Lava II

Mary Sheeran, Thomas Hallgren

Chalmers University of Technology

Generating VHDL

- In the simplest case
  writeVhdl "fullAdder" fullAdder

- Assigning names to the inputs
  writeVhdlInput "fullAdder" fullAdder (var "carryIn",(var "a",var "b"))

- Assigning names also to the outputs
  writeVhdlInputOutput "fullAdder" fullAdder
  (var "carryIn",(var "a",var "b")) (var "sum",var "carryOut")

- Generic circuits are not supported, so you need to pick a size
  writeVhdlInputOutput "rippleCarryAdder" rippleCarryAdder
  (var "carryIn",(varList 8 "a",varList 8 "b"))
  (varList 8 "sum",var "carryOut")

Generating VHDL (better)

Above method generates silly VHDL for combinational circuits

importing file VhdlNew11.hs   (on Schedule page)

allows gen. of clocked or unclocked VHDL netlists

Append   Clk or NoClk to end of previous function names

see Lava2.hs

```vhdl
library ieee;

use ieee.std_logic_1164.all;

entity
  rippleCarryAdder
is
port
  (

    carryIn : in std_logic
  ; a_0 : in std_logic
  ; a_1 : in std_logic
  ; a_2 : in std_logic
  ; a_3 : in std_logic
  ; b_0 : in std_logic
  ; b_1 : in std_logic
  ; b_2 : in std_logic
  ; b_3 : in std_logic


  ; sum_0 : out std_logic
  ; sum_1 : out std_logic
  ; sum_2 : out std_logic
  ; sum_3 : out std_logic
  ; carryOut : out std_logic
  );
end rippleCarryAdder;


architecture
  structural
of
  rippleCarryAdder
is
  signal w1 : std_logic;
  signal w2 : std_logic;
  signal w3 : std_logic;
  signal w4 : std_logic;
  signal w5 : std_logic;


…     • • •

  signal w29 : std_logic;
begin
  c_w2     : entity work.wire  port map (carryIn, w2);
  c_w4     : entity work.wire  port map (a_0, w4);
  c_w5     : entity work.wire  port map (b_0, w5);
  c_w29    : entity work.andG  port map (w25, w26, w29);
  c_w27    : entity work.xorG  port map (w28, w29, w27);


         • • •
  c_sum_0  : entity work.wire  port map (w1, sum_0);
  c_sum_1  : entity work.wire  port map (w6, sum_1);
  c_sum_2  : entity work.wire  port map (w13, sum_2);
  c_sum_3  : entity work.wire  port map (w20, sum_3);
  c_carryOut : entity work.wire  port map (w27, carryOut);
end structural;
```

Generic circuits again

Lava.Arithmetic.hs contains

binAdder :: ([Signal Bool], [Signal Bool]) -> [Signal Bool]

Generic circuits again

Lava.Arithmetic.hs contains

binAdder :: ([Signal Bool], [Signal Bool]) -> [Signal Bool]

> simulate binAdder ([low,high,low], [high,low,high])
[high,high,high,low]

Generic circuits again

Lava.Arithmetic.hs contains

binAdder :: ([Signal Bool], [Signal Bool]) -> [Signal Bool]

Let's check if it is commutative!

First attempt

```
prop_AdderCommutative (as,bs) = ok
  where
    out1 = binAdder (as,bs)
    out2 = binAdder (bs,as)
    ok   = out1 <==> out2
```

First attempt

prop_AdderCommutative (as,bs) = ok
  where
    out1 = binAdder (as,bs)
    out2 = binAdder (bs,as)
    ok   = out1 <==> out2

smv prop_AdderCommutative   does not work!

Need to fix size

```
prop_AdderCommutative_ForSize n =
    forAll (list n) $ \ as ->
      forAll (list n) $ \ bs ->
        prop_AdderCommutative (as,bs)
```

Need to fix size

```
prop_AdderCommutative_ForSize n =
    forAll (list n) $ \ as ->
      forAll (list n) $ \ bs ->
        prop_AdderCommutative (as,bs)
```

smv (prop_AdderCommutative_ForSize 16) works!

See Chapter 4 in the Lava tutorial.

Same effect but easier

```
prop_AdderComm1 n
    = prop_AdderCommutative (varList n "a", varList n "b")



fv_binAdd_Comm1 = smv (prop_AdderComm1 16)
```

works

Serial composition

useful connection pattern
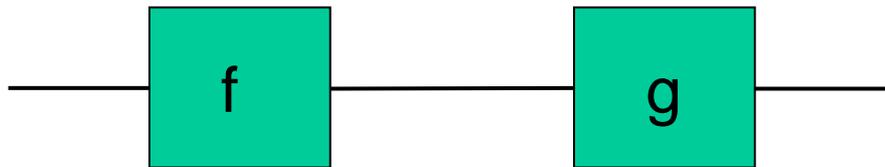
f ->- g

Serial composition   type

useful connection pattern

(->-) :: (a -> b) -> (b -> c) -> a -> c
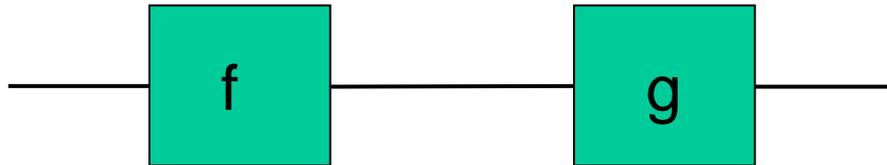
Serial composition example

```
doubSum :: [Signal Int] -> Signal Int
doubSum = map (*2) ->- sum
```

```
> simulate doubSum [1..8]
72
```

Serial composition example

```
doubSum :: [Signal Int] -> Signal Int
doubSum = map (*2) ->- sum
```

could also have written

```
doubSum1 :: [Signal Int] -> Signal Int
doubSum1 as = sum (map double as)
   where
       double a = a * 2
```

Feedback and sequential circuits

First example

```
bad inp = out
  where
    out = nand2(inp,out)
```

Feedback and sequential circuits

First example

```
bad inp = out
 where
   out = nand2(inp,out)
```

> simulate bad low
high
> simulate bad high
*** Exception: combinational loop

Delay in VHDL

Signal assignments have no delay by default:

out <= a nand b;

Delay can be introduced explicitly:

out <= a nand b after 4ns;

Delay in Lava

The logical gates in the Lava library are "ideal" and have zero delay

Delay has to be modelled explicitly:

delay init s

delays the signal s by one time unit

The output during the first time unit is init

Delay in Lava

The Lava library does not care how long a time unit is.

It could be the gate delay, for analyzing the effect of delay in combinational circuits

But usually it is one clock cycle in a synchronously clocked sequential circuit.

Feedback and sequential circuits

Second example

nand2D = nand2 ->- delay low

good a = out
  where
    out = nand2D(a,out)

Feedback and sequential circuits

nand2D = nand2 ->- delay low

good a = out
  where
    out = nand2D(a,out)

*Main> simulate good high
*** Exception: evaluating a delay component
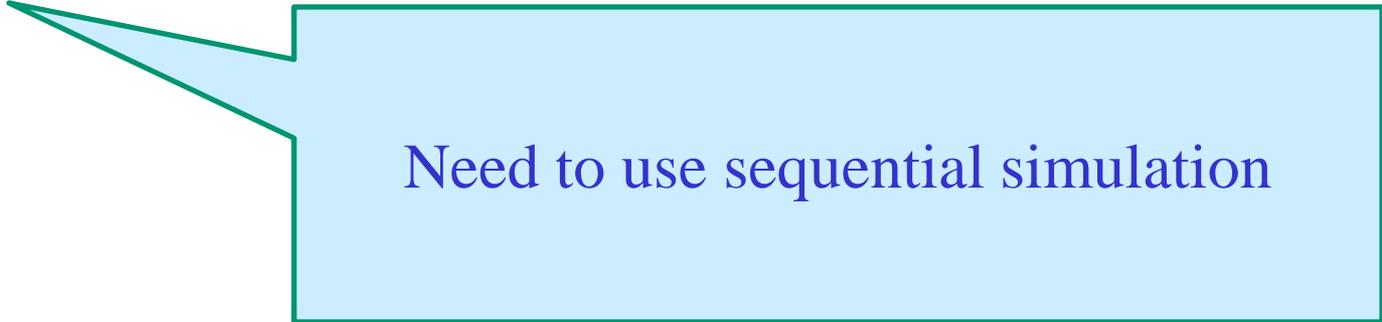
Need to use sequential simulation

Feedback and sequential circuits

nand2D = nand2 ->- delay low

good a = out
  where
    out = nand2D(a,out)

*Main> simulateSeq good [high,high,low,high]
[low,high,low,high]

Retiming

nand2D = nand2 ->- delay low

delNand2 = delay (high,high) ->- nand2

sim0 = simulateSeq nand2D [(low,low),(high,low),(high,high),(low,low)]
sim1 = simulateSeq delNand2 [(low,low),(high,low),(high,high),(low,low)]

> sim0
[low,high,high,low]
> sim1
[low,high,high,low]

Retiming

nand2D = nand2 ->- delay low

delNand2 = delay (high,high) ->- nand2

sim0 = simulateSeq nand2

sim1 = simulateSeq delNan

Note that delay works on many types, not just bits

> sim0
[low,high,high,low]
> sim1
[low,high,high,low]

FV

-- A general function for equivalence testing
propEQ circ1 circ2 inp = ok
  where
    out1 = circ1 inp
    out2 = circ2 inp
    ok = out1 <==> out2

prop0 = propEQ nand2D delNand2

fv_prop0 = smv prop0
(on my laptop ca .1 sec, 60 BDD nodes allocated)

Register
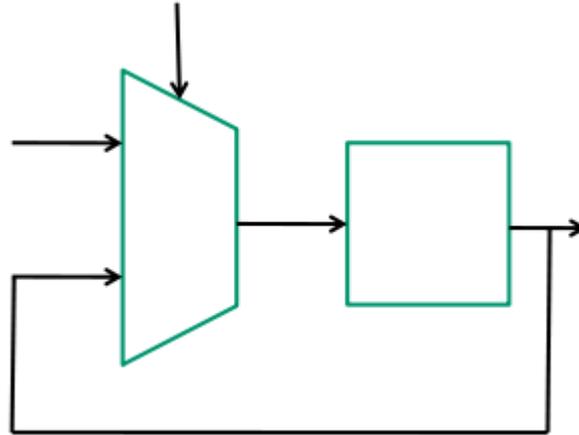

```
reg init (w,din) = dout
  where
    dout = delay init m
    m    = mux (w,(dout,din))
```

Register
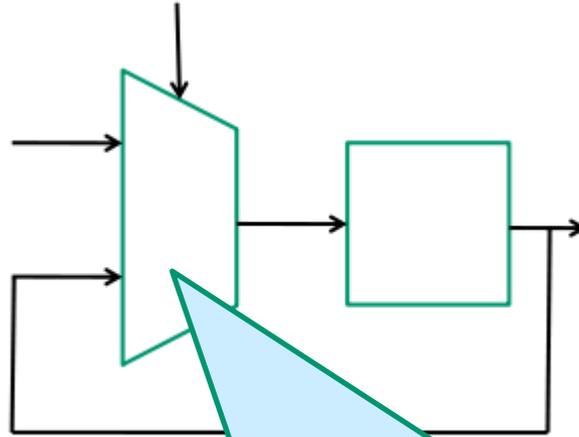
multiplexer (also polymorphic)

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

reg init (w,din) =
  where
    dout = delay init m
    m   = mux (w,(dout,din))

using Haskell to generate inputs

```
-- infinite lists
lh :: [Bit]
lh =  low : high : lh

ins :: Int -> [[Signal Int]]
ins n = map (replicate n) [1..]

regtst n = simulateSeq (reg (zeroList n)) (take 10 (zip lh (ins n)))
```

```
*Main> regtst 5
[[0,0,0,0,0],[0,0,0,0,0],[2,2,2,2,2],[2,2,2,2,2],[4,4,4,4,4],
 [4,4,4,4,4],[6,6,6,6,6],[6,6,6,6,6],[8,8,8,8,8],[8,8,8,8,8]]
```

Questions?

Connection patterns

Higher order functions that capture common ways
 of plugging circuits together

Connection patterns

Higher order functions that capture common ways
 of plugging circuits together

We saw   row

Connection patterns

Higher order functions that capture common ways
 of plugging circuits together

We saw   row

->-
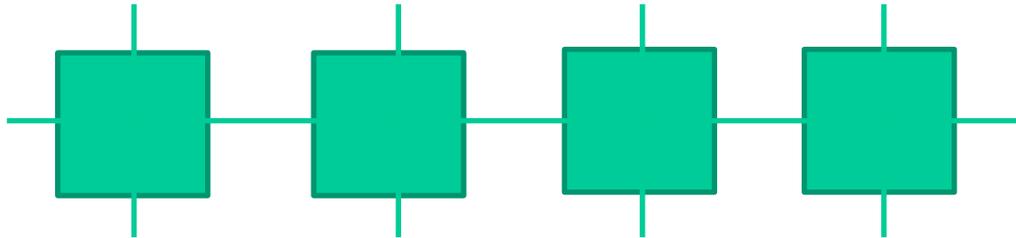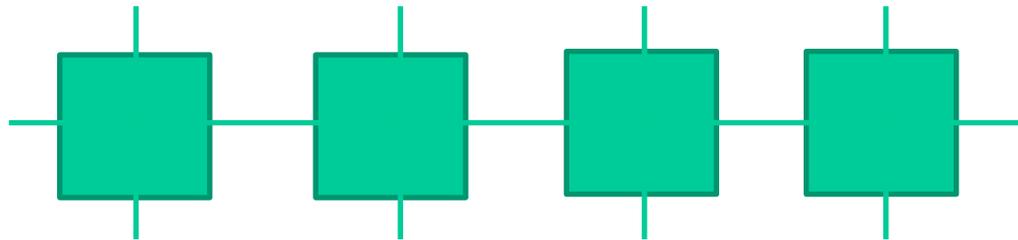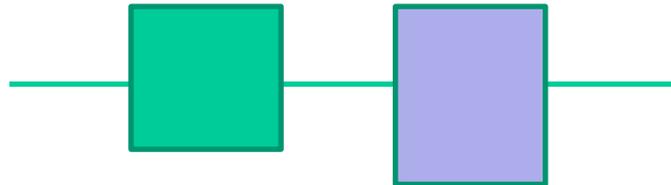
Connection patterns

Higher order functions that capture common ways
 of plugging circuits together

We saw   row
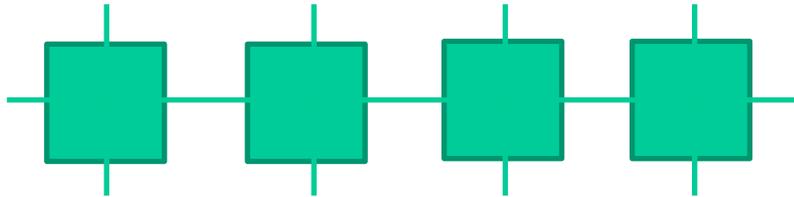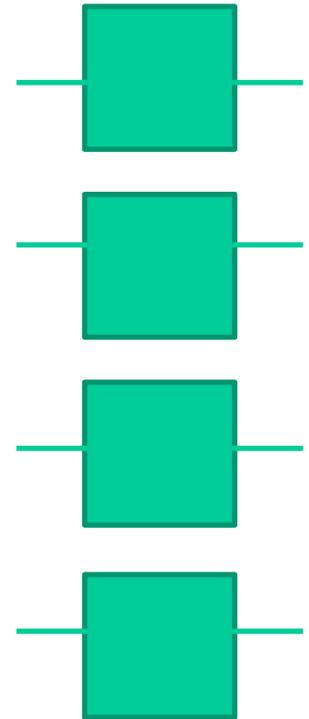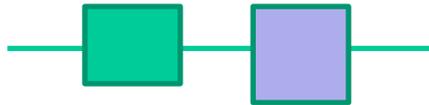
->-

map

map f ->- map g = ??
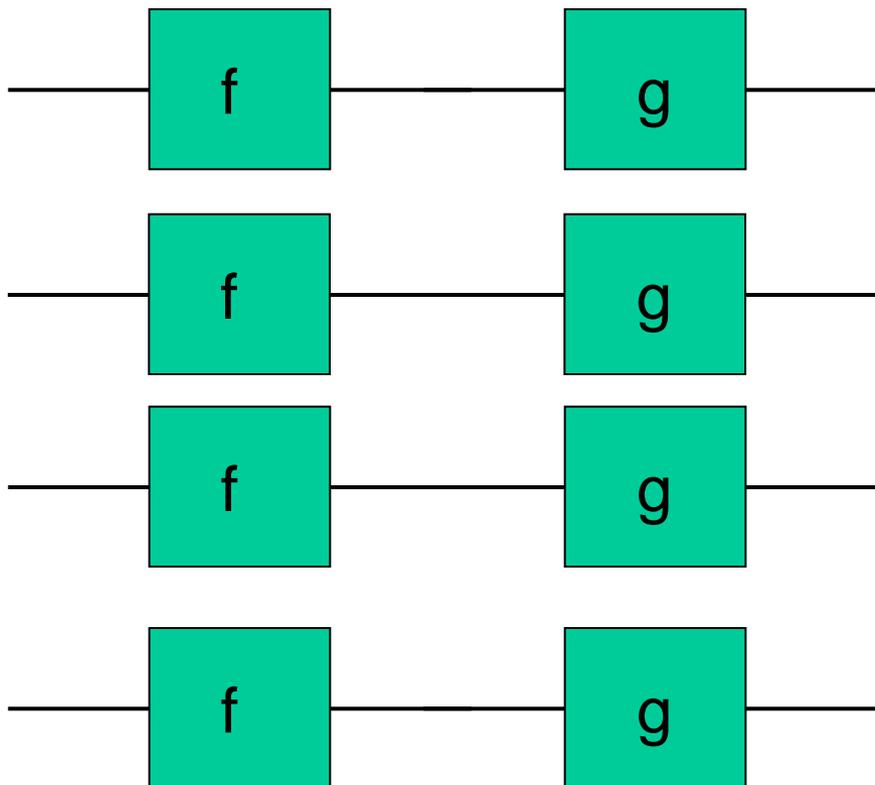
More connection patterns: column and grid
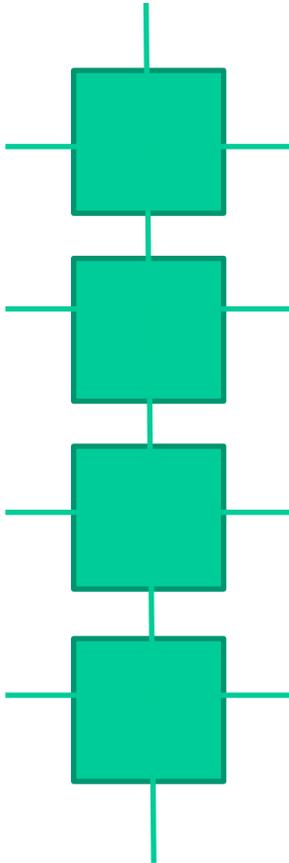
mirror circ (a, b) = (c, d)
  where
    (d, c) = circ (b, a)

column circ = mirror (row (mirror circ))

grid circ = row (column circ)

(in Lava.Patterns)
could just define column recursively (exercise)

More connection patterns: compose

[▮ , ▮ , ▮]
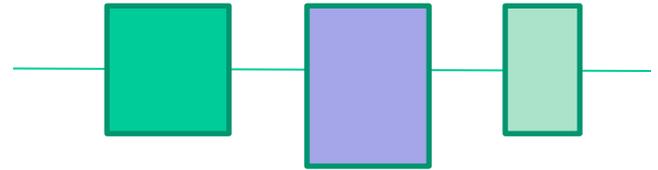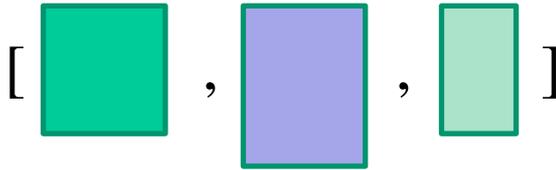
compose :: [a -> a] -> a -> a

(is in Lava.Patterns)

More connection patterns

[⬛ , ⬜ , ⬜]

⬛   [⬜ , ⬜]
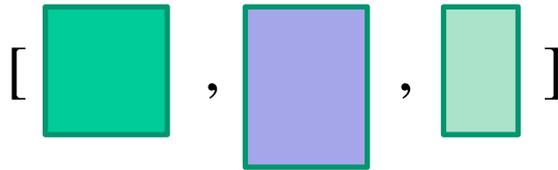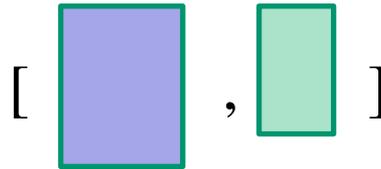
circ   circs

```
compose :: [a -> a] -> a -> a
compose []           =
compose (circ : circs) =
```

More connection patterns
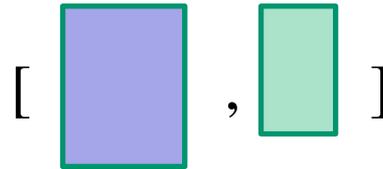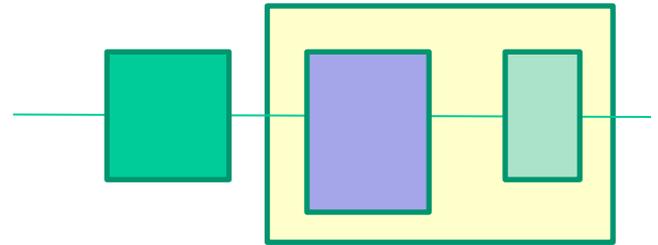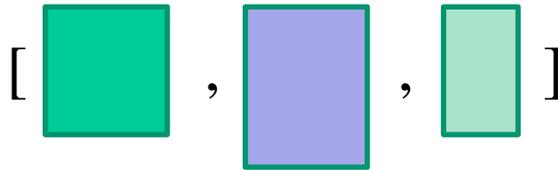
circ   circs

```
compose :: [a -> a] -> a -> a
compose []              =  id
compose (circ : circs) =  circ ->- compose circs
```

compose n copies of function

```
composeN :: Int -> (a -> a) -> a -> a
composeN n circ = compose (replicate n circ)
```

(in Lava.Patterns)

compose n copies of function

```
composeN :: Int -> (a -> a) -> a -> a
composeN n circ = compose (replicate n circ)
```

```
doubN :: Int -> Signal Int -> Signal Int
doubN n = composeN n (*2)
```

```
*Main> simulate (doubN 4) 1
16
```

compose n copies of function

```
composeN :: Int -> (a -> a) -> a -> a
composeN n circ = compose (replicate n circ)
```

Note that this is a Haskell Int
not a circuit level Int (Signal Int)

compose n copies of function

could also have defined this function recursively:

```
composeNR 0 circ = id
composeNR n circ = circ -> composeNR (n-1) circ
```

compose n copies of function

could also have defined this function recursively:

```
composeNR 0 circ = id
composeNR n circ = circ -> composeNR (n-1) circ
```

This is a second very standard way to write recursive functions in Haskell

Par

g

f
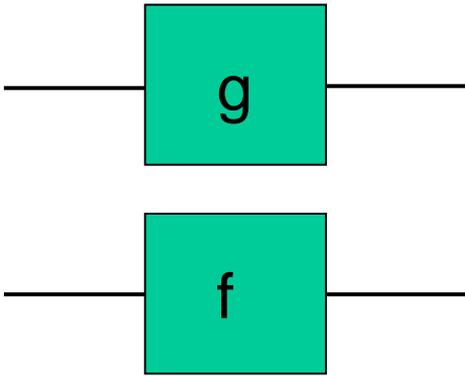
f -|- g

(in Lava.Patterns)

Par
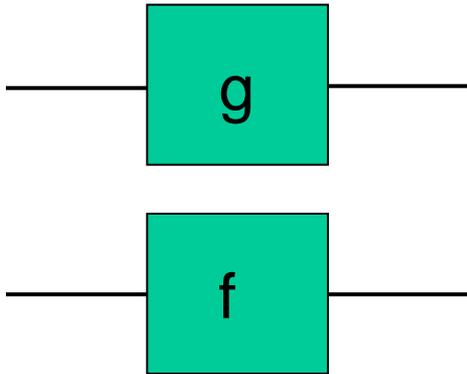
infixr 4 -|-

par circ1 circ2 (a, b) = (circ1 a, circ2 b)
circ1 -|- circ2   = par circ1 circ2

f -|- g

Par
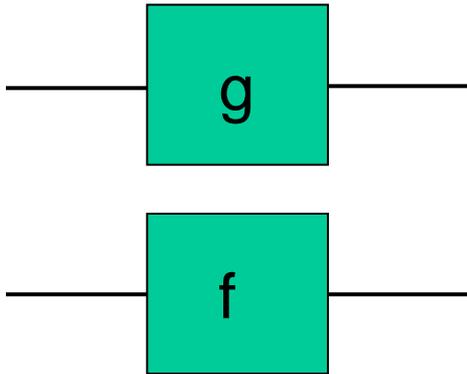
infixr 4 -|-

par circ1 circ2 (a, b) = (circ1 a, circ2 b)
circ1 -|- circ2   = par circ1 circ2

f -|- g

Q: What is the type of par?

red f
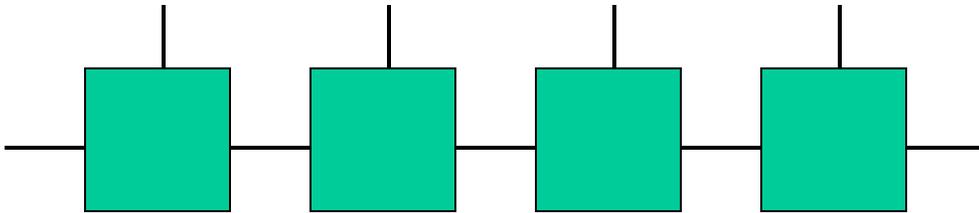
red f
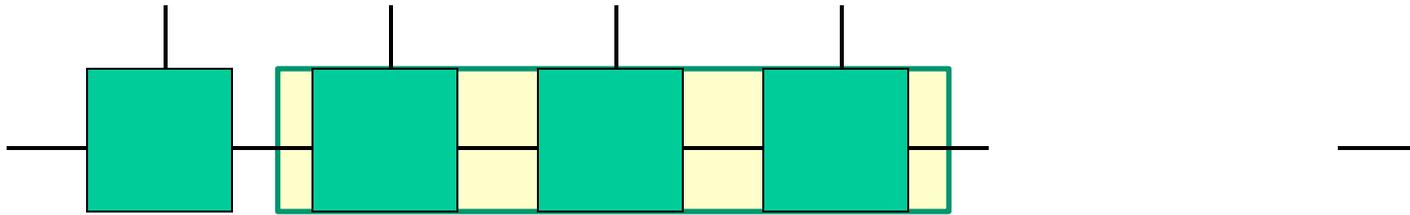
red f
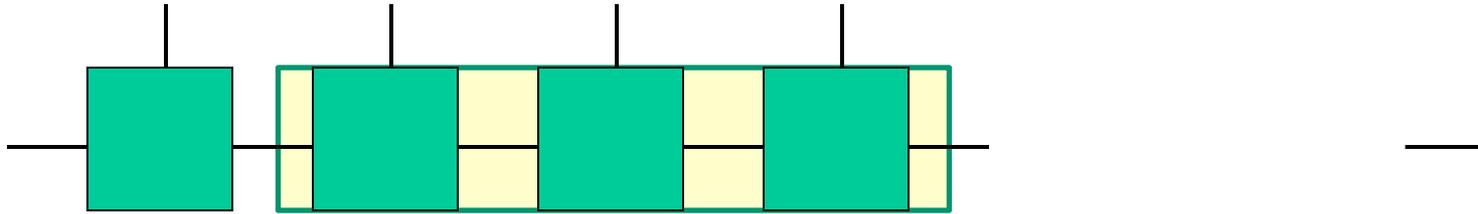
red :: ((a,b) -> a) -> (a, [b]) -> a

red f


```
red :: ((a,b) -> a) -> (a, [b]) -> a
red f (a,[])      =
red f (a, (b:bs)) =
```

red f
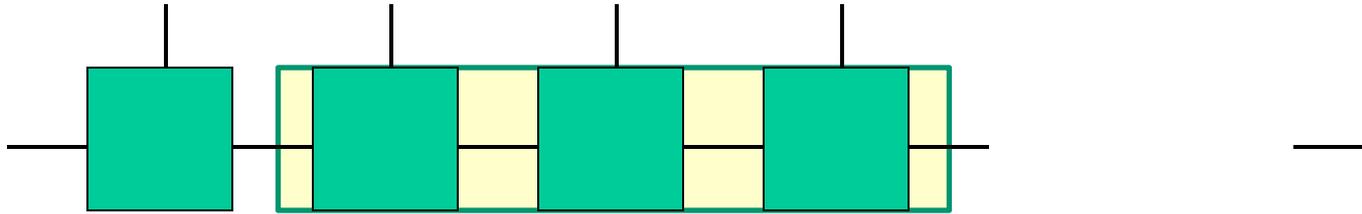

```
red :: ((a,b) -> a) -> (a, [b]) -> a
red f (a,[])        = a
red f (a, (b:bs)) = red f (f(a,b), bs)
```

red f
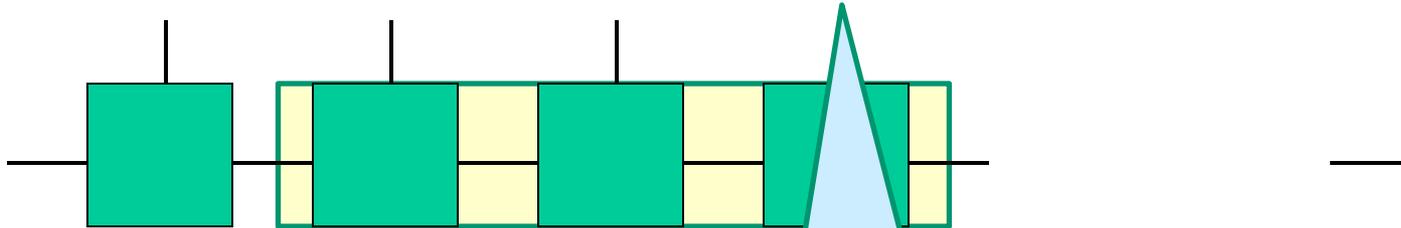

```
red :: ((a,b) -> a) -> (a        -> a
red f (a,[])        = a
red f (a, (b:bs)) = red f        s)
```

redtst = simula

>redtst
18

This is very like foldl in Haskell
Here, I like to gather circuit inputs into
one structure (pair). Makes it easier to
see where the circuits are and to
compose again with more connection
patterns

lin (more like foldl1)

```
lin f (a:as)  = red f (a,as)
lin _ []      =  error "lin: empty list"
```

> simulate (lin plus) [1..5]
15
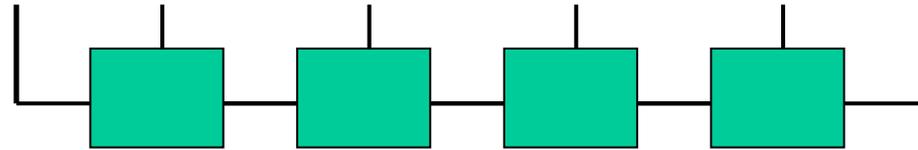
> simulate (lin plus) []
*** Exception: lin: empty list

tri f
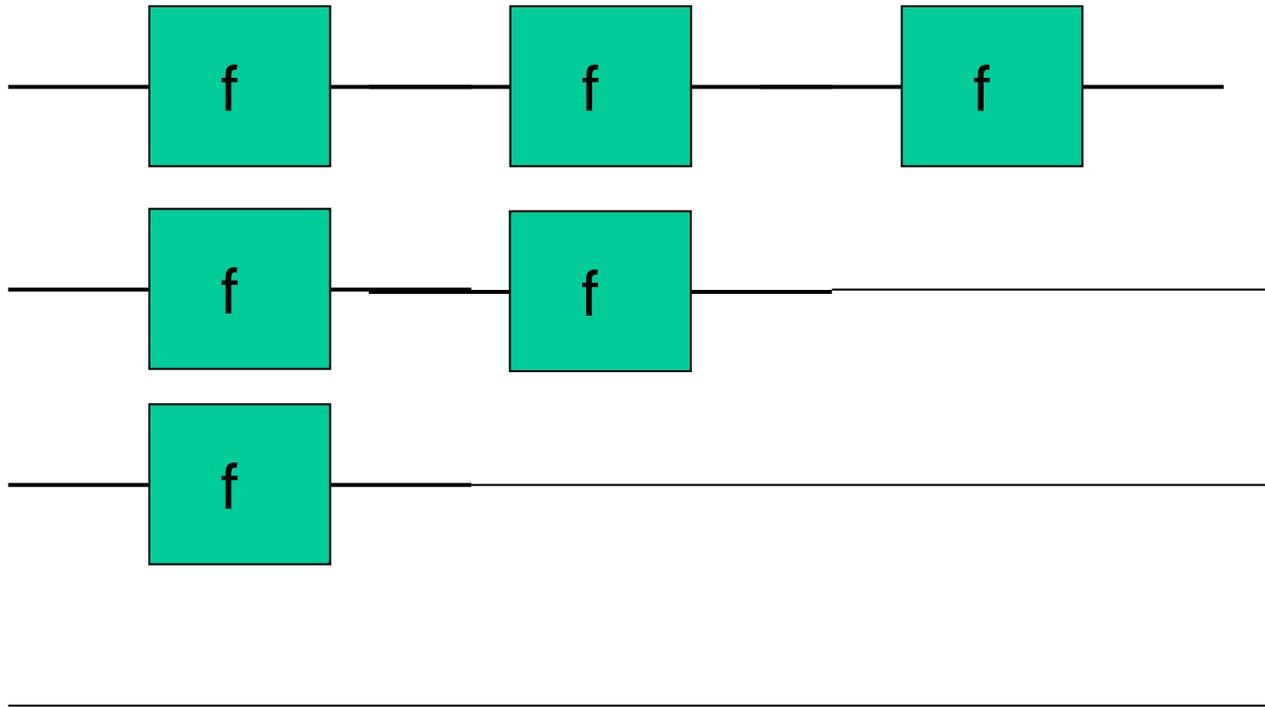
Triangles (tri in Lava.Patterns)

```
tri circ []          =
tri circ (inp:inps)  =
```

Triangles (tri in Lava.Patterns)

```
tri circ []          = []
tri circ (inp:inps) = inp : (map circ ->- tri circ) inps
```

Triangles (tri in Lava.Patterns)

```
tri circ []             = []
tri circ (inp:inps) = inp : (map circ ->- tri circ) inps
```

```
downtri f  = reverse ->- tri f ->- reverse
```

Converting msb first binary to integer
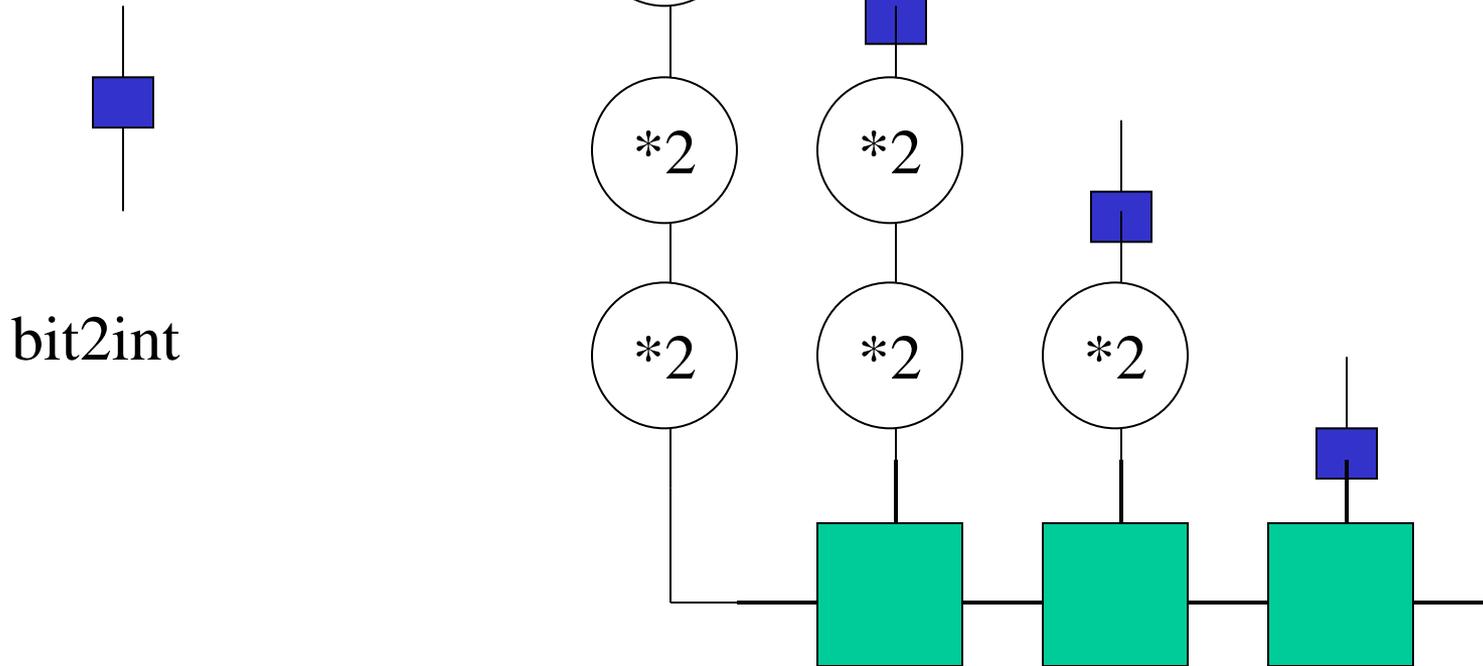
bit2int

binary to integer

msbbin2int = map bit2int ->- downtri (*2) ->- lin plus

(bin2int is in Arithmetic module, lsb first)

Another way
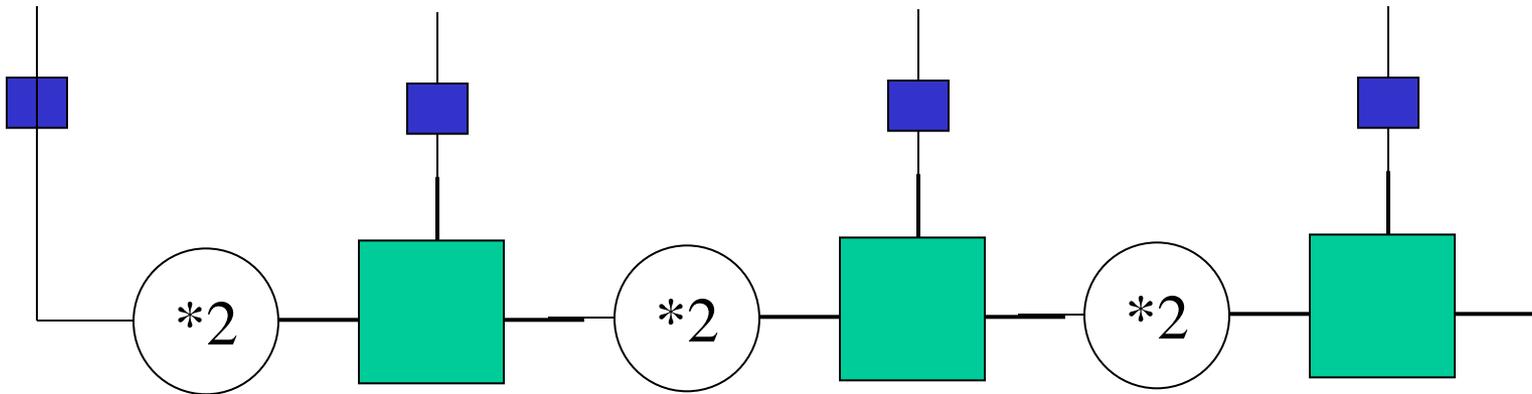
binary to integer

```
msbbin2int'  =  map bit2int ->- lin cell
  where
    cell (a,b)  =  2*a + b
```

binary to integer

```
msbbin2int'  =  map bit2int ->- lin cell
  where
    cell (a,b)  =  2*a + b
```

Note: we have no solver hooked up to Lava that can do arithmetic ☹
So we can't formally verify equivalence of the two different msb-bin to int functions

# On the bright side: a general rule!

If
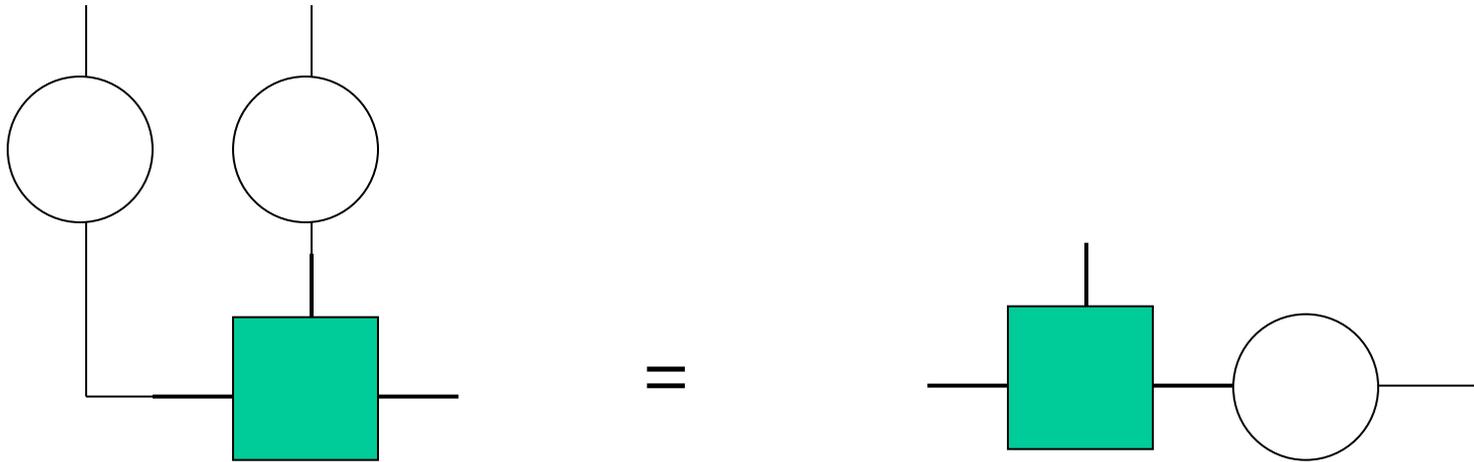
Then, this is the same as
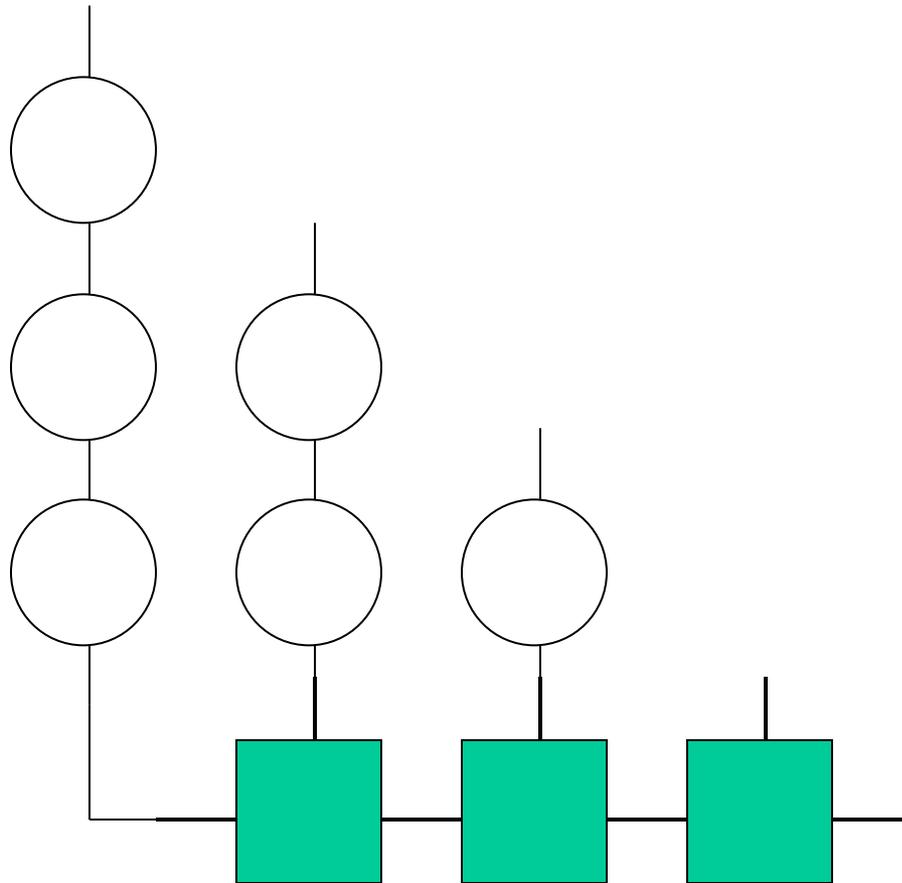
# this, no matter what the components are (and for any size)
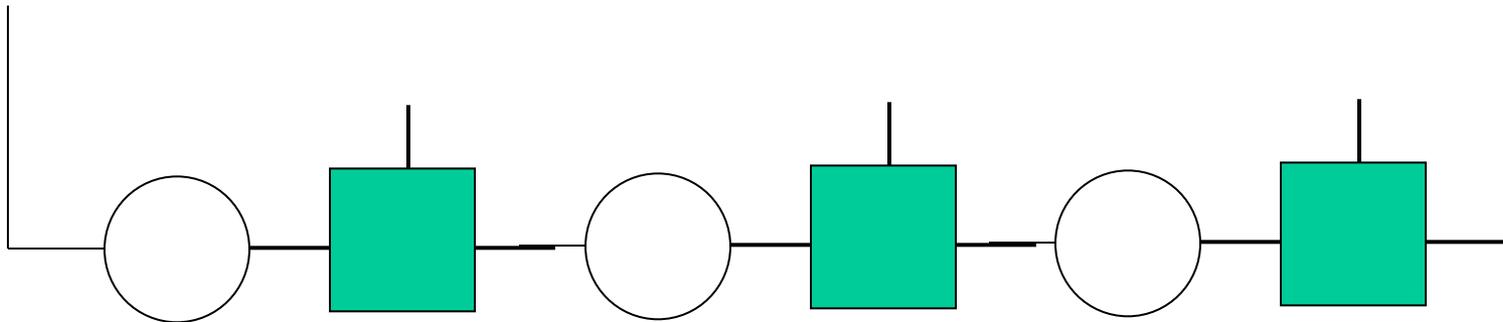
Why?

tri

map
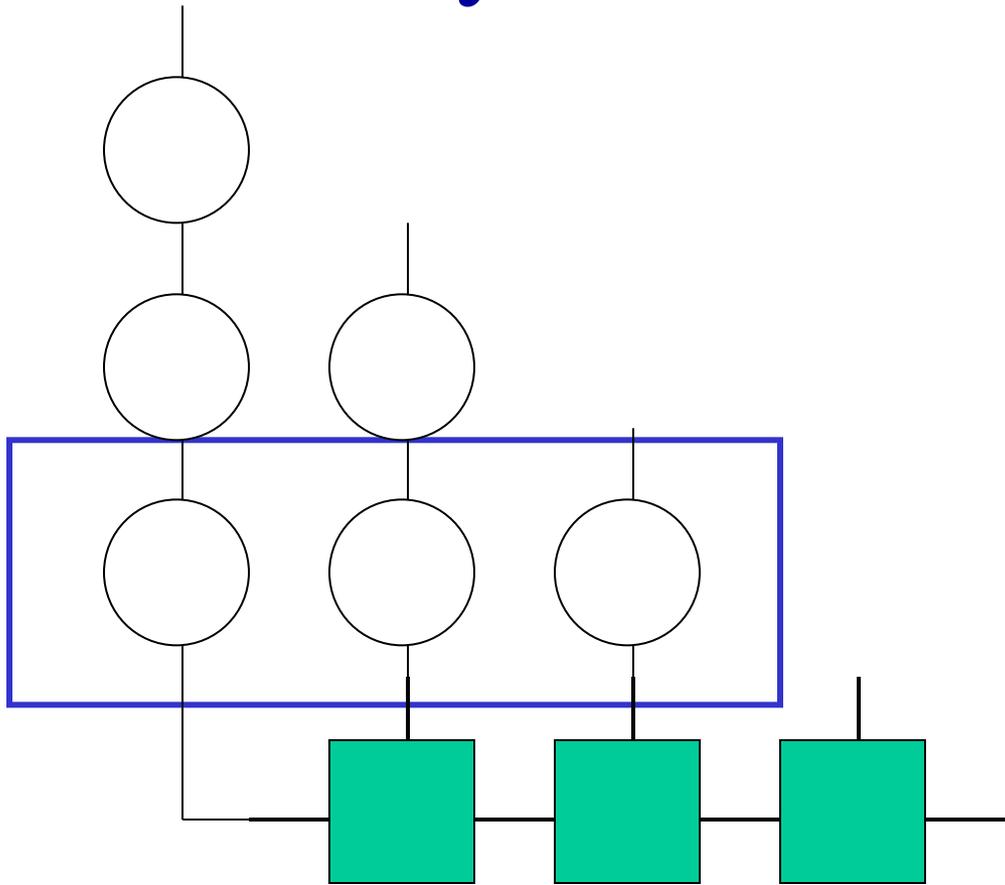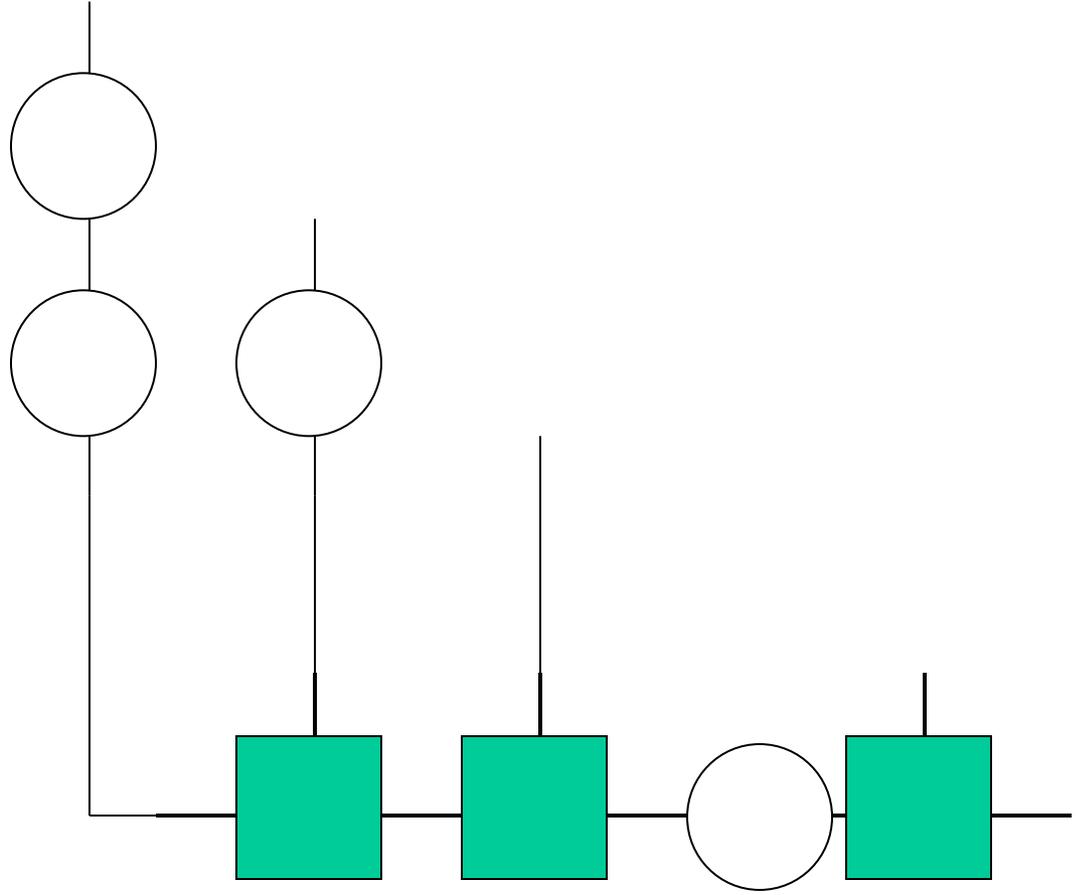
What's left?
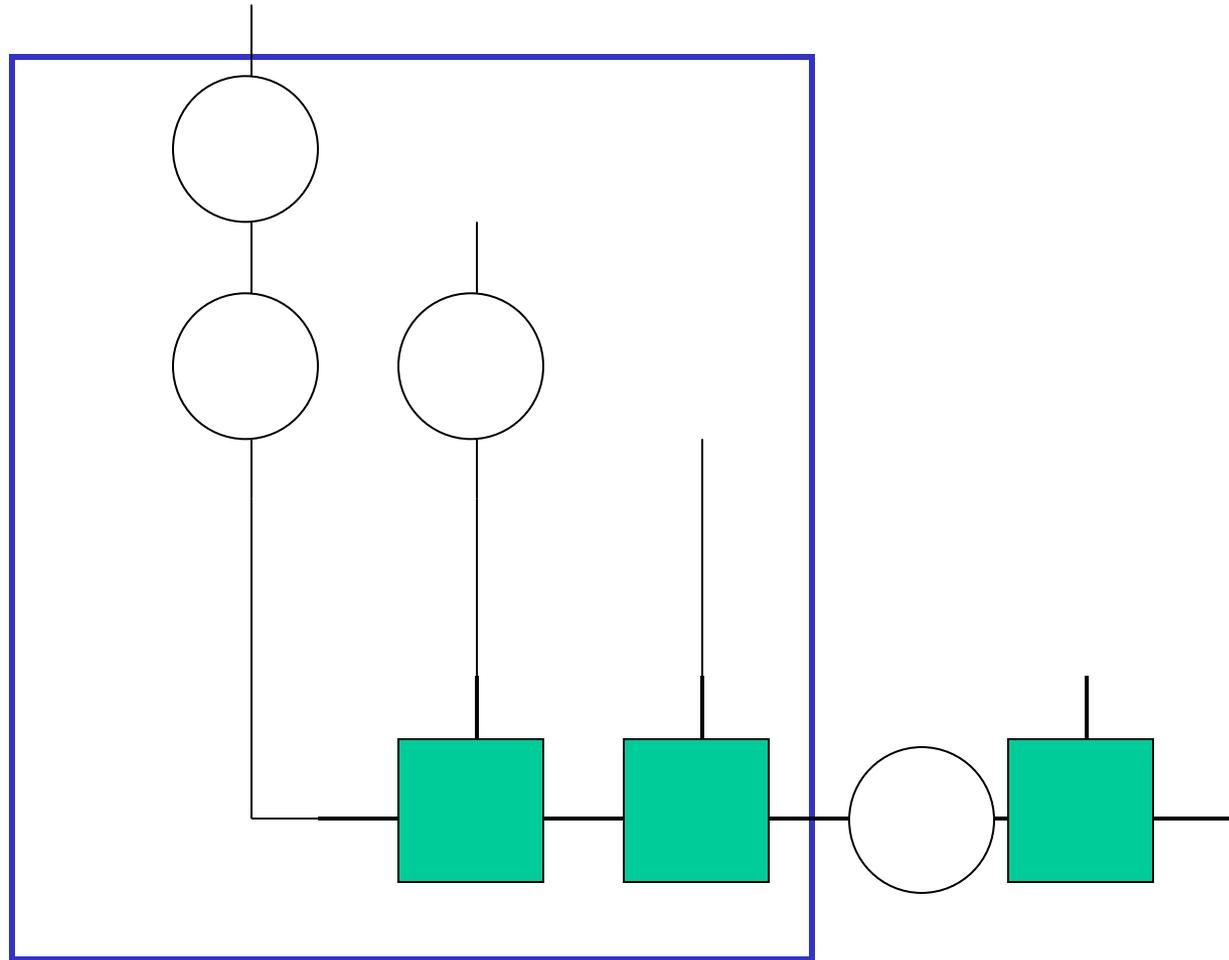
So, by induction
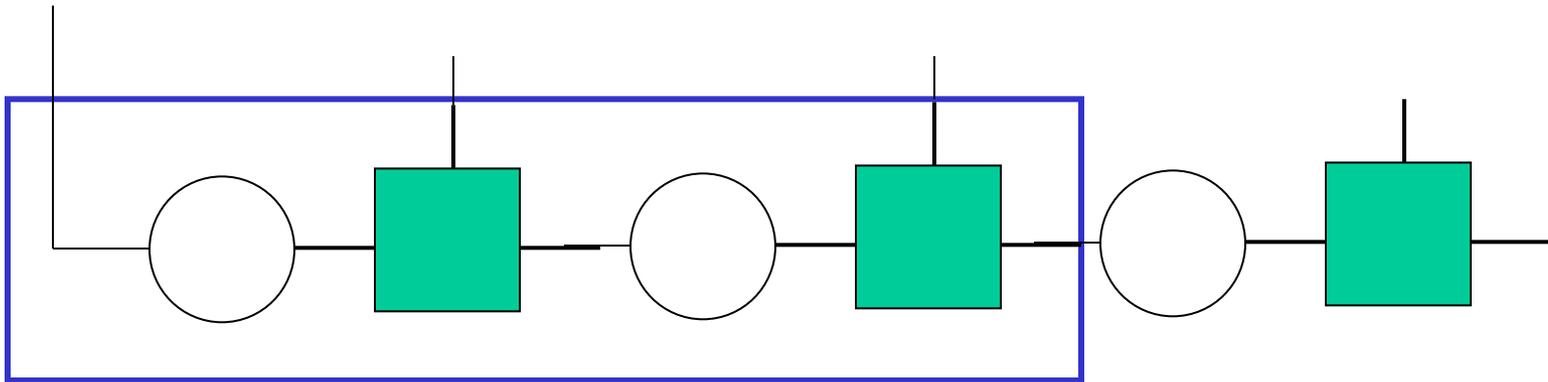
Does this look familiar?
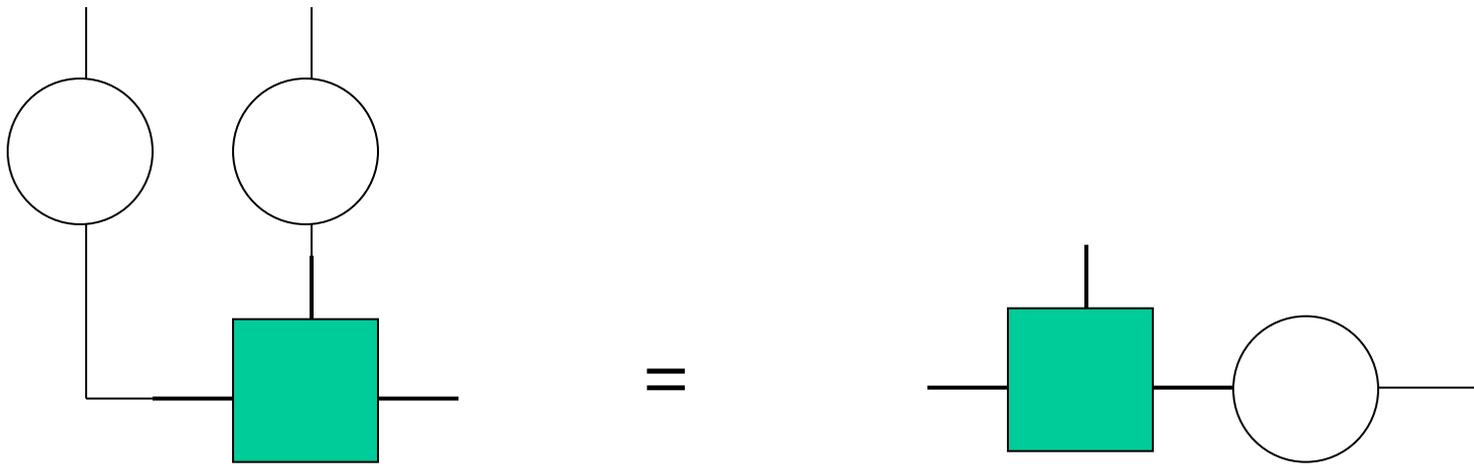
Pipelining!
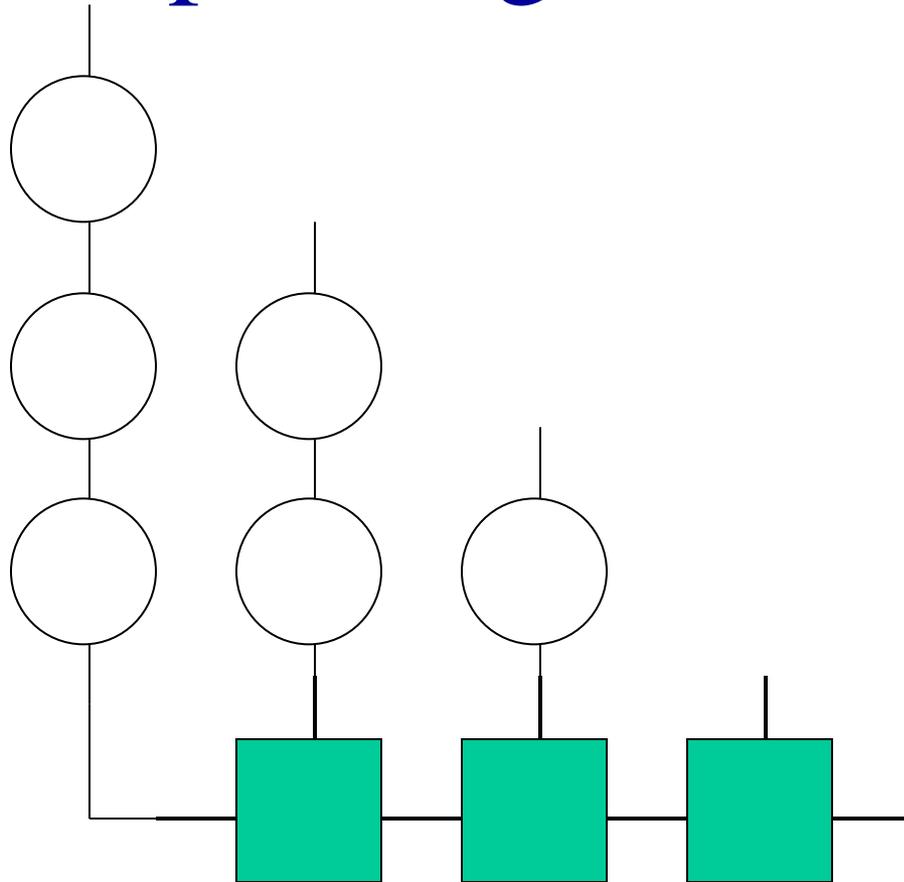
Equality holds for circle=delay
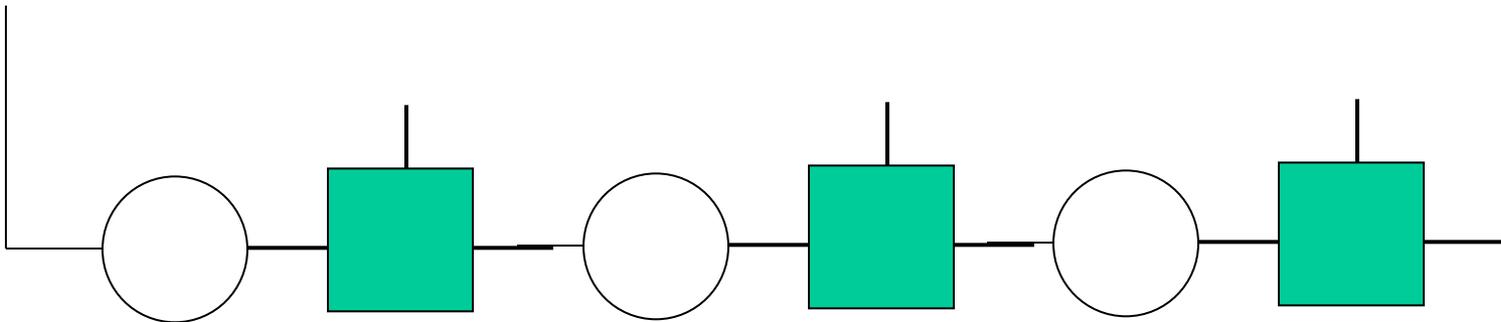
Checking equiv. of an instance

```
propEQ circ1 circ2 a = ok
  where
    out1 = circ1 a
    out2 = circ2 a
    ok = out1 <==> out2

propEQS circ1 circ2  n = propEQS circ1 circ2 (varList n "a")

withtri f = downtri (delay low) ->- lin f

piped f = lin cell
  where
    cell (a,b) = f (delay low a, b)

pipetst = smv (propEQS (withtri and2) (piped and2)  4)
```

Checking equiv. of an instance

```
propEQ circ1 circ2 a = ok
  where
    out1 = circ1 a
    out2
    ok =
```

```
propEQS                                                 rList n "a")
```

```
withtri f
```

On Mary's i5 laptop
Size 4 and 8 very quick
size 12 14 secs
size 16 didn't finish
Lava is good for stress-testing tools

```
piped f = lin cell
  where
    cell (a,b) = f (delay low a, b)
```

```
pipetst = smv (propEQS (withtri and2) (piped and2)  4)
```

Are connection patterns useful??

or do they just make programming harder?

Xilinx Lava provides part of the answer:

Full Adder in Xilinx Lava

cout

b → fa → sum

a

cin
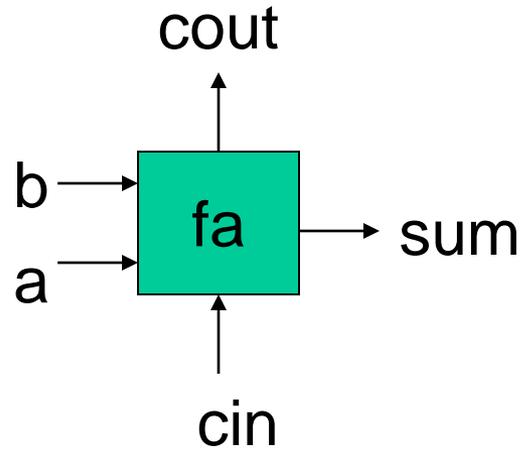
fa (cin, (a,b)) = (sum, cout)
     where
        part_sum = xor (a, b)
        sum = xorcy (part_sum, cin)
        cout = muxcy (part_sum, (a, cin))

Generic Adder
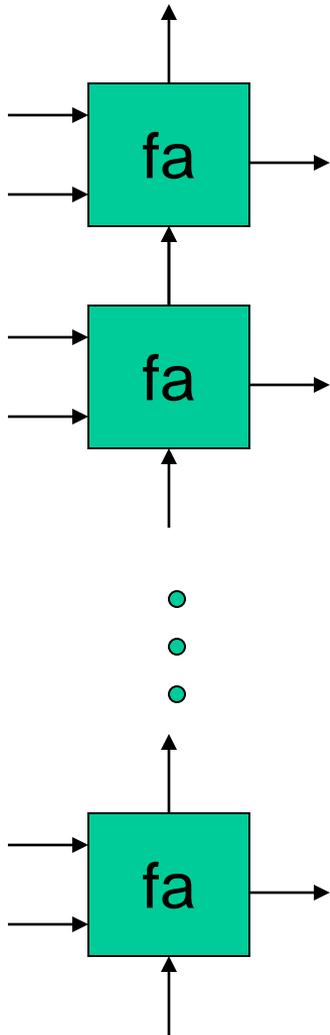
adder = **col** fa

Top Level

adder16Circuit
  = **do** a <- inputVec "a" (bit_vector 15 downto 0)
       b <- inputVec "b" (bit_vector 15 downto 0)
       (s, carry) <- adder1 (a, b)
       sum <- outputVec "sum" (s++[carry])
                              (bit_vector 16 downto 0)

? circuit2VHDL "add16" adder16Circuit
? circuit2EDIF "add16" adder16Circuit
? circuit2Verilog "add16" adder16Circuit

114 Lines of VHDL

```vhdl
library ieee ;
use ieee.std_logic_1164.all ;
entity add16 is
  port(a : in std_logic_vector (15 downto 0) ;
       b : in std_logic_vector (15 downto 0) ;
       c : out std_logic_vector (16 downto 0)
      ) ;
end entity add16 ;

library ieee, unisim ;
use ieee.std_logic_1164.all ;
use unisim.vcomponents.all ;
architecture lava of add16 is
  signal lava : std_logic_vector (0 to 80) ;
begin
...
 lut2_48 : lut2 generic map (init => "0110") port map (i0 => lava(5), i1 => lava(21), o => lava(48)) ;
  xorcy_49 : xorcy port map (li => lava(48), ci => lava(47), o => lava(49)) ;
  muxcy_50 : muxcy port map (di => lava(5), ci => lava(47), s => lava(48), o => lava(50)) ;
  lut2_51 : lut2 generic map (init => "0110") port map (i0 => lava(6), i1 => lava(22), o => lava(51)) ;
  xorcy_52 : xorcy port map (li => lava(51), ci => lava(50), o => lava(52)) ;
  muxcy_53 : muxcy port map (di => lava(6), ci => lava(50), s => lava(51), o => lava(53)) ;
  lut2_54 : lut2 generic map (init => "0110") port map (i0 => lava(7), i1 => lava(23), o => lava(54)) ;
...
```

EDIF...

```
(edif add16
 (edifVersion 2 0 0)
 (edifLevel 0)
 (keywordMap (keywordLevel 0))
 (status
  (written (timeStamp 2000 11 19 15 39 43)
   (program "Lava" (Version "2000.14"))
   (dataOrigin "Xilinx-Lava") (author "Xilinx Inc.")
  )
 )
...
 (instance lut2_78
      (viewRef prim
       (cellRef lut2 (libraryRef lava_virtex_lib))
      )
       (property INIT (string "6"))
      (property RLOC (string "R-7C0.S1"))
     )
…
 (net lava_bit38
      (joined
       (portRef o (instanceRef muxcy_38))
       (portRef ci (instanceRef muxcy_41))
       (portRef ci (instanceRef xorcy_40))
      )
     )
```

Xilinx FPGA Implementation

- 16-bit implementation on a XCV300 FPGA

- Vertical layout required to exploit fast carry chain

- No need to specify coordinates in HDL code

16-bit Adder Layout
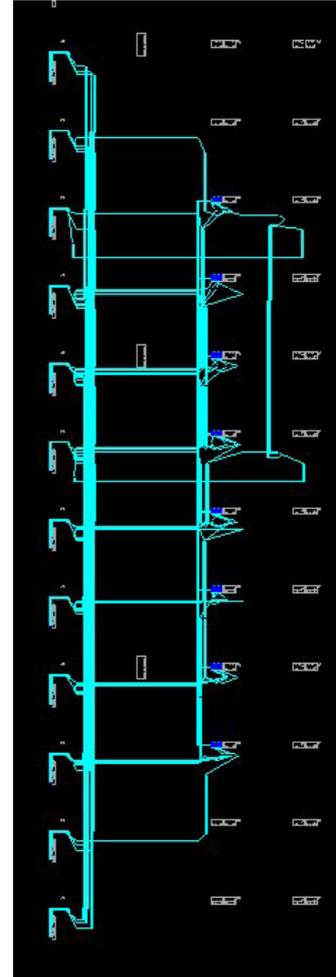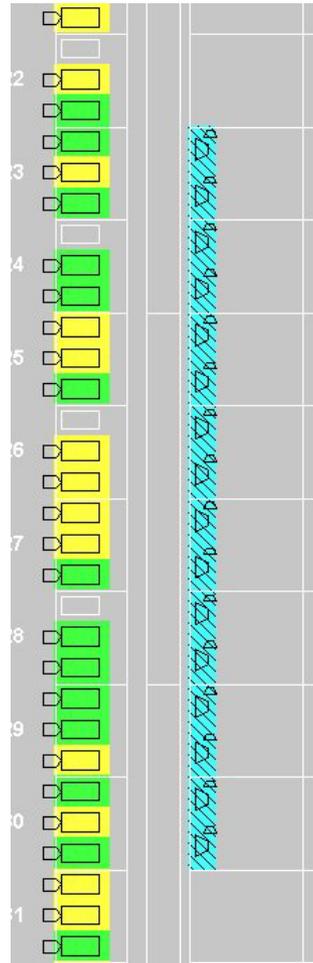
Four adder trees
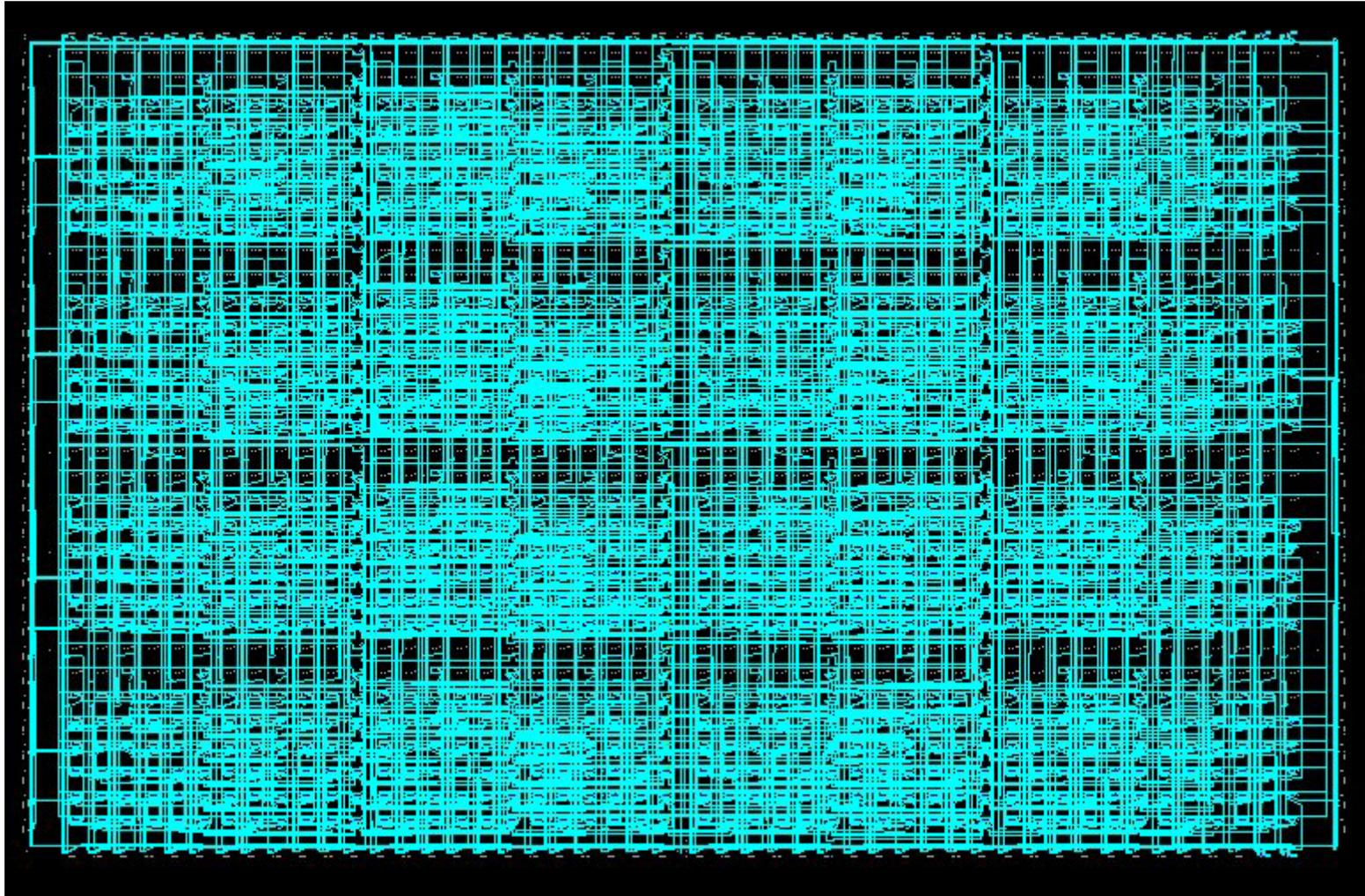
No Layout Information
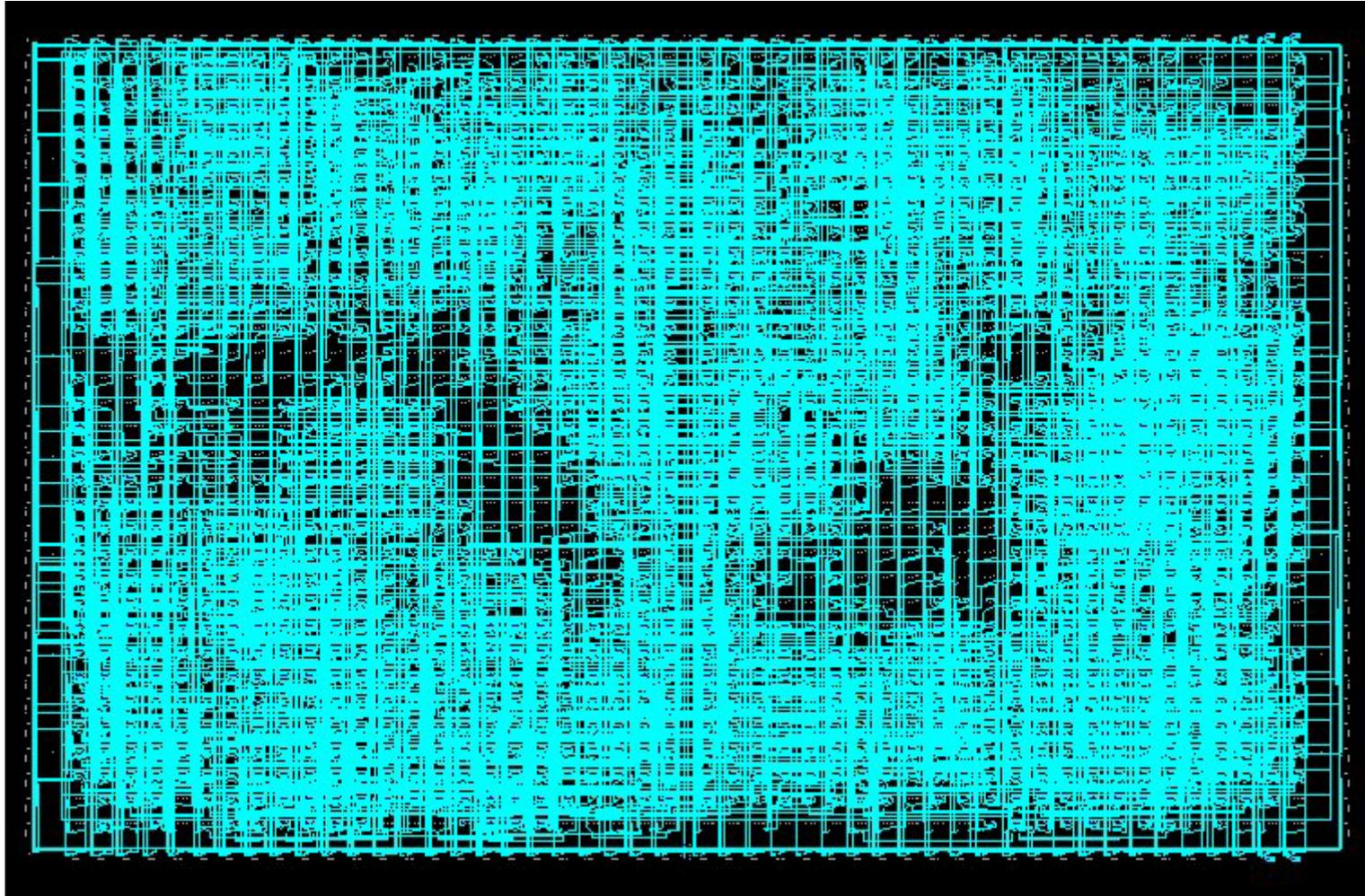
Research questions: FPGA gen.

It is already quite hard to make use of fast carry chains on FPGAs   (see papers on making fast adders, for instance)

How on earth can we make good use of DSPs, processors etc.??

One answer might be to use Functional Programming ☺
(This is what my latest grant proposal was about)

Another motivation for connection patterns and algebra

Work on Hawk for describing and reasoning about processors showed really nice applications of transformations

See John Matthew's slides

Notes

Abstract circuits useful during design exploration (in simulation rather than formal verification)

Verify fixed size instances of generic circuits

Input to solvers (or VHDL netlist) generated by symbolic evaluation

Notes

No mechanical assistance in algebraic reasoning currently in Lava (but earlier research by Cachera using PVS was promising, 30 interactions down to 4 in mult. verif.)

Research question: what could we do with a first order logic theorem prover or SMT solver?

Exercise: **Zero detection**

•Define a generic circuit that
  •inputs a bit vector, and
  •outputs high if all bits are zero.

```
zero_detect :: [Bit] -> Bit
```

•Simple solution first
•Also think about circuit depth and delay