PSL notation

Previous notation	PSL ASCII notation	
$P \wedge Q$	`P & Q`	
$P \Rightarrow Q$	`P -> Q`	
$\neg P$	`!P`	(exclamation mark is negation)
$\mathbf{X}P$	`next P`	
$\mathbf{F}P$	`eventually! P`	(exclamation mark is not negation)
$\mathbf{G}P$	`always P`	
$[P\ \mathbf{U}\ Q]$	`P until! Q`	
$[P\ \mathbf{W}\ Q]$	`P until Q`	
skip	`true`	
$R*$	`R[*]`	
R_1 ; R_2	R_1 : R_2	
R_1 ;**skip**; R_2	R_1 ; R_2	

Sequential Extended Regular Expressions (SEREs)

- **Similar to ITL – but weaker**
- **On earlier slide:** `R[*]`, R_1:R_2, R_1;R_2
- **Other SERE operators include**

R_1 | R_2	either R_1 or R_2 holds
R_1 && R_2	both R_1 and R_2 hold for same number of cycles
R_1 & R_2	both R_1 and R_2 hold, but one may finish before the other

- **Actually & is not primitive (braces { and } used for grouping)**

```
{r1} & {r2}  =  {{r1} && {r2;true[*]}} | {{r1;true[*]} ** {r2}}
```

- **SEREs can be used to improve readability of formulas, compare:**

```
always
 (reqin -> next(ackout -> next(!abortin -> (ackin & next ackin))))
```

with

```
always {reqin;ackout;!abortin} |-> {ackin;ackin}
```

where PSL formulas r_1 |-> r_2 defined later

SEREs in HOL

- **Syntax :**

r ::= `Atom`(p)	(Atomic formula)
| r_1 | r_2	(Disjunction)
| r_1 ; r_2	(Concatenation)
| r_1 : r_2	(Fusion: ITL's chop)
| r_1 && r_2	(Length matching conjunction)
| $r[*]$	(Repeat)

- **Semantics:**
(s ranges over states; w ranges over finite lists of states; "head" denotes head of a list; $|w|$ denotes the length; infix "." denotes concatenation)

`Atom`(p) $= \lambda w.\ p(\textbf{head}\ w) \wedge |w| = 1$

r_1 | r_2 $= \lambda w.\ r_1\ w\ \vee\ r_2\ w$

r_1 ; r_2 $= \lambda w.\ \exists w_1\ w_2.\ w = w_1.w_2\ \wedge\ r_1\ w_1\ \wedge\ r_2\ w_2$

r_1 : r_2 $= \lambda w.\ \exists w_1\ s\ w_2.\ w = w_1.s.w_2\ \wedge\ r_1(w_1.s)\ \wedge\ r_2(s.w_2)$

r_1 && r_2 $= \lambda w.\ r_1\ w\ \wedge\ r_2\ w$

$r[*]$ $= \lambda w.\ w = \langle\rangle \vee \exists w_1 \cdots w_l.\ w = w_1.\cdots.w_l\ \wedge\ r\ w_1\ \wedge\ \cdots\ \wedge\ r\ w_l$

PSL Foundation Language (FL)

- **Syntax:**

f ::= `Atom`(p)	(Atomic formula)
| $\neg f$	(Negation)
| $f_1 \vee f_2$	(Disjunction)
| `next` f	(successor)
| $\{r\}(f)$	(Suffix implication)
| $\{r_1\}$ |-> $\{r_2\}$	(Suffix next implication)
| [f_1 `until` f_2]	(Until)

- **Semantics**
(simplified – no clocking, weak/strong distinction omitted):

`Atom`(p) $= \lambda\sigma.\ p(\sigma(0))$

$\neg f$ $= \lambda\sigma.\ \neg(f\ \sigma)$

$f_1 \vee f_2$ $= \lambda\sigma.\ f_1\ \sigma\ \vee\ f_2\ \sigma$

`next` f $= \lambda\sigma.\ f(\text{Tail } 1\ (\sigma))$

$\{r\}(f)$ $= \lambda\sigma.\ \exists w\ \sigma'.\ \sigma = w.\sigma'\ \wedge\ r\ w\ \wedge\ f\ \sigma'$

$\{r_1\}$ |-> $\{r_2\}$ $= \lambda\sigma.\ \exists w_1\ \sigma'.\ \sigma = w_1.\sigma'\ \wedge\ r_1\ w_1\ \Rightarrow\ \exists w_2\ \sigma''.\ \sigma' = w_2.\sigma''\ \wedge\ r_2\ w_2$

[f_1 `until` f_2] $= \lambda\sigma.\ \exists i.\ f_2(\text{Tail } i\ \sigma)\ \wedge\ \forall j.\ j < i\ \Rightarrow\ f_1(\text{Tail } j\ \sigma)$

- **There is also an Optional Branching Extension (OBE)**
  - **completely standard CTL: EX, E[–U–], EG etc.**

Combining SEREs with LTL formulas

- **Formula $\{r\}f$ 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**

```
always {req;[*3];grant;!abortin}(eventually! ack)
```

- **where `eventually!` is LTL future operator `F`, so:**

```
eventually! f  =  [T U f]  =  [true until! f]
```

- **N.B. suffix "`!`" denotes "strong"**
  - strong/weak distinction not covered here – important for dynamic checking
  - gives semantics when simulator halts before an expected event occurs

SERE examples

- **How can we modify**

```
always {reqin;ackout;!abortin} |-> {ackin;ackin}
```

  so that the two cycles of `ackin` start the cycle after `!abortin`?

- **Two ways of doing this**

```
always {reqin;ackout;!abortin} |-> {true;ackin;ackin}

always {reqin;ackout;!abortin} |=> {ackin;ackin}
```

- **`|=>` is a defined operator**

```
{r1} |=> {r2}  =  {r1} |-> {true;r2}
```

- **Note: `true` and `T` are synonyms**

Examples of defined notations: consecutive repetition

- **Define**

```
r[+]       = {r;r[*]}
             ┌  false[*]     if i=0
r[*i]      = │
             └  {r;r;...;r}  otherwise (i repetitions of r)
r[*i..j]   = {r[*i]} | {r[*(i+1)]} | ... | {r[*j]}
[+]        = true[+]
[*]        = true[*]
```

- **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 consecutive data transfers, followed by the assertion of signal* `end_trans`*. A data transfer is indicated by the assertion of signal* `data`

```
always {req;ack} |=> {start_trans;data[*1..8];end_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* `start_trans`*, followed by 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`

- **Can represent by**

```
always {req;ack} |=>
       {start_trans;{{!data[*];data}[*8];!data[*]};end_trans}
```

- **Define**

```
b[= i] = {!b[*];b}[*i];!b[*]
```

- **Then have a nicer representation**

```
always {req;ack} |=> {start_trans;data[= 8];end_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)]} | ... | {b[= j]}
```

- **Then**

```
always{req;ack} |=> {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` $\longrightarrow$ `{!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

```
assert always (!en1 & en2))
   |      |          |
   |      |          |
   |      |          |--- Boolean layer
   |      |
   |      |-------------- temporal layer
   |
   |-------------------- verification layer
```

- **Modelling layer has HDL constructs for specifying inputs and auxiliary hardware**

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:

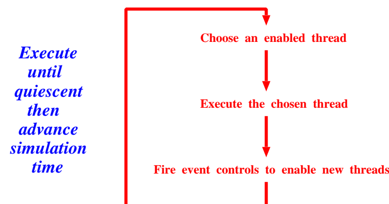
  `always @(`v_1 `or` $\cdots$ `or` v_n`)` v `:=` E
  - enabled by any change to $v_1, \ldots, v_n$
- Positive edge triggered sequential threads:

  `always @(posedge` clk`)` v `:=` E
  - enabled by clk changing to `T`
- Negative edge triggered sequential threads:

  `always @(negedge` clk`)` v `:=` E
  - enabled by clk changing to `F`

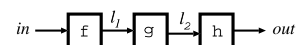
Simulation

- 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:

  `always @(`in`)` l_1 `:= f(`in`)` ..... thread T1
  `always @(`l_1`)` l_2 `:= g(`l_1`)` ..... thread T2
  `always @(`l_2`)` out `:= h(`l_2`)` ..... thread T3

- Suppose in changes to v at simulation time t
  - T1 will become enabled and assign `f`(v) to l_1
  - if l_1's value changes then T2 will become enabled
    (still simulation time t)
  - T2 will assign `g(f(`v`))` to l_2
  - if 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 δ-*time*
  (VHDL jargon)

Semantic gap

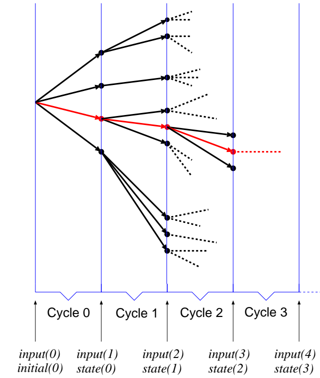
- Designers use HDLs and verify via simulation
  - event semantics
- Formal verifiers use logic and verify via proof
  - trace semantics
- **Problem:** show consistency between semantics
- Goal:

  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 termininates"quiesces" (total correctness)
  - extend to sequential threads

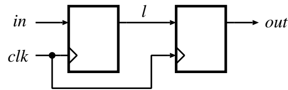
Trace defined by a simulation run

- Simulation defines a tree of states



- Inputs read at start of cycle
- State computed at end of cycle
- Traces = sequences of end-of-cycle states (example shown in red)
- Branching time

Sequential threads – event semantics

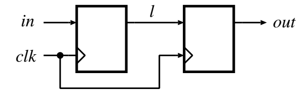


- Consider two Dtypes in series:

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

- If `posedge` *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:

  $(\forall t.\ l(t{+}1) \ = \ (\texttt{Rise}\ clk\ t\ \rightarrow\ in\ t\ \mid\ l\ t)) \wedge$

  $(\forall t.\ out(t{+}1) \ = \ (\texttt{Rise}\ clk\ t\ \rightarrow\ l\ t\ \mid\ out\ t))$

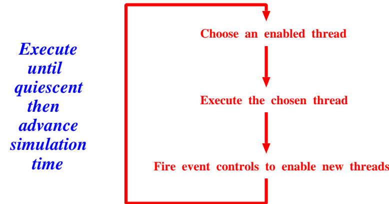
- Corresponds to right thread executed first
- How to ensure event and trace semantics agree?
- **Method 1:** use non-blocking assignments:

```
always @(posedge clk) l <= in;
always @(posedge clk) out <= l;
```

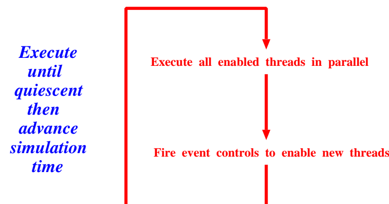
  - non-blocking assignments (`<=`) in Verilog
  - RHS of all non-blocking assignments first computed
  - assignments done at end of simulation cycle
- **Method 2:** make simulation cycle VHDL-like

Verilog versus VHDL simulation cycles

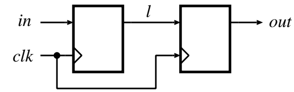
- Verilog-like simulation cycle:



- VHDL-like simulation cycle:



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 *l* 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 ($\exists$, $\wedge$)
  - 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!