to eight not necessarily consecutive data transfers, followed by the assertion of signal end\_trans. A data transfer is indicated by the assertion of signal data.

- Define
  \[ b[i..j] = \{b[i]\} | \{b[i+1]\} | \ldots | \{b[j]\} \]

- Then
  \[ \text{always(req;ack)} \implies \{\text{start\_trans;data[1..8];end\_trans}\} \]

- These examples are meant to illustrate how PSL/Sugar is much more readable than raw CTL or LTL.

Clocking

- Basic idea: \( b@clk \) abstracts \( b \) on rising edges of \( clk \)
- Can clock SEREs (\( r@clk \)) and formulas (\( f@clk \))
- Can have several clocks
- Official semantics messy due to clocking
- Can ‘translate away’ clocks by pushing \( @clk \) inwards
  - rules given in PSL manual
  - roughly: \( b@clk \rightarrow \{ !clk[*]; clk & b \} \)
- Same idea as temporal abstraction: \( b \) at \( clk \)

Model checking PSL

- SEREs checked by generating a finite automaton
  - recall: regular expressions can be recognised by finite automata
  - these automata are called “satellites”
- FL checked using standard LTL methods
- OBE checked by standard CTL methods
- Can also check formula for runs of a simulator
  - this is dynamic verification
  - semantics handles possibility of finite paths – messy!

PSL layer structure

- **Boolean layer** has atomic predicates
- **Temporal layer** has LTL (FL) and CTL (OBE) properties
- **Verification layer** has commands for how to use properties
  - e.g. assert, assume
    - \[ \text{assert always (!en1 & en2)} \]
      - \[ \implies \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
      - \[ | \]
PSL/Sugar summary

- Combines together LTL, ITL and CTL
- Regular expressions – SEREs
- LTL – Foundation Language formulas
- CTL – Optional Branching Extension
- Relatively simple set of primitives + definitional extension
- Boolean, temporal, verification, modelling layers
- Semantics for static and dynamic verification (needs strong/weak distinction)

New Topic: Simulation or Event semantics

- HDLs use discrete event simulation
  - changes to variables ⇒ threads enabled
  - enabled threads executed non-deterministically
  - execution of threads ⇒ more events
- Combinational thread:
  \[
  \text{always } @ (v_1 \text{ or } \cdots \text{ or } v_n) \ v := E
  \]
  - enabled by any change to \( v_1, \ldots, v_n \)
- Positive edge triggered sequential threads:
  \[
  \text{always } @ (\text{posedge } clk) \ v := E
  \]
  - enabled by \( clk \) changing to \( T \)
- Negative edge triggered sequential threads:
  \[
  \text{always } @ (\text{negedge } clk) \ v := E
  \]
  - enabled by \( clk \) changing to \( F \)

Simulation

- Given
  - a set of threads
  - initial values for variables read or written by threads
  - a sequence of input values (inputs are variables not in LHS of assignments)
- simulation algorithm ⇒ a sequence of states

- Simulation is non-deterministic

Combinational threads in series

- HDL-like specification:
  \[
  \begin{align*}
  &\text{always } @ (in) \ l_1 := f(in) \quad \ldots \quad \text{thread T1} \\
  &\text{always } @ (l_1) \ l_2 := g(l_1) \quad \ldots \quad \text{thread T2} \\
  &\text{always } @ (l_2) \ out := h(l_2) \quad \ldots \quad \text{thread T3}
  \end{align*}
  \]
- Suppose \( in \) changes to \( v \) at simulation time \( t \)
  - \( T1 \) will become enabled and assign \( f(v) \) to \( l_1 \)
  - \( l_1 \)'s value changes then \( T2 \) will become enabled (still simulation time \( t \))
  - \( T2 \) will assign \( g(f(v)) \) to \( l_2 \)
  - \( l_2 \)'s value changes then \( T \) will become enabled (still simulation time \( t \))
  - \( T3 \) will assign \( h(g(f(v))) \) to \( out \)
  - simulation quiesces (still simulation time \( t \))
- Steps at same simulation time happen in \( \delta \)-time
  (VHDL jargon)
Semantic gap

- Designers use HDLs and verify via simulation
- Formal verifiers use logic and verify via proof
  - trace semantics
- Problem: show consistency between semantics
- Goal:
  - traces = sequences of quiescent simulation states

Outline (see Section 4.4 of Notes for details):
- first analyse sets of combinational threads
- identify conditions for "non-looping"
- simulation terminates → trace semantics (partial correctness)
- simulation always terminates "quiesces" (total correctness)
- extend to sequential threads

Trace defined by a simulation run

- Simulation defines a tree of states

Sequential threads – event semantics

- Consider two Dtypes in series:
  - always @(posedge clk) l := in
  - always @(posedge clk) out := l

- If possedg clk:
  - both threads become enabled
  - race condition
  - Right thread executed first:
    - out gets previous value of l
    - then left thread executed
    - so l gets value input at in
  - Left thread executed first:
    - l gets input value at in
    - then right thread executed
    - so out gets input value at in

Sequential threads – trace semantics

- Trace semantics:
  \[ \forall t. l(t+1) = (\text{Rise } clk t \rightarrow in t \mid l(t)) \land \]
  \[ \forall t. out(t+1) = (\text{Rise } clk t \rightarrow l(t) \mid out(t)) \]
  - Corresponds to right thread executed first
  - How to ensure event and trace semantics agree?
  - Method 1: use non-blocking assignments:
    - always @(posedge clk) l <= in;
    - always @(posedge clk) out <= l;
    - non-blocking assignments (<=) in Verilog
    - RHS of all non-blocking assignments first computed
    - assignments done at end of simulation cycle
  - Method 2: make simulation cycle VHDL-like
Verilog versus VHDL simulation cycles

- **Verilog-like simulation cycle:**
  1. Choose an enabled thread
  2. Execute the chosen thread
  3. Fire event controls to enable new threads
  4. Execute until quiescent then advance simulation time

- **VHDL-like simulation cycle:**
  1. Execute all enabled threads in parallel
  2. Fire event controls to enable new threads
  3. Execute until quiescent then advance simulation time

VHDL event semantics

- Recall HDL:
  
  ```
  always @(posedge clk) l := in
  always @(posedge clk) out := l
  ```

- If `posedge clk`:
  - both threads become enabled

- **VHDL semantics:**
  - both threads executed in parallel
  - `out` gets previous value of `l` in parallel
  - `in` gets value input at `in`

- **Now no race**

- Event semantics matches trace semantics

Summary of dynamic versus static semantics

- Simulation (event) semantics different from trace semantics
- No standard event semantics (Verilog versus VHDL)
- Verilog: need non-blocking assignments
- VHDL semantics closer trace semantics

Summary of Specification I and II

- **Software specification and verification**
  - Hoare logic: partial and total correctness
  - proof by invariants and variants
  - mechanisation via VCs (WP or SP)
  - only nice for simple languages
  - can apply Hoare logic to behavioral view of hardware

- **Higher order logic (HOL)**
  - unifying general logic
  - supports Hoare logic via embedding
  - supports temporal logics via embedding
  - can directly represent hardware behavior and structure \((\exists, \forall)\)
  - hardware verification as pure logic proof
  - relating models: event vs trace vs RTL vs cycles

- **Hardware specification and verification**
  - automatic FV uses state machine models, fit nicely into HOL
  - reachable states calculated by iteration (fixed point)
  - symbolic representations: BDDs
  - model checking of properties (CTL, LTL, ITL, PSL)
  - event simulation used in industry

THE END - HAVE A GOOD VACATION!