Functional Languages for Synchronous Hardware Design and Verification

Gordon J. Pace

Department of Computer Science & AI

University of Malta

What this course is about ...

- Using a functional language to describe, manipulate, generate and verify synchronous hardware.
- An overview of a number of such functional hardware description languages, each with different objectives, different approaches.

What it is not about …

- Circuit design
- Functional programming
- Model checking and hardware verification
- Asynchronous circuits

Even if you'll be expected to know something and will learn more about the first three

Course Structure

- 4 days of 1.5hrs of lectures, 1hr practical session per day.
- The functional language we will be using is Haskell.
- The functional HDL is Lava.
- The verification tool is SMV.
- All this is downloadable as a ready-made package from the course website.

Course Outline

Part 1: Synchronous Circuits and Hardware Description Languages

- Short introduction to synchronous circuits
- Standard hardware description languages

Course Outline

Part 2: Lava

- Embedded languages
- Lava as an embedded HDL in Haskell
- Circuit descriptions in Lava
- Parametrised circuits in Lava
- Higher-order circuits in Lava
- Circuit verification in Lava

Course Outline

Part 3: Writing your own Functional HDL

- Designing a simple HDL for simulation
- Shortcomings of the HDL
- Redesign using naming, monads and non-updatable references
- Verification and netlist generation

Course Outline

Part 4: Embedded Hardware Compilers

- High level hardware design
- Hardware compilation
- Compilation techniques for different languages
- Compilers and verification

Course Outline

Part 5: Other embedded HDLs

- Hawk
- Wired
- reFLect

Functional Languages for Synchronous Hardware Design and Verification

Part 1: An Introduction to Synchronous Circuits and Hardware Description Languages

Gordon J. Pace

Department of Computer Science & AI

University of Malta

Synchronous Hardware

- All the circuits we will describe will use a single global clock controlling the system
- We will assume that the clock is not too fast

Synchronous vs Asynchronous

Synchronous circuits are

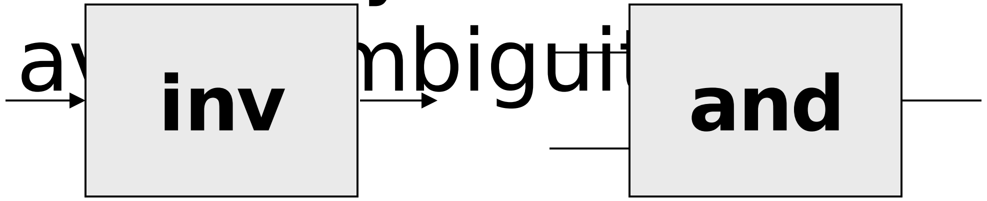
- much easier to reason about
- hence easier to design
- and more reliable.

But

- they are more power hungry
- and slower
- large circuits also have problems with clock skew

Timeless Gates

- Throughout the course figures will use named boxes for gates to avoid confusion
- We will generally be using **inv (or not)**, **and**, **or**, **nor**, **nand** and **xor** gates
- If necessary we add arrows to wires to avoid ambiguity

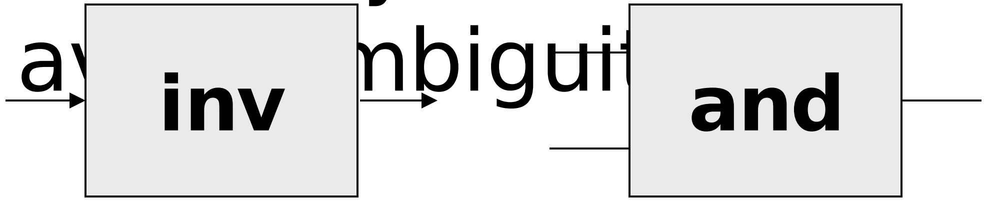


etc

Timeless Gates

- Throughout the course figures will use n o avoid confu
- We w **v (or not)**, d **xor** gates
- If necessary we add arrows to wires to a mbiguit

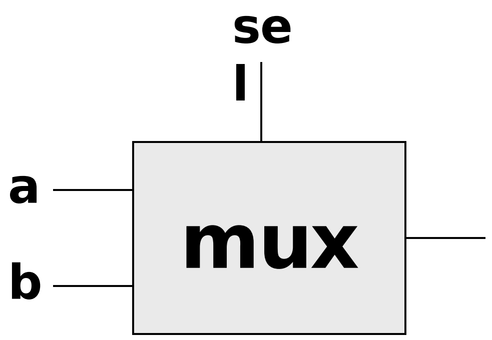
Sometimes, to avoid clutter, we use a small circle on an input wire of a gate to indicate that that input is inverted

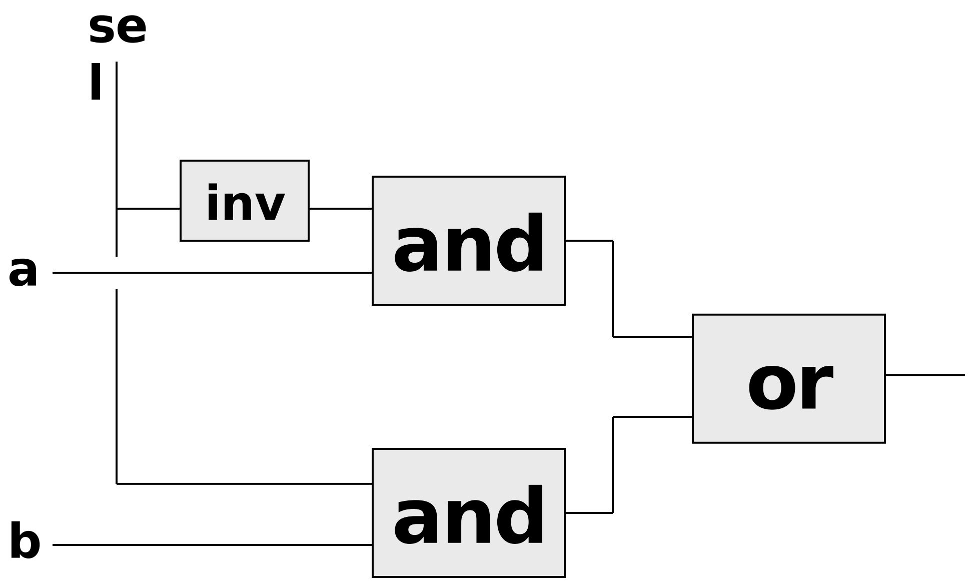


etc

Building a multiplexer

When the input **sel** is low output is equal to input **a**, otherwise it is equal to input **b**





What About Correctness?

We need a specification:

$P \equiv (\mathbf{sel} \Rightarrow \mathbf{out} = \mathbf{b}) \wedge (\neg\, \mathbf{sel} \Rightarrow \mathbf{out} = \mathbf{a})$

The circuit behaviour corresponds just to the gates replaced by Boolean operators:

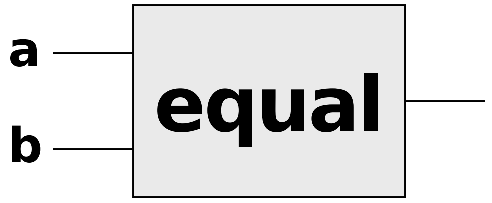
$C \equiv \mathrm{out} = ((\neg\, \mathbf{sel} \wedge \mathbf{a}) \vee (\mathbf{sel} \wedge \mathbf{b}))$

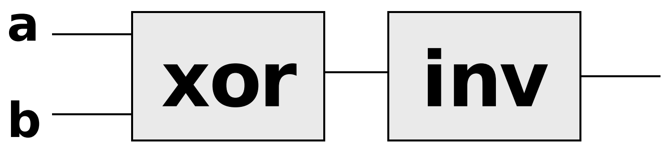
Now it's just a matter of showing that:

$$\forall \mathbf{sel}, \mathbf{a}, \mathbf{b}\ .\ C \Rightarrow P$$

Building an Equality Checking Circuit

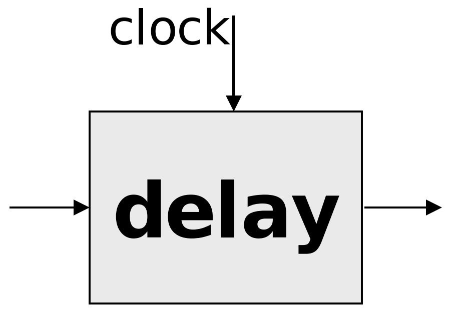
Output whether inputs **a** and **b** are equal.





Circuits with Memory

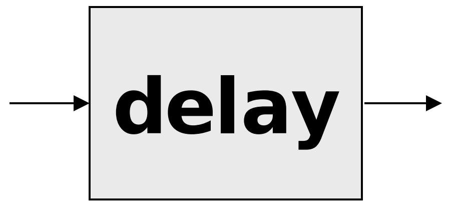
We will use one basic memory gate, a delay:



The **clock** signal is global to the whole circuit, hence we avoid drawing it, making it implicit

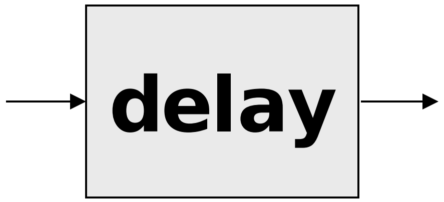
Circuits with Memory

We will use one basic memory gate, a delay:



Circuits with Memory

We will use one basic memory gate, a delay:

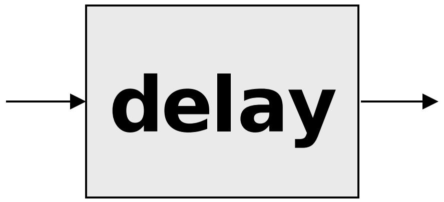


Note that the values on wires are now no longer a boolean value, but a stream of boolean values:

$$\text{TIME} \rightarrow \text{Boolean}$$

Circuits with Memory

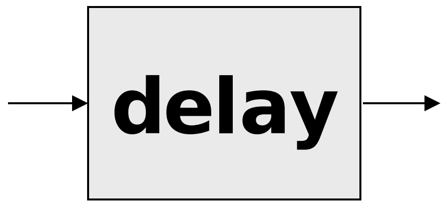
We will use one basic memory gate, a delay:



$\forall$ t: TIME . out(t) = in(t-1)

Circuits with Memory

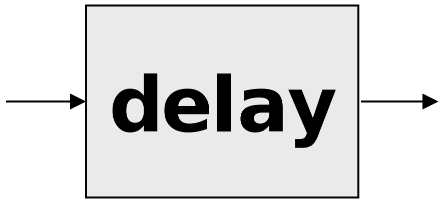
We will use one basic memory gate, a delay:



in	mem	out	mem'
0	0	0	0
0	1	1	0
1	0	0	1
1	1	1	1

Circuits with Memory

We will use one basic memory gate, a delay:

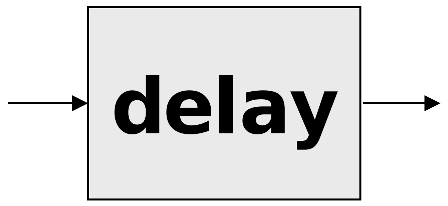


$$(\mathbf{out} = \mathbf{mem}) \wedge$$

$$(\mathbf{mem'} = \mathbf{in})$$

Circuits with Memory

We will use one basic memory gate, a delay:

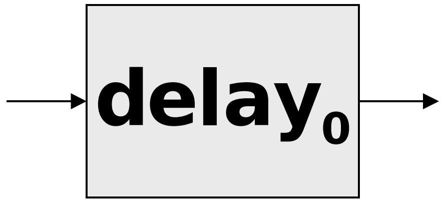


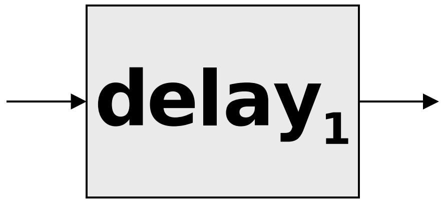
What about the initial value of the output (and memory)?

For the purposes of this course we will initialise it to a fixed value shown on the delay

Circuits with Memory

We will use one basic memory gate, a delay:



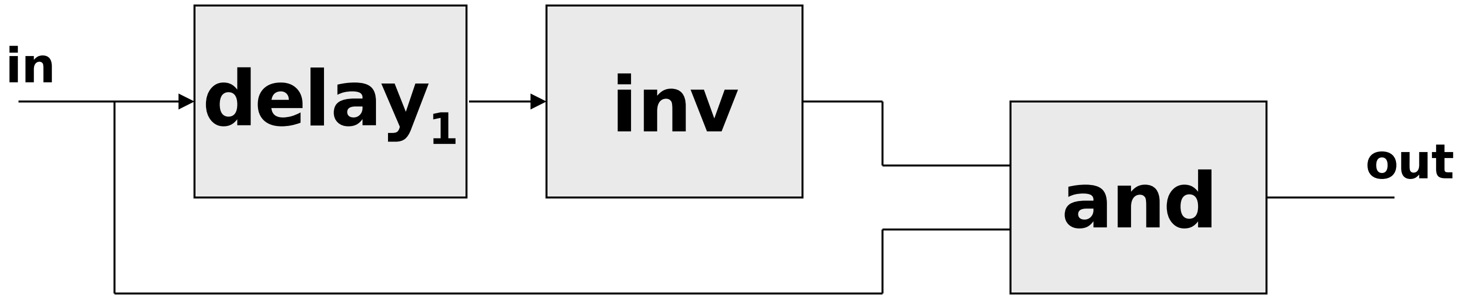


What about the initial value of the output (and memory)?

For the purposes of this course we will initialise it to a fixed value shown on the delay

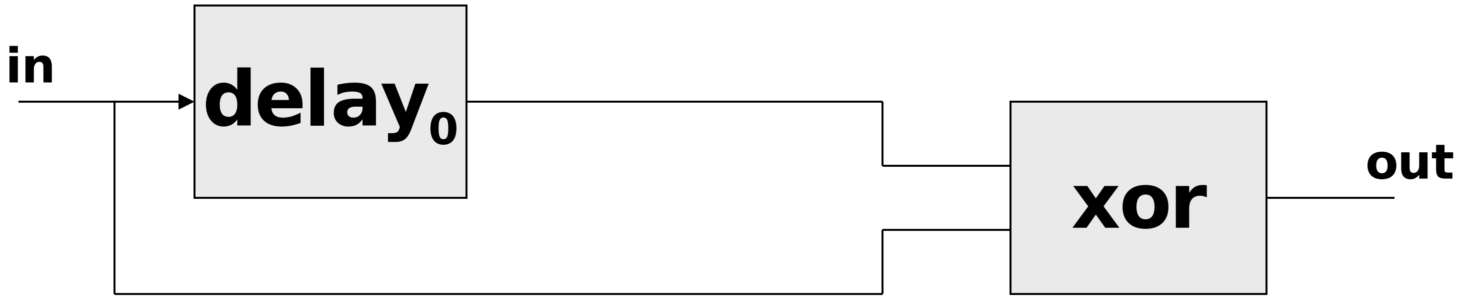
Detecting a Rising Edge

When the input **in** goes from low to high, output high, otherwise output low



Detecting Any Edge

When the input **in** changes its value, output high, otherwise output low

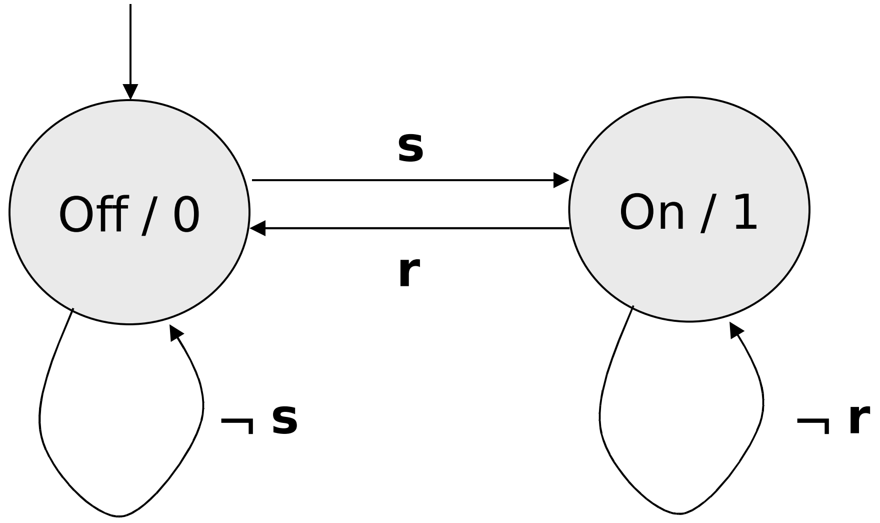


A Set-Reset Memory

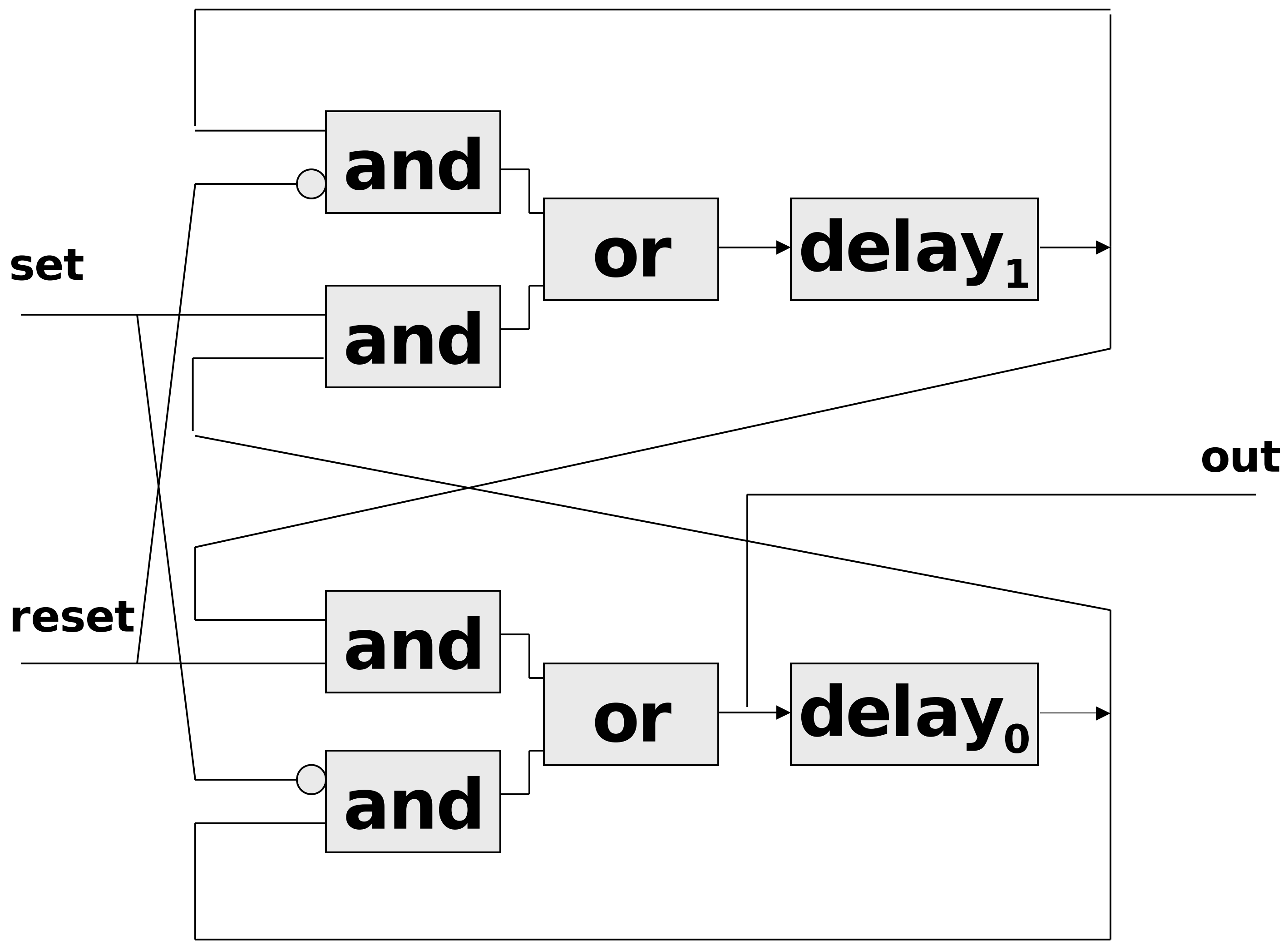
Output is always equal to the memory, which starts off as low, and is set to high when input **s** is high, and reset to low when input **r** is high.

A Set-Reset Memory

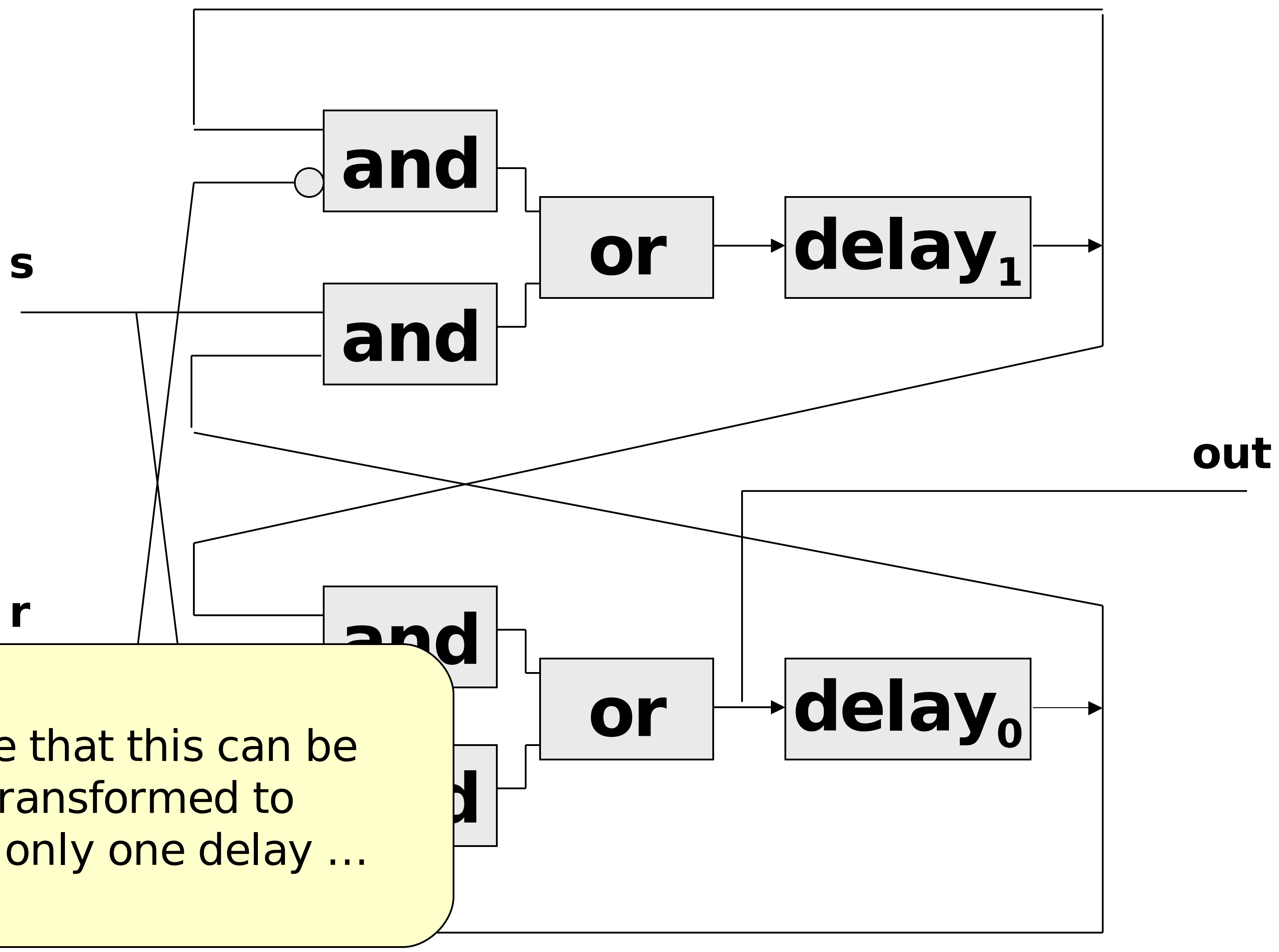
Output is always equal to the memory, which starts off as low, and is set to high when input **s** is high, and reset to low when input **r** is high.



A Set-Reset Memory



A Set-Reset Memory



Note that this can be transformed to use only one delay …

Block Diagram Descriptions of Circuits

Describing circuits using block diagrams is useful, but has various disadvantages:

- Notation is cumbersome;
- Does not scale up;
- We have various ways of describing the same circuit;
- We end up worrying about placement *and* functionality at the same time;
- Compositionality is not straightforward.

Block Diagram Descriptions of Circuits

Describing circuits using block diagrams is useful, but has various disadvan

- Notation
- Does not
- We have the same
- We end up worrying about placement *and* functionality at the same time;
- Compositionality is not straightforward.

The solution?
Use a text based HDL to describe the circuits

Textual Descriptions

Structural descriptions: Describe the circuits in terms of their submodules and gates.

Behavioural descriptions: Describe the behaviour of the circuit in terms of a software program which does not necessarily have an automatically deducible hardware counterpart.

Synthesisable descriptions: A program-like description which can be automatically compiled down to a circuit.

Textual Descriptions

Structural descriptions: Describe the circuits in terms of their submodules and gates.

Behavioural descri the behaviour of the ci software program which does not necessarily have an automatically deducible hardware counterpart.

This is what we will be mainly talking about in this course

Synthesisable descriptions: A program-like description which can be automatically compiled down to a circuit.

Textual Descriptions

Structural descriptions: Describe the circuits in terms of their submodules and gates.

Behavioural descript[illegible]**s:** Describe the behaviour of the ci[illegible] software program [illegible] necessarily have a [illegible] deducible hardware counterpart.

These are primarily used for testing and we will not be dealing with them

Synthesisable descriptions: A program-like description which can be automatically compiled down to a circuit.

Textual Descriptions

Structural descriptions: Describe the circuits in terms of their submodules and gates.

Behavioural descri the behaviour of the ci software program w necessarily have an automatically deducible hardware counterpart.

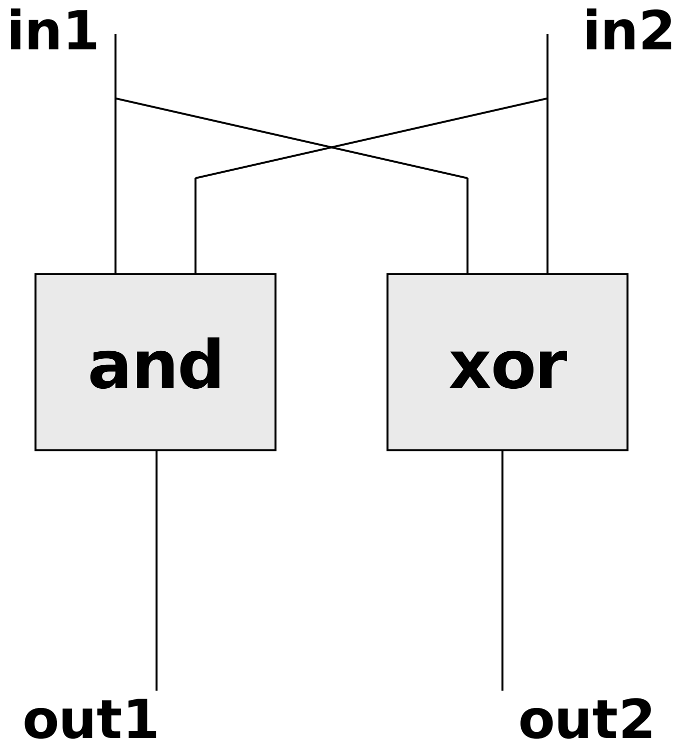
We will be talking about these in part 4 of the course

Synthesisable descriptions: A program-like description which can be automatically compiled down to a circuit.

Describing Circuits in Verilog

- Verilog is an industry standard text-based HDL (similar in many respects to VHDL)
- The language was primarily aimed at simulation but now tools extend its functionality in various ways.
- We’re just looking at this to compare to functional language based techniques. It is far, far from a comprehensive overview.

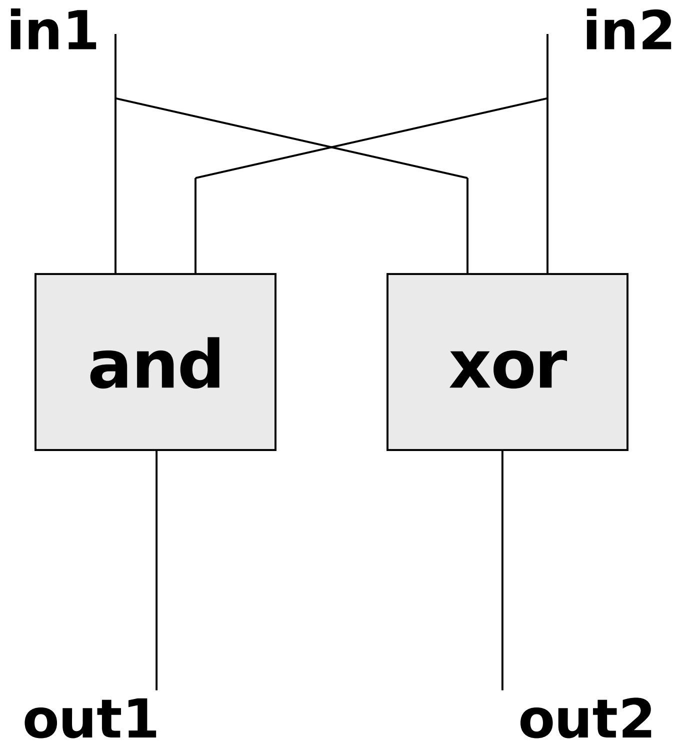
Adding Together Two 1-bit Numbers to a 2-bit Number



Specification:

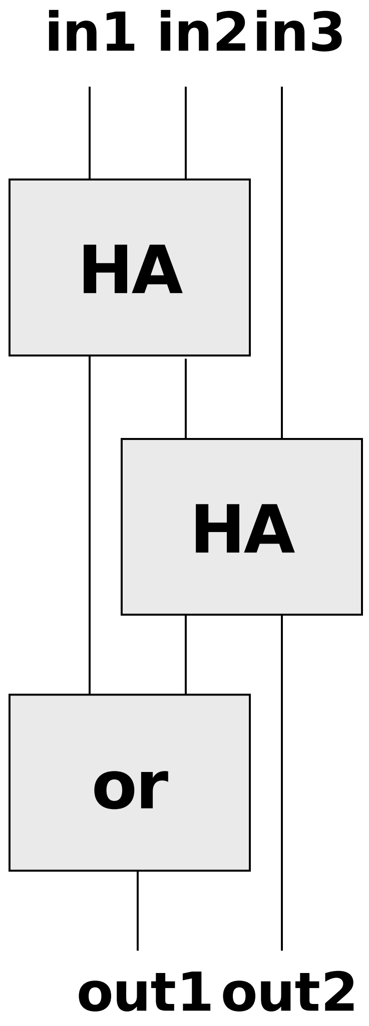
in1 + **in2** = 2 * **out1** + **out2**

Adding Together Two 1-bit Numbers to a 2-bit Number



```
module halfadder(out1, out2 , in1,
in2);
   input in1, in2;
   output out1, out2;

   and AndGate1(out1, in1, in2);
   xor XorGate1(out2, in1,in2);
endmodule;
```

Adding Together Three 1-bit Numbers to a 2-bit Number

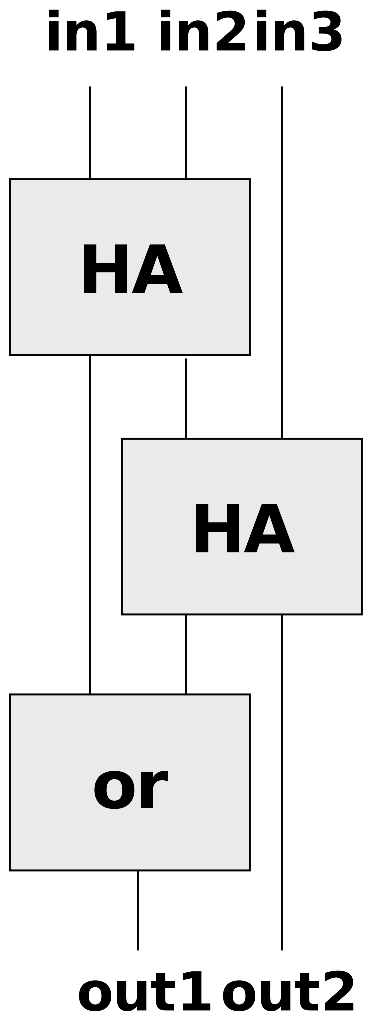


Specification:

in1 + **in2** + **in3** =

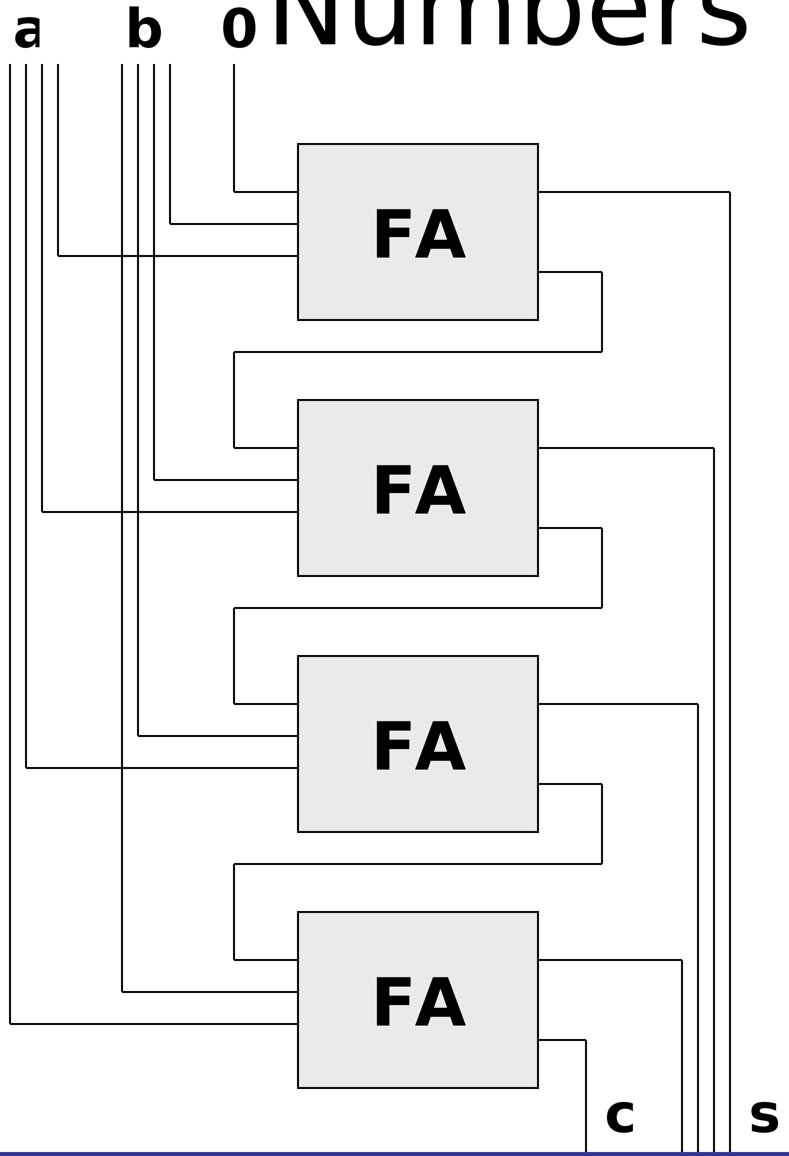
2 * **out1** + **out2**

Adding Together Three 1-bit Numbers to a 2-bit Number



```
module fulladder(out1, out2, in1, in2,
in3);
   input in1, in2, in3;
   output out1, out2;

   halfadder HA1(m1, m2, in1, in2);
   halfadder HA2(m3, out2, m2, in3);
   or OrGate1(out1, m1, m3);
endmodule;
```

Adding Together Two 4-Bit Numbers to a 5-Bit Number



```
module adder4(c, s, a, b);
   input [3:0] a, b;
   output [3:0] s;
   output c;

   wire [4:0] m;

   assign m[0] = 0;
   assign c = m[3];
   fulladder FA1(m[1], s[0], a[0], b[0], m[0]);
   fulladder FA2(m[2], s[1], a[1], b[1], m[1]);
   fulladder FA3(m[3], s[2], a[2], b[2], m[2]);
   fulladder FA4(m[4], s[3], a[3], b[3],
```

Observations

- It is possible to describe hardware modularly using Verilog or a similar language.
- Without resorting to extensions, it is however, impossible to describe general circuits (eg an n-bit adder).

Exercises (1)

- The *majority3* circuit is a combinational circuit with 3 inputs *x1*, *x2*, *x3*, and one output *y*. The circuit outputs high if at least 2 of the inputs are high, and outputs low if at least two of the inputs are low.
  - Design the *majority3* circuit in terms of logical gates. You may use block diagrams, textual notation, or any other description method you are familiar with.
  - What is the general scheme to design a *majorityk* circuit, that has *k* inputs?

Exercises (2)

- Design an assignment circuit with two input wires *assign* and *value*, and one output *out*. The output is initially low, and changes value to match *value* whenever *assign* is high. Otherwise, the value of *out* does not change.
- The *always* circuit has one input *in*, and one output *out*. As long as the input is high, the output *out* is also high. But as soon as *in* is low, then *out* becomes low and stays low forever. Design the circuit in terms of logical gates and delay components

Exercises (3)

- Design a circuit *all3* with three inputs *a, b* and *c*, and one output *ok.* The output should become true whenever *a, b* and *c* have all three been true at some point in the past (not necessarily at the same time). Generalise to *allk,* which works with *k* inputs.

  Write a program, which given a number *k*, returns (using a textual description) *allk*.

Exercises (4)

- A stack is data structure which supports three operations: *push*, *pop* and *top*. You will be designing a simple stack in hardware. For simplicity, the data elements that are going to be stored in the stack are booleans. The stack is implemented as a stateful circuit with three inputs, called *push*, *pop*, and *data*, and one output, called *top:*

  If *push* is high, then the *data* element is pushed on the stack. If *pop* is high, then the top of the stack is taken away, and the *data* input is ignored. If neither *push* nor *pop* is high, nothing happens. At all times, the output *top* reflects the value of the top of the stack, so no special request to see the top of the stack is required.

  It is unspecified what happens when *push* and *pop* are high at the same time, when the stack is empty and *pop* is high, and when the stack is full and *push* is high.

Exercises

- Design a stack which can store at least 4 data elements. Try to use a design consisting of four identical cells, each corresponding to a place in the stack.
- Add two extra outputs, called *empty* and *full*, which are high if the stack is respectively empty or full.
- Add one extra output, called *error*, which becomes high if something has gone wrong.
- What invariants hold for the delay elements in your design? (An invariant is a property which is true at all times).

Functional Languages for Synchronous Hardware Design and Verification

Part 2: Lava – A Hardware Description Language Embedded in Haskell

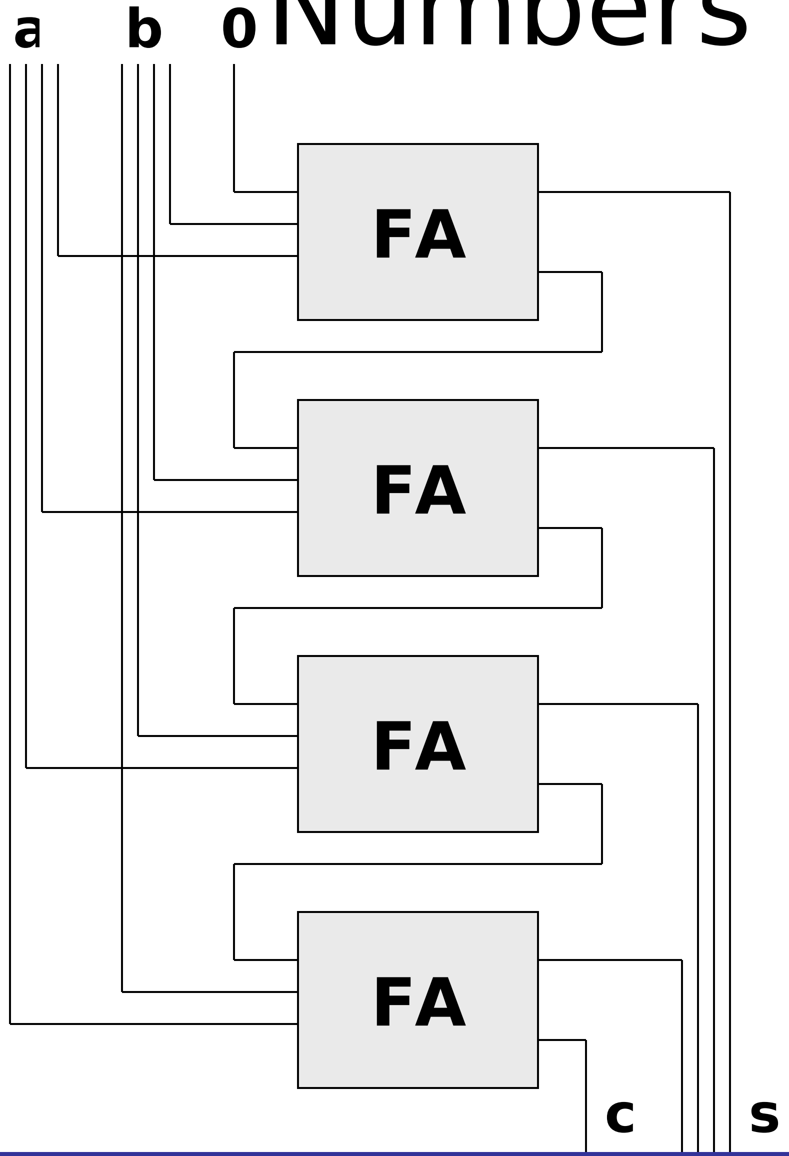
Gordon J. Pace

Department of Computer Science & AI

University of Malta

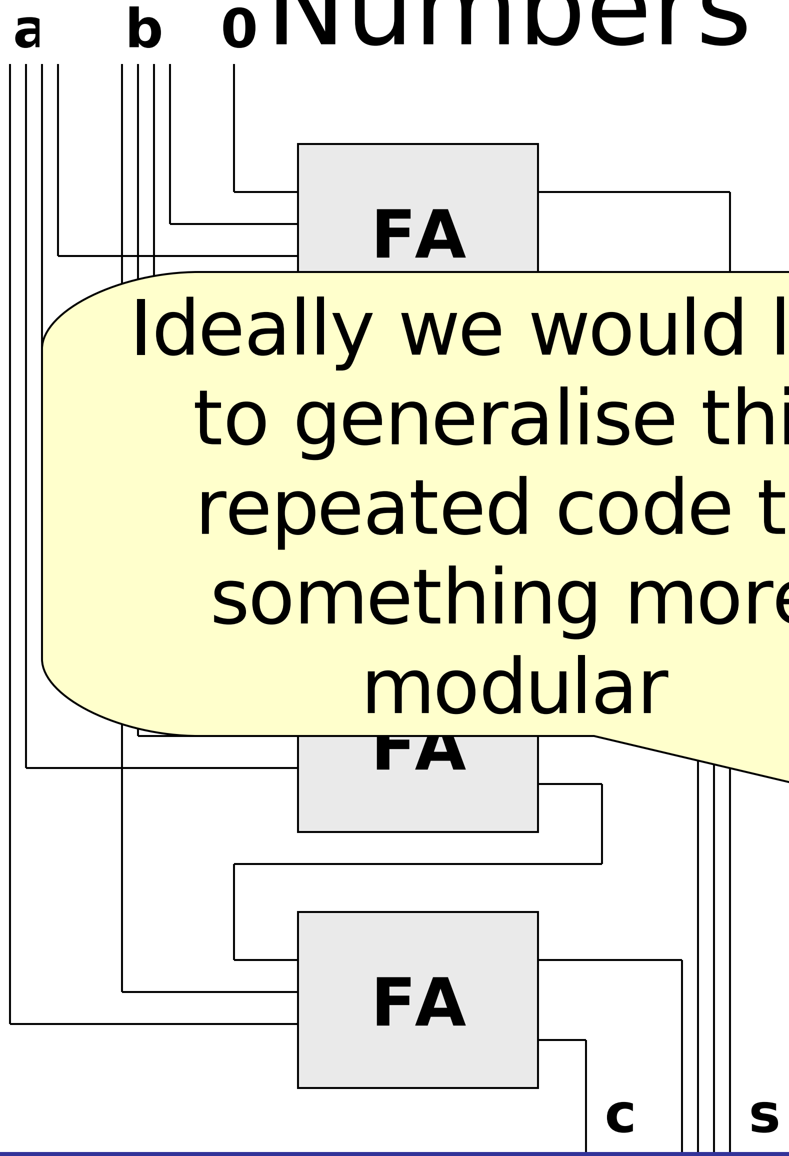
Let’s start by taking another look at that 4-bit adder written in Verilog

Adding Together Two 4-Bit Numbers to a 5-Bit Number



```
module adder4(c, s, a, b);
   input [3:0] a, b;
   output [3:0] s;
   output c;

   wire [4:0] m;

   assign m[0] = 0;
   assign c = m[3];
   fulladder FA1(m[1], s[0], a[0], b[0], m[0]);
   fulladder FA2(m[2], s[1], a[1], b[1], m[1]);
   fulladder FA3(m[3], s[2], a[2], b[2], m[2]);
   fulladder FA4(m[4], s[3], a[3], b[3],
```

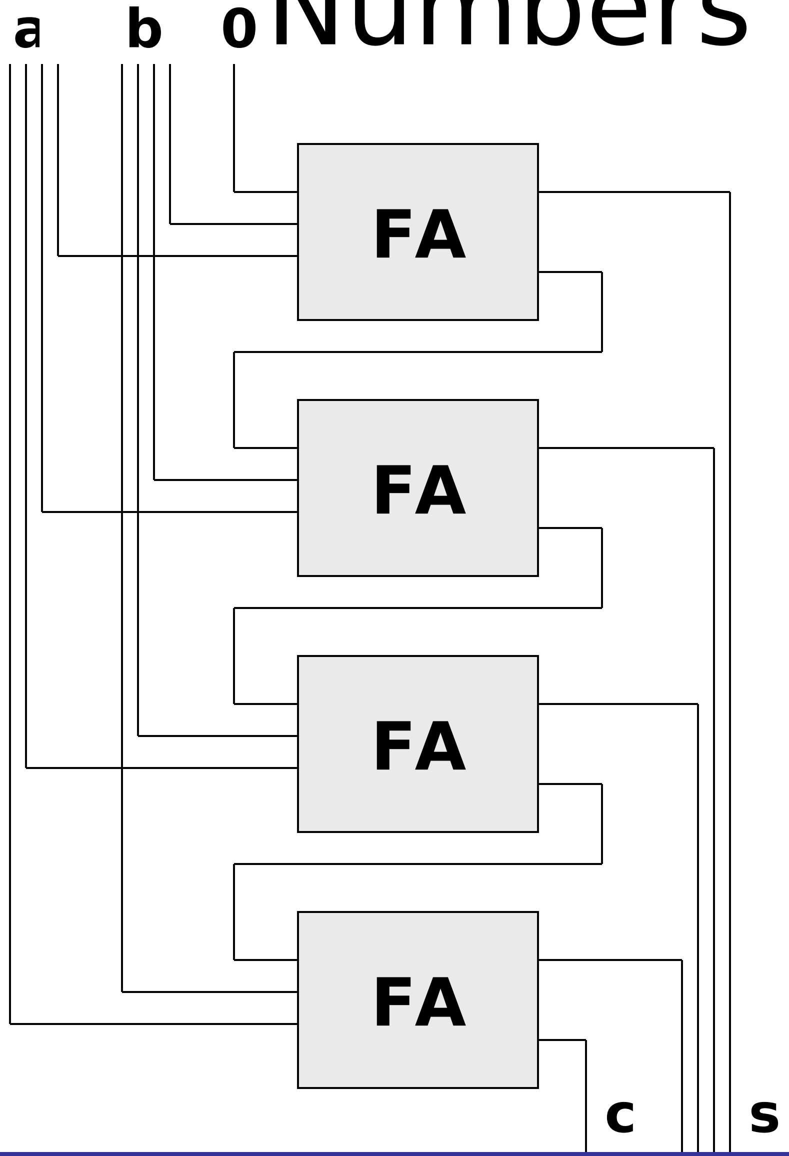
Adding Together Two 4-Bit Numbers to a 5-Bit Number



Ideally we would like to generalise this repeated code to something more modular

```
module adder4(c, s, a, b);
   input [3:0] a, b;
   output [3:0] s;
   output c;

   wire [4:0] m;

   assign m[0] = 0;
   assign c = m[3];
   fulladder FA1(m[1], s[0], a[0], b[0],
m[0]);
   fulladder FA2(m[2], s[1], a[1], b[1],
m[1]);
   fulladder FA3(m[3], s[2], a[2], b[2],
m[2]);
   fulladder FA4(m[4], s[3], a[3], b[3],
```

Adding Together Two 4-Bit Numbers to a 5-Bit Number

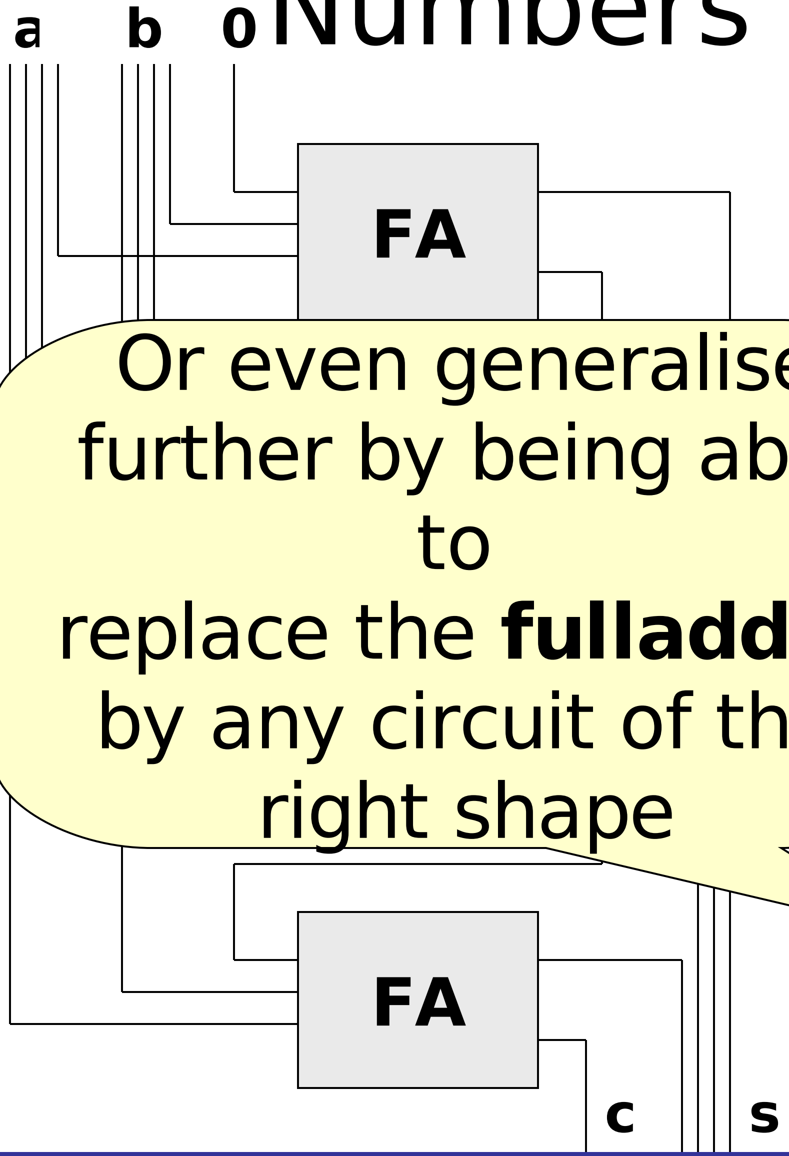


```
module adder(n, c, s, a, b);
   static int n;

   input [n:0] a, b;
   output [n:0] s;
   output c;
   wire [n+1:0] m;

   assign m[0] = 0;
   assign c = m[n];
   staticfor I = 0 to n
      fulladder FA(m[i+1], s[i], a[i], b[i],
m[i]);
   endfor;
endmodule;
```

WARNING

This is not standard Verilog

Adding Together Two 4-Bit Numbers to a 5-Bit Number



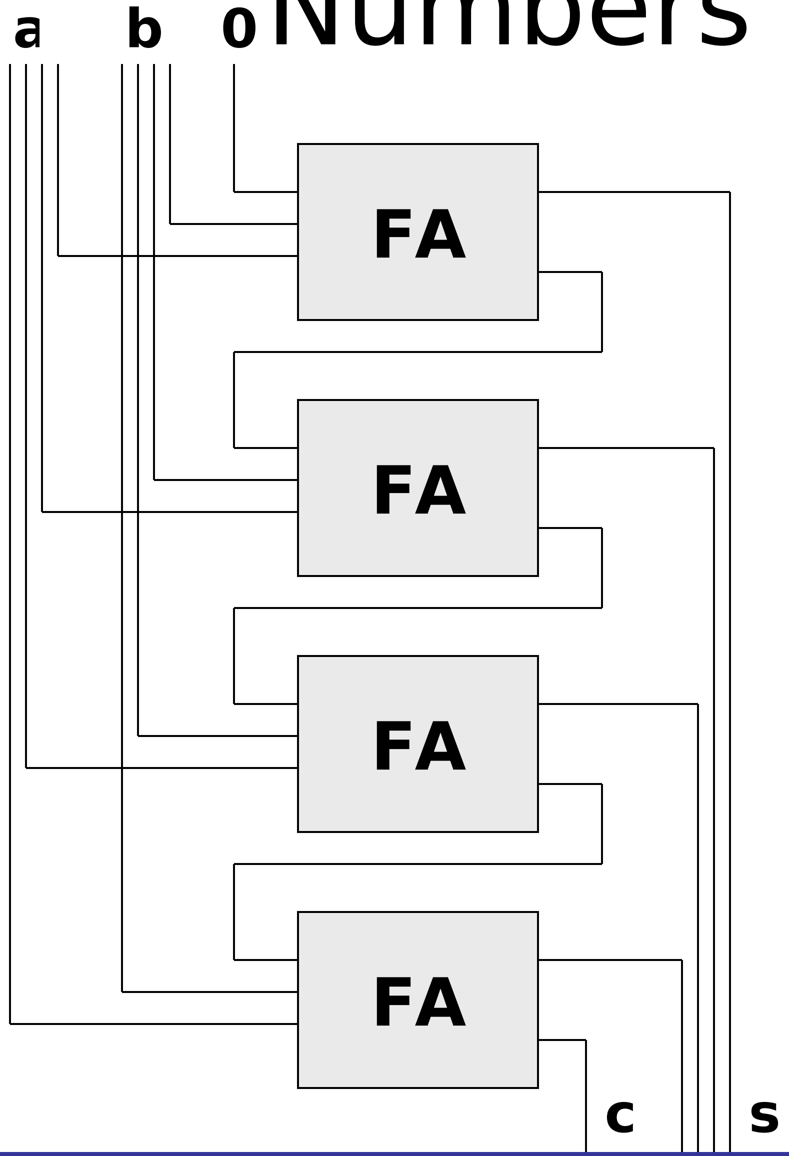
Or even generalise further by being able to replace the **fulladder** by any circuit of the right shape

```
module adder(n, c, s, a, b);
   static int n;

   input [n:0] a, b;
    utput [n:0] s;
    utput c;
    ire [n+1:0] m;

    ssign m[0] = 0;
   assign c = m[n];
   staticfor I = 0 to n
      fulladder FA(m[i+1], s[i], a[i], b[i],
m[i]);
   endfor;
endmodule;
```

WARNING

This is not standard Verilog

Adding Together Two 4-Bit Numbers to a 5-Bit Number



```
module column(blk, n, c, s, a, b);
   circuit … blk;
   …
   assign m[0] = 0;
   assign c = m[n];
   staticfor I = 0 to n
      blk BLK(m[i+1], s[i], a[i], b[i], m[i]);
   endfor;
endmodule;

module adder(n, c, s, a, b);
  …
  column(halfadder, n, c, s, a, b);
endmodule;
```

WARNING

This is not standard Verilog at all!

A Two-Level Language

- What we need are in fact two languages:
  - A basic structural HDL.
  - A richer programming language which can access the structural HDL to generate regular circuits – the meta-language.
- Most extensions to Verilog and VHDL take a two-language approach, with a simple meta-language.
- An alternative is to *embed* an HDL in a standard language.

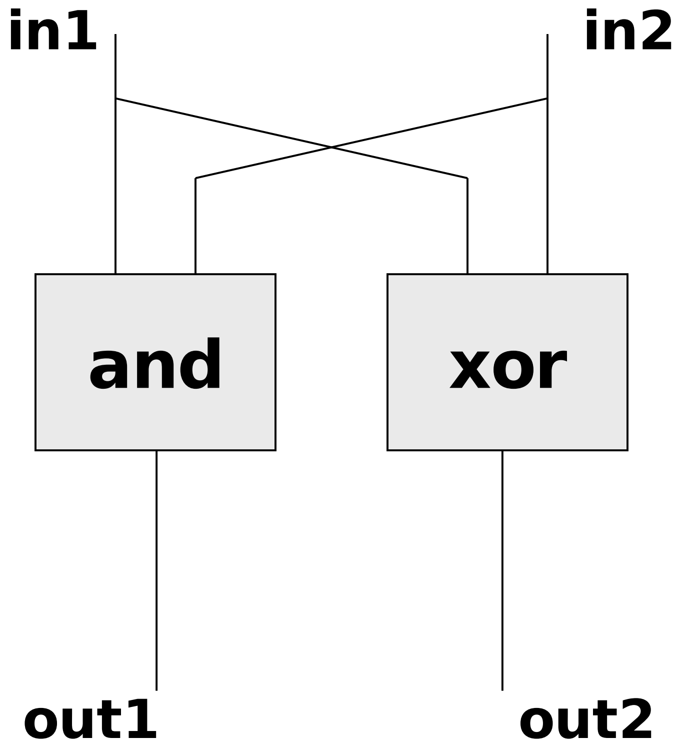
Embedded Languages

- Programs in the *embedded language* are just data objects within the *host language*, allowing:
  - Generation
  - Analysis
  - Manipulation
  - Semantics
  - Tools for free

Lava

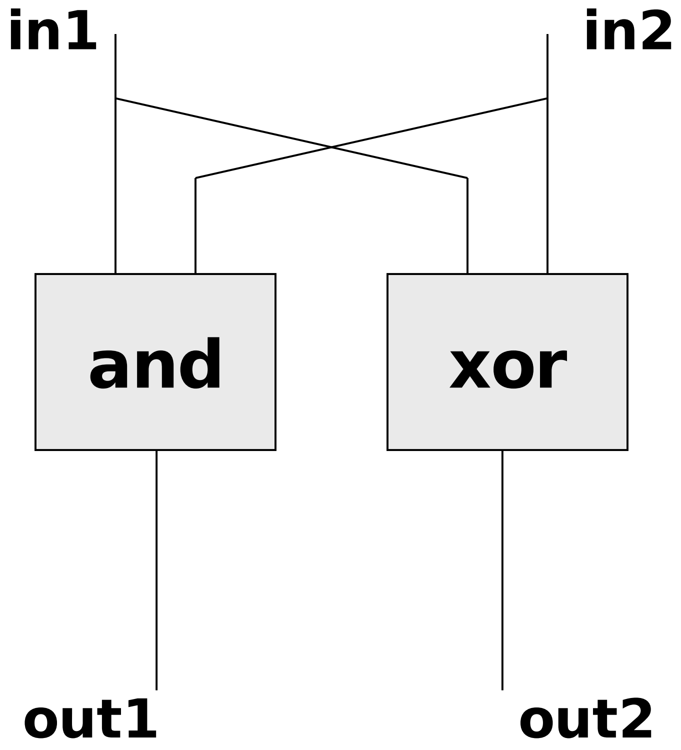
- Lava is an HDL embedded in Haskell
- Developed in Chalmers University in Gothenburg, but also separately developed and used in Xilinx.
- Allows description, verification, simulation, manipulation of synchronous hardware.
- Allows higher-order description of circuits.

Combinational Circuits in Lava



```
halfadder (in1, in2) = (out1, out2)
  where
    out1 = and2 (in1, in2)
    out2 = xor2 (in1, in2)
```

Combinational Circuits in Lava

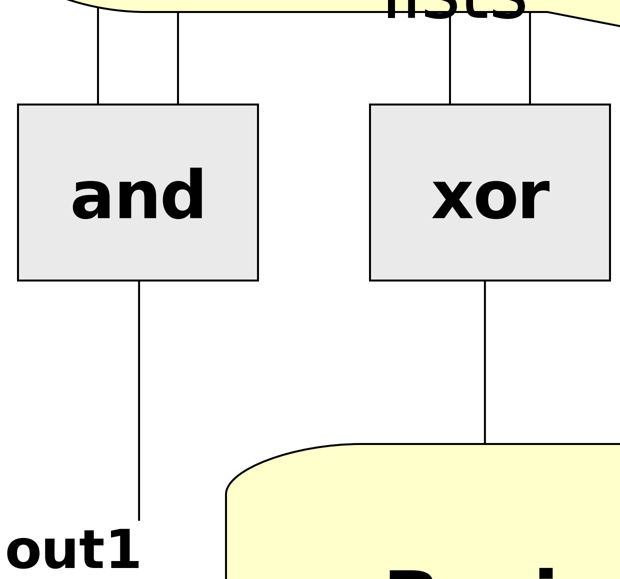


```
halfadder ::
   (Signal Bool, Signal Bool) ->
   (Signal Bool, Signal Bool)
halfadder (in1, in2) = (out1, out2)
   where
      out1 = and2 (in1, in2)
      out2 = xor2 (in1, in2)
```

Combinational Circuits in

Inputs
Do not use currying on the stream parameters, but combine in tuples or lists

Outputs



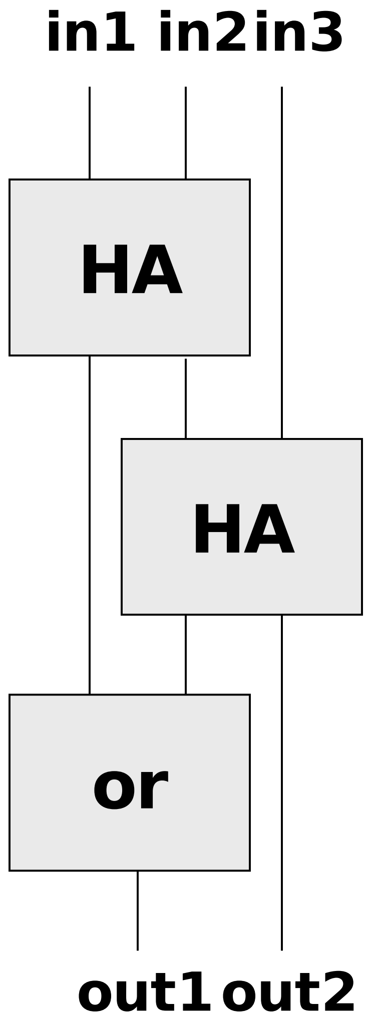
```
      der :
      eam 
   (Stream Bool, Stream B  l)
halfadder (in1, in2) = (out1, out2)
  where
    out1 = and2 (in1, in2)
       2 = xor2 (in1, in2)
```

Basic gates

Simulation

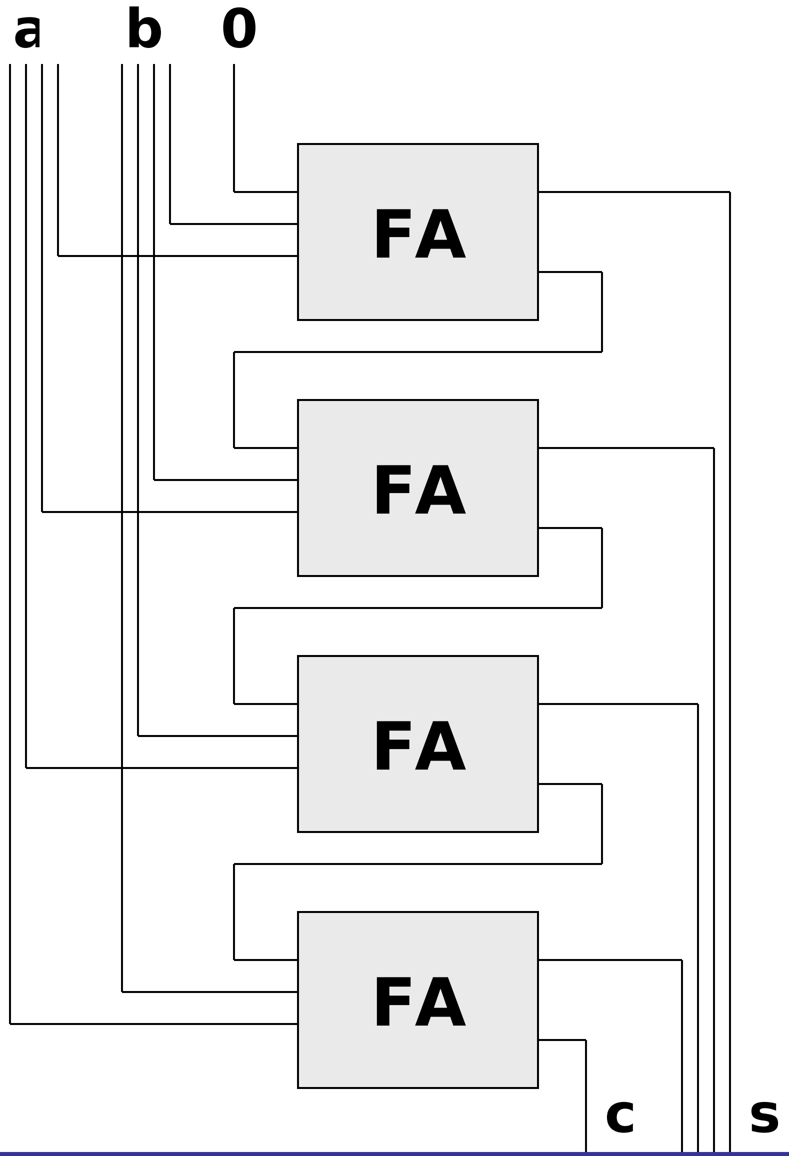
```
Main> simulate halfadder (high, low)
(low, high)

Main> simulate halfadder (high, high)
(high, low)
```

Reuse of Lava Components



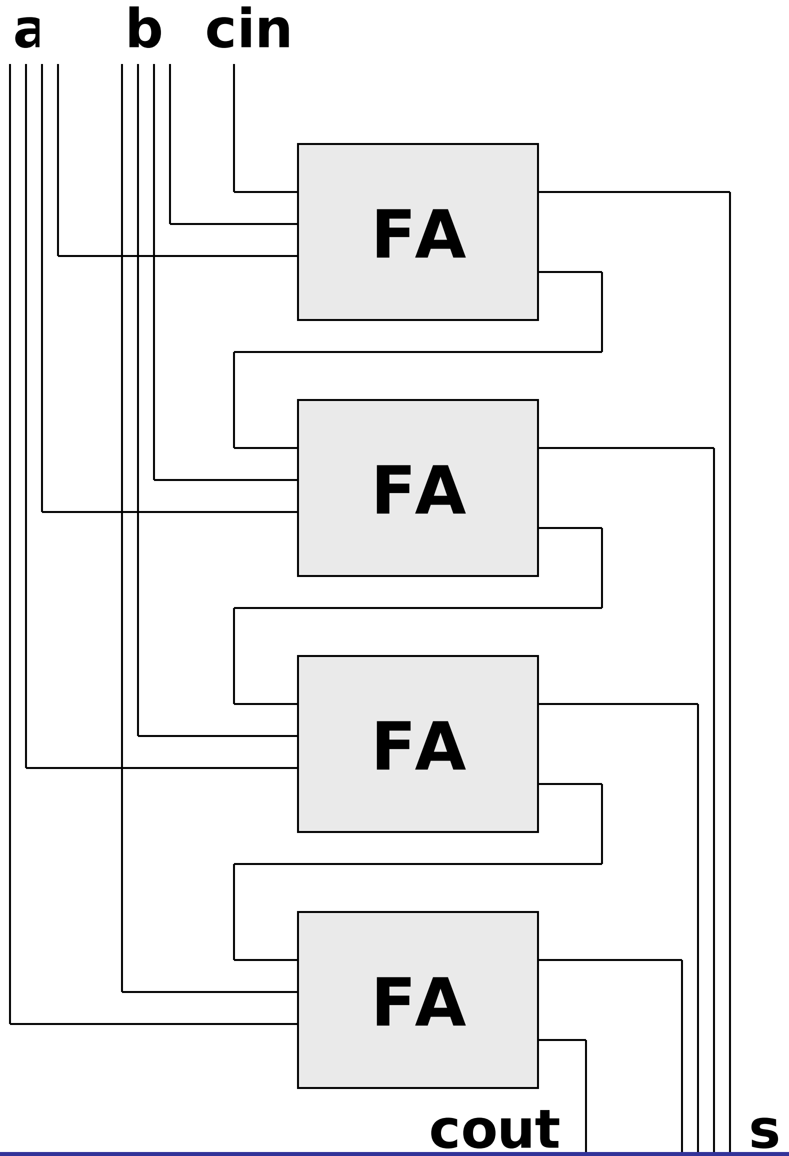
```
fulladder (in1, (in2, in3)) = (out1, out2)
  where
    (m1, m2) = halfadder (in1, in2)
    (m3, out2) = halfadder (m2, in3)
    out1 = or2 (m1, m3)
```

A Naïve 4-Bit Serial Adder



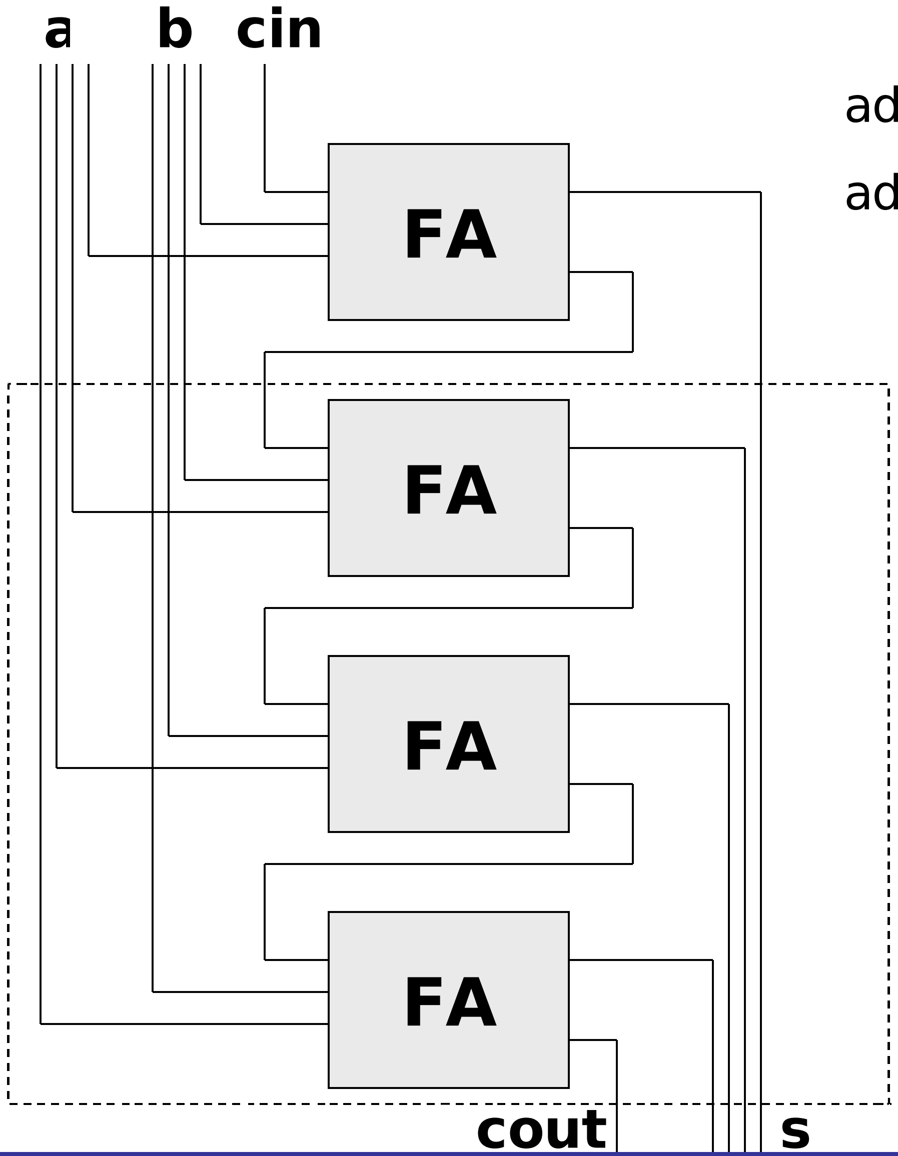
```
adder4 (a, b) = (c, s)
  where
    (a0, a1, a2, a3) = a
    (b0, b1, b2, b3) = b
    s = (s0, s1, s2, s3)

   (m0, s0) = fulladder (low, (a0, b0))
   (m1, s1) = fulladder (m0, (a1, b1))
   (m2, s2) = fulladder (m1, (a2, b2))
   (cout, s3) = fulladder (m2, (a3, b3))
```

A Naïve 4-Bit Serial Adder



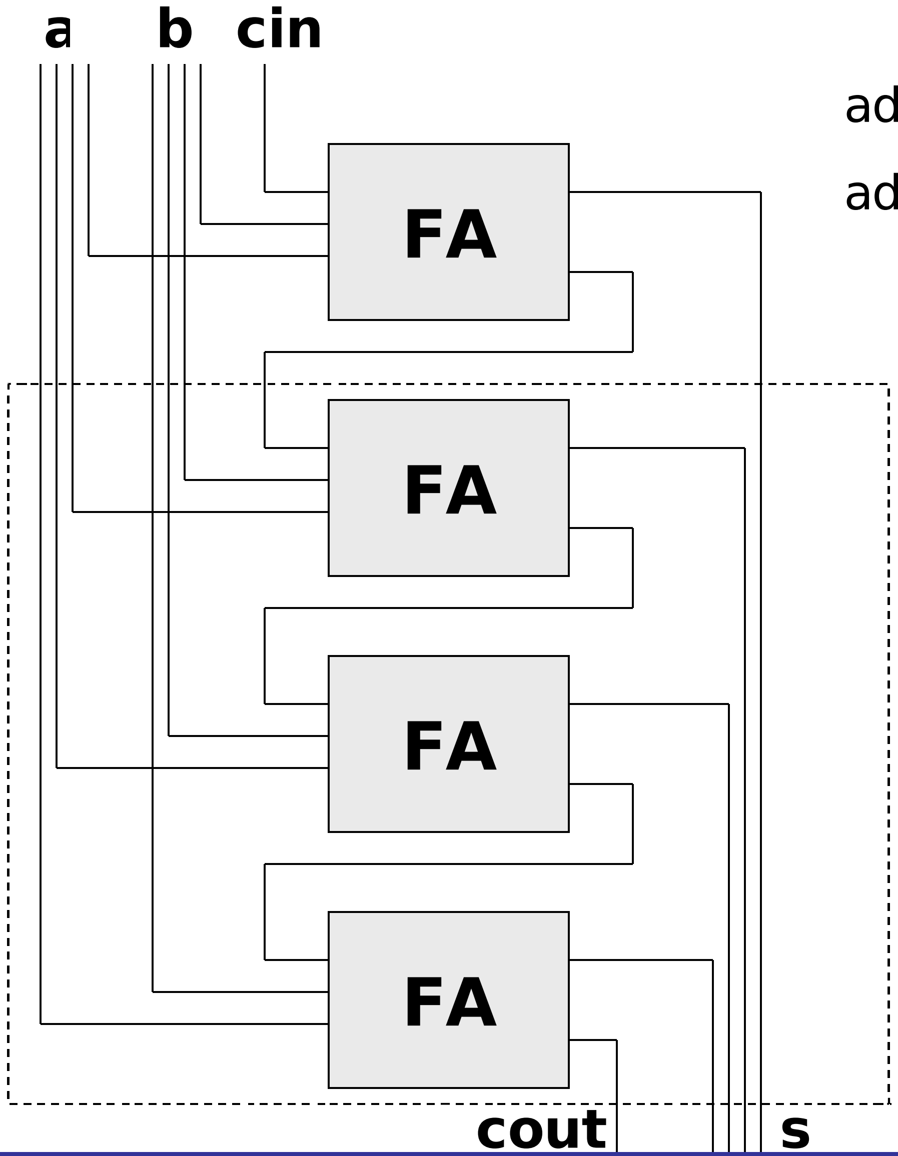
```
adder4 (cin, (a, b)) = (cout, s)
  where
    (a0, a1, a2, a3) = a
    (b0, b1, b2, b3) = b
    s = (s0, s1, s2, s3)

    (m0, s0) = fulladder (cin, (a0, b0))
    (m1, s1) = fulladder (m0, (a1, b1))
    (m2, s2) = fulladder (m1, (a2, b2))
    (cout, s3) = fulladder (m2, (a3,
b3))
```

A Better 4-Bit Serial Adder



```
adder (cin, ([], [])) = (cin, [])
adder (cin, (a:as, b:bs)) = (cout, s:ss)
  where
    (m, s) = fulladder (cin, (a, b))
    (cout, ss) = adder (m, (as, bs))
```

A Better 4-Bit Serial Adder



```
adder (cin, ([], [])) = (cin, [])
adder (cin, (a:as, b:bs)) = (cout, s:ss)
  where
    (m, s) = fulladder (cin, (a, b))
    (cout, ss) = adder (m, (as, bs))
```

Works for any width of input!

A Carry-Select Adder

- Strategy: split input wire lists in half – *as* into *as1* and *as2*, *bs* into *bs1* and *bs2.*
- (In parallel) calculate the following sums recursively:
  - *as1, bs1, cin* to get *ss1* and *cmid*
  - *as2, bs2, low* to get *ss2_0* and *cout_0*
  - *as2, bs2, high* to get *ss2_1* and *cout_1*
- Select *ss2* to be *ss2_0 or ss2_1* depending on *cmid.* Similarly select *cout.*
- The sum is now simply *ss1++ss2.*
- Combinational depth is O(*log n)* as opposed to O(*n)* (even if the number of gates has increased).

A Carry-Select Adder

```
adder2 (cin, ([], [])) = (cin, [])
adder2 (cin, ([a], [b])) = fulladder (cin, (a, b))
adder2 (cin, (as, bs)) = (cout, ss)
  where
    (as1, as2) = split as
    (bs1, bs2) = split bs

    (cmid, ss1) = adder2 (cin, (as1, bs1))
    (cout_0, ss2_0) = adder2 (low, (as2, bs2))
    (cout_1, ss2_1) = adder2 (high, (as2, bs2))

    cout = mux (cmid, (cout_0, cout_1))
    ss2 = mux (cmid, (ss2_0, ss2_1))

    ss = ss1++ss2
```

Simulation

```
Main> simulate adder (high, ([low, high], [high, low]))
(high, [low, low])

Main> simulate adder (low, ([high], [low]))
(low, [high])
```

Equality

```
equal (x,y) = inv (xor2 (x,y))
```

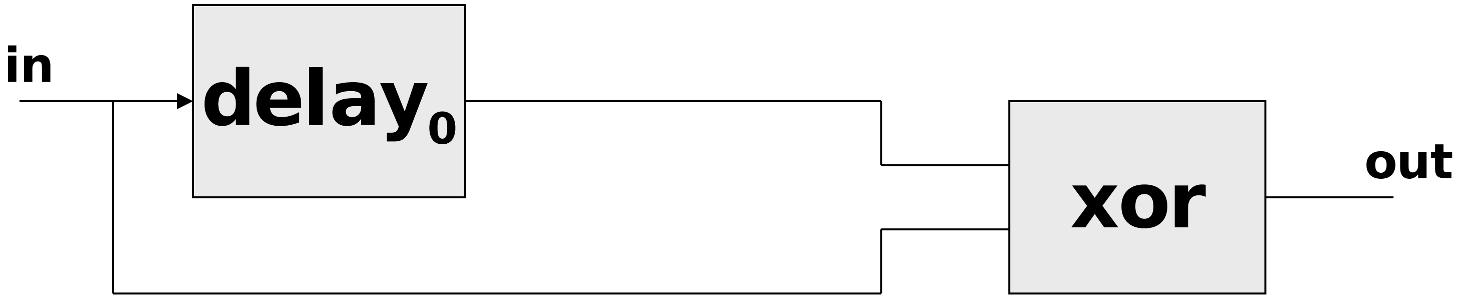
```
Main> simulate equal (low, high)
low
```

Equality of Lists of Wires

```
equals ([],[]) = high
equals (x:xs, y:ys) =
   equal (x,y) <&> equals xs ys
```

```
Main> simulate equals
                ([low,high], [high,low])
low
```

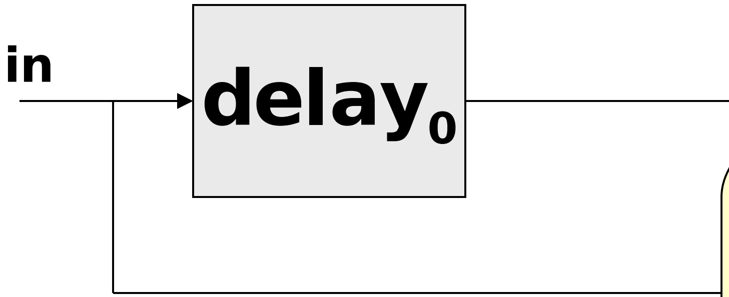
Sequential Circuits



The edge detector:

```
edge inp = xor2 (previous_inp, inp)
   where
        previous_inp = delay low inp
```

Sequential Circuits



The edge detecto

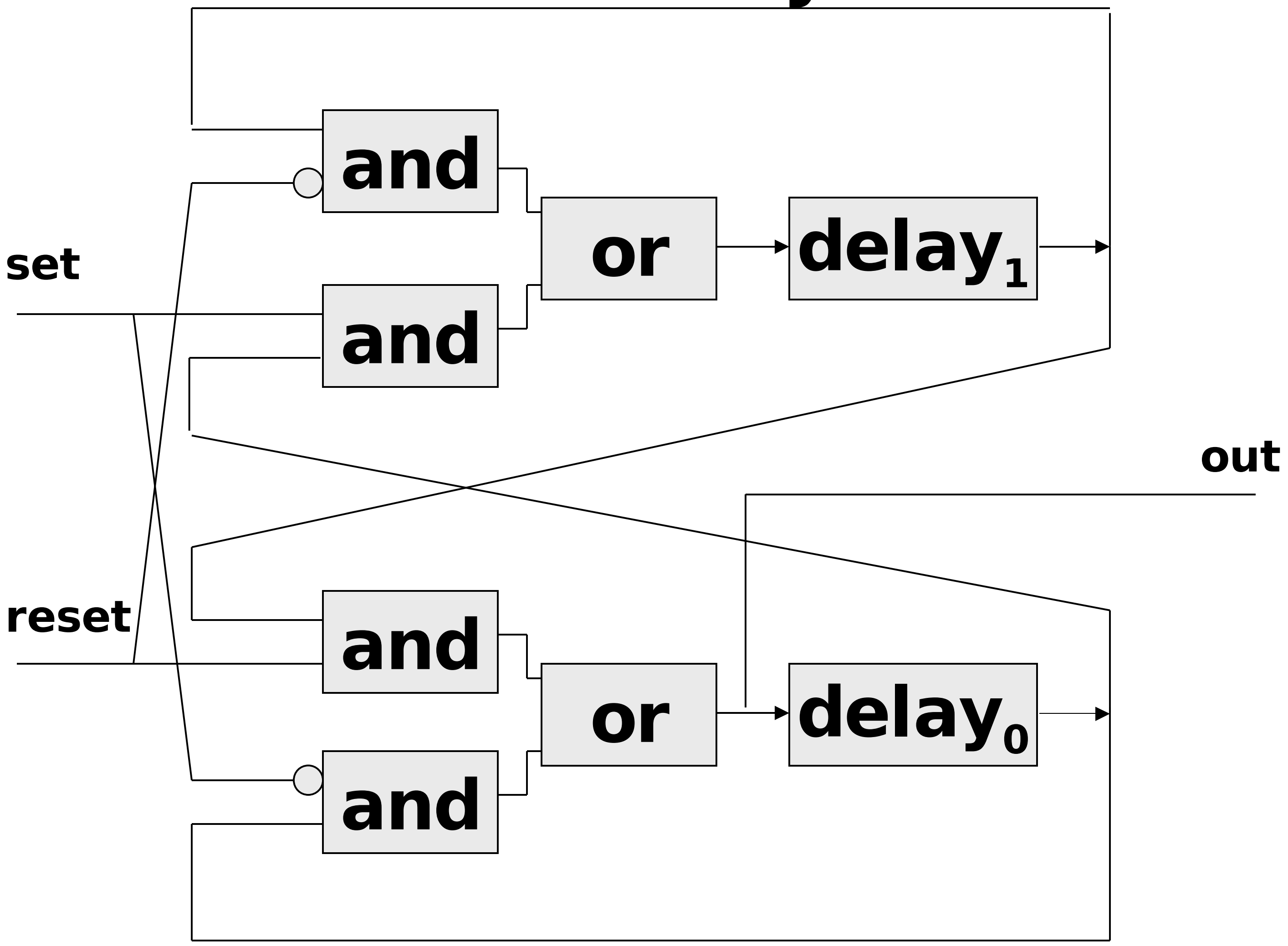
Note that **delay** is not a gate, but (**delay low**) is and so is (**delay high**)

edge in = xor2 (previ
   where
      previous_in = delay low in

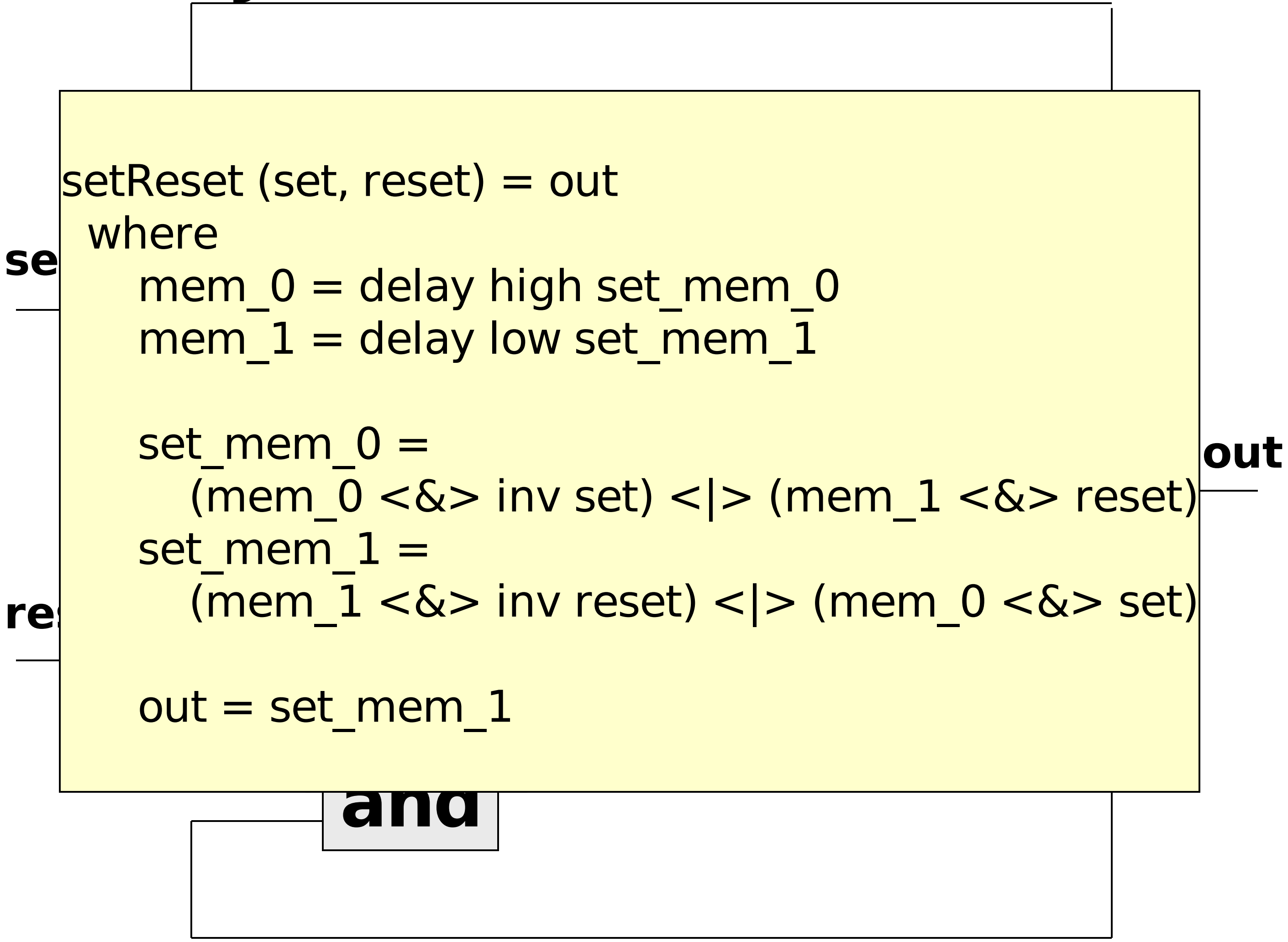
Sequential Circuit Simulation

```
Main> simulateSeq edge [low, high, high, low]
[low, high, low, high]

Main> simulateSeq edge [low, low, low, high]
[low, low, low, high]
```

Back to the Set-Reset Memory

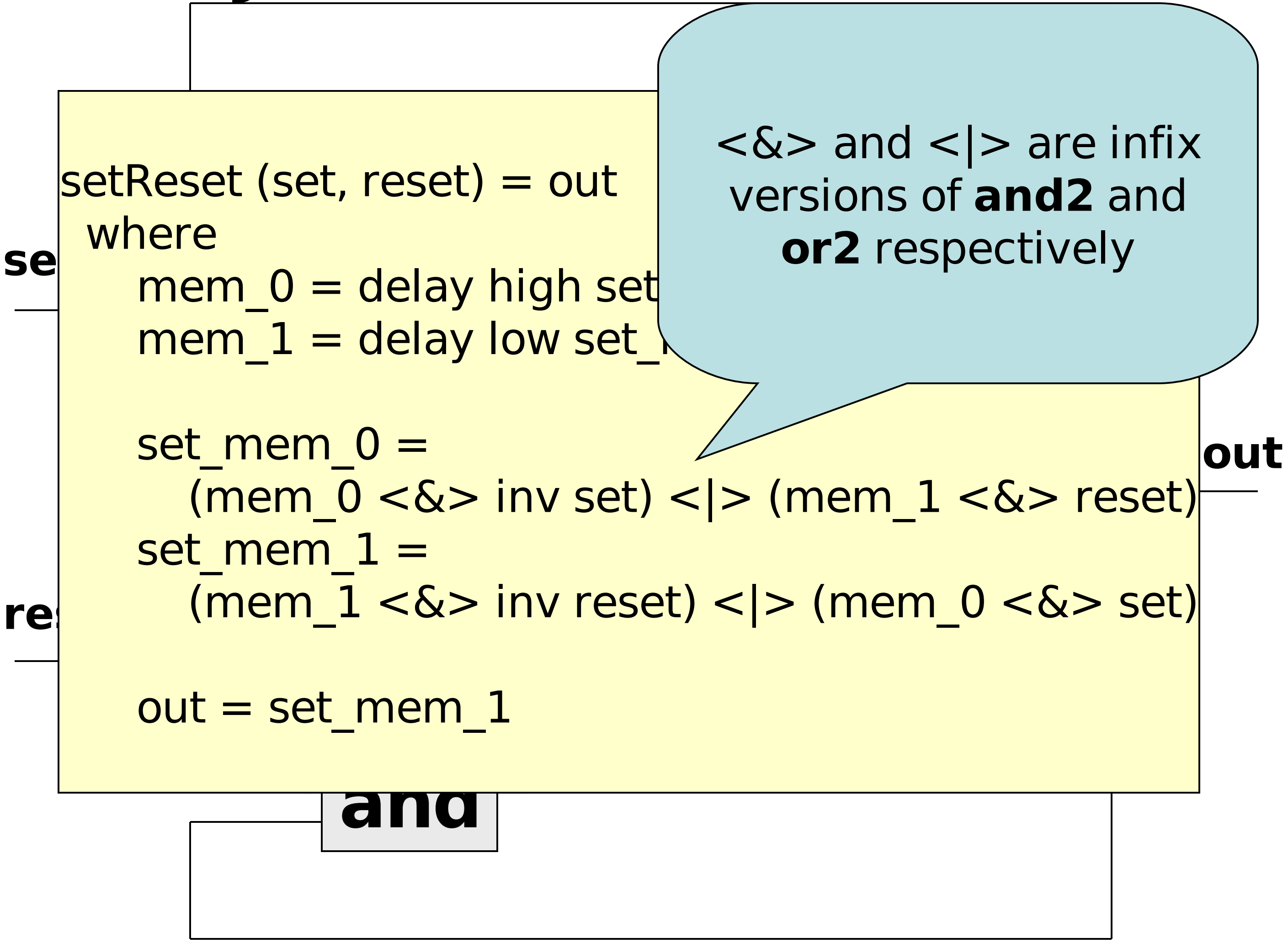


Back to the Set-Reset Memory



```
setReset (set, reset) = out
  where
      mem_0 = delay high set_mem_0
      mem_1 = delay low set_mem_1

      set_mem_0 =
          (mem_0 <&> inv set) <|> (mem_1 <&> reset)
      set_mem_1 =
          (mem_1 <&> inv reset) <|> (mem_0 <&> set)

      out = set_mem_1
```

Back to the Set-Reset Memory



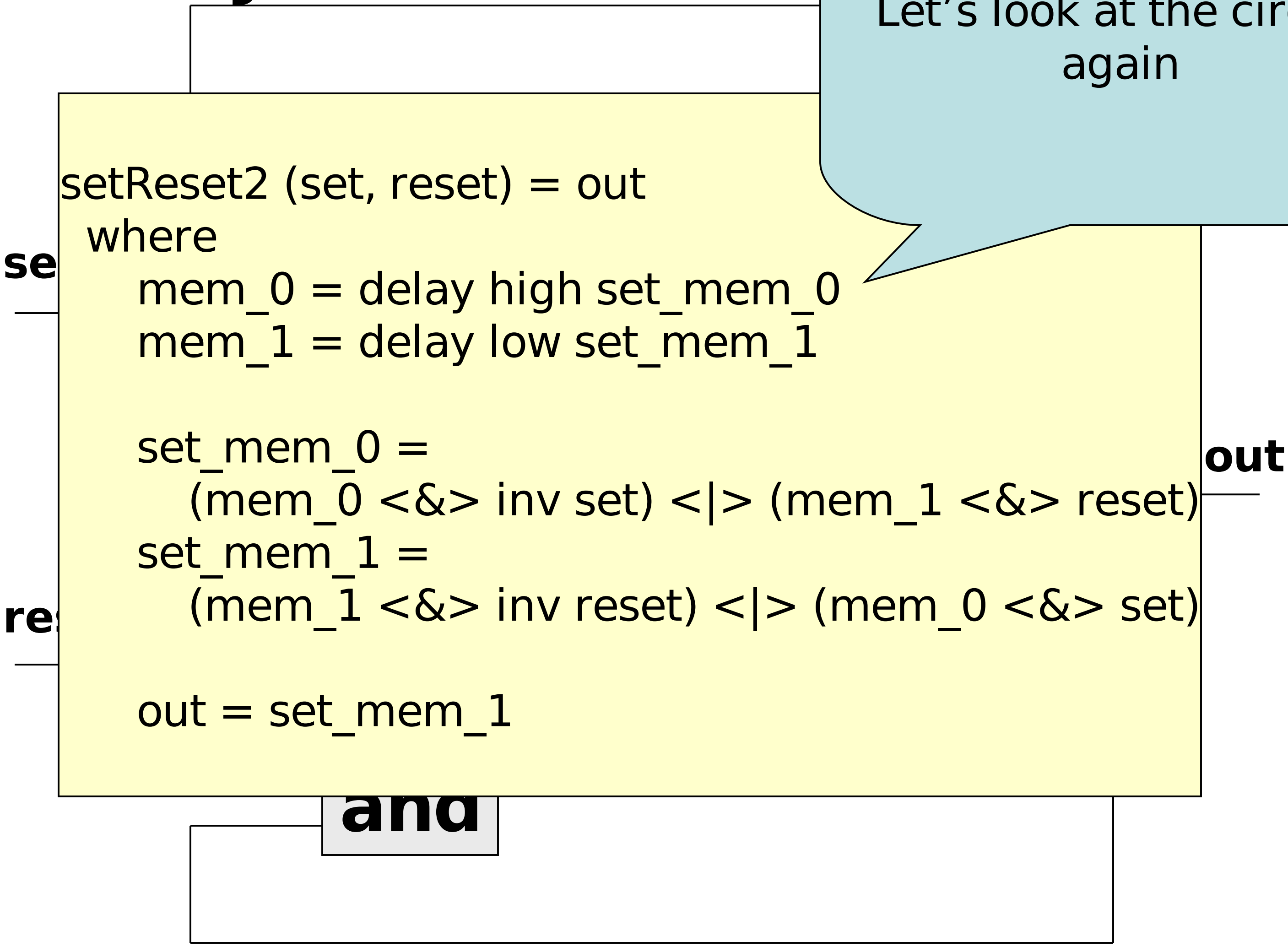
```
setReset (set, reset) = out
  where
     mem_0 = delay high set_
     mem_1 = delay low set_

     set_mem_0 =
        (mem_0 <&> inv set) <|> (mem_1 <&> reset)
     set_mem_1 =
        (mem_1 <&> inv reset) <|> (mem_0 <&> set)

     out = set_mem_1
```

<&> and <|> are infix versions of **and2** and **or2** respectively

Sequential Circuit Simulation

```
Main> simulateSeq setReset
                    [(low,high), (high, low), (low, low)]
[low, high, high]

Main> simulateSeq setReset
                    [(high, low), (high, low), (low, low)]
[high, high, high]
```

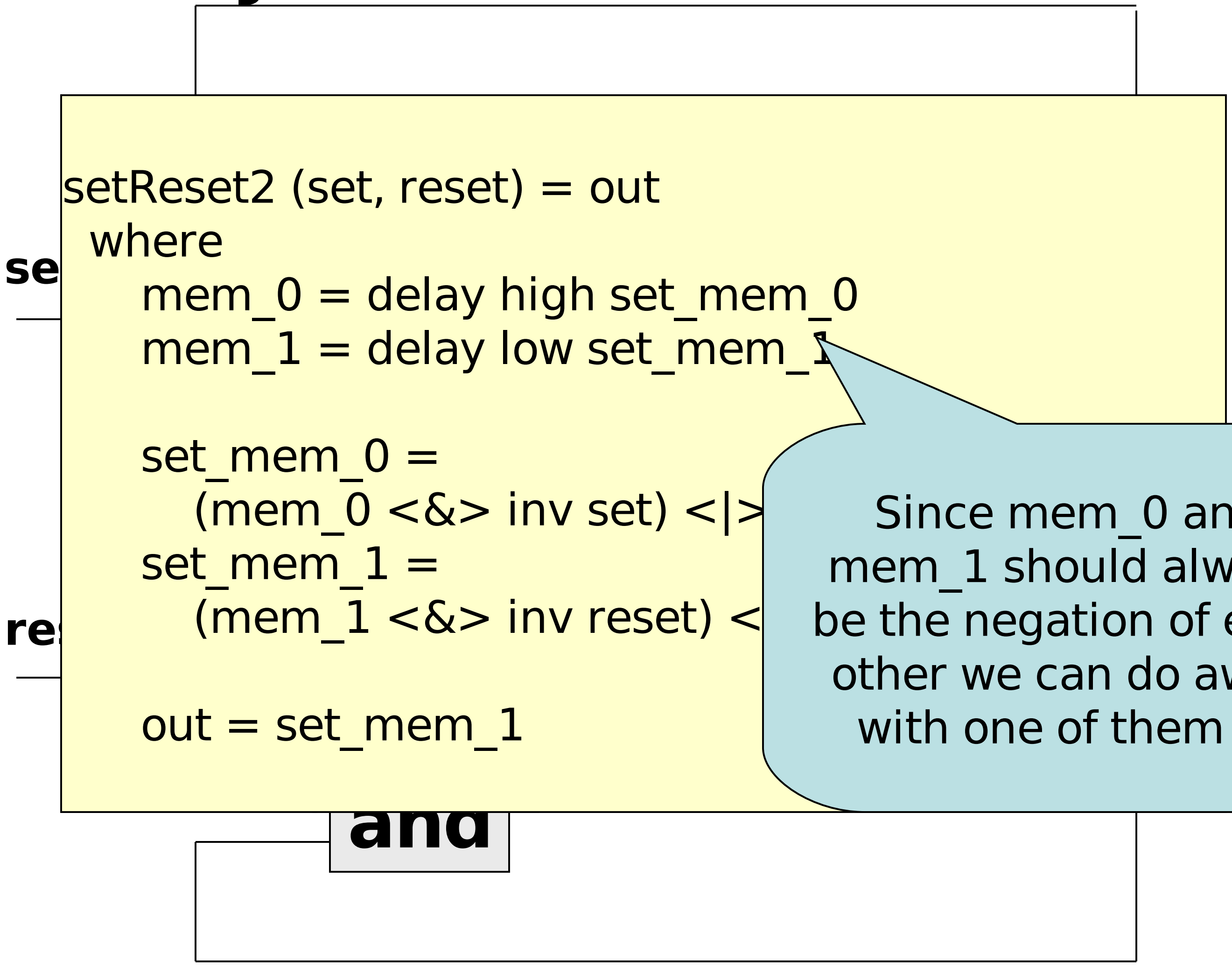
Back to the Set-Reset Memory



```
setReset2 (set, reset) = out
  where
     mem_0 = delay high set_mem_0
     mem_1 = delay low set_mem_1

     set_mem_0 =
        (mem_0 <&> inv set) <|> (mem_1 <&> reset)
     set_mem_1 =
        (mem_1 <&> inv reset) <|> (mem_0 <&> set)

     out = set_mem_1
```

Let’s look at the circuit again

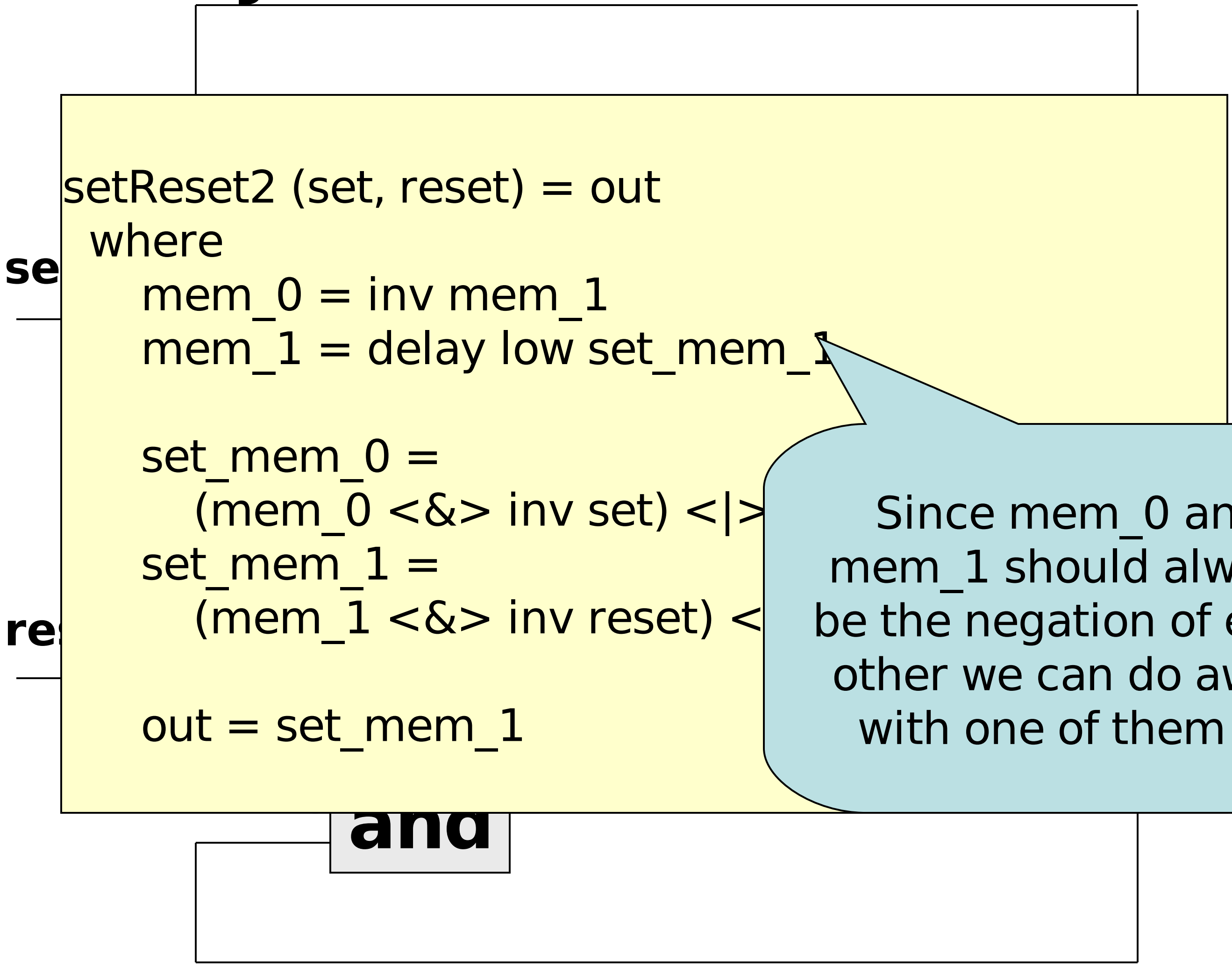
Back to the Set-Reset Memory



```
setReset2 (set, reset) = out
  where
      mem_0 = delay high set_mem_0
      mem_1 = delay low set_mem_1

      set_mem_0 =
          (mem_0 <&> inv set) <|>
      set_mem_1 =
          (mem_1 <&> inv reset) <

      out = set_mem_1
```

Since mem_0 and mem_1 should always be the negation of each other we can do away with one of them …

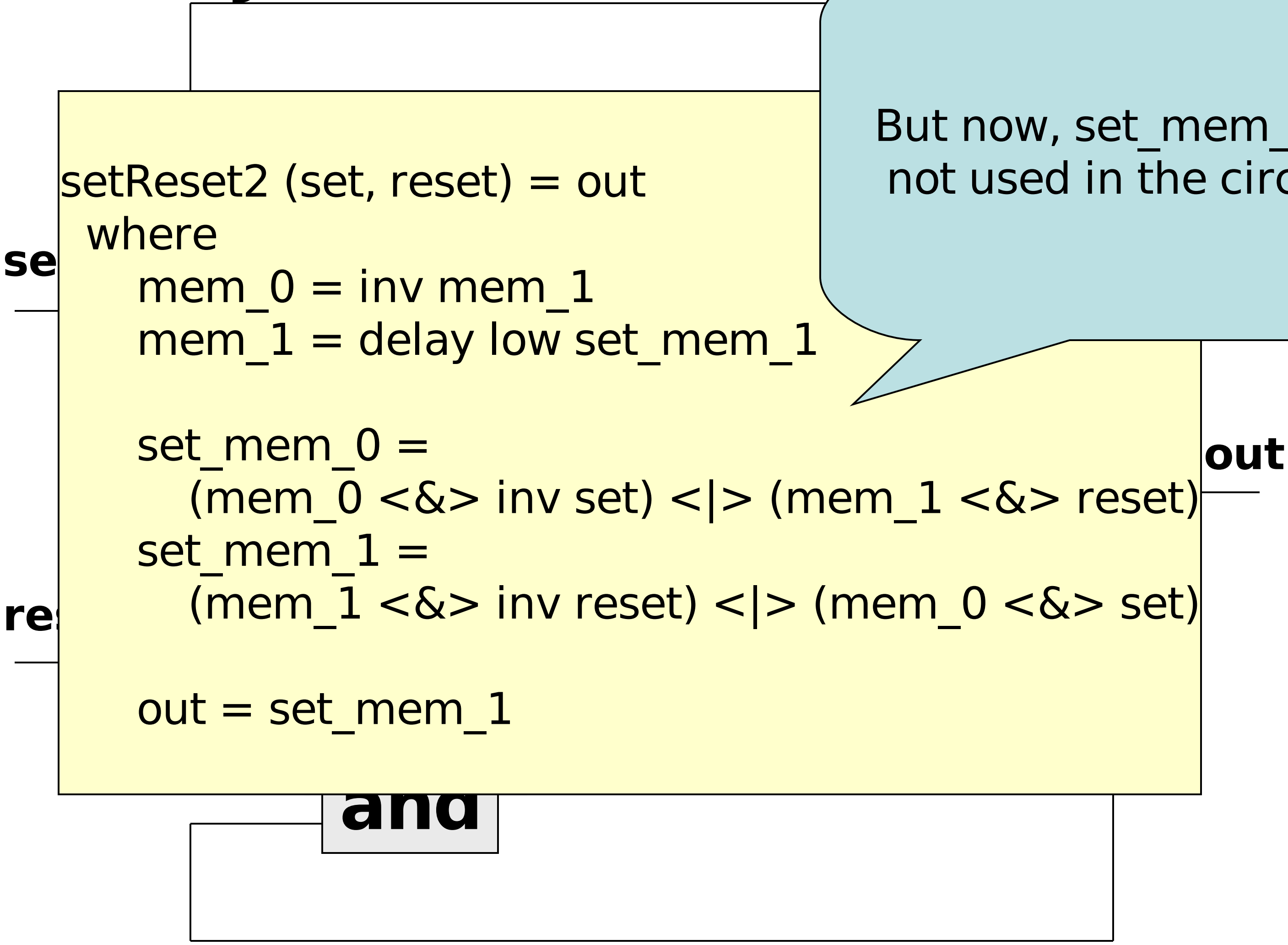
Back to the Set-Reset Memory



```
setReset2 (set, reset) = out
  where
      mem_0 = inv mem_1
      mem_1 = delay low set_mem_1

      set_mem_0 =
          (mem_0 <&> inv set) <|>
      set_mem_1 =
          (mem_1 <&> inv reset) <

      out = set_mem_1
```

Since mem_0 and mem_1 should always be the negation of each other we can do away with one of them …

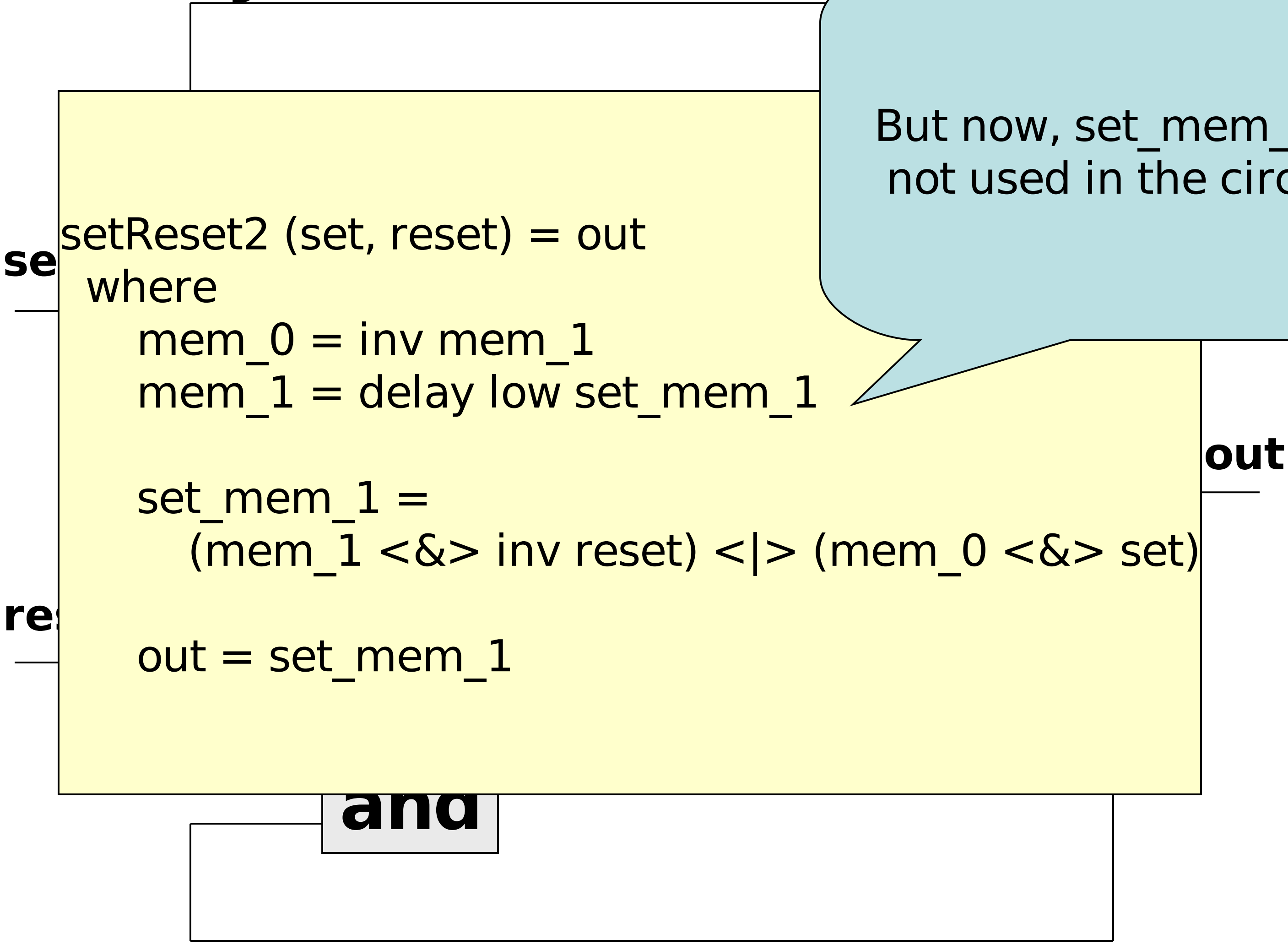
Back to the Set-Reset Memory



```
setReset2 (set, reset) = out
  where
    mem_0 = inv mem_1
    mem_1 = delay low set_mem_1

    set_mem_0 =
      (mem_0 <&> inv set) <|> (mem_1 <&> reset)
    set_mem_1 =
      (mem_1 <&> inv reset) <|> (mem_0 <&> set)

    out = set_mem_1
```

But now, set_mem_0 is not used in the circuit

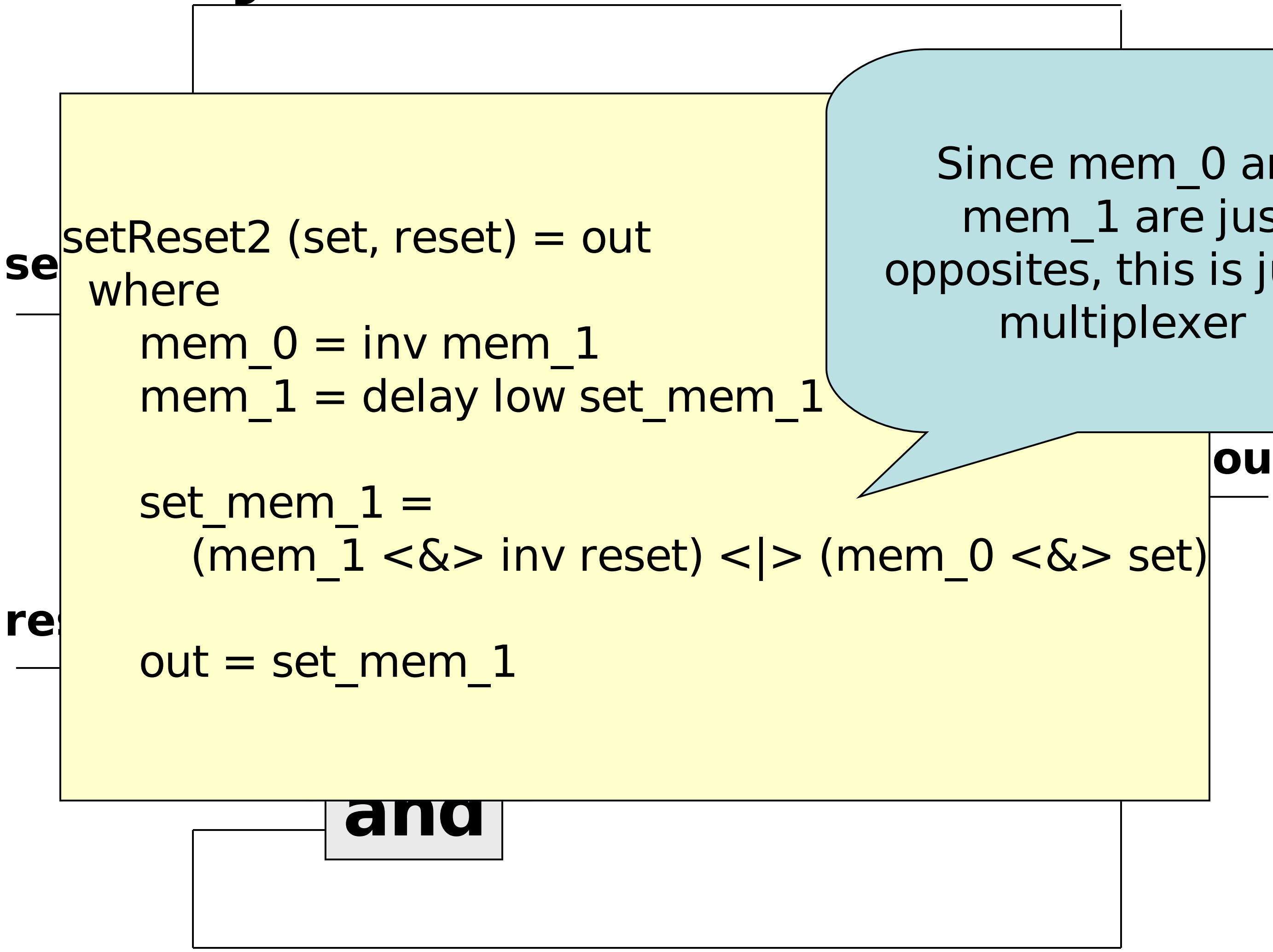
Back to the Set-Reset Memory



```
setReset2 (set, reset) = out
  where
    mem_0 = inv mem_1
    mem_1 = delay low set_mem_1

    set_mem_1 =
      (mem_1 <&> inv reset) <|> (mem_0 <&> set)

    out = set_mem_1
```

But now, set_mem_0 is not used in the circuit

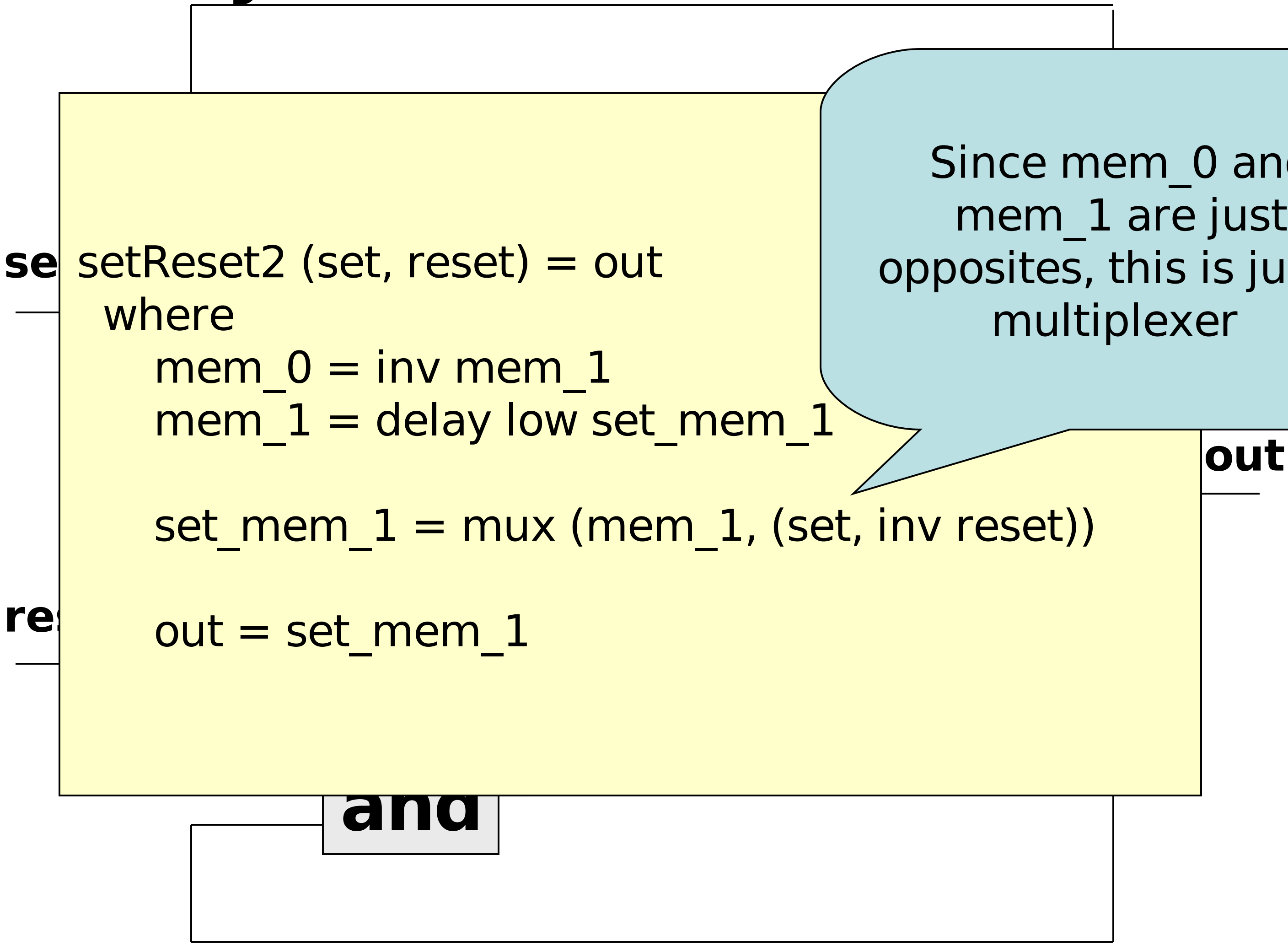
Back to the Set-Reset Memory



```
setReset2 (set, reset) = out
 where
    mem_0 = inv mem_1
    mem_1 = delay low set_mem_1

    set_mem_1 =
       (mem_1 <&> inv reset) <|> (mem_0 <&> set)

    out = set_mem_1
```

Since mem_0 and mem_1 are just opposites, this is just a multiplexer

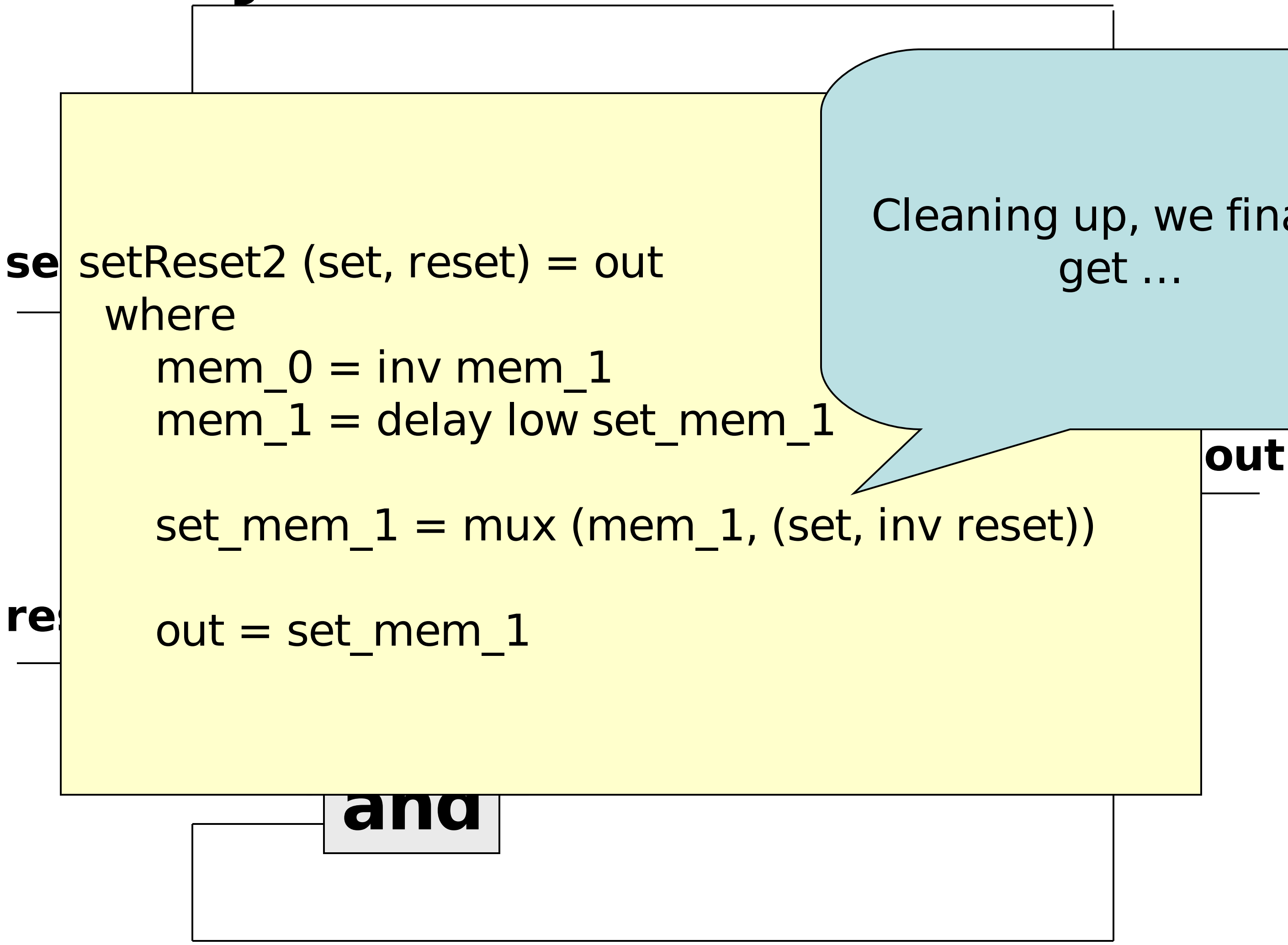
Back to the Set-Reset Memory



```
setReset2 (set, reset) = out
  where
    mem_0 = inv mem_1
    mem_1 = delay low set_mem_1

    set_mem_1 = mux (mem_1, (set, inv reset))

    out = set_mem_1
```

Since mem_0 and mem_1 are just opposites, this is just a multiplexer

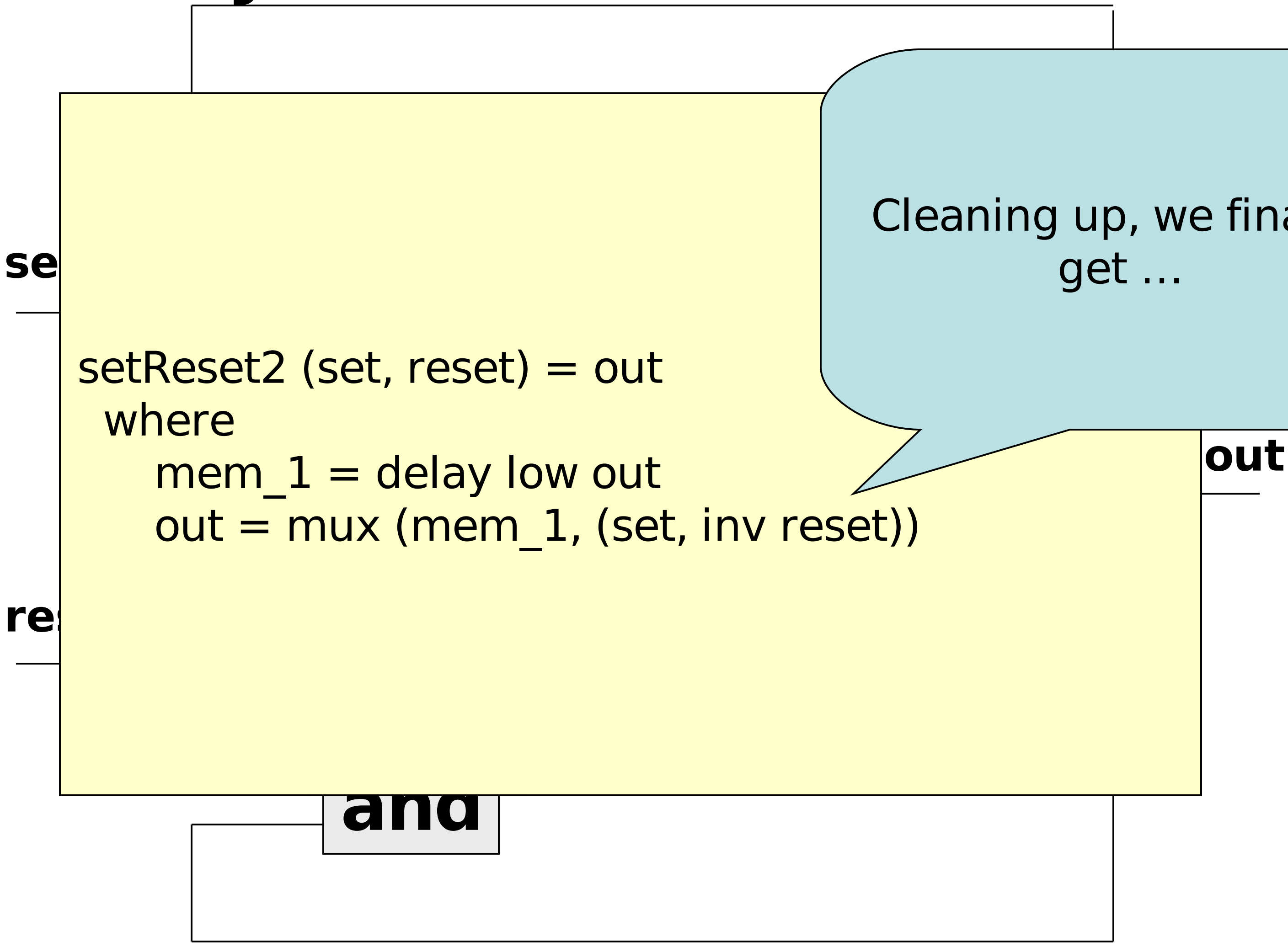
Back to the Set-Reset Memory



```
setReset2 (set, reset) = out
  where
     mem_0 = inv mem_1
     mem_1 = delay low set_mem_1

     set_mem_1 = mux (mem_1, (set, inv reset))

     out = set_mem_1
```

Cleaning up, we finally get …

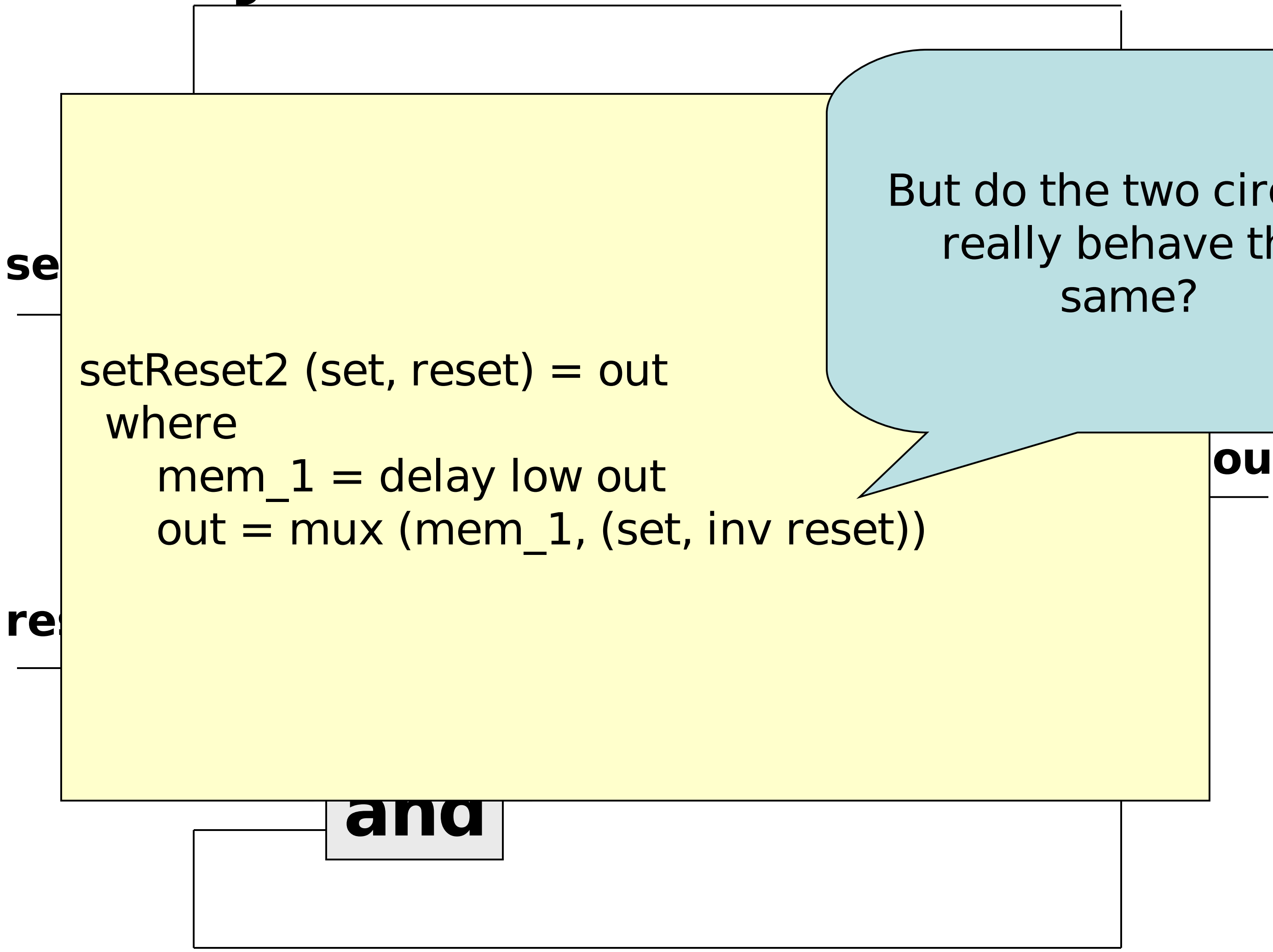
Back to the Set-Reset Memory



```
setReset2 (set, reset) = out
  where
    mem_1 = delay low out
    out = mux (mem_1, (set, inv reset))
```

Cleaning up, we finally get …

Back to the Set-Reset Memory



```
setReset2 (set, reset) = out
  where
      mem_1 = delay low out
      out = mux (mem_1, (set, inv reset))
```

But do the two circuits really behave the same?

Comparing the Set-Reset Memories

```
checkSetReset (set, reset) = ok
 where
    out1 = setReset (set, reset)
    out2 = setReset2 (set, reset)

    ok = equal (out1, out2)
```

Comparing the Set-Reset Memories

We can now use simulation to compare the different implementation with a given stream of inputs:

Main> simulateSeq checkSetReset

[(low,high), (high, low), (low, low), (low, high)]

[high, high, high, high]

Later on, we will see how to use model-checking to confirm its correctness.

Parametrised Circuits

- Since we can write whatever Haskell programs we want, we can write functions, which given an input return a circuit.
- We call such functions *parametrised circuits.*

Exactly *n* Bits are High

```
exactly :: Int -> [Signal Bool] -> Signal
  Bool
```

```
exactly 0 [] = high
exactly 0 (w:ws) = and2 (inv w, exactly
  0 ws)
```

Exactly *n* Bits are High

```
exactly :: Int -> [Signal Bool] -> Signal
   Bool


exactly 0 [] = high
exactly 0 (w:ws) = and2 (inv w, exactly
   0 ws)
exactly (n+1) [] = low
exactly (n+1) (w:ws) =
     mux (w, (exactly (n+1) ws, exactly n
   ws))
```

Exactly *n* Bits are High

exactly :: Int -> [Signal Bool] -> Signal Bool

exactly 0 [] = high

exactly 0 (w:ws) = and2 (inv w, exactly 0 ws)

exactly (n+1) [] = l

exactly (n+1) (w:ws

mux (w, (exactly

Note that **exactly** is a parametrised circuit. Given a natural number, it returns an actual circuit

Simulating Instances of **exactly**

```
Main> simulate (exactly 3) [high, low, high, high]
high


Main> simulateSeq (exactly 2)
      [[low,high,high], [high,low,low], [high,high,high]]
[high, low, low]
```

Some Common Mistakes

exactly 0 [] = high

exactly 0 (w:ws) = and2 (inv w, exactly 0 ws)

exactly (n+1) [] = low

exactly (n+1) (low:ws) = exactly (n+1) ws

exactly (n+1) (high ws

Pattern-matching on signals is not possible

Some Common Mistakes

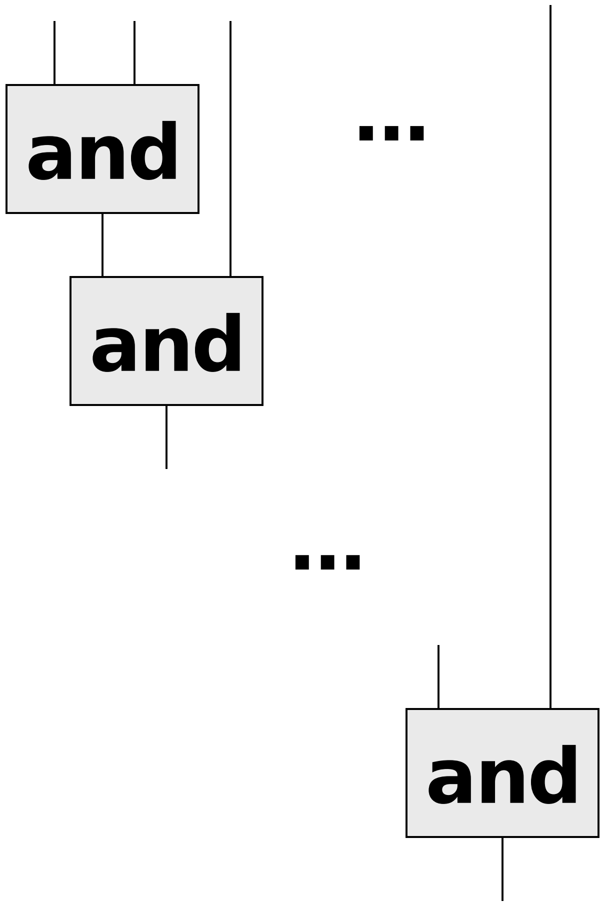
```
exactly 0 [] = high
exactly 0 (w:ws) = ar                y
  0 ws)
exactly (n+1) [] = low
exactly (n+1) (w:ws) =
  if w==low
    then exactly (n+1) ws
    else exactly n ws
```

Checking equality of signals is not possible, otherwise the result is not a circuit

Generic Circuits

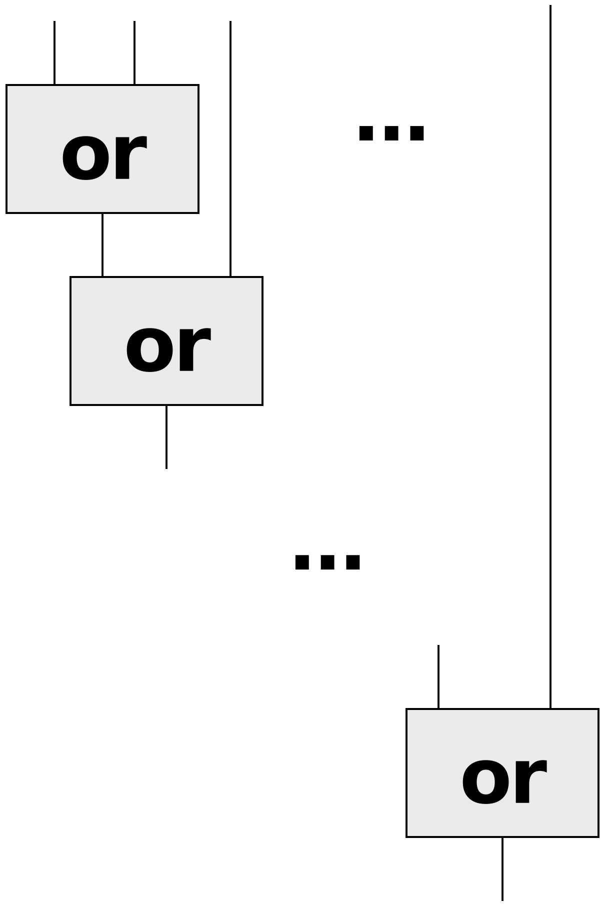
- We can go one step further, and parametrise circuits by other circuits.
- A generic circuit, or a connection pattern is a function, which given a circuit, returns another circuit.
- Using this approach to describe functions (circuits) without referring to the input-output wires is called *combinator-based programming*.

N-Input And Gate



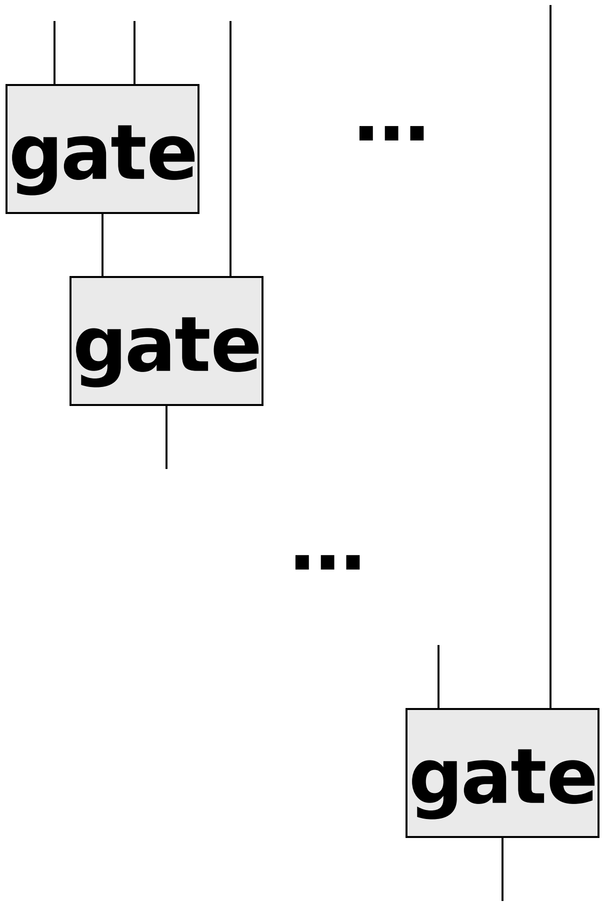
```
andl [w] = w
andl (w:ws) =
   and2 (w, andl ws)
```

N-Input Or Gate



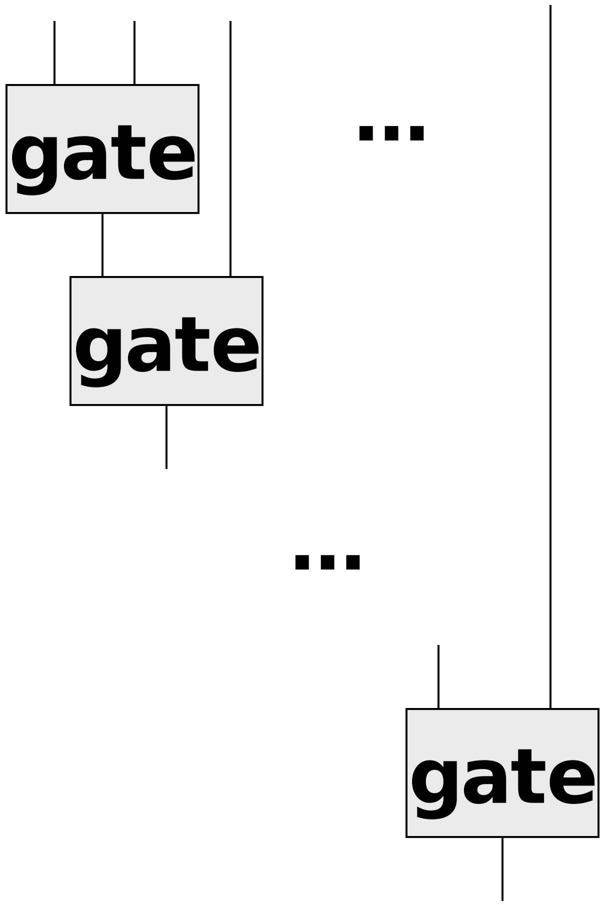
```
orl [w] = w
orl (w:ws) =
   or2 (w, orl ws)
```

Binary Gate Generalised to N-Inputs



```
gatel gate [w] = w
gatel gate (w:ws) =
   gate (w, gatel gate
   ws)
```

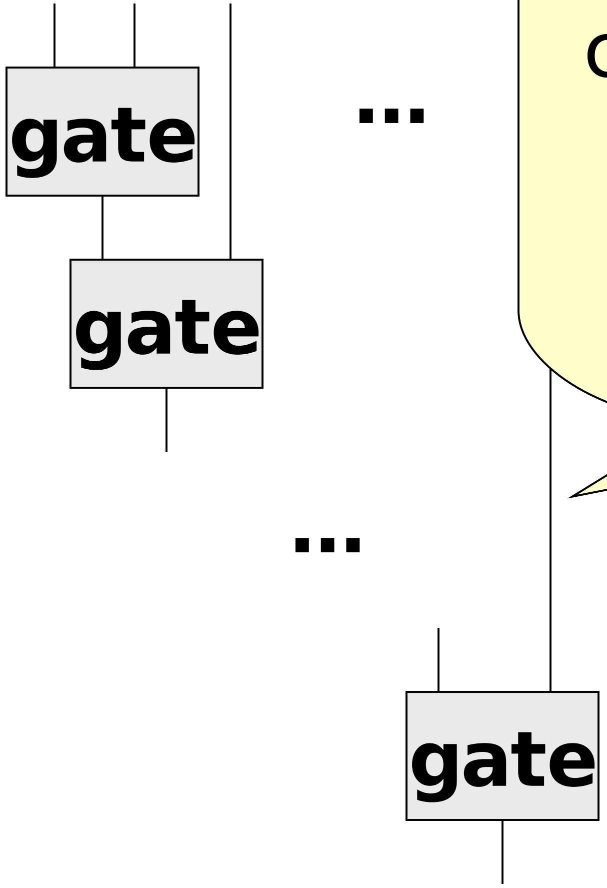
Binary Gate Generalised to N-Inputs



```
gatel gate [w] = w
gatel gate (w:ws) =
    gate (w, gatel gate
    ws)

orl = gatel or2
andl = gatel and2
```

Binary Gate Generalised to N-

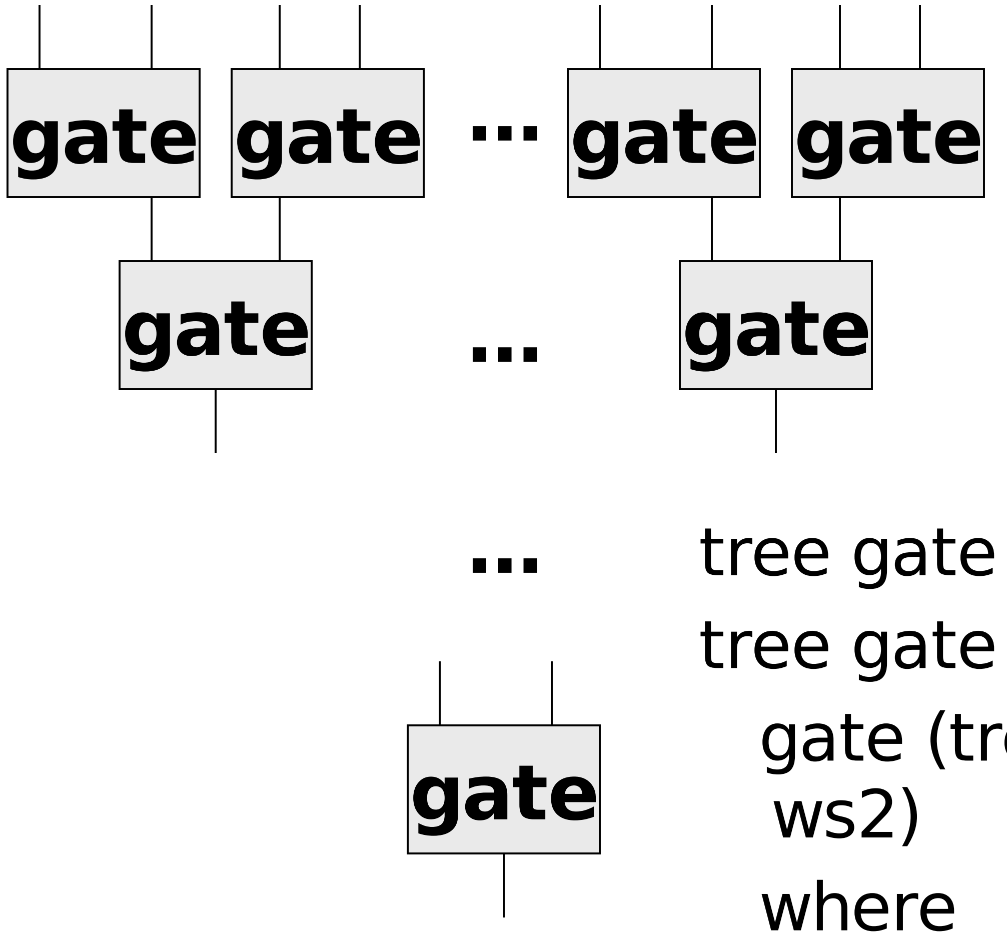
To avoid long combinational paths, we can reorganise to get the same result (provided the operators are associative).



orl = gatel or2

andl = gatel and2

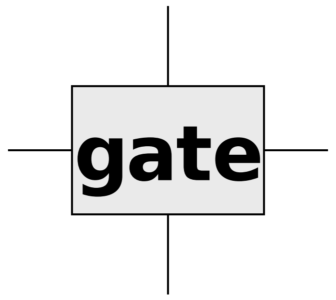
Binary Gate Generalised to N-Inputs



```
tree gate [w] = w
tree gate ws =
   gate (tree gate ws1, tree gate
    ws2)
   where
      (ws1, ws2) = split ws
```

Rows

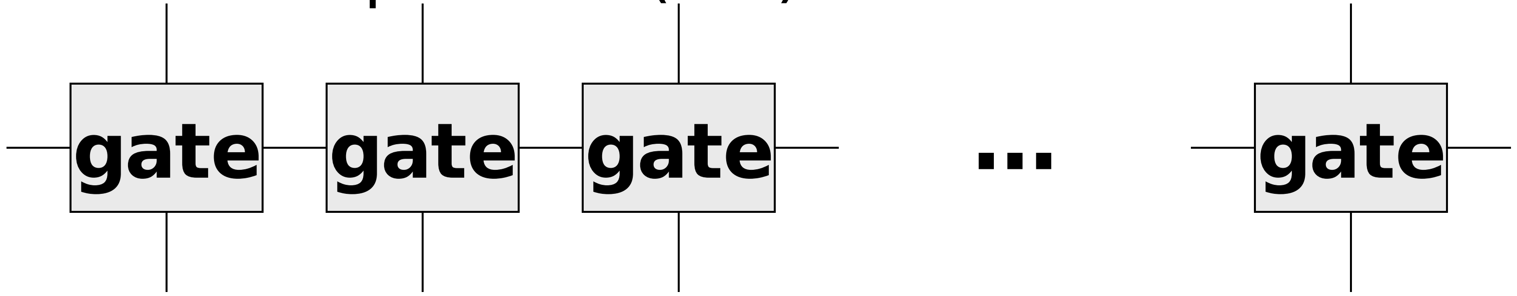
Given a gate with two inputs (left, top wires) and two outputs (bottom and right wires) ...



gate :: (Signal Bool, Signal Bool) -> (Signal Bool, Signal Bool)

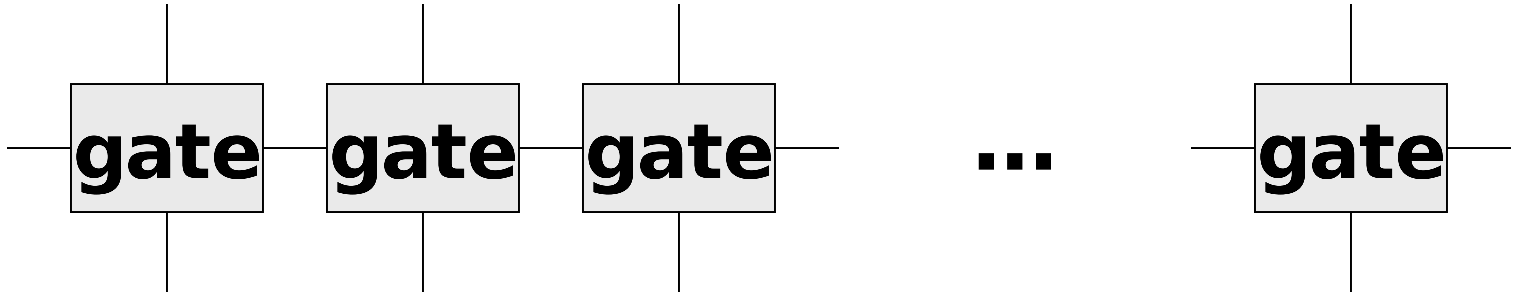
Rows

We can construct a circuit which given an input wire (left) and a list of input wires (top wires), produces a list of output wires (bottom wires) and an output wire (left):



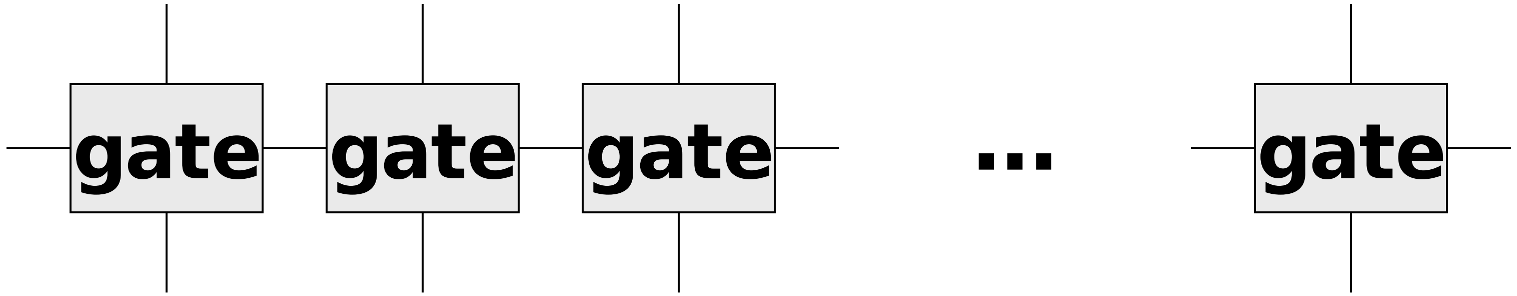
row gate :: (Signal Bool, [Signal Bool]) -> ([Signal Bool], Signal Bool)

Rows



```
row gate (left, []) = ([], left)
row gate (left, t:tops) = (b:bottoms,
  right)
  where
     (b, left’) = gate (left, t)
     (bottoms, right) = row gate (left’,
  tops)
```

Rows



row gate (left, []) = ([], left)

row gate (left, t:tops) = (b:bottoms, right)

where

(b, left')

(bottoms tops)

Actually, the row as implemented in Lava is more general than this, since it allows for arbitrary tuples of lists as the top inputs

Adders and ...

We can now define an adder simply as a row of full adders:

adder = row fulladder

What does a row of half adders do?

????? = row halfadder

The Carry-Select Pattern

```
select circ (cin, []) = (cin, [])
select circ (cin, xs) = (cout, ss)
  where
    (xs1, xs2) = split xs

    (cmid, ss1) = select circ (cin, xs1)
    (cout_0, ss2_0) = select circ (low, xs2)
    (cout_1, ss2_1) = select circ (high, xs2)

    cout = mux (cmid, (cout_0, cout_1))
    ss2 = mux (cmid, (ss2_0, ss2_1))

    ss = ss1++ss2
```

Parallel Adders and ...

We can now define an adder simply as a carry-select pattern of full adders:

```
adder2 = select fulladder
```

The Carry-Select Pattern

```
select circ (cin, []) = (cin, [])
select circ (cin, xs) = (cout, ss)
  where
    (xs1, xs2) = split xs

    (cmid, ss1) = select circ (cin, xs1)
    (cout_0, ss2_0) = select circ (low, xs2)
    (cout_1, ss2_1) = select circ (high, xs2)

    cout = mux (cmid, (cout_0, cout_1))
    ss2 = mux (cmid, (ss2_0, ss2_1))

    ss = ss1++ss2
```

Verification

- One of the unique features of Lava is its link to model checkers.
- Allows checking of circuit properties from within Lava, using external tools.
- Various model checking tools have been connected to Lava, but for this course we will only be using SMV.

Stating Properties

- Lava can be used to describe circuits.
- For verification we also need to state properties.

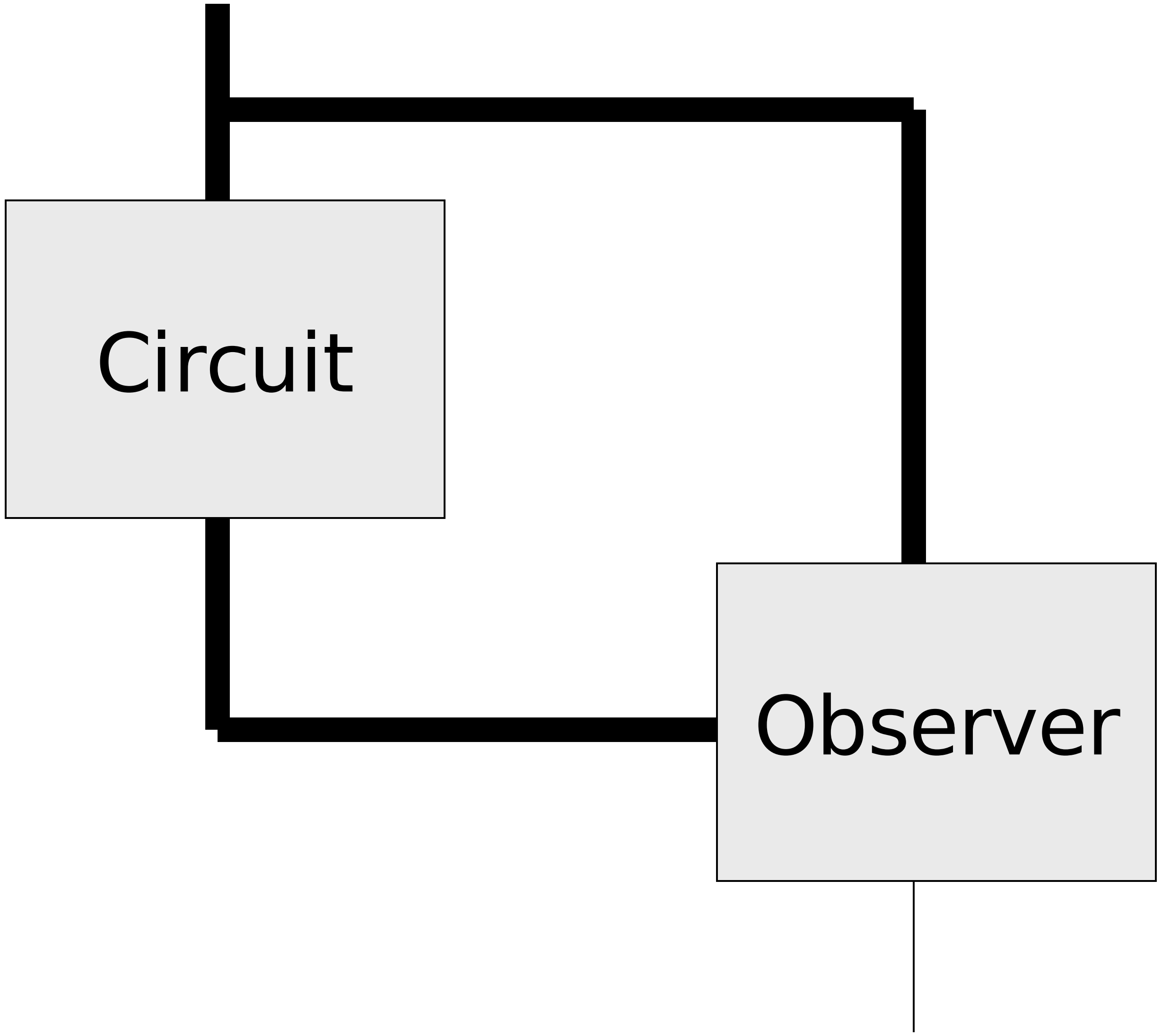
Two solutions:

- Give a separate language to express properties, *or*
- Describe properties as circuits, which output one bit, stating whether the circuit is working correctly.

Observers

- Lava takes the second option … we write *observers* which monitor the property in question.
- It is a well-known result that the expressiveness of observers is equivalent to safety properties – properties of the form *bad things never happen*.

Observers



Rising Edges

Rising edge implementation:

```
rise w = w <&> inv (delay low w)
```

Specification:

There can never be two rising edges in immediate succession.

```
property_rise w = inv (r <&> delay low
r)
    where
        r = rise w
```

Rising Edges

```
Main> smv property_rise
Proving: ... Valid
```

Falling Edges

Falling edge implementation:

```
fall w = inv w <&> delay low w
```

Specification:

Rising edges of a signal are equivalent to falling edges of the inverse of the signal.

```
property_edges w = equal (r, f)
      where
            r = rise w
            f = fall (inv w)
```

Falling Edges

Main> smv property_edges

Proving: ... Falsifiable

When *w* starts off with value high, *rise w* returns high, while *fall (inv w)* returns low. Otherwise, they match. Use Lava and SMV to check this!

Recall the Set-Reset Memories Comparison

```
checkSetReset (set, reset) = ok
 where
    out1 = setReset (set, reset)
    out2 = setReset2 (set, reset)

    ok = equal (out1, out2)
```

Result?

```
Main> smv checkSetReset
Proving: ... True
```

Similarly, we can compare adders ...

```
checkAdders (cin, (xs, ys)) = ok
 where
   (cout1, ss1) = adder   (cin, (xs, ys))
   (cout2, ss2) = adder2 (cin, (xs, ys))

   ok = equal (cout1, cout2) <&>
          equals (ss1, ss2)
```

Result?

But *smv checkAdders* gives an error because it doesn't know what width to use.

```
fourToList (x1,x2,x3,x4) = [x1,x2,x3,x4]

checkAdders (cin, (xs', ys')) = ok
  where
     xs = fourToList xs'
     ys = fourToList ys'

     (cout1, ss1) = adder   (cin, (xs, ys))
     (cout2, ss2) = adder2 (cin, (xs, ys))

     ok = equal (cout1, cout2) <&> equals (ss1, ss2)
```

Main> smv checkAdders

Proving: ... True

Other Features

Netlist Generation: Lava allows the user to generate a VHDL description of the circuit, to enable efficient simulation, testing, placement, etc.

```
Main> writeVhdl “rising” rise
```

Other Features

Signal types: Lava supports not only Signals of Booleans, but also signals of integers:

```
intSquarer n = times (n, n)
```

```
Main> simulateSeq intSquarer [4,5]
[16, 25]
```

Other Features

Signal types: Lava supports not only Signals of Booleans, but also signals of intege

intSqua

Main> S

[16, 25]

This sometimes leads to polymorphism on Signal types. For example:

mux :: (Signal Bool, (Signal a, Signal a)) -> Signal a

Conclusions

- Embedded languages give us more control over the generation and manipulation of the host language.
- Lava is excellent for describing regular circuits – the Haskell code runs to generate the actual circuit.
- In Part III we will look into the implementation of Lava and similar languages.

Exercises

- Implement the circuits you drew (or otherwise described) in Part 1 using Lava.
- Write a property to verify that *(all3 (a,b,c))* behaves just like *(all3 (c,a,b)).* Verify it.
- Write a property which states that a falling edge never appears on the output of *always w.* Verify it.
- Generalise the stack example to take a numeric parameter with the size of the stack.

  Express and verify the property that if you push an element on the stack, and then pop, the top element has not changed. What goes wrong? How can you fix this?

Functional Languages for Synchronous Hardware Design and Verification

Part 3: Writing Your Own Functional HDL

Gordon J. Pace

Department of Computer Science & AI

University of Malta

Objectives

- The elegance and expressivity of the functional approach to HDLs should be evident by now.
- Taking a look at the mechanisms beneath the hood will give us a better understanding.
- The aim of this part of the course is to build a simple functional HDL in Haskell.

The Language

- We will build a simple language, which will have just negation, conjunction and delay gates, and using only Boolean streams.
- Our principal aims are to be able to:
  - Describe circuits
  - Manipulate circuits
  - Simulate circuits
  - Produce a textual description of a circuit (this would be used in netlist generation and verification)

A Quick Hack

We can use a standard abstract datatype in Haskell to describe circuits:

```
data Circuit  =     Low
              |     High
              |     Not Circuit
              |     And Circuit Circuit
              |     Delay Bool Circuit
              deriving (Eq, Show)
```

A Quick Hack

With some helper functions, we can have a simple version of Lava running, no?

```
low = Low
high = High
inv w = Not w
and2 (u,v) = And u v

delay Low w = Delay False w
delay High w = Delay True w
delay _ _ =
    error "Delays can only take static
  initialisation"
```

More Utility Functions

```
or2 (u, v) = inv (and2 (inv u, inv v))

u <&> v = and2 (u,v)
u <|> v = or2 (u,v)

xor2 (u,v) = (u <&> inv v) <|> (inv u
   <&> v)
```

Evaluation of a Closed Circuit

A closed circuit is one with no inputs:

```
evaluate Low = Low
evaluate High = High

evaluate (Not c) =
  case evaluate c of
     Low -> High
     High -> Low

evaluate (And c1 c2) =
  case (evaluate c1, evaluate c2) of
      (High, High)         -> High
      _                    -> Low
```

Let’s Try to Simulate a Combinational Circuit

Recall that **simulate** takes a circuit which takes an input, and an input, and evaluates the output:

```
simulate fcircuit value =
            evaluate (fcircuit value)
```

Let’s try it out…

```
mux (sel, (l, h)) =
      (sel <&> h) <|> (inv sel <&> l)
```

```
Main> simulate mux (high, (high, low))
Low
```

What about counting gates?

```
count fcircuit = count’ (fcircuit Low)
  where
     count’ Low = 0
     count’ High = 0
     count’ (Not c) = 1+ count’ c
     count’ (And c1 c2) =
           1+ count’ c1 + count’ c2
     count’ (Delay _ c) = 1 + count’ c
```

Let’s try it out…

```
mux (sel, (l, h)) =
        (sel <&> h) <|> (inv sel <&> l)
```

```
Main> count mux
ERROR: Type error in application
```

Let’s try it out

What’s wrong?

```
mux (sel, (l, h))
      (sel <&> h
```

Main> count mux

ERROR: Type error in application

What about counting gates?

Main> count mux

ERROR: Type error in application

```
count fcircuit = count' (fcircuit Low)
  where
    ...
```

Count assumes that the circuit takes only one input. How can we fix it?

What about counting gates?

Main> count mux

ERROR: Type error in application

```
count fcircuit = count' (fcircuit Low)
  where
    ...
```

We need a way of generating arbitrary tuples (possibly of tuples) of Lows

Typeclasses to the Rescue

Types in class CircuitStructure can be populated by zeros…

```
class CircuitStructure a where
  zeros :: a
```

Typeclasses to the Rescue

Types in class CircuitStructure can be populated by zeros…

```
class CircuitStructure a where
   zeros :: a

count fcircuit = count’ (fcircuit zeros)
  where
     …
```

Solution Using Typeclasses

A circuit can be zero:

```
instance CircuitStructure Circuit where
    zeros = Low
```

Solution Using Typeclasses

Given two types which can be zeros, a pair of such types can also be zeros:

```
instance (CircuitStructure a,
   CircuitStructure b)
   => CircuitStructure (a, b) where
   zeros = (zeros, zeros)
```

Does it work now?

```
Main> count mux
7
```

Does it work now?

```
Main> count mux
7
```

7?! Yes, of course, because our **or2** gates are built from three **inv** and one **and2** gates

Another Example

```
example (u, v) = out
   where
      common = and2 (u, v)
      out = mux (u, (common, inv
   common))
```

How many gates should this have?

Another Example

```
example (u, v) = out
   where
       common = and2 (u, v)
       out = mux (u, (common, inv
   common))
```

How many gates should this have?

7 (mux) + 1 (inv) + 1 (and2) = 9

Another Example

```
example (u, v) = out
  where
    common = and2 (u, v)
    out = mux (u, (common, inv
  common))
```

Main> count example

10

Why?

```
example (u, v) = out
   where
      common = and2 (u, v)
      out = mux (u, (common, inv common))
```

is equivalent to

```
example (u, v) = out
   where
      out = mux (u, (and2 (u, v), inv (and2 (u,
   v))))
```

It is even worse than this ...

Consider the following circuit with feedback...

```
always s = out
   where
      out = s <&> delay low out
```

```
Main> count always
ERROR: Stack overflow
```

It is even worse than this ...

Consider the following circuit with feedback...

```
always s = out
   where
       out = s <&>
```

The function count traverses the delay over and over again until it runs out of stack space

```
Main> count always
ERROR: Stack overflow
```

Solutions: Name Circuits

1. **Naming circuits:** Circuits may be given names. Circuits with the same name are counted only once. It is up to the user to use different names.

```
example (u, v) = out
   where
     common = name “COMMON” (and2 
   (u, v))
     out = mux (u, (common, inv 
   common))
```

Solutions: Name Circuits

1. **Naming circuits:** Circuits may be given n he same name a It is up to the ames.

example (

where

comm and2 (u, v))

out = mux (u, (common, inv common))

Some functional HDLs, including Hydra used this solution (or a variant of it).

Solution: Carry a State Around

1. **Auxiliary state variable:** Since users may make mistakes, every circuit constructor is given a state and returns an updated state, which is used in the next constructor. This state (just a number) can be used to name circuits using unique names.

Carry a State Around

```
inv state w =
      (state+1, name (show state) Not w)
and2 state (u, v) =
      (state+1, name (show state) (And u v))

example state0 (u, v) = (state3, out)
   where
      (state1, common) = and2 state0 (u, v)
      (state2, inv_common) = inv state1 common
      (state3, out) = mux state2 (u, (common,
      inv_common))
```

Carry a State Around

```
inv state w =
      (state+1,
and2 state (u
      (state+1                                          ))

example stat
   where
      (state1,                                          )
      (state2,                                          mmon
      (state3, o                                      non,
      inv_common))
```

Here we hide the names, but still have the burden of carrying the state. Cumbersome and unwieldy.

Solution: Use a State Monad

1. **Use a state monad:** Since users may still make mistakes when passing the state around, the state may be made implicit using *monads* and the Haskell *do* notation which hides it all away.

```
example (u, v) =
   do
     common <- and2 (u, v)
     out <- mux (u, (common, inv common))
     return out
```

Solution: Use a State Monad

1. **Use a state monad:** Since users may still make ng the state ar nade implicit skell *do* notatio

example (u
  do
    comm
    out <- mmon))
    return out

If you have never heard about monads, this is not the time ...

Solution: Use a State Monad

1. **Use a state monad:** Since users may still make ng the state ar nade implicit skell *do* notatior

```
example (u
    do
      comm
      out <-                          ommon))
      return out
```

The first version of Lava used this trick to encode the state. Feedback loops complicate matters, though.

Use Non-Updatable References

1. **Use non-updatable reference:** If we are allowed to look at the memory address (pointer) of a data object, we can identify common sub-expressions. This assumes that the compiler/interpreter does not replicate expressions from let or where clauses.

```
example (u, v) = out
  where
     common = and2 (u, v)
     out = mux (u, (common, inv common))
```

Use Non-Updatable References

1. **Use non-updatable reference:** If we are allowed to look at the memory ad ct, we can id ns. This as rpreter dc from let or w

examp
 where
   common = and2 (u, v)
   out = mux (u, (common, inv common))

Count should realise that the two sub-expressions are the same, since they are located at the same memory location

Use Non-Updatable References

1. **Use non-updatable reference:** If we are allowed to look at the memory ad ct, we can id ns. This as preter d rom let or w

The result is not a 100% functionally pure solution, but is probably one of the better compromises around

examp
  where
    common = and2 (u, v)
    out = mux (u, (common, inv common))

Use Non-Updatable References

1. **Use non-updatable reference:** If we are allowed to look at the memory address (pointer) of a data object, we can identify common sub expressions. This as rpreter d rom let or w

Lava 2000, the version we are using, uses this solution

examp
  where
    common = and2 (u, v)
    out = mux (u, (common, inv common))

Implementing Named Circuits

We need to start by enriching the datatype:

```
data Circuit  =     Low
              |     High
              |     Not Circuit
              |     And Circuit Circuit
              |     Delay Bool Circuit
              |     Named String Circuit
              deriving (Eq, Show)
```

Implementing Named Circuits

We need to start by enriching the datatype and allowing the designers to access it:

```
data Circuit    =    Low
                |    ...
                |    Delay Bool Circuit
                |    Named String Circuit
                deriving (Eq, Show)
```

```
name string circuit = Named string circuit
```

Implementing Named Circuits

```
count fcircuit = snd (count' [] (fcircuit zeros))
  where
          count' nmes Low = (nmes, 0)
          count' nmes High = (nmes, 0)
          count' nmes (Not c) =
                    let (nmes', cnt) = count' nmes c
                    in  (nmes', 1+cnt)
          count' nmes (And c1 c2) =
                    let (nmes', cnt1) = count' nmes c1
                        (nmes'',cnt2) = count' nmes' c2
                    in  (nmes'', 1+cnt1+cnt2)
          count' nmes (Delay _ c) =
                    let (nmes', cnt) = count' nmes c
                    in  (nmes', 1+cnt)
          count' nmes (Named n c)
                    | n `elem` nmes = (nmes, 0)
                    | otherwise       = count' (n:nmes) c
```

Implementing Named Circuits

```
count fcircuit = snd (count’ [] (fcircuit
  where
        count’ nmes Low = (nmes,
        count’ nmes High = (nmes,
        count’ nmes (Not c) =
                let (nmes’, cnt) = c
                in  (nmes’, 1+cnt)
        count’ (And c1 c2) =
                let (nmes’, cnt1) =
                    (nmes’’,cnt2) =
                in  (nmes’’, 1+cnt1
        count’ (Delay _ c) =
                let (nmes’, cnt) = c
                in  (nmes’, 1+cnt)
        count’ nmes (Name n c)
                | n `elem` nmes = (nmes, 0)
                | otherwise      = count’ (n:nmes) c
```

As long as the user names all common sub-circuits and feedback loops, **count** will now work correctly

Simulation, Again

Simulation was performing extra work, recalculating common sub-expressions:

```
evaluate c = snd (evaluate [] c)
  where
    evaluate’ known Low = (known, Low)
    evaluate’ known High = (known, High)

    evaluate’ known (Not c) =
         let (known’, value) = evaluate’ known c
         in  case value of
                Low -> (known’, High)
                High -> (known’, Low)

    …
```

Some Limitations of Our System

- We can only have circuits which output one value. To be able to describe circuits like a halfadder, we need to perform a similar trick as we did on the input.
- To produce a netlist description, we need to be able to name inputs. We need to further enrich the circuit datatype, and the CircuitStructure class to be able to create a correct structure with distinct names.
- We did not touch on sequential simulation, in which we would need to evaluate a list of evaluated outputs, not just a single value.

Conclusions

- Building a practical functional HDL is not as straightforward as the final library may give the impression of.
- A major issue is that of shared circuits. We have presented some solutions to the problem.
- Overloading via typeclasses in Haskell works wonders!
- Lazyness can be useful when performing sequential simulation.

Exercises

- Implement an embedded hardware description language using named circuits as shown.
- Add a function to allow the user to run sequential simulation.
- Add a function to create a textual (VHDL/Verilog-like) description of a given closed circuit.

Functional Languages for Synchronous Hardware Design and Verification

Part 4: Embedded Hardware Compilers

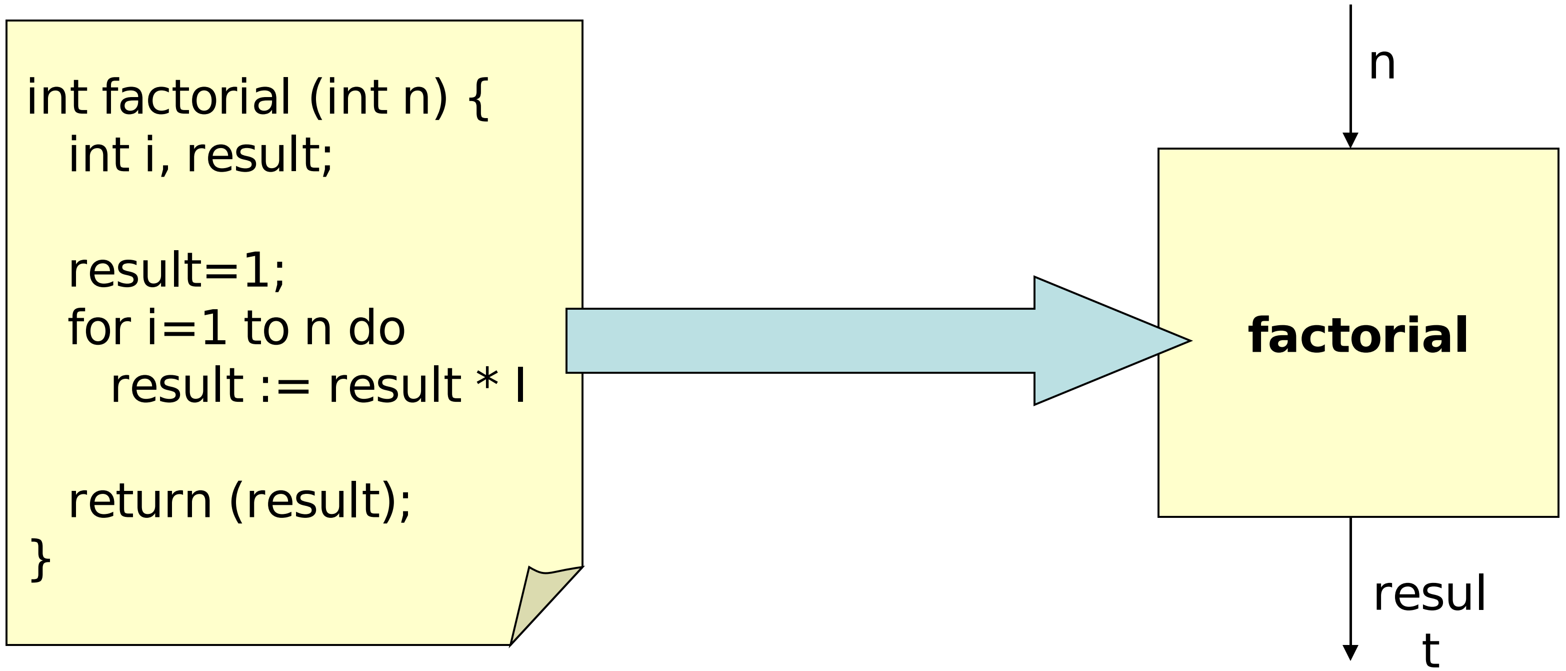
Gordon J. Pace

Department of Computer Science & AI

University of Malta

What is Hardware Compilation?

- Describe an algorithm using a high level formalism
- Compile directly into hardware



```
int factorial (int n) {
   int i, result;

   result=1;
   for i=1 to n do
      result := result * I

   return (result);
}
```

Issues Regarding Compilation

- Should the program run once every clock tick?
  - Problems with unbounded loops and non-termination
  - No memory between clock ticks
  - The same program runs every clock tick
- Should we allow execution to take more than one clock tick?
  - When is the result available?
  - What is the meaning of intermediate results?
  - Should the programmer be able to express clock barriers?

Issues Regarding Compilation

- Should the program run once every clock tick?
  - Pro n- ter
  - No
  - The k
- Shou nore than
  - When is the result available?
  - What is the meaning of intermediate results?
  - Should the programmer be able to express clock barriers?

During this course we will be dealing exclusively with compiled languages which:

3. Potentially run over multiple clock cycles
4. Timing is explicit in the programs themselves

Compiling Regular Expressions

RE	::=	a	Single symbol
	|	RE^+	Repetition
	|	RE + RE	Choice
	|	RE . RE	Catenation

For simplicity of exposition, we avoid regular expressions which accept the empty string.

Regular Expressions and Hardware

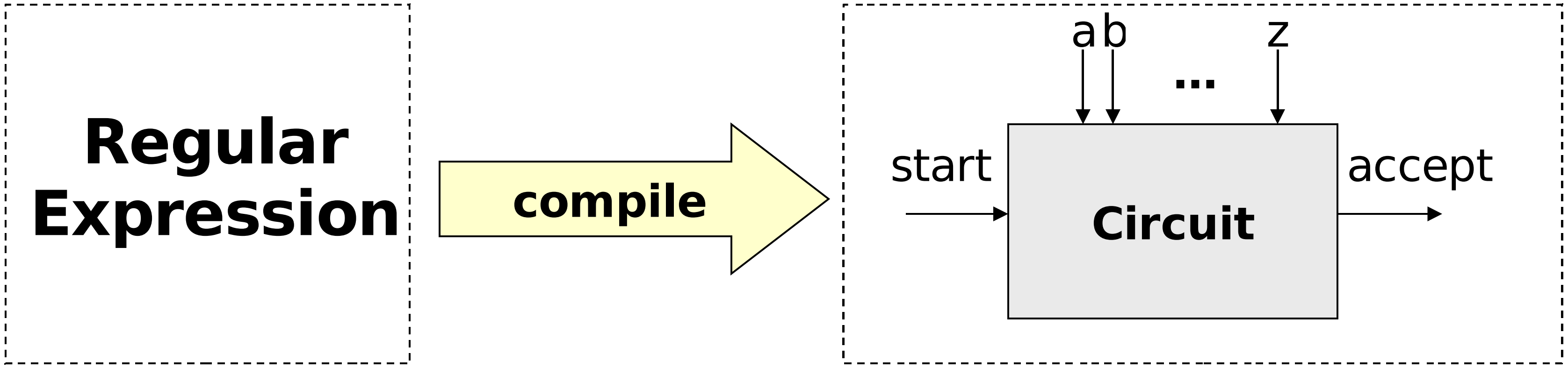
- The alphabet is the set of wires.
- A single symbol *a* is accepted during the next clock tick if wire *a* currently carries high.
- Choice, repetition and catenation are interpreted as usual.

Regular Expressions and Hardware

$$a \,.\, (b^{+} + a)$$

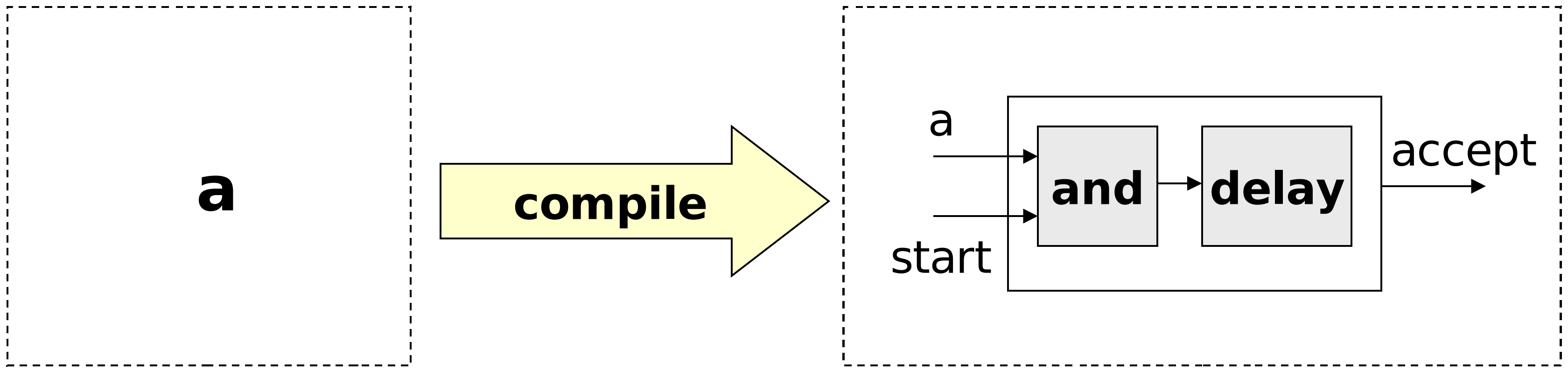
Is accepted on the third clock tick, if *a* was high during the first two clock ticks, or after three or more clock ticks, if *a* was high in the first, and *b* was high throughout the rest of the ticks (except possibly the last).

Compiling Regular Expressions to Hardware



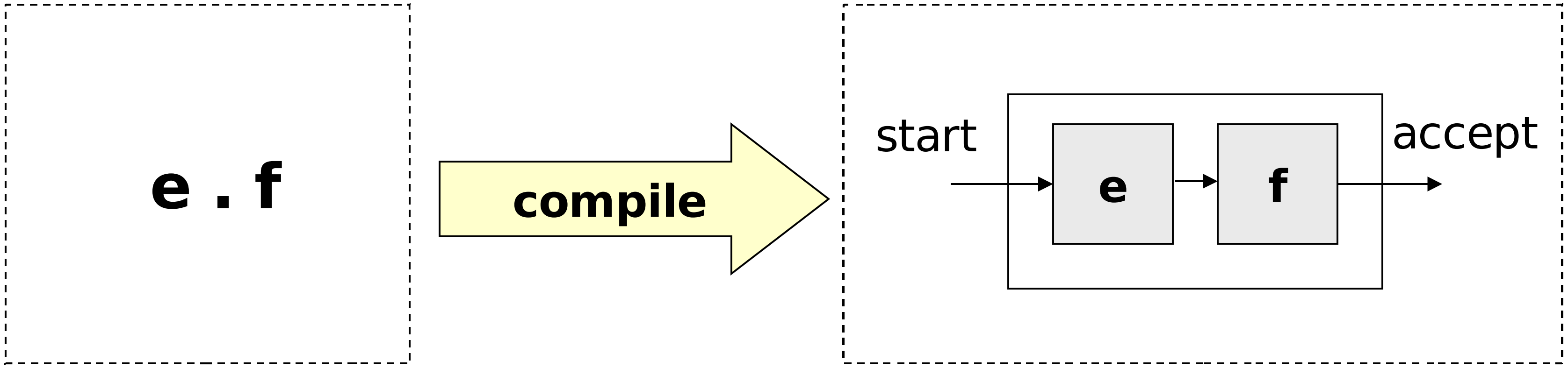
- The start input tells the circuit when to start parsing.
- A circuit can be started multiple times

Compiling Regular Expressions to Hardware



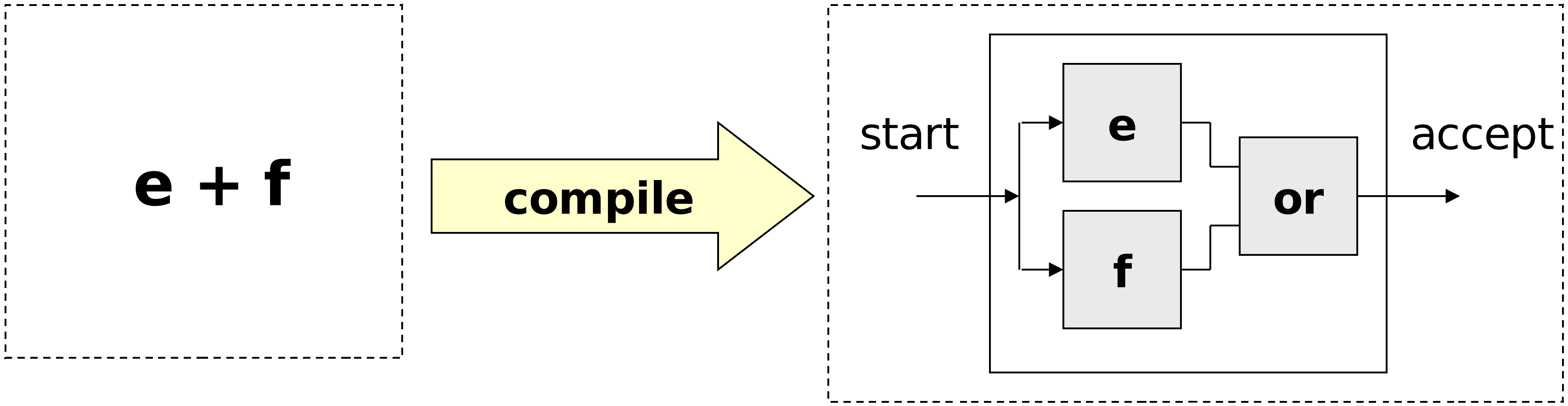
The single symbol *a* is accepted in the next clock tick if we start parsing now, and *a* is currently high.

Compiling Regular Expressions to Hardware



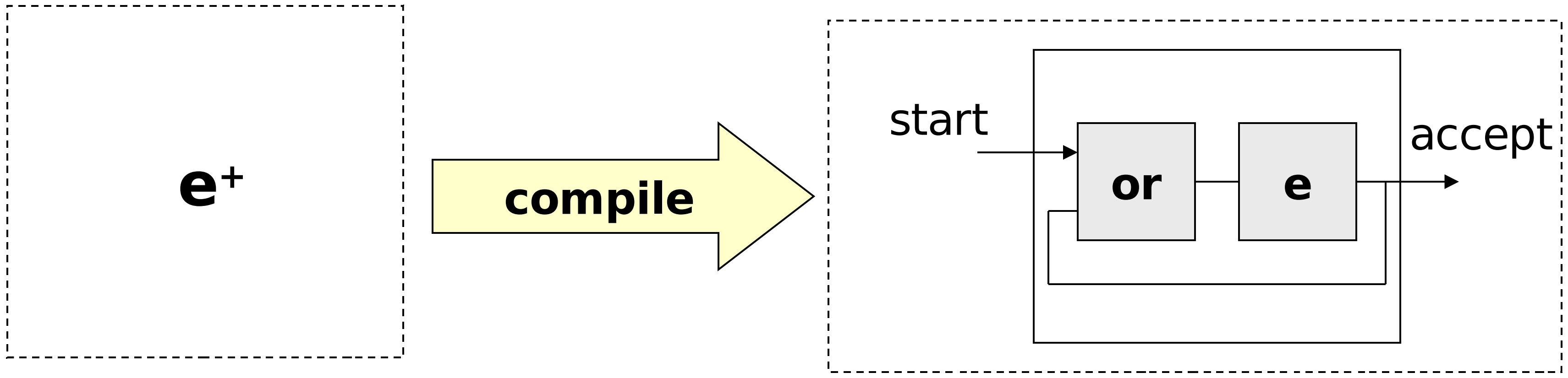
When *e.f* starts parsing, we start parsing *e*, and once it is accepted, we start parsing *f*.

Compiling Regular Expressions to Hardware



When *e+f* starts parsing, we start parsing both *e* and *f*, and accept once either of the two accepts.

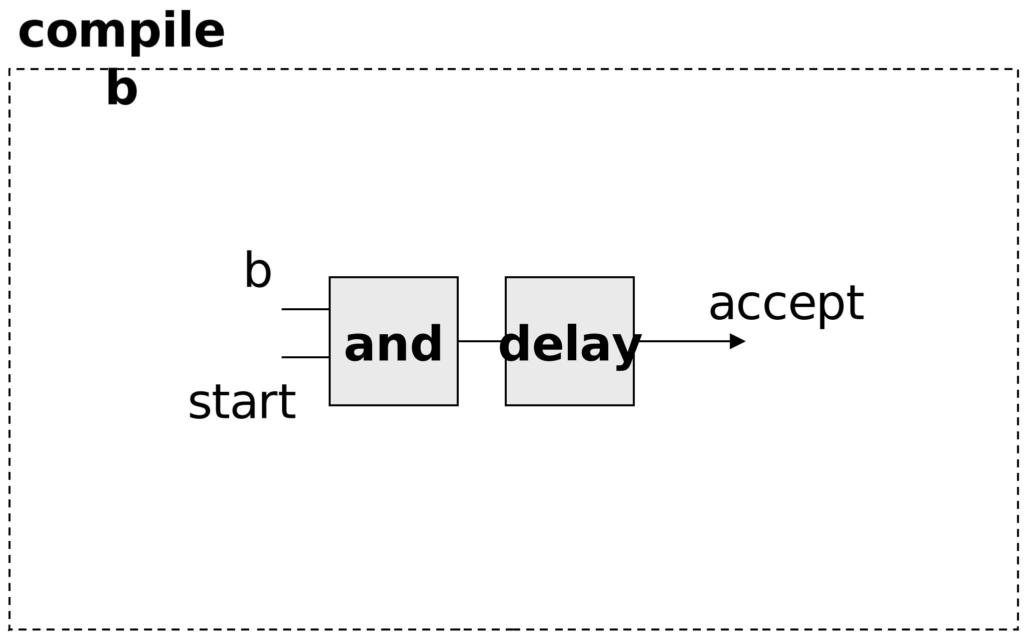
Compiling Regular Expressions to Hardware



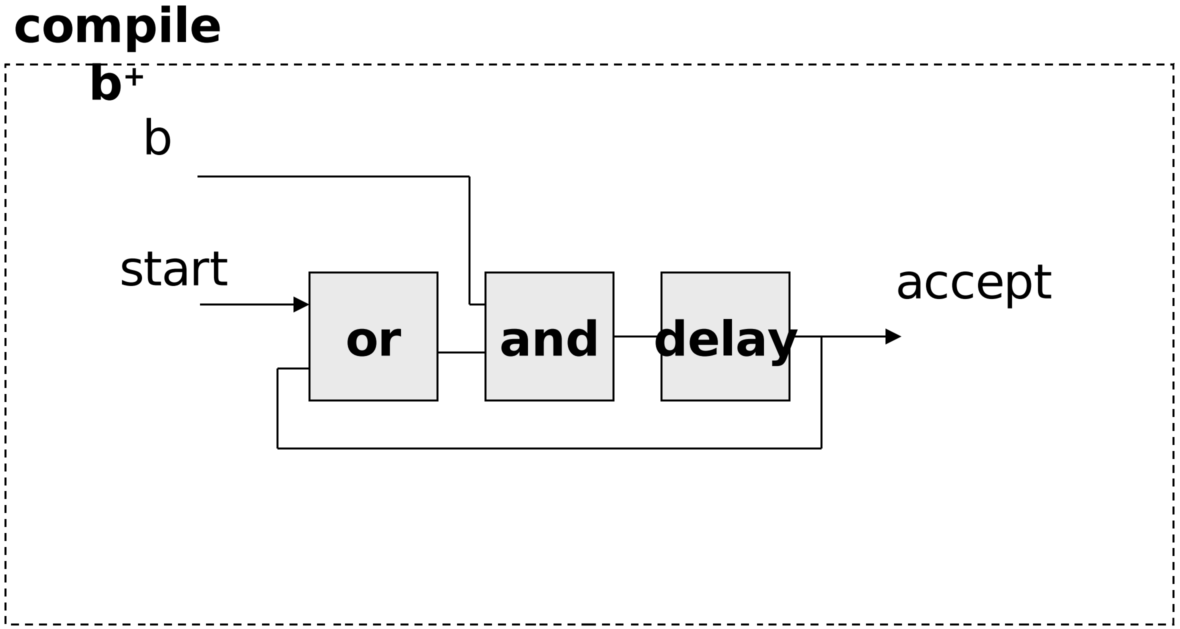
We start parsing *e* when *e*$^+$ starts parsing or *e* accepts. We accept every time *e* accepts.

Compiling a.(b$^+$+a)

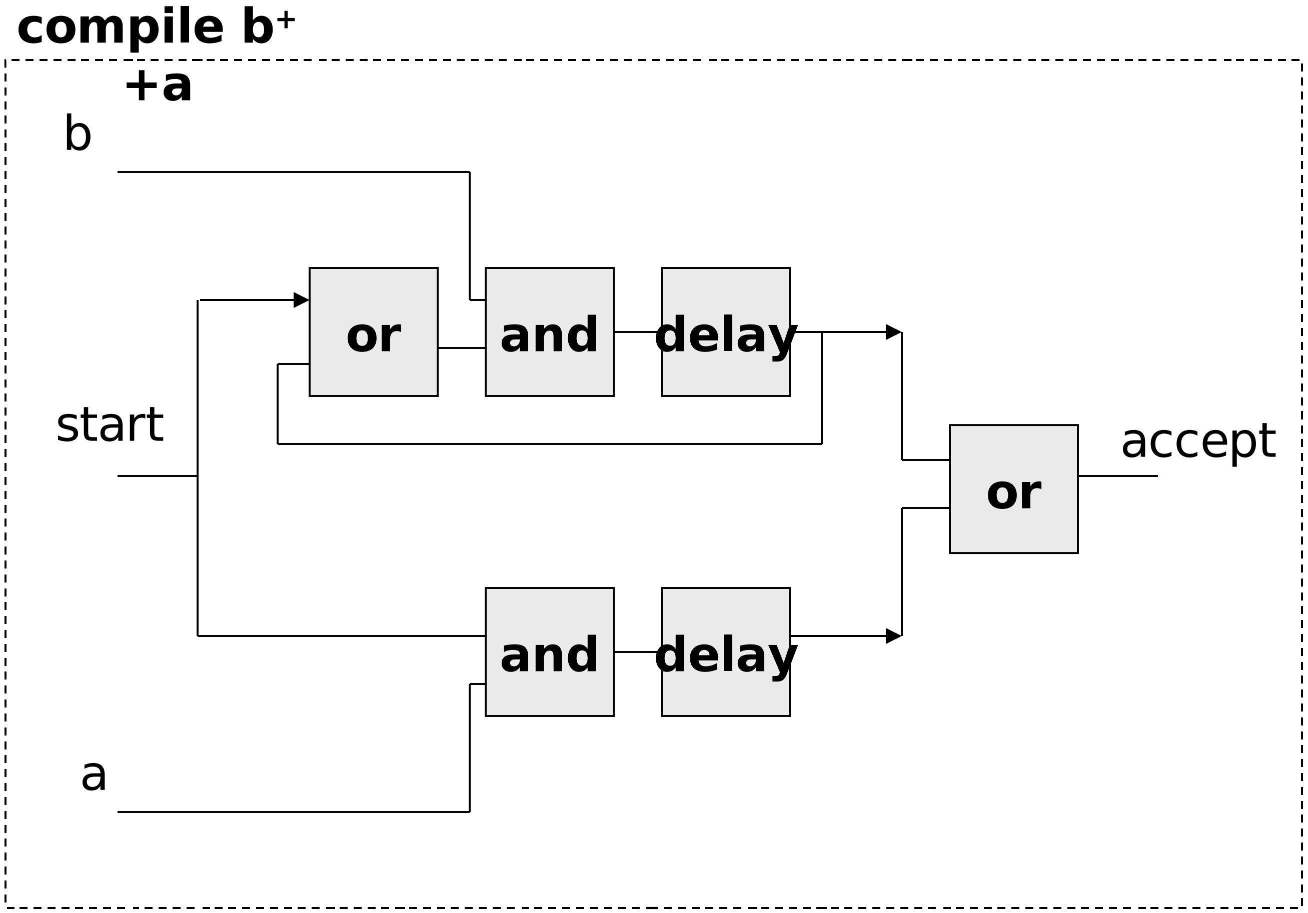
Compiling a.(b$^+$+a)



Compiling a.(b^++a)

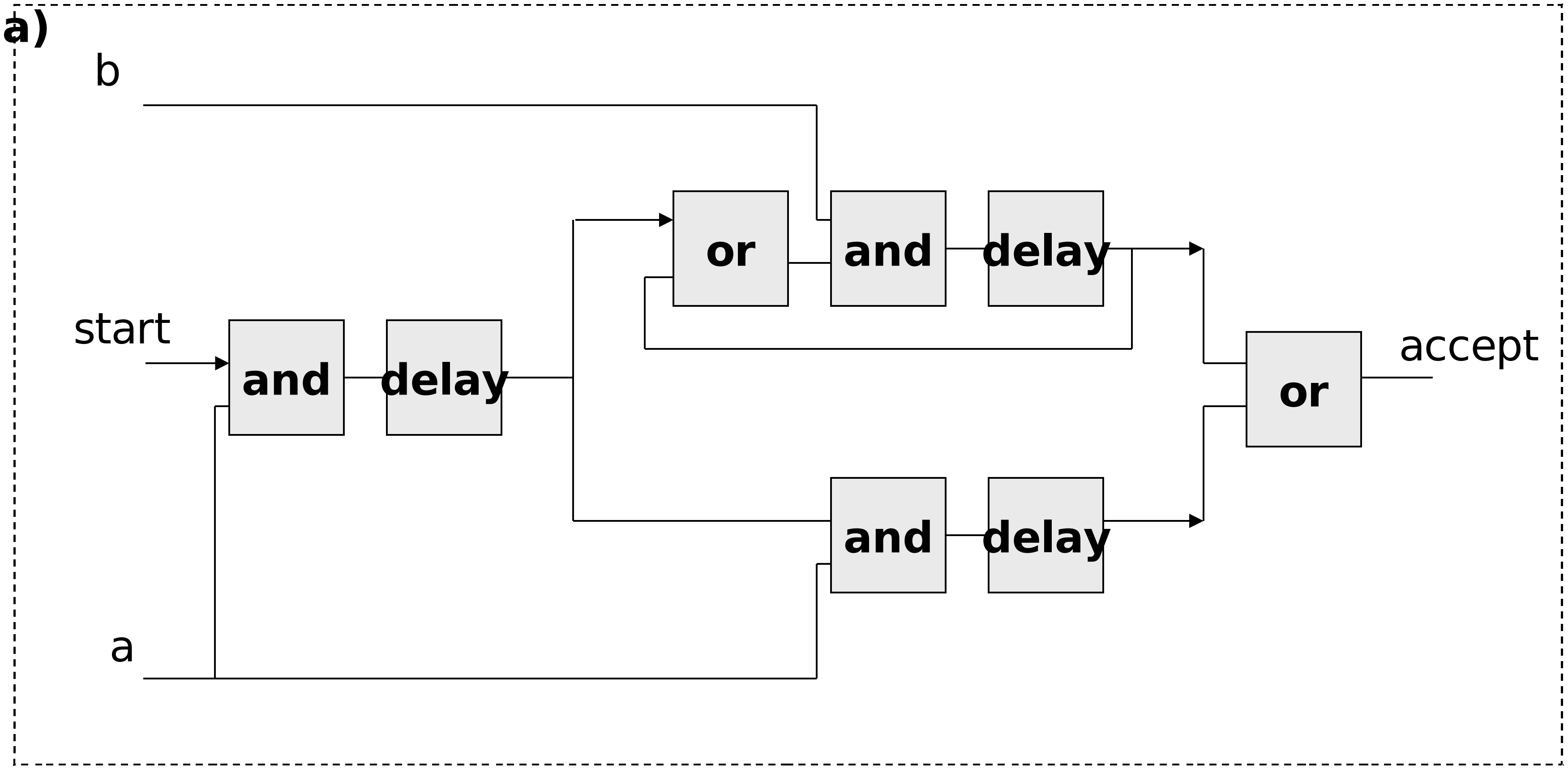


Compiling a.(b$^+$+a)



Compiling a.(b^++a)

compile a. (b^+ + a)



A Regular Expression Compiler in Lava

We embed regular expressions an a datatype:

```
data RegExp =
            Symbol (Signal Bool)
      |     Repeat RegExp
      |     RegExp :+: RegExp
      |     RegExp :>: RegExp
```

A Regular Expression Compiler in Lava

The circuits produced have a simple type:

```
type CircuitRE = Signal Bool -> Signal
   Bool
```

So does the compiler:

```
compileRE :: RegExp -> CircuitRE
```

A Regular Expression Compiler in Lava

The circuits produced have a simple type:

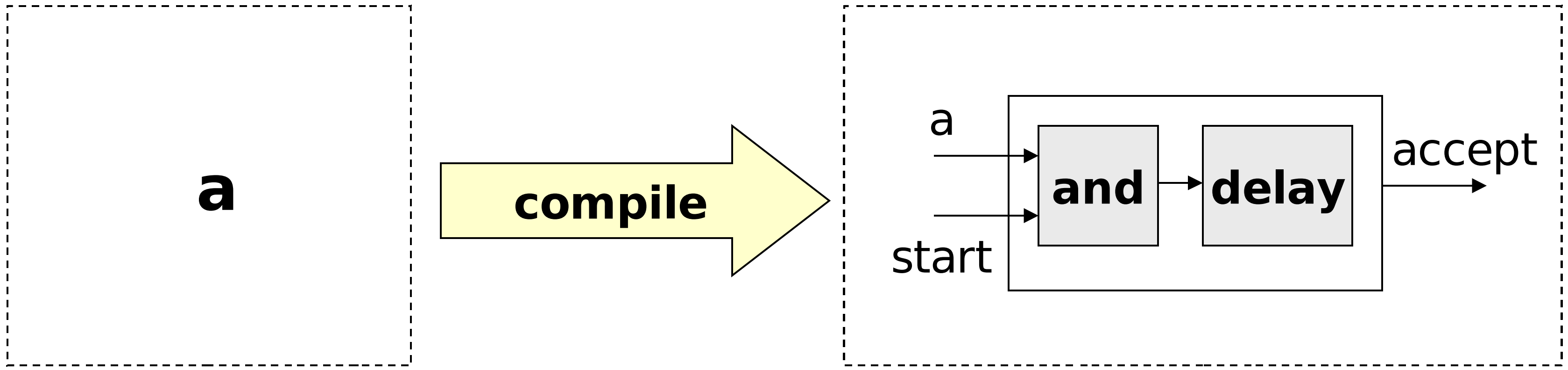
type CircuitR al Bool

We can now define compileRE using pattern matching on regular expressions

So does the compiler:

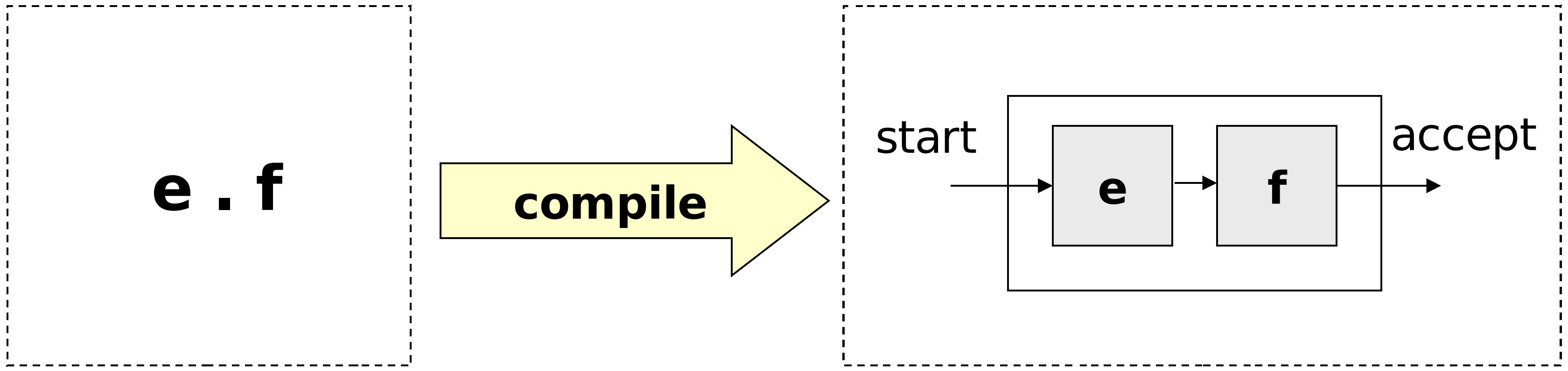
compileRE :: RegExp -> CircuitRE

A Regular Expression Compiler in Lava



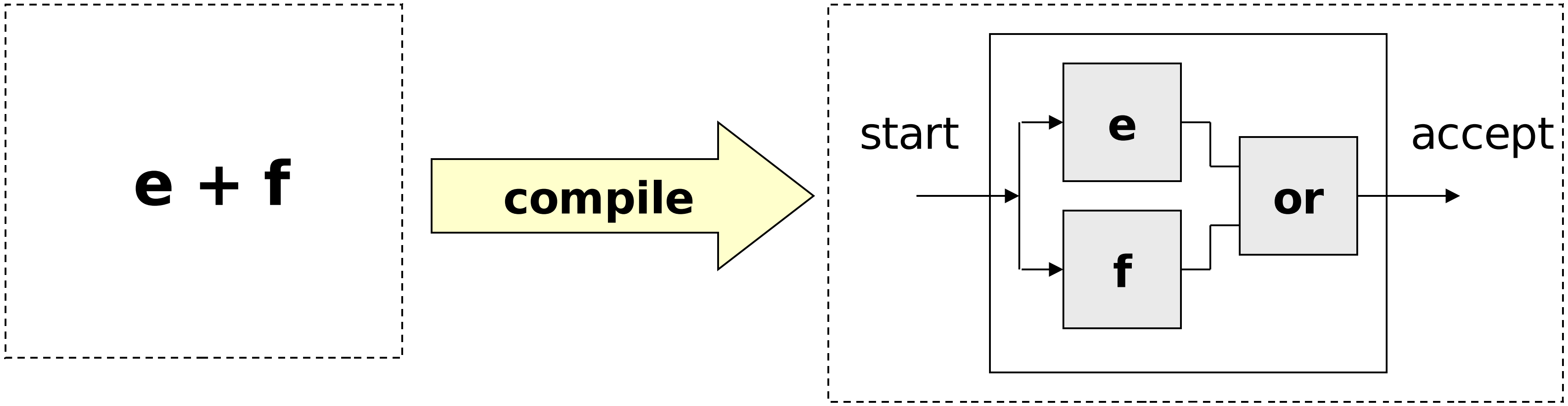
```
compileRE (Symbol a) start =
        delay low (a <&> start)
```

A Regular Expression Compiler in Lava



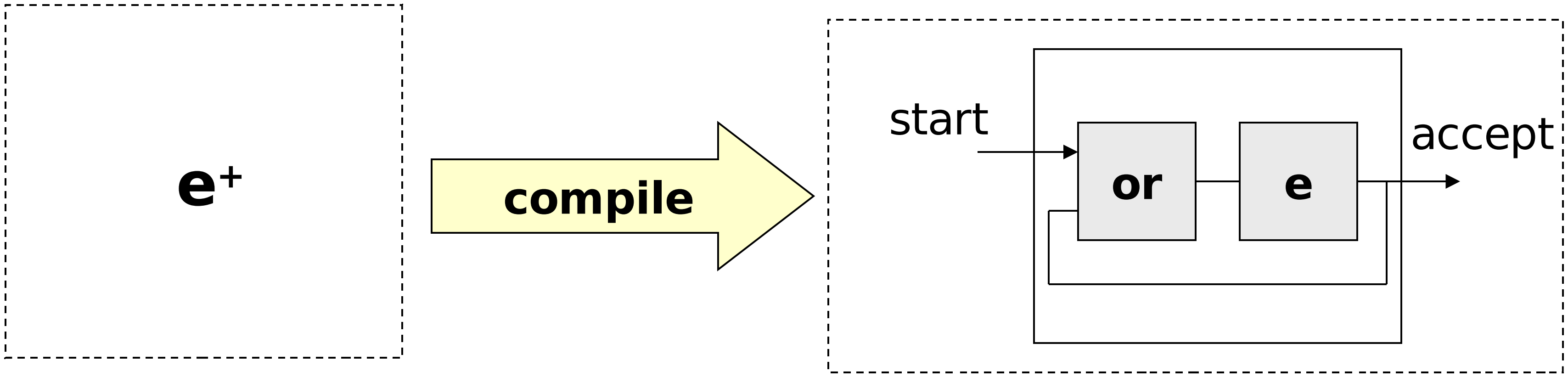
```
compileRE (e :>: f) start = accept
    where
        middle = compileRE e start
        accept = compileRE f middle
```

A Regular Expression Compiler in Lava



```
compileRE (e :+: f) start = accept
    where
          accept_e = compileRE e start
          accept_f = compileRE f start
          accept = accept_e <|> accept_f
```

A Regular Expression Compiler in Lava



```
compileRE (Repeat e) start = accept
    where
          accept = compileRE e start’
          start’ = start <|> accept
```

Simulating Regular Expression Circuits

```
example a b = s_a :>: (Repeat s_b :+: s_a)
    where
        s_a = Symbol a
        s_b = Symbol b

circuit (a,b) = compileRE (example a b) (delay high
    low)
```

```
Main> simulateSeq circuit
    [(high, high), (high, high), (low, high), (high, low)]
[low, low, high, high]
```

Compiling a Simple Imperative Language

$$
\begin{array}{lll}
\mathit{Prg} & ::= & \mathbf{skip} \\
 & \mid & \mathbf{delay} \\
 & \mid & \mathbf{emit} \\
 & \mid & \mathbf{if}\ \mathit{Signal}\ \mathbf{then}\ \mathit{Prg}\ \mathbf{else}\ \mathit{Prg} \\
 & \mid & \mathit{Prg}\ ;\ \mathit{Prg} \\
 & \mid & \mathbf{while}\ \mathit{Signal}\ \mathbf{do}\ \mathit{Prg} \\
 & \mid & \mathit{Prg}\ \|\ \mathit{Prg}
\end{array}
$$

Compiling a Simple Imperative Language

Prg ::= **skip**

|

lse *Prg*

A program has only one output variable, by default always carrying value low, unless it is actively pushed up to high.

| **while** *Signal* **do** *Prg*

| *Prg* || *Prg*

Compiling a Simple Imperative Language

Prg ::= **skip**

| **delay**

| **emit**

| **if** *Sig* … *Prg*

| *Prg* ; …

| **while** *Signal* **do** *Prg*

| *Prg* || *Prg*

Does nothing and terminates immediately

Compiling a Simple Imperative Language

Prg ::= **skip**
| **delay**
| **emit**
| **if** *Sig* *Prg*
| *Prg* ;
| **while**
| *Prg* || *Prg*

Does nothing and terminates one clock tick later

Compiling a Simple Imperative Language

Prg ::= **skip**
| **delay**
| **emit**
| **if** *Sign* … **e** *Prg*
| *Prg* ;
| **while**
| *Prg* ||

Terminates immediately. Pushes that the output of the program to high.

Compiling a Simple Imperative Lang

Standard conditional, branching depending on current value of a given signal

Prg ::= **skip**
| **delay**
| **emit**
| **if** *Signal* **then** *Prg* **else** *Prg*
| *Prg* ; *Prg*
| **while** *Signal* **do** *Prg*
| *Prg* || *Prg*

Compiling a Simple Imperative Language

Prg ::= **skip**

| **delay**

| **emit**

| **if** *Signal* **th** *rg* **else** *Prg*

| *Prg* ; *Prg*

| **while** *Signal* **do** *Prg*

| *Prg* || *Prg*

Standard sequential composition: execute the first program, and upon termination, execute the second

Compiling a Simple Imperative Language

Prg	::=	**skip**
	|	**delay**
	|	**emit**
	|	**if** *Signal* **t**
	|	*Prg* ; *Prg*
	|	**while** *Signal* **do** *Prg*
	|	*Prg* || *Prg*

Standard loop, choice depending on current value of a given signal

Compiling a Simple Imperative Language

Prg ::= **skip**
| **delay**
| **emit**
| **if** *Signal*
| *Prg* ; *Prg*
| **while** *Sign* **do** *Prg*
| *Prg* || *Prg*

Fork-join parallel composition. Start the two programs together, and terminate once both have terminated

Example Programs

```
alternate =
  while high do
      emit; delay; delay

wait s =
  while (inv s) do delay
```

Embedding in Haskell

```
data Prg =
        Skip
    |   Emit
    |   Delay
    |   IfThenElse (Signal Bool) (Prg, Prg)
    |   Prg :>: Prg
    |   While (Signal Bool) Prg
    |   Prg :|: Prg
```

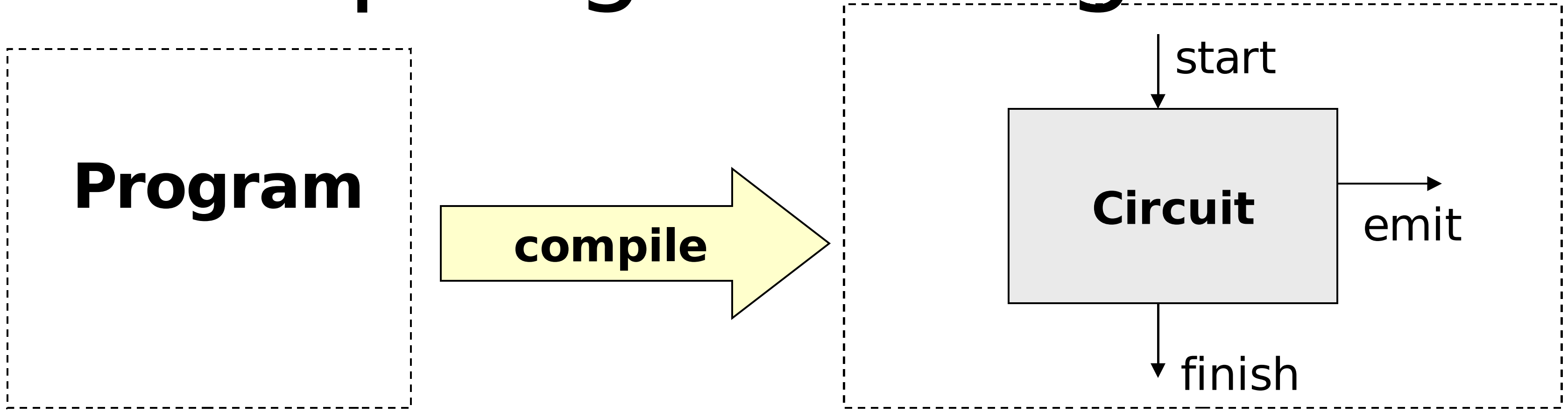
Generating Programs

```
forever prg = While high prg

emitD = Emit :>: Delay

wait s = While (inv s) Delay

inverter s = forever (wait (inv s) :>:
   emitD)
```

Example Programs

```
onOff = forever ( emitD :>: Delay )
offOn = Delay :>: onOff


rising1 s =
  forever (While (inv s) Delay :>:
  emitD)
rising2 s =
  forever (wait (inv s) :>: wait s :>:
  Emit)
```

Compiling the Programs

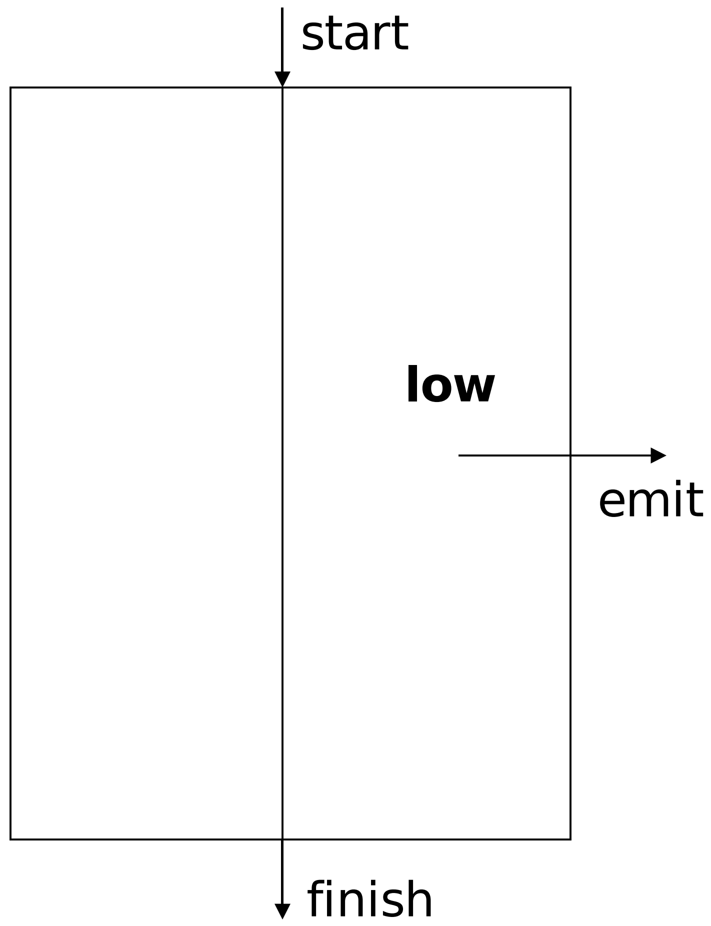


type CircuitPrg =

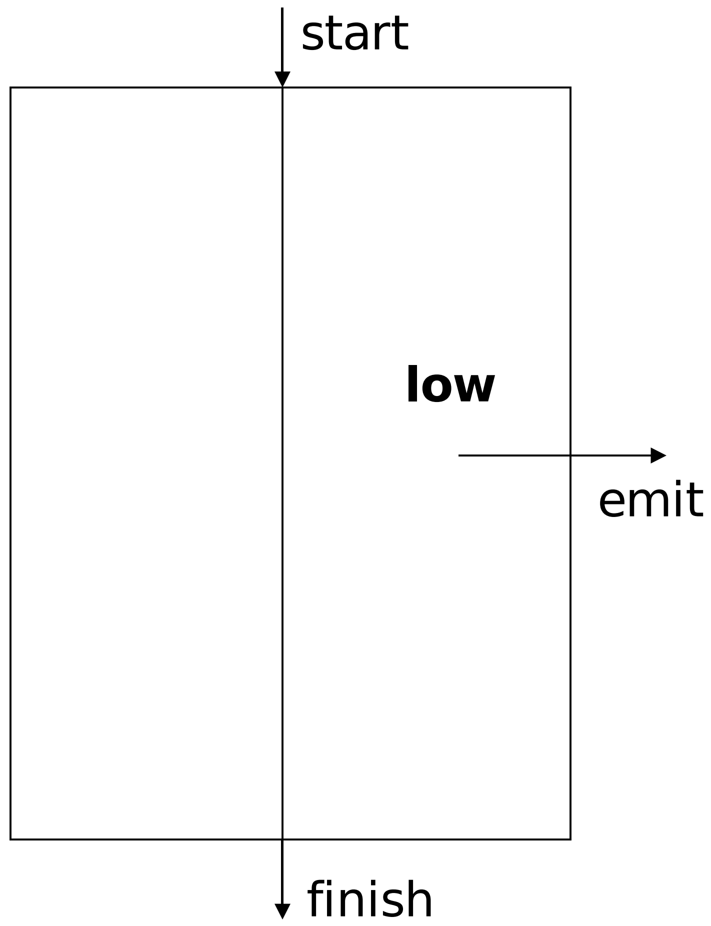
Signal Bool -> (Signal Bool, Signal Bool)

compilePrg :: Prg -> CircuitPrg

Compiling Skip

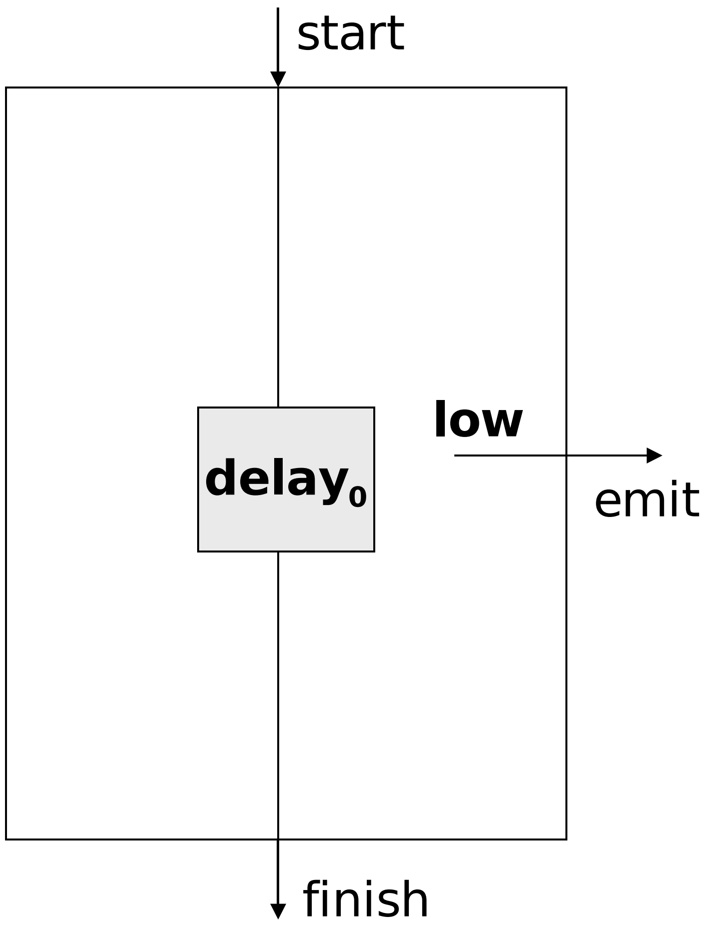


Compiling Skip

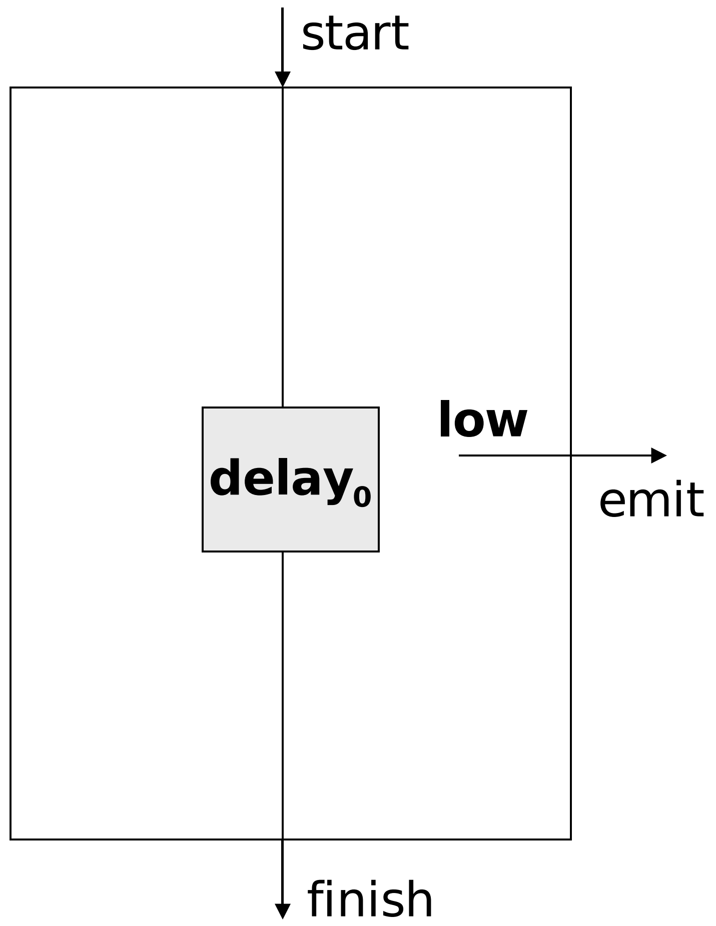


```
compilePrg Skip start =
   (finish, emit)
   where
      finish = start
      emit = low
```

Compiling Delay

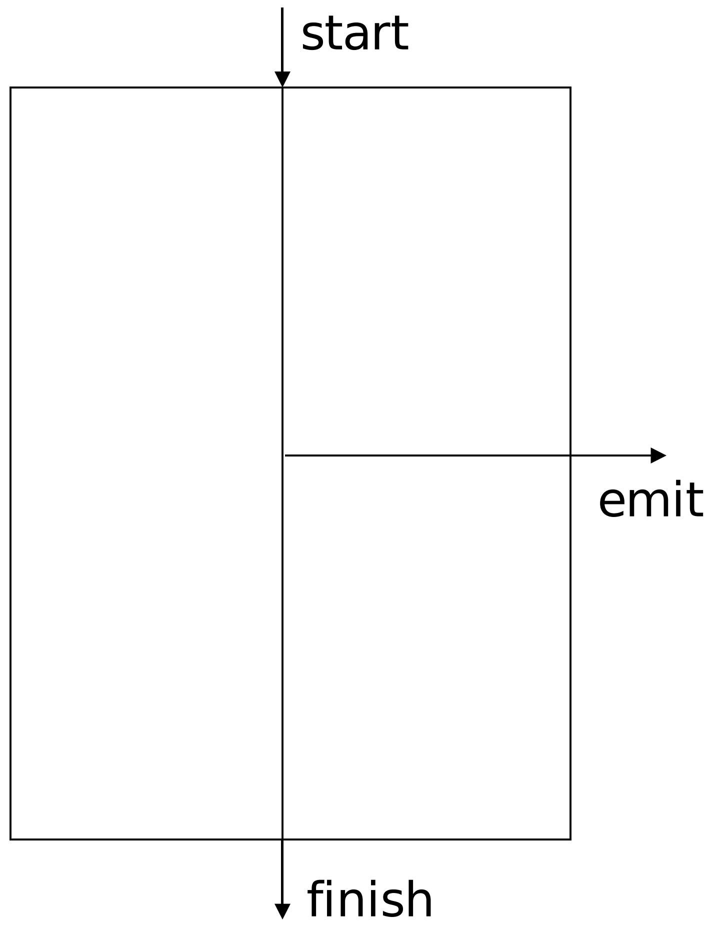


Compiling Delay

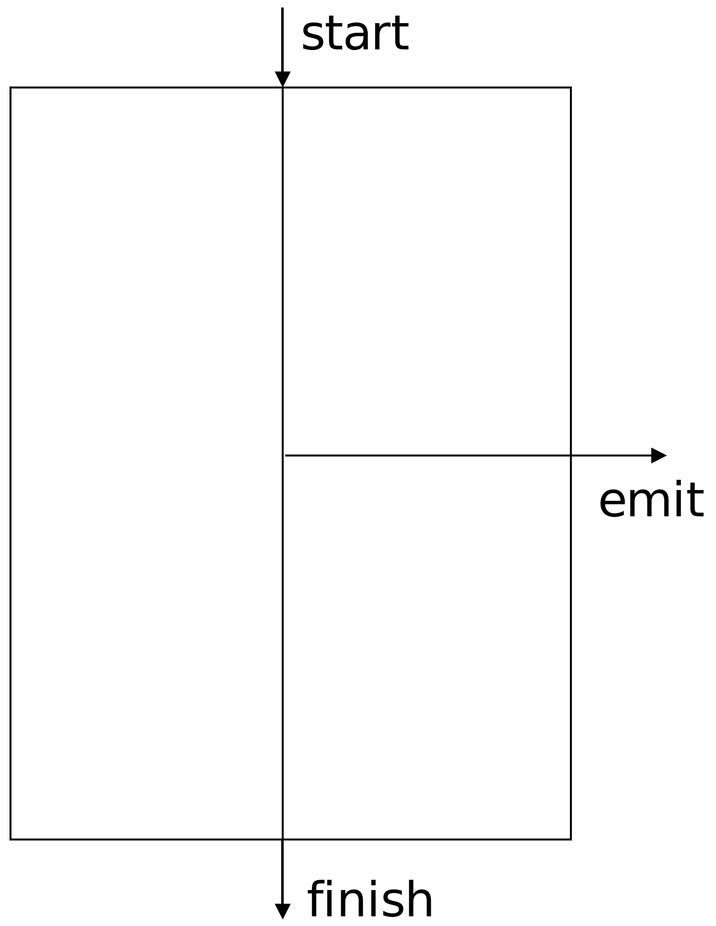


```
compilePrg Delay start
  =
  (finish, emit)
  where
     finish = delay low
  start
     emit = low
```

Compiling Emit

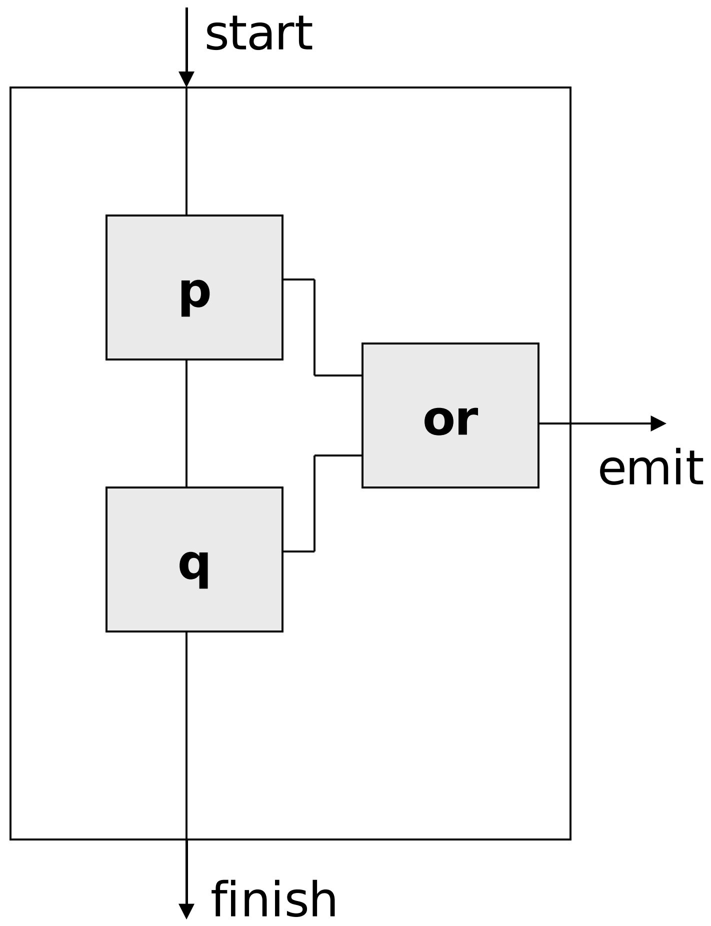


Compiling Emit

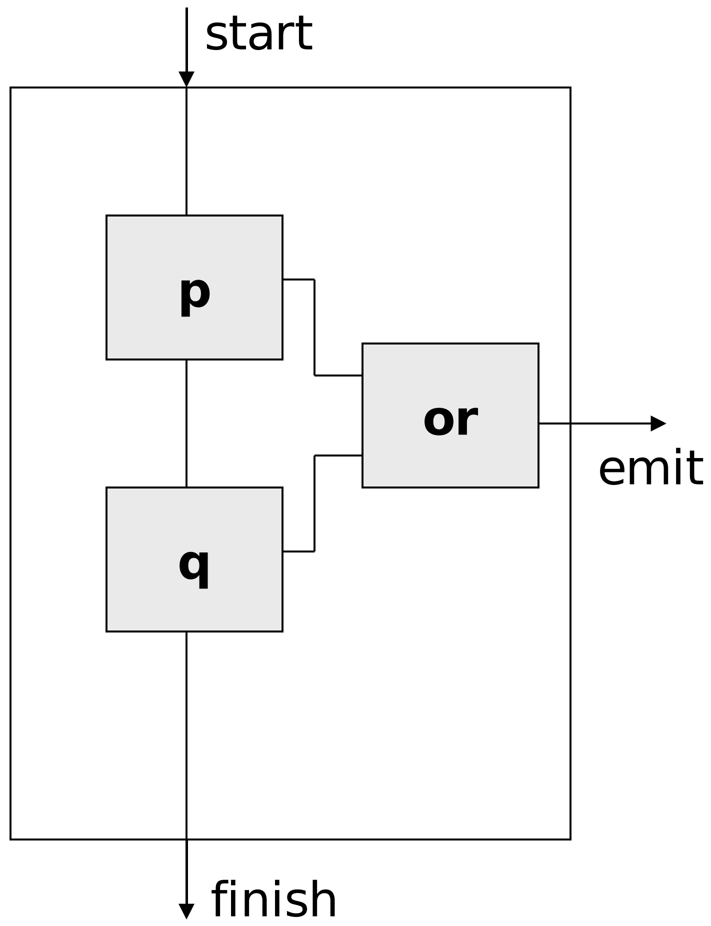


```
compilePrg Emit start =
   (finish, emit)
   where
      finish = start
      emit = start
```

Compiling Sequential Composition

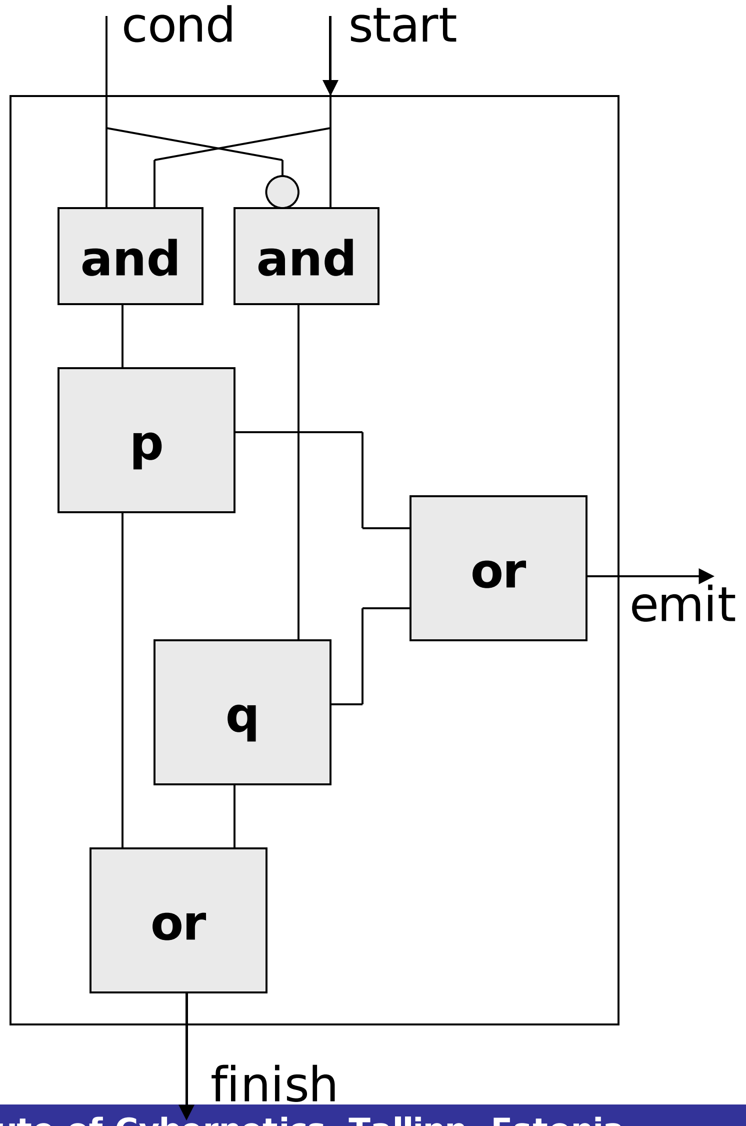


Compiling Sequential Composition

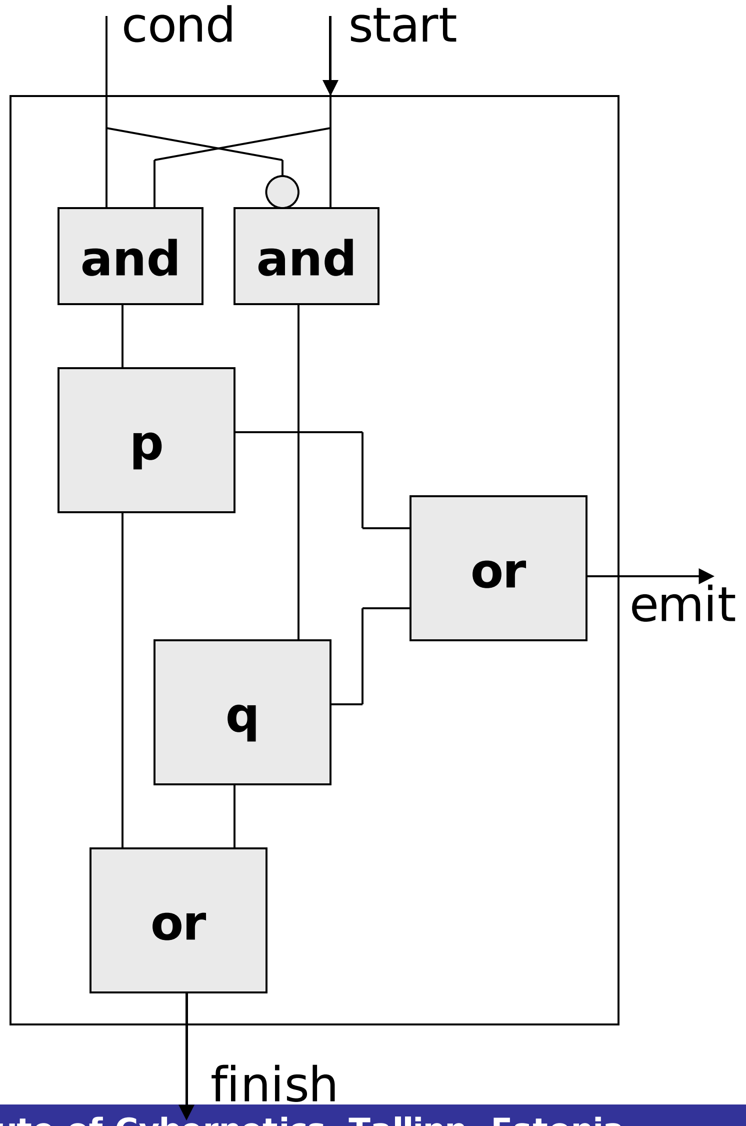


```
compilePrg (p :>: q) start =
   (finish, emit)
   where
      (finish_p, emit_p) =
        compilePrg p start
      (finish, emit_q) =
        compilePrg q finish_p

      emit = or2 (emit_p,
   emit_q)
```

Compiling Conditionals

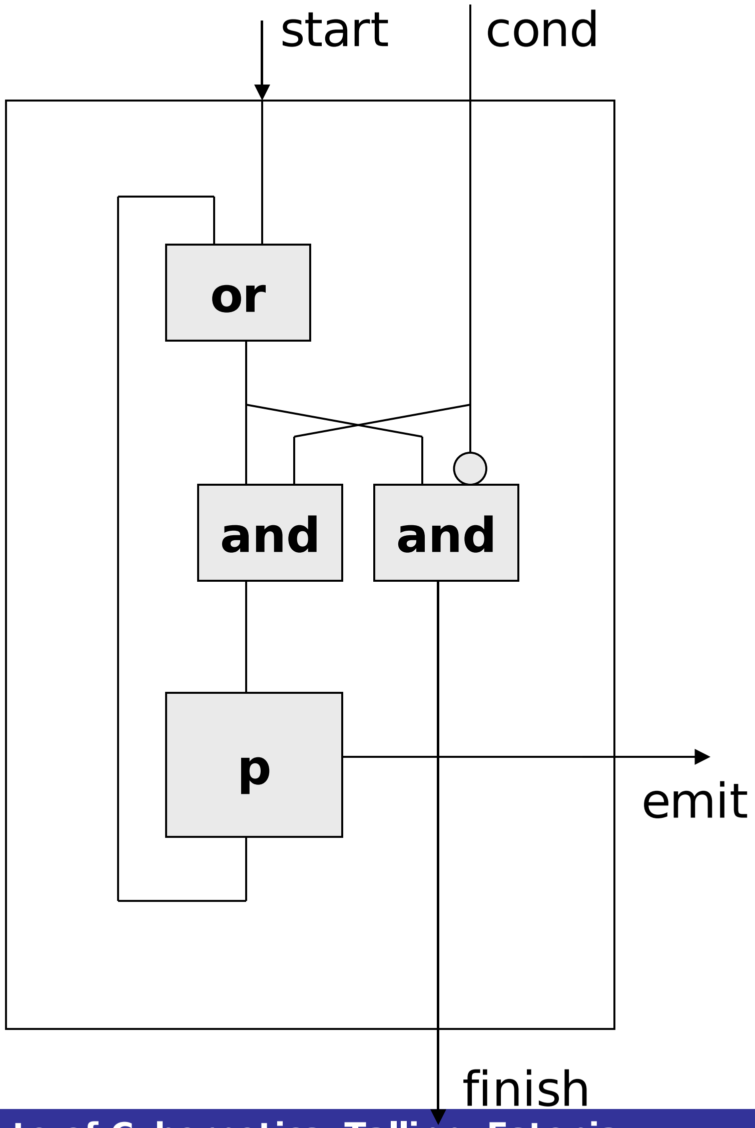


Compiling Conditionals

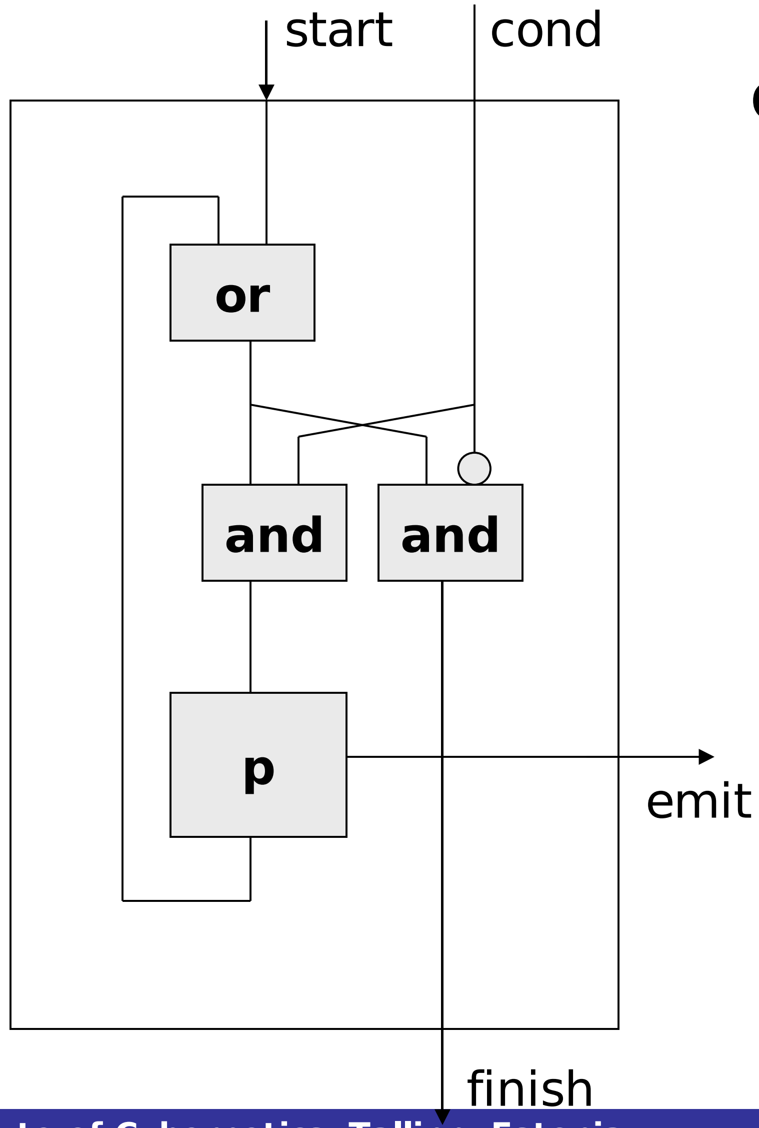


```
compilePrg
   (IfThenElse c (p,q)) start =
   (finish, emit)
   where
      start_p = and2 (start, c)
      start_q = and2 (start, inv c)

      (finish_p, emit_p) =
        compilePrg p start_p
      (finish_q, emit_q) =
        compilePrg q start_q

      emit = or2 (emit_p, emit_q)
      finish = or2 (finish_p, finish_q)
```

Compiling Loops



Compiling Loops



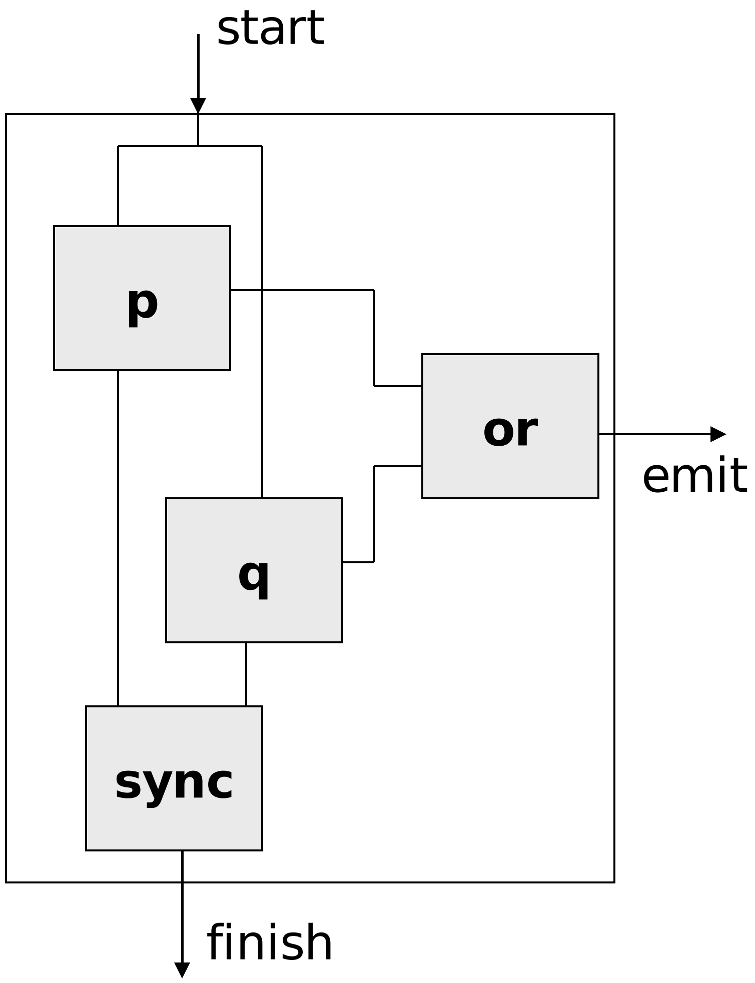
compilePrg (While c p) start
=
(finish, emit)
where
start' = or2 (start,
finish_p)

start_p = and2 (start', c)

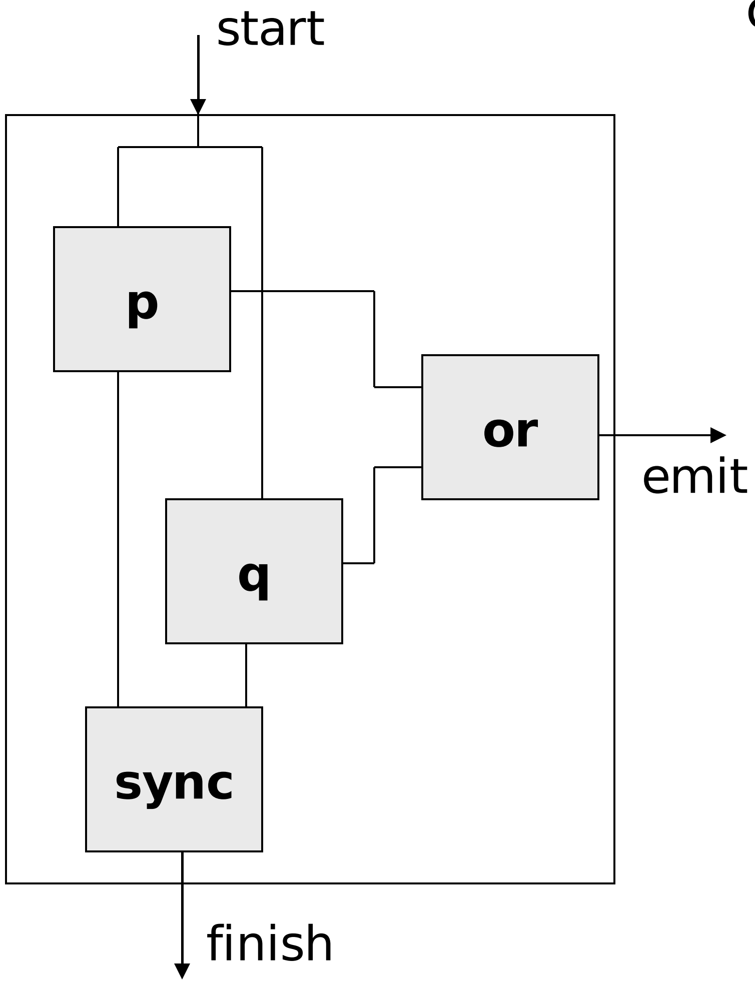
(finish_p, emit) =
compilePrg p start_p

finish = and2 (start', inv
c)

Compiling Parallel Composition

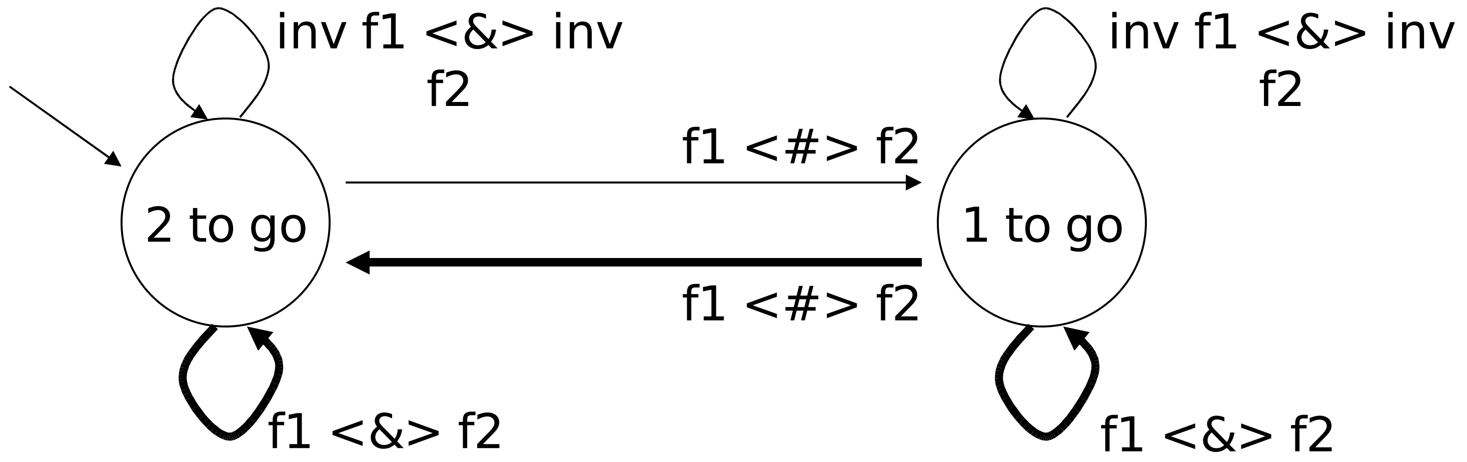


Compiling Parallel Composition



```
compilePrg (p :|: q) start =
  (finish, emit)
  where
    (finish_p, emit_p) =
      compilePrg p start
    (finish_q, emit_q) =
      compilePrg q start

    emit = or2 (emit_p, emit_q)
    finish = sync (finish_p,
  finish_q)
```

The Synchroniser



```
sync (f1, f2) = ...
  where
    state = ...
```

The Synchroniser (assuming neither branch may terminate immediately)

```
syncPrg (f1, f2) =
   forever (
      wait (f1 <|> f2) :>:
      IfThenElse (f1 <&> f2)
         ( Skip
         , wait (f1 <|> f2)
         ) :>:
      emitD
   )

sync f12 = compilePrg (syncPrg f12) (delay high
   low)
```

Verification

```
inverter s = forever (wait (inv s) :>: emitD)

propInv s = is1 <==> is2
  where
    is1 = inv s
    (_, is2) = compilePrg (inverter s) (delay high
  low)
```

Main> smv propInv

Proving: ... Valid

Verification

```
propRise s = r1 <==> r2
  where
    (_, r1) = compilePrg (rising1 s) (delay high
   low)
    (_, r2) = compilePrg (rising2 s) (delay high
   low)
```

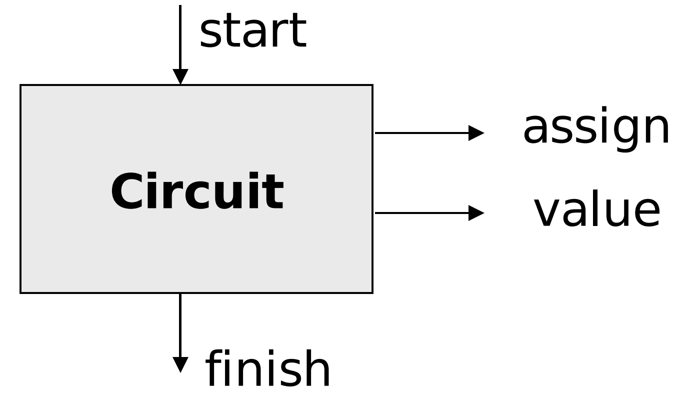
```
Main> smv propRise
Proving: ... Valid
```

Extensions: Assignment Variable

Instead of an **emit** output variable, we would like to have an output variable which can be assigned a value which is remembered:

```
data Prg = ...
         | Assign (Signal Bool)
           ...
```

Extensions: Assignment Variable



- The output wire **assign** is high when a new value is being assigned to the variable.
- The new value is available on the **value** wire.

Extensions: Assignment Variable

At the top level, the wires are combined together:

```
register (assign, value) = current
  where
     current = mux (assign, (previous, value))
     previous = delay low current

compile program start = (finish, output)
  where
     (finish, assign_value) = compileAux program
  start
     output = register assign_value
```

Extensions: Assignment Variable

To compile an assignment:

```
compileAux (Assign w) start =
    (finish, (assign, value))
    where
        finish = start
        assign = start
        value = w
```

Extensions: Assignment Variable

Combining the **assign** and **value** wires of two blocks is slightly more complicated than in the case of an **emit** variable:

```
combine ((a1, v1), (a2, v2)) = (a, v)
    where
        a = a1 <|> a2
        v = mux (a1, (v1, v2))
```

Extensions: Multiple Variables

Adding multiple output wires can be done in various ways:

- **Reference by index:** Before compilation a pass through the program will identify the number of wires required.
- **Scoped and named declarations:** The compiler will have to augment its behaviour with an auxiliary symbol table, relating variable names and output wire list index.

Extensions: Feedback of Variables

To add expressions involving output wires in conditions and assignments, feedback loops can be created. To resolve the problem:

- Delay assignment and emission by one clock tick; or
- Make a constructivity analysis before generating the circuit – this can be very expensive to perform…

Conclusions

- A compiler is just a sophisticated parametrised circuit.
- Although one loses in efficiency, the trade-off is more reliable circuitry.
- Adding more features can be challenging.
- Combining languages can be useful – although in most cases it is just a matter of combining *language features*.

Exercises (1)

- Implement the embedded regular language and the embedded imperative language (with one emit wire) hardware compiler with emit variables.

Exercises (2)

- Use Lava and SMV to verify whether:

sync (a,b)

=

compilePRG (wait a :|: wait b) (delay high low)

Exercises (3)

- It was remarked that loops only work if their body takes time to execute.
  - Define a function *loopBodies*, which given a program *p* returns all subprograms which are bodies of a loop of *p* in innermost first order.
  - Write a function *takesTime*, which given a program *p*, returns an observer which checks whether *p* always takes time to execute.
  - Combine the previous two functions to check that all loop bodies of a given program take time to execute.

Exercises (4)

- Add multiple emit variables to the language. The compiled program should output a list of variable values, and the instruction *Emit n* will push the value of the *n*th variable up to high.
- Add a new command in the imperative language *RE s*, where *s* is a regular expression, which allows a regular expression to appear as part of a program.

Functional Languages for Synchronous Hardware Design and Verification

Part 5: Other Functional HDLs

Gordon J. Pace

Department of Computer Science & AI

University of Malta

Overview

- Lava is primarily aimed at hardware design and verification.
- Other functional HDLs have different objectives.
- We will be looking at three other functional HDLs, which are different from Lava.

Wired

- Wired is a functional HDL, embedded in Haskell.
- Primary aim is to address non-functional aspects of a circuit:
  - Timing
  - Netlist topology
  - Layout
  - Routing
  - Power consumption

Wired

- Wires account for 75% of path delays and 50% of power consumption – unlike Lava, Wired is aware of wires.
- Lava is functional, due to a uniquely forward flow of information in the circuit. Wired requires bidirectional flow to calculate things such as load.
- Wired uses layout combinators to construct circuits, which are stored as relational blocks.

Case Study: Prefix Circuits

The Problem:

Given inputs x_1, x_2 … x_n and an associative operator $\otimes$.

Calculate:

x_1

$x_1 \otimes x_2$

$x_1 \otimes x_2 \otimes x_3$

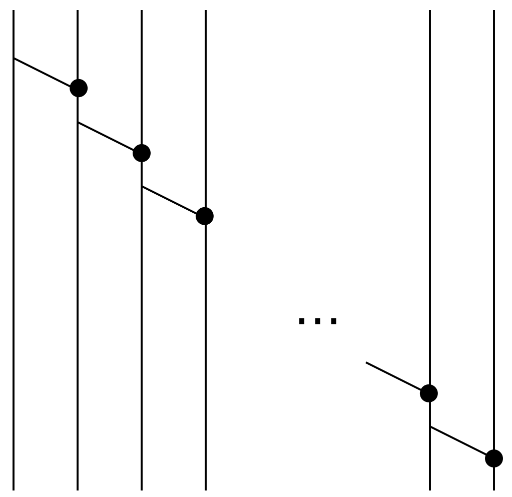
…

$x_1 \otimes x_2 \otimes \ldots \otimes x_n$

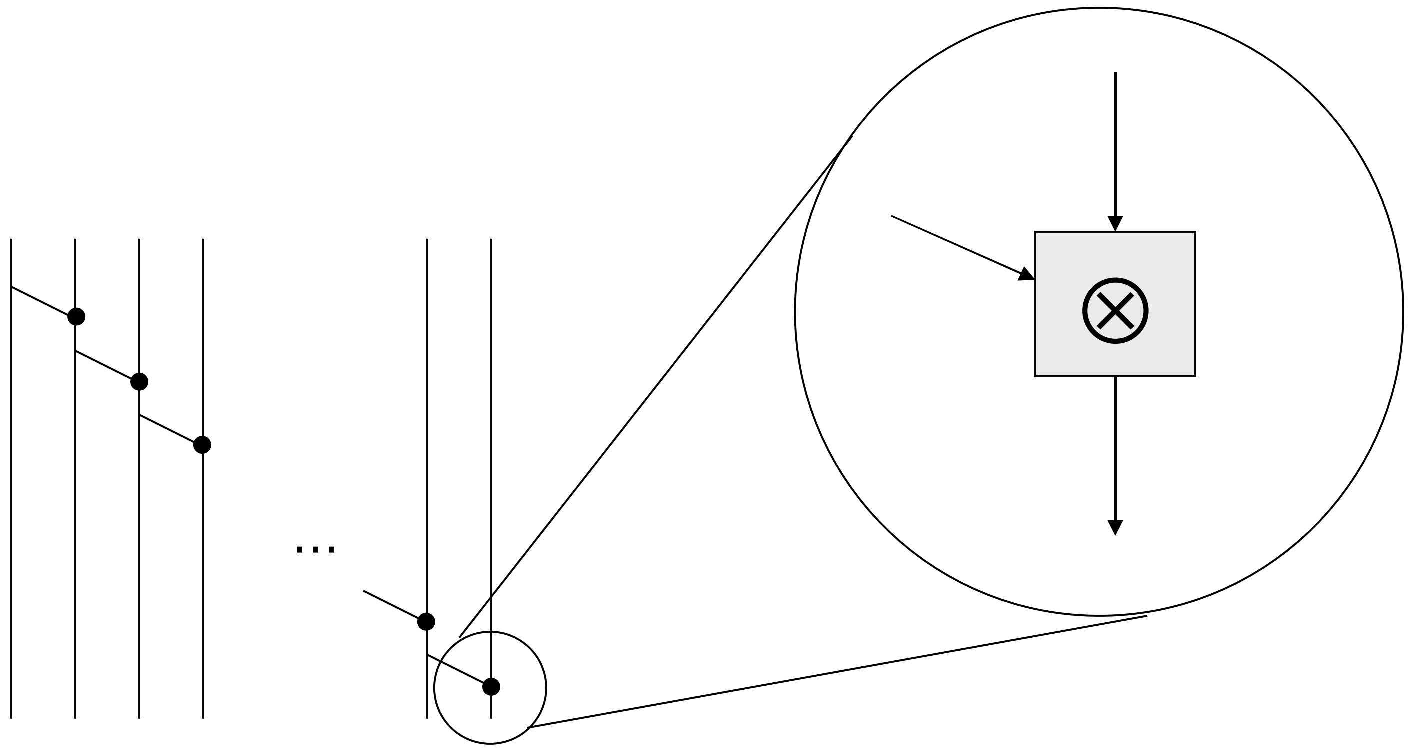
Case Study: Prefix Circuits

- Applications include fast adders, priority encoders, etc.
- The objective is to calculate the required outputs in a shallow and small circuit, parametrised by the binary operator $\otimes$.
- Wired uses non-standard interpretations to calculate delays and other circuit features.

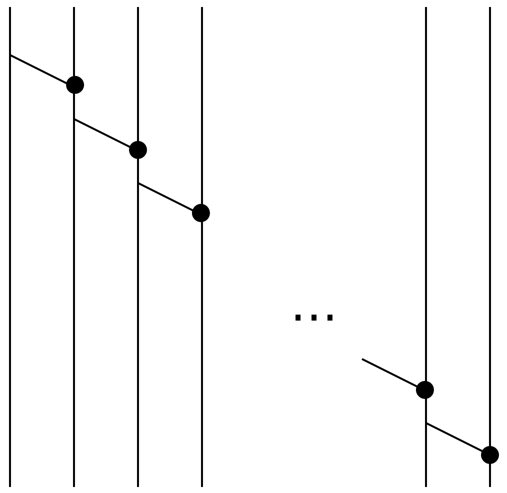
A Serial Prefix



A Serial Prefix



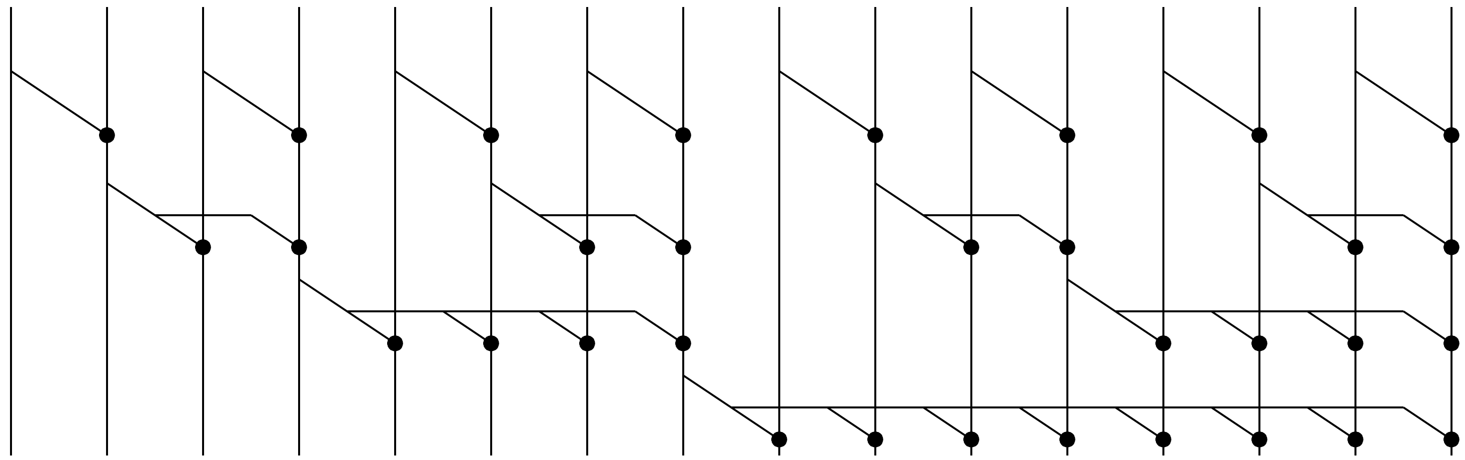
A Serial Prefix



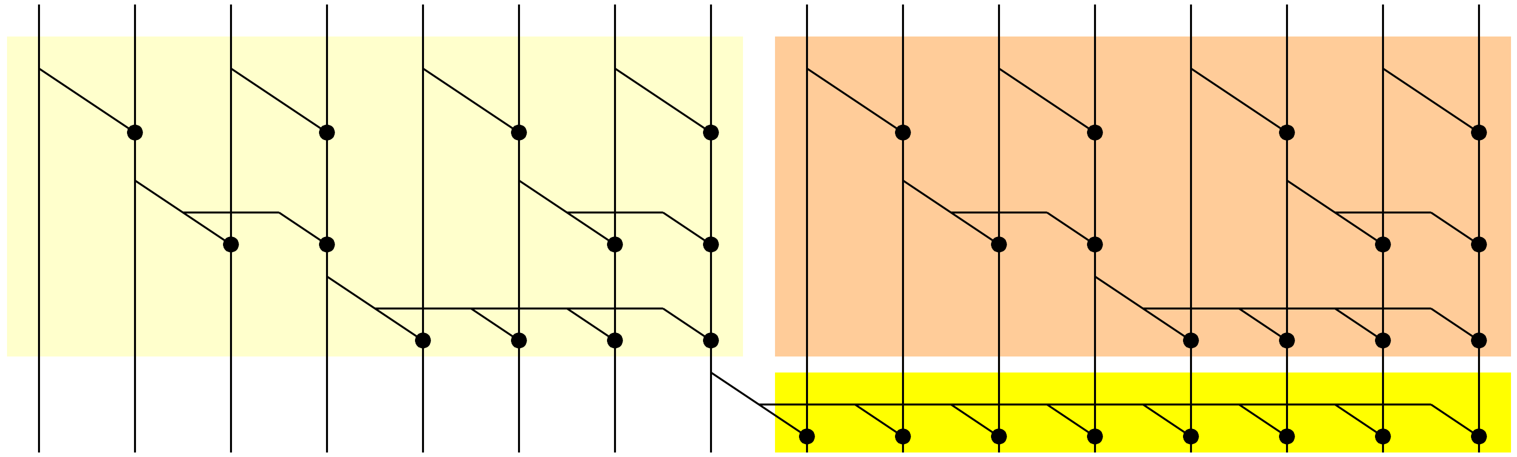
Lava:

```
serial op [x] = x
serial op (x:y:xs) =
  x: serial op (op (x,
   y):xs)
```

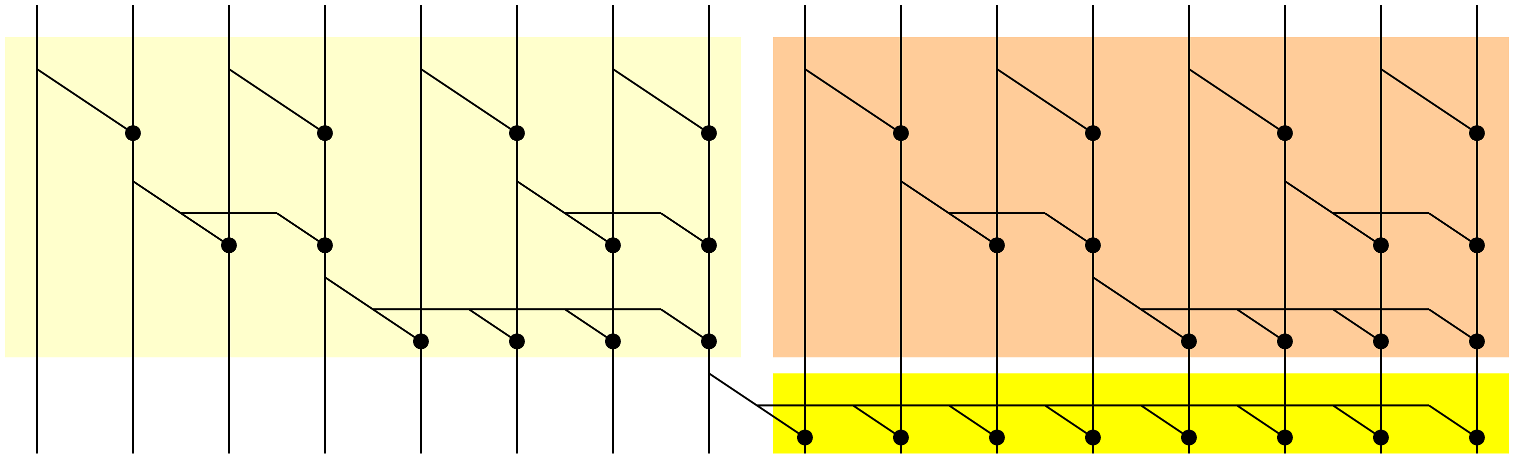
Sklansky Prefixes



Sklansky Prefixes

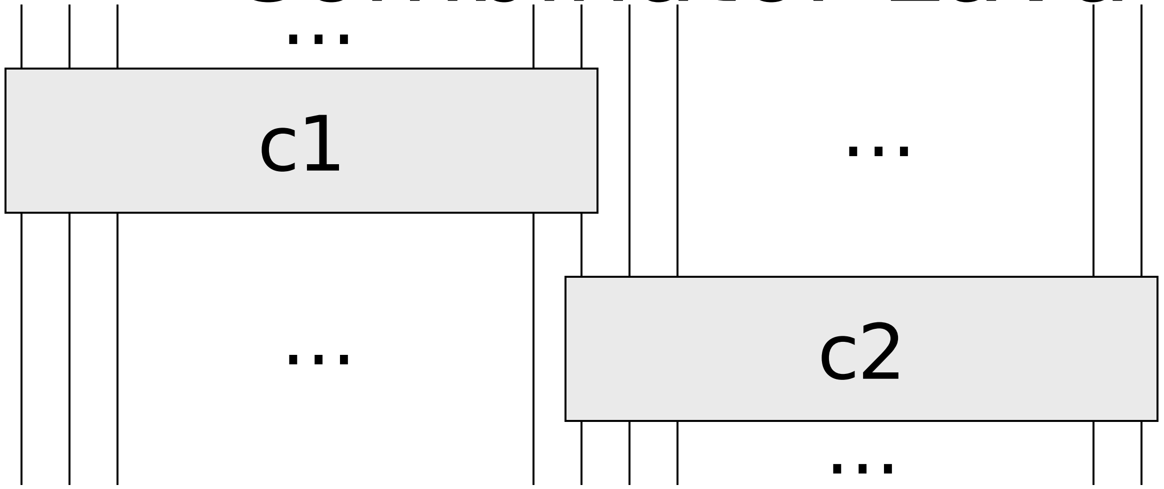


Sklansky Prefixes in Lava



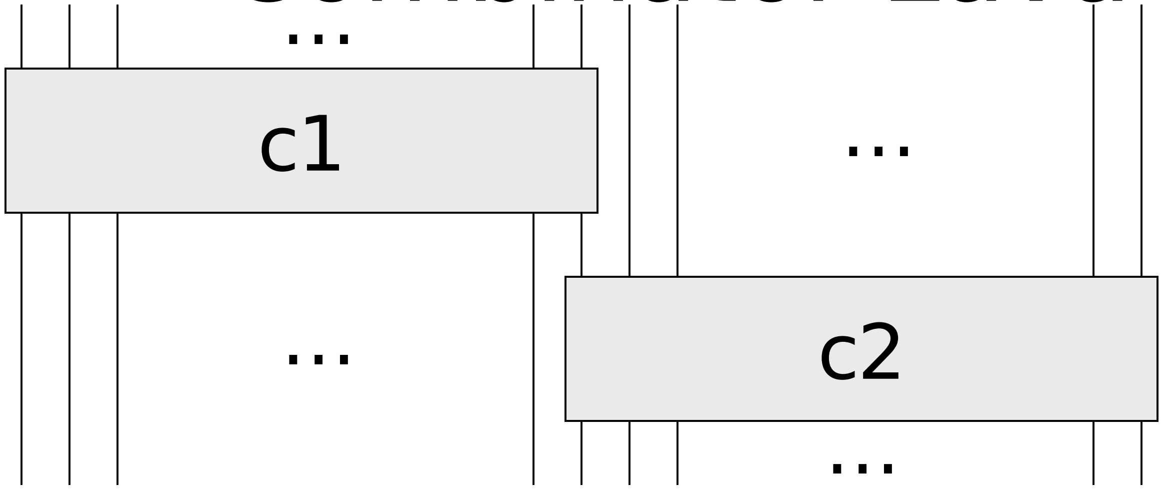
```
sklansky op [x] = [x]
sklansky op xs = ls' ++ [ op (carry, r) | r <- rs' ]
    where
        (ls, rs) = splitAt (length xs `div` 2) xs
        ls' = sklansky op ls
        rs' = sklansky op rs
        carry = last ls'
```

Redoing Things Using Combinator Lava



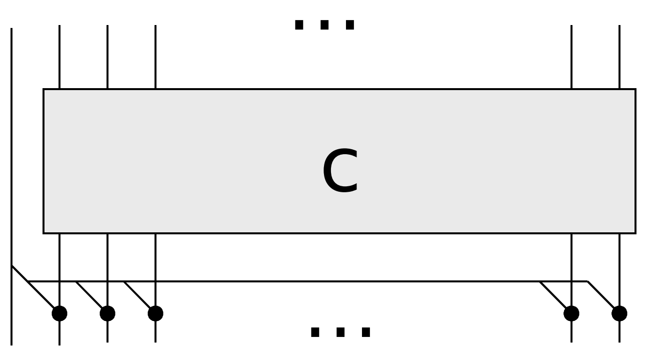
```
overlap k c1 c2 xs = init ls’ ++ rs’
   where
        (ls, rs) = splitAt k  xs
        ls’ = c1 ls
        rs’ = c2 (last ls’: rs)
```

Redoing Things Using Combinator Lava



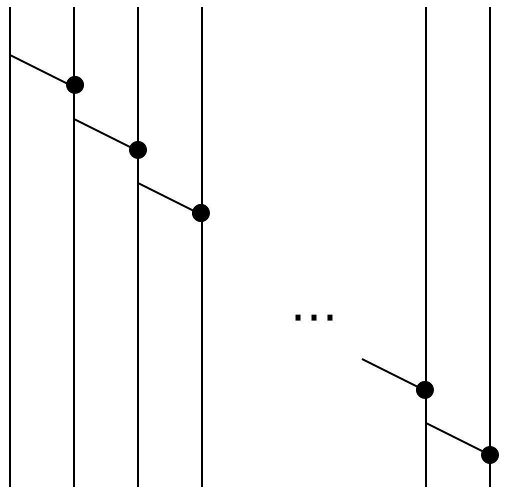
```
overlap2 c1 c2 xs =
        overlap (length xs `div` 2) c1 c2 xs
```

Redoing Things Using Combinator Lava



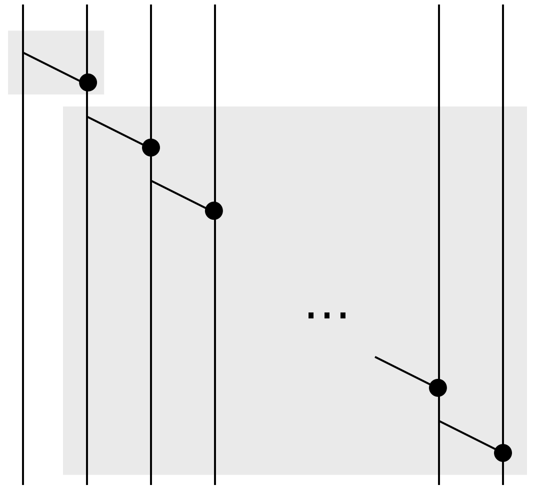
```
extend op c (x:xs) = x: [ op (x, x') | x'
   <- xs' ]
   where
       xs' = c xs
```

Back to the Serial Prefix



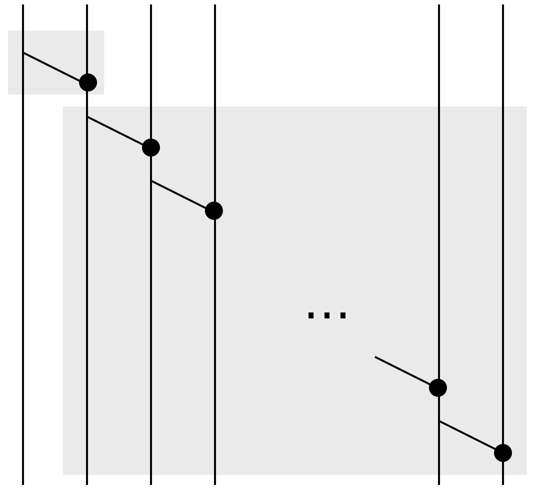
Combinator Lava:

Back to the Serial Prefix



Combinator Lava:

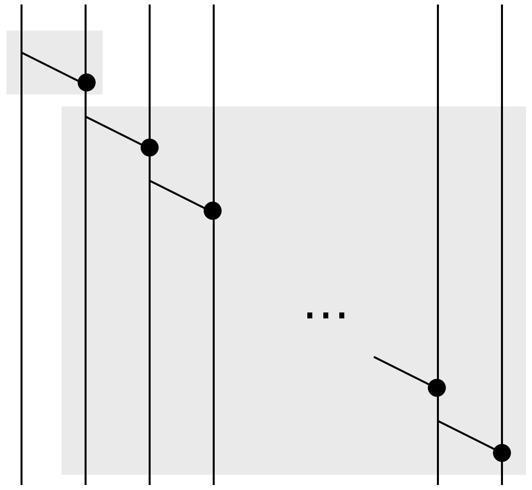
Back to the Serial Prefix



Combinator Lava:

```
serial op =
   overlap 2
        extend
        (serial op)
```

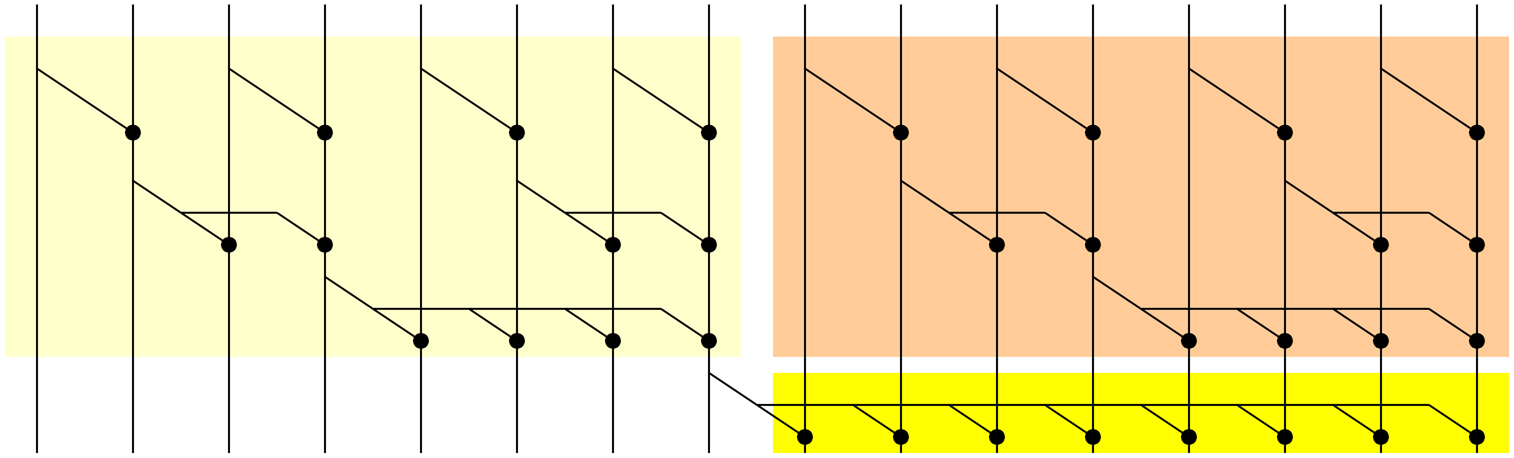
Back to the Serial Prefix



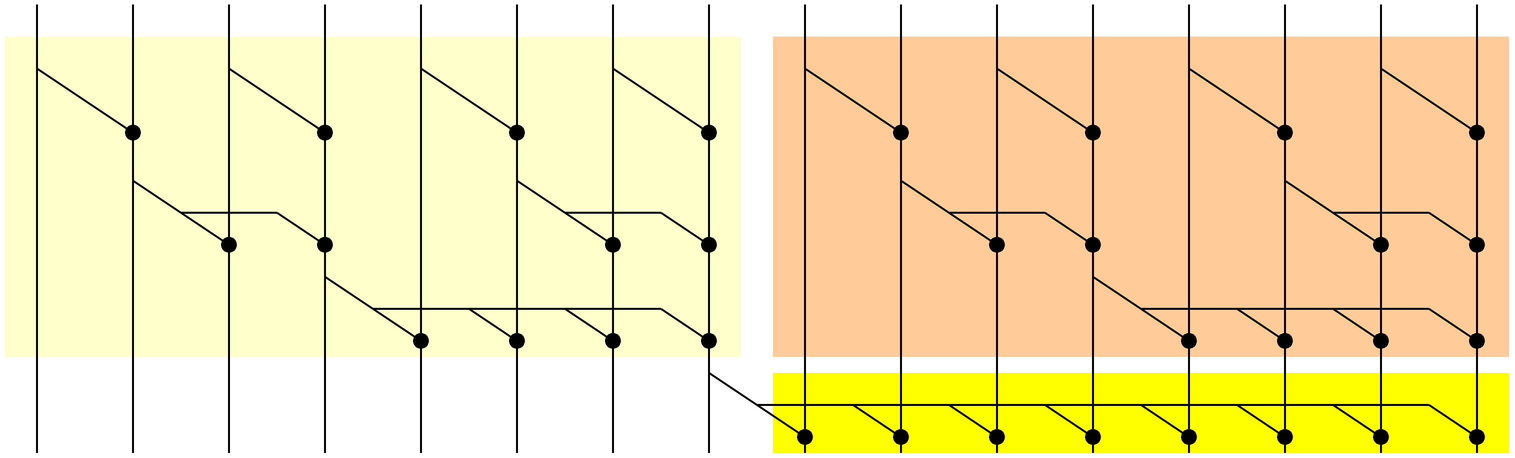
Combinator Lava:

```
serial op 1 = id
serial op (k+1) =
   overlap 2
        extend
        (serial op k)
```

Back to Sklansky Prefixes



Back to Sklansky Prefixes



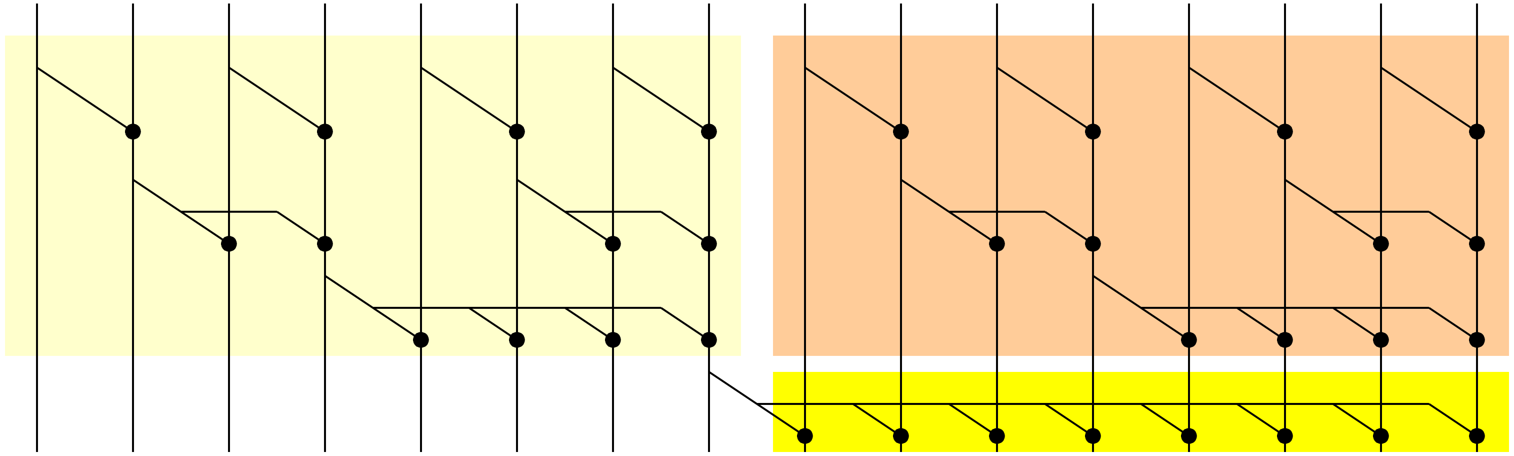
sklansky op = overlap2 left right

where

left = sklansky op

right = extend op (sklansky op)

Back to Sklansky Prefixes

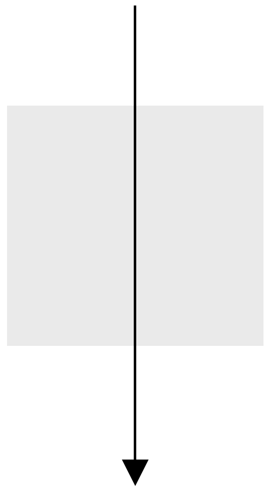


```
sklansky op 1 = id

sklansky op k = overlap2 left right

        where

                half_k = k `div` 2

                left = sklansky op half_k

                right = extend op (sklansky op
half_k)
```

And Now in Wired

Some basic combinator:

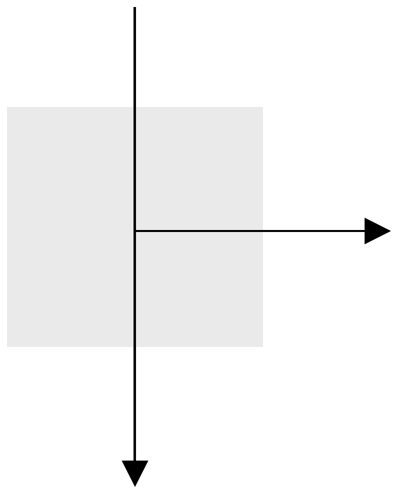
idWire =



And Now in Wired

Some basic combinator:

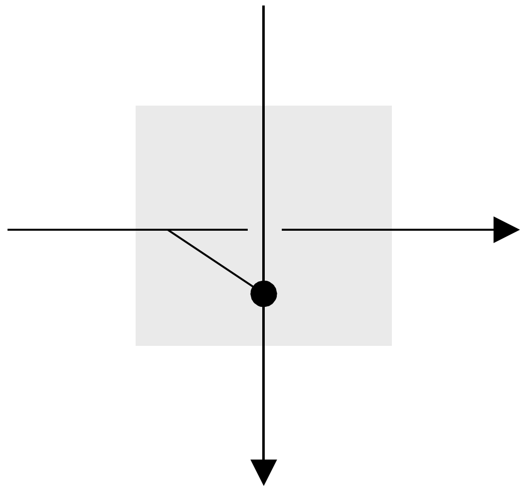
idFork =



And Now in Wired

Some basic combinator:

copy op (x,y) = (op (x,y), x)



And Now in Wired

And layout combinators:

row c = multiple copies of c in a row

c1 *||~ c2 = put c1 (consuming one wire) beside c2 (consuming the rest of the wires).

c1 *=~ c2 = put c1 (consuming one wire) beneath c2 (consuming the rest of the wires).

And Now in Wired

And layout com

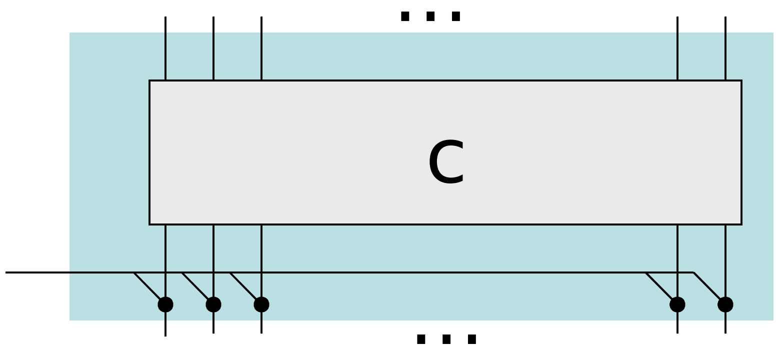
row c = multiple

c1 *||~ c2 = put
wire) beside c2
of the wires).

Similarly ~||*, ~=*,
~||~ and ~=~

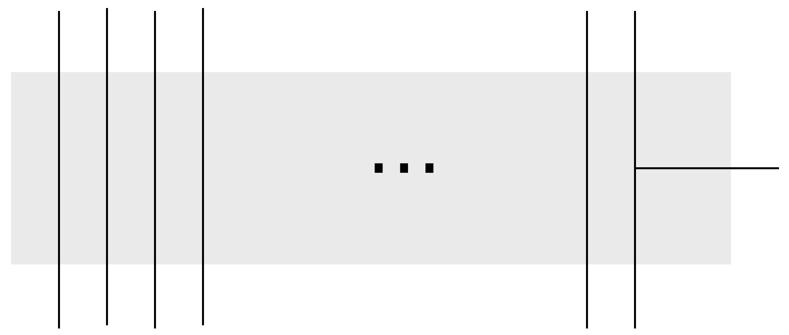
c1 *=~ c2 = put c1 (consuming one
wire) beneath c2 (consuming the rest
of the wires).

Using Wired



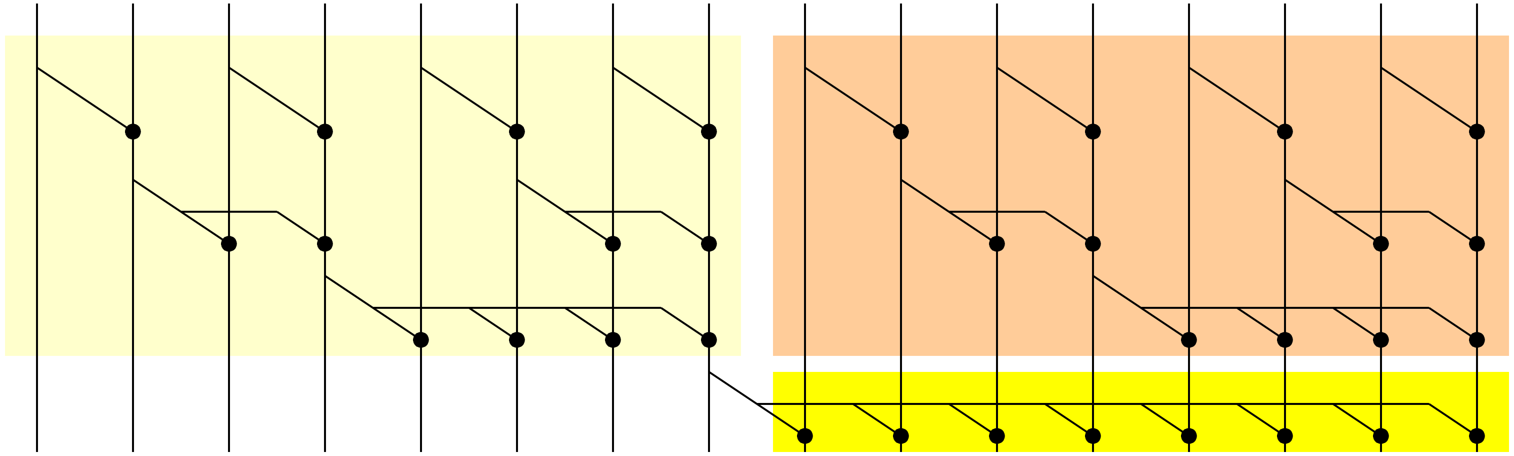
```
extend op c =
      (row (copy op) ~||* op) *=~ c
```

Using Wired



```
forkLast c =
        (row idWire ~||* idFork) *=~ c
```

Back to Sklansky Prefixes



```
sklansky op 1 = idWire

sklansky op k = left ~||~ right

    where

        half_k = k `div` 2

        left = forkLast (sklansky op half_k)

        right = extend op (sklansky op
half_k)
```

Wired Conclusions

- Essentially, Wired is a combinator library to join rectangles together, with pluggable circuits inside.
- The information generated can then used to perform analysis (delay, power consumption, layout, etc).
- Functionality can also be modelled by going down to a Lava-like representation (ignore all shape information).

Hawk

- Hawk is a HDL embedded in Haskell.
- It was primarily aimed at modelling microprocessors and reasoning about them.
- Includes symbolic simulation of microprocessors.
- Algebraic reasoning about microprocessors allows simplifying the models to enable automatic verification.

Concrete Simulation of a Microprocessor

```
data Op  =    MOV Addr Addr
         |    MOVI Addr Data
         |    ADD Addr Addr
         |    SUBI Addr Data
         |    JUMPZ Addr Loc
         |    JUMP Loc
```

Concrete Simulation of a Microprocessor

```
data Op  =    MOV Add
         |    MOVI Add
         |    ADD Addr A
         |    SUBI Addr Data
         |    JUMPZ Addr Loc
         |    JUMP Loc
```

For the moment, Addr, Data and Loc are all integers

Concrete Simulation of a Microprocessor

```
data MachineState =
  ST (     Loc          -- Program counter
     ,     [ Data ]     -- Memory
     ,     Program      -- Actual program
     )
```

Concrete Simulation of a Microprocessor

```
add a b (ST ( loc, mem, code )) =
  ST ( loc+1
     , put a (mem `at` a + mem `at` b)
  mem
     , code
     )

execute (ADD a b) s = add a b s
```

Concrete Simulation of a Microprocessor

```
subi a b (ST ( loc, mem, code )) =
  ST ( loc+1
      , put a (mem `at` a - b) mem
      , code
      )

execute (SUBI a b) s = subi a b s
```

Concrete Simulation of a Microprocessor

```
jumpz a b (ST ( loc, mem, code )) =
  if’ ( mem `at` a === 0)
      ( ST ( b, mem, code)
      , ST ( loc+1, mem, code)
      )

execute (JUMPZ a b) s = jumpz a b s
```

Concrete Simulation of a Microprocessor

```
run (ST (loc, mem, prg) )
  | loc >= length prg   = ST (loc, mem,
  prg)
  | otherwise        =
                run (execute (prg `at` loc) s)
```

Concrete Simulation of a Microprocessor

```
Main>
   run (ST (0, [1,2,3],
       [ MOV 0 2
        , SUBI  0 1
        , ADD 2 2
        , JUMPZ 0 1
        ]
   )

ST (4, [0,16, 3], [...])
```

Symbolic Simulation of a Microprocessor

```
data Symbolic
        =    Const Int
        |    Var String
        |    Plus Symbolic Symbolic
        |    Minus Symbolic Symbolic
        |    Times Symbolic Symbolic
```

Symbolic Simulation of a Microprocessor

```
instance Num Symbolic where
  Const x + Const y     = Const (x+y)
  Const 0 +  x          = x
  x + Const 0           = x
  x + y                 = Plus x y


  ...
```

Symbolic Simulation of a Microprocessor

```
instance Num Symbolic where
  Const x + Const y     = Const (x+y)
  Const 0 +  x          = x
  x + Const 0           = x
  x + y            = Plus x y


  …
```

Symbolic Simulation of a Microprocessor

```
instance Num Symbolic where
  Const x + Const y      = Const (x+y)
  Const 0 +  x           = x
  x + Const 0            = x
  x + y
  ...
```

By using Symbolic for Data (possibly, with additional machinery, even the other types), we can symbolically simulate the behaviour of a program

Concrete Simulation of a Microprocessor

```
Main>
   run (ST (0, [1,Var “x”,3],
       [ MOV 0 2
       , SUBI  0 1
       , ADD 2 2
       , JUMPZ 0 1
       ]
   )
```

ST (4, [0, 8x, 3], [...])

(with appropriate simplification of symbolic expressions)

Hawk Conclusions

- All this is actually done using typeclasses, hence allowing the user to choose between concrete and symbolic simulation.
- Hawk uses these techniques over a stream based HDL similar to Lava.
- By manipulating and simulating the circuits symbolically, the user can algebraically modify circuits whilst still ensuring correct behaviour.

reFLect

- reFLect is a strongly typed, functional *reflective* (or *meta*-) language.
  - Programs can be considered to be data objects in the language itself.
  - Does this using *quotation* and *anti-quotation* operators.
  - Allows pattern matching on quoted programs.
- Has been used to embed a Lava-like HDL.

reFLective Code

- Code is quoted using ⟨⟨ - ⟩⟩

  Quoted code is considered to denote the abstract syntax tree ie ⟨⟨ not True ⟩⟩ is not equivalent to ⟨⟨ False ⟩⟩.

- Code is unquoted using the ^- operator

  Unquotation evaluates quoted code, for example, ^⟨⟨ not True ⟩⟩ is equivalent to False.

- Pattern matching can be combined with quote-anti-quote operators:

  ⟨⟨ not (^x) ⟩⟩ pattern matches with ⟨⟨ not True ⟩⟩, with x matching the value of ⟨⟨ True ⟩⟩.

Deep vs Shallow Embedding

- We have already seen what an embedding of a language in another is.
- A deep embedding is when the syntax of the embedded language is encoded as data, or in a manner that the host language has direct access to it.
- A shallow embedding is one in which embedded language programs are constructed using functions. Inspection of the programs is impossible.

Deep vs Shallow Embedding

- We have already seen what an embedding of a language in another is.
- A deep embedding is when the syntax of the embedded langua is encoded as data, or in a mann language has direc
- A shallow embedd embedded langua constructed using the programs is im

The examples we've seen are all of deep embeddings of HDLs, since the circuits are just data objects which one can not only construct but also manipulate.

What would a shallow embedding of an HDL look like?

Shallow Embedding of a Stream Language using Lazy Lists

```
type Stream a = [a]

stream `at` time = stream !! time
```

Shallow Embedding of a Stream Language using Lazy Lists

```
low :: Stream Bool
low = repeat False
```

Shallow Embedding of a Stream Language using Lazy Lists

```
inv :: Stream Bool -> Stream Bool
inv xs = [ not x | x <- xs ]
```

Shallow Embedding of a Stream Language using Lazy Lists

```
inv :: Stream Bool -> Stream Bool
inv xs = [ not x | x <- xs ]


and2 :: (Stream Bool, Stream Bool) ->
                                         Stream
  Bool
and2 (xs, ys) = [ x && y | (x,y) <- zip xs
  ys ]
```

Shallow Embedding of a Stream Language using Lazy Lists

The stream *(delay x xs)* returns *x* in the first time unit, then the elements of *xs* in order.

```
delay :: a -> Stream a -> Stream a
delay x xs = x:xs
```

Shallow Embedding of a Stream Language using Lazy Lists

```
or2 (xs,ys) = inv (and2 (inv xs, inv
ys))


mux (sel, (xs, ys)) =
    or2 ( and2(sel, ys), and2(inv sel,
xs) )


always xs = outs
    where
          outs = and2 (xs, delay True
```

Shallow Embedding of a Stream Language using Lazy Lists

```
Main> mux [ (False, (True, False)), (True, (True, False))]
[True, False]

Main> always [True, True, False, True]
[True, True, False, False]
```

Shallow Embedding of a Stream Language using Lazy Lists

Question:

This seems to be so easy. Why aren't the FHDLs we've seen implemented this way?

Shallow Embedding of a Stream Language using Lazy Lists

Question:

This seems to be so easy. Why aren't the FHDLs we've seen implemented this way?

Answer:

We can only simulate these circuits. Try, for instance, writing a function to count the number of gates. It is impossible, since we have no access to the gates as data objects.

Shallow Embedding of a Stream Language using Lazy

Disclaimer: This is only partially true, since we can have different interpretations of the various gates depending on what we want to do, and use the different interpretations (via typeclasses) to perform what we want.

Question:

This seems to t the FHDLs we' this way?

Answer:

We can only simulate these circuits. Try, for instance, writing a function to count the number of gates. It is impossible, since we have no access to the gates as data objects.

A Deep and Shallow Embedding

- Clearly, a deep embedding is needed for most applications;
- But shallow embeddings are more straightforward to build.

- **Solution (?):** reflective languages allow us to talk about code as data objects – a shallow embedding *is* a deep embedding!

A Deep+Shallow Embedding in reFLect

```
lowC = repeat False
invC xs = map not ^xs
and2C (xs, ys) = [ x && y | (x,y) <- zip ^xs
   ^ys ]
delayC x xs = x: ^xs

low = ⟨⟨ lowC ⟩⟩
inv xs = ⟨⟨ invC xs ⟩⟩
and2 (xs, ys) = ⟨⟨ and2C (xs, ys) ⟩⟩
delay x xs = ⟨⟨ delayC x xs ⟩⟩
```

A Deep+Shallow Embedding in reFLect

lowC = repeat False

inv

ar s

de

lo

inv xs = ⟨⟨ invC xs ⟩⟩

and2 (xs, ys) = ⟨⟨ and2C (xs, ys) ⟩⟩

delay x xs = ⟨⟨ delayC x xs ⟩⟩

For example:

and2 (low, inv low)

=

⟨⟨ and2C (⟨⟨ lowC ⟩⟩, ⟨⟨ invC ⟨⟨ lowC ⟩⟩ ⟩⟩) ⟩⟩

reFLect: Conclusions

- Using a meta-language for embedded languages has the advantage of giving a cheap deep embedding.
- The use of such languages for functional HDLs is a new field of research, and is still very much under investigation.
- *reFLect* is used in the Forte hardware verification environment, combining model checking, decision algorithms and theorem proving.

Conclusions

- Clearly there is no ***one*** way of embedding a functional HDL.
- Various other languages exist: Hydra, SAFL, Lucid Synchrone, etc.
- The common main advantages are:
  - Access to a meta-language for the HDL;
  - Strong abstraction techniques allow concise descriptions of regular circuits;
  - We can manipulate generated circuits.

Final Exam (1)

Question 1:

Implement an n-bit multiplier in Lava (using a design of your choice). Use SMV and Lava to specify and verify that multiplying a 4-bit number by an even number always gives an even number.

Question 2:

Design a four-bit accumulator which has two inputs *update* (one bit) and *value* (four-bits), and one output *sum* (four bits). The output starts off at 0, and is incremented by *value* whenever *update* is true.

Using the accumulator, implement a 4-bit counter, which starts off outputting zero, and increments its output with every clock tick (resetting to zero when it overflows).

Final Exam (2)

Question 3:

Modify the compiled imperative language with emit given in part 4 of the course to also count the number of emits (using a 4-bit unsigned integer) happening at that instant of time.

egat the initial moment, (emit; emit || emit)

should output 3.

Final Exam (3)

Question 3 (continued):

- Add also an accumulator, which counts the total number of emits sent by the program.
- Finally, make programs terminate whenever the program has output more than 10 emit signals (ignore overflows)   .

Final Exam (4)

Question 4:

Explain and implement in Lava another parallel prefix network circuit implementation (different than the ones given in the slides). You may find the following link useful:

http://www.stanford.edu/class/ee371/handouts/ha

Verify that with an **and** gate, your four input prefix network is equivalent to the naïve one, and the Sklansky one.