Event, Trace & Cycle semantics of Hardware Description Languages

Overview:

- A simple HDL
- Examples
- Kinds of semantics:
  - event semantics
  - trace semantics
  - cycle semantics
- For the future ...

A Simple Language (based on Verilog)

Programs

$\mathcal{P}$ ::= `forever` $\mathcal{S}_1$ $\cdots$ `forever` $\mathcal{S}_n$ (Threads)

Statements

$\mathcal{S}$	::=	$\mathcal{R}$ `:=` $\mathcal{E}$	(Assignment)
	$\mid$	$\mathcal{S}_1$ `;` $\cdots$ `;` $\mathcal{S}_n$	(Sequencing)
	$\mid$	`if` $\mathcal{E}$ `then` $\mathcal{S}_1$ $\{$`else` $\mathcal{S}_2\}$	(Conditional)
	$\mid$	`@(`$\mathcal{T}$`)` $\mathcal{S}$	(Timing)

Event expressions

$\mathcal{T}$	::=	$\mathcal{V}$	(Change of value)
	$\mid$	`posedge` $\mathcal{V}$	(Positive edge)
	$\mid$	`negedge` $\mathcal{V}$	(Negative edge)
	$\mid$	$\mathcal{T}_1$ `or` $\cdots$ `or` $\mathcal{T}_n$	(Disjunction list)

Combinational Examples

- A simple adder

```
forever @(a or b) c := a + b
```

- Some random logic:

```
forever
 @(a or b or c or d)
  f := a;
  if b
   then if c then f := d
             else f := ¬d
```

this is equivalent to:

```
forever
 @(a or b or c or d)
  f := b ? c ? d : ¬d : a
```

where b ? x : y is a conditional expression

Flip-flops

- Edge-triggered flip-flop

```
forever @(posedge clk) q := d
```

set `q` to `d` on rising edge of `clk`

- Edge-triggered flip-flop with built-in adder

```
forever @(posedge clk) x := y + z
```

set `x` to `y+z` on rising edge of `clk`

- Edge-triggered flip-flop with separate adder

```
forever @(y or z) d := y + z

forever @(posedge clk) x := d
```

Latches

- Transparent latch

```
forever
 @(clk or d)
  if clk then q := d
```

  - connects `q` to `d` when `clk` is high
  - latches `q` to `d` on falling edge of `clk`

- Implementation (if latch not available)

```
forever
 @(clk or d or i)
  q := clk ? d : i

forever @(negedge clk) i := d
```

Event (Simulation) Semantics

- Each `forever` $\mathcal{S}$ generates a thread:
  - executes pseudo-code instructions
- Global simulation state consists of:
  - simulation time
  - values of variables
  - program counter for each thread
- Inputs supplied externally
  - change when simulation time advances

Pseudo-code

- Pseudo-code instructions

$\mathcal{R}$ `:=` $\mathcal{E}$	assignment
`@(`$\mathcal{T}$`)`	timing control
`go` n	unconditional jump to n
`ifnot` $\mathcal{E}$ `go` n	jump to n if $\mathcal{E}$ is false

- Each thead executed concurrently
  - simulation algorithm (event semantics)
  - non-deterministic thread interleaving

- Pseudo-code instructions are atomic actions

Combinational Example

```
forever
 @(a or b or c or d)
  f := a;
  if b
   then if c then f := d
                  else f := ¬d
```

compiles to

```
0:    @(a or b or c or d)
1:    f := a
2:    ifnot b go 7
3:    ifnot c go 6
4:    f := d
5:    go 7
6:    f := !d
7:    go 0
```

Sequential Example

```
forever
  @(posedge clk) total := data;
  @(posedge clk) total := total + data;
  @(posedge clk) total := total + data;
```

translates to

```
0:   @(posedge clk)
1:   total:= data
2:   @(posedge clk)
3:   total := total + data)
4:   @(posedge clk)
5:   total := total + data
6:   go 0
```

"Enabled" and "Firing"

- The current instruction of a thread is the one pointed to by its program counter
- A thread is enabled if its current instruction is not a timing control
- Conditions for an instruction to fire: (if current instruction is `@(`$\mathcal{T}$`)`)
  - `@(`$\mathcal{V}$`)` fires if the current value of $\mathcal{V}$ differs from the previous one
  - `@(posedge` $\mathcal{V}$`)` fires if the current value of $\mathcal{V}$ is `1` and the previous values was not `1`
  - `@(negedge` $\mathcal{V}$`)` fires if the current value of $\mathcal{V}$ is `0` and the previous values was not `0`
  - `@(`$\mathcal{T}_1$ `or` $\cdots$ `or` $\mathcal{T}_n$`)` fires if any $\mathcal{T}_i$ fires

Simulation Cycle

1 If there are no enabled threads then go to **3**, else choose an enabled thread and execute its current instruction as follows:

(a) $\mathcal{V}$ `:=` $\mathcal{E}$ — change value of $\mathcal{V}$ to value of $\mathcal{E}$, increment program counter, go to **2**

(b) `go` n — set the program counter to n, go to **1**

(c) `ifnot` $\mathcal{E}$ `go` n — if $\mathcal{E}$ is true then increment program counter, otherwise set it to n, go to **1**

2 Increment the program counters of all threads whose current instruction is a timing control that fires and go to **1**

3 Increment simulation time, update the state with any changes from inputs and go to **2**

Example (race condition)

Consider:

```
forever @(y or z) d := y + z

forever @(posedge clk) x := d
```

which translates to two threads

```
0:  @(y or z)              0:  @(posedge clk)
1:  d := y + z             1:  x := d
2:  go 0                   2:  go 0
```

- race if `posedge clk` and `y or z` fire simultaneously

  A: if `d := y + z` is done first then `x` ends up with the value of `y + z`

  B: if `x := d` is done first then `x` ends up with the previous value of `d`

- Synthesizers generate flip-flop in series with combinational logic
  - corresponds to A.

Example (another race condition)

Consider:

```
forever @(posedge clk) i := d

forever @(posedge clk) q := i
```

which translates to two threads

```
0:  @(posedge clk)    0:  @(posedge clk)
1:  i := d            1:  q := i
2:  go 0              2:  go 0
```

- race after `posedge clk` fires

  A: if `i := d` is done first then `q` ends up with the value of `d`

  B: if `q := i` is done first then `q` ends up with the previous value of `i`

- Synthesizers generate two flip-flops in series
  - corresponds to B.

Trace Semantics

- Simulation cycle either loops or quiesces
- A trace is the sequence of quiescent states at successive simulation times
- Trace semantics is the set of possible traces that a program can yield under simulation
  - really need a tree (branching time) rather than a sequence
  - sequences OK for synthesized hardware
- Trace semantics an (approximate) abstraction from event semantics

  Goal: find syntactic restrictions sufficient to ensure trace semantics is a correct abstraction

Steps

- A step specifies transitions between quiescent states
  - a trace is a sequence of steps
- Steps give the cumulative effect of a simulation cycle
  - races between threads ignored
- Steps are computed by symbolic execution
  - symbolically accumulate effects from each timing control to the next one
  - symbolic execution of `ifnot` instruction yields a conditional expression

Example Step

```
forever
 @(posedge clk) if ld then tmp := d1;
 @(negedge clk) r := tmp + d2;
```

compiles to the pseudo-code

```
0:   @(posedge clk)
1:   ifnot ld go 3
2:   tmp := d1
3:   @(negedge clk)
4:   r := tmp + d2
5:   go 0
```

which yields the step:

```
0 : @(posedge clk)
      pc := 3
      tmp := ld ? d1 : previous(tmp)
      r := previous(r)
3 : @(negedge clk)
      pc := 0
      tmp := previous(tmp)
      r := previous(tmp) + d2
```

Trace Semantics

```
0 : @(posedge clk)
      pc := 3
      tmp := ld ? d1 : previous(tmp)
      r := previous(r)
3 : @(negedge clk)
      pc := 0
      tmp := previous(tmp)
      r := previous(tmp) + d2
```

$$
\begin{array}{l}
\exists \mathtt{pc}.\ \mathtt{pc(0)=0} \wedge \mathtt{clk(0)=X} \wedge \mathtt{ld(0)=X} \wedge \mathtt{d1(0)=X} \wedge \\
\quad \mathtt{d2(0)=X} \wedge \mathtt{tmp(0)=X} \wedge \mathtt{r(0)=X} \wedge \\
\forall t\mathtt{>0}. \\
\quad \mathtt{pc(}t\mathtt{-1)=0} \Rightarrow \textit{if}\ \mathtt{clk(}t\mathtt{-1)}\neq\mathtt{1} \wedge \mathtt{clk(}t\mathtt{)=1} \\
\qquad\qquad \textit{then}\ \mathtt{pc(}t\mathtt{)=3} \wedge \\
\qquad\qquad\qquad \mathtt{tmp(}t\mathtt{)=ld(}t\mathtt{)?d1(}t\mathtt{):tmp(}t\mathtt{-1)} \wedge \\
\qquad\qquad\qquad \mathtt{r(}t\mathtt{)=r(}t\mathtt{-1)} \\
\qquad\qquad \textit{else}\ \mathtt{pc(}t\mathtt{)=pc(}t\mathtt{-1)} \wedge \\
\qquad\qquad\qquad \mathtt{tmp(}t\mathtt{)=tmp(}t\mathtt{-1)} \wedge \\
\qquad\qquad\qquad \mathtt{r(}t\mathtt{)=r(}t\mathtt{-1)} \\
\quad \wedge \\
\quad \mathtt{pc(}t\mathtt{-1)=3} \Rightarrow \textit{if}\ \mathtt{clk(}t\mathtt{-1)}\neq\mathtt{0} \wedge \mathtt{clk(}t\mathtt{)=0} \\
\qquad\qquad \textit{then}\ \mathtt{pc(}t\mathtt{)=0} \wedge \\
\qquad\qquad\qquad \mathtt{tmp(}t\mathtt{)=tmp(}t\mathtt{-1)} \wedge \\
\qquad\qquad\qquad \mathtt{r(}t\mathtt{)=tmp(}t\mathtt{-1)+d2(}t\mathtt{)} \\
\qquad\qquad \textit{else}\ \mathtt{pc(}t\mathtt{)=pc(}t\mathtt{-1)} \wedge \\
\qquad\qquad\qquad \mathtt{tmp(}t\mathtt{)=tmp(}t\mathtt{-1)} \wedge \\
\qquad\qquad\qquad \mathtt{r(}t\mathtt{)=tmp(}t\mathtt{-1)}
\end{array}
$$

Combinational Example

```
forever @(clk or d or i) q := clk ? d : i
```

Step:

```
0 : @(clk or d or i)
      pc := 0
      q := clk ? d : i
```

Trace:

$$
\begin{array}{l}
\exists \mathtt{pc}.\ \mathtt{pc(0)=0} \wedge \mathtt{clk(0)=X} \wedge \mathtt{d(0)=X} \wedge \\
\quad \mathtt{i(0)=X} \wedge \mathtt{q(0)=X} \wedge \\
\ \forall t\mathtt{>0}. \\
\ \ \mathtt{pc(}t\mathtt{-1)=0} \Rightarrow \textit{if}\ \mathtt{clk(}t\mathtt{-1)}\neq\mathtt{clk(}t\mathtt{)} \vee \\
\qquad\qquad\qquad \mathtt{d(}t\mathtt{-1)}\neq\mathtt{d(}t\mathtt{)} \vee \\
\qquad\qquad\qquad \mathtt{i(}t\mathtt{-1)}\neq\mathtt{i(}t\mathtt{)} \\
\qquad\qquad\quad \textit{then}\ \mathtt{pc(}t\mathtt{)=0} \wedge \\
\qquad\qquad\qquad\quad \mathtt{q(}t\mathtt{)=clk(}t\mathtt{)?d(}t\mathtt{):i(}t\mathtt{)} \\
\qquad\qquad\quad \textit{else}\ \ \mathtt{pc(}t\mathtt{)=pc(}t\mathtt{-1)} \wedge \\
\qquad\qquad\qquad\quad \mathtt{q(}t\mathtt{)=q(}t\mathtt{-1)}
\end{array}
$$

Simplification of Trace Semantics

```
∃pc. pc(0)=0 ∧ clk(0)=X ∧ d(0)=X ∧
     i(0)=X ∧ q(0)=X ∧
 ∀t>0.
  pc(t-1)=0 ⇒ if clk(t-1)≠clk(t) ∨
                 d(t-1)≠d(t) ∨
                 i(t-1)≠i(t)
              then pc(t)=0 ∧
                   q(t)=clk(t)?d(t):i(t)
              else pc(t)=pc(t-1) ∧
                   q(t)=q(t-1)
```

simplifies to:

```
 clk(0)=X ∧ d(0)=X ∧ i(0)=X ∧ q(0)=X ∧
 ∀t>0. if clk(t-1)≠clk(t) ∨
          d(t-1)≠d(t) ∨
          i(t-1)≠i(t)
       then q(t)=clk(t)?d(t):i(t)
       else q(t)=q(t-1)
```

and then to:

```
 clk(0)=X ∧ d(0)=X ∧ i(0)=X ∧ q(0)=X ∧
 ∀t>0. q(t)=clk(t)?d(t):i(t)
```

Temporal Operators

- Suppose p represents a set of events:
  - $p\ t$ true exactly when event happens at t
- Define `last` $p\ f\ t$ to be:
  - the value of f at the last time before (or including) t that p was true
  - if p is not true at any time up to and including t, then `last` $p\ f\ t\ =$ `x`
- Lifted operators (bigger versions of familiar symbols):
  - $\forall\, t.\ \mathsf{1}(t)\ =\ 1$
  - $\forall\, t.\ (f_1 \boldsymbol{+} f_2)(t)\ =\ f_1(t) + f_2(t)$
  - $\forall\, t.\ (f_1 \,\mathbf{?}\, f_2 \,\mathbf{:}\, f_3)(t)\ =\ f_1(t)?f_2(t){:}f_3(t)$
  - $f_1\ \equiv\ f_2$ means $\forall\, t.\ f_1(t) = f_2(t)$

Edge Operators

- `posedge` f `0` is false
- $\forall\, t > 0.$

  $\texttt{posedge}\ f\ t\ =\ (f(t{-}1) \neq 1 \wedge f(t) = 1)$
- `last (posedge clk)` $f\ t$ is the value of f at the last time before or equal to t when `clk` has just had a rising edge.
- `negedge` f `0` is false
- $\forall\, t > 0.$

  $\texttt{negedge}\ f\ t\ =\ (f(t{-}1) \neq 0 \wedge f(t) = 0)$
- `last (negedge clk)` $f\ t$ is the value of f at the last time before or equal to t when `clk` has just had a falling edge.
- The trace `previous(`f`)` is defined by:
  - `previous(`f`)(0) = x`
  - `previous(`f`)(`t`+1) = `f`(`t`)`

Trace Semantics of Flip-flop

HDL description of a negative edge triggered flip-flop

```
forever @(negedge clk) q := d
```

generates the step:

```
0 : @(negedge clk)
      pc := 0
      q := d
```

denotes (after some simplification):

$$
\begin{array}{l}
\texttt{d(0)=X} \wedge \texttt{clk(0)=X} \wedge \texttt{q(0)=X} \wedge \\
\forall t\texttt{>0. } \textit{if } \texttt{clk(}t\texttt{-1)}\neq\texttt{0} \wedge \texttt{clk(}t\texttt{)=0} \\
\qquad\qquad \textit{then } \texttt{q(}t\texttt{)=d(}t\texttt{)} \\
\qquad\qquad \textit{else } \texttt{q(}t\texttt{)=q(}t\texttt{-1)}
\end{array}
$$

which is equivalent to:

$$
\begin{array}{l}
\texttt{d(0)=X} \wedge \texttt{clk(0)=X} \wedge \texttt{q(0)=X} \wedge \\
\texttt{q} \;\equiv\; \texttt{last(negedge clk)d}
\end{array}
$$

Trace Semantics of Latch

HDL description of a transparent latch

```
forever @(clk or d) if clk then q := d
```

generates the step:

```
0 : @(clk or d)
    pc := 0
    q := clk ? d : previous(q)
```

denotes (after some simplification):

$$
\begin{array}{l}
\texttt{clk(0)=X} \wedge \texttt{d(0)=X} \wedge \texttt{q(0)=X} \wedge \\
\forall t\texttt{>0.}\ \textit{if}\ \texttt{clk(}t\texttt{-1)}\neq\texttt{clk(}t\texttt{)} \vee \texttt{d(}t\texttt{-1)}\neq\texttt{d(}t\texttt{)} \\
\qquad\quad \textit{then}\ \texttt{q(}t\texttt{)=clk(}t\texttt{)?d(}t\texttt{):q(}t\texttt{-1)} \\
\qquad\quad \textit{else}\ \texttt{q(}t\texttt{)=q(}t\texttt{-1)}
\end{array}
$$

which is (believed to be) equivalent to:

$$
\begin{array}{l}
\texttt{clk(0)=X} \wedge \texttt{d(0)=X} \wedge \texttt{q(0)=X} \wedge \\
\texttt{q} \equiv \texttt{clk}\,?\,\texttt{d}\,:\,\texttt{last(negedge clk)q}
\end{array}
$$

Trace Level Equivalence

Consider a flip-flop connected to a multiplexer:

```
forever @(negedge clk) i = d

forever @(clk or d or i) q = clk ? d : i
```

which are equivalent to:

d(0)=X $\wedge$ clk(0)=X $\wedge$ q(0)=X $\wedge$
i $\equiv$ last(negedge clk)d

and

clk(0)=X $\wedge$ d(0)=X $\wedge$ i(0)=X $\wedge$ q(0)=X $\wedge$
q $\equiv$ clk ? d : i

Hence via substitution with $\equiv$:

clk(0)=X $\wedge$ d(0)=X $\wedge$ q(0)=X $\wedge$
q $\equiv$ clk ? d : last(negedge clk)d

Which is the trace semantics of a transparent latch

```
forever @(clk or d) if clk then q := d
```

Cycle Semantics

- Cycle semantics is derived from trace semantics
  - sequential (Mealy) machine
  - state transitions determined by a clock
  - clock event ('tick') usually `posedge clk`
- Only certain design styles admit a cycle semantics
  - transparent latches don't abstract well

Cycle Semantics Example

Consider

```
forever
 @(posedge clk)
 if (set) q = 0; else q = q + d;
```

Trace semantics:

$\mathtt{d(0)=X} \wedge \mathtt{set(0)=X} \wedge \mathtt{clk(0)=X} \wedge \mathtt{q(0)=X} \wedge$
$\mathtt{q} \equiv \mathtt{last(posedge\ clk)(set}\,?\,0 : \mathtt{(previous(q)} + \mathtt{d))}$

Cycle semantics given by f:

$$f(\mathtt{d},\mathtt{set},\mathtt{q}) \;=\; \mathtt{set}\ ?\ 0\ :\ \mathtt{q}+\mathtt{d}$$

All variables in initial state have value X

For the future ...

- Get a compositional model of event semantics
  - may not be possible
  - would be nice, but not essential
- Verify sufficient conditions for trace and cycle semantics to be correct abstractions
- Automate semantics extraction
  - link to proof tools
- Systems on a chip?