Lava I

Mary Sheeran, Thomas Hallgren
Chalmers University of Technology

Where are we?

- Take a look at the schedule
- First half of the course: Industry standard languages and tools
  - VHDL
  - PSL
  - Jasper Gold

- Also LTL, CTL, Model Checking algs, SAT based verification

Second half of the course: exploring alternatives

- Hardware designs are becoming more and more complex
- Need higher level languages with better abstractions, easier re-use
- There is a need to control low-level details even at high levels of design
- Better languages are needed

Better Hardware Description Languages?

- Remember course synopsis?
  - *Getting hardware designs right using ideas from computer science*
- Idea: transfer progress in programming languages to Hardware Description Languages
- From the Computer Science department at Chalmers:
  - **Strong Functional Programming group**, particular expertise in Haskell (involved in the design), also interested in automated verification methods inc. SAT-based verif., we like to make tools   => Lava

Hardware Description in Functional Languages

- Advantages of Functional Languages:
  - Provide a concise notation
  - Powerful abstraction mechanisms to deal with complexity
  - Good support for generic hardware descriptions
  - Suitable for making embedded Domain Specific Languages (this is Haskell's forte)

Hardware Description in Functional Languages

Examples (see links page for more info)

1) Warren Hunt's use of ACL2 in processor verification at Centaur
2) Intel's Forte system (the mainstay of their formal verification programme)
3) Intel's IDV system (Integrating Design and Verification)
4) Hawk (cool work at OGI on processor desc. and verif.)
5) Cryptol (used at Galois Inc for crypto, inc FPGA gen.)

Hardware Description in Functional Languages

Examples

6) Lava (variants: Chalmers, Xilinx, York, Kansas)

7) using Haskell directly as a hardware description lang.

8) Bluespec

…. and more not mentioned

Hardware Description in Functional Languages

Examples

6) Lava (variants: Chalmers, Xilinx, York, Kansas)

7) using F

8) Bluespec

We will use Chalmers Lava even though it is old and a little tired. It suits our purposes and our interest in verification

Xilinx Lava (Singh) gives fine control over layout on FPGA

York Lava allows one to work at a higher level of abstraction, used in processor design

Kansas Lava also targets FPGAs and aims to be a modern reimplementation of our Lava

Hardware Description in Functional Languages

Examples

6) Lava (variants: Chalmers, Xilinx, York, Kansas)

7) using Haskell directly as a hardware description lang.

8) Bluespec

Guest lecture by Satnam Singh, wed. 11th May

Hardware Description in Functional Languages

Examples

6) Lava (variants: Chalmers, Xilinx, York, Kansas)

7) using Haskell directly as a hardware description lang.

8) Bluespec

…. and more not mentioned

Guest lecture by Lennart Augustsson, wed. 18th May

What is Lava?

- **Lava is a hardware description language embedded in Haskell**
- Haskell is a purely functional programming language.
- Like VHDL, Haskell is a strongly typed language.
- A compiler (GHC) and an associated interactive system (ghci) are available.
- Everything about Haskell: www.haskell.org.
- See also the Links page for pointers to intro. material

What is Lava?

- **Lava is essentially a Haskell library from which you can import types and functions for**
- describing circuits,
- simulating circuits,
- feeding circuits to other tools, e.g. for formal verification

What is Lava?

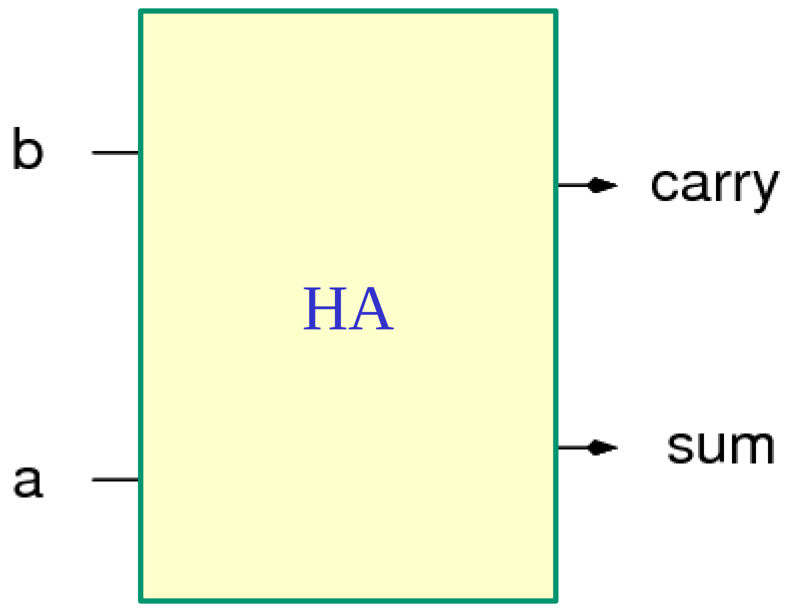
- **Lava is essen Haskell library from which y tions for**
- describi
- simulati
- feeding c[illegible] for formal verification

Or to put it another way:
It’s a tool to allow control freaks to generate netlists

Lava Documentation

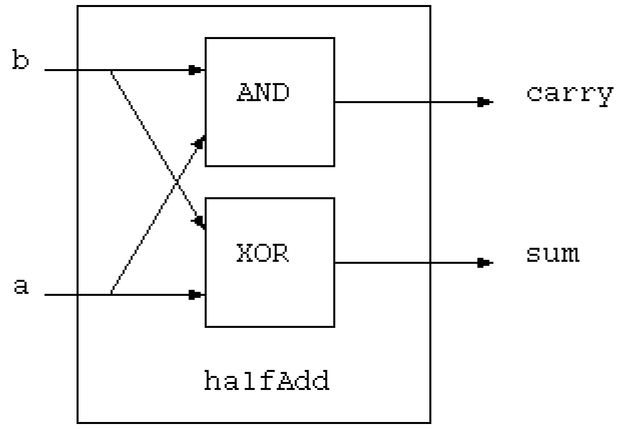
- The Lava Tutorial introduces Lava without requiring previous knowledge of Haskell.
- There is also the guide How to Use the Lava System.
- Instructions for accessing the tools
- Please get back to me or Emil if you have problems getting started with Lava

First example



a	b	carry	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Half Adder implementation



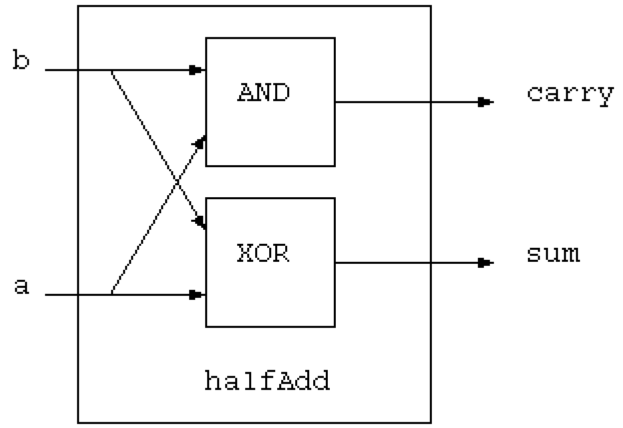
Half Adder in VHDL

```
entity halfAdder is
    port (a,b : in bit; sum,carry : out bit);
end halfAdder;

architecture ha_beh of halfAdder is
begin
   sum <= a xor b;
   c_out <= a and b;
end ha_beh;
```

- to have something to compare to

Half Adder in Lava

```
halfAdder (a, b) = (sum, carry)
   where
       sum   = xor2 (a, b)
       carry = and2 (a, b)
```



Note: it's a direct transcription of the circuit diagram!

Running the examples

Download the file LavaIntro.hs

In that directory, type `ghci` at the prompt., and then at the ghci prompt

```
:l LavaIntro.hs
```

Running the examples

Download the file LavaIntro.hs

In that directory, type `ghci` at the prompt., and then at the ghci prompt

```
:l LavaIntro.hs
```

You get

```
[1 of 1] Compiling Main             ( LavaIntro.hs, interpreted )
Ok, modules loaded: Main.
*Main>
```

Running the examples

Download the file LavaIntro.hs

In that directory, type `ghci` at the prompt., and then at the ghci prompt

```
:l LavaIntro.hs
```

You get

```
[1 of 1] Compiling Main          ( LavaIntro.hs, interpreted )
Ok, modules loaded: Main.
*Main>
```

and now you are all set and can ask questions like:

```
*Main> :t halfAdder
```

Running the examples

Download the file LavaIntro.hs

In that directory, type ghci at the prompt., and then at the ghci prompt

```
:l LavaIntro.hs
```

You get

```
[1 of 1] Compiling Main          ( LavaIntro.hs, interpreted )
Ok, modules loaded: Main.
*Main>
```

and now you are all set and can ask questions like:

```
*Main> :t halfAdder
```

What is the type of halfAdder?

Running the examples

Download the file LavaIntro.hs

In that directory, type ghci at the prompt., and then at the ghci prompt

```
  :l LavaIntro.hs
```

You get

```
[1 of 1] Compiling Main          ( LavaIntro.hs, interpreted )
Ok, modules loaded: Main.
*Main>
```

and now you are all set and can ask questions like:

```
*Main> :t halfAdder
halfAdder
  :: (Signal Bool, Signal Bool) -> (Signal Bool, Signal Bool)
```

Half Adder Interface

halfAdder **::** (**Signal Bool**,**Signal Bool**) -> (**Signal Bool**,**Signal Bool**)

halfAdder (a,b) = (sum,carry)

where …

- The first line is the type signature for halfAdder.
- A circuit is represented as a **function** from input to output

  A->*B*: a function with input of type *A* and output of type *B*

  (A_1,A_2): a pair. Pairing allows several signals to be grouped together and treated as one signal.

  Signal *A*: signals carrying values of type *A*

  Bool: boolean values, False or True

Half Adder Interface

Introducing a shorter name for the type of boolean signals

```
type Bit = Signal Bool
```

We can now write

```
halfAdder :: (Bit,Bit) -> (Bit,Bit)
halfAdder (a,b) = (sum,carry)
  where ...
```

Simulating Lava circuits

- Simulating a single cycle   (at the ghci prompt)

  simulate *circuit input*

- Example:

```
simulate halfAdder (low,high)
 (high,low)
simulate halfAdder (high,high)
 (low,high)
```

- More later

Logical Gates in the Lava library

From Appendix A (Quick Reference) in the Lava Tutorial:

```
id, inv :: Bit -> Bit
and2, nand2, or2, nor2, xor2, equiv, impl :: (Bit,Bit) ->Bit
<&>, <|>, <#>, <=>, ==> :: (Bit,Bit) -> Bit
```

a <&> b   is the same as   and2 (a,b) etc.

Signals can also carry Int values (more later)

Half Adder in Lava, other possible versions

```
halfAdder (a, b) = (sum, carry)
   where
       sum   = xor2 (a, b)
       carry = and2 (a, b)
```

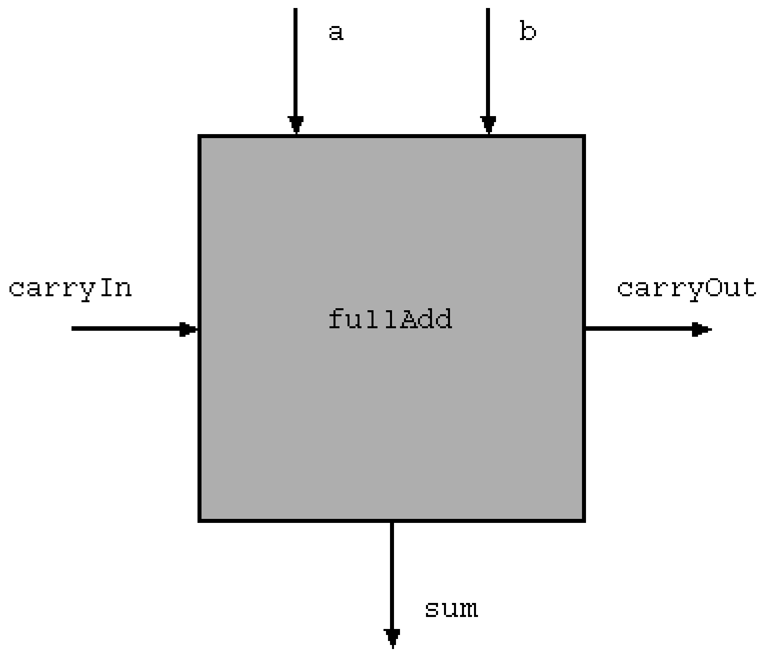
• In functional languages, you can substitute equals for equals:

```
halfAdder (a, b) = (xor2 (a,b),and2(a,b))
```

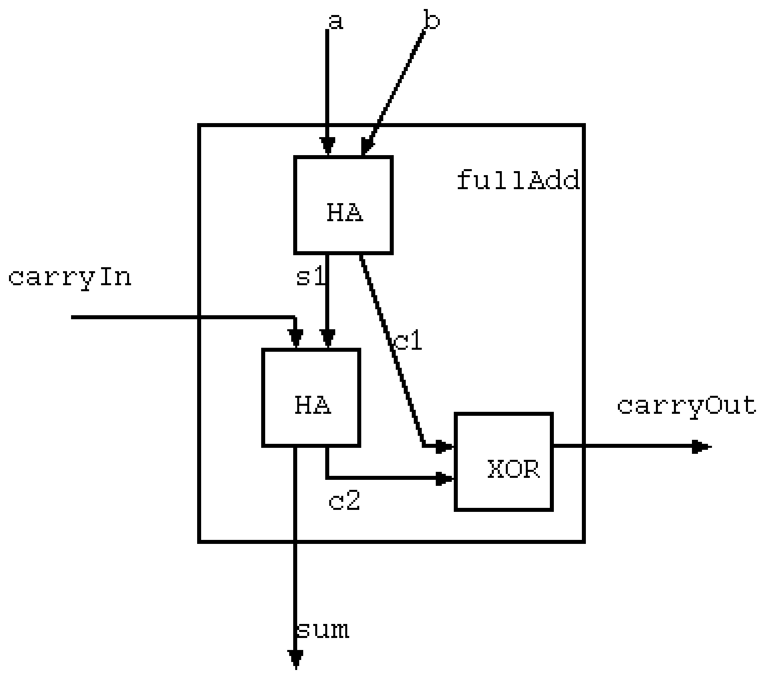
•Using the alternative infix operators:

```
halfAdddder (a, b) = (a <#> b, a <&> b)
```

Second Example: a Full Adder



Full Adder implementation



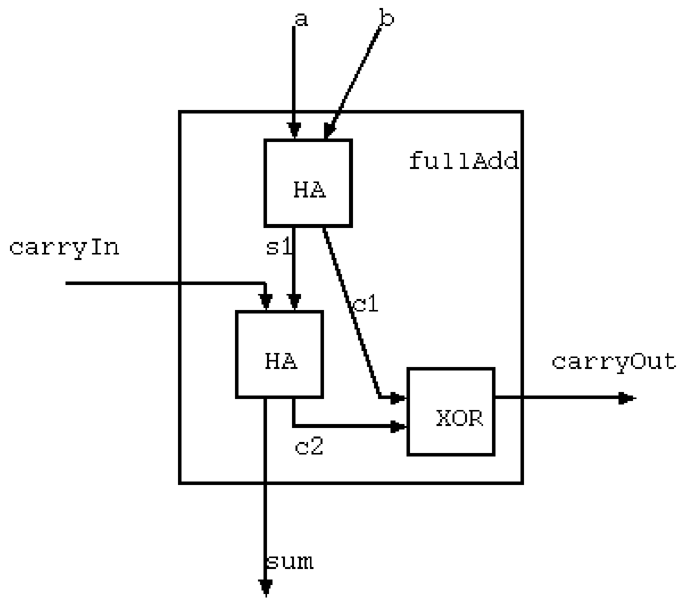
Full Adder in VHDL

```
entity fullAdder is
  port (a,b,carryIn : in bit; sum,carryOut : out bit);
end fullAdder;

architecture fa_beh of fullAdder is
   signal s1,c1,c2 : bit;
begin -- fa_beh
     ha1: entity work.halfAdder
          port map (a, b, s1, c1);
     ha2: entity work.halfAdder
          port map (carryIn, s1, sum, c2);
     xor1: carryOut <= c1 xor c2;
end fa_beh;
```

A structural description that refers to the previously defined entity halfAdder.

Full adder in Lava

```
fullAdder (carryIn, (a,b)) = (sum, carryOut)
   where
        (s1, c1)     = halfAdder (a, b)
        (sum, c2)  = halfAdder (carryIn, s1)
        carryOut   = xor2 (c2, c1)
```



Again, it should be a direct transcription of the circuit diagram.
Using previously defined components is just as easy as using basic gates.

Full Adder Interface

```
fullAdder :: (Bit,(Bit,Bit)) -> (Bit,Bit)
fullAdder (carryIn, (a,b)) = (sum, carryOut)
  where ...
```

The first line is the type signature of function fullAdder.

It is inferred automatically if you leave it out.

Another Full Adder

```
fa :: (Bit,(Bit,Bit)) -> (Bit,Bit)
fa (cin, (a,b)) = (sum, cout)
  where
      part_sum = xor2 (a, b)
      sum      = xor2 (part_sum, cin)
      cout     = mux (part_sum, (a, cin))
```

Another Full Adder

```
fa :: (Bit,(Bit,Bit)) -> (Bit,Bit)
fa (cin, (a,b)) = (sum, cout)
   where
         part_sum = xor2 (a, b)
         sum      = xor2 (part_sum, cin)
         cout     = mux (part_sum, (a, cin))
```

This is a multiplexer

mux (a, (l,h)) chooses l if a low and h if a high

It is generic, so l and h can have any suitable type (e.g. pairs or lists of bits)

Is it a correct Full Adder?

Check by exhaustive simulation

```
*Main> simulate fa (low,(low,low))
(low,low)
*Main> simulate fa (low,(low,high))
(high,low)
etc.
```

Can also define and name tests in the file itself and then run them at the prompt

```
tst1 = simulate fa (low,(low,low))
*Main> tst1
(low,low)
```

Is it a correct Full Adder?

Check by exhaustive simulation

```
> simulate fa (low,(low,low))
(low,low)
> simulate fa (low,(low,high))
(high,low)
```

This is single cycle simulation

etc.

Can also define a

them at the prompt

```
tst1 = simulate fa (low,(low,low))
 > tst1
(low,low)
```

Is it a correct Full Adder?

simulateSeq *circuit list_of _inputs*

simulates a sequence of cycles

useful for exhaustive testing of combinational ccts

```
tst2 = simulateSeq fa [(low,(low,low)), (low,(low,high)), (low,(high,low))]
> tst2
[(low,low),(high,low),(high,low)]
```

Is it a correct Full Adder?

simulateSeq *circuit list_of _inputs*

simulates a sequence of cycles

useful for exhaustive testing of combinational ccts

```
tst2 = simulateSeq fa [(low,(low,low)), (low,(low,high)), (low,(high,low))]
> tst2
[(low,low),(high,low),(high,low)]
```

```
tst2 = simulateSeq fa domain
> tst3
[(low,low),(high,low),(high,low),(low,high),(high,low),(low,high),(low,high),
  (high,high)]
```

Is it a correct Full Adder?

simulateSeq *circuit list_of _inputs*

simulates a sequence of cycles

useful for exhaustive testing of combinational ccts

```
tst2 = simulateSeq fa [(low,(low,l
> tst2
[(low,low),(high,low),(high,low)]
```

Useful function that gives all values of a particular input shape

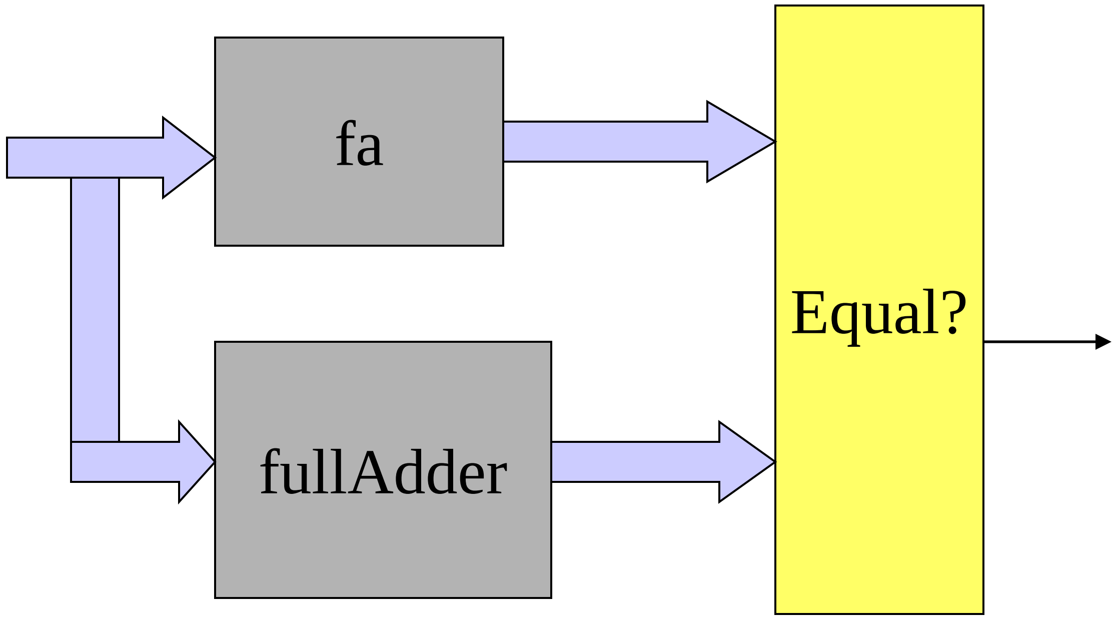
```
tst2 = simulateSeq fa domain
> tst3
[(low,low),(high,low),(high,low),(low,high),(high,low),(low,high),(low,high),
   (high,high)]
```

Is it a correct Full Adder?

Previous approach not completely satisfactory

More convincing to compare with a golden model (say our fullAdder)

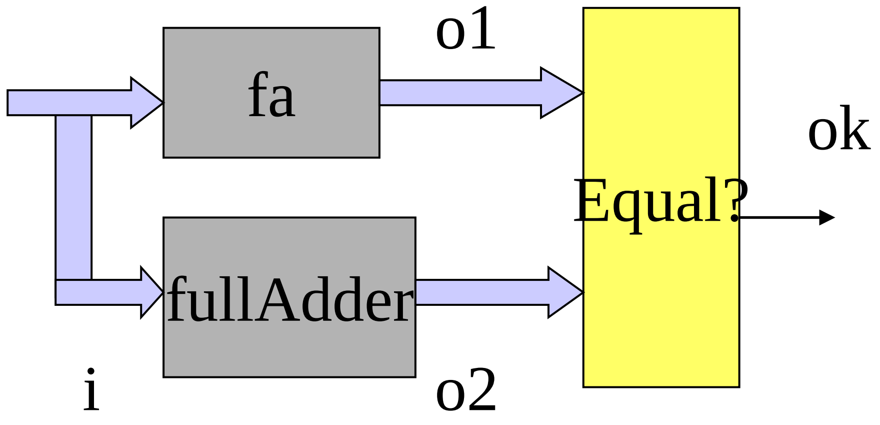
Equivalence Checking (simulation)



Give all possible inputs in sequence and check output is always high

Describing this circuit

```
prop_fa :: (Bit,(Bit,Bit)) -> Bit
prop_fa i = ok
  where
    o1 = fa i
    o2 = fullAdder i
    ok = o1 <==> o2
```



```
test_fa = simulateSeq  prop_fa  domain

> test_fa
[high,high,high,high,high,high,high,high]
```

Are we happy?

Formal verification of equivalence

```
> smv prop_fa
Smv: ... (t=0.00system) \c
Valid.
Valid
```

Formal verification of equivalence

```
> smv prop_fa
Smv: ... (t=0.00system) \c
Valid.
Valid
```

What happened??

SMV input file generated and fed to SMV (a CTL MC)

```
-- Generated by Lava

MODULE main
VAR i0 : boolean;
VAR i1 : boolean;
VAR i2 : boolean;
DEFINE w6 := i0;
DEFINE w7 := i1;
DEFINE w5 := !(w6 <-> w7);
DEFINE w8 := i2;
DEFINE w4 := !(w5 <-> w8);
DEFINE w10 := !(w6 <-> w7);
DEFINE w9 := !(w8 <-> w10);
DEFINE w3 := !(w4 <-> w9);
DEFINE w2 := !(w3);
DEFINE w15 := w5 & w8;
DEFINE w17 := !(w5);
DEFINE w16 := w17 & w6;
DEFINE w14 := w15 | w16;
DEFINE w19 := w8 & w10;
DEFINE w20 := w6 & w7;
DEFINE w18 := !(w19 <-> w20);
DEFINE w13 := !(w14 <-> w18);
DEFINE w12 := !(w13);
DEFINE w21 := 1;
DEFINE w11 := w12 & w21;
DEFINE w1 := w2 & w11;
SPEC AG w1
```

```
-- Generated by Lava

MODULE main
VAR i0 : boolean;
VAR i1 : boolean;
VAR i2 : boolean;
DEFINE w6 := i0;
DEFINE w7 := i1;
DEFINE w5 := !(w6 <-> w7);
DEFINE w8 := i2;
DEFINE w4 := !(w5 <-> w8);
DEFINE w10 := !(w6 <-> w7);
DEFINE w9 := !(w8 <-> w10);
DEFINE w3 := !(w4 <-> w9);
DEFINE w2 := !(w3);
DEFINE w15 := w5 & w8;
DEFINE w17 := !(w5);
DEFINE w16 := w17 & w6;
DEFINE w14 := w15 | w16;
DEFINE w19 := w8 & w10;
DEFINE w20 := w6 & w7;
DEFINE w18 := !(w19 <-> w20)
DEFINE w13 := !(w14 <-> w18)
DEFINE w12 := !(w13);
DEFINE w21 := 1;
DEFINE w11 := w12 & w[illegible]
DEFINE w1 := w2 & [illegible]1;
SPEC AG w1
```

Generated by Lava

SPEC AG w1

SAT solver also

```
> satzoo prop_fa
Satzoo: ...
real    0m0.006s
user    0m0.000s
sys     0m0.000s
(t=) \c
Valid.
Valid
```

in Verify/circuit.cnf
(conj. normal form)

```
c Generated by Lava
c
c i0 : 6
c i1 : 7
c i2 : 8
p cnf 21 57
-5 6 7 0
-5 -6 -7 0
5 -6 7 0
5 -7 6 0
-4 5 8 0
-4 -5 -8 0
4 -5 8 0
4 -8 5 0
-10 6 7 0
-10 -6 -7 0
10 -6 7 0
10 -7 6 0
-9 8 10 0
-9 -8 -10 0
…
9 -8 10 0
9 -10 8 0
-3 4 9 0
-3 -4 -9 0
3 -4 9 0
3 -9 4 0
-3 -2 0
3 2 0
15 -5 -8 0
-1 2 0
-1 11 0
-1 0
```

•

```
> satzoo prop_fa
Satzoo: ...
real    0m0.006s
user    0m0.000s
sys     0m0.000s
(t=) \c
Valid.
Valid
```

in Verify/circuit.cnf
(conj. normal form)

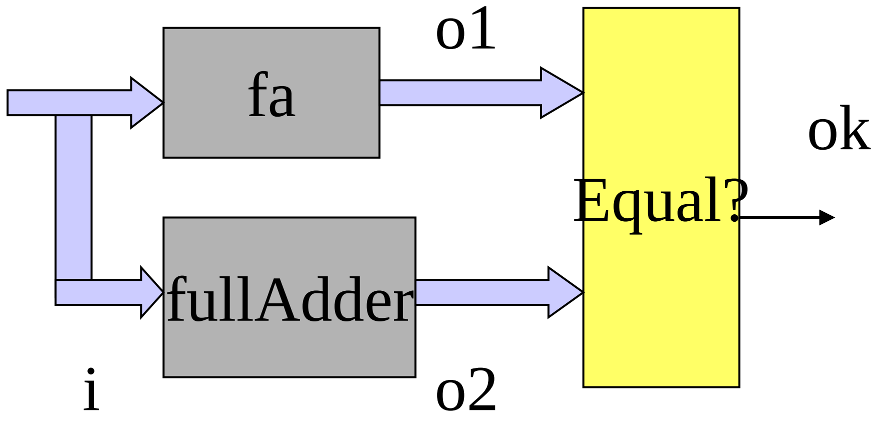
```
c Generated by Lava
c
c i0 : 6
c i1 : 7
c i2 : 8
p cnf 21 57
-5 6 7 0
-5 -6 -7 0
5 -6 7 0
5 -7 6 0
-4 5 8 0
-4 -5 -8 0
4 -5 8 0
4 -8 5 0
10
3 -9 4 0
-3 -2 0
3 2 0
15 -5 -8 0
-1 2 0
-1 11 0
-1 0
```

A SAT solver (predecessor of miniSAT)

Works for combinational circuits

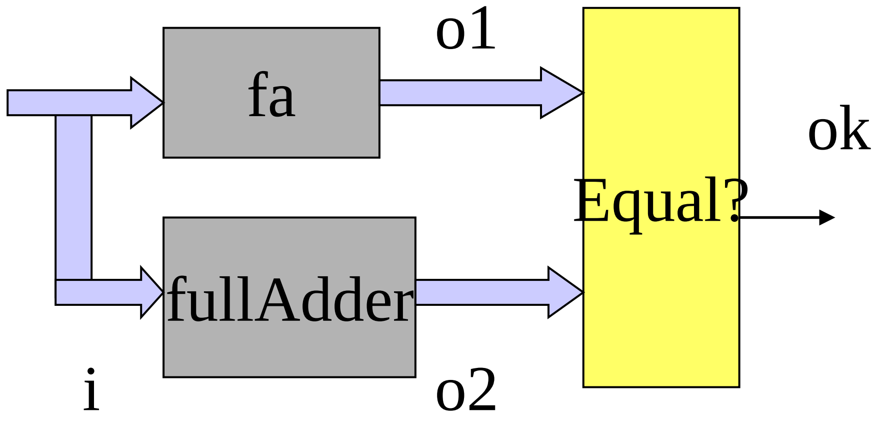
Back to Describing this circuit

```
prop_fa :: (Bit,(Bit,Bit)) -> Bit
prop_fa i = ok
  where
    o1 = fa i
    o2 = fullAdder i
    ok = o1 <==> o2
```



Back to Describing this circuit

```
prop_fa :: (Bit,(Bit,Bit)) -> Bit
prop_fa i = ok
  where
    o1 = fa i
    o2 = fullAdder i
    ok = o1 <==> o2
```



We can generalise this by making the two circuits parameters

Checking equivalence

```
prop_Equivalent  circ1  circ2  inp = ok
  where
    out1 = circ1 inp
    out2 = circ2 inp
    ok    = out1 <==> out2

prop_fa1 = prop_Equivalent  fa  fullAdder
```

Checking equivalence

```
prop_Equivalent  circ1  circ2  inp = ok
  where
    out1 = circ1 inp
    out2 = circ2 inp
    ok    = out1 <==> (

prop_fa1 = prop_Equivalent  fa  fullAdder
```

circ1 and circ2 are inputs
natural in Haskell

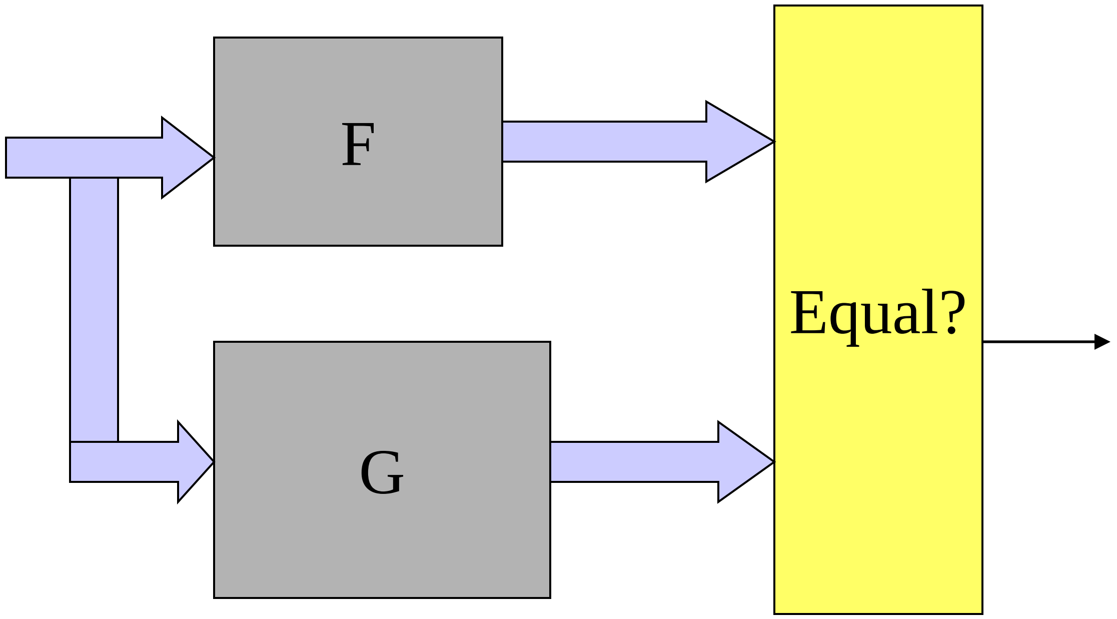
Safety property checking via SMV

smv *property*

- For verifying safety properties, the *property* can a circuit with
  - a number of inputs of fixed size
  - a single boolean output

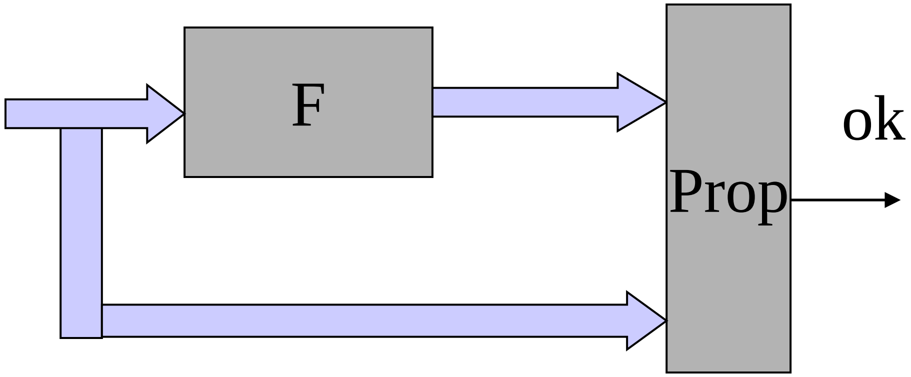
- For verifying generic circuits, a size has to be chosen…

EC fits that shape



Generally, Synchronous Observer

- Only one language (so easier to use)
- Safety properties
- Used in verification of control programs (Lustre, SCADE)

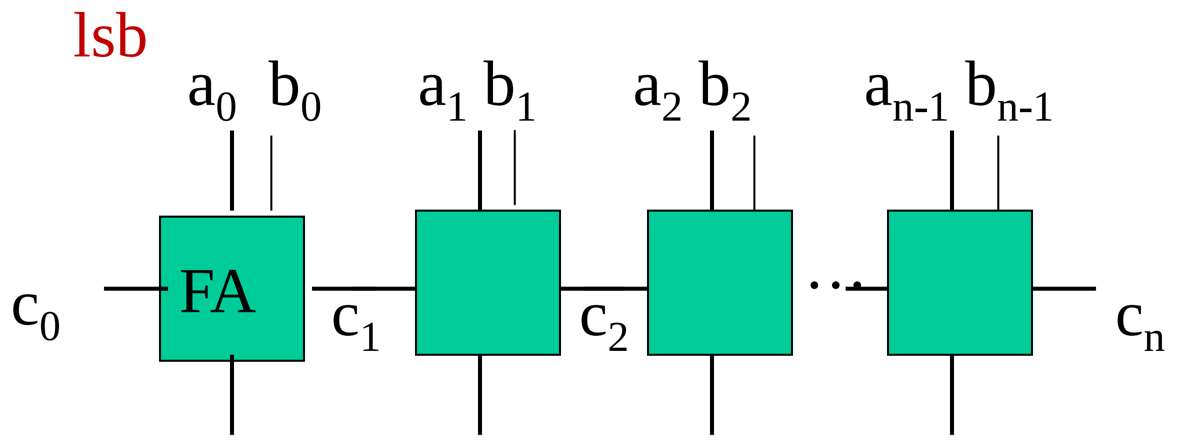


Missing so far

Generic circuit descriptions

Sequential circuits

Ripple Carry Adder (RCA)



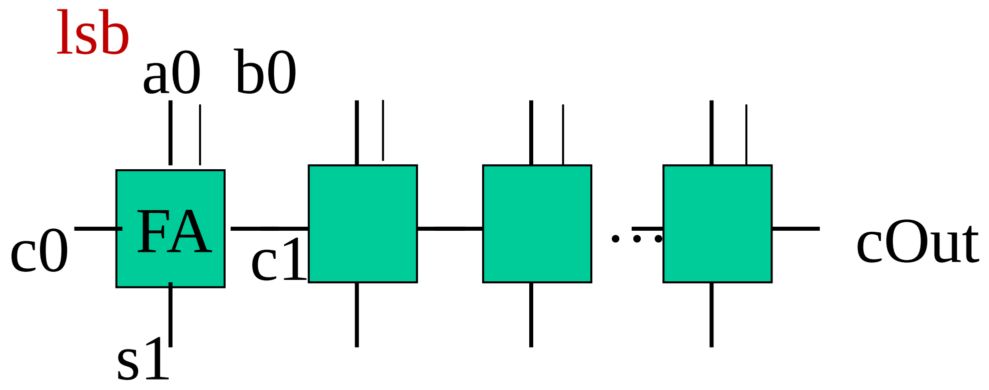
Assume as and bs (lists of bits representing binary numbers) have the same length

RCA in VHDL

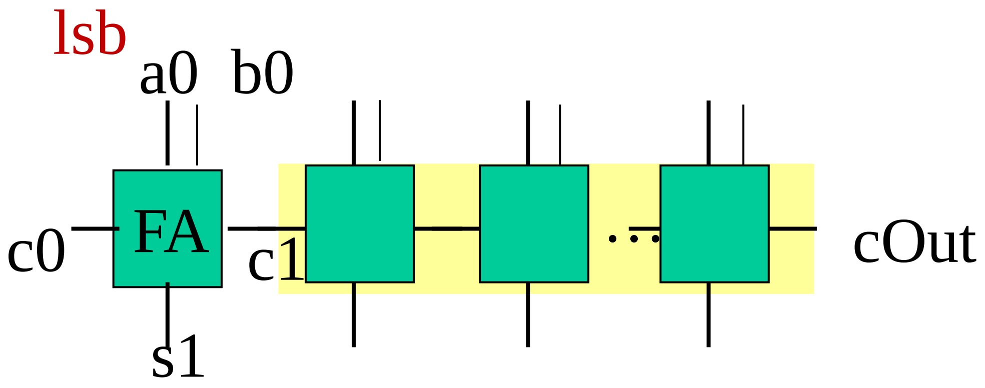
```
entity rippleCarryAdder is
  generic (n : natural);
  port ( carryIn      : in bit;
           a, b         : in bit_vector(n-1 downto 0);
           sum           : out bit_vector(n-1 downto 0);
           carryOut    : out bit);
end rippleCarryAdder;

architecture rca_beh of rippleCarryAdder is
  signal c : bit_vector(0 to n);
begin
  c(0) <= carryIn;

  adders: for i in 0 to n-1 generate
  begin
    bit : entity work.fullAdder
        port map (c(i),a(i),b(i),sum(i),c(i+1));
  end generate;

  carryOut <= c(n);

end rca_beh;
```

A structural description that refers to the previously defined entity fullAdder.

First attempt in Lava using recursion

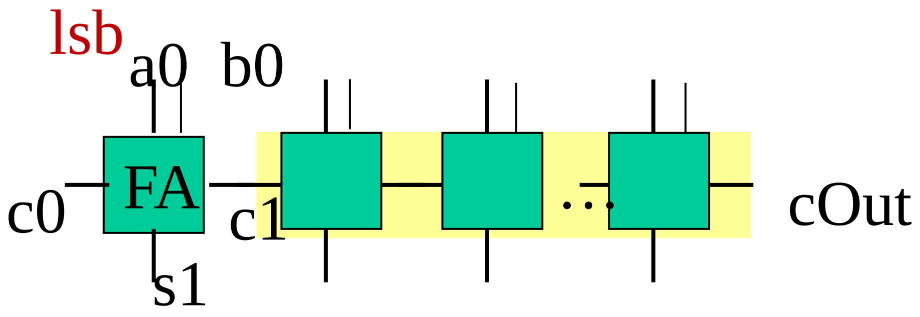


Interface



```
rcAdder0 :: (Bit,([Bit],[Bit])) -> ([Bit],Bit)
```

Interface



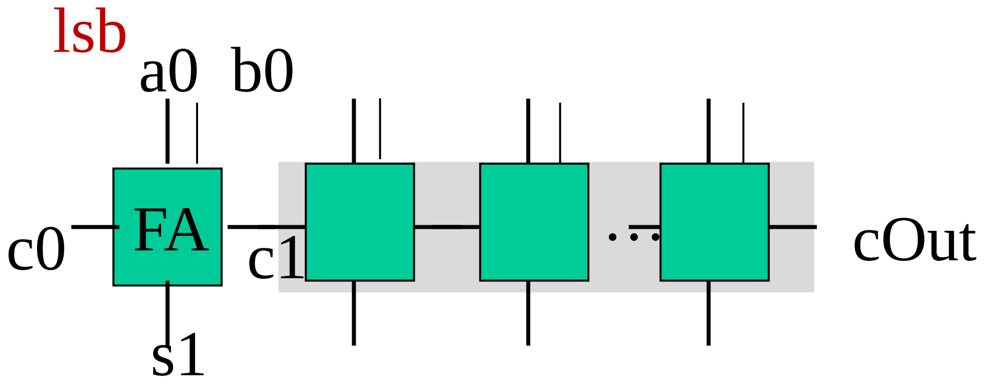
```
rcAdder0 :: (Bit,([Bit],[Bit])) -> ([Bit],Bit)
```

[A]: lists of values of type A. Examples of lists:
[] (empty list)
[low,high,low,low] :: [Bit]
[(low,high),(low,low)] :: [(Bit,Bit)]

List are used both for sequences in time and for parallel signals (busses).

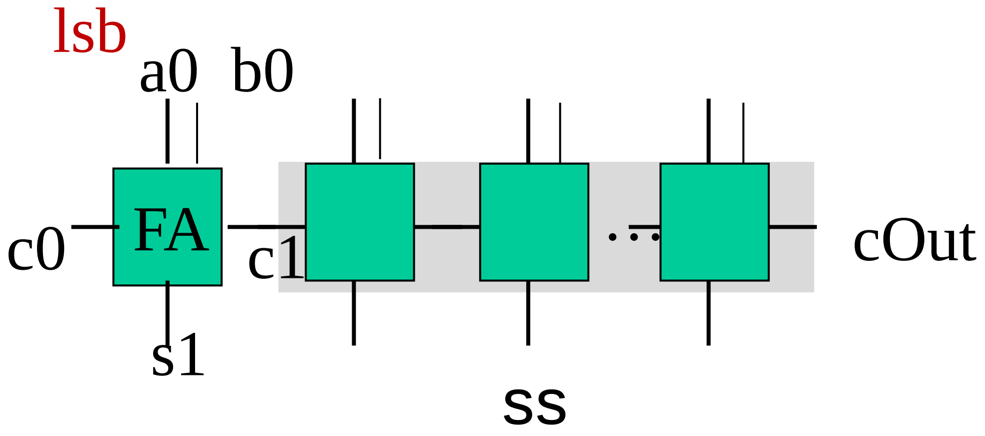
Length can be arbitrary and is not indicated in the type.

Code (base case)



```
rcAdder0 :: (Bit,([Bit],[Bit])) -> ([Bit],Bit)
rcAdder0 (c0, ([], []))          = ([], c0)
```

Code (recursive step)



```
rcAdder0 :: (Bit,([Bit],[Bit])) -> ([Bit],Bit)
rcAdder0 (c0, ([], []))            = ([], c0)
rcAdder0 (c0, (a0:as, b0:bs)) = (s1:ss, cOut)
  where
    (s1, c1)     = fullAdder (c0, (a0, b0))
    (ss, cOut) = rcAdder0 (c1, (as, bs))
```

Second attempt in Lava

```
rcAdder1 :: (Bit,([Bit],[Bit])) -> ([Bit],Bit)
rcAdder1 (c0, (as, bs)) = (sum, cOut)
  where
    (sum, cOut) = row fullAdder (c0, zipp (as,bs))
```

Second attempt in Lava

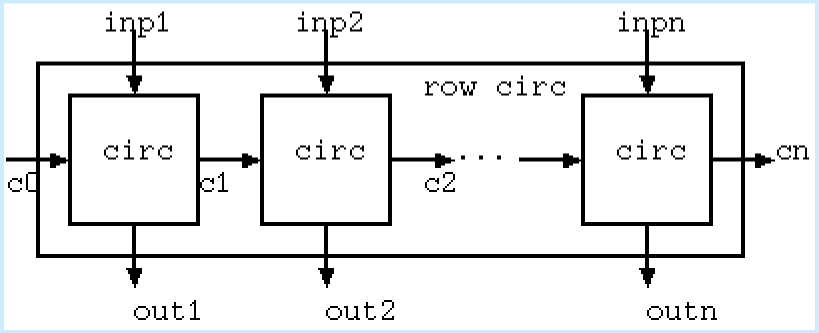
```
rcAdder1 :: (Bit,([Bit],[Bit])) -> ([Bit],Bit)
rcAdder1 (c0, (as, bs)) = (sum, cOut)
  where
    (sum, cOut) = row fullAdder (c0, zipp (as,bs))
```

row is a connection pattern

Second attempt in Lava

```
rcAdder1 :: (Bit,([Bit],[Bit])) -> ([Bit],Bit)
rcAdder1 (c0, (as, bs)) = (sum, cOut)
  where
    (sum, cOut) = row fullAdder (c0, zipp (as,bs))
```

row is a connection pattern



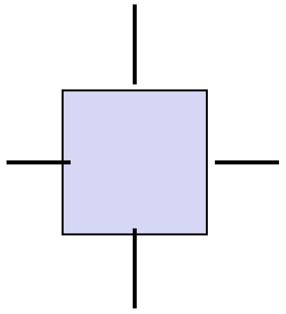
row interface

```
row :: ((c, a) -> (o, c)) -> (c, [a]) -> ([o], c)
```

row

```
row :: ((c, a) -> (o, c)) -> (c, [a]) -> ([o], c)
```

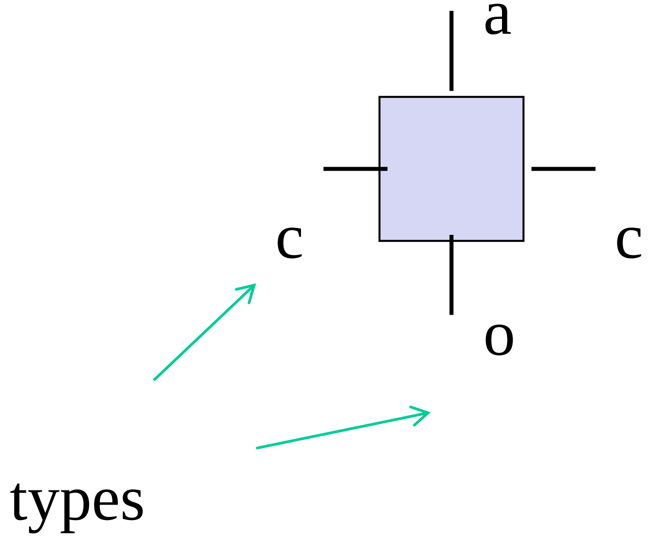
takes a pair-to-pair function



row

```
row :: ((c, a) -> (o, c)) -> (c, [a]) -> ([o], c)
```

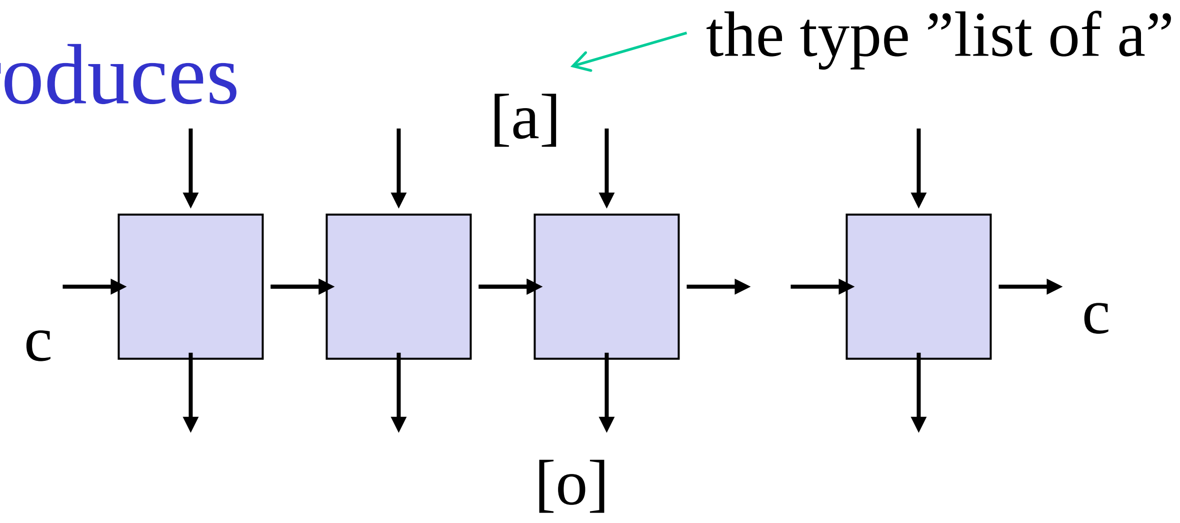
takes a pair-to-pair function



row

```
row :: ((c, a) -> (o, c)) -> (c, [a]) -> ([o], c)
```

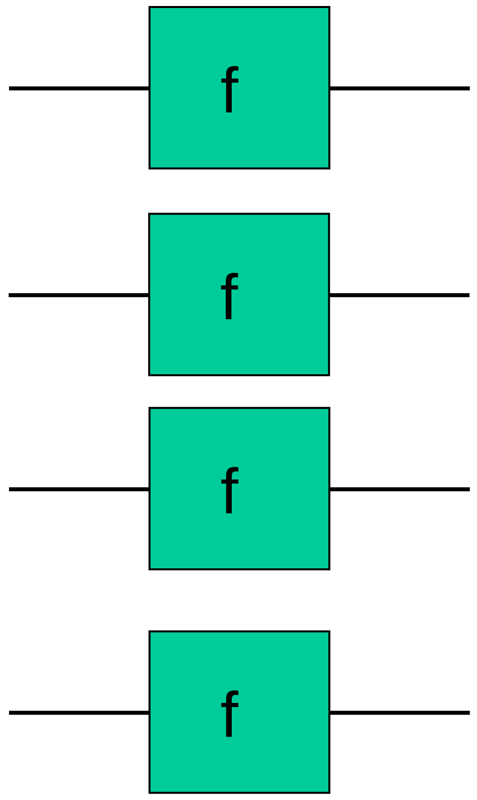
and produces



Definition of row (from Lava.Patterns.hs)

```
row circ (carryIn, [])      = ([], carryIn)
row circ (carryIn, a:as) = (b:bs, carryOut)
  where
    (b, carry)          = circ (carryIn, a)
    (bs, carryOut) = row circ (carry, as)
```

Another connection pattern: map



map f

```
map :: (a -> b) -> [a] -> [b]
```

Example (map and integers)

```
inc :: Signal Int -> Signal Int
inc a = a + 1

tstmap = simulate (map inc) [1..8]
```

```
> tstm
[2,3,4,5,6,7,8,9]
```

Back to Second attempt in Lava

```
rcAdder1 :: (Bit,([Bit],[Bit])) -> ([Bit],Bit)
rcAdder1 (c0, (as, bs)) = (sum, cOut)
  where
    (sum, cOut) = row fullAdder (c0, zipp (as,bs))
```

zipp turns a pair of lists into a list of pairs, to match interface of row

zipp   (from Lava.Patterns.hs)

```
zipp ([],     [])    = []
zipp (a:as, b:bs) = (a,b) : zipp (as, bs)
```

zipp   (from Lava.Patterns.hs)

```
zipp ([],     [])    = []
zipp (a:as, b:bs) = (a,b) : zipp (as, bs)

ziptest :: [Signal Int] -> [(Signal Int,Signal Int)]
ziptest as = zipp (halveList as)
```

zipp

```
zipp ([],     [])    = []
zipp (a:as, b:bs) = (a,b) : zipp (as, bs)
```

use function halveList to get some inputs to zipp

```
> :t halveList
halveList :: [a] -> ([a], [a])

hltst = simulate halveList [1..9 :: Signal Int]

> hltst
([1,2,3,4],[5,6,7,8,9])
```

zipp   (from Lava.Patterns.hs)

```
zipp ([],    [])    = []
zipp (a:as, b:bs) = (a,b) : zipp (as, bs)

ziptest :: [Signal Int] -> [(Signal Int,Signal Int)]
ziptest as = zipp (halveList as)

> simulate ziptest [1..8]
[(1,5),(2,6),(3,7),(4,8)]
> simulate ziptest [1..9]
*** Exception: Lava\Patterns.hs:(24,0)-(25,40): Non-
   exhaustive patterns in function zipp
```

zipp   (from Lava.Patterns.hs)

```
zipp ([],     [])    = []
zipp (a:as, b:bs) = (a,b) : zipp (as, bs)
```

How can we make it cope with unequal length lists?

Exercise

How could we improve our ripple carry adder solution?

```
rcAdder1 :: (Bit,([Bit],[Bit])) -> ([Bit],Bit)
rcAdder1 (c0, (as, bs)) = (sum, cOut)
  where
    (sum, cOut) = row fullAdder (c0, zipp (as,bs))
```

How could we improve the solution?

A : by making the full adder component a parameter

After all, we have more than one such….

Third attempt in Lava

```
rcAdder2 :: ((Bit,(Bit,Bit)) -> (Bit,Bit)) ->(Bit,([Bit],[Bit])) -> ([Bit],Bit)
rcAdder2 fadd (c0, (as, bs)) = (sum, cOut)
  where
    (sum, cOut) = row fadd (c0, zipp (as,bs))
```

Note

Could be viewed as Lustre (or similar) embedded in Haskell

Generic circuits and connection patterns easy to describe   (the power of Haskell)

Verify FIXED SIZE circuits   (squeezing the problem down into an easy enough one, see next lecture)

Next

Verifying generic circuits

Generating VHDL

Sequential circuits

Analysing circuits

More connection patterns and examples

Making circuits cleverer -> circuits that adapt to their contexts