Lustre: a synchronous dataflow language

Design of reactive systems:
- run in an infinite loop, and
- produce an output every $n$ milliseconds
Lustre: a synchronous dataflow language

Design of **reactive** systems:
- run in an infinite loop, and
- produce an output every $n$ milliseconds

<table>
<thead>
<tr>
<th></th>
<th>$in_0$</th>
<th>$in_1$</th>
<th>$in_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>clock</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>CPU</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Design of **reactive** systems:

- run in an infinite loop, and
- produce an output every $n$ milliseconds
Exercises

Model a switch with two buttons, Set and Reset.

\[\text{node Switch}( \text{Set, Reset, Init : bool } ) \text{ returns ( State : bool )};\]

such that:

- pressing Set turns the switch on;
- pressing Reset turns the switch off;
- the initial position of the switch is determined by a third signal Init if Set and Reset are initially both unpressed.
Exercises

Model a switch with two buttons, Set and Reset.

\[
\text{node Switch( Set, Reset, Init : bool ) returns ( State : bool );}
\]

such that:

- pressing Set turns the switch on;
- pressing Reset turns the switch off;
- the initial position of the switch is determined by a third signal \( \text{Init} \) if Set and Reset are initially both unpressed.

\[
\text{node Switch( Set, Reset, Init : bool ) returns ( X : bool );}
\]

let

\[
X = \begin{array}{ll}
\text{if Set then true} & \\
\text{else if Reset then false} & \\
\text{else (Init -> pre X)} & 
\end{array}
\]

tel
Exercises

Model a switch with two buttons, Set and Reset.

```plaintext
node Switch( Set, Reset, Init : bool ) returns (  
  State : bool );
```

such that:

- pressing Set turns the switch on;
- pressing Reset turns the switch off;
- the initial position of the switch is determined by a third signal Init
  if Set and Reset are initially both unpressed.

Equivalently:

```plaintext
node Switch( Set, Reset, Init : bool )
returns ( X : bool );
let
  X = Set or (not Reset and (Init -> pre X)) ;
tel
```
node ??? (r, b: bool) returns (out: int);
let

out = if r then 0
  else if b then (0 -> pre out) + 1
  else (0 -> pre out);

tel
node ??? (r, b: bool) returns (out: int);

let

\[
\text{out} = \begin{cases} 
\text{if } r \text{ then } 0 \\
\text{else if } b \text{ then } (0 \rightarrow \text{pre out}) + 1 \\
\text{else } (0 \rightarrow \text{pre out})
\end{cases}
\]

tel

\[
\begin{array}{ccc}
\text{r}_0 & \text{b}_0 \\
0 & 1 \\
\downarrow & \downarrow \\
\text{out at 0}
\end{array}
\quad
\begin{array}{ccc}
\text{r}_1 & \text{b}_1 \\
0 & 1 \\
\downarrow & \downarrow \\
\text{out at 1}
\end{array}
\quad
\begin{array}{ccc}
\text{r}_2 & \text{b}_2 \\
1 & 1 \\
\downarrow & \downarrow \\
\text{out at 2}
\end{array}
\]
node ??? (r, b: bool) returns (out: int);

let

\[
out = \begin{cases} 
0 & \text{if } r \\
(0 \rightarrow \text{pre out}) + 1 & \text{if } b \\
(0 \rightarrow \text{pre out}) & \text{else}
\end{cases}
\]

tel
Counter with reset:

\[\text{node } ??? \ (r, b: \ \text{bool}) \ \text{returns} \ (\text{out}: \ \text{int});\]

\[
\text{let out = if } r \ \text{then } 0 \\
\quad \text{else if } b \ \text{then} \ (0 \rightarrow \text{pre out}) + 1 \\
\quad \text{else} \quad (0 \rightarrow \text{pre out});
\]

tel
Counter with reset:

```plaintext
node cnt (r, b: bool) returns (out: int);
var pre_out : int;
let pre_out = 0 -> pre out;
  out = if r then 0
       else if b then pre_out + 1
       else pre_out;
end
```

<table>
<thead>
<tr>
<th>r</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- out at 0: 1
- out at 1: 2
- out at 2: 0
Exercises

Counter with reset:

```haskell
node cnt (r, b: bool) returns (out: int);
var pre_out: int;
let pre_out = 0 -> pre out;
out = if r then 0
    else if b then pre_out + 1
    else pre_out;
tel
```

<table>
<thead>
<tr>
<th>r</th>
<th>b</th>
<th>pre_out_1</th>
<th>pre_out_0</th>
<th>pre_out_1</th>
<th>pre_out_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

pre_out_1 = nil

out at 0 = 1

out at 1 = 2

out at 2 = 0
Once defined, a node can be used as a basic operator
Modularity

Once defined, a node can be used as a basic operator

What does A look like?

\[ X = \text{true} \rightarrow (\text{pre} \ A = 3) \]
\[ A = \text{cnt}(X, \text{true}); \]
Once defined, a node can be used as a basic operator

What does A look like?

\[ X = \text{true} \rightarrow (\text{pre } A = 3) \]
\[ A = \text{cnt}(X, \text{true}); \]
\[ A = 0, \]
Modularity

Once defined, a node can be used as a basic operator

What does $A$ look like?

$X = \text{true} \rightarrow (\text{pre } A = 3)$

$A = \text{cnt}(X, \text{true});$

$A = 0, 1,$
Modularity

Once defined, a node can be used as a basic operator

What does $A$ look like?

$$X = \text{true} \rightarrow (\text{pre} \ A = 3)$$

$$A = \text{cnt}(X, \text{true});$$

$A = 0, 1, 2,$
Modularity

Once defined, a node can be used as a basic operator

What does $A$ look like?

$X = \text{true} \rightarrow (\text{pre } A = 3)$

$A = \text{cnt}(X, \text{true});$

$A = 0, 1, 2, 3,$
Once defined, a node can be used as a basic operator

What does $A$ look like?

$$X = \text{true} \rightarrow (\text{pre} \ A = 3)$$

$$A = \text{cnt}(X, \text{true});$$

$$A = 0, 1, 2, 3, 0,$$
Modularity

Once defined, a node can be used as a basic operator

What does $A$ look like?

\[
X = \text{true} \implies (\text{pre } A = 3)
\]

\[
A = \text{cnt}(X, \text{true});
\]

\[
A = 0, 1, 2, 3, 0, 1, 2, 3, 0, 1\ldots
\]
A node can have several outputs:

```plaintext
node MinMax( X : real ) returns ( Min, Max : real );
let
    Min = X -> if (X < pre Min) then X else pre Min ;
    Max = X -> if (X > pre Max) then X else pre Max ;
tel

node minMaxAverage ( X: real ) returns ( Y: real ) ;
var Min, Max: real ;
let
    Min, Max = MinMax(X) ;
    Y = (Min + Max)/2.0 ;
tel
```
Complete example: specification

Stopwatch:

- one integer output: `time` “to display”;
- three input buttons:
  - `on_off` starts and stops the stopwatch,
  - `reset` resets the stopwatch if not running,
  - `freeze` freezes the displayed time if running, cancelled if stopped
Complete example: available nodes

-- Bistable switch
node switch (on, off: bool) returns (state: bool);
let
  state =
    if (false -> pre state) then not off else on;
tel

-- Counts steps if inc is true, can be reset
node counter (reset, inc: bool) returns (out: int);
let
  out =
    if reset then 0
    else if inc then (0 -> pre_out) + 1
    else (0 -> pre_out);
tel

-- Detects raising edges of a signal
node edge (in: bool) returns (out: bool);
let
  out = false -> in and (not pre in);
tel
Unsatisfactory solution not using edge:

```plaintext
node stopwatch (on_off, reset, freeze: bool)
returns (time: int);
var actual_time: int;
    running, frozen: bool;

let

running = switch(on_off, on_off);
 frozen = switch(
    freeze and running, freeze or on_off
);
actual_time = counter(reset and not running, running);
time = if frozen then (0 -> pre time) else actual_time;
tel
```
Satisfactory solution:

```haskell
node stopwatch (on_off, reset, freeze: bool)
returns (time: int);
var actual_time: int;
    running, frozen,
    on_off_pressed, r_pressed, f_pressed: bool;
let
    on_off_pressed = edge(on_off);
    r_pressed = edge(reset);
    f_pressed = edge(freeze);
    running = switch(on_off_pressed, on_off_pressed);
    frozen = switch(
        f_pressed and running, f_pressed or on_off_pressed
    );
    actual_time = counter(r_pressed and not running, running);
    time = if frozen then (0 -> pre time) else actual_time;
tel
```
Part of these notes are based on the following lectures notes:

The Lustre Language — Synchronous Programming
by Pascal Raymond and Nicolas Halbwachs
Verimag-CNRS