# CS:5810 Formal Methods in Software Engineering

## Reactive Systems and the Lustre Language

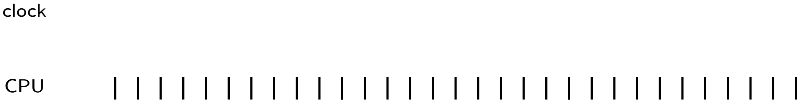
## Part 2

Adrien Champion
adrien-champion@uiowa.edu

# Lustre: a synchronous dataflow language

Design of **reactive** systems:

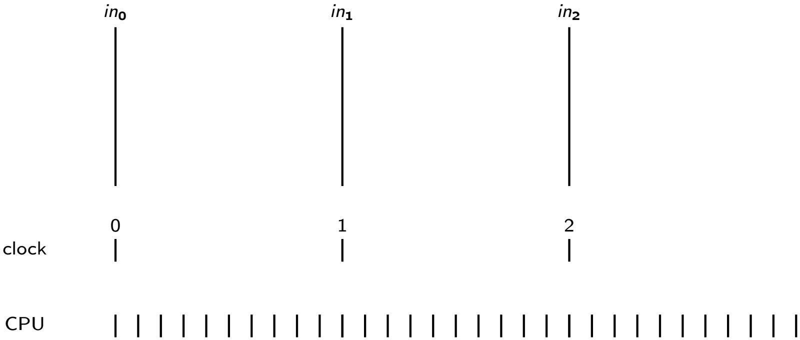
- run in an infinite loop, and
- produce an output every *n* milliseconds



# Lustre: a synchronous dataflow language

Design of **reactive** systems:

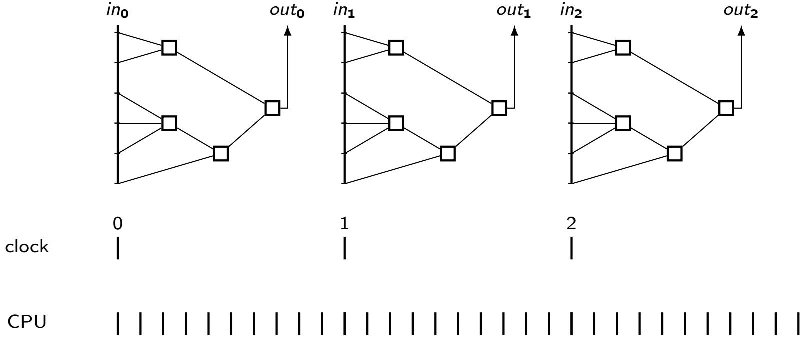
- run in an infinite loop, and
- produce an output every *n* milliseconds



# Lustre: a synchronous dataflow language

Design of **reactive** systems:

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



# Exercises

Design a node

```
node switch (on,off: bool) returns (state: bool);
```

such that:

- `state` raises (false to true) if `on`;
- `state` falls (true to false) if `off`;
- everything behaves as if `state` was false at the origin;
- `switch` must work properly even if `on` and `off` are the same

# Exercises

Design a node

```
node switch (on,off: bool) returns (state: bool);
```

such that:

- state raises (false to true) if `on`;
- state falls (true to false) if `off`;
- everything behaves as if `state` was false at the origin;
- `switch` must work properly even if `on` and `off` are the same

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

# Exercises

Compute the sequence $1,\ 1,\ 2,\ 3,\ 5,\ 8\ \ldots$

# Exercises

Compute the sequence $1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21\ \ldots$

Fibonacci sequence:

$u_0 = 1$

$u_1 = 1$

$u_n = u_{n-1} + u_{n-2} \quad \text{for } n \geq 2$

# Exercises

Fibonacci sequence:

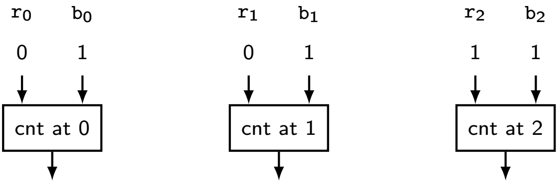
```
node fib (a: bool) returns (uN: int);
let
  uN =
    1 -> pre (
      1 -> uN + pre uN
    );
tel
```

# Exercises

```
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
```

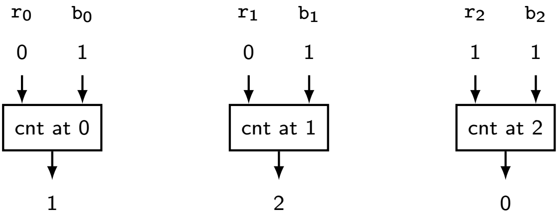
# Exercises

```
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
```



# Exercises

```
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
```



$r_0$ $b_0$

0 1

cnt at 0

1

$r_1$ $b_1$

0 1

cnt at 1

2

$r_2$ $b_2$

1 1

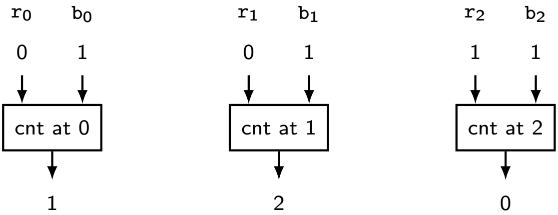
cnt at 2

0

# Exercises

Counter with reset:

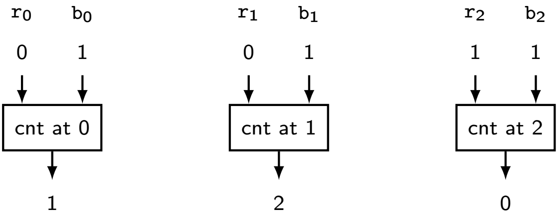
```
node ??? (r,b: bool) returns (out: int);
let

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



# Exercises

Counter with reset:

```
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
```



$r_0$ $b_0$

0 1

cnt at 0

1

$r_1$ $b_1$

0 1

cnt at 1

2

$r_2$ $b_2$

1 1

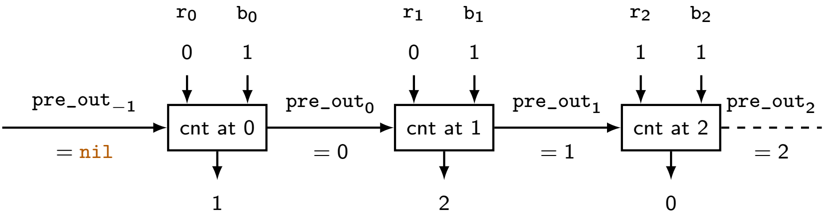
cnt at 2

0

# Exercises

Counter with reset:

```
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
```



# Modularity

- Once defined, a node can be used as a basic operator
- Instantiation is function-like

# Modularity

- Once defined, a node can be used as a basic operator
- Instantiation is function-like

  What is the output of

```
A = cnt(true -> (pre A = 3), true);
--      |--------r---------| |-i-|
```

# Modularity

- Once defined, a node can be used as a basic operator
- Instantiation is function-like

  What is the output of

```
A = cnt(true -> (pre A = 3), true);
--      |--------r--------| |-i-|
```

  0,

# Modularity

- Once defined, a node can be used as a basic operator
- Instantiation is function-like

  What is the output of

```
A = cnt(true -> (pre A = 3), true);
--      |--------r---------| |-i-|
```

0, 1,

# Modularity

- Once defined, a node can be used as a basic operator
- Instantiation is function-like

  What is the output of

```
A = cnt(true -> (pre A = 3), true);
--       |--------r---------| |-i-|
```

  0, 1, 2,

# Modularity

- Once defined, a node can be used as a basic operator
- Instantiation is function-like

  What is the output of

```
A = cnt(true -> (pre A = 3), true);
--      |--------r---------| |-i-|
```

  0, 1, 2, 3,

# Modularity

- Once defined, a node can be used as a basic operator
- Instantiation is function-like

  What is the output of

```
A = cnt(true -> (pre A = 3), true);
--      |--------r---------| |-i-|
```

0, 1, 2, 3, 0,

# Modularity

- Once defined, a node can be used as a basic operator
- Instantiation is function-like

  What is the output of

```
A = cnt(true -> (pre A = 3), true);
--       |--------r---------| |-i-|
```

  0, 1, 2, 3, 0, 1 , 2, 3, 0, 1 . . .

# Modularity

- Once defined, a node can be used as a basic operator
- Instantiation is function-like

  What is the output of

  ```
  A = cnt(true -> (pre A = 3), true);
  --        |---------r---------| |-i-|
  ```

  0, 1, 2, 3, 0, 1 , 2, 3, 0, 1 . . .
- Several outputs:

  ```
  node minMaxAverage (in: int) returns (out: int);
  var min, max: int;
  let
    out = average(min,max);
    min, max = minMax(in);
  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
```

# Complete example: solution(s)

Unsatisfactory solution not using edge:

```
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
```

# Complete example: solution(s)

Satisfactory solution:

```
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
```

# Specification of Lustre systems

Past Time Linear Temporal Logic (ptLTL):

- Safety properties:

  "$\mathcal{P}$ holds in all steps"

  e.g., "the output of this node is positive"

- Liveness property:

  "if $\mathcal{P}$ becomes true, then eventually $\mathcal{P}'$ will become true"

  e.g., "if the brake pedal is pressed, then eventually the car should brake"

# Specification of Lustre systems

Past Time Linear Temporal Logic (ptLTL):

- Safety properties:

  "$\mathcal{P}$ holds in all steps"

  e.g., "the output of this node is positive"

- Liveness property:

  "if $\mathcal{P}$ becomes true, then eventually $\mathcal{P}'$ will become true"

  e.g., "if the brake pedal is pressed, then eventually the car should brake"

- In practice, liveness properties are often bounded:

  "if $\mathcal{P}$ becomes true, then $\mathcal{P}'$ will become true in at most $n$ steps"

  Bounded liveness properties can be represented as safety properties

# Specification of Lustre systems: examples

```
node count (start: bool) returns (time: int);
var started: bool;
let
  started = start or (false -> pre started);
  time = if started then (0 -> pre time) + 1
                    else (0 -> pre time);
tel
```

# Specification of Lustre systems: examples

```
node count (start: bool) returns (time: int);
var started: bool;
let
  started = start or (false -> pre started);
  time = if started then (0 -> pre time) + 1
                    else (0 -> pre time);
tel
```

```
node nodeName (in: ...) returns (out: ...);
var p1,p2,ok: bool; n: int;
let
  -- ...
  p1 = -- trigger
  p2 = -- consequence
  n  = -- a constant
  ok = ( count(p1) >= n ) => p2;
tel
```

# Specification of Lustre systems: examples

Useful nodes for specification:

```
node first (x: int) returns (f: int);
let
  f = x -> pre f;
tel

node soFar (p: bool) returns (out: bool);
let
  out = p -> p and (pre out);
tel

node since (p1, p2: bool) returns (out: bool);
let
  out = p1 or (p2 and (false -> pre out));
tel
```

# Credits

These notes are based on the following lectures notes:

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