Simulation semantics (a.k.a. event semantics)

- HDLs use *discrete event simulation*
  - changes to variables $\Rightarrow$ threads enabled
  - enabled threads executed non-deterministically
  - execution of threads $\Rightarrow$ 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:

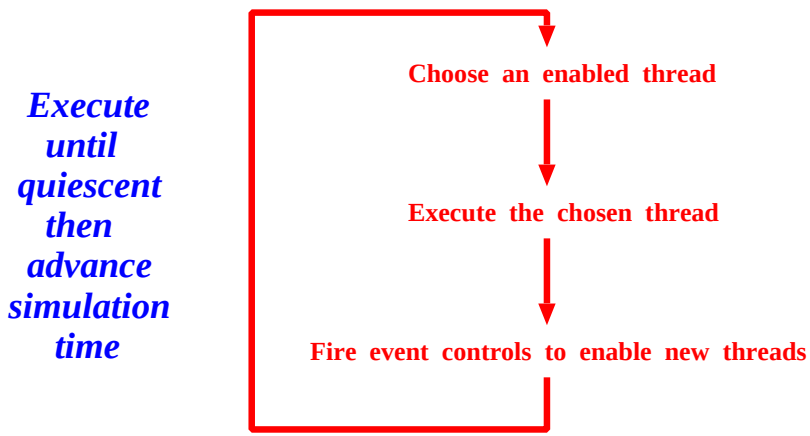
  `always @(posedge` *clk*`)` v`:=`E
  - enabled by *clk* changing to `T`
- Negative edge triggered sequential threads:

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

Simulation

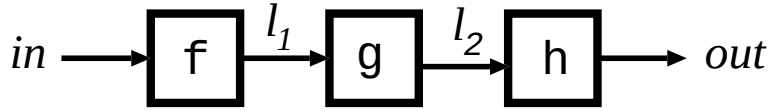
- Given
  - a set of threads
  - initial values for variables read or written by threads
  - a sequence of input values
    (inputs are variables not in LHS of assignments)

- *simulation algorithm* $\Rightarrow$ a sequence of states



- Simulation is non-deterministic

Combinational threads in series



- HDL-like specification:

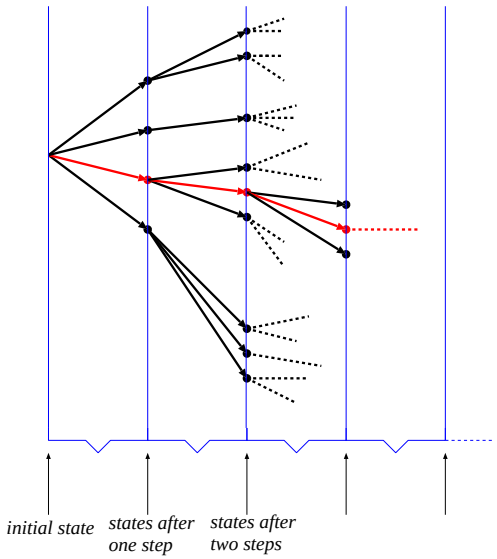
```
always @(in)  l1  := f(in)   ..... thread T1
always @(l1)  l2  := g(l1)   ..... thread T2
always @(l2)  out := h(l2)   ..... thread T3
```

- Suppose *in* changes to `x` at simulation time *t*
  - T1 will become enabled and assign `f(x)` to l_1
  - if l_1's value changes then T2 will become enabled (still simulation time *t*)
  - T2 will assign `g(f(x))` to l_2
  - if l_2's value changes then T3 will become enabled (still simulation time *t*)
  - T3 will assign `h(g(f(x)))` to *out*
  - simulation quiesces (still simulation time *t*)
- Steps at same simulation time happen in "δ-*time*" (VHDL jargon)

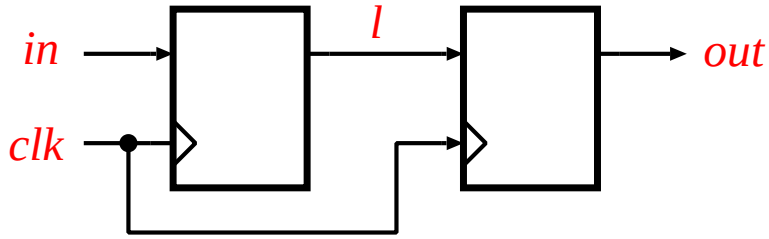
Semantic gap

- Designers use HDLs and verify via simulation
  - event semantics
- Formal verifiers use logic and verify via proof
  - path semantics
- Problem: do path and simulation semantics agree?
- Would like:

  paths = sequences of quiescent simulation states



Sequential threads: alternative simulation semantics

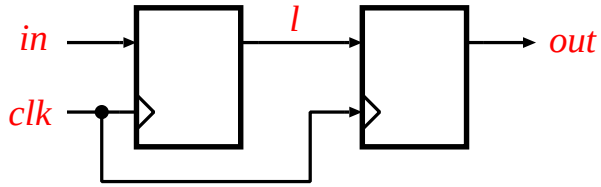


- Consider two Dtypes in series:

```
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: aligning semantics



- If right thread executed first get formal model semantics
  $R(in, l, out)(in', l', out') \;=\; (l' = in) \wedge (out' = l)$
- If left thread executed first get weird semantics
  $R(in, l, out)(in', l', out') \;=\; (l' = in) \wedge (out' = in)$
- How to ensure formal model semantics?
- **Method 1:** use non-blocking assignments:

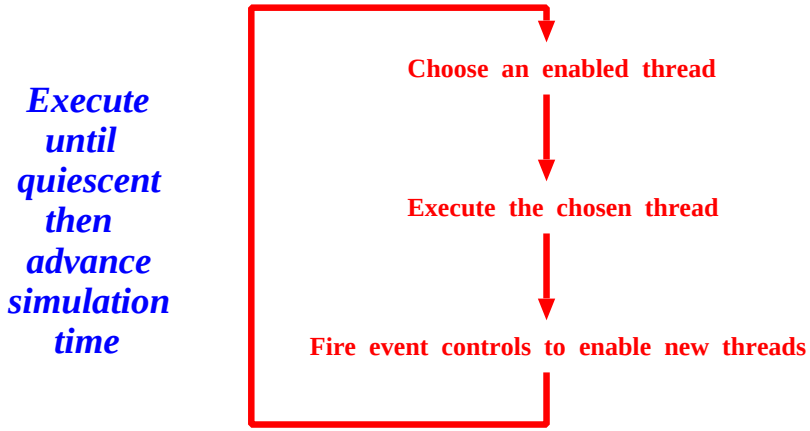
```
always @(posedge clk) l <= in;
always @(posedge clk) out <= l;
```

  - non-blocking assignments (<=) in Verilog
  - RHS of all non-blocking assignments first computed
  - assignments done at end of simulation cycle

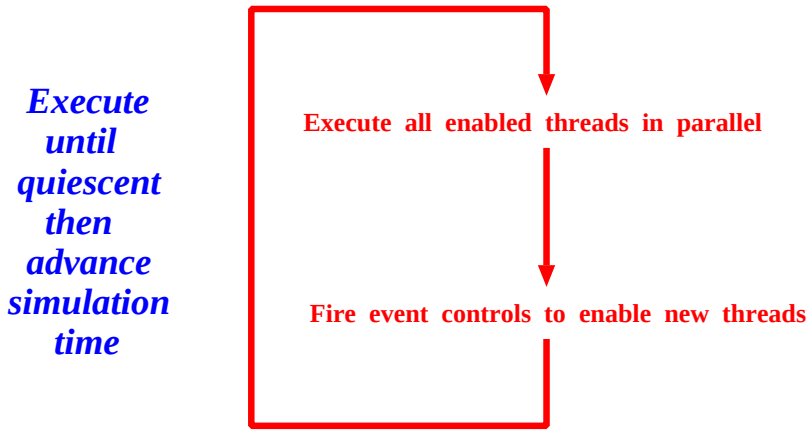
- **Method 2:** make simulation cycle VHDL-like

Verilog versus VHDL simulation cycles

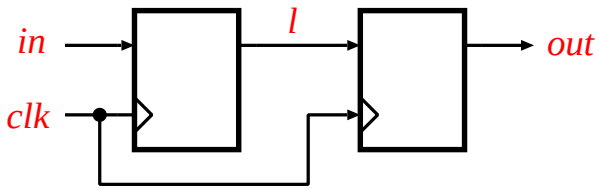
- Verilog-like simulation cycle:



- VHDL-like simulation cycle:



VHDL event semantics

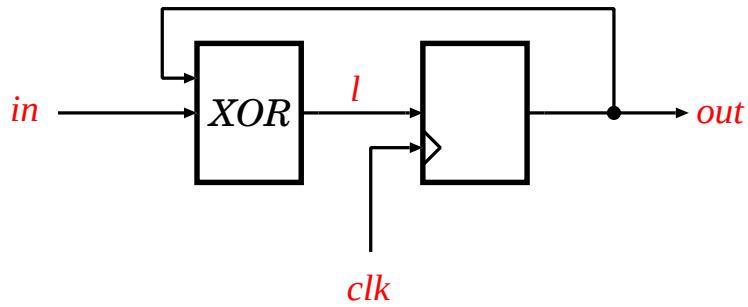


- Recall HDL:

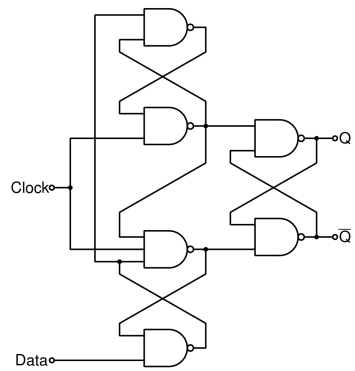
```
always @(posedge clk) l := in
always @(posedge clk) out := l
```

- 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 path semantics

Another example: combinational + sequential



- Exercise: Do VHDL and Verilog event semantics agree?
- Ignoring race if input does change at clock edge
  - in real world might get meta-stability problems
  - also in previous example
  - need analogue simulation (e.g. using SPICE)



Summary of dynamic versus static semantics

- Simulation (event) semantics different from path semantics
- No standard event semantics (Verilog versus VHDL)
- Verilog: need non-blocking assignments
- VHDL semantics closer path semantics
- Simulation runs generate finite sequences
  - better fit with LTL than CTL